6
Loops and infinity

“The bounds of arithmetic were however outstepped the moment the idea of applying the [punched] cards had occurred; and the Analytical Engine does not occupy common ground with mere “calculating machines.””

... In enabling mechanism to combine together general symbols, in successions of unlimited variety and extent, a uniting link is established between the operations of matter and the abstract mental processes of the most abstract branch of mathematical science.” Ada Augusta, countess of Lovelace, 1843

“... What is the difference between a Turing machine and the modern computer? It’s the same as that between Hillary’s ascent of Everest and the establishment of a Hilton hotel on its peak.”, Alan Perlis, 1982.

The model of Boolean circuits (or equivalently, the NAND-CIRC programming language) has one very significant drawback: a Boolean circuit can only compute a finite function \( f \), and in particular the number of inputs of \( f \) is always smaller than (twice) the number of gates of the circuit. This does not capture our intuitive notion of an algorithm as a single recipe to compute a potentially infinite function. For example, the standard elementary school multiplication algorithm is a single algorithm that multiplies numbers of all lengths, but yet we cannot express this algorithm as a single NAND-CIRC program, but rather need a different NAND-CIRC program for every input length (see Fig. 6.1).

Let us consider the case of the simple parity or XOR function XOR : \( \{0, 1\}^* \rightarrow \{0, 1\} \) where XOR\((x)\) equals 1 iff the number of 1’s in \( x \) is odd. (In other words, XOR\((x) = \sum_{i=0}^{\left| x \right|-1} x_i \mod 2 \) for every \( x \in \{0, 1\}^* \).) As simple as it is, the XOR function cannot be computed by a NAND-CIRC program. Rather, for every \( n \), we can compute XOR\(_n\).

1 Translation of “Sketch of the Analytical Engine” by L. F. Menabrea, Note A.

2 The reason the number of inputs is at most twice the number of gates is because we consider gates such as AND, OR, NOT and NAND that each have two inputs. For different sets of gates the constant factor can be different. However, the conceptual point of being only able to handle a finite number of inputs holds for any model of circuits or straight-line programming language.
Figure 6.2: The NAND circuit and NAND-CIRC program for computing the XOR of 5 bits. Note how the circuit for XOR\textsuperscript{5} merely repeats four times the circuit to compute the XOR of 2 bits.

In this chapter we will show how we can extend the model of Boolean circuits / straight-line programs so that it can capture these kinds of constructs. We will see two ways to do so:

- **Turing machines**, invented by Alan Turing in 1936, are an hypothetical abstract device that can yields a finite description of an algorithm that can handle arbitrarily long inputs.

- The **NAND-TM Programming language** extends NAND-CIRC with the notion of loops and arrays to allow a finite program that can compute a function with arbitrarily long inputs.

It turns out that these two models are **equivalent**, and in fact they are equivalent to a great many other computational models including programming languages you may be familiar with such as C, Java, Python, Javascript, OCaml, and so on and so forth. This notion, known as **Turing equivalence** or **Turing completeness**, will be discussed in Chapter 7. We start off by presenting Turing machines and then show their equivalence to NAND-TM, though it is also possible to present these in the opposite order.

**Remark 6.1 — Finite vs infinite computation.** Previously in this book we studied the computation of finite functions \( f : \{0, 1\}^n \to \{0, 1\}^m \). Such a function \( f \) can always be described by listing all the \( 2^n \) values it takes on inputs \( x \in \{0, 1\}^n \).

In this chapter we consider functions that take inputs of **unbounded size**, such as the function

```python
# s is the "running parity", initialized to 0
while i<len(X):
    u = NAND(s,X[i])
    v = NAND(s,u)
    w = NAND(X[i],u)
    s = NAND(v,w)
    i+= 1
Y[0] = s
```
The “granddaddy” of all models of computation is the Turing Machine. Turing machines were defined in 1936 by Alan Turing in an attempt to formally capture all the functions that can be computed by human “computers” (see Fig. 6.4) that follow a well-defined set of rules, such as the standard algorithms for addition or multiplication.

Turing thought of such a person as having access to as much “scratch paper” as they need. For simplicity we can think of this scratch paper as a one dimensional piece of graph paper (or tape, as it is commonly referred to), which is divided to “cells”, where each “cell” can hold a single symbol (e.g., one digit or letter, and more generally some element of a finite alphabet). At any point in time, the person can read from and write to a single cell of the paper, and based on the contents can update his/her finite mental state, and/or move to the cell immediately to the left or right of the current one.

XOR : \{0, 1\}∗ → \{0, 1\} that maps \( z \) to \( \sum_{i=0}^{\lfloor |z|/2 \rfloor} x_i \mod 2 \). While we can describe XOR using a finite number of symbols (in fact we just did so in the previous sentence), it takes infinitely many possible inputs and so we cannot just write down all of its values. The same is true for many other functions capturing important computational tasks including addition, multiplication, sorting, finding paths in graphs, fitting curves to points, and so on and so forth.

To contrast with the finite case, we will sometimes call a function \( F : \{0, 1\}^* \rightarrow \{0, 1\} \) (or \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \)) infinite but we emphasize that the functions we are interested in always take an input which is a finite string. It’s just that, unlike the finite case, this string can be arbitrarily long and is not fixed to some particular length \( n \).

6.1 TURING MACHINES

“Computing is normally done by writing certain symbols on paper. We may suppose that this paper is divided into squares like a child’s arithmetic book. The behavior of the [human] computer at any moment is determined by the symbols which he is observing, and of his ‘state of mind’ at that moment… We may suppose that in a simple operation not more than one symbol is altered.”

