Chapter 5, §5.2: Every abacus-computable function is Turing-computable

**Method of proof:** show how the graph of any abacus-computable function can be transformed into the flow graph of a Turing machine that computes the same function.

1. **Definition of an abacus-computable function**

Suppose $A$ is any abacus. Then $A$ defines (computes) $f$ as follows:

1. Start with $x_1 = [1], \ldots, x_r = [r], \text{ and } 0 = [r+1] = [r+2] = \ldots$
2. Specify a solution register $n$. If the computation halts with $y = [n]$, then $f(x_1, \ldots, x_r) = y$. (Note: the other registers don’t have to be empty when machine halts.)
3. If the computation never halts, then $f(x_1, \ldots, x_r)$ is undefined.

Given an abacus $A$, there are two parameters needed to determine a function: $r$ (the number of arguments) and $n$ (the index of the solution register). Write $A^n_r$ for the function computed by $A$ that has $r$ arguments and solution in register $n$.

**Example:** Let $A$ be the following definite version of the factorial machine.

```
1 → 3
↓
[1] → 2
↓
2-→ e
↓
[1][3] → 3
↓
1-→ e
```

$A^1_3$ is the factorial function: $A^1_3(x) = x!.$  
$A^1_4(x) = 0$ for all $x$, and similarly $A^1_5(x) = 0$…

$A^2_3(x, y) = x!$, $A^3_3(x, y, z) = x!$, and so on.

2. **Outline of Solution**

**Problem:** Given an abacus $A_n$, with solution register specified as $R_n$, find a Turing machine $M$ such that for each $r$ (# of arguments), $M$ defines the same function of $r$ arguments as $A^n_r$.

**A) Register/Block Correspondence.**

The registers, taken in order, correspond to blocks on the tape separated by a single blank, taken in order. For instance:
corresponds to
\[
\begin{array}{cccccc}
1 & 0 & 4 & 0 & 2 \\
R_1 & R_2 & R_3 & R_4 & R_5 \text{ (and 0 elsewhere)};
\end{array}
\]

- If \( n \neq 0 \), a register containing \( n \) is represented by a block of \( n+1 \)'s
- If \( n = 0 \), a register containing \( n \) is represented by a blank or by a single 1. The single 1 is mandatory if there are any 1’s further to the right
- Two blanks in a row signify no further 1’s on the tape.

B) Three Procedures

1. Replace \( n+ \) nodes with the following TM graph (always starting in std. Config.):

   move to the blank to the right of block \( n \), replacing B with 1 for empty registers along the way
   \[
   \begin{array}{c}
   \text{↓} \\
   \text{write a 1} \\
   \text{↓} \\
   \text{shift any remaining blocks over to the right by 1} \\
   \text{↓} \\
   \text{return to std. final position}
   \end{array}
   \]

   N.B. The replacement is needed in case \( n \) is beyond any of the registers where initial non-zero information was stored.

   **Special cases:**
   - there are no remaining blocks
   - 0 arguments

   Turing machine: Figs. 5-9, 5-10.

2. Replace \( n- / e \) nodes with the following TM graph

   \[
   \begin{array}{c}
   \text{Go to first 1 in nth block, replacing B by 1 for empty registers as needed} \\
   \text{↓} \\
   \text{If } [n] = 0, \text{ return to std. position} \\
   \text{↓} \\
   \text{If } [n] \neq 0, \text{ go to rightmost 1 in nth block and erase it.} \\
   \text{↓} \\
   \text{Skip over the blank and shift any remaining blocks back one square. If there are remaining blocks, repeat the procedure; if not, return to std. position.}
   \end{array}
   \]

   Turing machine: Fig. 5-11, 5-12
Result of steps 1 and 2: you get the right number of 1’s in the n’th block if the machine halts. But there may also be lots of other blocks of 1’s on the tape.

3. [After replacing all n+ and n- routines] Point all loose arrows to the initial node in a ‘mop-up’ graph that erases all but the n’th block of 1’s.

N.B. B.B&J ensure that the n-th block is also re-positioned so that the machine halts at the same square where it started. We won’t bother with this, but it is possible to do it.

To do this:

If n=1, erase all but the first block and halt in std. position.
   Move to end of first block.
   Erase each subsequent block until there are two blanks in a row; then return to std. position.

If n ≠ 1:
   Go to leftmost 1 of block n, erasing every 1 on the way
   Move to end of block n
   Erase subsequent blocks
   Return to std. position.

**Conclusion:** Every abacus-computable function is Turing-computable.