

Loop and While programs

To quote from page 261 of the text: “There are many equivalent definitions of the class of recursive functions. Several of these definitions employ idealized computing devices. These computing devices are like digital computers but are free of any limitation on memory space [or running time]. The first definition of this type was published by Alan Turing in 1936; similar work was done by Emil Post at roughly the same time.” The text goes on to describe Shepherdson–Sturgis register machines.

We want to give here another machine-like characterization of the class of recursive functions. For this purpose, we will use a very simple programming language.

The variables of the language are X_0, X_1, X_2, \dots . Although there are infinitely many variables in the language, any one program, being a finite string of commands, can have only finitely many of these variables. (These variables are like the “registers” on page 261.) As before, if we want the language to consist of words over a finite alphabet, we can replace X_3 , say, by X''' .

In running a program, each variable in the program gets assigned a natural number. There is no limit on how large this number can be. Initially, some of the variables will contain the input to the function; the language has no “input” commands. Similarly, the language has no “output” commands; when (and if) the program halts, the value of X_0 is to be the function value.

The commands of the language come in five kinds:

1. $X_n \leftarrow 0$. This is the *clear* command; its effect is to assign the value 0 to X_n .
2. $X_n \leftarrow X_n + 1$. This is the *increment* command; its effect is to increase the value assigned to X_n by one.
3. $X_n \leftarrow X_m$. This is the *copy* command; its effect is just what you think it is; in particular it leaves the value of X_m unchanged.
4. `loop X_n` and `endloop X_n` . These are the *loop* commands, and they must be used in pairs. That is, if \mathcal{P} is a program—a syntactically correct string of commands—then so is the string:

$$\begin{array}{c} \text{loop } X_n \\ \mathcal{P} \\ \text{endloop } X_n \end{array}$$

What this program means is that \mathcal{P} is to be executed a certain number k of times. And that number k is the *initial* value of X_n , the value assigned to X_n

before we start executing \mathcal{P} . Possibly \mathcal{P} will change the value of X_n ; this has no effect at all on k .

5. `while $X_n \neq 0$ and endwhile $X_n \neq 0$.` These are the *while* commands; again, they must be used in pairs, like the the loop commands. But there is a difference. The program

$$\begin{array}{c} \text{while } X_n \neq 0 \\ \quad \mathcal{P} \\ \text{endwhile } X_n \neq 0 \end{array}$$

also executes the program \mathcal{P} some number k of times. But now k is *not* determined in advance; it matters very much how \mathcal{P} changes the value of X_n . The number k is the least number (if any) such that executing \mathcal{P} that many times causes X_n to be assigned the value 0. The program will run forever if there is no such k .

And those are the only commands. A *while* program is a sequence of commands, subject only to the requirement that the loop and while commands are used in pairs, as illustrated.

As mentioned, a while program has no input or output commands. Moreover, there are no if-then commands, and no go-to commands. Why study such a tiny language? The reason is that we want to prove statements *about* the language; for that purpose we'd like the language to be as simple as possible, to keep our proofs simple.

A *loop* program is while program with no while commands; that is, it has only clear, increment, copy, and loop commands. Note the important property: A loop program *always halts*, no matter what. But it is easy to make a while program that never halts.

We say that a k -place partial function f on \mathbb{N} is *while-computable* if there exists a while program \mathcal{P} that, whenever started with a k -tuple \vec{x} assigned to the variables X_1, \dots, X_k and 0 assigned to the other variables, behaves as follows:

- If $f(\vec{x})$ is defined, then the program eventually halts, with X_0 assigned the value $f(\vec{x})$.
- If $f(\vec{x})$ is undefined, then the program never halts.

The *loop-computable* functions are defined in the analogous way. But there is the difference that any loop-computable function is total.

Theorem. A partial function on \mathbb{N} is while-computable if and only if it is a recursive partial function.

The proof is much like the poof of Theorem 36K, as outlined on page 263.