“We compare a man in the process of computing … to a machine which is only capable of a finite number of configurations… The machine is supplied with a ‘tape’ (the analogue of paper) … divided into sections (called ‘squares’) each capable of bearing a ‘symbol’”, Alan Turing, 1936

Figure 6.3: Aside from his many other achievements, Alan Turing was an excellent long distance runner who just fell shy of making England’s olympic team. A fellow runner once asked him why he punished himself so much in training. Alan said “I have such a stressful job that the only way I can get it out of my mind is by running hard; it’s the only way I can get some release.”

Figure 6.4: Until the advent of electronic computers, the word “computer” was used to describe a person that performed calculations. These human computers were absolutely essential to many achievements including mapping the stars, breaking the Enigma cipher, and the NASA space mission. Photo taken from [Sob17].

Figure 6.5: Steam-powered Turing Machine mural, painted by CSE grad students at the University of Washington on the night before spring qualifying examinations, 1987. Image from https://www.cs.washington.edu/building/art/SPTM.
Thus, Turing modeled such a computation by a “machine” that maintains one of \( k \) states, and at each point can read and write a single symbol from some alphabet \( \Sigma \) (containing \{0, 1\}) from its “work tape” (see Fig. 6.6). To perform computation using this machine, we write the input \( x \in \{0, 1\}^* \) on the tape, and the goal of the machine is to ensure that at the end of the computation, the value \( F(x) \) will be written on the tape. Specifically, a computation of a Turing Machine \( M \) with \( k \) states and alphabet \( \Sigma \) on input \( x \in \{0, 1\}^* \) proceeds as follows:

- Initially the machine is at state 0 (known as the “starting state”) and the tape is initialized to \( \triangleright, x_0, \ldots, x_{n-1}, \emptyset, \emptyset, \ldots \).
- The location \( i \) to which the machine points to is set to 0.
- At each step, the machine reads the symbol \( \sigma = T[i] \) that is in the \( i^{th} \) location of the tape, and based on this symbol and its state \( s \) decides on:
  - What symbol \( \sigma' \) to write on the tape
  - Whether to move Left (i.e., \( i \leftarrow i - 1 \)), Right (i.e., \( i \leftarrow i + 1 \)), Stay in place, or Halt the computation.
  - What is going to be the new state \( s \in [k] \)
- The set of rules the Turing machine follows is known as its transition function.
- When the machine halts then its output is obtained by reading off the tape from the second location (just after the \( \triangleright \)) onwards, stopping at the first point where the symbol is not 0 or 1.

\[\text{Figure 6.6: A Turing machine has access to a tape of unbounded length. At each point in the execution, the machine can read a single symbol of the tape, and based on that and its current state, write a new symbol, update the tape, decide whether to move left, right, stay, or halt.}\]
6.1.1 Example: A Turing machine for palindromes

Let \( \text{PAL} \) (for {\it palindromes}) be the function that on input \( x \in \{0, 1\}^* \), outputs 1 if and only if \( x \) is an (even length) palindrome, in the sense that \( x = w_0 \cdots w_{n-1} w_{n-1} w_{n-2} \cdots w_0 \) for some \( n \in \mathbb{N} \) and \( w \in \{0, 1\}^n \).

We now show a Turing Machine \( M \) that computes \( \text{PAL} \). To specify \( M \) we need to specify (i) \( M \)'s tape alphabet \( \Sigma \) which should contain at least the symbols \( 0, 1, \triangleright, \) and \( \emptyset \), and (ii) \( M \)'s transition function which determines what action \( M \) takes when it reads a given symbol while it is in a particular state.

In our case, \( M \) will use the alphabet \( \{0, 1, \triangleright, \emptyset, \times\} \) and will have \( k = 14 \) states. Though the states are simply numbers between 0 and \( k - 1 \), for convenience we will give them the following labels:

<table>
<thead>
<tr>
<th>State</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>START</td>
</tr>
<tr>
<td>1</td>
<td>RIGHT_0</td>
</tr>
<tr>
<td>2</td>
<td>RIGHT_1</td>
</tr>
<tr>
<td>3</td>
<td>LOOK_FOR_0</td>
</tr>
<tr>
<td>4</td>
<td>LOOK_FOR_1</td>
</tr>
<tr>
<td>5</td>
<td>RETURN</td>
</tr>
<tr>
<td>6</td>
<td>REJECT</td>
</tr>
<tr>
<td>7</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>8</td>
<td>OUTPUT_0</td>
</tr>
<tr>
<td>9</td>
<td>OUTPUT_1</td>
</tr>
<tr>
<td>10</td>
<td>\emptyset_AND_BLANK</td>
</tr>
<tr>
<td>11</td>
<td>1_AND_BLANK</td>
</tr>
<tr>
<td>12</td>
<td>BLANK_AND_STOP</td>
</tr>
</tbody>
</table>

We describe the operation of our Turing Machine \( M \) in words:

- \( M \) starts in state \( \text{START} \) and will go right, looking for the first symbol that is 0 or 1. If we find \( \emptyset \) before we hit such a symbol then we will move to the \( \text{OUTPUT}_1 \) state that we describe below.

- Once \( M \) found such a symbol \( b \in \{0, 1\} \), \( M \) deletes \( b \) from the tape by writing the \( \times \) symbol, it enters either the \( \text{RIGHT}_0 \) or \( \text{RIGHT}_1 \) mode according to the value of \( b \) and starts moving rightwards until it hits the first \( \emptyset \) or \( \times \) symbol.

- Once we found this symbol we into the state \( \text{LOOK}_\text{FOR}_0 \) or \( \text{LOOK}_\text{FOR}_1 \) depending on whether we were in the state \( \text{RIGHT}_0 \) or \( \text{RIGHT}_1 \) and make one left move.

- In the state \( \text{LOOK}_\text{FOR}_b \), we check whether the value on the tape is \( b \). If it is, then we delete it by changing its value to \( \times \), and move to the state \( \text{RETURN} \). Otherwise, we change to the \( \text{OUTPUT}_0 \) state.
• The RETURN state means we go back to the beginning. Specifically, we move leftward until we hit the first symbol that is not 0 or 1, in which case we change our state to START.

• The OUTPUT_\(b\) states mean that we are going to output the value \(b\). In both these states we go left until we hit \(\triangleright\). Once we do so, we make a right step, and change to the \(1\_\text{AND\_BLANK}\) or \(0\_\text{AND\_BLANK}\) states respectively. In the latter states, we write the corresponding value, and then move right and change to the \(\text{BLANK\_AND\_STOP}\) state, in which we write \(\emptyset\) to the tape and halt.

The above description can be turned into a table describing for each one of the \(13 \cdot 5\) combination of state and symbol, what the Turing machine will do when it is in that state and it reads that symbol. This table is known as the transition function of the Turing machine.

### 6.1.2 Turing machines: a formal definition

The formal definition of Turing machines is as follows:

**Definition 6.2 — Turing Machine.** A (one tape) Turing machine with \(k\) states and alphabet \(\Sigma \supseteq \{0, 1, \triangleright, \emptyset\}\) is represented by a transition function \(\delta_M : [k] \times \Sigma \rightarrow [k] \times \Sigma \times \{L, R, S, H\}\).

For every \(x \in \{0, 1\}^*\), the output of \(M\) on input \(x\), denoted by \(M(x)\), is the result of the following process:

1. We initialize \(T\) to be the sequence \(\triangleright, x_0, x_1, \ldots, x_{n-1}, \emptyset, \emptyset, \ldots\), where \(n = |x|\). (That is, \(T[0] = \triangleright, T[i + 1] = x_i\) for \(i \in [n]\), and \(T[i] = \emptyset\) for \(i > n\).)

2. We also initialize \(i = 0\) and \(s = 0\).

3. We then repeat the following process:
   1. Let \((s', \sigma', D) = \delta_M(s, T[i])\).
   2. Set \(s \rightarrow s', T[i] \rightarrow \sigma'\).
   3. If \(D = R\) then set \(i \rightarrow i + 1\), if \(D = L\) then set \(i \rightarrow \max\{i - 1, 0\}\).
      (If \(D = S\) then we keep \(i\) the same.)
   4. If \(D = H\) then halt.

4. The result of the process, which we denote by \(M(x)\), is the string \(T[1], \ldots, T[m]\) where \(m > 0\) is the smallest integer such that \(T[m + 1] \notin \{0, 1\}\). If the process never ends then we write \(M(x) = \bot\).
You should make sure you see why this formal definition corresponds to our informal description of a Turing Machine. To get more intuition on Turing Machines, you can explore some of the online available simulators such as Martin Ugarte’s, Anthony Morphett’s, or Paul Rendell’s.

**Remark 6.3 — Transition functions vs computed functions, comparison to Sipser.** One should not confuse the transition function $\delta_M$ of a Turing machine $M$ with the function that the machine computes. The transition function $\delta_M$ is a finite function, with $k|\Sigma|$ inputs and $4k|\Sigma|$ outputs. (Can you see why?) The machine can compute an infinite function $F$ that takes as input a string $x \in \{0, 1\}^*$ of arbitrary length and might also produce an arbitrary length string as output.

In our formal definition, we identified the machine $M$ with its transition function $\delta_M$ since the transition function tells us everything we need to know about the Turing machine, and hence serves as a good mathematical representation of it. This choice of representation is somewhat arbitrary, and is based on our convention that the state space is always the numbers $\{0, \ldots, k - 1\}$ with 0 as the starting state. Other texts use different conventions and so their mathematical definition of a Turing machine might look superficially different, but these definitions describe the same computational process and have the same computational powers.

For example, Sipser’s text [Sip97] allows a more general set of states $Q$. Sipser also restrict attention to Turing machines that output only a single bit and therefore designates two special halting states: the “0 halting state” (often known as the rejecting state) and the other as the “1 halting state” (often known as the accepting state). Thus instead of writing 0 or 1 on an output tape, the machine will enter into one of these states and halt. This again makes no difference to the computational power, though we prefer to consider the more general model of multi-bit outputs.

Sipser considers also functions with input in $\Sigma^*$ for an arbitrary alphabet $\Sigma$ (and hence distinguishes between the input alphabet which he denotes as $\Sigma$ and the tape alphabet which he denotes as $\Gamma$), while we restrict attention to functions with binary strings as input. Again this is not a major issue, since we can always encode an element of $\Sigma$ using a binary string of length $\log |\Sigma|$. Finally (and this is a very minor point) Sipser requires the machine to either move left or right in every step, without the Stay operation, though stay-
Unlike the extended Church Turing Thesis which we discussed in Section 5.5, the Church-Turing thesis itself is widely believed and there are no candidate devices that attack it.

The bottom line is that Sipser defines Turing machines as a *seven tuple* consisting of the state space, input alphabet, tape alphabet, transition function, starting state, accepting state, and rejecting state. Superficially this might look like a very different definition than Definition 6.2 but it is simply a different representation of the same concept, just as a graph can be represented in either adjacency list or adjacency matrix form.

### 6.1.3 Computable functions

We now turn to making one of the most important definitions in this book, that of *computable functions*.

