the method of proof in the given system. We shall return to Gödel's result in a later chapter. After much preliminary work by Martin Davis, Hilary Putnam and Julia Robinson, Problem 10 was ultimately disposed of by Yuri Matijasevič, as we shall see presently. But before tackling this problem, one had to determine what Hilbert meant by an 'effective method'.

By a *numerical function* we mean any function $f : \mathbf{N}^n \to \mathbf{N}$, where $\mathbf{N}$ is the set of natural numbers (including 0) and $n \geq 0$. Among these are the *identity* function $fx = x$, and the successor function $fx = Sx$. These basic functions can surely be calculated. Moreover, the set of calculable functions is evidently closed under the following schemes, where we have written $\bar{x}$ for the string $x_1 x_2 \ldots x_m$, $\bar{z}$ for $z_1 \ldots z_k$, etc:

1. substituting one calculable function $g\bar{y}$ for u in another calculable function $f\bar{x}u\bar{z}$ to obtain a function $h\bar{x}\bar{y}\bar{z} = f\bar{x}g\bar{y}\bar{z}$;
2. interchanging two variables: if $f\bar{u}xy\bar{v}$ can be calculated, so can $g\bar{u}xy\bar{v} = f\bar{u}yx\bar{v}$;
3. contracting two variables: if $f\bar{u}xy\bar{v}$ can be calculated, so can $g\bar{u}x\bar{v} = f\bar{u}xx\bar{v}$;
4. introducing superfluous variables: if $f\bar{u}\bar{v}$ can be calculated, so can $g\bar{u}x\bar{v} = f\bar{u}\bar{v}$;
5. the *recursion scheme*: if $g\bar{x}$ and $h\bar{x}yz$ are calculable, so is $f\bar{x}y$ defined 'recursively' by the equations

$$f\bar{x}0 = g\bar{x}, \qquad f\bar{x}Sy = h\bar{x}yf\bar{x}y.$$

With the help of (5) we can calculate $x + y$, $x \times y$, x^y and many other numerical functions.

The functions which can be calculated so far, namely from the basic functions with the help of schemes (1) to (5), are called *primitive recursive functions.* They were introduced by Gödel in the proof of his famous Incompleteness Theorem, which led to the disposal of Hilbert's second problem and to which we shall return in a later chapter.

How are these primitive recursive functions to be calculated? Before the invention of modern computers and pocket calculators, we could make calculations with pencil and paper; but to explain what this means in precise terms is not easy. The first people to give rigorous answers to the question of what a calculation is were Alan Turing and Emil L. Post, independently in the same year, 1936. The work of Post is not as well-known as that of Turing, who invented a theoretical machine, the *Turing machine*, which may be seen as the ancestor of all modern computers. However, building a Turing machine is not an easy way to compute a given primitive recursive function.

A simpler answer would have been available if people had thought about the origin of the word 'calculation', which is derived from the Latin word 'calculus' meaning 'pebble'. Indeed, the ancients performed calculations by moving pebbles from one groove of a table, called an 'abacus', to another. We shall discuss how an abacus may be programmed to calculate any primitive recursive function.

For theoretical purposes we shall assume that an *abacus* consists of a potentially infinite number of *locations* and that we have an inexhaustable supply of *pebbles.* A *program* is made up of two basic instructions:

- put a pebble at location A, then go to ...

$$\begin{tikzcd} {} \arrow[d] \\ A^+ \arrow[d] \\ {} \end{tikzcd}$$

- take a pebble from location B, if B is not empty, then go to ..., else go to ...

$$\begin{tikzcd} & {} \arrow[d] & \\ & B^- \arrow[dl] \arrow[dr] & \\ {} & & {} \end{tikzcd}$$

(the right arrow referring to the 'else' case).

A program is easily illustrated by a *flow diagram.* For example, the following program

$$\begin{tikzcd} \text{Start} \arrow[d] & \\ A^- \arrow[loop left] \arrow[dr] & \\ & \text{stop} \end{tikzcd}$$

is the program 'empty A' and the next program

$$\begin{tikzcd}
& \text{Start} \arrow[d] & \\
& A^- \arrow[dl] \arrow[dr] & \\
B^+ \arrow[ur, bend left] & & \text{stop}
\end{tikzcd}$$

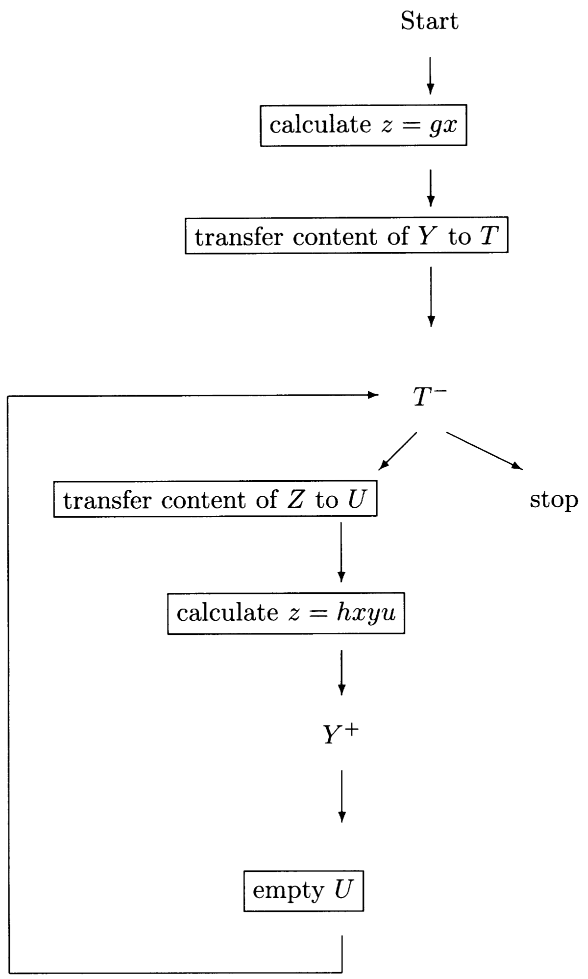
is the program 'transfer the content of A to B'.

To *calculate* a numerical function $z = fxy$ we begin with x pebbles at location X, y pebbles at location Y and no pebbles at any other location. We expect the calculation to stop with x pebbles at X, y pebbles at Y, fxy pebbles at Z and no pebbles at any other location. Similarly we define what it means to calculate $z = fx_1, \dots, x_n$. The following flow diagram illustrates the programs for calculating the successor function $y = Sx$:

$$\begin{tikzcd}
& & \text{Start} \arrow[d] & \\
& & X^- \arrow[dl] \arrow[dr] & \\
& Y^+ \arrow[dl] & & \boxed{\text{transfer content of } T \text{ to } X} \arrow[d] \\
T^+ \arrow[uurr, bend left] & & & Y^+ \arrow[d] \\
& & & \text{stop}
\end{tikzcd}$$

Note the *subroutine* 'transfer content of T to X'. T is a temporary storage location, which is empty at the beginning and at the end of the calculation.

To convince the reader that in fact all primitive recursive functions can be calculated on an abacus, we present a program for calcuating a function fxy obtained by the recursion scheme $fx0 = gx$, $fxSy = hxyfxy$ from functions gx and $hxyz$, which are already known to be calculable:



Primitive recursive functions are not the only numerical functions which can be calculated on an abacus. There is one more scheme, the *minimization scheme*:

6. given a calculable function $g\bar{x}y$, $\bar{x} = x_1x_2 \ldots x_n$, we can calculate $f\bar{x} =$ *the smallest* y *such that* $g\bar{x}y = 0$.

Here is the flow diagram: