CHAPTER 5

What's a computer, anyway?

We will now turn our attention to **decision problems** and **algorithms**. I kind of hinted at decision problems, when after the proof of the completeness theorem for propositional logic I discussed a "Decidability Theorem". The keen-eyed amongst you may have noticed that in many ways the Completeness Theorem for first-order logic was kind of similar to that of propositional logic, but when we finished the proof of that we sure didn't seem to mention any decidability results.

Intuitively, a decision problem is some kind of "formal yes or no question". An example of a decision problem is:

GIVEN: A graph G and a natural number k.
DECIDE: Is there a k-colouring of G.

another decision problem, the one that I was just talking about in the previous paragraph is the following:

GIVEN: A complete first-order theory T and a sentence ϕ.
DECIDE: Does $T \vdash \phi$?

We will for a minute say that such a decision problem is **decidable** if there is a computer program that given <u>any</u> instance of the problem can tell us if the answer is YES or NO. At this point if we want to be serious in our discussion of decision problems we should really decide what a computer is.

Well... I'm sure you all know what a computer is. In fact, when I learned logic everyone knew what a computer was. That being said, I'm pretty sure that when the professor that taught me logic learned logic computers were a more mysterious object. And knowing who taught that professor logic, I'm sure that when they were learning logic computers were really a thing that mostly logicians knew about. In any case we really need a (hardware-free) "mathematical model of computation". Soon we'll see that the exact model we pick doesn't reaaaaally matter, but let's take things one step at a time.

1. A step-by-step guide to computation

We'll start by discussing two abstract mathematical models of computation, just to see why this sucks.

1.1. Register Machines.

This model of computation is due to Minsky and feels a bit closer to a modern-day computer than a Turing Machine (although the rule of cool prevents me from writing a chapter on models of computation which does not define, even in passing, Turing machines). Anyway, a **register machine** is built out of two parts. We can think of them as the "hardware" part and the "software" part, respectively.

The *registers*:

- A (finite) set $R_1, \ldots, R_m$ of registers each of which may at any time hold some natural number.

The *instructions*:

- A finite set $\mathtt{L}_1, \ldots, \mathtt{L}_n$ of instructions.

Each instruction is given a label $\mathtt{L}_i \in \mathbb{N}$ and can be one of the following:

(1) $\mathtt{L}_i$: Let $R_j := R_j + 1$ then "Go to instruction $\mathtt{L}_j$".

(2) $\mathtt{L}_i$: <u>If</u> $R_j = 0$
  <u>then</u> "Go to instruction $\mathtt{L}_l$"
  <u>else</u> "Let $R_j := R_j - 1$" and go to $\mathtt{L}_k$.

(3) $\mathtt{L}_n$: $\mathtt{HALT}$.[1]

This is already starting to look a bit like a computer program in a very basic language. Indeed, this is pretty much how low-level code works to this day, it sees the contents of the RAM of our computer (or whatever), i.e. the registers, and has a very limited set of instructions.

Pictorially, we can think of instructions of the form (1) and (2) as:

$$\begin{tikzcd} \mathtt{L}_i \arrow[r, "R_j+1"] & \mathtt{L}_k \end{tikzcd} \qquad \begin{tikzcd} \mathtt{L}_i \arrow[r, "R_j-1"] \arrow[dr, dashed, bend right, "R_j=0"'] & \mathtt{L}_k \\ & \mathtt{L}_l \end{tikzcd}$$

[1]Halt is just posh for stop running. We'll always assume that this is the last instruction of our machines.