**Definition 6.4 — Computable functions.** Let \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \) be a (total) function and let \( M \) be a Turing machine. We say that \( M \) *computes* \( F \) if for every \( x \in \{0, 1\}^* \), \( M(x) = F(x) \).

We say that a function \( F \) is *computable* if there exists a Turing machine \( M \) that computes it.

Defining a function “computable” if and only if it can be computed by a Turing machine might seem “reckless” but, as we’ll see in Chapter 7, it turns out that being computable in the sense of Definition 6.4 is equivalent to being computable in essentially any reasonable model of computation. This is known as the *Church Turing Thesis*.\(^4\)

\(^4\) Unlike the extended Church Turing Thesis which we discussed in Section 5.5, the Church-Turing thesis itself is widely believed and there are no candidate devices that attack it.

This is a good point to remind the reader of the distinction between *functions* and *programs*:

\[
\text{Functions } \neq \text{ Programs .} \tag{6.1}
\]

A Turing machine (or program) \( M \) can *compute* some function \( F \), but it is not the same as \( F \). In particular there can be more than one program to compute the same function. Being computable is a property of *functions*, not of machines.

We will often pay special attention to functions \( F : \{0, 1\}^* \rightarrow \{0, 1\} \) that have a single bit of output. Hence we give a special name for the set of functions of this form that are computable.

\[
\text{Functions } \neq \text{ Programs .} \tag{6.1}
\]
**Definition 6.5 — The class R.** We define $R$ to be the set of all computable functions $F : \{0, 1\}^* \rightarrow \{0, 1\}$.

**Remark 6.6 — Languages vs. functions.** Many texts use the terminology of “languages” rather than functions to refer to computational tasks. The name “language” has its roots in formal language theory as pursued by linguists such as Noam Chomsky. A formal language is a subset $L \subseteq \{0, 1\}^*$ (or more generally $L \subseteq \Sigma^*$ for some finite alphabet $\Sigma$). The membership or decision problem for a language $L$, is the task of determining, given $x \in \{0, 1\}^*$, whether or not $x \in L$. A Turing machine $M$ decides a language $L$ if for every input $x \in \{0, 1\}^*$, $M(x)$ outputs 1 if and only if $x \in L$. This is equivalent to computing the Boolean function $F : \{0, 1\}^* \rightarrow \{0, 1\}$ defined as $F(x) = 1$ iff $x \in L$. A language $L$ is decidable if there is a Turing machine $M$ that decides it. For historical reasons, some texts also call such a language recursive (which is the reason that the letter $R$ is often used to denote the set of computable Boolean functions / decidable languages defined in Definition 6.5).

In this book we stick to the terminology of functions rather than languages, but all definitions and results can be easily translated back and forth by using the equivalence between the function $F : \{0, 1\}^* \rightarrow \{0, 1\}$ and the language $L = \{x \in \{0, 1\}^* | F(x) = 1\}$.

### 6.1.4 Infinite loops and partial functions

One crucial difference between circuits/straight-line programs and Turing machines is the following. Looking at a NAND-CIRC program $P$, we can always tell how many inputs and how many outputs it has (by simply looking at the $X$ and $Y$ variables). Furthermore, we are guaranteed that if we invoke $P$ on any input then some output will be produced.

In contrast, given any Turing machine $M$, we cannot determine a priori the length of the output. In fact, we don’t even know if an output would be produced at all! For example, it is very easy to come up with a Turing machine whose transition function never outputs $H$ and hence never halts.

If a machine $M$ fails to stop and produce an output on some input $x$, then it cannot compute any total function $F$, since clearly on input $x$, $M$ will fail to output $F(x)$. However, $P$ can still compute a partial function.$^6$
For example, consider the partial function $DIV$ that on input a pair $(a, b)$ of natural numbers, outputs $\lceil a/b \rceil$ if $b > 0$, and is undefined otherwise. We can define a turing machine $M$ that computes $DIV$ on input $a, b$ by outputting the first $c = 0, 1, 2, \ldots$ such that $cb \geq a$. If $a > 0$ and $b = 0$ then the machine $M$ will never halt, but this is OK, since $DIV$ is undefined on such inputs. If $a = 0$ and $b = 0$, the machine $M$ will output 0, which is also OK, since we don’t care about what the program outputs on inputs on which $DIV$ is undefined. Formally, we define computability of partial functions as follows:

**Definition 6.7 — Computable (partial or total) functions.** Let $F$ be either a total or partial function mapping $\{0, 1\}^*$ to $\{0, 1\}^*$ and let $M$ be a Turing machine. We say that $M$ computes $F$ if for every $x \in \{0, 1\}^*$ on which $F$ is defined, $M(x) = F(x)$. 7 We say that a (partial or total) function $F$ is computable if there is a Turing machine that computes it.

**Remark 6.8 — Bot symbol.** We often use $\perp$ as our special “failure symbol”. If a Turing machine $M$ fails to halt on some input $x \in \{0, 1\}^*$ then we denote this by $M(x) = \perp$. This does not mean that $M$ outputs some encoding of the symbol $\perp$ but rather that $M$ enters into an infinite loop when given $x$ as input.

If a partial function $F$ is undefined on $x$ then can also write $F(x) = \perp$. Therefore one might think that Definition 6.7 can be simplified to requiring that $M(x) = F(x)$ for every $x \in \{0, 1\}^*$, which would imply that for every $x$, $M$ halts on $x$ if and only if $F$ is defined on $x$. However this is not the case: for a Turing Machine $M$ to compute a partial function $F$ it is not necessary for $M$ to enter an infinite loop on inputs $x$ on which $F$ is not defined. All that is needed is for $M$ to output $F(x)$ on $x$’s on which $F$ is defined: on other inputs it is OK for $M$ to output an arbitrary value or not to halt at all. 8

---

### 6.2 Turing Machines as Programming Languages

The name “Turing machine”, with its “tape” and “head” evokes a physical object, while in contrast we think of a program as a piece of text. But we can think of a Turing machine as a program as well.

For example, consider the Turing Machine $M$ of Section 6.1.1 that computes the function $PAL$ such that $PAL(x) = 1$ iff $x$ is a palindrome.

We can also describe this machine as a program using the Python-like pseudocode of the form below:

---

7 Note that if $F$ is a total function, then it is defined on every $x \in \{0, 1\}^*$ and hence in this case, this definition is identical to Definition 6.4.

8 To borrow a term from the C programming language, on inputs $x$ on which $F$ is not defined, what $M$ does is “undefined behaviour”.
Most programming languages use arrays of fixed size, while a Turing machine's tape is unbounded. But of course there is no need to store an infinite number of $\emptyset$ symbols. If you want, you can think of the tape as a list that starts off just long enough to store the input, but is dynamically grown in size as the Turing machine's head explores new positions.

The particular details of this program are not important. What is important is that we can describe Turing machines as programs. Moreover, note that when translating a Turing machine into a program, the tape becomes a list or array that can hold values from the finite set $\Sigma$. The head position can be thought of as an integer valued variable that can hold integers of unbounded size. The state is a local register that can hold one of a fixed number of values in $[k]$.

More generally we can think of every Turing Machine $M$ as equivalent to a program along the following lines:

```python
# Gets an array Tape that is initialized to [">", x_0, x_1 ...
# At the end of the execution, Tape[1] is equal to 1 if x
# is a palindrome and is equal to 0 otherwise

def PAL(Tape):
    head = 0
    state = 0  # START
    while (state != 12):
        if (state == 0 && Tape[head]=='0'):
            state = 3  # LOOK_FOR_0
            Tape[head] = 'x'
            head += 1  # move right
        if (state==0 && Tape[head]=='1')
            state = 4  # LOOK_FOR_1
            Tape[head] = 'x'
            head += 1  # move right
    ... # more if statements here
```

\[ Most \text{ programming languages use arrays of fixed size, while a Turing machine's tape is unbounded. But of course there is no need to store an infinite number of } \emptyset \text{ symbols. If you want, you can think of the tape as a list that starts off just long enough to store the input, but is dynamically grown in size as the Turing machine's head explores new positions.} \]
... 

```python
elif Tape[i] == '>' and state == 29:
    break  # Halt
```

If we wanted to use only Boolean (i.e., 0/1-valued) variables then we can encode the state variables using \( |\log k| \) bits. Similarly, we can represent each element of the alphabet \( \Sigma \) using \( |\log |\Sigma|| \) bits and hence we can replace the \( \Sigma \)-valued array \( \text{Tape}[] \) with \( |\log |\Sigma|| \) Boolean-valued arrays \( \text{Tape}_0[], \ldots, \text{Tape}_\ell[] \) for \( \ell = |\log |\Sigma|| \).

### 6.2.1 The NAND-TM Programming language

We now introduce the NAND-TM programming language, which aims to capture the power of a Turing machine in a programming language formalism. Just like the difference between Boolean circuits and Turing Machines, the main difference between NAND-TM and NAND-CIRC is that NAND-TM models a single uniform algorithm that can compute a function that takes inputs of arbitrary lengths. To do so, we extend the NAND-CIRC programming language with two constructs:

- **Loops**: NAND-CIRC is a straight-line programming language- a NAND-CIRC program of \( s \) lines takes exactly \( s \) steps of computation and hence in particular cannot even touch more than \( 3s \) variables. Loops allow us to capture in a short program the instructions for a computation that can take an arbitrary amount of time.

- **Arrays**: A NAND-CIRC program of \( s \) lines touches at most \( 3s \) variables. While we can use variables with names such as \( \text{Foo}_17 \) or \( \text{Bar}[22] \), they are not true arrays, since the number in the identifier is a constant that is “hardwired” into the program.

Thus a good way to remember NAND-TM is using the following informal equation:

\[
\text{NAND-TM} = \text{NAND-CIRC} + \text{loops} + \text{arrays} \quad (6.2)
\]

![Figure 6.7](image-url)
Concretely, the NAND-TM programming language adds the following features on top of NANC-CIRC (see Fig. 6.7):

- We add a special integer valued variable $i$. All other variables in NAND-TM are Boolean valued (as in NAND-CIRC).

- We add arrays to the language by allowing variable identifiers to have the form `Foo[i]` with $i$ being the special integer-valued variable mentioned above. `Foo` is an array of Boolean values, and `Foo[i]` refers to the value of this array at location equal to the current value of the variable $i$.

- We use the convention that arrays always start with a capital letter, and scalar variables (which are never indexed with $i$) start with lowercase letters. Hence `Foo` is an array and `bar` is a scalar variable.

- The input and output $X$ and $Y$ are now considered arrays with values of zeroes and ones.

- We add a special `MODANDJUMP` instruction that takes two boolean variables $a, b$ as input and does the following:
  - If $a = 1$ and $b = 1$ then `MODANDJUMP(a, b)` increments $i$ by one and jumps to the first line of the program.
  - If $a = 0$ and $b = 1$ then `MODANDJUMP(a, b)` decrements $i$ by one and jumps to the first line of the program. (If $i$ is already equal to 0 then it stays at 0.)
  - If $a = 1$ and $b = 0$ then `MODANDJUMP(a, b)` jumps to the first line of the program without modifying $i$.
  - If $a = b = 0$ then `MODANDJUMP(a, b)` halts execution of the program.

