

HOW TO PROGRAM AN INFINITE ABACUS*†

Joachim Lambek
(received June 15, 1961)

This is an expository note to show how an “infinite abacus” (to be defined presently) can be programmed to compute any computable (recursive) function. Our method is probably not new, at any rate, it was suggested by the ingenious technique of Melzak [2] and may be regarded as a modification of the latter.

By an *infinite abacus* we shall understand a countably infinite set of *locations* (holes, wires etc.) together with an unlimited supply of *counters* (pebbles, beads etc.). The locations are distinguishable, the counters are not. The confirmed finitist need not worry about these two infinitudes: To compute any given computable function only a finite number of locations will be used, and this number does not depend on the argument (or arguments) of the function. Moreover, to evaluate such a function at a given argument only a finite number of counters are required.

So far our abacus does not differ from Melzak’s machine. However, while he admits one ternary operation, we require two unary operations $X+$ and $X-$. $X+$ means: Place one counter into location X . $X-$ means: Remove one counter from location X if this is possible, that is, if X is not empty.

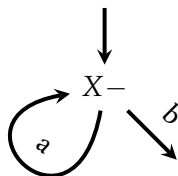
By a *program* we shall understand a finite set of instructions to perform these two elementary operations together with indications to show the following:

1. Which instruction is to be carried out first?
2. Which instruction comes after $X+$?
3. Which two instructions come after $X-$ in the two cases: (a) X is not empty, (b) X is empty?
4. When do we stop?

The idea of using a sometimes impossible instruction to determine two different steps to follow if it is taken from Melzak’s paper.

For a rigorous mathematical definition of “program”, see Appendix II. In the meantime we illustrate this concept by two simple examples. As in [2], we use “flow charts” to represent programs. An incoming arrow indicates *start* and an outgoing arrow indicates *stop*.

Example 1

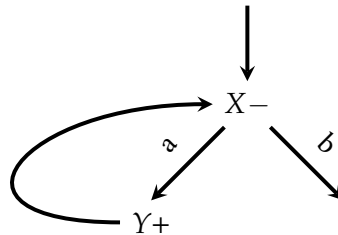


*Canad. Math. Bull. vol. 4. no. 3, September 1961, pp 295–302.

†Text transcribed by Apostolos Syropoulos, Xanthi, Greece, and drawings made by Georgios Georgio, Nicosia, Cyprus.

This program tells us to remove one counter from location X and to repeat this operation until X is empty. We may therefore translate this program to read: Empty X .

Example 2



This program tells us to transfer one counter from X to Y and to keep on doing this until X is empty. We may translate this to read: Transfer the content of X to Y .

By a *configuration* of the abacus we understand an assignment of counters to locations. Thus the program of example 1 transforms the configuration

$$\begin{array}{l} X \\ x \end{array} \quad (X \text{ has content } x)$$

into the configuration

$$\begin{array}{l} X \\ 0 \end{array} \quad (X \text{ has content zero}).$$

Similarly the program of example 2 transform the configuration

$$\begin{array}{ll} X & Y \\ x & y \end{array}$$

into the configuration

$$\begin{array}{ll} X & Y \\ 0 & x + y. \end{array}$$

The contents of all locations not shown are understood to remain constant. A small Roman letter when first introduced will usually denote the current content of the location denoted by the corresponding capital.

We are concerned with functions in n variables ($n \geq 1$), whose arguments and values are non-negative integers. To compute such functions on our abacus, it will be convenient to divide the set of locations into two disjoint infinite subsets. One subset of locations will be used for showing the arguments and values of these functions, the other for temporary storage of intermediate results in the calculations. When computing a particular function, only a finite number of the temporary storage locations will be used, and these will be reserved once and for all for this function.