In part, **Church's Thesis** (see later) says that any computer program, in any programming language, can be translated into the program of a register machine. More on that in a bit. For some terminology, to see how computation happens:

Register machines have finite programs, hence they will always use a finite number of registers in any single computation. In fact, the number of registers a machine can use is bounded above by the number of instructions it has (namely a register machine with n instructions can access at most n registers). However it does not hurt to imagine that our register machines have infinitely many registers at their disposal.[2] Thus, we can identify register machines with their program, and can forget about the number of registers. At some point when the distinction will start to matter, we will assume that register machines with n instructions only use registers $R_1, \ldots, R_n$. Then:

- A **state** of a register machine is a sequence $\underline{s} = (n_1, n_2, \ldots)$ of natural numbers, where we understand that n_i is the content of register R_i. By assumption, there is some $N \in \mathbb{N}$ such that for all $n \geq N$, we have that that $s_n = 0$.
- A **configuration** of a register machine $R = (\mathtt{L}_1, \ldots, \mathtt{L}_n)$ is a pair $(\underline{s}_i, \mathtt{L}_{j_i})$, where $\underline{s}_i$ is a state and $\mathtt{L}_{j_i}$ is one of the instructions of R.
- A **run** of a register machine R is a possibly infinite sequence of configurations $(\underline{s}_1, \mathtt{L}_1), (\underline{s}_2, \mathtt{L}_{j_2}), \ldots$, starting from $(\underline{s}_1, \mathtt{L}_1)$ where state $\underline{s}_{i+1}$ is obtained from state $\underline{s}_i$ by applying instruction $\mathtt{L}_{j_i}$, which also tells us to go to instruction $\mathtt{L}_{j_{i+1}}$. More precisely, if $\underline{s}_i$ is $(n_1, n_2, \ldots)$ then we obtain $(\underline{s}_{i+1}, \mathtt{L}_{j_{i+1}})$ as follows:
  - If $\mathtt{L}_{j_i}$ is:
    Let $R_j := R_j + 1$ then "Go to instruction $\mathtt{L}_j$",

    then
    $\underline{s}_{i+1}$ is $(n_1, \ldots, n_{j-1}, n_j + 1, n_{j+1}, \ldots)$ and $\mathtt{L}_{j_{i+1}}$ is $\mathtt{L}_j$.
  - If $\mathtt{L}_{j_i}$ is:
    ```
    If $R_j = 0$
        then "Go to instruction $\mathtt{L}_l$"
        else "Let $R_j := R_j - 1$" and go to $\mathtt{L}_k$.
    ```
    then

    * in the former case $\underline{s}_{i+1}$ is $\underline{s}_i$ and $\mathtt{L}_{j_{i+1}}$ is $\mathtt{L}_l$. and

[2]Think formulas and variables!

  - $*$ in the latter case $\underline{s}_{i+1}$ is $(n_1, \dots, n_{j-1}, n_j - 1, n_{j+1}, \dots)$ and $\mathtt{L}_{j_{i+1}}$ is $\mathtt{L}_j$.

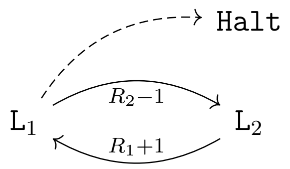
- If $\mathtt{L}_{j_i}$ is:
  HALT,

  then
  The machine stops the computation.

We say that a run **terminates** or **halts** if $\mathtt{L}_{j_i} = \mathtt{HALT}$ for some $i \in \mathbb{N}$. In that case, we call $\underline{s}_i$ the **final state** of the run. We, of course, call $\underline{s}_1$ the **initial state** of the run.

That's all good and well, but I'm pretty sure we should do some examples at this point.

Example 1.1.1. Here is a register machine that adds the contents of R_2 to R_1:


$$\begin{tikzcd} & \mathtt{Halt} \\ \mathtt{L}_1 \arrow[ur, dashed, bend left] \arrow[rr, bend left, "R_2-1"] & & \mathtt{L}_2 \arrow[ll, bend left, "R_1+1"] \end{tikzcd}$$

The instructions of this register machine are of course:

- $\mathtt{L}_1$: If $R_2 = 0$ then "Go to $\mathtt{L}_3$" else "Let $R_2 := R_2 - 1$" and go to $\mathtt{L}_2$.
- $\mathtt{L}_2$: Let $R_1 := R_1 + 1$ then "Go to $\mathtt{L}_1$".
- $\mathtt{L}_3$: HALT.