- Since `MODANDJUMP` always either jumps to the first line of the program or halts the computation, the instructions following it will never get executed. Hence in NAND-TM programs the `MODANDJUMP` instruction appears in the last line of the program and nowhere else.
Remark 6.10 — Default values and conventions. We also need one more convention to handle “default values”. In a Turing machine the tape could contain in addition to 0 and 1 (and possibly other values) also the special symbol ∅ to indicate that this location is “blank”. All our variables, including our arrays, will be Boolean, and so they can contain either the value 0 or 1. All values default to 0 if they have not been initialized to another value.

To keep track of whether a 0 in an array corresponds to a true zero or to an uninitialized cell, a programmer can always add to an array Foo a “companion array” Foo_nonblank and set Foo_nonblank[i] to 1 whenever the i’th location is written to. In particular we will use this convention for the input and output arrays X and Y. When a NAND-TM program is executed it has four special arrays X, X_nonblank, Y, and Y_nonblank. When a NAND-TM program is executed on input \( x \in \{0, 1\}^* \) of length \( n \), the first \( n \) cells of the array \( X \) are initialized to \( x_0, \ldots, x_{n-1} \) and the first \( n \) cells of the array \( X_{\text{nonblank}} \) are initialized to 1. (All uninitialized cells default to 0.)

The output of a NAND-TM program is the string \( Y[0], \ldots, Y[m-1] \) where \( m \) is the smallest integer such that \( Y_{\text{nonblank}}[m] = 0 \). A NAND-TM program gets called with \( X \) and \( X_{\text{nonblank}} \) initialized to contain the input, and writes to \( Y \) and \( Y_{\text{nonblank}} \) to produce the output.

Formally, NAND-TM programs are defined as follows:

Definition 6.11 — NAND TM programs. A NAND-TM program consists of a sequence of lines of the form \( \text{foo} = \text{NAND}(\text{bar}, \text{blah}) \) ending with a line of the form \( \text{MODANDJMP}(\text{foo}, \text{bar}) \), where \( \text{foo}, \text{bar}, \text{blah} \) are either scalar variables (sequences of letters, digits, and underscores) or array variables of the form \( \text{Foo}[i] \) (starting with capital letter and indexed by \( i \)).

If \( P \) is a NAND-TM program and \( x \in \{0, 1\}^* \) is an input then an execution of \( P \) on \( x \) is the following process:

1. The arrays \( X \) and \( X_{\text{nonblank}} \) are initialized by \( X[i] = x_i \) and \( X_{\text{nonblank}}[i] = 1 \) for all \( i \in [\|x\|] \). All other variables and cells are initialized to 0. The index variable \( i \) is also initialized to 0.

2. The program is executed line by line, when the last line \( \text{MODANDJMP}(\text{foo}, \text{bar}) \) is executed then we do as follows:
a. If $\text{foo} = 1$ and $\text{bar} = 0$ then jump to the first line without modifying the value of $i$.

b. If $\text{foo} = 1$ and $\text{bar} = 1$ then increment $i$ by one and jump to the first line.

c. If $\text{foo} = 0$ and $\text{bar} = 1$ then decrement $i$ by one (unless it is already zero) and jump to the first line.

d. If $\text{foo} = 0$ and $\text{bar} = 0$ then halt and output $Y[0], \ldots, Y[m - 1]$ where $m$ is the smallest integer such that $Y_{\text{nonblank}}[m] = 0$.

6.2.2 Sneak peak: NAND-TM vs Turing machines

As the name implies, NAND-TM programs are a direct implementation of Turing machines in programming language form. We will show the equivalence below but you can already see how the components of Turing machines and NAND-TM programs correspond to one another:

<table>
<thead>
<tr>
<th>Turing Machine</th>
<th>NAND-TM program</th>
</tr>
</thead>
<tbody>
<tr>
<td>State: single register that takes values in $[k]$</td>
<td>Scalar variables: Several variables such as $\text{foo}$, $\text{bar}$ etc., each taking values in ${0, 1}$.</td>
</tr>
<tr>
<td>Tape: One tape containing values in a finite set $\Sigma$. Potentially infinite but $T[t]$ defaults to $\emptyset$ for all locations $t$ that have not been accessed.</td>
<td>Arrays: Several arrays such as $\text{Foo}$, $\text{Bar}$ etc., for each such array $\text{Arr}$ and index $j$, the value of $\text{Arr}$ at position $j$ is either 0 or 1. The value defaults to 0 for position that have not been written to.</td>
</tr>
<tr>
<td>Head location: A number $i \in \mathbb{N}$ that encodes the position of the head.</td>
<td>Index variable: The variable $i$ that can be used to access the arrays.</td>
</tr>
<tr>
<td>Accessing memory: At every step the Turing machine has access to its local state, but can only access the tape at the position of the current head location.</td>
<td>Accessing memory: At every step a NAND-TM program has access to all the scalar variables, but can only access the arrays at the location $i$ of the index variable</td>
</tr>
<tr>
<td>Control of location: In each step the machine can move the head location by at most one position.</td>
<td>Control of index variable: In each iteration of its main loop the program can modify the index $i$ by at most one.</td>
</tr>
</tbody>
</table>
6.2.3 Examples

We now present some examples of NAND-TM programs.

