Random Thoughts on Machines and Computability

Thomas Forster

February 21, 2023

Contents

1 Functions in intension and in extension **3**
1.1 ...and Why I was a traumatised Child 3
1.2 Functions in intension and in extension, and Slide Rules 4

2 Revelation and Computability **6**

3 DFAs **11**
3.1 Word ladders 11
3.2 What can DFAs remember? 12
3.3 How to Pump a Wasp 13
3.4 The Easy Way into the Pumping Lemma 14
3.5 How to spot a regular language 15
3.6 A Riff on van der Waerden's Theorem 15
3.6.1 A new Way of proving van der Waerden? 17
3.7 NFAs are nondeterministic not probabilistic! 18
3.8 Polynomial growth? 18
3.9 Enumerating DFA's 18
3.10 Any connection between Quantifier Elimination and Automaticity? 20
3.11 Regular Languages for Numerals 21

4 Context-free Languages and PDAs **23**
4.1 Interleavings 24
4.2 A thought about regular and context-free languages 24
4.3 Products of PDAs? 25
4.4 Re-use of variables 26

5 Computable Functions **28**
5.1 A conversation with two of my Queens' 1B CS students 28
5.1.1 Turing Degrees 29
5.2 Inverting Partial Computable Functions 29
5.3 This should be an exam question 30
5.3.1 Over dinner at the Boffafest: discriminators 30

6 Supervision Notes on the Part II Automata and Formal Languages Course **33**
6.1 Sheet 1 . . . 33
6.2 Sheet 2 . . . 39
6.3 CS 2017 Part IB Exercise sheet ex 12 . . . 42
6.4 Sheet 3 . . . 48
6.5 Sheet 4 . . . 54
6.5.1 Question 4, starred part . . . 54
6.5.2 Question 8 . . . 56
6.5.3 Question 9 . . . 56
6.5.4 Question 11 . . . 57

7 Discussions of questions on examples sheets for Part II Maths Languages and Automata from previous years **58**
7.0.1 Question 12 on Sheet 2 2018/9 . . . 58

8 Old Tripos Questions for Part II Maths Languages and Automata **59**
8.1 2017 . . . 59
8.1.1 2017:4:4H . . . 60
8.1.2 2017:3:11H . . . 60
8.2 2018 . . . 62
8.2.1 2018:1:12G . . . 62
8.3 2019 . . . 62
8.3.1 Paper 1, Section II 12H . . . 62
8.3.2 Paper 2, Section II 4H . . . 63
8.4 2020 . . . 63
8.4.1 Paper 1: 12F . . . 63
8.4.2 Paper 3: 12F . . . 63

9 Some Answers **64**

10 Appendix on The empty string **64**

11 Some thoughts about Trakhtenbrot's Theorem **65**

12 Prof Pitts' 1B CS Computation theory notes: A Discussion of Exercise 10 part (c) **65**
If i had a blog, i would put this in it. It's a collection of random stuff that occurred to me while I was supervising Languages and Automata, and 1B Computation Theory for the Compscis. Read it at your own risk.

The rhyme scheme of a sonnet is three quatrains $ABAB$ plus a couplet AA. Here A is the language of iambic pentameters whose final phoneme is a, and the other capital letters similarly. However there is no suggestion that the A and the B can be reidentified across the units. So one says that the rhyme scheme of a sonnet is $ABABCDCDEFEFGG$ (if one is a literature student) or $(AB)^2(CD)^2(EF)^2G^2$ (if one is a maths or CS student).

Notice that since the alphabet of phonemes is finite the rhyme scheme is not merely a regular language but is actually finite.

The usual breadth-first-search proof that every semidecidable set is the range of a total computable function actually proves that to any computable function there corresponds a total computable function with the same *multiset* of values.

1 Functions in intension and in extension

1.1 . . . and Why I was a traumatised Child

Why is the area under the hyperbola $\log(n)$? I was taught this at school and i'm trying to reconstruct it. And i am trying to reconstruct it beco's i didn't like the proof they showed me.

I think it goes as follows. Think of $\mathbb{R}^2$—the plane. The operation that stretches distances parallel to the x-axis (let's be posh and give its proper name: the *abcissa*) by a factor of m and shrinks distances parallel to the y-axis (the *ordinate*[1]) by a factor of m. It must preserve area. Not entirely sure how to prove that but never mind. (It certainly preserves the area of rectangles with their sides parallel to the two axes and that's probably enough). It's going to have to preserve the hyperbola too, so it had better send the point $(a, 1/a)$ to a point on the hyperbola. It will send it to $(an, 1/an)$ which is on the hyperbola so that's OK. Actually that was obvious, wasn't it, beco's the hyperbola is the locus of points p in the plane s.t. that the product of p's distance to the x-axis and its distance to the y axis is a constant. Duh.

The hyperbola is the graph on the plane of the function $x \mapsto 1/x$. Now consider the figure under the hyperbola bounded by the four points $(1, 0)$, $(1, 1)$, $(m, 0)$, $(m, 1/m)$. When we do the transformation the four points go to $(m, 0)$, $(m, 1/m)$, $(m^2, 0)$, $(m^2, 1/m^2)$. This new figure has the same area as the old, and the figure formed by joining them, namely the figure bounded by $(1, 0)$, $(1, 1)$, $(m^2, 0)$, $(m^2, 1/m^2)$, which is the area under the hyperbola up to m^2 is twice the area under the hyperbola up to m.

That's clearly the way to go. That must be what they did.

More generally . . . consider the figure under the hyperbola bounded by the four points $(1, 0)$, $(1, 1)$, $(n, 0)$, $(n, 1/n)$. When we do the transformation that shrinks/expand by a factor of m the four points go to $(m, 0)$, $(m, 1/m)$, $(mn, 0)$, $(mn, 1/mn)$. This new figure has the same area as the old. So we have shown

[1] No reason why we should rerserve the letters 'x' and 'y' in this way for input and output. Roll on λ-calculus.

that—for any m and n—the area under the hyperbola from 1 up to n is the same as the area under the hyperbola from m to mn. So the area under the hyperbola up to mn is the sum of the area under the hyperbola up to m and the area under the hyperbola up to n. Looks like logarithms to us, guv. Yes. Now we argue that, up to a constant factor, the log function is the only function that has this multiplicative property. I'm not quite sure how you prove that last bit but that's not actually the problem. For me, at any rate. The problem (for me) is that we have shown that the two functions (log, and the area-under-the-hyperbola) have the same *ex*tension; we haven't shown that they have the same *in*tension. I was probably expecting some clever syntactic manipulation. I think i would have been more satisfied with some reasoning concerning how to differentiate/integrate converses of analytic functions but of course they don't do that at A-level.

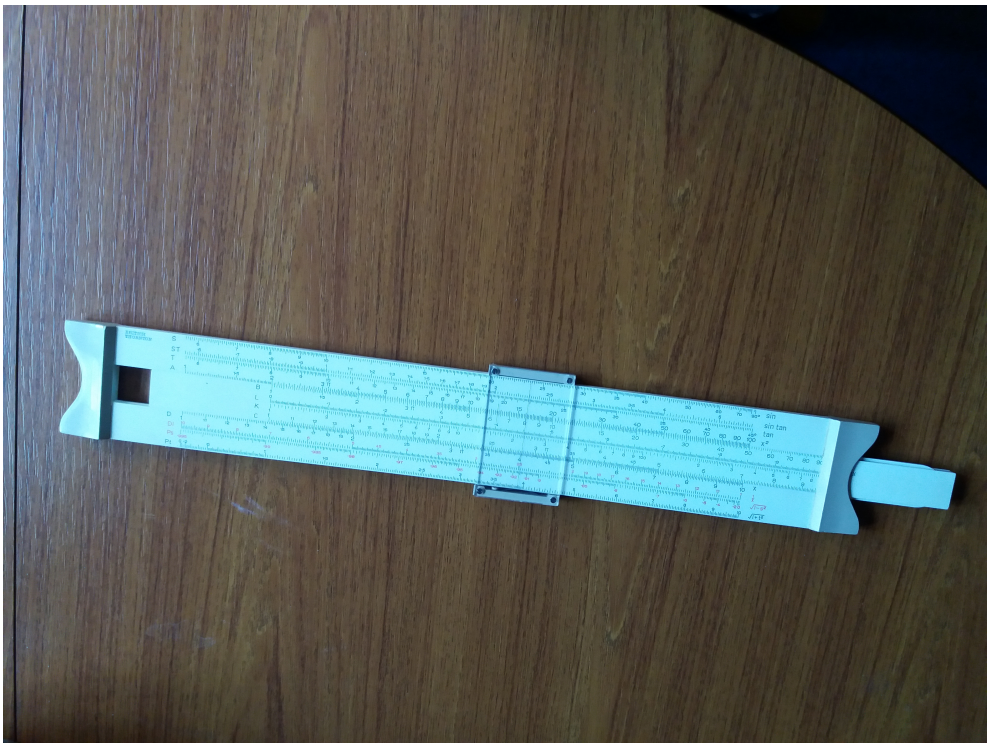
Come to think of it, why can't you derive it by using the chain rule? Differentiate $e^{log(x)}$ wrt 'x'. You must get $e^{log(x)}$ times whatever $\mathrm{d}log(x)/\mathrm{d}x$ is. And the answer had better be 1; so $\mathrm{d}log(x)/\mathrm{d}x$ had better be $1/x$.

Why didn't they do that?

1.2 Functions in intension and in extension, and Slide Rules

The function-in-intension/function-in-extension distinction looks clear enough when one is given paradigmatic examples: programs *vs* lookup tables. However these concepts have blurred edges, as one can see if one thinks about devices such as a slide-rule. (I am hoping both that your childhood was neither so deprived that you ever had to use one, nor so deprived that you don't know what they are!)[2].

[2] I keep one in my room in college, if you want to see one. I offered it to DPMMS for their cabinet of curiosities, but they turned up their noses.



Is a slide rule a function in extension or a function in intension? Well, it's much more like a function in extension than a function in intension. But's it's not just *one* function in intension: it's a cleverly designed, cleverly packed *family* of functions-in-extension. However it does look a bit like a function-in-intension (or perhaps a bundle of them) beco's there is an activity that looks like running a function on (or using) a slide-rule. If you want to extract information about a function from a slide rule you act on it in some way, or get it to act, and that makes it feel a bit more like a machine. I remember writing an essay (when i was a philosophy student) under the title "Does an adding machine add?" The point being that a pen doesn't write, it's the person *holding the pen* who writes—*by use of* the pen. Who is doing the adding? Is it the adding machine? Or the person *by use of* the adding machine? Of course that was in 1968, and there are no adding machines any more. Or rather there are, but they're all digital. The question is the same tho'.

This raises a question about packing. A function-in-extension is a lookup table. Now a lookup table is no different (in the relevant respects) from a .jpg file. It can be compressed in the same sort of way as an image file, by exploiting regularities. The more cleverly you compress the lookup table the more the result of that compression looks like a function-in-intension.

Probably *still* a good essay question!

A sly drool is of course an *analogue* device not a digital one, and that is one reason why it is hard to think about. It is *discrete* mathematics we are

concerned with here, not continuous mathematics. We haven't got a good theory of analogue computation.

I mention these issues around slide rules not to say that the intension/extension distinction is obscure, but to say that—despite slide rules—it is a useful one; there are plenty of situations where it is clear which of the entities in the picture are the intensions and which are the extensions.

2 A semiserious talk about Revelation: dedicated to Nathan Bowler

Not sure whether to call it 'Revelation and Recursiveness' (with the alliteration) or something that doesn't abuse the word 'recursive'. Anyway . . . !

A semidecidable set is one to which the external world has acquired complete information at the end of time. A set is semidecidable if there is a procedure that emits all its members.

There are two relevant theorems, due to Craig and Kleene respectively:

> (i) Every theory with a semidecidable set of axioms has an axiomatisation (in the same language) that is decidable.
>
> (ii) For every theory T in a language $\mathcal{L}$ with a decidable axiomatisation there is an extension $\mathcal{L}'$ of $\mathcal{L}$ and a theory T' in $\mathcal{L}$ that is a conservative extension of T and is finitely axiomatisable.

(The discussion of Computability and Revelation that follows started off as the author's attempt at an amusing (and possibly helpful) non-mathematical illustration of the difference between decidable and semidecidable sets. It appears to be spiralling out of control.)

According to Islam, Mohammad is the last prophet; there are to be No Further Revelations. Thus the set of revelations-according-to-Islam is finite, and therefore decidable. The set of its deductive consequences is semidecidable—at least; it may additionally be decidable. . . we don't know.

In contrast to Islam, both Christianity and Judaism hold out at least the *theoretical* possibility that there will be further revelation[3] Thus, in contrast to the situation with Islam, whose revealed set of truths is finite and therefore decidable, we cannot be sure that the set of truths that are (to be eventually) revealed by Christianity or Judaism is any better than *semi*decidable.

Now the set of deductive consequences of any semidecidable set is itself semidecidable. (Why?) So the set T_C of deductive consequences of Christian Revelation is a semidecidable theory. Now we can appeal to fact (i) above to infer that there is a decidable set Ax_C of revelations and deductive-consequences-of-revelations from which all of T_C follows.

[3]Spoiler: our chaplain at Queens' points out that this is contestable: the writer of the Book of Revelation ends his disquisition with a curse on the head of anyone who lets out any further revelations. One can easily see how the author of that book could feel he had had enough visions to be going on with thank you very much.

It's worth thinking a little bit about how we obtain this decidable set Ax_C, and what the decision procedure is. A decidable set is one to which we have *access* in the sense that its characteristic function is total computable. (Not that this carries any guarantee that we are *acquainted* with any means of computing this characteristic function!) Now: in the real world there is a difference between *sequential* access devices and *random* access devices ... and this distinction might be useful here: Ax_C is not a *random* access device, it's a *sequential* access device.

Once we have milked this example for all the insights it can afford us about the sequential/random distinction we can procede to consider the implications for us of point (ii). There will be a new language and a finitely axiomatisable theory $\mathcal{T}$ in it which is a conservative extension of Ax_C. Did i say 'will be'? *There is already*. Now what would theologians *not* give to get their hands on this text??

I think that if we are to understand what in God's name is going on it could be a very good idea to understand how (i) and (ii) are proved, and persuade ourselves that they work for all theories, not just those with mathematical subject matter. For my part I would be very glad to be pushed into such an investigation. I sort-of understand (i), but I have never worked through a proof of (ii).

Here is a proof of (i).

Remark 1 *(Craig)*
If a theory T has a semidecidable set of axioms, then a decidable Ax_T set of axioms can be found for it (in the same language).

Proof:
Let $\mathfrak{M}$ be a volcano that emits axioms of T, and notate the nth axiom emitted by $\mathfrak{M}$ as ϕ_n. Then we obtain a decidable axiomatisation Ax_T for T as

$$\{(\bigwedge_{0\leq i<n} \phi_i) \to \phi_n : n \in \mathbb{N}\}$$

We need to spell out why this axiomatisation is decidable. Let $\langle \phi_n : n \in \mathbb{N}\rangle$ be the stream of axioms emitted by the volcano that is zigzagging over the computable function f whose range is the set of axioms. (Of course in our present setting these are the Revelations that pop up from time to time, so we don't have to worry about any zigzagging). The axioms in the decidable set are

$$(\bigwedge_{i<n} \phi_i) \to \phi_n$$

The decision procedure for this set of axioms is as follows; on being presented with a formula ...

> If the formula is the first thing emitted by the volcano, accept; else
>
> If the formula is not a conditional, reject;
>
> If its antecedent A is not a list-conjunction ask whether or not A is the first thing emitted by the volcano. If it isn't, reject;
>
> If the antecedent is a list conjunction of length n check that, for each $i < n$, the ith thing in the list is the ith thing emitted by the volcano and that the consequent is the nth thing emitted by the volcano. Accept iff this condition is satisfied.

■

It may be worth noting that if $\langle \phi_i : i \in \mathbb{N} \rangle$ is the sequence of revelations presented to us in time, and our axioms are the conditionals outlined above then this axiomatisation stands a good chance of being independent. It may be a fair assumption (i'd have to ask the theologians) that—for all i—ϕ_i does not follow from earlier ϕ_j. After all, if ϕ_i follows from the earlier ϕ_j then—in fairness—it can hardly be said to be a revelation, can it?[4] If this assumption holds good then none of the axioms

$$\{(\bigwedge_{0 \leq i < n} \phi_i) \to \phi_n : n \in \mathbb{N}\}$$

follow from any of the others: the axiomatisation Ax_C is *independent*.

Suppose I am given a formula and i wish to know if it is an axiom of the decidable axiomatisation. I might have to wait for the volcano to emit n axioms from the semidecidable set, where n depends on the length of the candidate. How long might that take? One's first thought is that it might take a ridiculously long time. (Indeed it might). *So* long, in fact, that there is no computable bound on the time taken. But that doesn't follow. The question is not:

> (1) If f"$\mathbb{N}$ is not recursive can we bound time taken to learn that $x \in f$"$\mathbb{N}$ by a computable function applied to x?

but rather

> (2) How long does a volcano for f take to emit x values? Can we bound this time by a computable function applied to x?

The answer to (1) is—obviously—"no", and for the usual reasons; the answer to (2) might be 'yes'!

If a theory has a semidecidable set of axioms then in some sense it has finite character; remark 1 captures part of this sense by telling us it will have a decidable set of axioms. In both these descriptions the finite character is expressed in a metalanguage. The following remark tells us that this finite character can be expressed in a language for T.

[4] Perhaps the Revelations are not flagged as such...? Perhaps checking that something is a revelation involves establishing its independence from earlier revelation...?

I suppose the point that is disquieting me is the thought that the *finite object* that is the Turing machine or register machine that guards the decidable axiomatisation is one that we can't locate in finite time. Or can we? It's finite, so we must have found it at some finite stage. It's just that we don't know when we've found it. We have lots of candidates of course but we never know when we have reached a stage when no revision is necessary. A detailed discussion may be in order.

Remark 2 *(Kleene, [2])*

If T is a recursively axiomatisable theory in a language $\mathcal{L}$ with only infinite models, then there is a language $\mathcal{L}' \supseteq \mathcal{L}$ and a theory T' in $\mathcal{L}'$ and T' is finitely axiomatisable and is a conservative extension of T.

Proof: Omitted. ■[5]

(We can uniformly expand any $\mathcal{L}$-structure that is a model of T into a $\mathcal{L}'$-structure that is a model of T'.)

(The idea in the proof is to formalize the inductive clauses of the truth definition for T. The basic references are [2] and [1]. There is a very clear review of both papers by Makkai [3] that also provides a sketch of the proof.) You may have seen (or be about to see) some examples of this phenomenon in Part II *Logic and Set Theory*. Bipartite graphs, algebraically closed fields... Another illustration of this process is afforded by the way in which the (pure) set theory ZF (which cannot be finitely axiomatised) corresponds to the class theory NBG, which *can* be finitely axiomatised. Why would one expect this to be true in general? A theory that is recursively axiomatisable is underpinned by a finite engine that generates all the axioms. It ought to be possible to hard-code this engine into the syntax, if necessary by enlarging the language. I have the feeling that it should be possible to do this without invoking truth-definitions. I keep hoping that someone will show me a proof.

