Abacus machines

Abacus machines: overview

- The notion of Turing-computability was developed before the age of high-speed digital computers.
- In contrast to Turing machines, computers today have **random-access storage**.
- An **abacus machine** is an idealized version of such modern computers.

Abacus machines: overview

- We shall prove that a function $N^k \longrightarrow N$ is abacus-computable if and only if it is Turing-computable.
- Abacus machines are easier to program than Turing machines; we shall take advantage of this fact and show the computability of e.g. multiplication.

Abacus machine: description

- An abacus machine has an enumerably infinite number of **registers** $R_1, R_2, R_3, \ldots$
- Each register can contain a non-negative integer.

Programs for an abacus machine

An **abacus program** is a **finite** list of commands:

1: $command_1$

2: $command_2$

3: $command_3$

$\vdots$

n: $command_n$

There are only two kinds of commands:

- i+; `goto` l
- `if` i=0 `then` { `goto` l_1 } `else` { i-; `goto` l_2 }

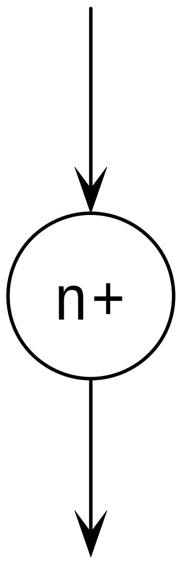
Meaning of the commands

- The command
  i+; `goto` l
  means: add 1 to register R_i and then go to line l.
- The command
  `if` i=0 `then` { `goto` l_1 } `else` { i-; `goto` l_2}
  means: if R_i contains 0, then goto line l_1; otherwise, subtract 1 from R_i and then go to line l_2.

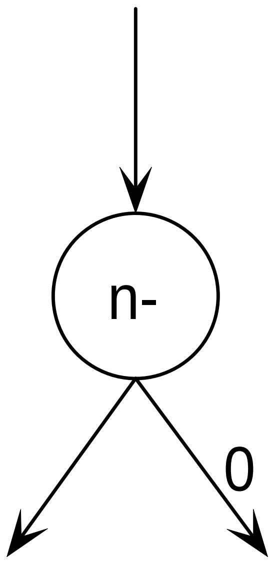
Abacus machines vs. real-life computers

- Real-life computers have only finitely many storage cells (e.g. RAM + hard disk).
- Not a real issue, because each abacus program uses only finitely many registers.
- More serious: the storage cells of real-life computers have limited size.
- But infinite registers make sense, because in a theoretical setting, there is no point in restricting register size arbitrarily (e.g. 16bit, 32bit, or 64bit).

Abacus programs as flow graphs



Add 1 to R_n.



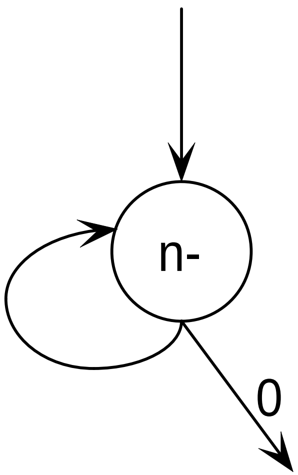
If R_n is 0, come out on the arrow marked “0”, otherwise, subtract 1 from R_n and come out on the other arrow.

Example: making R_n zero

0: `if` $n = 0$ `then` { `goto` 99 } `else` { n-; `goto` 0 }

We consider a `goto` to a missing line (e.g. line 99 in the program above) to be a halting command.

Flow graph:



Exercise

Define a reasonable way of coding abacus machines by natural numbers.

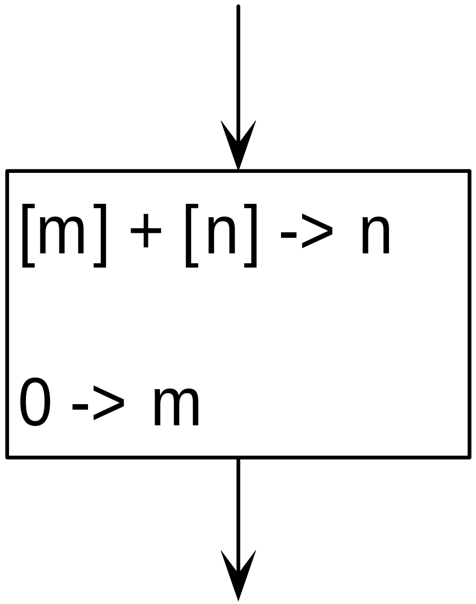
Addition

The program below puts $m + n$ into register n and makes m zero.

```
0: if m=0 then {goto 99} else {m-; goto 1}
1: n+; goto 0
```

(We assume that $m \neq n$.)

