

Languages and Automata 2020: notes on the sheets

Thomas Forster

November 21, 2020

1 Sheet 1

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—I'm glad i don't have to repeat it. The point here is that any function $\mathbb{N} \rightarrow \mathbb{N}$ 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 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 $\mathbb{N} \rightarrow \mathbb{N}$ 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.

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 dragons¹.

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 $\mathbb{N} \rightarrow \mathbb{N}$. 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

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

¹But no dragon icon; i looked for L^AT_EX dragon icons but i couldn't find one. If you know of one do please el me.

whether or not a *given function declaration* (which is a piece of syntax) matches the template of primitive recursion. So the first thing to do in our quest for E is to collect all the primitive recursive function declarations. That looks like a candidate for the desired set E , but remember that for the purposes of this course f_n is the function-in-intension computed by the *machine* with code number n , not the function computed by the *declaration* with code n^2 . Then we reflect that there is a finite deterministic procedure for obtaining—from a primitive recursive function declaration—a register machine that will compute the function thus declared. Not every register machine is obtained in this way but it doesn't matter. (For example register machines that have silly extra registers that can never be reached are never the result of this process). Next consider the set of all register machines that are obtained in this way. The set E that we want is the set of all such machines—or rather their codes. E is certainly r.e. (semidecidable); it might even be decidable. It depends on whether or not I can tell, by looking at a register machine, whether or not it is obtained from a primitive recursive function declaration as above. If the process of obtaining a register machine from a primitive recursive function declaration is smooth enough we may be able to run it backwards. I'm not making any primroses.

Q11

The point being made here 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 numbers. So 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 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 \perp)—0 and 1 often being reserved for this purpose. The connectives \wedge , \vee , \neg , \rightarrow etc then become 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.

²The point is that altho' you can tell by looking at a function declaration whether it invokes minimisation or not, you can't do the same for a register machine

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 products³, 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*?

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.

Some of you—much to your credit—think there must be something wrong with this, 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 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. 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 HALTING set. So you want the complement of the HALTING set to consist of numbers that have passed infinitely

³He'd been thinking about category theory

many tests. If $\{i\}(i)$ never halts, what tests has i passed? When you put it like that it becomes obvious: the n th 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 n th 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 $\{i : \{i\}(i) \text{ is still running after } n \text{ steps}\}$.

To be precise, we obtain the index set $I \subseteq \mathbb{N}$ as follows. For each n , obtain the set $A_n = \{i : \{i\}(i) \text{ is still running after } n \text{ steps}\}$ 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 *semidecidable* (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 HALTING 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 HALTING 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 HALTING set was, indeed, not a member of the HALTING 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 necessary.

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—heaven forbid!—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 **HALTING** 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 **HALTING** problem you consider the function that throws away its input and then runs p on i .

(b) (second part)

To use an oracle that detects functions that **HALT** on infinitely many inputs to solve the **HALTING** 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.

(c) Obviously not. If any of them were, you would be able to solve the **HALTING** 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*⁴ for a function f is an engine that runs a machine for f on all possible inputs in zigzag parallel with itself, so that you can put it in the corner of your room and watch it emit values of f while you relax.

Question 6

Well, the empty set is recursive, so for the first part find m s.t. the m th 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.

⁴This is *not* standard terminology.

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 HALTs 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

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

There is a connection here with

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 $\mathbf{B} = \lambda f g x. g x \mathbf{I} (f(gx))$

Assuming the fact about normal order reduction mentioned on slide 115, show that if partial functions $f, g \in \mathbb{N} \rightarrow \mathbb{N}$ 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 $\mathbf{B F G}$ ”

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: $g : \mathbb{N} \rightarrow \mathbb{N}$ 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 ground⁵ 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.

⁵This 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!

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, \rightarrow_{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 \rightarrow_{NOR} starting from $\mathbf{G} x$ is infinite;

but then by the first rule for generating a step of \rightarrow_{NOR} out of an application, we get an infinite sequence of steps of \rightarrow_{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,

$$(\lambda x.M)M' \rightarrow_{\text{NOR}} M[M'/x]$$

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

$$(\lambda x.N)M' \rightarrow_{\text{NOR}} N[M'/x]$$

?]

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 $2n$ or $2n+1$ depending on whether you are appending a '0' or a '1'. (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 $n \bmod 3$. On the other hand if you append on the left instead of the right, you need to know not only $n \bmod 3$, 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.

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, but they now mean (i) congruent to 1 and of even length, or congruent to 2 and odd length; (ii) congruent to 2 and of even length, or congruent to 1 and odd length; (iii) congruent to 0.

Can that be right? What happens if you are in state (iii) and receive a 1?

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. (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.

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 $0 \bmod 3$, $1 \bmod 3$, and $2 \bmod 3$ 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)⁶

⁶What 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 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

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 $R_{1,1}^{(3)}$. Recall we have the recursive definition

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

So thus we are looking for

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

We build this up inductively:

$$R_{1,1}^{(0)} = \epsilon + \mathbf{0}$$

$$R_{1,2}^{(0)} = \mathbf{1}$$

$$R_{1,3}^{(0)} = \emptyset$$

$$R_{2,1}^{(0)} = \mathbf{1}$$

$$R_{2,2}^{(0)} = \epsilon$$




$$R_{2,3}^{(0)} = \mathbf{0}$$

$$R_{3,1}^{(0)} = \emptyset$$

$$R_{3,2}^{(0)} = \mathbf{0}$$

$$R_{3,3}^{(0)} = \epsilon + \mathbf{1}$$

Now for the next step:

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?