Might it be easier to prove Kleene's theorem for automatic theories than for arbitrary recursively axiomatisable theories...?

The sequential/random distinction can be applied to abstract devices as well. How long does it take to read an entry e in the device? Well, at least e, so one doesn't really want to say that a random access device is one whose characteristic function is computable in constant time and a sequential device one whose characteristic function has strictly (perhaps *steeply*) increasing cost function.

A possible distraction: every decidable—indeed every *semidecidable*—set is the range of a primitive recursive function. But of course that doesn't mean that every characteristic function is primrec.

we need a section entitled:

Characteristic Functions and the sequential/random Distinction

Of course one can make the exact same point about the revelation of Mathematics.

[5] I am omitting the proof since i cannot find a proof that doesn't use truth-definitions, and i haven't got time or space to go into them.

A fatal flaw!

The set of truths revealed by Christianity or Judaism looks semidecidable but actually it needn't be. Suppose, once it's all revealed, it turns out not to be semidecidable; that doesn't contradict what we know. All we know so far is that we have access to a finite initial segment of it. But then we have access to finite initial segments of—all sorts of things: the set of codes for total functions; we are daily unearthing new members of the highly undecidable set that is True Arithmetic. And who is to say that we won't have unearthed every last one of them by the end of time? (Does this even make sense?)

This prompts us to think about the following situation. There is a set X, concerning which positive information about membership is revealed to us in finite dribbles. Possibly negative information too, but let's not assume that. The nature of the dribbling is such that *at the end of time* we have complete information about membership of X; every member has been dribbled. (perhaps 'leaked' is better). Does this mean that X was semidecidable? One wants to say not: the dribbles might be no more than *leaks*—crumbs capriciously dropped from the table of a highly noncomputable gatekeeper for a highly undecidable set. The important question is whether or not the leaks were the result of *the activity of a finite engine*. If they aren't then we cannot infer that the set of revelations is semidecidable.

The wider point here is this: one tends to say that the hallmark of the semidecidable set is that each and every one of its members gets revealed to us at some point before the end of time. That is the intuition one tries to get across to students, but it's not the whole story. It's a necessary condition all right, but it's not sufficient. It's necessary also that the revelation be done by a humble *finite* engine. For consider the set of gnumbers of total computable functions. The oracle for this set could divulge all its members to us over time—in increasing order indeed—and we would know all its members and all its nonmembers by the end of time, but that doesn't make it semidecidable. The oracle is not a finite engine!

Another way to see it. God could recite to us the members of the complement of the `HALT`ing set—in increasing order, one at a time, at noon every day like the shipping forecast. That doesn't make that set semidecidable.

So no Templeton money. Chiz.

But what about the engine that reveals mathematics to us over the ages? It's a natural process. If all of physics is computable then the argument/construction that i have tried to run for theology actually works. So there really is a finite body of first-order mathematics from which all the first-order mathematics that we will ever know can be deduced.

3 DFAs

> Regegegexp quarks quarks!!
>
> *A chorus of DFAs.* (with apologies to Aristophanes)

I have always thought that set of acceptable acronyms is in some sense dense in the semantic space of words: you can get any acronym to mean anything, or something arbitrarily close to anything. Here is a story from *Flight* magazine **171** number 5089 in 2007.

"Those clever people at DARPA want to demonstrate technology for a fixed-wing UAV that can stay aloft for five or more years. The programme is called VULTURE for Very high altitude ULTra-long endurance Unmanned Reconnaissance Element."

"I don't think VULTURE sounds right ... Gives the wrong impression"

"So ... what would you have called it?"

"Something with a more soaring theme ... maybe ALBATROS—Airborne Long Beyond Any Time Reasonable Or Sane"

To settle the conundrum we asked *Budgie News* random acronym guru Max Cue to come up with some other solutions:

> PARROT—Permanently AiRborne Ridiculously Over the Top
>
> DODO—Decidedly Over-Designed Observer
>
> BUDGIE—Big Unmanned Definitely Guaranteed to Injure Eventually.
>
> And of course to deal with the mind-boggling boredom of operating and monitoring the system, he suggests a parallel training and support unit called BLUE TIT(S) ...
>
> Big Loitering Unmanned Experiment Temporary Insanity Training System.

3.1 Word ladders

Can I extract an exercise out of this, I wonder?

You know about word ladders: things like

SKY
SAY
SAD
SAT
SET
SEE
SEA

set in stone on the waterfront in Wellington: `20201111_162558.jpg`

Or
WORD
WORE
GORE
GONE
GENE

which i found in *Towards a Theoretical Biology*. It makes a point about genes being words in a suitable language.

Ian Stewart (in *Nature's numbers* p 41) makes the point that if you want a word-ladder that takes you from SHIP to DOCK, then you must, somewhere *en route*, have a word with two vowels (at least if every legal word has a vowel)—beco's the vowels in SHIP and in DOCK are in different places. I wondered if there is a theorem lurking in there....

Now! let L be any language you please, regular. context-free, whatever. The relation between two words of L of being connected by a word-ladder (of words all in L, of course) is obviously an equivalence relation. My question is: does this equivalence relation actually *do* anything? Can i extract from it an exercise for my students? Can you?

3.2 What can DFAs remember?

Perhaps the pædogogical point to make it that altho' a DFA can remember only a finite amount of stuff it can nevertheless remember it for an arbitrarily long time. Worth thinking about this distinction. Space and time are not interchangeable!

• One thinks of this in connection of vowel harmony in the phonological rules of natural languages. In languages with vowel harmony, `https://en.wikipedia.org/wiki/Vowel_harmony` in any one word the vowels must either all be (for example) front vowels (Kirribilli) or all be back vowels (Wooloomooloo[6] but not *Nullarbor* which is not an aboriginal word.). Lots of Australian Aboriginal languages have vowel harmony. The language of legal *sounds* (strings-of-phonemes) for any natural language is always a regular language. This is beco's the sounds that you are allowed to use depend only on the last k phonemes you have seen, for some fixed k. The rules might tell you that if all the vowels in the last 5 characters were back vowels then the next character has to be a back vowel too (if it is a vowel). This can propagate out to arbitrary length.

• Think about the latch. `https://en.wikipedia.org/wiki/Flip-flop_(electronics)` Often pointed out that it underlies memory devices. It's very easy to design a (*finite state*!) machine that remembers the last string of three identical characters that it saw.

• A simple case. Let Σ be an alphabet with $0 \in \Sigma$ and $1 \in \Sigma$. Of course it may have lots of other stuff too. We can design a two-state machine that is in one state when the last character from $\{0, 1\}$ that it read was 0 and is in the

[6] There really is a place called Wooloomooloo; i didn't make it up—nor did Python: `https://en.wikipedia.org/wiki/Bruces_sketch`

other state when the last character from $\{0,1\}$ that it read was 1. Notice that what we have here is really a kind of superposition of two machines, one of which recognises strings whose last character from $\{0,1\}$ was 0 and the other one of which recognises strings whose last character from $\{0,1\}$ was 1. Fortunately for us these two machines have the same transition table (they differ only in their accepting states) so we can superimpose them to get a single machine which has *two* accepting states.

• In this connection consider the language over the alphabet $\{a,b\}$ where no block of k (k fixed in advance) consecutive characters contains three evenly spaced 'a's or three evenly spaced 'b's. This is obviously regular, beco's you only have to remember the last k characters. (Brief reality check: you need at most 2^k states, and the '2' comes beco's there are two characters.) But what happens if you discard the k bound, and reject any word that has three evenly spaced 'a's or 'b's, *irrespective of how widely spaced they are?* Is that a regular language? If it is, it will be for entirely different reasons. (And of course there's nothing special about the number three here). This is a hard question!

There is a distant connection with the Ackermann function which i will tell you about below, section 3.6

3.3 How to Pump a Wasp

One sense in which DFAs do not have memory is that, altho' in some sense they know what state they are in (at least to the extent of actually *being* in that state) in no sense do they have any awareness of having been in that state before. It is this idea that lies behind the pumping lemma.

(I suppose the corollary ought to be that, for machines that do not obey the pumping lemma, there ought to be a way of thinking of them as being able to remember what states they have been in earlier. However i have never found a way of telling that story in a way that makes sense.)

I remember, when i was a neuro student, being very struck by this passage in D. Wooldridge: "The Machinery of the Brain", (McGraw Hill 1963) p 82. I made a note of it at the time, and i have never forgotten it.

> "When the time comes for egg-laying, the wasp *sphex* builds a burrow for the purpose and seeks out a cricket which she stings in such a way as to paralyse but not kill it. She drags the cricket into the burrow, lays her eggs alongside, closes the burrow, then flies away, never to return. In due course the eggs hatch, and the wasp grubs feed off the paralysed cricket, which has not decayed, having been kept in the wasp equivalent of deep freeze. To the human mind, such an elaborately organized and seemingly purposeful routine conveys a convincing flavour of logic and thoughtfulness—until more details are examined. For example, the wasp's routine is to bring the paralysed cricket to the burrow, leave it on the threshold, go inside to see that all is well, emerge, and then drag the cricket in. If, while the wasp is inside making her preliminary inspection, the cricket is

> moved a few inches away, the wasp on emerging from the burrow, will bring the cricket back to the threshold, but not inside, and will then repeat the preparatory procedure of entering the burrow to see that everything is all right. If again the cricket is removed a few inches while the wasp is inside, once again the wasp will move the cricket up to the threshold and re-enter the burrow for a final check. The wasp never thinks of pulling the cricket straight in. On one occasion, this procedure was repeated forty times, always with the same result."

Clearly you can apply the Pumping Lemma to the wasp! [To my shame, it is only recently that i have made the connection![7]]

3.4 The Easy Way into the Pumping Lemma

Forget the symbols and the statement, get the *idea*.

Suppose you have a machine with 17 states, and you give it a string of length 23, and it's ended in state s. It's gone through a loop. Now identify the substring that sent it through that loop. You can "pump up"that substring that sent it through the loop, by putting in as many copies of that substring at that point as you like—even zero! When you feed the machine the pumped-up string it will end up in the same state s as before. Obvious, isn't it!?

What use is this? You have some language L in mind. You take a string for L that is longer than the number of states in the machine you are trying to refute. The machine reads your string and ends in an accepting state S. You now pump up the string. Pumping up the string doesn't affect whether or not the result is accepted by the machine (as we have seen) but it might affect whether or not the pumped string is in L. That is the possibility you have to trade on.

There is an essay to be written about *agency* and the pumping lemma. Key idea in the pumping lemma is the pigeonhole principle. You feed the machine a string of length 23 and it has only 17 states, so it has been through a loop. You, the user sitting at the machine's console, have no access to information about which state it is in. All you know is whether or not the machine is in an accepting state. (Of course if your machine is a Moore or Mealy machine you know a bit more, but you aren't given a circuit diagram). So here is a challenge: if i am given an arbitrary DFA, equipped with a reset button, how do i recover its state diagram? Simple. I just try it on all finite words by exhaustive search, using the `RESET` button. What if there is no `RESET` button? Then i think i'm stuffed.

"All you know is whether or not the machine is in an accepting state." That is, you know only the coarsest equivalence relation on words. Is there a system—

[7] And yet more recently still that i discovered this: `https://www.youtube.com/watch?v=xdFyQJyo3Ps`

a sequence—of experiments that gives you the progressively finer equivalence relations? Yes, but it's a bit laborious.

It's because you don't have the information you need (in the absence of a `RESET` button) that you have to interact with the owner of the DFA; hence the *game*. You are allowed to ask the owner how many states it has.

3.5 How to spot a regular language

I am in a darkened room, whose sole feature of interest (since it has neither drinks cabinet nor tea-making facilities) is a wee hatch through which somebody every now and then throws at me a character from the alphabet Σ. My only task is to say "yes" if the string of characters that I have had thrown at me so far is a member of L and "no" if it isn't (and these answers have to be correct!)

After a while the lack of tea and drinks cabinet becomes a bit much for me so I request a drinks break. At this point I need an understudy, and it is going to be you. Your task is to take over where I left off: that is, to continue to answer correctly "yes" or "no" depending on whether or not the string of characters that *we* (= first I and then you) have been monitoring all morning is a member of L.

> **What information do you want me to hand on to you when I go off for my drinks break? Can we devise in advance a form that I fill in and hand on to you when I go off duty? That is to say, what are the parameters whose values I need to track? How many values can each parameter take? How much space do I require in order to store those values?**

The point is that thinking about the problem in this way can help make it clear whether or not the language you are looking at is regular. If you can bound in advance the number of bits of information that you want from me, then we know that the language is regular because the machine that i am emulating in my person has only finitely many states.

3.6 A Riff on van der Waerden's Theorem

I think i owe you a health warning about the thought-experiment in section 3.5. The thought experiment is very useful, and when it tells you that a language ought to be regular (beco's you need to maintain only a finite amount of information) then you can trust it. However there are examples of languages that are regular for quite deep and obscure reasons which nevertheless appear to be non-regular according to the thought experiment. And it is one such that is the topic of this section. Let me explain.

Let your alphabet be, say, $\Sigma = \{a, b, c, d\}$, and consider the language L consisting of those strings from Σ^* which do *not* contain five evenly spaced as, nor five evenly spaced bs, nor five evenly spaced cs, nor five evenly spaced ds. The thought experiment strongly suggests that L is not regular, because it looks

as if you need to store the whole of the string-seen-so-far. After all, the spacing could be as wide as you please, and the a that you are about be given through the hatch might line up with four earlier widely-spaced beans i mean as to make five. How do you exclude the possibility that the bean you have just received isn't the same colour as the very first bean you received and as three other evenly-spaced beans in between?

Indeed it even looks as if there is a straightforward application of the Pumping Lemma to prove that this language L is not regular. Suppose L is regular; let w be some word in this language; divide it up into w_1uw_2 where u is the pumpable bit in the middle. Then $w_1u^nw_2$ is in L for any n. Now certainly $w_1u^5w_2$ contains (in u^5) five (indeed n) evenly spaced as (or bs, or whatever), and therefore cannot be in L. So L is not regular.

Nevertheless the language is actually *finite* and is therefore regular after all! This is very far from obvious(!)

What has gone wrong?

The point is that not every regular language is pumpable. At least not in any useful sense. You can always take the middle bit (the y in xyz) to be empty—but then in that sense everything is pumpable. Only *infinite* regular languages are pumpable. The DFA for a finite language cannot go through a loop on its way to an accepting state.

van der Waerden's theorem concerns the situation where you have a potentially infinite supply of beads—of k different colours, k finite—and you are laying them down in a line, each the same distance from its neighbours. If i want my bead display to have, say, n evenly spaced beads of one colour, is there a j so large that whenever i put down j of these beads in a line, all evenly spaced, then there are n evenly spaced beads of one colour? Van der Waerden says there is always such a j. The standard proof of van der Waerden gives a function that bounds the length of the string you need. It's a nice (*very nice*) proof, but the function it gives is an Ackermann function, something that isn't primitive recursive. This contradicts what for years i used to tell my students, namely that unless you go out of your way to *look* for trouble you will not encounter any functions $\mathbb{N} \to \mathbb{N}$ that aren't primitive recursive. There are no naturally-occurring computable functions $\mathbb{N} \to \mathbb{N}$ that are not primitive recursive. Well, that's not quite true, as this example shows. All the same it is—so i am reliably informed—true that if you work very hard on optimising the proof then you can get a primitive recursive bound. But you have to work very hard, as i say, and I for one have never read the details.

Joe Hurd has supplied me with the following direct proof of van der Waerden. which I think is the standard proof, since it sounds remarkably like a proof i heard Imre give years ago. I've hacked it about.

We start with two colours—red and blue—and try to find three evenly spaced beads of the same colour. So, assume we have a colouring function $c : \mathbb{N} \to \{\text{red}, \text{blue}\}$.

We imagine a finite string of beads, of indeterminate length, It has a start point but the end point is in the far distance somewhere. A **block** is a contiguous

set of beads, and the string of beads is partitioned into blocks. How many different ways are there of colouring a block of length 5? (Don't worry for the moment why we have picked on 5). Answer: obviouly $2^5 = 32$. So—think *pigeonhole*—consider the first 33 blocks of length 5 with their colourings: call them $B_1, \ldots B_{33}$. There must be a block that is repeated, say $B = B_i = B_j$ for $1 \leq i < j \leq 33$.

Now B (up to red-blue symmetry) must have one of the following forms:

red	red	xxx*	xxx	xxx
blue	red	red	xxx*	xxx
red	blue	red	xxx	xxx*

where xxx can be either colour, and the * is the **focus** of the block. The focus is the address that will supply us with the third of our evenly spaced beads of the one colour. If the focus in B is red, then we have found our three evenly spaced reds, so assume the focus in B is blue.

Now consider B_k, where $k = j + (j - i)$, (so we are considering the three—evenly spaced—blocks B_i, $B_{i+(j-i)}$ and $B_{i+2(j-i)}$ and look at the point n in B_k with the same position as the focus in B. Call n the **superfocus**.

If n is blue, then it lines up with the blue foci in B_i and B_j to give us three evenly spaced blue beads.

If n is red, then it lines up with the first red in B_i and the second red in B_j, to give three evenly spaced red beads.

Either way this tells us that that if we have $33 + 33 = 66$ blocks each of length 5—making 330 beads—then we must have three evenly spaced red beads or three evenly spaced blue beads. 330 is ridiculously large—you can check that 9 beads are enough. However it's a proof of concept, and it shows us how to do an induction.

Details of the induction will follow!

As you can see from the proof of the induction step (or will be able to see once i have written out the bloody details!) the proof involves a double induction rather like the double induction used to prove the totality of the Ackermann function. The outer loop is the induction on the length of the monochromatic strings, and the inner loop is on the number of colours.

3.6.1 A new Way of proving van der Waerden?

To return to start of this section, and the language L .... There is a possibility here that i haven't explored.

Van der Waerden's theorem says that (i abbreviate) the set of counterexamples is finite. (Every sufficiently long string contains four evenly spaced a or etc etc). Let $L(l, c)$ be the language over a c-sized alphabet consisting of those words that do not contain l evenly spaced tokens of any of those characters. These languages are all finite and they all have regular expressions. I'm not asking for those regular expressions, but it would be nice to know if there is a way of obtaining the regular expression for $L(l, c)$ from regular expressions for

assorted $L(l', c')$ for smaller l', c'. If there is, there would be a proof by induction that all the $L(l, c)$ are regular. We would then be in a position to use the Pumping Lemma to show that since $L(l, c)$ it regular it must be finite.

So the project is to prove that if all $L(l'c')$ have regular expressions for l', c' suitably bounded then so does $L(l, c)$.