Addition: block diagram



Block diagram that summarizes the effect

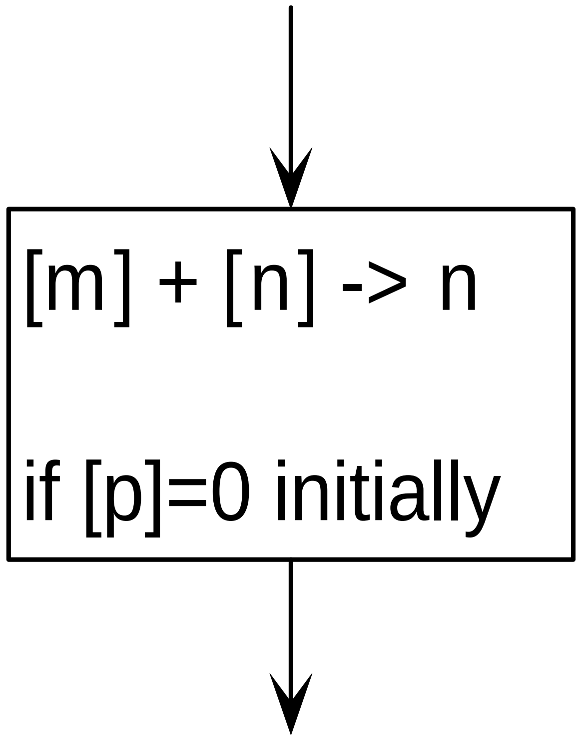
Addition without destroyed argument

The program below puts $m + n$ in n without destroying m.

```
0: if m=0 then {goto 3} else {m-; goto 1}
1: n+; goto 2
2: p+; goto 0
3: if p=0 then {goto 99} else {p-;goto 4}
4: m+; goto 3
```

if Register p differs from n and m and is initially zero.

Addition without destroyed argument



Block diagram

Multiplication

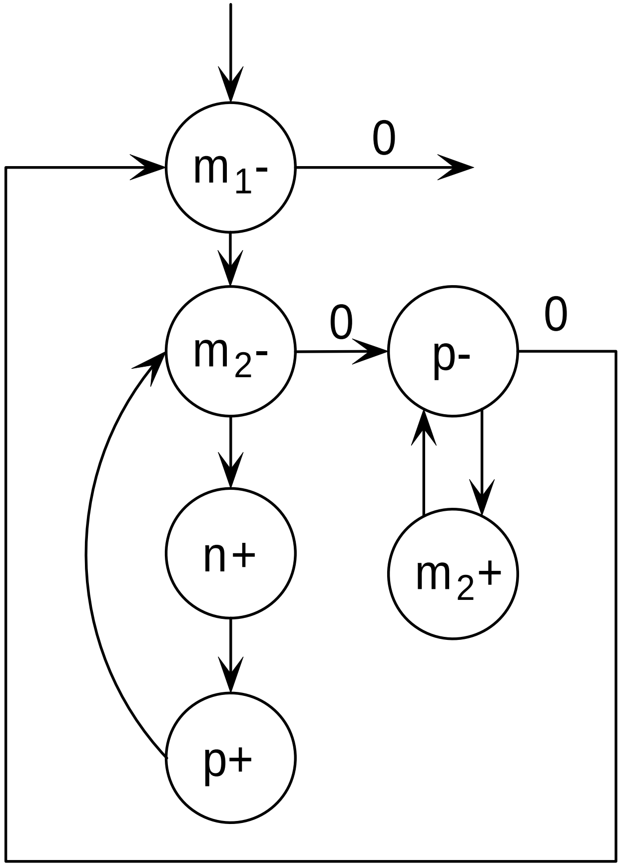
The program below adds $m_1 * m_2$ to n and empties m_1.

```
0: if m1=0 then {goto 99} else {m1-; goto 1}
1: [m2] + [n] -> n; goto 0
```