If Zach Weber was here he would probably say that there is a point to be made about the *ex falso*. And he’s probably right.

$$\begin{aligned}
R_{1,1}^{(1)} &= R_{1,1}^{(0)} + R_{1,1}^{(0)}(R_{1,1}^{(0)})^* R_{1,1}^{(0)} = \dots = \mathbf{0}^* \\
R_{1,2}^{(1)} &= R_{1,2}^{(0)} + R_{1,1}^{(0)}(R_{1,1}^{(0)})^* R_{1,2}^{(0)} = \dots = \mathbf{1} + \mathbf{0}^* \mathbf{1} \\
R_{1,3}^{(1)} &= R_{1,3}^{(0)} + R_{1,1}^{(0)}(R_{1,1}^{(0)})^* R_{1,3}^{(0)} = \dots = \emptyset \\
R_{2,1}^{(1)} &= R_{2,1}^{(0)} + R_{2,1}^{(0)}(R_{1,1}^{(0)})^* R_{2,1}^{(0)} = \dots = \mathbf{1} + \mathbf{1}(\mathbf{0})^* \\
R_{2,2}^{(1)} &= R_{2,2}^{(0)} + R_{2,1}^{(0)}(R_{1,1}^{(0)})^* R_{2,2}^{(0)} = \dots = \epsilon + \mathbf{1}(\mathbf{0})^* \mathbf{1} \\
R_{2,3}^{(1)} &= R_{2,3}^{(0)} + R_{2,1}^{(0)}(R_{1,1}^{(0)})^* R_{2,3}^{(0)} = \dots = \mathbf{0} \\
R_{3,1}^{(1)} &= R_{3,1}^{(0)} + R_{3,1}^{(0)}(R_{1,1}^{(0)})^* R_{3,1}^{(0)} = \dots = \emptyset \\
R_{3,2}^{(1)} &= R_{3,2}^{(0)} + R_{3,1}^{(0)}(R_{1,1}^{(0)})^* R_{3,2}^{(0)} = \dots = \mathbf{0} \\
R_{3,3}^{(1)} &= R_{3,3}^{(0)} + R_{3,1}^{(0)}(R_{1,1}^{(0)})^* R_{3,3}^{(0)} = \dots = \epsilon + \mathbf{1}
\end{aligned}$$

Recall that we need only build $R_{1,1}^{(2)}$, $R_{1,3}^{(2)}$, $R_{3,3}^{(2)}$ and $R_{3,1}^{(2)}$ in the final stage.

So:

$$\begin{aligned}
R_{1,1}^{(2)} &= R_{1,1}^{(1)} + R_{1,2}^{(1)}(R_{2,2}^{(1)})^* R_{2,1}^{(1)} = \dots = \mathbf{0} + (\mathbf{0}^* \mathbf{1})(\mathbf{1}(\mathbf{0})^* \mathbf{1})^*(\mathbf{1}(\mathbf{0})^*) \\
R_{1,3}^{(2)} &= R_{1,3}^{(1)} + R_{1,2}^{(1)}(R_{2,2}^{(1)})^* R_{2,3}^{(1)} = \dots = (\mathbf{0}^* \mathbf{1})(\mathbf{1}(\mathbf{0})^* \mathbf{1})^* \mathbf{0} \\
R_{3,3}^{(2)} &= R_{3,3}^{(1)} + R_{3,2}^{(1)}(R_{2,2}^{(1)})^* R_{2,3}^{(1)} = \dots = \epsilon + \mathbf{1} + \mathbf{0}(\mathbf{1}(\mathbf{0})^* \mathbf{1})^* \mathbf{0} \\
R_{3,1}^{(2)} &= R_{3,1}^{(1)} + R_{3,2}^{(1)}(R_{2,2}^{(1)})^* R_{2,1}^{(1)} = \dots = \mathbf{0}(\mathbf{1}(\mathbf{0})^* \mathbf{1})^*(\mathbf{1} + (\mathbf{0})^*)
\end{aligned}$$

Finally, we get

$$R_{1,1}^{(3)} = (\mathbf{0} + (\mathbf{0}^* \mathbf{1})(\mathbf{1}(\mathbf{0})^* \mathbf{1})^*(\mathbf{1}(\mathbf{0})^*)) + ((\mathbf{0}^* \mathbf{1})(\mathbf{1}(\mathbf{0})^* \mathbf{1})^* \mathbf{0})(\mathbf{1} + \mathbf{0}(\mathbf{1}(\mathbf{0})^* \mathbf{1})^* \mathbf{0})^*(\mathbf{0}(\mathbf{1}(\mathbf{0})^* \mathbf{1})^*(\mathbf{1} + (\mathbf{0})^*)))$$

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:

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

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.

Question 11

The 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. Every regular language is the set of strings that cause some DFA to be in an 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.

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 $\mathbb{N} \rightarrow \{\text{yes}, \text{no}\}$ which tells you whether or not a natural number is a code for a 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 grammar⁷) and returns code for the total computable characteristic function as above. So yes, DFAs are weaker than register machines: they have no memories⁸. 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!

⁷No time for regular grammars in this course but if you want to pursue this stuff you will need to know about regular grammars.

⁸Tho' see the earlier discussion of what DFAs can remember.