We shall say that a program *computes* the function $z = \varphi(x, y)$, if it transforms the configuration

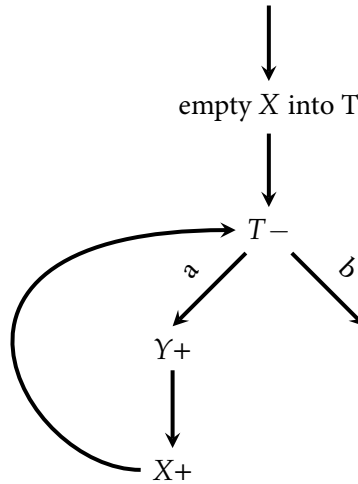
$$\begin{array}{cccccc} X & Y & Z & T_1 & \dots & T_m \\ x & y & 0 & 0 & \dots & 0 \end{array}$$

into the configuration

$$\begin{array}{cccccc} X & Y & Z & T_1 & \dots & T_m \\ x & y & \varphi(x, y) & 0 & \dots & 0 \end{array}$$

where T_1, \dots, T_m are the reserved temporary storage locations. A similar definition is used for functions of n variables, $n \geq 1$.

Example 3 Compute the function $y = x$. Let T be a temporary storage location. We assume that Y and T are empty.



The phrase “empty X into T ” should of course be replaced by the program of example 2 (with T instead of Y). It is easy to see that the first instruction transforms the configuration

X	Y	T
x	0	0

into the configuration

X	Y	T
0	0	x .

Then going once round the loop we obtain

X	Y	T
1	1	$x - 1$.

After traversing the loop x times we get

X	Y	T
x	x	0 .

Now going along arrow b we come to a stop, with the required output configuration.

From now on we shall use the same temporary storage location T whenever we compute the identity function, be it called $y = x$ as above or $z = u$, for that matter, as long as neither variable is t .

According to Kleene [1], the set of *recursive* functions is the smallest set containing the function described in 1 to 3 below and closed under the schemes 4 to 6.

$$\varphi(x) = x + 1. \tag{1}$$

$$\varphi(x_1, \dots, x_n) = k \tag{2}$$

$$\varphi(x_1, \dots, x_n) = x_i, 1 \leq i \leq n. \tag{3}$$

$$\varphi(x_1, \dots, x_n) = \psi(\chi_1(x_1, \dots, x_n), \dots, \chi_m(x_1, \dots, x_n)). \tag{4}$$

$$\varphi(0, x_2, \dots, x_n) = \psi(x_2, \dots, x_n), \tag{5}$$

$$\varphi(y + 1, x_2, \dots, x_n) = \chi(y, \varphi(y, x_2, \dots, x_n), x_2, \dots, x_n).$$

$$\varphi(x_1, \dots, x_n) = \text{the smallest } y \text{ such that } \chi(x_1, \dots, x_n, y) = 0. \tag{6}$$

Here $k \geq 0, n \geq 1, m \geq 1$ and the function χ in 6 is assumed to satisfy the condition

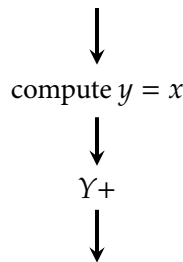
$$\forall x_1 \dots \forall x_n \exists y (\chi(x_1, \dots, x_n, y) = 0).$$

To say that the set of recursive functions is closed under the rule 4, for example, means that the function φ defined by 4 is in the set whenever ψ and χ are given functions in the set.

Theorem 1 *Every recursive function can be computed by a (finite) program on an infinite abacus.*

We could prove this theorem by showing that our abacus is equivalent to Melzak's machine and by quoting Melzak's result that every Turing machine can be simulated on his machine as well as the well-known result that every recursive function can be computed on a Turing machine. However, it will be more instructive to sketch a direct proof of our theorem.