**Example 6.12 — XOR in NAND-TM.** The following is a NAND-TM program to compute the XOR function on inputs of arbitrary length. That is \(\text{XOR} : \{0, 1\}^* \rightarrow \{0, 1\}\) such that \(\text{XOR}(x) = \sum_{i=0}^{\lfloor |x| - 1 \rfloor} x_i \mod 2\) for every \(x \in \{0, 1\}^*\).

\[
\begin{align*}
temp_0 &= \text{NAND}(X[0],X[0]) \\
Y_{\text{nonblank}}[0] &= \text{NAND}(X[0],\text{temp}_0) \\
temp_2 &= \text{NAND}(X[i],Y[0]) \\
temp_3 &= \text{NAND}(X[i],\text{temp}_2) \\
temp_4 &= \text{NAND}(Y[0],\text{temp}_2) \\
Y[0] &= \text{NAND}(\text{temp}_3,\text{temp}_4) \\
&\text{MODANDJUMP}(X_{\text{nonblank}}[i],X_{\text{nonblank}}[i])
\end{align*}
\]

**Example 6.13 — Increment in NAND-TM.** We now present NAND-TM program to compute the *increment function*. That is, \(\text{INC} : \{0, 1\}^* \rightarrow \{0, 1\}^*\) such that for every \(x \in \{0, 1\}^n\), \(\text{INC}(x)\) is the \(n + 1\) bit long string \(y\) such that if \(X = \sum_{i=0}^{n-1} x_i \cdot 2^i\) is the number represented by \(x\), then \(y\) is the (least-significant digit first) binary representation of the number \(X + 1\).

We start by showing the program using the “syntactic sugar” we’ve seen before of using shorthand for some NAND-CIRC programs we have seen before to compute simple functions such as \(\text{IF}, \text{XOR}\) and \(\text{AND}\) (as well as the constant one function as well as the function \(\text{COPY}\) that just maps a bit to itself).

\[
\begin{align*}
carry &= \text{IF}(\text{started},\text{carry},\text{one}(\text{started})) \\
\text{started} &= \text{one}(\text{started}) \\
Y[i] &= \text{XOR}(X[i],\text{carry}) \\
\text{carry} &= \text{AND}(X[i],\text{carry}) \\
Y_{\text{nonblank}}[i] &= \text{one}(\text{started}) \\
&\text{MODANDJUMP}(X_{\text{nonblank}}[i],X_{\text{nonblank}}[i])
\end{align*}
\]

The above is not, strictly speaking, a valid NAND-TM program. If we “open up” all of the syntactic sugar, we get the following valid program to compute this syntactic sugar.

\[
\begin{align*}
temp_0 &= \text{NAND}(\text{started},\text{started}) \\
temp_1 &= \text{NAND}(\text{started},\text{temp}_0) \\
temp_2 &= \text{NAND}(\text{started},\text{started}) \\
temp_3 &= \text{NAND}(\text{temp}_1,\text{temp}_2)
\end{align*}
\]
temp_4 = \text{NAND}(\text{carry}, \text{started})
carry = \text{NAND}(\text{temp}_3, \text{temp}_4)
temp_6 = \text{NAND}(\text{started}, \text{started})
\text{started} = \text{NAND}(\text{started}, \text{temp}_6)
temp_8 = \text{NAND}(X[i], \text{carry})
temp_9 = \text{NAND}(X[i], \text{temp}_8)
temp_{10} = \text{NAND}(\text{started}, \text{temp}_8)
Y[i] = \text{NAND}(\text{temp}_9, \text{temp}_{10})
temp_{12} = \text{NAND}(X[i], \text{carry})
carry = \text{NAND}(\text{temp}_{12}, \text{temp}_{12})
temp_{14} = \text{NAND}(\text{started}, \text{started})
Y_{\text{nonblank}}[i] = \text{NAND}(\text{started}, \text{temp}_{14})
\text{MODANDJUMP}(X_{\text{nonblank}}[i], X_{\text{nonblank}}[i])

Working out the above two example can go a long way towards understanding the NAND-TM language. See the appendix and our [GitHub repository](https://github.com/boazbk/tcscode) for a full specification of the NAND-TM language.

6.3 EQUIVALENCE OF TURING MACHINES AND NAND-TM PROGRAMS

Given the above discussion, it might not be surprising that Turing machines turn out to be equivalent to NAND-TM programs. Indeed, we designed the NAND-TM language to have this property. Nevertheless, this is an important result, and the first of many other such equivalence results we will see in this book.

Theorem 6.14 — Turing machines and NAND-TM programs are equivalent. For every $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$, $F$ is computable by a NAND-TM program $P$ if and only if there is a Turing Machine $M$ that computes $F$.

Proof Idea:

To prove such an equivalence theorem, we need to show two directions. We need to be able to (1) transform a Turing machine $M$ to a NAND-TM program $P$ that computes the same function as $P$ and (2) transform a NAND-TM program $P$ into a Turing machine $M$ that computes the same function as $P$.

The idea of the proof is illustrated in Fig. 6.8. To show (1), given a Turing machine $M$, we will create a NAND-TM program $P$ that will have an array Tape for the tape of $M$ and scalar (i.e., non array)
variable(s) state for the state of $M$. Specifically, since the state of a Turing machine is not in \{0, 1\} but rather in a larger set $[k]$, we will use $\lceil \log k \rceil$ variables state_0, ..., state_$_{\lceil \log k \rceil} - 1$ variables to store the representation of the state. Similarly, to encode the larger alphabet $\Sigma$ of the tape, we will use $\lceil \log |\ Sigma| \rceil$ arrays Tape_0, ..., Tape_$_{\lceil \log |\ Sigma| \rceil} - 1$, such that the $i^{th}$ location of these arrays encodes the $i^{th}$ symbol in the tape for every tape. Using the fact that every function can be computed by a NAND-CIRC program, we will be able to compute the transition function of $M$, replacing moving left and right by decrementing and incrementing $i$ respectively.