To make sure we understand this, let's do a run. Suppose that the initial state of our register machine is $(2, 2, 0, 0, \dots)$. Then a run from this state would be:

- $((2, 2, 0, 0, \dots), \mathtt{L}_1)$,
- $((2, 1, 0, 0, \dots), \mathtt{L}_2)$,
- $((3, 1, 0, 0, \dots), \mathtt{L}_1)$,
- $((3, 0, 0, 0, \dots), \mathtt{L}_2)$,
- $((4, 0, 0, 0, \dots), \mathtt{L}_1)$,
- $((4, 0, 0, 0, \dots), \mathtt{HALT})$,

This is already getting rather annoying to write. Let's agree on some conventions:

We will write:

$$\mathtt{L}_i : (R_j, +, \mathtt{L}_k)$$

as shorthand for:

$\mathtt{L}_i$: Let $R_j := R_j + 1$ then "Go to instruction $\mathtt{L}_k$".

and

$$\mathtt{L}_i : (R_j, -, \mathtt{L}_k, \mathtt{L}_l)$$

as shorthand for:

$\mathtt{L}_i$: <u>If</u> $R_j = 0$
<u>then</u> "Go to instruction $\mathtt{L}_l$"
<u>else</u> "Let $R_j := R_j - 1$" and go to $\mathtt{L}_k$.

So, in this notation, the register machine from before becomes:

- $\mathtt{L}_1 : (R_2, -, \mathtt{L}_2, \mathtt{L}_3)$
- $\mathtt{L}_2 : (R_1, +, \mathtt{L}_1)$
- 3 : HALT

To reiterate a point made earlier, register machines have many registers but only a few of them matter. We'll keep track of this as follows: Given a register machine R, we will think of a certain initial segment of the registers as the **input registers** and we'll think of the remaining registers as the **scratch registers**.

Definition 1.1.2. Let $f : \mathbb{N}^m \to \mathbb{N}$ be a function. We say that f is *register machine computable* if there is a register machine $R = (\mathtt{L}_1, \ldots, \mathtt{L}_n)$ with input registers $R_1, \ldots, R_m$ such that for all $x_1, \ldots, x_m \in \mathbb{N}^m$ the run of R with initial state $(x_1, \ldots, x_m, 0, 0, \ldots)$ terminates with final state $(f(x_1, \ldots, x_m), 0, 0, \ldots)$.[3]

Let's look at some register machine computable functions:

Example 1.1.3. Here's a program that computes $n_1 - n_2$ if $n_1 \geq n_2$ and 0 otherwise:

[3]That our register machines clean up after themselves, i.e. delete all contents of $R_2, R_3, \ldots$ is a purely cosmetic assumption, but it makes life easier. Observe that a register machine with n instructions will access at most $n-1$ registers. Thus, given a register machine with n instructions we can append to it register machine code that replaces HALT with instructions that start at R_2 and empties the contents of all $n-2$ registers that were used in the computation, before HALTing.

$$\begin{tikzcd}
& \texttt{HALT} \\
\texttt{L}_1 \arrow[r, bend left, "R_2{-}1"] \arrow[ur, dashed, bend left] & \texttt{L}_2 \arrow[l, bend left, "R_1{-}1"] \arrow[u, dashed]
\end{tikzcd}$$

The function we defined in the example above is a rather important function when dealing with natural numbers, we call it **restricted subtraction** and denote it by $\dot{-}$. Explicitly, this is the function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined as follows:

$$x \mathbin{\dot{-}} y := \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise.} \end{cases}$$

Think of this function exactly in terms of the register machine we defined above. It subtracts from x as much of y as possible. It stops when there's no more of y to subtract or if by subtracting from x it's hit 0.

Let's see how we can put our register machines together to compute more complicated things:

Proposition 1.1.4. *The following functions are register machine computable:*

(1) The function $S : x \mapsto x + 1$.

(2) The constant function 0.

(3) For each $n \in \mathbb{N}$ and each $i \leq n$ the function:

$$\begin{aligned} \mathbb{N}^n &\to \mathbb{N} \\ (x_1, \ldots, x_m) &\mapsto x_i \end{aligned}$$

Proof.