Perhaps the answer is: yes, there is a proof, and it's just the focussing argument. So perhaps there is nothing to be gained.

3.7 NFAs are nondeterministic not probabilistic!

You could decorate the directed arrows in an NFA with probabilities, so that, for each state σ and each letter l, the real numbers on all the edges exiting σ and labelled 'l' add up to 1. If you are then given a probability distribution on the letters of the alphabet you can say things about the probability of your being in any particular state long-term. The study of this sort of thing is called 'Markov processes' (i *think*). But it is **nothing** to do with our concerns here!

3.8 Polynomial growth?

Let L be an infinite regular language. Is there anything we can say about the growth rate of $|L \cap \Sigma^n|$ (the number of words in L that are of length n)? It can be exponential: $(a|b)^*$ is an example. Or the set of binary representations of multiples of 3. Does the pumping lemma tell us *anything*...? Well, the lim inf is linear. And it might be no more than that: 0^*.

3.9 Enumerating DFA's

[also 2017 paper 1 section II 11H]

A question on Maurice Chiodo's Languages-and-Automata sheet (and it's probably on Languages-and-Automata sheets anywhere in the universe) invites his victims to produce a function that enumerates all the deterministic finite state machines. Yawn, yawn. *Of course* there are only countably many of them yawn yawn and *of course* one can set up a specification language for them and then order the specifications lexicographically etc etc yawn yawn *yawn*!

But this is actually slightly problematic (as we progressives are fond of saying). Part of the description of a DFA is the alphabet of characters that it reads. For any DFA that alphabet is finite. We want a uniform description of the DFAs (*that*, after all, was the point) so we have to enumerate—somehow—all the alphabets the DFAs might use. Each alphabet is a finite subset of some cosmic collection of all characters in God's mind's eye, and there could be, well, God knows how many. So we have to assume that there are only countably many characters. So we dole out to each machine a finite subset of this countable set. A subset? One doesn't want to have two distinct machines which can be turned into each other by a permutation of the cosmic alphabet.... So we have to fix an enumeration of the cosmic alphabet and then think of each machine's alphabet as an initial segment of it.

To be fair to the examiners they tell you to assume that there is, indeed, precisely one cosmic alphabet, and that it is precisely $\mathbb{N}$.

The DFA is to be identified with a finite set of states and binary relation on that set, plus a few other minor details. It has n states. OK, what order do we write them in? We can distinguish the states by the words that lead us thither from the start state. Associate to each state the first such word, in the lexicographic order on words. But for that we need to have an order on the alphabet. So every alphabet has to be canonically ordered and—since we are looking for a global enumeration of all DFAs—this means that the union of the alphabets has to be wellordered. That's why the examiners told you that you may take the alphabets all to be subsets of $\mathbb{N}$.

Give an example of a regular language such that every DFA that recognises it has more than one accepting state. Is the minimum number of accepting states possessed by any machine that recognises L an interesting parameter of L?

The language over $\{0, 1\}$ that has, say, an even number of 0s and of 1s or an odd number of both. Something along those lines. Should be possible to cook up a language s.t. no machine that recognises it has only one accepting state. I'm wondering if examples of this kind can be processed into languages over larger alphabets whose characters are words in the original alphabet. Then you get a machine with only one accepting state and the old language is a quotient. A vague thought.

Actually August Liu has nailed this for me. Let L be any finite language over a singleton alphabet. If it has n strings the machine that recognises it must have n states. Duh!

What is the correct notion of product of machines with only one accepting state?

On of my students pointed out to me that the concept of the ϵ-*closure* of a state is useful. The ϵ-closure of a state σ (or do we mean of $\{\sigma\}$?) is the set of states one can reach from σ by following ϵ-transitions. He's probably right, but i'd never tho'rt about it beco's ϵ-transitions are The Work Of The Devil, as any fule kno.

Here's how to see what the finite state machine is that recognises the language you wish to show to be regular. Consider the example of the set of strings in $\{\text{`0'}, \text{`1'}\}^*$ that denote (in the ordinary semantics) numbers divisible by 3.

You are in the following situation. You are given a string w from this alphabet, and are told that you will be given a character, either '0' or '1', to append to the string w, after which you will have to say whether or not the new string w'—which is w with the new character stuck on the end—denotes a number divisible by 3. *What do you want to know about w??*

But actually, it's slightly worse than that, because ... altho' what you *actually* want to know about w is whether or not the new string w'—which is w with the new character stuck on the end—denotes a number divisible by 3, nevertheless knowing a value of some parameter F on the basis of which you can answer whether w' denotes a multiple of 3 or not ... is not enough! You also have to be able to compute the value of F for the any string w'' obtained by sticking characters on the end of w', beco's you are going to be asked the very same questions about them as you were being asked about w'.

So the question is: "What is this parameter F?" I have heard people use the word 'maintain' in this connection: "What information about the string-one-has-so-far-seen does one have to *maintain*?"

In general the challenge is this: Given a language $L \subseteq \Sigma^*$, can i find a finite-valued parameter F such that... Whenever i know the value of $F(w)$ then, for any $c \in \Sigma$, i can both

(i) answer the question "$w :: c \in L$?"; and
(ii) compute the value of $F(w :: c)$.

?

(Here i am using the `Standard ML` notation of double colons for `cons`ing).

The finitely many values of F become the states of the DFA.

3.10 Any connection between Quantifier Elimination and Automaticity?

If you are doing CS IB Logic and Proof you will have encountered Quantifier-elimination in the final section of Larry's Logic-and-Proof notes. A theory obeys quantifier-elimination iff every formula (of the appropriate language) is equivalent to one without quantifiers. The theory DLO of dense linear order has Quantifier Elimination. Illustration: $(\exists z)(x < z < y)$ is equivalent to $x < y$. Not many theory have QE (that's not quantitative easing btw) but it's very useful when a theory does.

Presburger Arithmetic `https://en.wikipedia.org/wiki/Presburger_arithmetic` is a theory with signature $\langle \mathbb{N}, +, \leq, 0, 1 \rangle$. There is also, for each concrete k, a unary function `div` k. It does not have QE. However, if we add, for each concrete k, a unary predicate is-divisible-by-k we have a theory that *does* have QE. This is in Marker's book: `https://www.springer.com/gp/book/9780387987606`

I have this cute thought-experiment that tests whether or not a language is regular; is there a tweak to it that gives us something that tests whether or not a language is context-free?

Fox Thompson makes a rather good point. Consider additions written in unary notation, things like $111+111 = 111111$. Is the set of those assertions that are true a context-free language? Pretty clearly yes: build a PDA that pushes 1s on the stack (ignoring '+') until it sees a '=', after which it pops 1s off the

stack, and accepts with an empty stack. But what about true multiplications? I bet they're not a CFL ...

Observe that there is a 2-state Mealy machine that adds two binary strings. Its alphabet is $(\{0,1\}\times\{0,1\})\cup\{\texttt{EOF}\}$. It has two states: `carry` and `don't-carry`. The initial state is `don't-carry`, and its transition table is

If in state	and reading	go to state	and emit
`carry`	$\langle 0,0\rangle$	`don't-carry`	1
`carry`	$\langle 0,1\rangle$	`carry`	0
`carry`	$\langle 1,0\rangle$	`carry`	0
`carry`	$\langle 1,1\rangle$	`carry`	1
`carry`	`EOF`	`don't-carry`	1
`don't-carry`	$\langle 0,0\rangle$	`don't-carry`	0
`don't-carry`	$\langle 0,1\rangle$	`don't-carry`	1
`don't-carry`	$\langle 1,0\rangle$	`don't-carry`	1
`don't-carry`	$\langle 1,1\rangle$	`carry`	0
`don't-carry`	`EOF`	`don't-carry`	`null`

Take a moment or two to think about the challenge of designing a Mealy machine to *multiply* two bit strings.

Mind you, we didn't define *automatic structure* in terms of Mealy machines but rather in terms of FSAs. So what one should really be doing is defining a finite state machine whose alphabet is $\{0,1,\texttt{EOF}\}^3$. It will have three ports not two, and it will have an accepting state which it reaches if the string of entries in the third port is the sum of the string of entries in the first two ports. (It will also need a `fail` state. The reader might like to supply details of this machine.)

3.11 Regular Languages for Numerals

I remember talking to Jack Button about how there might be a good exam question about this, and promising not to improperly alert my students to this possibility. Well, a question on this topic has now come up! It's 2020 paper 3 12F.

Whether or not the set of notations for members of an infinite set is a regular language or not isn't controlled to any great extent by the set. Consider powers of 2. The set of base-2 notations for powers of 2 is obviously a regular language; initially i expected that the set of base-10 notations for powers of 2 would likewise be regular but it is not. Suppose it were. Then there are [decimal] natural numbers a, b, c such that any word of the form ab^nc denotes a power of 2. Let us notate the power of 2 thus denoted as 't_n', and write 'β' for the length of b. Then $t_{n+1} - 10^\beta \cdot t_n$ is less than some quantity k determined by b and c and not depending on n, thus:

$$t_{n+1} - 10^\beta \cdot t_n < k.$$

Divide through by t_n:

$$t_{n+1}/t_n - 10^\beta < k/t_n.$$

Now consider what happens when n gets large; the RHS eventually becomes less than 1, so we must have

$$t_{n+1}/t_n = 10^\beta$$

which is impossible beco's the LHS is a power of 2 and the RHS is a power of 10.

So it's just not true that, for any n and any b, the set of base-b notations for powers of n is a regular language.

In contrast, for any n and any b, the set of base-b notations for *multiples* of n *is* a regular language.

Perhaps there is a point to be made about the difference between exponentiation and multiplication .... Is the set of base-b representations of powers of a a context-free language?

There's certainly a point to be made about whether or not a set of natural numbers corresponds to a regular language is not a property of the set but a property of the representation of it. But that is an old point. Easy to check divisibility by 7 in octal but hard in decimal. For divisibility by 11 it's the other way round.

4 Context-free Languages and PDAs

This came up in connection with stratification, substitution and weak stratification. Typing and substitution are good things for CompScis to think about. A **stratifiable** expression of the language of set theory (only predicate symbols are '=' and '$\in$') is one where all the variables are decorated with integers in such a way that if '$x \in y$' is a subformula then the decoration on 'x' is one less than the decoration on 'y' and if '$x = y$' is a subformula then the decoration on 'x' equals the decoration on 'y'. Such a decoration is a **stratification**. A formula is *weakly stratifiable* iff you can stratify its *bound* variables (The free variables can go to hell). Thus '$x \in x$' is not stratifiable but it is weakly stratifiable. The collection of stratifiable formulæ is not closed under substitution ('$x \in x$' is a subformula of '$x \in y$' after all) but the set of weakly stratifiable formulæ *is* closed under substitution.

The class of wffs of the language of set theory is context-free. (I think this is a known standard fact) Is the class of stratifiable formulæ context-free? Presumably not, tho' i'd like to see a proof one way or the other. The class of weakly stratifiable formulæ? I'm guessing not, but—again—i'd like to see a proof one way or the other.

A challenge from my colleague Marcel Crabbé. Consider the following language L over $\{a, b\}$. It contains a, b and ab. Also, if w and u are in L then so are $[w/a]u$ and $[w/b]u$. I.E., any letter in any word can be replaced by any word. Is this language regular? Context-free?

It's pretty obviously not regular (tho' i don't know how to prove it) and i'm guessing it's not CF either.

Marcel says (and i quote):

> "Consider the language on the alphabet $\{a, b\}$ consisting of: a, b, ab and all the strings resulting from substitution of a grammatical expression for a or for b in a grammatical expression.
>
> For example, a, ab, b, $ababb$ and $aabaabab$ are ok, but ba is not.
>
> Is there an alternative (more appealing) description of this language?"

Not clear whether he requires that a substitution should replace *all* occurences of the variable being replaced, or whether we are free to replace only *some* of them.

How about

$S \to A \mid B \mid AB$

$A \to a \mid S$

$B \to b \mid S$

Marcel sez:

"Yes, but how do you get $ababab$, $abbabbbb$... Note also that $abbab$ is NOT grammatical."

We get ...

length 1: a, b

length 2: ab but no others

length 3: $[ab/a]ab = abb$ and $[ab/b]ab = aab$ but no others.

I think Marcel is wrong: once one gets these two we can substitute abb for the first a in aab. But then perhaps he insists that *all* occurrences of a variable should be substituted (or none).

4.1 Interleavings

The interleaving of two CFLs is not reliably a CFL. $a^n b^n$ and $c^m d^m$ are both CFL. If their interleaving were CFL then we could intersect it with the regular language $a^* c^* b^* d^*$ to get $a^n c^m b^n d^m$ which (being the intersection of a regular language with a CFL) would be a CFL, but it ain't.

Worth pointing out that the operation $I(L, M)$ of interleaving is associative and commutative. Also the operation $L \mapsto I(L, L)$ is idempotent... oops, or is it?

4.2 A thought about regular and context-free languages

I think this is the question:

Suppose L is a context-free language that is not regular. Can you always find a finite sequence of operations (which may depend on L) each of which preserves regularity (eg complementation, that sort of thing) so that when you apply them to L you reach a language that is not context-free?

Andy[8],

I am running this past you, beco's when i think of the intersection of the class of people who understand coinduction and the class of people who understand context-free languages, your name comes up. I hope you don't mind!

Every regular language is context free, but not every context-free language is regular. This is because the class of regular languages over a fixed alphabet is closed under complementation and intersection whereas the class of CF languages is not. Does this leave open the possibility that there is a coinductive definition of the set of regular languages over an alphabet? It seems to me that would look something like this.

We have a countable alphabet Σ and the set of words over it, which we call Σ^* as usual. $\mathcal{P}(\Sigma^*)$ is the set of languages over Σ, tho' of course we are interested only in those languages that are over a finite subset of σ. The set of functions $\Sigma \to \Sigma$ acts on $\mathcal{P}(\Sigma^*)$ in an obvious way: let us call this *alphabetic variance*. (e.g., the transposition (a, b) turns the language $a^* b^*$ into $b^* a^*$.) Then the set of regular languages over Σ is closed under the usual things (Kleene

[8]Prof Andy Pitts

closure, juxtaposition/concatenation ...) plus of course alphabetic variance. The conjecture will now be something like

> *The class* **REG** *of regular languages over* Σ *is the largest subset of the class* **CFL** *of context-free languages over* Σ *that contains the singleton languages, the empty language and is closed under alphabetic variance, Kleene closure, concatenation, intersection and complement.*

Do we need to include complementation here? We get it for nothing when we define **REG** as the smallest set containing ... and closed under .... Closure under complementation is an *admissible rule* in the jargon of logicians.

And that old fact that i've never really paid any attention to until today—that the intersection of a **REG** and a **CFL** is a **CFL**. Might that come to life here..?

If the conjecture is correct it would mean that if L_1 and L_2 are two languages in **CFL** $\setminus$ **REG** then, if we take **REG** $\cup\{L_1\}$ and form the closure under Kleene closure, union, intersection, juxtaposition and alphabetic variance (and complementation?) then we find that it contains L_2.

That sounds very strong!

Do you know anything about this?

later

27/xi/18. Donald Hobson says that there isn't enuff information in 0^n1^n to be able to parlay that into computing the complement of $\{ww : w \in \Sigma^*\}$. I think his point is that each word of 0^n1^n contains only $\log n$ bits of information and that is not enough. I bet he's right, but i'm not sure how to turn this into a rigorous argument.

4.3 Products of PDAs?

We prove that the intersection of two regular languages is regular by considering the product of two DFA's. The product of two DFAs is a DFA. Similarly NFAs. So can we prove that the intersection of two context-free languages is context-free by considering the product of two PDA's? Evidently not: we can find two CFLs whose intersection is not CF. The conclusion is that there is no robust notion of a product of two PDAs ... the attempt to create a product results in something with two stacks. But perhaps there is a good notion of a product of a PDA with a NFA. After all, the intersection of a CF language with a regular language is CF. And an NFA (and *a fortiori* a DFA) is a degenerate PDA... with an inactive stack.

The intersection of a regular language and a CFL is recognised by an NFA and a PDA running in parallel. That complex makes good sense beco's PDAs—like NFAs—(and unlike Turing or register machines) have this nice feature that they don't run for arbitrary periods after reading a character. They do one thing and then come back for more. Since they (like NFAs) are *clocked* one can describe the simultaneous running of two NFAs in parallel harness as the running

of a single machine—an NFA. Presumably one can describe the simultaneous running of an NFA and a PDA in parallel harness as the running of a *single* machine of some kind ... presumably a PDA. But PDAs are nondeterministic. What happens if you do the power set construction to a (nondeterministic) PDA? Does that even make *sense*?

Is the intersection of two CFLs nice in any way? Perhaps it's recognised by a machine with two unlinked stacks ("chinese walls").

Apparently every context-free language over a singleton alphabet is regular.

If L obeys the pumping lemma must its complement do so as well?

$a^n b^n c^n$ is the intersection of two CFLs and can be recognised by a machine with two stacks. But a machine with two stacks can recognise also $a^n b^n c^n d^n$ and even $a^n b^n c^n d^n e^n$. It would be nice to find a way of adding noninteracting stacks so that one needed three stacks to recognise $a^n b^n c^n d^n$ and four stacks to recognise $a^n b^n c^n d^n e^n$.

We really need the notion of a disjoint union of two PDAs. Think about the PDA that recognises $\{a^i b^j c^k : i = j \vee i = k\}$. After pushing all the as onto the stack you have to guess whether you should be counting bs or counting cs. But then (as Tom Carey says in a supervision) you do the two things simultaneously, so really you make two copies of the machine and run them in parallel. Now a disjoint union of two PDAs is a gadget that has two stacks that don't (indeed *can't*) communicate. A useful idea perhaps.

Have to be careful how you formulate the thought that the set of regular languages over an alphabet is the largest subset of the set of context-free languages closed under *inter alia* complementation and homorphism. Have to assert it for each alphabet separately.

Write up relation between productions and transitions in DFAs ...

I think each nonterminal corresponds to a state, and each production corresponds to a transition from one state to another by means of an input character. This should give us a steer on how to connect CFGs with PDAs.

4.4 Re-use of variables

The language of first-order logic is context-free. If you "improve" it by forbidding re-use of variables it ceases to be context free. Down the back of the sofa i found this gem from an unidentified student of mine. I have edited and abbreviated it slightly. I wish i knew who this student was ...

Question

Is the language of first-order logic context-free? What if we forbid a quantifier's bound variable from appearing outside its scope? (For example, $((\forall x)p(x) \Rightarrow (\forall y)q(y))$ is OK, but $((\forall x)p(x) \Rightarrow (\forall x)q(x))$ and $((x = 3) \Rightarrow (\forall x)p(x))$ are not.)

Answer

It is context-free. Suppose we have some function symbols $\Omega = \{f_1, \ldots, f_n\}$ and predicates (relation symbols) $\Pi = \{\pi_1, \ldots, \pi_m\}$ with arities $\alpha(f_i)$ and $\alpha(\pi_i)$. The definition of the language $\mathcal{L}$ of first-order logic is almost a description of a CFG as-is, so there is not much to do except rewrite the definition in a more formal syntax.

Our alphabet of terminals Σ (not to be confused with the signature Σ!) is

$$\Sigma = \{x, ', =, \Rightarrow, (,), \bot, \forall\} \cup \Omega \cup \Pi.$$

Now we just define nonterminals P (representing primes), V (variables), T (terms), A (atomic formulae), and the start symbol F (formulae), together with the following productions:

$$\begin{array}{lcl}
P & \longrightarrow & \epsilon \mid {}'P \\
V & \longrightarrow & xP \\
T & \longrightarrow & V \mid f_1 \underbrace{TT\ldots T}_{\alpha(f_1) \text{ times}} \mid \ldots \mid f_n \underbrace{TT\ldots T}_{\alpha(f_n) \text{ times}} \\
A & \longrightarrow & \bot \mid T = T \mid \pi_1 \underbrace{TT\ldots T}_{\alpha(\pi_1) \text{ times}} \mid \ldots \mid \pi_m \underbrace{TT\ldots T}_{\alpha(\pi_m) \text{ times}} \\
F & \longrightarrow & A \mid (F \Rightarrow F) \mid (\forall V)F
\end{array}$$

So $\mathcal{L}$ is a CFL. However, it is not a regular language, because of the matching brackets, as we by now understand.

What if we implement the restriction on bound variables? This language $\mathcal{L}'$ is no longer a CFL; we can show this by supposing it were, and applying the CFL pumping lemma to a sufficiently long word of the form $w = tuvyz =$

$$(\ldots(((\forall x)\, \bot \Rightarrow (\forall x')\, \bot) \Rightarrow (\forall x'')\, \bot) \Rightarrow (\forall x''')\, \bot) \Rightarrow \ldots).$$

Suppose we had $tu^k vy^k z \in \mathcal{L}'$ for all k. If uy contains $\forall$, we're dead, as pumping will only produce legal words if uy contains some $(\forall x'' \ldots')$, contradicting the non-reuse of variables. Similarly, we can't have uy containing x as these only show up in quantifiers. So uy contains only $(,), ', \bot, \Rightarrow$. Laborious case analysis shows that any other possibility generates illegal strings on pumping, except if $uy ='' \ldots'$, in which case tvz has a reused variable. Note that we can't have e.g. $u = ($ and $y =)$; even if $tu^k vy^k z$ is bracket-matched and human-parseable for $k \neq 1$, it does not lie in $\mathcal{L}'$. Indeed, for bracket consistency, uy needs: number of '(' = number of ')' = number of '$\Rightarrow$'. This simplifies the checking substantially.

5 Computable Functions

A union of a semidecidable family of semidecidable sets is semidecidable. Obvious why it's true and obvious how to prove it. And of course an arbitrary union of semidecidable sets is *not* reliably semidecidable. Obvious counterexample: let the nth semidecidable set be the first n natural numbers not in the halting set. Each of these sets is semidecidable but for silly reasons (every finite set is decidable) and this fact is crucial for the trick of making the union not semidecidable. Must find something intelligent to say about this.

Realistic machines wot actual people write actual code for have registers whose contents can be seen as `booleans` or as `natural numbers` *ad lib.* This equivocation on datatypes is actually quite important in real-life assembly-language programming. It's probably worth making a fuss about the fact that the binary boolean operations on bit-strings all correspond to primitive recursive operations on the corresponding numbers. The proof probably looks quite nasty, but it can do you no harm to think about why this should be true and how you might prove it ... a simple example: explain bitwise `and` as a primitive recursive operation on the corresponding natural numbers.

Consider the equivalence relation on natural numbers of encoding functions with the same graph. Can the quotient have a semidecidable transversal?

If you apply one church numeral to another you get exponentiation, but which way round? Is $n\ m$ equal to n^m or m^n. It's easy! $n\ 1\ f$ is obviously f, so $n\ m$ must be m^n.

5.1 A conversation with two of my Queens' 1B CS students

Suppose we have an oracle $\mathcal{O}$ for the `HALT`ing problem. Let **TOT** be the set of numeric codes for total functions. We persuaded ourselves that we can solve the membership question for **TOT** as long as we have access to such an oracle $\mathcal{O}$ for the `HALT`ing problem. I said at the time that there must be a mistake. I have now found the mistake!

Let $\{n\}$ be the function encoded by the number n, as usual.

Suppose we have an oracle $\mathcal{O}$ for the `HALT`ing set. (Here the `HALT`ing set is the set $\{\langle p, i\rangle : p$ `HALT`s on $i\}$.)

We define a function $\mathfrak{T}$ ('T' for Total) such that $\mathfrak{T}(n)$ performs as follows: run $\{n\}$ on the increasing stream of naturals, asking $\mathcal{O}$ at each input i whether or not $\{n\}$ `HALT`s on input i. If $\mathcal{O}$ tells us that $\{n\}$ `HALT`s on input i then we procede to $i+1$; if $\mathcal{O}$ says 'No' we `HALT`. Thus $\mathfrak{T}(n)$ returns the least i s.t. $\{n\}(i) \uparrow$ if there is one and loops forever if there isn't. Evidently $\{n\}$ is a total function iff $\mathfrak{T}(n) \downarrow$ ... and this is something we can ask $\mathcal{O}$. The way to check whether or not $\{n\}$ is total is to ask $\mathcal{O}$ whether or not $\mathfrak{T}(n) \downarrow$.

So! We have reduced **TOT** (the set of indices for total computable functions) to the `HALT`ing set!!

Except we haven't! The final querying of whether or not $\mathfrak{T}(n)\downarrow$ cannot be done by $\mathcal{O}$, because $\mathfrak{T}$ is not a computable function! We need something a lot more powerful than $\mathcal{O}$. However we do get *something*. What this tells us is that we can compute **TOT** if we have an oracle not for the ordinary `HALT`ing problem but the `HALT`ing problem *for functions that are allowed to call an oracle for the* `HALT`*ing problem* (for computable functions).

5.1.1 Turing Degrees

There are these things called *Turing degrees*. They are equivalence classes of functions. If i can compute f given an oracle for g, and i can compute g given an oracle for f, then f and g have the same degree. The unsolvability of the `HALT`ing problem really shows that, for any degree d, the degree of functions that-are-allowed-to-call-an-oracle-for-a-function-in-d is above d. This degree is notated d'. (This notation is standard) The degree of computable functions is 0; the degree of the `HALT`ing set is $0'$. So it seems that the degree of **TOT** is $\leq 0''$. That, at least, is what i understand Martin Hyland to say.

Can we prove a converse? Given an oracle for **TOT** can we solve ¿$\{n\}^{\mathcal{O}}(m)\downarrow$? where $\mathcal{O}$ is an oracle for the `halt`ing problem?

5.2 Inverting Partial Computable Functions

The process of mathematising the concept of computable functions proceeds—at least initially—by erecting a recursive datatype of function declarations: find some functions that are uncontroversially computable, and close under operations that uncontroversially preserve computablity, and keep fingers crossed that every function one might consider computable gets swept up. One helpful thought is that the inverse of a computable function ought to be computable. Clearly one can invert total computable functions. If $f : \mathbb{N} \to \mathbb{N}$ is total computable then $f^{-1}(n) =$ least k s.t. $f(k) = n$ if there is one, and undefined otherwise. This f^{-1} is partial computable. It might not be total but it most assuredly is computable.

This simple-minded construction relies on f being total. Presumably one cannot reliably invert partial computable functions. There is an obvious strategy for finding a k s.t. $f(k) = n$ if there is one: zigzag thru' all inputs until you get the output n. But that is nasty, hacky and exquisitely sensitive to the way in which one zigzags; the answer you get is certainly not guaranteed to be the smallest.

Let the **hard inverse** of a partial computable function $f : \mathbb{N} \to \mathbb{N}$ be the function $\lambda n.$ least m s.t. $f(m) = n$ if there is one, `fail` otherwise. There seems to be no reason why this hard inverse of f should be computable merely on the basis that f is. Indeed it is natural to harbour a very strong suspicion that there are computable partial functions whose hard inverses are *not* computable. So strong in fact that I for one for years never felt the need to go to the effort of finding one. It was only when explaining to my students a history of the project to mathematise the idea of computable function that it occurred to me that i

really should exhibit such a computable partial function, in order to make the point that one can't just simple-mindedly close under (hard) inverse. Here is one, supplied by Michael Beeson.

Define the partial function f by

$f(2x) \quad = \texttt{if } \{x\}(x)\downarrow \texttt{ then } x \texttt{ else fail};$
$f(2x+1) = x.$

Then f is partial recursive, and surjective. f^{-1}"$\{x\}$ is either $\{2x+1\}$ or $\{2x, 2x+1\}$. The hard inverse of f, on being given x, returns either $2x+1$ (which it does if $\{x\}(x)\uparrow$) or $2x$ (if $\{x\}(x)\downarrow$). Since this solves the diagonal HALTing problem for us we conclude that the hard inverse of f is not computable.

5.3 This should be an exam question

Suppose $\leq_1$ and $\leq_2$ are two wellorderings of $\mathbb{N}$, both with decidable graphs and both of order type ω. Clearly they are isomorphic, and the isomorphism is unique, but is it computable? There is an obvious algorithm for finding it, by a series of finite approximations, as follows. We line up $\langle \mathbb{N}, \leq_1 \rangle$ on the left pointing skywards and $\langle \mathbb{N}, \leq_2 \rangle$ on the right pointing skywards. At stage n we pair off the numbers $0-n$ on the L with $0-n$ on the R in an order-preserving way. Initially we pair 0 on the L with 0 on the R. Then we add the 1's. If $1 \geq_1 0 \wedge 1 \geq_2 0$ or $0 \geq_1 1 \wedge 0 \geq_2 1$ then we just pair off the two 1s, but if not then we have to re-pair. Thus we have to adjust from time-to-time but *each number has to be re-paired only finitely often*. Specifically a number n has to be re-paired only when we find $m > n$ s.t. $m <_1 n$ or $m <_2 n$, and there are only finitely many such m. Simply from the information that the two orders are decidable we cannot find a bound on how often a number gets re-paired so we cannot know when it has settled down. The best we can do—it seems—is to get the isomorphism to be $\exists\forall$. However, if the two orders *nearly* agree then the settling down will take place quite quickly, and we might be able to bound the time it takes for the pairing for a given number to settle down. So the bijection *might* be computable. Here's a thought. Think about the WQO $\langle \mathbb{N}, \leq_1 \cap \leq_2 \rangle$. So if the rank of the tree of bad sequences in $\langle \mathbb{N}, \leq_1 \cap \leq_2 \rangle$ (its *maximal order type*) is small enough the bijection will be computable?

Does this give a metric on wellorderings of $\mathbb{N}$? The distance between two wellorderings is the maximal order type of the intersection? Does this obey the triangle inequality? What notion of addition do we have? Hessenberg? It's an odd sort of metric beco's its values are countable ordinals not reals.

5.3.1 Over dinner at the Boffafest: discriminators

Suppose $f : \mathbb{N} \to \mathbb{N}$ total computable with no odd cycles. Then a function $d : \mathbb{N} \to \{0,1\}$ with $(\forall n \in \mathbb{N})(d(f(n) = 1 - d(n))$ is a *discriminator* for f. Clearly every total computable f with no odd cycles has a discriminator.

Conjecture: there is a recursive f with no recursive discriminator.

Thinking about how λ-calculus crops up in Computation Theory ...

Suppose i am trying to many-one reduce $\mathcal{K}$ to **TOT**. I want to know if $p(i)\downarrow$. I cook up a function that, on being given input n, throws it away and runs p on i. Sounds like the K combinator to me! [This is why i still supervise]

Another thought on many-one reduction

Many-one reducibility says "There is a computable total function $f : \mathbb{N} \to \mathbb{N}$ with the property that if $x \in A$ then $f(x) \in B$.' The idea is that such an f, used in conjunction for a gatekeeper for B (a function that says 'yes' to members of B but otherwise says nothing) gives you a gatekeeper for A that is no more uncomputable than the gatekeeper for B. The existence of such a computable f means that if you can recognise members of B then you can recognise members of A.

f has to be computable but it doesn't have to be defined on the *whole* of $\mathbb{N}$. If the candidate c is a member of A, then $f(c)$ has to be defined and be a member of B. So we must have $A \subseteq \mathrm{dom}(f)$. Except in trivial cases A will not be semidecidable, whereas $\mathrm{dom}(f)$ will always be semidecidable, so the inclusion will be proper: there will be candidates c on which f halts but $f(c) \notin B$ so you might as well have had $f(c)\uparrow$. Either way c does not get authenticated as a member of A.

However there is something slightly nonconstructive about this. Note that $\mathrm{dom}(f)$ might have to be a *proper* superset of A; f has to come up with an answer for the input c without being told whether or not $c \in A$. It is true that we need $f(c)$ only if $c \in A$ but if $\mathrm{dom}(f) = A$ precisely then f isn't computable ... so, for at least some c, f has to give an answer which is then not used.

Contrast

$$(\forall n)(\exists m)(n \in A \to m \in B) \text{ and } (\forall n)(n \in A \to (\exists m)(m \in B))$$

The m is of course $f(n)$

They are not constructively equivalent, the first being stronger than the second. The core fact here is that $p \to (\exists x)F(x)$ and $(\exists x)(p \to F(x))$ are not constructively equivalent—the second is stronger than the first. Given the parallel between $\exists$ and $\vee$ we can see the propositional version:

$$A \to (B \vee C). \to .(A \to B) \vee (A \to C)$$

is not constructively correct.

$$(\forall x)(A(x) \to (B(x) \vee C(x)). \to .(\forall x)(A(x) \to B(x)) \vee (\forall x)(A(x) \to C(x))$$

is not even classically correct.

Another point i find myself having to make about many-one reduction .... Suppose you are trying to reduce A to B where A and B are classes of functions. I have a gatekeeper/gremlin for B. I have to turn a candidate for membership

in A into a candidate for membership in B. This is done by some computable function . . . which i have to be able to run. However the *output* of this reducing function—even tho' it may itself be a function—isn't one i have to worry about *running*. The gremlin might run it. Or it might not. The gremlin might just lay its hands on it and feel the vibrations. It doesn't matter. What the gremlin does is of no interest to us.

6 Supervision Notes on the Part II Automata and Formal Languages Course

Worth emphasising to beginners that the way in which we (or at least many of us, i for one) think of—*visualise*—the natural numbers, as a snake wandering through space ... has the potential to seriously mislead. The temptation is to think of a subset $X \subseteq \mathbb{N}$ as the snake with some of its nodes lit up. This is OK if X is decidable but not otherwise. The snake makes you think that $\mathbb{N}$ and all its subsets are random access devices, or at least (if you don't like that—and you mightn't) that it's a *sequential* access device. You can access members of a decidable subset $X \subseteq \mathbb{N}$ by using the enumeration in increasing order. However, if X is merely *semidecidable* then it's still a sequential access device all right, but the order in which you get access to the elements is not in order of magnitude! Also if a set is merely semidecidable then its complement is *not* (even) a sequential access device.

6.1 Sheet 1

Before i get to discussions of individual questions i want to recap what students should have in their knapsack by the end of the first supervision. Thus the following paragraphs will contain the points that I made to you in supervision. Well, *tried* to! I didn't manage to make every point to everyone.

There have been two big theorems so far. You do not need at this stage to be able to prove them but you do need to be able to state them correctly and understand them.

One of them is the equivalence of the two conceptualisation of computable functions: the two concepts that Dr Button calls 'partial computable' and 'partial recursive' or what i prefer to call the syntactic and the semantic concepts. The proof i favour relies on bringing out into the open Kleene's T function.

The other is Rice's theorem. The proof is hard work—i was initially quite taken aback at the thought that this result could be shoehorned into a C course but it seems to have worked! It's not realistic to expect you to have mastered the proof this early in the piece but i want you to at least understand what it says, so you can see why the teaching committee put it in. You need the ideas of a *function-in-intension* and of *function-in-extension*.

A few thoughts on the definition of "recursive" and "recursively enumerable" sets. When i prompted you for definitions of these terms you were able to come up with them. However, in asense it's not the literal text of the definition that is the point. It's good practice in mathematics—when you are given a definition of something—to ask yourself "Why am i being given this definition? What is it *for*?" The point in this case is that these definitions are an attempt to use the novel sexy concept of computable function to capture pre-theoretic intuitions of decidable and semidecidable sets. When you come (as you very soon will) to the concept of *many-one reducibility* it is essential to ask yourself "What is this definition intended to capture?"

Now for the sheet

Q1 and Q2

The point of the first two questions is to get your hands dirty writing register-machine code. You don't want to write register machine code for a living; i did for a while, and—altho' it was a character-forming experience that i was glad to have had—I'm glad i don't have to repeat it. The point here is that any function $\mathbb{N} \to \mathbb{N}$ that can be computed *at all* can be computed by a register machine, and it's good for you to get a sense of what this fact *feels like* under the hands, as it were ... so you know it viscerally and not just intellectually.

Q3 and Q4

Next he wants you to write some function declarations. This is—analogously—to give you a feel for the fact that any function $\mathbb{N} \to \mathbb{N}$ that can be computed at all can be "declared" by means of the basic functions, composition, primitive recursion and minimisation. It's not obvious why this should be so and you'll have to wait to see a proof of it, but you can get a taste here and now.

(By the time you are reading this you will have seen a proof in lectures).

Notice that all the functions in Q3 are in fact *primitive* recursive.

Q5

The point is to make you think about *zigzagging*, a strategy which will be useful in the weeks to come. Incidentally don't get into the habit of calling it *a diagonal process*. I don't know where Dr Chiodo got this unfortunate terminology from, since i've never encountered it anywhere else. 'Diagonal' refers to constructions such as Cantor's proof that the reals are uncountable, and you should not use the word to denote any other type of construction. There is a genuine diagonal construction in Q9 below.

Q6 and Q8

These two questions have similar character. The facts you are invited to prove are basic uncomplicated facts that help you to get your thoughts straight. A point about question 8: It's obvious how to compute the inverse of a computable permutation of $\mathbb{N}$. How do you find $f^{-1}(17)$? Easy: you compute $f(0)$, $f(1)$ ... until you get the answer 17. But this is clearly an invocation of minimisation. This floats the possibility that there might be a primitive recursive permutation of $\mathbb{N}$ whose inverse is not primitive recursive. And in fact there are such primitive recursive permutations. They are all of infinite order. (Why?) Here be dragons[9].

[9] But no dragon icon; i looked for LaTeX dragon icons but i couldn't find one. If you know of one do please tell me.

Q7

Question 7 is making the point that the kind of computability that we are considering here—finite, discrete, deterministic but with no finite bound on the resources (time or space) used—is actually rather unnatural. It would seem to be more natural, more *realistic*, to study computation with bounded resources, seeing as how we are finite beings with bounded resources. However the study of computation *without* restraints (which is the subject of this part of the course) is much better behaved than the study of computation within restraints. For example the class of functions that are computable-in-principle-no-quibbling-about-resources is clearly closed under composition, whereas the class of functions computable under restraint (whatever your notion of restraint) might well not be. It's worth recording in this setting that there are a million and one different concepts of computation-with-bounded-resources (in contrast to the concept of finite deterministic discrete unbounded computation where all the concepts turn out to be the same) and—worse—it seems to be hard (*puzzlingly* hard) to tell when two of them are the same. As i say, here be dragons; lots of 'em.

Q9

This question makes two points. One is that it's pretty obvious how to obtain (from m, a code for a machine) the code for the machine that computes $m(x)+1$. If we think about this process it becomes clear that it is a computable function $\mathbb{N}\to\mathbb{N}$. So it's a point about Church's thesis. However it is also a sleeper for the diagonal arguments we will encounter later one. Rehearse diagonal arguments from *Numbers and Sets* (uncountably many reals...)

Q10*

> "Show that there is an r.e. set E such that, for every $n \in E$, $f_{n,1}$ is primitive recursive and, moreover, every primitive recursive function on 1 variable occurs as $f_{n,1}$ for some $n \in E$.

This is quite subtle. Although it is not clear *in general* whether a function (given somehow) admits a primitive recursive declaration, it *is* always clear whether or not a *given function declaration* (which is a piece of syntax) matches the template of primitive recursion.

I am going to give a fairly detailed discussion of this because i tripped myself up preparing a model answer, and convinced myself that it couldn't be done. And if i can get it wrong (and i'm supposed to know this stuff) then you can probably get it wrong too.

It's probably clear to you, Dear Reader—since you went to the lectures and i didn't—that $f_{n,1}$ is the function computed by the nth **machine** and not the nth **function declaration**.

The correct way to proceed is to first think of the function that enumerates all the primitive recursive function declarations. This is obviously a decidable set, and can be enumerated by a total computable function. Every time this function gives an output you proceed to turn its output (which is a primitive recursive function declaration) into a description of a register machine that computes that function. This is straightforward, and we needed this ability when we proved that every partial recursive function is partial computable. The set of codes of these descriptions is now the set E that we desired.

But i started doing it the wrong way, and convinced myself that the question-setter had got it wrong. Suppose you start by enumerating all the register machines, and then hope to obtain E by retaining only those numbers that are codes for register machines that compute functions that admit a primitive recursive function declaration. The problem is that there is no algorithm that will look at a code for a register machine and tell its handler whether or not the function computed by that machine can be declared as a primrec function. It's not even going to be r.e. beco's—even if you can *guess* a primrec declaration for the function computed by the register machine in your hand—you still have to prove that the function thus declared is, in fact, computed by that machine. And there is no reason to suppose that can be done in finite time.

If—like me—you haven't been to the lectures then it is possible to misread this question (as i have been doing) by thinking that $f_{n,1}$ is the nth **function declaration** in a listing of the function declarations. This is a misreading, but it enables one to make the point that the set E of function declarations that are primitive recursive is a a straight-up decidable set, co's you can see at a glance whether a function declaration contains any gadgetry that isn't primitive recursive, and you don't need to go scouring the universe for an equivalent function declaration with a fancy property.

It's worth noting that the two sets-of-codes have different properties: the second one is obviously decidable, whereas the first is *prima facie* merely semidecidable. I can't see offhand a proof that Dr Button's set E is not decidable, but i bet it isn't.

Q11

It's not immediately obvious how to show that Cantor's pairing function is a bijection between $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$ but it becomes easier once you spot that the first term in the definiens is the $x + y$th triangular number, and that the y that you are adding to it doesn't take you as far as the next triangular number. This enables you to show that the pairing function is surjective, as follows. Think of a number, any number—don't tell me what it is—and look for the largest triangular number $\leq$ it. That gives you $x + y$ (since that triangular number is the $x + y$th triangular number). And the difference between that triangular number and the number you first thought of is y. So you have now decoded your number as an ordered pair.

One of my students proved, by dint of some clever rearragements, that (as long as $x > 0$) $\langle x, y\rangle + 1 = \langle x - 1, y + 1\rangle$. You have to do some work when

$x = 0$. But i think my proof is better.

Typing and Computation

Abstract Computability is not normally taught in a strongly typed context, a context where one worries about what types things are. (You may have noticed that nobody has mentioned typing in the course you have just sat thru', whether you are a 1B compsci having done Computation Theory, or a Part II mathmo doing Automata and Formal Languages.)

Natural numbers, as you meet them at school, or in your 1A year, either in Comp Sci DM or mathmo Numbers-and-Sets "support" (as they say) addition and multiplication and exponentiation. They do not "support" pairing and unpairing: a natural number is not a pair of natural numbers. Nor do they "support" application: you cannot apply a natural number to a natural number. The really disconcerting thing about Computation theory is the abstract data type of natural number (according to Comp Th) *does* support application and pairing-and-unpairing. And one of the ways in which it disconcerts it that this violates a typing intuition *which is never spelt out*. Mathematics is actually quite strongly typed, but thisfact is never brought out into the open. And because it is never brought out into the open one is not aware of when it is being violated. One is disconcerted by the violation of course, but one is not aware of the fact that it is that violation that gives one the feeling of disconcertment that one experiences.

There is a reason for this. Think about the proof of the unsolvability of the `halt`ing problem. This depends—*crucially*—on the fact that your data objects are simultaneously *both* machines (or gnumbers of machine) *and* inputs to those machines. You have to apply a machine to its own navel, i mean gnumber. It's probably worth thinking a bit about what the abstract data types are of the objects in play here. Are there two data types, numbers and machines? Or is there a single ADT of dual aspect? I've long had the idea that a strongly typed theory of computable functions would render the diagonal arguments unworkable. There's nothing wrong with the data object being simultaneously a program and a natural, *as long as you don't allow it to be both at the same time*. There's this weirdo gadget called "Linear Logic" which is an example from a suite of gadgets called *Resource Logic*s, which consider reasoning in situations where you have constraints on what you can have simultaneously in your head. For example in some of these logics you can use each assumption precisely *once*. Thus you can't prove $(A \to B) \to (A \to (A \to B))$!

The point being made here in Q11 is that Cantor's pairing function—and its inverses—are so smooth that for many purposes you can think of tuples of natural numbers just as being natural numbers themselves. So—if you want—you can think of a function of k variables as a function of *one* variable. If f is a function of k variables then in some sense it is the same function as the *unary* function g that accepts a *single* argument, which it thinks of as a k-tuple, decomposes it accordingly, and feeds the k components to f. This possibility of thinking of polyadic functions as monadic functions is central to λ-calculus,

and λ-calculus is something you definitely want to look at if you want to take the material in this course further (tho' it's not lectured and is not examinable)

This is additional to the fact that you can think of natural numbers as machines (since you can set up—once for all—a coding of machines as natural numbers). This Janus-faced nature of natural numbers is essential to many proofs in Computation Theory, such as Rice's theorem and many others. Natural numbers can even be taken to be the "booleans" (truth-values) `true` and `false` (or $\top$ and $\bot$)—0 and 1 often being reserved for this purpose. The connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$ etc then become computable functions ... in fact *primitive recursive* functions.

Numbers can also be *times*, if we think of our processes as clocked. See volcanoes below.

The fact that numbers can code strings becomes important in Q11 on the next sheet.

6.2 Sheet 2

For the first part.... There are various questions that can be answered swiftly using Rice's theorem. I think it's clear from the overall context that he does not want you to use Rice's theorem, but instead to get your hands dirty doing actual problem reductions.

Question 1

Perhaps worth commenting that the concept in play here is *disjoint union*. The nastiness of the disjoint union of two sets (of naturals) is the precise least upper bound of the two nastinesses.

When i mentioned this to one of my students he immediately asked me the parallel question about cartesian products[10], and i had to confess to my shame that i had never thought about it. *You* might like to think about it: if A and B are decidable (semidecidable) must $A \times B$ be decidable (semidecidable)? And *vice versa*...?? In particular can you many-one reduce $A \sqcup B$ to $A \times B$ or *vice versa*? I'm going to put the answer at the end of this file, in section 9.

Another student asked me: OK any two sets have a lub wrt nastiness; do they have a glb? I didn't know that either! There's certainly no obvious candidate for a glb. It turns out these glbs do not always exist. I think of this as an illustration of something one keeps finding in recursion theory, namely that if there isn't an obvous way of doing the thing you want, then there is no way of doing it at all.

Question 2

If you are alert you will spot that all three parts can be dealt with by the one trick: find a set of the flavour you want: r.e., recursive, whatever, and consider the set of singletons of its members. Every singleton is recursive, so every set—of whatever flavour—can be expressed as a union of countably many recursive sets.

[Actually, one smartie-pants—who has been given name suppression—noticed that the empty set is a recursive set that is a union of a countably infinite family of recursive sets!]

Some of you—much to your credit—think there must be something wrong with this question, that it is *cheating*, and you suspect that you must have misunderstood something. It isn't cheating, and you haven't misunderstood anything. Dr Chiodo (for it is he who was the Evil Genius who devised this question) is making the point that you need to put restrictions on the **index set** (of the family of recursive sets) whose sumset you are forming if you are to get anything sensible. In this connection you might like to look at question 13 on Dr Chiodo's version of this sheet from earlier years (still on the dpmms home page) this sheet ... which is not as scary as its asterisk might lead you to suppose.

[10] He'd been thinking about category theory.

I think the question should be tweaked. A question that wrong-foots the good students but not the weak ones isn't doing its job.

Question 3

I'm trying to think how to make this obvious. You want the intersection of this family to be a set that is not r.e. What is your favourite example of a set that is not r.e.? Yes, it's the complement of the `HALT`ing set. So you want the complement of the `HALT`ing set to consist of numbers that have passed infinitely many tests. If $\{i\}(i)$ never halts, what tests has i passed? When you put it like that it becomes obvious: the nth test is: "Is $\{i\}(i)$ still running after n steps?"

Hmmmm... When i wrote that i was assuming that the 'W' in 'W_n' was just a random letter but it now occurs to me that he might have meant *literally* 'W_n' as in "the domain of definition of the nth function". But that's doable. We just have to let our index set consist of some of the $j \in \mathbb{N}$ s.t. W_j is a set of the form $\{i : \{i\}(i)$ `is still running after` n `steps`$\}$.

To be precise, we obtain the index set $I \subseteq \mathbb{N}$ as follows. For each n, obtain the set $A_n = \{i : \{i\}(i)$ `is still running after` n `steps`$\}$ and let j be the index of a function that halts on any member of A_n and on nothing else. The process that obtains j from n is clearly finite and deterministic etc etc so its output is an r.e. set by Church's thesis. So the set of all such j is r.e. and is the $I \subseteq \mathbb{N}$ that we want.

Question 4

It's obviously at least *semi*decidable (r.e.) so how do we show that it is not actually decidable (recursive)? One thing you can do is wheel out Rice's theorem to say that the complement cannot be semidecidable (r.e.). This certainly works, but the result is that you have evaded the purpose of the exercise, which was to force you to get your hands dirty doing a bit of problem reduction. Why is the complement of this set not semidecidable? Beco's, if it were, the complement of the `HALT`ing set would be semidecidable too, and it ain't. Your job is to show how to exploit a gremlin that can detect members of the complement of this set and put it to work recognising non-members of the `HALT`ing set.

I noticed that lots of you fell into a trap i can only call an error of *attachment*. You spotted that you had to cook up a function that halted on at least six inputs iff some given candidate for *non*-membership of the `HALT`ing set was, indeed, not a member of the `HALT`ing set. However you got distracted by the number 6. You wanted a function that did something different to the smallest 5 inputs from what it did to others. That isn't neccessary. Given n, you want the function that ignores its input and just runs n on n.

Question 5

A riff on Problem Reduction

The way to sell these reduction problems to students is to say: suppose i have a gremlin that knows the characteristic function of X. Can i use it to calculate the characteristic function for Y? I keep it in inhumane and degrading conditions in the back of my shop and never let it out or see daylight or let anyone know it's there. (It's been trafficked from a village in Gremlinland). I put a brass plate on my door saying i can calculate the characteristic function for Y. The challenge for the student is: *What do i ask the gremlin?*

One thing that has struck me in supervisions devoted to this sheet (and problem reduction in particular) is the tendency of students to take a reduce-this-problem-to-that-problem challenge too literally, in that they are actually looking for a register-machine computable function, or even—heavan forfend!—a register machine program. I think this is beco's your point of departure is the *definition* of many-one reducibility rather than the intuition behind it. The way to reduce A to B (or is it the other way round? i always forget) is to ask yourself: if i had a machine that recognises As can i use it to recognise Bs? You then apply Church's thesis.

> (Narrate it by making use of the old joke about "thereby reducing it to a problem already solved")

(b) (first part)

The `HALT`ing problem (well, one version of it) is: Does program p `HALT` on i? To use an oracle for **TOT** to solve the $\{p\}(i)$ instance of the `HALT`ing problem you consider the function that throws away its input and then runs p on i.

(b) (second part)

To use an oracle that can tell whether or not a function `HALT`s on infinitely many inputs to solve the `HALT`ing problem you consider the function that, for input n, runs p on i for n steps, returns `YES` if it is still running, and loops forever if not.

Actually that was overkill, wasn't it? You can use the same function as you did in the first part of 5(b)! Quite a number of you spotted this but weren't happy about it . . . how can the one function do both tasks? Well, there may be lots of ways of reducing A to B, and so there's nothing obviously absurd in the idea that a function that reduces A to B might also reduce it to C.

(c) Obviously not. If any of them were, you would be able to solve the `HALT`ing problem—as you have showed. (Reminder: don't just appeal to Rice's theorem).

(d) To use an oracle for **TOT** to determine whether or not a function has an infinite domain you use a volcano. A *volcano*[11] for a function f is an engine that runs a machine for f on all possible inputs in zigzag parallel with itself, so

[11] This is *not* standard terminology.

that you can put it in the corner of your room and watch it emit values of f while you relax with a cuban cigar and a bottle of claret.

(Notice that the volcano function that you test with your gremlin not only has the same *set* of values as your input function, it has the same *multiset* of values.)

Question 6

Well, the empty set is recursive, so for the first part find m s.t. the mth function is everywhere undefined. As Prof Pitts says, *the Register Machine from Hell*, with number 666, never halts. So set $m =: 666$; then every function has a domain-of-definition which is a superset of W_m.

For the second part let m be any code for a total computable function. Then T_m is the set of codes for total functions, and we proved earlier that this is not r.e.

Question 7

Let g be total computable. (Don't be distracted by information about how many arguments it takes). The challenge is to show that if you have a gremlin that can tell when a given function-in-intension f has the same graph as g then you can put that gremlin to work telling you when a function is total.

The difficulty the student experiences here is in accepting the idea that you might want to run a function and then throw away its output. "How on earth" (one wonders) "can it be necessary to perform a computation if you don't need the output?" It's reminiscent of the oddity in question 4 where you have to throw away your input. Good housekeeping seems to require that you shouldn't throw away anything that might be informative. There's a kind of cortical censorship that prevents you from considering this possibility... (Orwell would have called it *crimestop*). But—unfortunately—*here* that is exactly what you have to do. You might need to be told that a computation `HALT`s but that might be *all* you need to know about it. Get used to it, beco's problem reduction is full of tricks like that.

You have a gremlin whose eyes light up when it is shown code for a function that has the same graph as g. I want to know if f is total. We need to find a function that agrees with g iff f is total, co's that's the function we want to show to the gremlin. *What function agrees with g iff f is total?* Obviously

$$\lambda n.\texttt{if } f(n)\downarrow \texttt{ then } g(n) \texttt{ else fail}$$

There is a connection here with

6.3 CS 2017 Part IB Exercise sheet ex 12

"*Let* $\mathbf{I}$ *be the* λ*-term* $\lambda x.x$*. Show that* $n\ \mathbf{I} = \mathbf{I}$ *holds for every Church numeral* n*.*

Now consider $\mathbf{B} = \lambda fgx.gx\ \mathbf{I}\ (f(gx))$

Assuming the fact about normal order reduction mentioned on slide 115, show that if partial functions $f, g \in \mathbb{N} \to \mathbb{N}$ *are represented by closed* λ*-terms* $\mathbf{F}$ *and* $\mathbf{G}$ *respectively, then their composition* $(f \circ g)(x) = f(g(x))$ *is represented by* $\mathbf{B}\ \mathbf{F}\ \mathbf{G}$*"*

The point *there* is that if f is a function that ignores its input and returns something anyway than the naïve composition combinator won't crash if g crashes, when really it should. So you test to see if $g(x)$ crashes and, if it does, you crash too. In detail: $g : \mathbb{N} \to \mathbb{N}$ and $x : \mathbb{N}$. Then $g(x)$ (which is the first thing we do) either crashes or—if it doesn't—it returns a Church numeral ... which it then applies to $\mathbf{I}$, getting $\mathbf{I}$ of course, which it then applies to f, getting f (as we were asked to show in the first part of the question) which we then apply to x.

Robert Hönig writes:

Ok, so I asked Prof Pitts and I think we got to the ground[12] of this.

I misunderstood the definition of normal-order reduction. The correct definition fully reduces M before N in $(\lambda x.M)N$, so, if $\mathbf{G}\ x$ has no β-nf, then $\mathbf{G}\ x\ \mathbf{I}$ has no β-nf.

Here's the relevant extract from his response:

"I agree that there is more to say about why $\mathbf{G}\ x$ having no β-nf implies $\mathbf{G}\ x\ \mathbf{I}\ (\mathbf{F}\ (\mathbf{G}\ x))$ does not have one either, beyond the fact mentioned on Slide 115 of the notes.

In lectures I snuck in an extra slide after 115, giving a syntax-directed inductive definition of normal order reduction, `->_NOR`—see last page of `https://www.cl.cam.ac.uk/teaching/1920/CompTheory/lectures/lecture-10.pdf`.

If you accept that inductive characterisation, then:

> if $\mathbf{G}\ x$ has no β-nf,
>
> then, by the fact on p.115, the sequence of steps of `->_NOR` starting from $\mathbf{G}\ x$ is infinite;
>
> but then by the first rule for generating a step of `->_NOR` out of an application, we get an infinite sequence of steps of `->_NOR` starting from $\mathbf{G}\ x\ \mathbf{I}\ (\mathbf{F}\ (\mathbf{G}\ x))$, and so, by the p.115 fact again, $\mathbf{G}\ x\ \mathbf{I}\ (\mathbf{F}\ (\mathbf{G}\ x))$, has no β-nf."

