# 1

# Models of Computation

We begin our study by developing several models of computation, each of which reflects all of the features inherent in computation. In an effort to simplify the arguments, all artifacts of computation will be absent from our models. We will start with existing computational paradigms and remove the "bells and whistles" of convenience, arriving at simplified versions of each paradigm that contain all the fundamentally important features of the original. The simplified versions will be shown to be equivalent in a strong sense. This will suggest a model-independent view of computation in terms of "programs" computing functions.

The first artifact of computation that we will dismiss is the notion that arguments may be of different types. All our inputs, outputs, temporary variables, etc. will be natural numbers (members of $\mathbb{N}$). We will argue informally that all the other commonly used argument types are included solely for convenience of programming and are not essential to computation. The argument is based on how computers encode everything into sequences of bits. Boolean arguments can be represented by using the first two members of $\mathbb{N}$, 0 and 1. Floating point numbers, as a consequence of their finite representation, are actually rational. Rational numbers are pairs of natural numbers. In the section on coding, we will show how to encode pairs of natural numbers as a single natural number. Consequently, members of $\mathbb{N}$ suffice to represent the rational numbers, and hence, the floating point numbers. Natural numbers represent character strings via an *index*, or position, in some standard list of all strings. For example, a standard list of all strings using the alphabet $\{a \ldots z\}$ would start by associating 0 with the empty string and then numbering the strings in lexicographical order: $a$, $b$, $\cdots$, $z$, $aa$, $ab$, $\cdots$, $az$, $ba$, $\cdots$.

Analog computation can also be viewed as computing with natural numbers. This follows from the observation that any voltage level can only be measured in increments determined by the measuring device. Even though the voltages are theoretically continuous, all our devices to measure voltages render discrete values. For common examples of turning essentially analog information into a numeric representation we need look no further than the digitally encoded music found in compact disk technology

or the digitally encoded images of high definition television and compact disk memories. We are now ready to present our first model of computation.

## §1.1 Random Access Machines

All contemporary computers allow the "random" accessing of their memory. An address is sent to the memory which returns the data stored at the given address. The name "random access machine" stems from the fact that the earlier models were not random access; they were based on sequential tapes or they were functional in nature. We will consider these models later. As a starting point of our investigation, we will consider a model that strongly resembles assembly language programming on a conventional computer. The model that we introduce will perform very simple operations on registers. Every real-world computer has a fixed amount of memory. This memory can be arbitrarily extended, at great loss of efficiency, by adding a tape or disk drive. Then the ultimate capacity of the machine is limited only by one's ability to manufacture or purchase tapes or disks. Not wanting to consider matters of efficiency just yet, our *random access machine* (RAM) will have a potentially infinite set of registers, `R1`, `R2`, $\cdots$, each capable of holding any natural number. Notice that we have just eliminated main storage and peripheral storage (and their management) as artifacts of how we (necessarily) perform computations. We will be concerned with data, and computation on that data. The issues of input and output will not be addressed in any detail.

RAM programs will be abstractions of assembly language programs in the sense that they use a very limited set of instruction types and the only control structure allowed is the much maligned branch instruction. There is nothing special about our choice of instructions. Among the possible choices for instruction sets, the one chosen here is based on some nontechnical notion of simplicity. RAM programs are finite sequences of very basic instructions. Hence, each RAM program will reference only finitely many of the registers. Even though the memory capacity of a RAM is unlimited, any computation described by a RAM program will access only finitely much data, unless, of course, the computation described never terminates. In case of a nonterminating computation, the amount of data accessed at any given time is finite. Each instruction may have a label, where label names are chosen from the list: `N0`, `N1`, $\cdots$. Each instruction is of one of the following seven types:

1. `INC` R$i$ Increment (by 1) the contents of register R$i$.
2. `DEC` R$i$ Decrement (by 1) the contents of register R$i$. If R$i$ contains 0 before this instruction is executed, the contents of R$i$ remain unchanged.
3. `CLR` R$i$ Place 0 in register R$i$.
4. R$i$ $\leftarrow$R$j$ Replace the contents of register R$i$ with the contents of register R$j$. The contents of R$j$ remain unchanged.

5. `JMP` `N`$ix$ If $x = a$ then the next instruction to execute is the closest preceding instruction with label `N`$i$. If $x = b$ then the next instruction to execute is the closest following instruction with label `N`$i$. The $a$ stands for "above" and the $b$ for "below." This unusual convention allows for the pasting together of programs without paying attention to instruction labels.
6. `R`$j$ `JMP` `N`$ix$ Perform a `JMP` instruction as above if register `R`$j$ contains a 0.
7. `CONTINUE` Do nothing.

**Definition 1.1**: A RAM *program* is a finite sequence of instructions such that each `JMP` instruction (conditional or otherwise) has a valid destination, e.g., the label referred to in the instruction exists, and the final statement is a `CONTINUE`.

**Definition 1.2**: A RAM program *halts* if and when it reaches the final `CONTINUE` statement.

**Definition 1.3**: A RAM program $P$ *computes* a partial function $\phi$, of $n$ arguments, iff when $P$ is started with $x_1, \ldots, x_n$ in registers `R1`, ..., `R`$n$ respectively and all other registers used by $P$ contain 0, $P$ halts only if $\phi(x_1, \ldots, x_n)$ is defined and `R1` contains the value $\phi(x_1, \ldots, x_n)$. A partial function is RAM *computable* if some RAM program computes it.

