Abacus Machines

An Abacus machine is a (hypothetical) device that has the following components:

1. A finite number of *registers* R_1, …,R_k, each of which can hold an arbitrarily large number of *beads*.
2. A program, which is a finite set of instructions.

There are two kinds of instructions for Abacus machines:

1. n+: Add a bead to register R_n, and go to some new instruction.
2. n-: See if there is at least one bead in register R_n. If so, remove one bead from that register, and go to some new instruction I. If not, do nothing, but go to some instruction J, where J may be different from I.

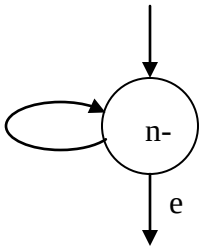
While there are a number of differences between Abacus machines and Turing machines, the fact that Abacus machines do not have any internal states defined, whereas Turing machines do, is *not* one of those differences. To see this, notice that Abacus machines need internal states to keep track of where is it in the program, just like Turing machines do. However, these internal states are usually not defined as part of an Abacus machine, because the most common way to depict an Abacus machine is through the use of a flow graph, where each of the nodes represents some instruction, and where the arrows indicate what the next instruction is going to be (possibly depending on whether some register is empty or not). Such flow graphs are therefore not unlike Minsky-style flow graphs for Turing machines. In fact, notice that in the latter flow graphs, there is no mention of internal states anymore. So, if we wanted to, we could define Turing machines without any explicit mentioning of any internal states either.

We can also draw a parallel between the cells of a Turing machine and the registers of an Abacus machine: both are places where we can put something. However, while Turing machines have an infinite number of cells, Abacus machines have only finitely many. On the other hand, there is only one of finitely many things that a Turing machine can put into each of its cells (there are only finitely many symbols), whereas the contents of each register in an Abacus machine is one of infinitely many possibilities.

Notice that these differences have implications for how each of the machines can be used in an effective way, all of which are reflected in the nature of the instructions for each of the machines. In a Turing machine, it is assumed that the contents of a cell can be inspected in one effective step, but in an Abacus machine, the specific contents of any register (i.e. how many beads are in a register) cannot be ‘grasped’ in one step. On the other hand, a Turing machine needs to use a head to effectively deal with its infinite number of cells, but since there are only finitely many registers in an Abacus machine, the machine can be supposed to be able to directly access any of its registers.

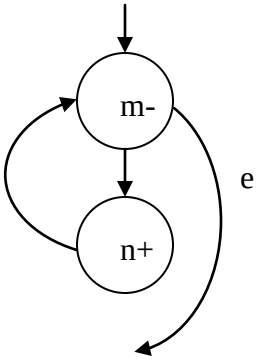
Example: Emptying, Copying, Adding

Suppose we want to completely empty some register R_n. The following flow graph will do so:



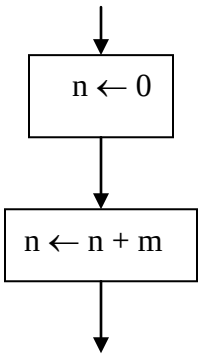
The program will start by following the incoming arrow, and then repeat the instruction n- until register R_n is empty, in which case the arrow with the e (for empty) is followed at which point (since this arrow doesn't point to any instruction) the machine halts. We will write the result of this flow graph as $n \leftarrow 0$.

The following flow graph adds a number of beads to register R_n that is equal to the number of beads in register R_m:



We can write the result of this graph as $n \leftarrow n + m$.

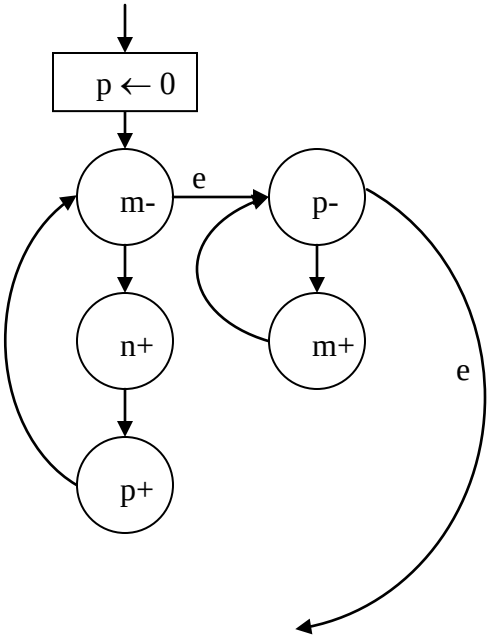
The following flow graph copies the number of beads that are in register R_m into register R_n:



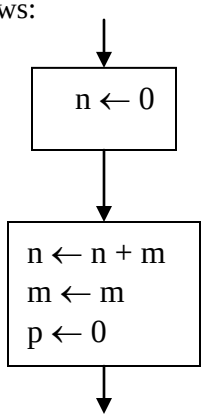
In this graph, which can be written as $n \leftarrow m$, the blocks are used to substitute for the corresponding flow diagrams.

Notice that in the last two examples, register R_m will be empty when the machine is done. For that reason, the last two flow graphs may more accurately be represented as:

If we want to make sure that R_m still has m beads when the machine is finished, we have to do something else. One easy way to do this is to use a third register, say R_p, which we use to temporarily store the contents of R_m. For example, the following flow graph would be a possible flow graph to add the contents of m to n, while keeping the contents of m:

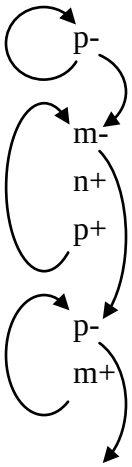


The graph for copying the contents from register R_m to register R_n, while keeping the contents in register R_m the same is now as follows:



Abacus Machines and Programming Languages

The nice thing about Abacus machines is that they relate to programming languages much more so than Turing machines. We already see this in notations like $n \leftarrow n + m$, and the immediately intuitive use of subroutines. However, we can also write Abacus machine programs in the following way:



In this notation, the next instruction is always the instruction below, except as indicated otherwise by an arrow. This is why the n- kind of operation is sometimes referred to as a “decrement or jump” instruction. Moreover, taking one more simple step, we get the following program, which resembles the kind of program we are familiar with:

```
while (p>0) {p-;}
while (m>0) {m-; n+; p+;}
while (p>0) {p-; m+;}
```