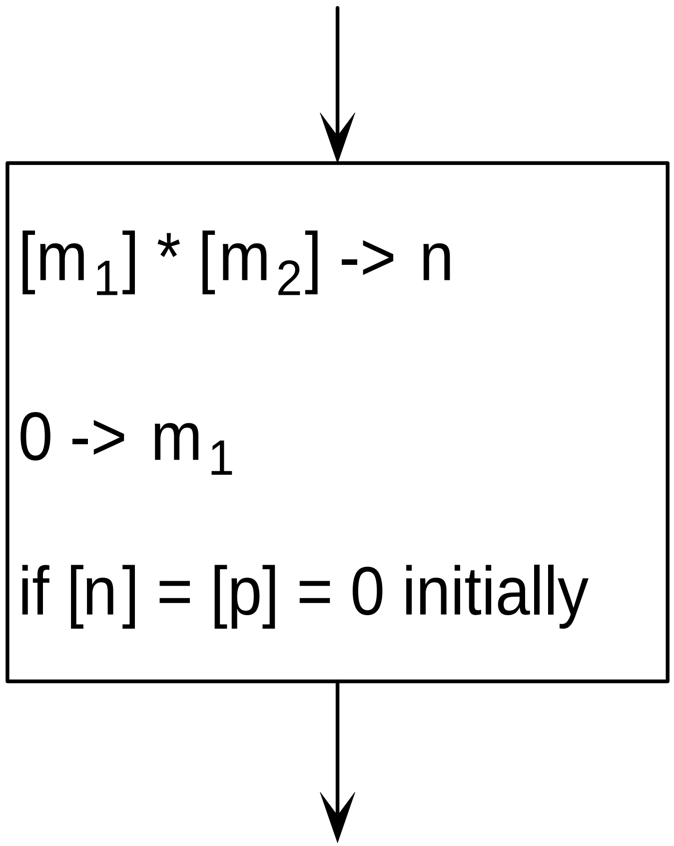
The "command" in line 1 is really a **macro**–that is, an abbreviation for an actual program (here: for the addition program seen previously).

Multiplication

Full flow graph:



Multiplication



Block diagram

Exercise

Define an abacus machine that copies Register m into Register n, without destroying m. (Note that the initial value of n might differ from 0.)

Exercise

- Given different registers x and y, define an abacus machine that puts $x \dot{-} y$ into x, where $\dot{-}$ is defined by

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

- Given mutually different registers x, y and z, define an abacus machine that puts $x \dot{-} y$ into z.

Exercise

The signum function is defined by letting

$$sg(x) = 1 \text{ if } x > 0$$
$$sg(x) = 0 \text{ otherwise.}$$

Define an abacus machine that puts $sg(x)$ into Register x.

Exercise

Let f be the function

$$f(x,y) = \begin{cases} 1 & \text{if } x < y \\ 0 & \text{otherwise.} \end{cases}$$

Given different registers x, y, and z, define an abacus machine that puts $f(x,y)$ into z.

Exercise

The **quotient** and the **remainder** when the positive integer x is divided by the positive integer y are the unique natural numbers q and r such that $x = qy + r$ and $0 \leq r < y$. Let the functions quo and rem be defined as follows: $rem(x, y) =$ the remainder of dividing x by y if $y \neq 0$, and $= x$ if $y = 0$; $quo(x, y) =$ the quotient of dividing x by y if $y \neq 0$ and $= 0$ if $y = 0$. Design abacus machines for rem and quo. (Hint: tackle rem first.)

Abacus-computable functions

Definition. A function $f : N^k \to N$ is called **abacus-computable** if there is an abacus machine M such that:

- If $f(x_1, x_2, \ldots, x_k) = y$, then M, starting with storage $R_1 = x_1, R_2 = x_2, \ldots, R_k = x_k$ and $R_i = 0$ for $i > k$, halts with $R_1 = y$.
- If $f(x_1, x_2, \ldots, x_k)$ is undefined, then M, starting with the same storage as above, never halts.

From abacus to Turing machine

Theorem. Every abacus-computable function is Turing-computable.

Proof: for every abacus machine that computes a function $f : N^k \to N$, we build a TM that also computes f. (The construction will take several slides.)

Minor change to TM's

Because we changed N to include 0, we modify the notion of Turing computability accordingly:

- From now on, a block $\ldots 00100 \ldots$ containing a single stroke on the tape of a TM represents no longer the number 1, but the number 0.
- More generally, a block of n strokes on the tape represents no longer n, but $n-1$.
- For example, the tape 11101101111 is now the standard initial configuration for the arguments $(2, 1, 3)$.