(1) Well this one is easy: $\texttt{L}_1 : (R_1, +, \texttt{L}_2)$, $\texttt{L}_2 : \texttt{HALT}$.

(2) This one is also easy: $\texttt{L}_1 : (R_1, -, 1, \texttt{L}_2)$, $\texttt{L}_2 : \texttt{HALT}$.

(3) Here we may have to do a bit more work. Well, for $n = 1$ the program $\texttt{L}_1 : \texttt{HALT}$ does the trick. For $n \geq 1$ we need to write a program that empties R_1 and then transfers the contents of R_i to R_1. This can be done as follows:

$$\begin{tikzcd}
 & & \mathtt{HALT} \\
\mathtt{L}_1 \arrow[loop above, "R_1{-}1"] \arrow[r, dotted] & \mathtt{L}_2 \arrow[ur, dashed, bend left] \arrow[r, bend left, "R_i{-}1"] & \mathtt{L}_3 \arrow[l, bend left, "R_1{+}1"]
\end{tikzcd}$$

□

Exercise 1.1.5. Prove that $+$ is register machine computable.

That was pretty fun. Now that we have a few register machine computable functions, let's see some ways that we can put them together. From now on, we'll be a little bit informal when discussing register machines. Rather than writing out their full code or their full diagram, we'll simply describe their procedure (think pseudo-pseudocode) and trust that we could write them out formally.

Proposition 1.1.6 (Closure under composition)**.** *Let* $f_1, \ldots, f_m : \mathbb{N}^n \to \mathbb{N}$ *and* $g : \mathbb{N}^m \to \mathbb{N}$ *be register machine computable. Then, so is:*

$$\begin{aligned} h : \mathbb{N}^n &\to \mathbb{N} \\ (x_1, \ldots, x_n) &\mapsto g(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)). \end{aligned}$$

For short, we will denote h *by* $g(f_1, \ldots, f_m)$*.*

Proof. Let $R^1, \ldots, R^m$ be the register machines computing $f_1, \ldots, f_m$ respectively and R the register machine computing g. We use the register machine from Example 1.1.1 to copy the contents of $R_1, \ldots, R_n$ to to $R_{n+1}, \ldots, R_{2n+1}$ and then run the computation of R^1 with all registers transposed n to the right. After this computation, R_{n+1} contains $f_1(x_1, \ldots, x_m)$. We repeat the same process, copying the contents of $R_1, \ldots, R_n$ to R_{n+2}, and then run the computation of R^2, with all registers transposed 2 to the right. After we have done this for each $i \leq m$, we have that R_{n+i} contains $f_i(x_1, \ldots, x_n)$. We then empty $R_1, \ldots, R_m$ and move the contents of R_{n+i} to R_i. We then run the computation of R. □

The proof of the above would be too mean to have you do in an exercise, but make sure you understand what the content of the few lines above is.

Proposition 1.1.7 (Closure under primitive recursion)**.** *Suppose that $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$ are register machine computable. Then, so is:*

$$h : \mathbb{N}^{k+1} \to \mathbb{N}$$
$$(x_1, \dots, x_{k+1}) \mapsto \begin{cases} f(x_1, \dots, x_k) & \textit{if } x_{k+1} = 0, \\ g(x_1, \dots, x_k, x_{k+1} - 1, h(x_1, \dots, x_k, x_{k+1} - 1)) & \textit{otherwise.} \end{cases}$$

Before we prove this, let's try to think about what this says. We have shown that $+$ is register machine computable. By closure under composition, the function $(x_1, x_2, x_3) \mapsto x_1 + (x_2 + x_3)$ is also register machine computable. Let's define a function:

$$h : \mathbb{N}^2 \to \mathbb{N}$$
$$(x_1, x_2) \mapsto \begin{cases} 0 & \text{if } x_2 = 0, \\ x_1 + x_1 + h(x_1, x_2 - 1) & \text{otherwise.} \end{cases}$$

We have just shown that multiplication is register machine computable!

Exercise 1.1.8. Using ideas similar to the above prove that exponentiation is register machine computable.

In the context above, we say that h is **defined by recursion** with **initial condition** f and **recursive step** g.