1. The function $y = \varphi(x) = x + 1$ is computed by the following program.

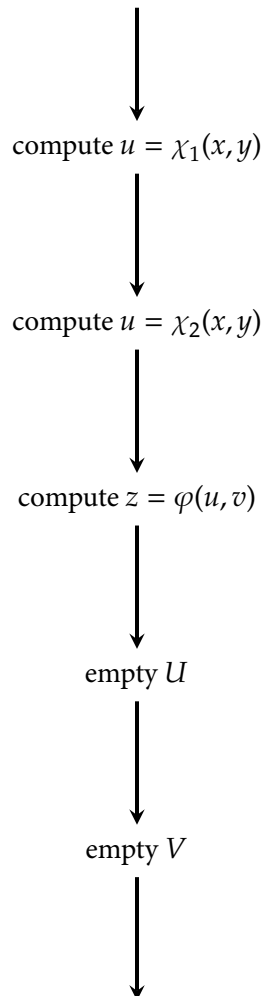


Here the phrase "compute $y = x$ " must of course be replaced by the program of example 3.

2. For the sake of concreteness, let us take $k = 2$. The function $z = \varphi(x, y) = 2$ is computed by the following program.



3. For concreteness take $n = 2, i = 2$. The function $z = \varphi(x, y) = y$ is computed by example 3.
4. For concreteness, take $m = 2, n = 2$. By inductual assumption we know that χ_1, χ_2 and ψ are computable on an abacus, using only a finite number of temporary storage locations. The function $z = \varphi(x, y) = \psi(\chi_1(x, y), \chi_2(x, y))$ is then computed by the following program:

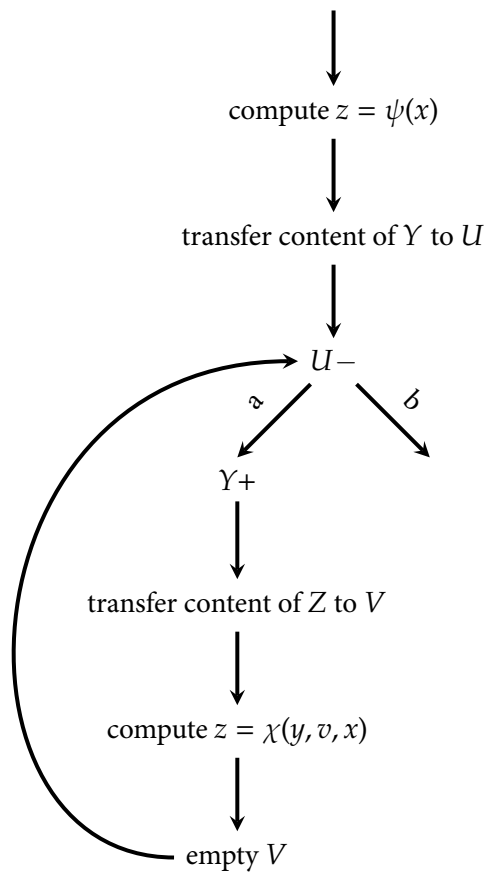


Here U and V are temporary storage locations not used in the computation of χ_1, χ_2 and ψ .

5. For concreteness take $n = 2$. The function $x = \varphi(y, x)$ which is defined recursively by the equations

$$\begin{aligned} \varphi(0, x) &= \psi(x), \\ \varphi(y + 1, x) &= \chi(y, \varphi(y, x), x), \end{aligned}$$

is computed thus:

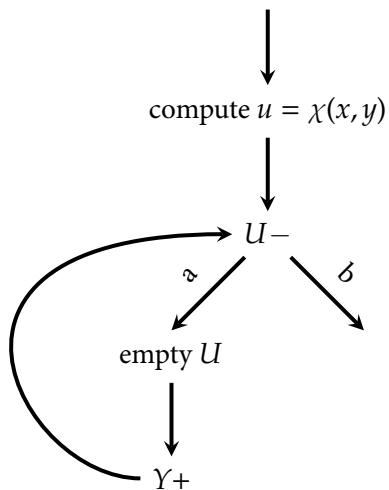


Here U and V are two temporary storage locations not used in computing the identity function as well as the functions ψ and χ .