Three simulation stages

The TM will proceed in three stages:

- Initialization
- Simulation
- Cleanup

Initialization

- Suppose the abacus machine uses the registers $R_1, R_2, \ldots, R_n$.
- Then the initialization process extends the tape by blocks of single strokes, so that there is one block of strokes for every used register.
- For example, if the TM's initial tape is 11101101111 and the abacus uses registers $R_1, R_3, \ldots, R_6$, then the tape will become $1110110111101010 1$.

Simulation

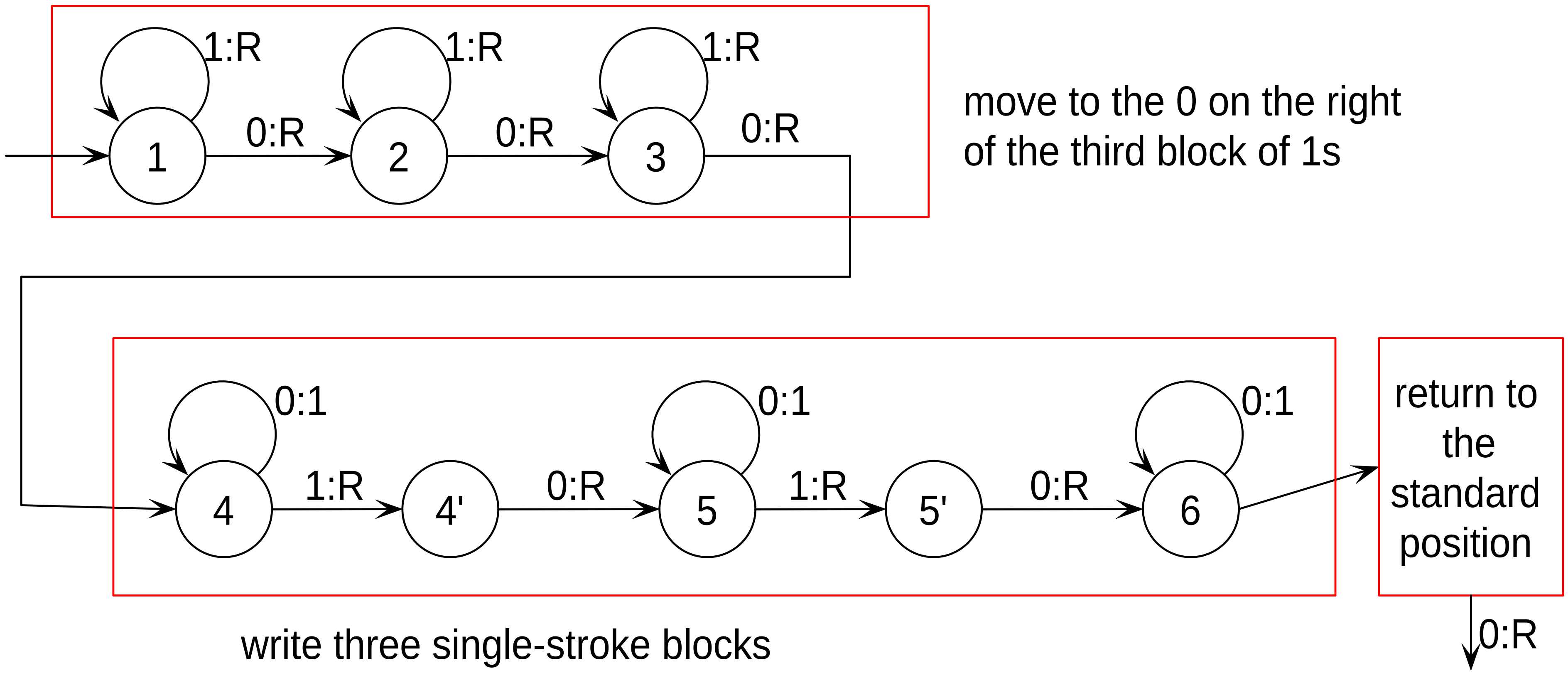
- In this stage, the Turing machine simulates the commands of the abacus program step by step.
- The tape continues to have one block of strokes for every used register.