[RH sez: So the crux lies in the inductive definition of normal-order-reduction. (So my mistake was to initially only think of the prose definition, which is ambiguous.) That inductive definition contained a mistake, which has now been fixed:

I think the inductive definition on slide 116 has a small mistake that makes it non-deterministic: The second application reduction rule,

[12] This is his literal translation from the German: *Grund*. That's the word you use in German for the root of a problem. Always learn from your students!

$(\lambda x.M)M'$ `->_NOR` $M[M'/x]$

should only apply when $(\lambda x.M)$ is in β-nf. (Because otherwise we can do left-most reduction.) Thus, I think it should read

$(\lambda x.N)M'$ `->_NOR` $N[M'/x]$

?]

Question 8(c) (multiples of 3)

Design a DFA that accepts strings from $\{0,1\}^*$ that evaluate to multiples of 3. There is a standard way of reading bit strings as numbers, so you do that, but you do seem to have a choice about how you obtain a string of length $n+1$ from a string of length n: do you append the new character on the right or on the left? To me, it seems obvious that you should append the new character on the right, as the least significant bit. That way the number represented by the string of length $n+1$ is either $2\mathfrak{n}$ or $2\mathfrak{n}+1$ depending on whether you are appending a '0' or a '1'. ($\mathfrak{n}$ is the number represented by the n characters shown to us so far). And you can calculate the residue mod 3 of the number represented by the first $n+1$ characters (which is—as we all agree—what you need to keep track of) just by knowing $\mathfrak{n}$ mod 3. On the other hand if you append on the left instead of the right, you need to know not only $\mathfrak{n}$ mod 3, you also need to know the parity of n. This is beco's if you plonk a '1' on at the front you are adding a power of 2, and it is congruent to 1 or to 2 depending on the parity of the length of the string-so-far!! You don't get a different regular language but you do need six states rather than three.

Either way you get strings that have leading zeroes, but that doesn't seem to matter.

One might have to think about the regular expressions you get for the two machines. I fretted about it for a bit but i now can't see anything to worry about. You get the same regular expression ... *eventually*; Kleene's algorithm will give you regular expression from the two DFAs but they will simplify to the same shortest regular expressions. Don't ask.

Michael He makes a useful contribution here. He says: if i read the strings the wrong way, appending on the left instead of on the right, i still have only three states, it's just that they are more complicated.

What are these three states? *Prima facie* i have six states:

(i) odd length, residue 0;
(ii) odd length, residue 1;
(iii) odd length, residue -1;
(iv) even length,residue 0;
(v) even length, residue 1;
(vi) even length,residue -1...

... and these have to be paired up somehow. I think they go:

(ii)–(v), (iii)–(iv) and (i)–(vi). (The two with residue 0 have to be paired off co's we want only one accepting state, and the rest follows from that.) Call them A, B and C respectively. A is fixed by 0, C is fixed by 1. 0 swaps C and B. 1 swaps B and A.

This is actually quite a nice illustration of how one machine can be a quotient of another. There is an algebraic theory of DFAs but it's only for people with strong stomachs.

adj228 does it the wrong way round, appending on the left instead of on the right. He says his language is the reverse of mine. He and mn492 say that a number in base 2 is a multiple of 3 iff it has the same number of odd bits set as even bits set. That's not true actually, but something like it is true... I think it's necessary and sufficient for the difference between the number of *odd* bits that are set and the number of *even* bits that are set to be a multiple of 3. Either way the reverse of a binary representation of a multiple of 3 is also a binary representation of a multiple of 3.

I think that must be right. Think about test for divisiblity by 11 for numbers written in base 10.

Multiples of 3 in Base 2; an answer from Maurice Chiodo, doctored by me

"A: Construct a DFA D with:
Alphabet $\{0, 1\}$.
States $\{1, 2, 3\}$ (corresponding to remainders 0 mod 3, 1 mod 3, and 2 mod 3 respectively).
Start state 1.
Accept state 1.
(Notice that this means that we consider the empty string to denote zero. I'm happy with that but you might not be)[13]

Transition function: $(1, 0) \mapsto 1$; $(1, 1) \mapsto 2$; $(2, 0) \mapsto 3$; $(2, 1) \mapsto 1$; $(3, 0) \mapsto 2$; $(3, 1) \mapsto 3$.
Note that I am accepting "leading 0's"; *e.g.*, 000101 is a binary representation for 5.
The remainder of the long binary integer is 1 mod 3. (Use the DFA just constructed, and see which state you end up in).

Now, a regular expression for this language will be $R^{(3)}_{1,1}$. Recall we have the recursive definition

$$R^{(k)}_{i,j} := R^{(k-1)}_{i,j} + R^{(k-1)}_{i,k}(R^{(k-1)}_{k,k})^* R^{(k-1)}_{k,j}$$

So thus we are looking for

$$R^{(3)}_{1,1} := R^{(2)}_{1,1} + R^{(2)}_{1,3}(R^{(2)}_{3,3})^* R^{(2)}_{3,1}$$

[13]The footnote that used to be here, a discussion of the semantics of the empty string, has grown beyond all reason and has been moved to section 10.

We build this up inductively:
$R^{(0)}_{1,1} = \epsilon + \mathbf{0}$
$R^{(0)}_{1,2} = \mathbf{1}$
$R^{(0)}_{1,3} = \emptyset$
$R^{(0)}_{2,1} = \mathbf{1}$
$R^{(0)}_{2,2} = \epsilon$
$R^{(0)}_{2,3} = \mathbf{0}$
$R^{(0)}_{3,1} = \emptyset$
$R^{(0)}_{3,2} = \mathbf{0}$
$R^{(0)}_{3,3} = \epsilon + \mathbf{1}$

Now for the next step:

$R^{(1)}_{1,1} = R^{(0)}_{1,1} + R^{(0)}_{1,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,1} = \ldots = \mathbf{0}^*$
$R^{(1)}_{1,2} = R^{(0)}_{1,2} + R^{(0)}_{1,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,2} = \ldots = \mathbf{1} + \mathbf{0}^*\mathbf{1}$
$R^{(1)}_{1,3} = R^{(0)}_{1,3} + R^{(0)}_{1,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,3} = \ldots = \emptyset$
$R^{(1)}_{2,1} = R^{(0)}_{2,1} + R^{(0)}_{2,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,1} = \ldots = \mathbf{1} + \mathbf{1}(\mathbf{0})^*$
$R^{(1)}_{2,2} = R^{(0)}_{2,2} + R^{(0)}_{2,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,2} = \ldots = \epsilon + \mathbf{1}(\mathbf{0})^*\mathbf{1}$
$R^{(1)}_{2,3} = R^{(0)}_{2,3} + R^{(0)}_{2,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,3} = \ldots = \mathbf{0}$
$R^{(1)}_{3,1} = R^{(0)}_{3,1} + R^{(0)}_{3,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,1} = \ldots = \emptyset$
$R^{(1)}_{3,2} = R^{(0)}_{3,2} + R^{(0)}_{3,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,2} = \ldots = \mathbf{0}$
$R^{(1)}_{3,3} = R^{(0)}_{3,3} + R^{(0)}_{3,1}(R^{(0)}_{1,1})^* R^{(0)}_{1,3} = \ldots = \epsilon + \mathbf{1}$

Recall that we need only build $R^{(2)}_{1,1}$, $R^{(2)}_{1,3}$, $R^{(2)}_{3,3}$ and $R^{(2)}_{3,1}$ in the final stage. So:
$R^{(2)}_{1,1} = R^{(1)}_{1,1} + R^{(1)}_{1,2}(R^{(1)}_{2,2})^* R^{(1)}_{2,1} = \ldots = \mathbf{0} + (\mathbf{0}^*\mathbf{1})(\mathbf{1}(\mathbf{0})^*\mathbf{1})^*(\mathbf{1}(\mathbf{0})^*)$
$R^{(2)}_{1,3} = R^{(1)}_{1,3} + R^{(1)}_{1,2}(R^{(1)}_{2,2})^* R^{(1)}_{2,3} = \ldots = (\mathbf{0}^*\mathbf{1})(\mathbf{1}(\mathbf{0})^*\mathbf{1})^*\mathbf{0}$
$R^{(2)}_{3,3} = R^{(1)}_{3,3} + R^{(1)}_{3,2}(R^{(1)}_{2,2})^* R^{(1)}_{2,3} = \ldots = \epsilon + \mathbf{1} + \mathbf{0}(\mathbf{1}(\mathbf{0})^*\mathbf{1})^*\mathbf{0}$
$R^{(2)}_{3,1} = R^{(1)}_{3,1} + R^{(1)}_{3,2}(R^{(1)}_{2,2})^* R^{(1)}_{2,1} = \ldots = \mathbf{0}(\mathbf{1}(\mathbf{0})^*\mathbf{1})^*(\mathbf{1} + (\mathbf{0})^*)$

Finally, we get

$$R^{(3)}_{1,1} = \big(\mathbf{0}+(\mathbf{0}^*\mathbf{1})(\mathbf{1}(\mathbf{0})^*\mathbf{1})^*(\mathbf{1}(\mathbf{0})^*)\big)+\big((\mathbf{0}^*\mathbf{1})(\mathbf{1}(\mathbf{0})^*\mathbf{1})^*\mathbf{0}\big)\big(\mathbf{1}+\mathbf{0}(\mathbf{1}(\mathbf{0})^*\mathbf{1})^*\mathbf{0}\big)^*\big(\mathbf{0}(\mathbf{1}(\mathbf{0})^*\mathbf{1})^*(\mathbf{1}+(\mathbf{0})^*)\big)$$

which is the desired regular expression (though you may wish to double-check my working here...)

There is an easier way to do this, via 'elimination of states' (See Hopcroft, Section 3.2.2.) Doing this yields:

$$\big(\mathbf{0} + \mathbf{11} + \mathbf{10}(\mathbf{1} + \mathbf{00})^*\mathbf{01}\big)^*$$

or, even shorter still,

$$(\mathbf{0} + \mathbf{1}(\mathbf{01}^*\mathbf{0})^*\mathbf{1})^*$$

However, this was not taught in the lectures".

Question 9, 10

Ad coda to q9...

Of course, in principle, the DFA obtained from an NFA by the subset construction might be exponentially larger. One of my students was asking me: is there a natural example of an NFA whose corresponding DFA genuinely is exponentially larger? I couldn't think of one offhand, but on reflection, the following might be an example.

Suppose we have a deterministic machine $\mathfrak{M}$ for a language L. We can obtain from it an NFA for the set of all substrings of strings in L by putting in lots of "fast-forward" arrows. I hate to think what that looks like when you make it deterministic. Do you get exponential blowup...? Worth a thought.

Incidentally there is a connection here with question 7 (check the numbering!) on sheet 4 of Dr Chiodo's (now Dr Button's) sheet¡ldots the one about the language of substrings of a CFL

Question 11

The ultimate point of this question is that each regular language stands in the same relation to some DFA as each r.e. set stands to some register machine. (Even tho' that is not what you are explicitly asked to prove) Every regular language is the set of strings that cause some DFA to be in a accepting state; analogously every r.e. set is the set of those natural numbers that cause some register machine to `HALT`. The parallel is important and will help you orient yourself. It won't help you prove anything but it does give you a sense of what is going on. Later we will see how this parallel extends to context-free languages and PDAs.

It's worth asking how good are the parallels between the rows:

Register Machines	$\leftrightsquigarrow$ Semidecidable Sets
Pushdown Automata	$\leftrightsquigarrow$ Context-free Languages
DFAs	$\leftrightsquigarrow$ Regular Languages

Is is possible to see DFAs as degenerate PDAs and it is also possible to think of DFAs as degenerate Turing Machines (which are of course equivalent to Register Machines). A Turing machine can be seen as Mealy/Moore machine with an external memory device that compels it to read its own outputs: a Turing machine is an autocoprophagous[14] Mealy/Moore machine.

This question makes rather heavy weather of the fact that any regular language, thought of as a set of numerical codes for strings, is decidable ("recursive"). Every regular language L has a DFA $\mathfrak{M}$ of its very own (it will have several of course) and if we want to know whether or not a candidate string is a member of L we just feed it to $\mathfrak{M}$ and we get an answer—in time linear in the length of the string. So, by Church's thesis, there is a computable total function $\mathbb{N} \to \{\texttt{yes}, \texttt{no}\}$ which tells you whether or not a natural number is a code for a

[14] That's today's long word

member of L. Indeed there is even a computable function that takes a description of a regular language (as a DFA, an NFA, or a regexp or a grammar[15]) and returns code for the total computable characteristic function as above. So yes, DFAs are weaker than register machines: they have no memories[16]. We will see soon how you can sex DFAs up by giving them *some* memory, but less than register machines. Don't miss next week's thrilling installment on PDAs!

6.4 Sheet 3

Question (1)

Some of you asked about the motivation for the last part. The question first appeared on an example sheet devised by Dr Chiodo. He is interested in privacy, and questions of Ethics in Mathematics. (He has lectured courses on this). In his idiots'-guide-for-supervisors he writes:

> "Such a machine/expression would be used as part of a script to scrape webpages for personal details (name, address, date of birth, etc.) These are used because they are extremely fast, and can scrape large (or many) webpages quickly. Perhaps raise with the students the question of whether it is a *good idea* to be making tools to scrape the personal details of vast numbers of people from the internet."

Question (3)

There is a subtlety here that one shouldn't entirely ignore. Quite which machine turns out to be minimal for a language depends on what the alphabet is. The language in part (a) contains only strings of as but nothing has been said about the alphabet. If the machine is expecting to receive characters from $\Sigma = \{a, b\}$ rather than from $\Sigma = \{a\}$ then it has to have a state to run and hide in when it gets a b.

Question (4)

Let R, S, T be regular expressions. For each of the following statements, either prove it or find a counterexample.

(a) $\mathcal{L}(R(S+T)) = \mathcal{L}(RS) + \mathcal{L}(RT)$
(b) $\mathcal{L}((R^*)^*) = \mathcal{L}(R^*)$
(c) $\mathcal{L}((RS)^*) = \mathcal{L}(R^*S^*)$
(d) $\mathcal{L}((R+S)^*) = \mathcal{L}(R^*) + \mathcal{L}(S^*)$
(e) $\mathcal{L}((R^*S^*)^*) = \mathcal{L}((R+S)^*)$

If you make the mistake i initially made, and read the R's, 'S's and 'T's as characters from an alphabet rather than as regular expressions then it's all terribly easy. However, that was a mistake!

[15] No time for regular grammars in this course but if you want to pursue this stuff you will need to know about regular grammars.

[16] Tho' see the earlier discussion (section 3.2) of what DFAs can remember.

The way to show that two sets are the same is to show that they have the same members, and the way to show they have the same members is to take an arbitrary member of one and show that it belongs to the other, and *vice versa*. It can be quite a good idea to prove these inclusions by induction on the length of the strings. An induction on the structure of regular expressions can be good too

The equations that either have only one word in them, or lack asterisks, are OK. So there is no problem with (a) or (b). The others require thought. It's comparatively easy to demonstrate the falsehood of the false equations: it is sufficient to exhibit a counterexample. $(01)^* \neq 0^*1^*$ is a counterexample to (c); $(0+1)^* \neq 0^* + 1^*$ is a counterexample to (d).

Some of you have asked me about Part (e). Dr Chiodo said in his idiots'-guide-for-supervisors that the answer is (i quote) *elementary*. Well yes, but elementary doesn't always mean straightforward. A string on the LHS can be thought of (is obtained) as a series of blobs where each blob is some-Rs-followed-by-some-Ss. Put a '(' and a ')' round each blob and concatenate the blobs. So the brackets delineate the blobs and tell you how your string came to be a member of the LHS. Now throw away the brackets. You now have a thing on the RHS. For the other direction start with a thing in the RHS. It's a series of blobs each of which is a-few-Rs or a-few-Ss. Put a '(' and a ')' round each blob. You now have something that lives on the LHS.

Question (6)

Some of these things are regular, some not. You use the pumping lemma to show that something is not regular. To show that something is regular you can exhibit a DFA or a regular expression. Sometimes one is much easier than the other. "Contains a substring of five consecutive 0's"? Obviously a regular expression. "Has an even number of 0's and an even number of 1's"? Of course you want a machine.

Question (6e)

Liam Goddard puts it very well. The language in question is the union

$$\bigcup_{k<1001} \{a^n b^k : n \geq k\}$$

of $\{a^n b : n > 1\}$, $\{a^n b^2 : n > 2\}$, $\{a^n b^3 : n > 3\}$, ... all of which are regular, and there are only finitely many of them.

Question (6f)

The primes do not contain infinite arithmetic progressions!

Question (7)

We all know the pumping lemma. What is rather special about $\{a^n b^n : n \in \mathbb{N}\}$ is that *every* $w \in \{a^n b^n : n \in \mathbb{N}\}$, *however* decomposed into xyz with y nonzero gives $xy^n z \notin \{a^n b^n : n \in \mathbb{N}\}$ *for every* n. Of course every finite subset of $\{a^n b^n : n \in \mathbb{N}\}$ is regular, but if L is any infinite subset of $\{a^n b^n : n \in \mathbb{N}\}$ then any word in L can be pumped to obtain something not in $\{a^n b^n : n \in \mathbb{N}\}$ and therefore *a fortiori* not in L.

Question 8

There is a trap here that some of you fell into. It's certainly true that at least some of the strings in this language can be pumped up to obtain other strings in this language, but that isn't enough to show that the language is regular. Regular implies pumpable but not *vice versa*. As it happens this language is *not* regular.

The easiest way to show that is to recall that the complement of a regular language is regular, and that therefore the difference of two regular languages is regular. If $\{a^n b^m : m \neq m\}$ were regular, so, too, would be $a^* b^* \setminus \{a^n b^m : m \neq m\}$, and that is $\{a^n b^n : n \in \mathbb{N}\}$ and we know that that is not regular.

However it is possible to do it without trading on this fact. And (as so often!) it's more instructive to do it the wrong way. It's a lot harder, and it throws up some interesting concerns. For example Jamie Hiley says:

> "Consider $L = \{a^n b^m : n \neq m\}$. Assume it is regular, then there exists N as in the pumping lemma. Take $w = a^N b^m \in L$ where $m \neq N$ is to be decided. Then by the PL have $w = xyz$ where $x = a^p$, $y = a^q$ for $0 \leq p \leq N$ and $0 < p + q \leq N$. Moreover $xz = a^{N-q} b^m \in L$. But by choosing $m = N - q \neq N$ this is a contradiction and hence L is not regular."

He then asks "Am I allowed to let m depend on q? If not I can make the choice $m = n + n!$ work but am curious".

There is something here to worry about, as Jamie correctly suspects. One way of thinking of applications of the Pumping Lemma is as a game played between you and the machine—or, perhaps between you and the machine's minder, a shady character who takes the machine around fairgrounds making fraudulent claims that it can recognise a language L which he says is regular, but isn't. Your first move is to ask how many states the machine has. You get an answer, which Jamie calls N. You now cook up a word $w \in L$ with $|w| > N$. You do this beco's you *know* that w can be decomposed into xyz in such a way that you can pump up y. So the deal is this: you tell the machine what w is, and the machine has to tell you what x, y and z are. The point here is that: *you don't get to choose* y; it's the machine (or the minder) that tells you what y is.