6. For concreteness take $n = 1$. We are given that

$$\forall x \exists t (\chi(x, t) = 0).$$

The function $y = \varphi(x) = \text{smallest } z \text{ such that } \chi(x, z) = 0$ is computed thus:



Here U is a temporary storage location not used in computing χ .

It becomes clear from the proof sketched above that the program for computing a given computable function is of the same order of complexity as its derivation from rules 1 to 6. That is to say, the program for computing a function is no more complicated than the proof that it is recursive.

APPENDIX I.

The theory of the infinite abacus may be developed further along lines that are familiar from Turing machines [1, p. XIII]. We shall give a brief outline of such a development.

A partial function is a function which is defined only for a subset of its argument set, here the set of non-negative integers. The set of *partial recursive functions* is defined like the set of recursive functions, except that the above condition on rule 6 is now dropped. The partial recursive functions can also be programmed on an infinite abacus, the point being that the computation of $\varphi(c)$ will never come to an end if $\varphi(x)$ is not defined for $x = c$. It is possible to correlate the set of all programs in an effective way with the non-negative integers. Let Π_0, Π_1, \dots be such an enumeration of all programs. For certain $n \geq 0$, Π_n will compute a function $y = \varphi(x)$, and we write $\varphi = \varphi_n$. Consider the partial function $y = \Phi(z, x) = \varphi_z(x)$ if Π_z does compute a recursive function, otherwise we leave $\Phi(z, x)$ undefined. It can be shown that Φ is partial recursive, and so is a *universal* program Π_m which (partially) computes Φ . It also follows that every function of one variable which can be computed by a program on an infinite abacus is a recursive function, and this result may easily be extended to any number of variables. A Cantor type argument shows that the problem of deciding for arbitrary $n \geq 0$ whether Π_n computes a function $y = \varphi(x)$ cannot be programmed on the abacus.

APPENDIX II.

We wish to give a formal definition of "program". We recall that an infinite abacus is essentially a countable set L of locations. A program is described by a finite set N of nodes, together with two distinguished elements n_0 (start) and n_∞ (stop) of N and three functions

$$\begin{aligned} a: N \setminus \{n_\infty\} &\rightarrow N \setminus \{n_0\}, \\ b: N \setminus \{n_\infty\} &\rightarrow N \setminus \{n_0\}, \\ c: N \setminus \{n_0, n_\infty\} &\rightarrow L, \end{aligned}$$

subject to the condition that $a(n_0) = b(n_0)$.

With any node we associate an instruction as follows:

Case 1: $n = n_0$. Go to node $a(n_0)$.

Case 2: $n \neq n_0, n_\infty$; $a(n) = b(n)$. Add 1 to content of location $c(n)$ and proceed to node $a(n)$.

Case 3: $n \neq n_0, n_\infty$; $a(n) \neq b(n)$. Take 1 from content of location $c(n)$, if this is possible, then go to node $a(n)$. Otherwise go to node $b(n)$.

Case 4: $n \neq n_\infty$. Stop.

We illustrate this definition by exhibiting the three functions belonging to the program of example 2.

n	$a(n)$	$b(n)$	$c(n)$
1	2	2	
2	3	4	X
3	2	2	Y
4			

Here $N = \{1, 2, 3, 4\}$, $n_0 = 1$, $n_\infty = 4$.

References

- [1] S.C. Kleene. *Introduction to metamathematics*. D. Van Nostrand Company, Inc., Princeton, N.J., 1952.
- [2] Z.A. Melzak. An informal arithmetical approach to computability and computation. *Canadian Mathematical Bulletin*, 4:279–293, 1961.

Correction to my paper “How to Program an Infinite Abacus”[†] J. Lambek

In my paper “How to program an infinite abacus” (Can. Math. Bull. 4(1961) p. 300) the program in part 5 of the proof of the theorem should be de-bugged by placing the instruction “Y+” two steps further down in the flow chart, that is just before the instruction “empty V ”.

[†]Canadian Mathematical Bulletin, Vol 5, No 3, 297–297, September 1962.