Proof (Sketch). This is an exercise in transforming recursive definitions into "while loops". Register machines can easily be seen to be able to simulate while loops. Indeed, the loop:

```
while $n > 0$ {
    Do a register machine computable procedure (that may depend on $n$)
    let $n = n - 1$
}
```

Is almost by definition:

- $\mathtt{L}_1$: If $R_n = 0$ go to $\mathtt{L}_k$ otherwise $R_n = R_n - 1$ and go to $\mathtt{L}_2$
- $\mathtt{L}_2$: Let $R_n = R_n + 1$ go to $\mathtt{L}_3$
- $\mathtt{L}_3$: Run register machine code that terminates at instruction $\mathtt{L}_{k-1}$. Replace instruction $\mathtt{L}_{k-1}$ with:

- $\mathtt{L}_{k-1}$: If $R_n = 0$ go to $\mathtt{L}_k$ otherwise $R_n = R_n - 1$ and go to $\mathtt{L}_1$
- $\mathtt{L}_k$: HALT

An easy adaptation of the argument above let's us write loops of the form:

```
while $n < x$ {
    Do a register machine computable procedure (that may depend on $n$)
    let $n = n + 1$
}
```

for some fixed $x \in \mathbb{N}$ in terms of register machine instructions.

Here's a way to compute h. Move the contents of $R_1, \ldots, R_{k+1}$ to $R_3, \ldots, R_{k+3}$ and empty R_1 and R_2. In the pseudocode below I will identify $x_1, \ldots, x_k, x_{k+1}$ with the initial inputs of R and n_i with the contents of register R_i;

```
Compute $f(x_1, \ldots, x_k)$
Store the result at register $R_1$
while $n_2 < x_{k+1}$ {
    Compute $g(x_1, \ldots, x_k, \ldots, n_2, n_1)$
    let $R_2 = R_2 + 1$
    Store the result at register $R_1$
}
```

It is easy to check that this computes h. □

1.2. Turing Machines. Ah the rule of cool. Ever since the Imitation Game[4] everyone knows about Alan Turing (which is a good thing), but also everyone expects to hear something about Turing Machines when learning logic (which is maybe a less good thing?). Let's just give a rather informal description of what a **Turing Machine** is composed of:

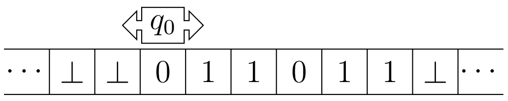
- A **tape** (broken up in blocks) on which we can read or write letters from a **fixed alphabet**.

[4]Or like ever since the British government decided to release the Bletchley files in the 70s and also that being gay was okay (which unsurprisingly happened MUCH later). Like genuinely, the latter happened in your lifetimes.

- A **head** which can read and write on the tape.
- A set of **states** (which depend on what the head has read on the tape), with a designated **initial state** and a designated set of **final states**.
- A **transition function**, a partial map from the set of pairs consisting of non-final states and possible inputs (stuff the head may be reading) to the set of states possible outputs (stuff the head could write) and an instruction to the head to move *left* or *right* on the tape.

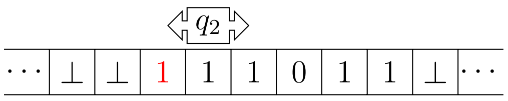
You can (and should) think of a Turing machine as a version of cavepeople computer!

Here's a way of drawing a Turing Machine, with alphabet $\{0, 1\}$ (the $\perp$ symbol meaning an unused square), and the head in state $q_0 \in Q$:



The precise running of a Turing machine is more or less the obvious thing (it starts at an initial state with some input, runs its transition function, and maybe reaches a final state, at which point, whatever is on the tape is its output).

If, say in the example above, the transition function said that on state q_0 if the input is 0, then "write 1 and move to the right, and go to state q_2 we'd have:



The real important thing, is the following fact, which I implore you to think about (even without a formal definition of a Turing machine):

FACT. *A function is register machine computable if and only if it is Turing machine computable.*[5]

End of digression

[5]This means what you think it means.