Now let's look at Jamie's proof. He wants to choose m depending on q. But he has to choose m *before* he gets told what q is! That is: m is part of Jamie's

move, but q is part of the machine's *reply* to that move. So, no: he can't do that. However, this is not a huge problem, since if you choose the number of bs to have enough small factors then you can pump up the number of as to match the number of bs, and therefore land outside L. (As he says)

Question (9)

Obviously! If D_2 is not minimal then there is D_3 which is, and you take *its* complement and that will be smaller than D_1, contradicting assumption.

Question (10)

I think this is quite hard. If L and M are languages over the one alphabet accepted by machines $\mathfrak{M}_1$ and $\mathfrak{M}_2$ then the difference $L \setminus M$ is accepted by the product $\mathfrak{M}_1 \cdot \overline{\mathfrak{M}_2}$. (You know what I mean). Having L belong to an alphabet Σ and M belong to an entirely different alphabet Γ stuffs up the proof without altering the fact that $L \setminus M$ is regular. So why don't we just say that both L and M are over the alphabet $\Sigma \cup \Gamma$? We can, of course, but there is some housekeeping to do. The machine $\mathfrak{M}_1$ that recognises L (as a subset of Σ^*) has no arrows in its innards labelled with characters from $\Gamma \setminus \Sigma$. To turn it into a machine that processes strings from $\Sigma \cup \Gamma$ we have to add, for each of its states and for each character c in $\Gamma \setminus \Sigma$, an arrow to a terminally unhappy state. So far so good. We do the analogous thing for $\mathfrak{M}_2$ of course. We now have two modified machines $\mathfrak{M}_1'$ and $\mathfrak{M}_2'$. We take the product $\mathfrak{M}_1' \cdot \overline{\mathfrak{M}_2'}$. (Think about what has happened, in $\overline{\mathfrak{M}_2'}$, to the terminally unhappy states in $\mathfrak{M}_2'$; they have become blissed out states that accept everything.) I wonder what sort of smiley one should notate them with?! Perhaps

The time has now come to think through the question of which strings are accepted by $\mathfrak{M}_1' \cdot \overline{\mathfrak{M}_2'}$. We shall continue to think of this as $\mathfrak{M}_1'$ and $\overline{\mathfrak{M}_2'}$ being run in parallel.

- What happens when we feed to this consortium a string that contains characters from $\Gamma \setminus \Sigma$? As soon as we hit such a character $\mathfrak{M}_1'$ goes into a terminally unhappy state and forbids the consortium to accept.
- As soon as it hits a character that is in $\Sigma \setminus \Gamma$ the consortium member $\overline{\mathfrak{M}_2'}$ goes into a blissed-out state, so that all decisions about whether to accept end-extensions of that string are made by the other consortium member, $\mathfrak{M}_1'$.

Which is exactly what we wanted.

Question (11)

We covered most of this in sheet 2, but i do have the extra thought.... You remember the test for divisibility by 11 for numbers notated in base 10? There is an analogous test for multiples of 3 for numbers written in base 2. What is it? I mention this beco's it might give you another regular expression for multiples of 3 written in base 2. Worth a try.

Just occurred to me. What happens if you write this regular expression backwards?

Generally i see no reason why Kleene's algorithm for obtaining a regular expression from a DFA should give you the shortest such expression. In fact it's pretty clear that it doesn't.

Question (13)

This harks back to Q11 of sheet 2.

The reverse of a regular language is regular, as any fule kno. Therefore, if the displayed language were regular, so—too—would be the language

$$\{1^n 0w : w \in \{0,1\}^* \wedge n \in \mathbb{K}\}$$

and also the intersection of that second language with the regular language 1^*0, which is

$$\{1^n 0w : n \in \mathbb{K}\}$$

and it's easy enuff to show that *that* ain't regular. A machine that recognised it would accept all and only those strings of the form 1^n0 where $n \in \mathbb{K}$, so it would solve the `HALT`ing problem. In your dreams innit.

(It is possible to do it without turning the language back-to-front, but this is the cutest proof known to me).

Now to show that the original language can be pumped. What do we mean by that? We mean that any sufficiently long word in this language can be split into w_1vw_2 in such a way that, for all $n \in \mathbb{N}$, $w_1v^nw_2$ is also in the language. (Brief reality check: the reverse of a pumpable language is pumpable.)

Notice that the claim is not that *however* you split a suff long word into three bits you can pump up the middle bit s.t. etc etc, merely that *there is a way of* splitting up such a word into threee bits s.t. etc etc.

A string consisting entirely of 1s can clearly be pumped. So what do we do with a string that contains a 0? Such a string is $w1^n$ for some $n \in \mathbb{K}$ and some $w \in \{0,1\}^*$ whose last element is 0. Then you just pump up the w!

I think one of the points being made here is that you use the pumping lemma to show that a language is *not* regular, not to show that it *is*.

Question (14)

Beware! He's asking you about the empty language, not the language containing only the empty string!

Question (15)*

I don't think this is hard enuff to justify a star, but i might just be cruel and old-fashioned enuff to think students should have to do a bit of work every now and then.

The *complement* of a DFA D is obtained from D by turning all accepting states into nonaccepting states and *vice versa* . We have a concept of the product of two machines (for accepting the intersection of two languages)[17]. Consider then the product of D_1 with the complement of D_2. Does this machine accept any strings? You can ascertain this by a breadth-first search for an accepting state starting at the start state. It takes linear time. Piece of *cake*.

Question (16)*

I don't think this is hard enuff to justify a star, but—again—i might just be cruel and old-fashioned enuff to think students should have to do a bit of work every now and then. ("We lived at the bottom of a lake..."—"Luxury!")

As Alistair Bill points out, if X is finite, as it might be $\{u, v, w\}$ then X^* has the regexp $(u|v|w)^*$, but of course it might not be.

The strings in X differ only in their length, so this is not really a question about strings but about natural numbers. Think of X as a subset of $\mathbb{N}$, and then X^* becomes the closure of X under addition. If you think about this for a bit it becomes obvious that X^* contains all sufficiently large multiples of $\mathrm{LCM}(X)$[18]. The set of all strings-of-1s whose lengths are a multiple of a natural number k is a regular language, and any language with finite symmetric difference from this is also regular. Piece of cake.

If there is a moral to this it's probably something along the lines of: *never pass up a chance to discard irrelevant information.* (Remember how you had to ignore the number 5 in question 4 of sheet 2?)

It also gives you something of the flavour of research mathematics. You encounter a problem and it seems to be about dingbats, but it's actually about wombats.

Actually there might be another moral. Some of you were worried by the prospect of forming the highest common factor of an *infinite* set of natural numbers. You are right in your suspicion that there is something to think about here; after all, *HCF* is (in the first instance) an operation on *two* numbers. Since it is associative it can be extended so it is defined on all finite multisets of numbers. (It's an important triviality that any associative binary operation on a set can be naturally extended to an operations of lists from that set, not just ordered pairs.) You were encouraged to worry about infinite sums and products in Analysis II, so justification is neeeded if one is to apply it to infinite sets. The justification here is rather different from the justification of infinite sums in Analysis II; it's OK, but not just (as some of you said) because *HCF* is

[17] I don't think either of these facts were lectured, so perhaps that is why the question gets a star

[18] Tho' i can't offhand see a cute upper bound. It would be nice to have one.

monotonic (decreasing) wrt $\subseteq$ (on subsets of $\mathbb{N}$) and every set of naturals has a least member, so you learn your eventual answer in finite time. [actually even montonicity is not enuff: what is needed is continuity at limits] That doesn't make it computable, and it's a distraction. More to the point is that

$$\mathrm{HCF}(X) = \mathrm{LCM}\{y : (\forall x \in X)(y|x)\}$$

And clearly the LCM on the RHS is the LCM of a finite set, since only finitely many things can divide *everything* in X. (They've all got to be smaller than $\min(X)$.)

Harking back to the first part of the course ... How hard is it to find the HCF of an infinite set of naturals anyway? Let $n \in \mathbb{N}$, ask for $\mathrm{HCF}(\{n\}\text{“}\mathbb{N})$. Is this function computable? It shouldn't be hard to show that it isn't.

6.5 Sheet 4

I haven't yet renumbered these questions to make them compliant with the number in 2021/2

There is a question that makes the point that if L is a context-free language then the set of all substrings of members of L is likewise context-free. It doesn't say that every subset of a CF language is CF! That second thing sounds the same, but it isn't! ... and that's for the obvious reason that every language is a subset of Σ^*, the universal language, and that language *is* context-free.

6.5.1 Question 4, starred part

Show that $\{a, b\}^* \setminus \{ww : w \in \{a, b\}^*\}$ is context-free.

Dr Chiodo gives a grammar:

$S \to A|B|AB|BA;$
$A \to CAC|a;$
$B \to CBC|b;$
$C \to a|b$

He supplies us this grammar, but I think a determined student would probably be able to work out for themselves that something along those lines might work. The hard part comes in showing that it not only *might* work, but that it does *in fact* work.

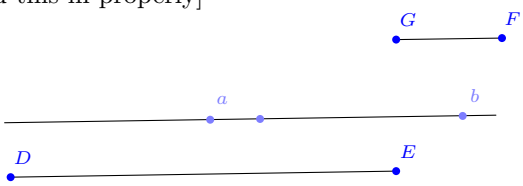
Every string corresponding to an A or a B (let's call them *A strings and B strings*) is of odd length and therefore can't be of the form ww. However we *do* have to show that every string AB is not of the form ww. Every A-string is of odd length and has an 'a' at its heart; every B-string is of odd length and has a 'b' at its heart. In fact the A-strings are *precisely* the set of those strings of odd length with an 'a' in the middle and the B-strings are *precisely* the set of those strings of odd length with an 'b' in the middle. We want the set of

AB strings and BA strings to be precisely our putatively context-free language, and if the A string and the B string that go into our AB string are the same length we get what we want. However in an AB string (*mutatis mutandis* a BA string) the A and B moieties might be of different lengths. But this is OK! Suppose A has become the three string $(\cdot a \cdot)$ and B has become the five-string $(\cdot\cdot b \cdot\cdot)$. Now comes the clever bit. AB is the 8-string $(\cdot a \cdots b \cdot\cdot)$, and you think of it as the concatenation of two 4-strings. Now the first 4-string has 'a' as its second member and the second 4-string has 'b' as its second member—so they are distinct!!

Let's write this out properly for the general case. Suppose we have a string s of even length, that is: an AB string or a BA string, is without loss of generality an AB string. It's of length $2n+1 \ + \ 2k+1$, where the first $2n+1$ characters are an A string and the following $2k+1$ characters are a B string. s is of the form ww' where w and w' are both of length $n+k+1$. w is a string whose nth element is 'a' and w' is a string whose nth element is 'b', giving $w \neq w'$.

There is an extra wrinkle however, which Mr Cowperthwaite of Girton mentioned in a supervision. The grammar does not invite us to add AA or BB to the list of words it generates. How can we be sure that this omission is safe? Might there not be two A words A and A' such that AA' is not of the form ww but which nevertheless cannot be obtained as an AB nor as a BA? Well, suppose we have a word W which can be obtained as a concatenation AA' of two A words but cannot be decomposed into an A word followed by a B word or *vice versa*. In particular if we chop up AA' into-one-character-followed-by-$|AA'|-1$-characters then this decomposition is not an A word followed by a B word nor *vice versa*. So the first character of AA' must be the same as the $|AA'|/2$th character. And the same must go for the second character! So our candidate was of the form ww after all.

[blend this in properly]



The line in the middle is a word of even length that is not ww. At the wee blue dot it divides into two bits of equal length. These two bits differ beco's at one position the left half has a a and the right half has a b. The lower line DE is an A-word, co's it has that a at its midpoint, and the upper line GF is (rather shorter) B word with the above-mentioned b in the middle. So the line in the middle is a A word followed by a B word as desired.

Callum Hobbis asks a *very* good question:
How about the language $\{a,b\}^* \setminus \{www : w \in \{a,b\}^*\}$*?!*

6.5.2 Question 8

Most of you did this by the technically defensible way of taking a grammar such as

$$A \to SS \mid S + S \mid S^* \mid (S)$$

which has the effect that (((((((a))))))) is a regular expression—and of course you can pump up something like that. You probably heeded the siren voices asking you to look at the brackets. However (he says severely) (((((((a))))))) is not morally correct. What am i complaining about? Well, if i were to ask you what alphabet the regular expression $(a|b)c*$ (for example) comes from, you would say $\Sigma = \{a, b, c\}$ wouldn't you? You woudn't say $\Sigma = \{a, b, c, (,)\}$ would you? My take is that the parentheses are not part of the alphabet, and they are only there for punctuation. How many letters does the english alphabet have? Yes, twenty six, not thirty-two ... with '.', ',', ';',':','?' and '!'.

You should have a grammar like

$$A \ \to \ (S)(S) \mid (S) + (S) \mid (S)^*$$

or perhaps

$$A \ \to \ (SS) \mid (S + S) \mid (S^*)$$

where parentheses are inserted only round strings that are being incorporated into longer strings. *This* grammar doesn't tell you that (((((((a))))))) is a regular expression. No more it should, co's morally (((((((a))))))) isn't a regular expression, now, is it!? You *knew* that all along—admit it. The only reason why you need the brackets is for punctuation when composing things with binary constructors. Accordingly you should put them in *only* when composing things using binary constructors.

But of course the path of virtue is not terrible easy. Now we need the concept of a homomorphic image of a language. Suppose Σ_1 and Σ_2 are two alphabets, and $f : \Sigma_1 \to \Sigma_2$ and $L \subseteq \Sigma_1$. Then (i think!) L is regular iff f"L_1 is regular. And apparently it works for context-free languages too. This is related to the fact that there is a good notion of homomorphism of DFA (and probably NFAs too). By this means we can show that if the language of regular expressions were regular so too would be the matching bracket language, and it ain't, as any fule kno.

6.5.3 Question 9

$L_1 := \{a^n b^n c^i : n, i \in \mathbb{N}\}$ and $L_2 := \{a^n b^i c^i : n, i \in \mathbb{N}\}$. Clearly both are CF and clearly their intersection isn't. Charlie Brooker says: We know that $\overline{\overline{L_1} \cup \overline{L_2}}$ isn't CF, but is $\overline{L_1} \cup \overline{L_2}$ CF? And which of the two complements $\overline{L_1}$ and $\overline{L_2}$ are CF? Off the top of my head i have no idea, but it's a very good follow-up question. Of course if you want a direct proof that the complement of a CF language is not reliably CF you look at the last two parts of question 4 on this sheet.

Incidentally one point that this makes is that there is no good notion of the complement of a PDA (nor a good notion of the product of two PDAs) in the way that there is a good notion of complement of a DFA.

6.5.4 Question 11

The key to this question is finding a specification of the Abstract Data Type of PDA's that allows that an NFA is just a degenerate PDA. You are by now familiar with a product construction for DFAs. Tweak it a bit so you have a good notion of a product of two NFAs, and then combine that with your ADT for PDAs that allows that an NFA is just a degenerate PDA. Then tweak your notion of product of DFAs so you have a notion of a product of an NFA and a PDA. The significance of *product* here is that the intersection of two regular languages L_1 and L_2 is the language recognised by the product of the two DFAs that recognise L_1 and L_2.

I have to admit that this is a very logic-y question. To do it you have to think quite hard about specifications of Abstract Data Types, and this is something that mathematicians don't normally expect to have to do. One tends to expect that the specifications of the abstract data types that we use have been sorted out before we sit down to work.

I should really do this myself one day! This is definitely a conceptual problem (and therefore an example sheet question) rather than a technical performance-art one (which is why it isn't an exam question!)

7 Discussions of questions on examples sheets for Part II Maths Languages and Automata from previous years

If you want to try some old tripos questions warn me in advance and i'll work up some discussion answers.

7.0.1 Question 12 on Sheet 2 2018/9

For the punchline, look up: *"There Are No Safe Virus Tests" William F. Dowling, The American Mathematical Monthly* **96**, *No. 9 (Nov., 1989), pp. 835–836*

Question 13 on Sheet 2 2018/9

Let's sharpen this up a bit. Reader: go forth and find a family $\langle A_n : n \in \mathbb{N} \rangle$ of decidable (OK, you can call them recursive if you like) sets that are *nested*: $n < m \rightarrow A_m \subseteq A_n$, whose intersection is not semidecidable (OK, not r.e.) Start by asking yourself what your favourite example is of a set that is not semidecidable.

8 Old Tripos Questions for Part II Maths Languages and Automata

https://tartarus.org/gareth/maths/tripos/II/Automata_and_Formal_Languages.pdf

8.1 2017

2017 paper 1 section II 11H

It's an old example sheet question: section 3.9 p. 18.

2017 paper 3 section II 11H

(a) Given $A, B \subseteq \mathbb{N}$, define a many-one reduction of A to B. Show that if B is recursively enumerable (r.e.) and $A \leq_m B$ then A is also recursively enumerable.

(b) State the s-m-n theorem, and use it to prove that a set $X \subseteq \mathbb{N}$ is r.e. if and only if $X \leq_m \mathbb{K}$.

(c) Consider the sets of integers $P, Q \subseteq \mathbb{N}$ defined via

$P := \{n \in \mathbb{N} : n \text{ codes a program and } W_n \text{ is finite}\}$

$Q := \{n \in \mathbb{N} : n \text{ codes a program and } W_n \text{ is recursive}\}$.

Show that $P \leq_m Q$.

I have no idea how to do this. Go to the author and play the Helpless Oldie[19].

A message from Maurice Chiodo. He says:

"Look at question 13 of example sheet 2. In the crib sheet, i discuss approximating recursive sets by finite sets.

Here is the way to answer part (c) of that question:

- First, fix a register machine T which halts on the Halting Set $\mathbb{K}$.
- Take W_n
- Start enumerating elements of W_n
- Each time you find a new element (say the kth element) in W_n, form the k-step approximation A_k to the halting set $\mathbb{K}$ (that is, take the machine T, run it on all inputs from 0 to k, doing k steps of computation on each input).
- It is clear that A_k is finite, and we can construct A_k uniformly from k (I just showed how, and remember you have the machine T for K in your pocket).
- Now take the union $B = \bigcup A_k$ over all the k that are enumerated from the re set W_n. (B approximates the Halting Set $\mathbb{K}$)
- It is clear that we can construct B from the re set W_n.
- If W_n is finite, then B is finite, and thus recursive.
- If W_n is infinite, then $B = \mathbb{K}$, and so B is not recursive.

This is your many-one reduction :)"

[19] I do quite a good Helpless Oldie, if i say so myself; i've had years of practice.

2017 paper 3 11H—Duh!

from 2021.