Cleanup

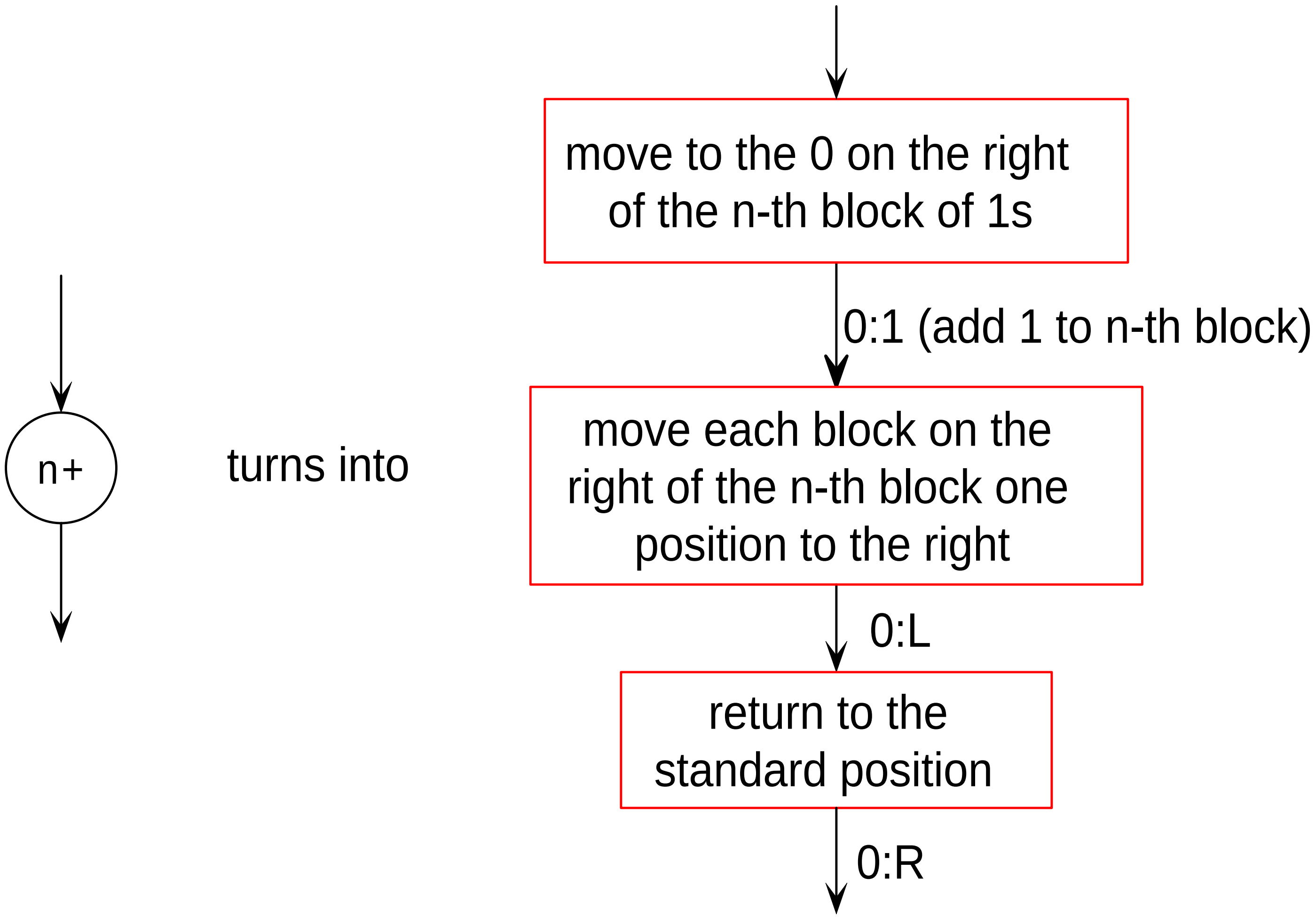
- If the abacus machine halts, the simulating TM will reach a configuration whose tape describes the final content of the abacus machine's registers.
- During the cleanup stage, the TM deletes all blocks of strokes except for the first (which corresponds to the result according to the definition of abacus computability).

Initialization

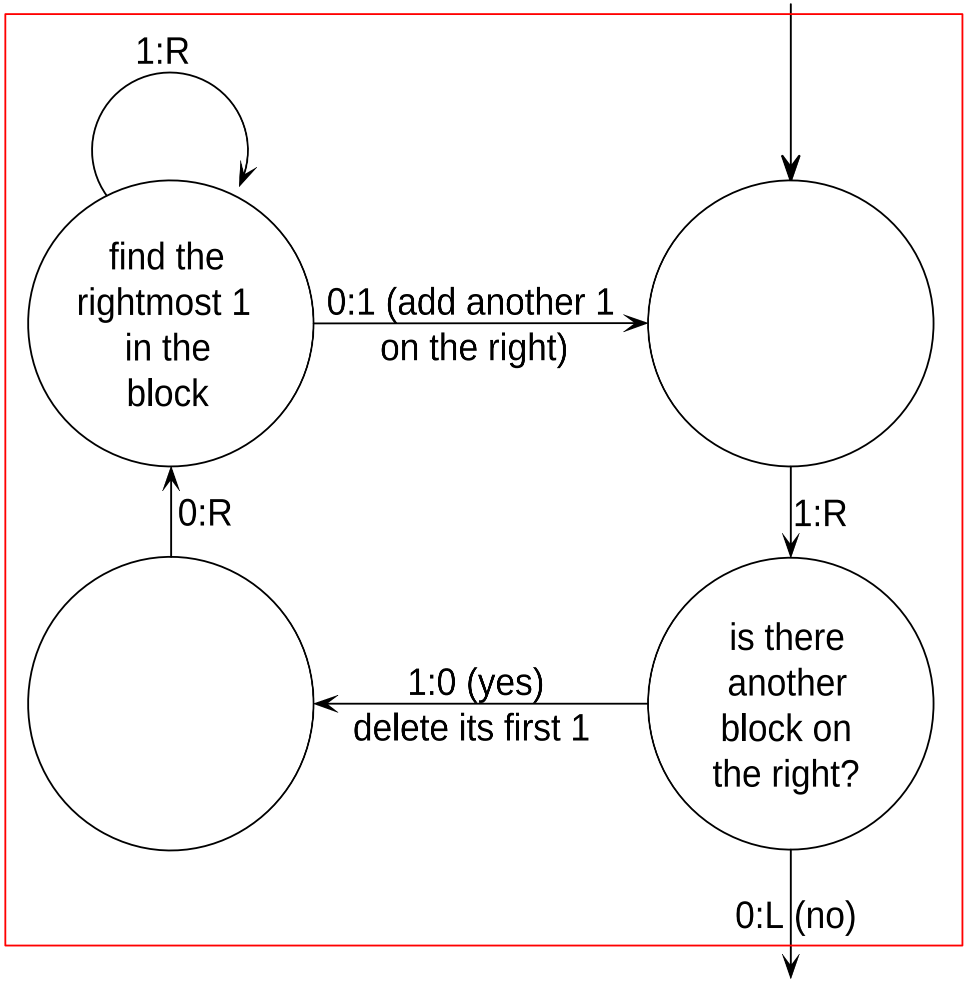
The case for 3 arguments and 6 used registers:



Simulation of $n+$

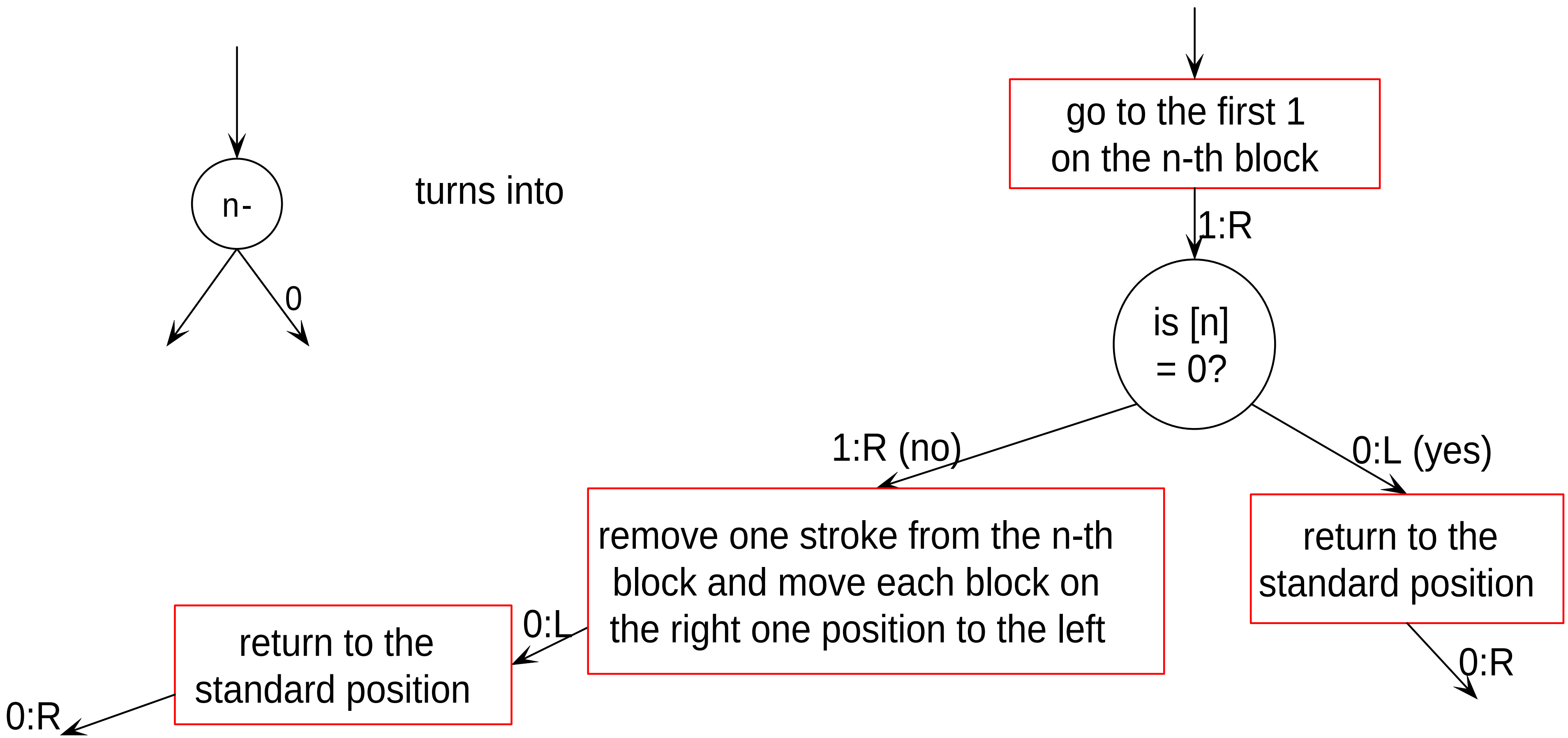


Simulation of $n+$

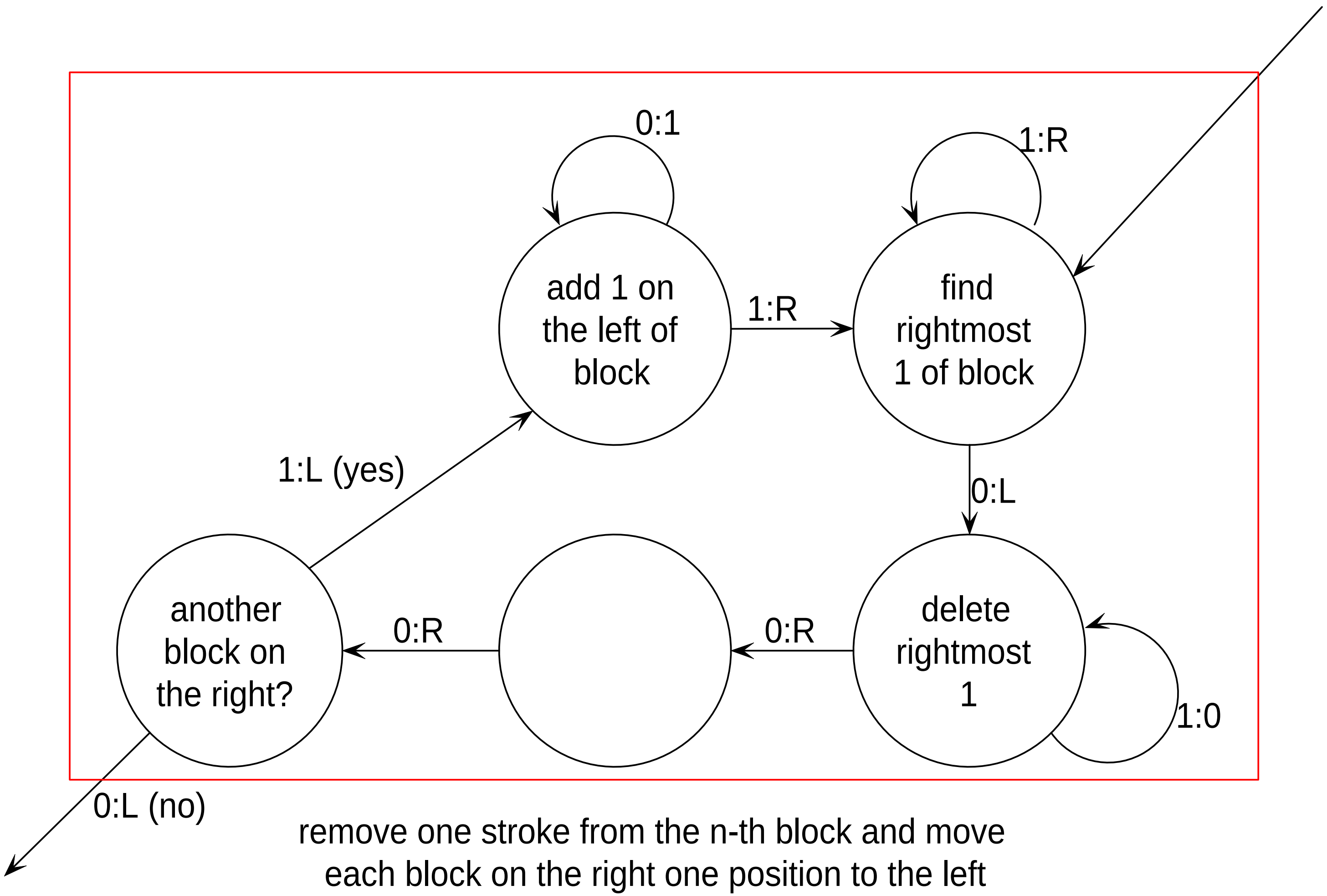


move each block on the right of the n-th block one position to the right

Simulation of $n-$

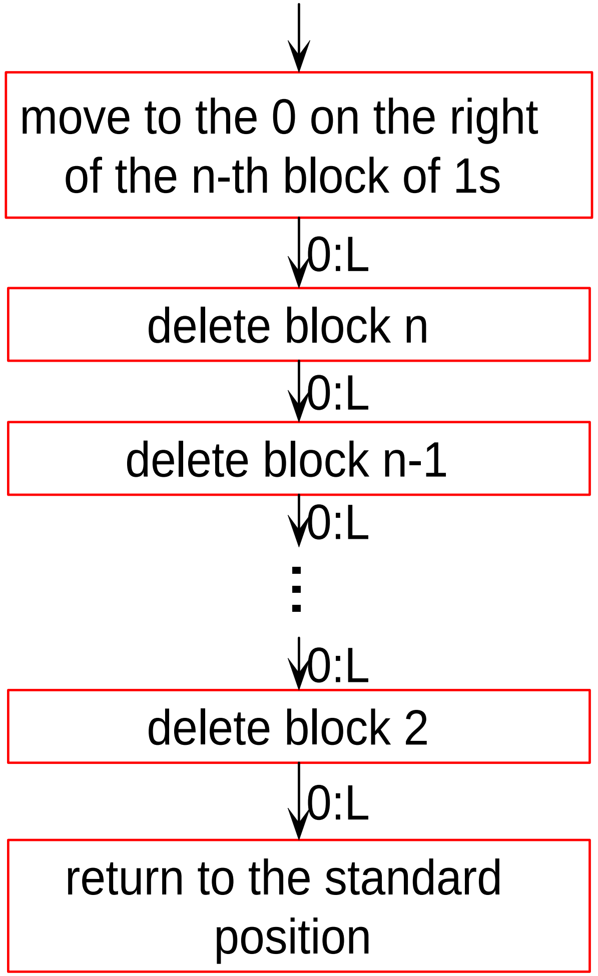


Simulation of $n-$



remove one stroke from the n-th block and move each block on the right one position to the left

Cleanup


move to the 0 on the right of the n-th block of 1s
0:L
delete block n
0:L
delete block n-1
0:L
0:L
delete block 2
0:L
return to the standard position

How to put all parts together

To finish building the simulating TM, we must put all parts together.

- Connect the “loose end” of the initialization flow-graph with the start of the simulation flow-graph.
- Connect all loose ends of the simulation flow-graph with the start of the cleanup flow-graph.
- The resulting flow-graph describes the Turing machine that simulates the abacus machine.