There is a subtle difference between asserting the existence of a RAM program computing some function and actually being able to produce the program. Consider, for example, the problem of trying to decide how many times the word "recursion" appears as a substring in some arbitrary but fixed infinite random string of symbols from the alphabet $\{a \ldots z\}$. No matter how much of the string we examine, we will never know if we have seen all of the occurrences of the word "recursion." However, there exists a RAM program that will tell us exactly how many occurrences there are. Owing to the possibility of there being infinitely many instances of the word "recursion" embedded in the infinite random string, we will use the convention that a RAM program can signal that there are exactly $n$ repetitions of the word "recursion" in the mystery string by outputting $n + 1$. The output 0 will be reserved to indicate the situation where there are infinitely many occurrences of the substring we are trying to count. Now, for any natural number $n$, there is a RAM program that computes the constant $n$ function. One of these programs will tell us exactly how many occurrences of the word "recursion" are in the string. Which program we cannot say, so it will be impossible to deliver the RAM program that solves our problem.

Most assembly languages have more powerful arithmetic instructions. Recall that our purpose here is not ease of writing RAM programs, but rather ease of proving things about RAM programs. As a first example, the following program computes the sum of two arguments.

```
N1  R2 JMP N2b
    INC R1
    DEC R2
    JMP N1a
N2  CONTINUE
```

**Exercise 1.4**: Show that exponentiation is RAM computable.

**Exercise 1.5**: Show that integer division is RAM computable.

It should be a straightforward, tedious exercise to show that all your favorite assembly language instructions have implementations in the RAM programming language described above. In fact, not all seven of the instruction types are necessary.

**Proposition 1.6**: For every RAM program $P$ there is another RAM program $P'$ computing the same function such that $P'$ only uses statement types 1, 2, 6 and 7.

*Proof*: Suppose $P$ is a RAM program. We show how to transform $P$ into the desired $P'$ in steps, eliminating one type of offending instruction at each step. First, we eliminate the unconditional jumps of statement type 5. Choose $n$ least such that $P$ makes no reference to register R$n$. Form RAM program $P''$ from $P$ by replacing each "N$k$ JMP N$ix$" instruction with the following code segment:

```
Nk  CLR Rn
    Rn JMP Nix
```

Next, we eliminate the register transfers of statement type 4. Choose $m$ and $n$ least such that $P''$ makes no reference to register R$m$ or register R$n$. Let N$c$ and N$d$ be two labels not used in $P''$. Form RAM program $P'''$ from $P''$ by replacing each "N$k$ R$i$ ←R$j$" with the following code segment:

```
Nk  CLR Ri
    CLR Rn
    CLR Rm
Nc  Rj JMP Ndb
    DEC Rj
    INC Ri
    INC Rn
    Rm JMP Nca
Nd  Rn JMP Ncb
    DEC Rn
    INC Rj
    Rm JMP Nda
Nc  CONTINUE
```

Finally, we eliminate the register clear instructions. Let N$c$ be a label not used by $P'''$. Choose $n$ large enough such that no register R$m$ is referenced by $P'''$ for any $m \geq n$. This will guarantee that register R$n$ will initially contain a zero. Finally, form $P'$ from $P'''$ by replacing each "N$k$ CLR R$i$" instruction with the following code segment:

```
Nk  Ri JMP Ncb
    DEC Ri
    Rn JMP Nka
Nc  CONTINUE
```

This completes the proof of the Proposition. The end of proofs will normally be indicated by the symbol ⊗.

## §1.2 Partial Recursive Functions

The next model of computation that we will examine resembles programming in LISP. Actually, it is the other way around — LISP resembles the following computation paradigm. We start by defining the *base* functions.

The class of base functions contains the zero function $Z$ where $Z(x) = 0$ for each $x$ and the successor function $S$ where $S(x) = x + 1$ for each $x$. The class of base functions also contains the *projection* functions. For each positive $n$ and each positive $j \leq n$ there is a projection function $U_j^n$ such that $U_j^n(x_1, \ldots, x_n) = x_j$. Essentially, the projection function selects one of its arguments. The situation is analogous to the UNIX convention of using \$1, \$2, $\cdots$ to represent individual arguments in the programs that we call shell scripts.

The base functions can be combined to obtain other functions. Large classes of functions can be obtained in this fashion. We will look at three operations for defining new functions. Since these operators map functions to functions, they can (and will) be viewed as closure operators. The first of these operators is an iteration operator that is analogous to the well-known "for loops" of FORTRAN and other subsequent programming languages.

**Definition 1.7**: A function $f$ of $n + 1$ arguments is defined by *primitive recursion* from $g$, a function of $n$ arguments, and $h$, a function of $n + 2$ arguments, iff for each $x_1, \ldots, x_n$:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)).$$

For the $n = 0$ case of the above definition we adopt the convention that a function of 0 arguments is a *constant*.

The recursion of the above definition is "primitive" because the value $f(x_1, \ldots, x_n, y)$ can be determined from the value of $f(x_1, \ldots, x_n, z)$ for some $z < y$. Forms of recursion that are not primitive will be discussed extensively later in the book.

**Definition 1.8**: A function $f$ of $n$ arguments is defined by *composition* from $g$ a function of $m$ arguments and functions $h_1$, $h_2$, ..., $h_m$, each of $n$ arguments iff for each $x_1, \ldots, x_n$:

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)).$$

The $m = 1$ case of the above definition yields the traditional composition scheme.