# Languages and Automata sheet 1; a Riff

Thomas Forster

November 5, 2020

## 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 it was a character-forming experience that i was glad to have had but glad i don't have to repeat. 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.

## 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.

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

## Q5

The point of Question 5 is to make you think about *zigzagging*, a strategy which will be useful in the weeks to come.

## Q6 and Q8

Questions 6 and 8 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. I'm pretty sure that there is one but i can't put my hand on it. Here be dragons[1].

[1] But no dragon icon; i looked for LaTeX dragon icons but i couldn't find one.

## 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. 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, whereaas 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

Question 9 makes two points. One is that it's pretty obvious how to obtain (from the code $m$) the code for the machine that computes $m(x)$ and then adds 1 to it. 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

Question 10 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. 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. $E$

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

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 in Question 11 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 $\lambda$-calculus, and $\lambda$-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 $\bot$)—0 and 1 often being reserved for this purpose. The connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$ etc then become primitive recursive functions.