I should've looked—i had an answer on file already!!

Part (b)
This is the bit i have difficulty remembering

Remark 3 *$A \leq_m \mathbb{K}$ iff A is semidecidable.*

Proof:
L $\rightarrow$ R

Since $\mathbb{K}$ is semidecidable it is the domain of a partial recursive g. If $A \leq_m \mathbb{K}$ in virtue of f (that is to say: $f : V \rightarrow V$ is total computable and the range of $f \upharpoonright A$ is included in $\mathbb{K}$) then A is the range of $g \circ f$, which makes A semidecidable.

For the other direction, suppose A is semidecidable. Define a binary partial (computable) function f by

$$f(e, x) =: \texttt{if } e \in A \texttt{ then } 1 \texttt{ else } \uparrow$$

By the S-m-n theorem there is now a computable function g such that, for all x and e, $\{g(e)\}(x) = f(e, x)$. From this we have $(\forall e)(\{g(e)\}(g(e))\downarrow \longleftrightarrow e \in A)$. Thus $(\forall e)(e \in A \longleftrightarrow g(e) \in \mathbb{K})$. (Here we take $\mathbb{K}$ to be the *diagonal* halting set). But now $A \leq_m \mathbb{K}$ in virtue of g.

Part (c)

The question to ask yourself is: "I have a candidate k for membership of P. What computable function f is such that $k \in P$ iff $f(k) \in Q$?" Somehow—in a computable way—i have to obtain from k something which is in Q if $k \in P$. What is that something?

I think you want the function that does the following. Input m. If $\{m\}(m) \downarrow$, run the volcano for $\{n\}$ until it has output m things. Then `halt`. This function is trying to output $\mathbb{K}$ and it will succeed if W_n is infinite; if W_n is finite then our function has finite range. If W_n is infinite then W of our function is the `halt`ing set, wot ain't recursive.

8.1.1 2017:4:4H

Part (b)

As Sam Watts says, $L \backslash M$ is regular if L and M are, and the reverse of a regular language is regular. So, if this language is regular, so is $\{1^n 0w : w \in \{0,1\}^* \wedge n \in \mathbb{K}\}$, and therefore so too would be $\{1^n 0 : n \in \mathbb{K}\}$ and that clearly isn't regular, co's any machine that recognised it would solve the `Halt`ing problem.

8.1.2 2017:3:11H

I couldn't do this bit. But then (according to the examiners' *post mortem*) nor could any of the candidates. However my supervisee Andrew Slattery could.

May he live for ever. I gave him a chocolate but it should've been two; what a star. This presentation is mine, but it's his answer.

Suppose i have a gremlin in my attic that will, on being given $n \in \mathbb{N}$, think for an indeterminate time and say 'yes' if W_n is recursive but will remain silent if it isn't. I have to obtain from this gremlin a gremlin that will, on being given $n \in \mathbb{N}$, think for an indeterminate time and say 'yes' if W_n is finite but will remain silent if it isn't.

Here's how i do it.

The function A_m (for 'Andrew') is defined as follows. I equip myself with a volcano $V_{\mathbb{K}}$ for the machine that computes $\lambda n.$ if $\{n\}(n) \downarrow$ then n.

Someone comes in off the street and shows me a natural number m and wants to be told that W_m is finite—if it is, that is.

I build a volcano V_m for the function $\{m\}$. The kth time V_m emits something that it hasn't emitted before[20] I run $V_{\mathbb{K}}$ until it emits its kth value, and I emit that as $A_m(k)$.

(This question is in a context where the S-m-n theorem is being waved about. It's the S-m-n theorem that tells you that the function that accepts m and returns A_m is computable. I tend to think of it as Church's thesis that guarantees computability of this function but you could say that it's the S-m-n theorem that guarantees it and that the job of Church's thesis is to make the S-m-n theorem obvious.)

This describes A_m. Notice that i am not proposing to *run* it; i merely need the code for it, since i intend to show the code to my gremlin. It may seem odd that it is essential that one should cook up code to do something-or-other, but then doesn't need to run it. Computation theory is full of weird things like that. My gremlin can tell me if the range of A_m is decidable. Clearly if the range of $\{m\}$ is finite then the range of my function is finite and my gremlin will detect this fact. The only circumstances in which the range of A_m is decidable is when it is finite. This is beco's if the range is infinite then the range of A_m is the halting set (so my gremlin will say nothing). Thus $\{m\}$ has finite range iff my gremlin says that the range of A_m is decidable.

Being a wally i had entirely forgotten that i had written out an answer to this question a year earlier!!!

Problem reduction exercises are always hacky and *ad hoc* ("ad hack" the wags say). There are a few tricks you can try but none of them really amount to a technique. You just have to snoop around looking for unlocked windows.

This particular problem reduction exercise is this: given a machine that, given n, will go PING! if W_n is decidable (we know it is at least *semi*decidable), use it to answer questions of the kind: "Is W_m finite?"

[20] When i wrote this i was assuming that W_n is $\{n\}$"$\mathbb{N}$, the set of *values* of the function $\{n\}$. My understanding was that the 'W' came from German *Wertebereich* which (ought to mean) *domain of values*. But perhaps it means: *domain on which* $\{n\}$ *is defined* .... Either way this construction works.

To do this tweaking we have to be able to do the following: given m, come up (computably!) with n s.t. W_n is recursive iff W_m is finite. So: if W_m is *in*finite then W_n must fail to be decidable. What is your favourite example of a semidecidable set that is semidecidable but not decidable? As Imre would say: "Switch brain off and do the obvious thing". Yes, $\mathbb{K}$. Duh.

So, given m, i compute n as follows. I get a volcano V (a machine running in parallel with itself that emits numbers without being asked) that emits members of W_m. While this is going on i am trying to compute members of $\mathbb{K}$. I do this by running $\{k\}(k)$ for lots of k in parallel. Which k? Well, at each stage i am running $\{k\}(k)$ in parallel on all the k that are below the largest number emitted by V so far. This process i have described is parametrised by m and so represents a function from $\mathbb{N}$ to $\mathbb{N}$. The set of numbers i get is of course semidecidable and is W_n for some n, and, yes, i can compute this n from m.

By assumption i have a machine $\mathfrak{M}$, wot i have trafficked from Eastern Europe and am keeping in inhumane and degrading conditions in a mouldy and rat-infested attic, where I use it to answer questions of the form "Is W_n a decidable set?" I now force open the jaws of my machine and insert the number n that i got from the preceding paragraph. What might W_n be? It might be finite, and if it's finite, well, it's finite. But it might be infinite. But if it's infinite it must be $\mathbb{K}$, and so is not decidable! So if $\mathfrak{M}$ says that W_n is recursive it must be that it is finite, but that means that W_m was finite!

8.2 2018

8.2.1 2018:1:12G

I like to use the word *gatekeeper* in connection with $\leq_m$. The point is that, if $A \leq_m B$ then if you have a gatekeeper for B (someone who can recognise members of B[21]) then you can tweak it into a gatekeeper for A. That's why $\leq_m$ matters.

Andrea Szocs points out that, the way $\oplus$ is defined, $X \oplus Y$ is $\mathbb{N} \setminus (Y \oplus X)$. This is a perfect example of the kind of hacky fact that an assembly language programmer might make use of, but which a mathematician need take no account of. It's not a fact about the $\oplus$ operation (which is of course disjoint union) but about the particular implementation of $\oplus$ that we have been told to use.

8.3 2019

8.3.1 Paper 1, Section II 12H

For the last part you are going to need seven states, beco's you need to keep track of the remainder mod 7 of the number-in-hand. One way of locating the machine that does this is to recall that if you write a number in octal then it is divisible by 7 iff the sum of its digits is a multiple of 7. (why?) The other way

[21] 'Bouncer' is also a possibility, but we're in Cambridge, so perhaps *porter* would be better!

of doing it (not exploiting the sum-of-digits trick) is to have one state for each remainder-mod-7. That's seven states too. Seven is clearly best possible.

8.3.2 Paper 2, Section II 4H

There is a subtlety in the last part. What you *want* to say of course is

$$f(0) = 0;\ f(1) = 0;\ f(S(n) = S(f(n))$$

... but that doesn't match the template. You have to use the auxilliary primrec function $g(x, y) = x$.

Then you can say

$$f(0) = 0; f(S(n)) = g(S(f(n)), n)$$

8.4 2020

8.4.1 Paper 1: 12F

(a) Define a register machine, a sequence of instructions for a register machine and a partial computable function. How do we encode a register machine?

(b) What is a partial recursive function? Show that a partial computable function is partial recursive. [You may assume that for a given machine with a given number of inputs, the function outputting its state in terms of the inputs and the time t is recursive.]

(c)

> (i) Let $g : \mathbb{N} \to \mathbb{N}$ be the partial function defined as follows: if n codes a register machine and the ensuing partial function $f_{n,1}$ is defined at n, set $g(n) = f_{n,1}(n) + 1$. Otherwise set $g(n) = 0$. Is g a partial computable function?
>
> (ii) Let $h : \mathbb{N} \to \mathbb{N}$ be the partial function defined as follows: if n codes a register machine and the ensuing partial function $f_{n,1}$ is defined at n, set $h(n) = f_{n,1}(n) + 1$. Otherwise, set $h(n) = 0$ if n is odd and let $h(n)$ be undefined if n is even. Is h a partial computable function?

ANSWER

(i) Observe that g is a total function. Observe also that the way g is defined—it *diagonalises out of* the family of computable functions—it can't be computable if it's total. (Apply it to its own index). But it is total by construction; so it cannot be computable.

(ii) I think the same trick of self-application will help in this second part too, but i haven't yet seen how. It's obviously not recursive, but ...

8.4.2 Paper 3: 12F

Look at section 3.11.

9 Some Answers

.

The answer to my query at the end of question 1 on Sheet 2 (p 39) seems to me to be as follows.

Evidently if A and B are both semidecidable—so that, if $a \in A$ and $b \in B$, then we will learn this fact in finite time—then we learn in finite time that $\langle a, b\rangle \in A \times B$—if it is. For the other direction, reflect that ... if $A \times B$ is semidecidable then how do we learn that $a \in A$? Well, if it is, then—for any $b \in B$—$\langle a, b\rangle \in A \times B$. So we just zigzag across attempts to prove that $\langle a, b\rangle$ over all $b \in \mathbb{N}$ and we will eventually be told that $\langle a, b\rangle \in A \times B$ for some b and that will tell us that $a \in A$.

10 Appendix on The empty string

Recall the machine that recognises multiples of 3 in base 2.

What does one want to say about the start state, the state it is in when it hasn't been fed anything? Is that to be the same as the accepting state or not? To what natural number does the empty string from $\{0, 1\}$ correspond? If it corresponds to 0 then the start state is the same as the accepting state. If it corresponds to anything (and how big an 'if' that is is a matter for discussion) then it must correspond to 0. That's because when we append a '0' to it we have the string 0 which of course corresponds to the natural number zero, and this appending-of-'0' corresponds to multiplying-by-2, and an appending-of-'1' corresponds to multiplying-by-2-and-then-adding-1. Both these thoughts tell us that the empty string must point to the number 0.

But, says Harry Roberts (in his silly Father Christmas hat which he has been wearing all week—i mean: how can you take seriously anyone wearing such a hat, I ask you?!) it doesn't *prima facie* mean anything at all, so it's *by convention* that you decide it means 0. His thought then is (I've monkeyed around with this a bit) is that if you wake the machine up, and then press ⏎ before you have entered a character from the alphabet $\{0, 1\}$ you'd get an error message. He's right. In fact if, at *any* point in a sequence of keystrokes: ... `character`, ⏎ , `character`, ⏎ ... I press ⏎ without immediately preceding it with a character i'll get ... what? Harry says i'd get an error message. I might, i suppose. But it wouldn't be from the DFA, it would be from the DFA's *minder*, the *operating system*. Actually the O/S probably wouldn't even bother to send me an error message, on the grounds that the simplest thing to do in these circumstances is to ignore the ectopic carriage return altogether. But i think it's important that the error message [if there is one] will come from the user interface not from the machine.

How perverse is Harry being? Suppose i were to pretend that i don't know what multiplication by 0 is, and that $x \cdot 0$ (for $x \in \mathbb{N}$) is *prima facie* undefined. I could probably be talked into agreeing (with a becomingly modest display of reluctance, of course) to a convention that says that $x \cdot 0 = 0$, on the grounds

that it makes the distributivity *etc etc* work. But we all think that that is perverse . . . don't we?

Yes it *is* perverse, but perverse rather than actually *false*. If i define an operation of multiplication on $(\mathbb{N} \setminus \{0\}) \times (\mathbb{N} \setminus \{0\})$ then how i extend it to the whole of $\mathbb{N}^2$ is entirely up to me. Isn't it. . . ?

And it's surely not *by convention* that we think that when, in the course of doing a proof by resolution, we have resolved to the empty disjunction, then we have proved the false?

OK, so is it OK to think that the empty string denotes 0? I have been gaily saying that it is, but Angus Matthews has been impressing on me that it depends on what you want to do with it. What we have here is a homomorphism from strings (of 0s and 1s) to natural numbers, and we want the homomorphism to behave nicely so that—for example—concatenation of strings corresponds nicely to something arithmétic. Mostly it does, but allowing the empty string to denote 0 buggers up the homomorphism. The problem is that $s :: \{0\}$ (which is s with the singleton string $\{0\}$ consed on the end) denotes (as it were) $2s$ whereas $s :: \epsilon$ (s with the empty string consed on the end) is of course just s. (If we cons them the other way round it's all right, but that's no consolation). So—in this context—the difference between $\{0\}$ and ϵ actually matters.

Mind you, as Angus says, this problem with consing isn't a problem about the empty string, co's you get the same problem with the two strings 0 and 00; they both point to zero, but they have different effects when you cons them on the end of a string.

11 Some thoughts about Trakhtenbrot's Theorem

Let us say that a fmla of FOL is valid$'$ iff it is true in all finite models. Trakhtenbrot's theorem says that the set of valid$'$ FOL sentences is not semidecidable. Naturally this is something to do with the Halting problem—undecidability always is! The idea is, on being given an instance of the Halting problem $f(n)\downarrow$? is to cook up a expression such that any finite model is a turing machine for computing $f(n)$ that has not HALTed. If this sentence is true in all finite structures then $f(n)\uparrow$. If we can detect the fact that it is true in all finite structures then we can detect the fact that $f(n)$ never HALTS. And of course we can't.

But what about the valid$'$ expressions of $\mathcal{L}(\in, =)$ that happen to be stratified? Is it easy to show that that is not semidecidable? And what about the valid$'$ expressions of the strongly typed language $\mathcal{L}(\langle \in_i : i \in \mathbb{Z} \rangle, =)$?

12 Prof Pitts' 1B CS Computation theory notes: A Discussion of Exercise 10 part (c)

A note from april 2020.

Two of my supervisees were asking me about the last part of Q 10 on [4]. I think it may be worth setting down in writing the off-the-cuff answer i gave them, and amplifying it.

The question is: show that all the slices of the Ackermann function are primitive recursive. You prove by induction on n that the nth slice is primitive recursive. That is to say, you show how to obtain a primitive recursive declaration for the $n+1$th slice from a primitive recursive declaration for the nth slice. I might write out a proof of this later if there is a call for it. It's fiddly, but there is no great mystery to it.

The next part is: why does this show that the Ackermann function is total recursive? The answer to this is quite subtle. How do I compute, say, ack$(15, 17)$? Well, I reach for the code for the 15th slice and I run it on input 17. Since the 15th slice is primitive recursive it is total (remember that we proved by induction on the recursive datatype of primitive recursive functions that they are all total) so when we run the code for the 15th slice on the input 17 we can be confident of getting an answer.

This prompts the question (which was what I took my supervisee to be asking) Why does this not prove that the Ackermann function is primitive recursive?—and i shall answer that later. However it later occurred to me that what she was unhappy about was perhaps the expression 'total recursive'. Here 'recursive'—as so often—just means 'computable'. It is also recursive in a more natural sense in that it has a declaration which uses recursion, but is not *primitive* recursive in that it recurses on two variable places simultaneously.

What Prof. Pitts wanted you to say was that, on the basis that all the slices are primitive recursive, one can conclude that one can reliably compute Ackermann on all inputs.

What i took her question to be was: "Why does this not prove that the Ackermann function is *primitive* recursive?" Good question.

The first point to make is that the recursive datatype of primitive recursive functions is closed under `if` ... `then` ... `else` ... in the sense that if f and g are primitive recursive functions and p is a predicate whose characteristic function is primitive recursive then

`if` p `then` f `else` g

is primitive recursive too. (I don't know if it is proved in the lectures—there isn't time to prove everything! You might like to try it as an exercise ... it's a standard fact)

Indeed we can clearly stitch together any fixed finite number of primitive recursive functions in this way.

Contrast this with what we are doing when we try to compute Ackermann of x and y. We call up the code for the xth slice of Ackermann, and feed y to it. But there are infinitely many slices! And we can't reach them all with finitely many `branch` commands like the one above. We have to do something like:

`Input` x `and` y;
`compute the code for the` x`th slice of Ackermann`;

`run that code on` y.

. . . which calls the function that accepts input i and outputs code for the ith slice of the Ackermann function. This is a perfectly respectable computable function; it just doesn't happen to be primitive recursive. [proving that it isn't primitive recursive relies on showing that Ackermann itself is not primitive recursive and we don't need to get into why that is so unless you want to.]

This gives us another illustration of the central ambiguity in Computation Theory: a number can be both a data object (input to a computable function) and—simultaneously—a numeric code for a program.

What is being pointed up here is the difference between two ways in which code for functions g, h, . . . might be embedded/encoded/represented inside the code for a function f:

> • In the first case (with the `if` . . . `then` . . . `else` . . .) the code for the embedded functions is hard-coded: code for each of the embedded functions is there *as itself* as it were. This route is available if the number of functions to be embedded is finite.
>
> • In the second case (the case of the Ackermann function we are considering) no code is there as–itself; what we have is code that will output code for each of the embedded function on demand.

Frege, writing about something closely related to this, has a wonderful image that may be of help:

> "*Sie sind in der Tat in den Definitionen enthalten, aber wie die Pflanze im Samen, nicht wie der Balken im Hause.*"

". . . they are indeed contained in the definitions, but rather in the way that plants are contained in seeds, not in the way that timbers are contained in a building."

References

[1] Craig and Vaught, "Finite axiomatizability using additional predicates", JSL, **23** (1958), pp. 289–308.

[2] S.C. Kleene, "Finite Axiomatizability of theories in the predicate calculus using additional predicate symbols" Memoirs of the AMS, **10**.

[3] M. Makkai. Review of [2] JSL **36** (1971), pp. 334–335.

[4] Prof A.M. Pitts: Lecture notes for CS1B Computation Theory at `https://www.cl.cam.ac.uk/teaching/1920/CompTheory/lectures/lecture-10.pdf`.