We show (2) using very similar ideas. Given a program $P$ that uses $a$ array variables and $b$ scalar variables, we will create a Turing machine with about $2^b$ states to encode the values of scalar variables, and an alphabet of about $2^a$ so we can encode the arrays using our tape. (The reason the sizes are only “about” $2^a$ and $2^b$ is that we will need to add some symbols and steps for bookkeeping purposes.) The Turing Machine $M$ will simulate each iteration of the program $P$ by updating its state and tape accordingly.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6_8.png}
\caption{Comparing a Turing Machine to a NAND-TM program. Both have an unbounded memory component (the tape for a Turing machine, and the arrays for a NAND-TM program), as well as a constant local memory (state for a Turing machine, and scalar variables for a NAND-TM program). Both can only access at each step one location of the unbounded memory, this is the “head” location for a Turing machine, and the value of the index variable $i$ for a NAND-TM program.}
\end{figure}

Proof of Theorem 6.14. We start by proving the “if” direction of Theorem 6.14. Namely we show that given a Turing machine $M$, we can find a NAND-TM program $P_M$ such that for every input $x$, if $M$ halts on input $x$ with output $y$ then $P_M(x) = y$. Since our goal is just to show such a program $P_M$ exists, we don’t need to write out the full code of $P_M$ line by line, and can take advantage of our various “syntactic sugar” in describing it.

The key observation is that by Theorem 4.11 we can compute every finite function using a NAND-CIRC program. In particular, consider the transition function $\delta_M : [k] \times \Sigma \rightarrow [k] \times \Sigma \times \{L, R\}$ of our Turing Machine. We can encode the its components as follows:

- We encode $[k]$ using $\{0, 1\}^{\ell}$ and $\Sigma$ using $\{0, 1\}^{\ell'}$, where $\ell = \lceil \log k \rceil$ and $\ell' = \lceil \log |\Sigma| \rceil$. 

\begin{proof}[Proof of Theorem 6.14]
We start by proving the “if” direction of Theorem 6.14. Namely we show that given a Turing machine $M$, we can find a NAND-TM program $P_M$ such that for every input $x$, if $M$ halts on input $x$ with output $y$ then $P_M(x) = y$. Since our goal is just to show such a program $P_M$ exists, we don’t need to write out the full code of $P_M$ line by line, and can take advantage of our various “syntactic sugar” in describing it.

The key observation is that by Theorem 4.11 we can compute every finite function using a NAND-CIRC program. In particular, consider the transition function $\delta_M : [k] \times \Sigma \rightarrow [k] \times \Sigma \times \{L, R\}$ of our Turing Machine. We can encode the its components as follows:

- We encode $[k]$ using $\{0, 1\}^{\ell}$ and $\Sigma$ using $\{0, 1\}^{\ell'}$, where $\ell = \lceil \log k \rceil$ and $\ell' = \lceil \log |\Sigma| \rceil$. 

We encode the set \{L, R, S, H\} using \{0, 1\}². We will choose the encode L \mapsto 01, R \mapsto 11, S \mapsto 10, H \mapsto 00. (This conveniently corresponds to the semantics of the MODANDJUMP operation.)

Hence we can identify \(\delta_M\) with a function \(M : \{0, 1\}^{\ell + \ell'} \to \{0, 1\}^{\ell + \ell' + 2}\). By Theorem 4.11 there exists a finite length NAND-CIRC program \(\text{Compute}_M\) that computes this function \(M\). The NAND-TM program to simulate \(M\) will essentially be the following:

\[
\text{state, Tape}[i], \text{dir0, dir1} = \text{Compute}_M(\text{state, Tape}[i])
\]

\[
\text{MODANDJUMP}(\text{dir0, dir1})
\]

where we use state as shorthand for the tuple of variables \(\text{state}_0, ..., \text{state}_{\ell - 1}\) and Tape[1] as shorthand for \(\text{Tape}_0[i], ..., \text{Tape}_{\ell'} - 1[i]\). (We need to add a little “book keeping” to translate the output and the input from the tape to the X and Y variables, but the above is basically it.) Since every step of the main loop of the above program perfectly mimics the computation of the Turing Machine \(M\) as \(\text{Compute}_M\) computes the transition of the Turing Machine, and the program carries out exactly the definition of computation by a Turing Machine as per Definition 6.2.

For the other direction, suppose that \(P\) is a NAND-TM program with \(s\) lines, \(\ell\) scalar variables, and \(\ell'\) array variables. We will show that there exists a Turing machine \(M_P\) with \(2^\ell + C\) states and alphabet \(\Sigma\) of size \(C' + 2^\ell\) that computes the same functions as \(P\) (where \(C, C'\) are some constants to be determined later).

Specifically, consider the function \(\overline{P} : \{0, 1\}^\ell \times \{0, 1\}^{\ell'} \to \{0, 1\}^\ell \times \{0, 1\}^{\ell'}\) that on input the contents of \(P\)’s scalar variables and the contents of the array variables at location \(i\) in the beginning of an iteration, outputs all the new values of these variables at the last line of the iteration, right before the MODANDJUMP instruction is executed.

If foo and bar are the two variables that are used as input to the MODANDJUMP instruction, then this means that based on the values of these variables we can compute whether \(i\) will increase, decrease or stay the same, and whether the program will halt or jump back to the beginning. Hence a Turing machine can simulate an execution of \(P\) in one iteration using a finite function applied to its alphabet. The overall operation of the Turing machine will be as follows:

1. The machine \(M_P\) encodes the contents of the array variables of \(P\) in its tape, and the contents of the scalar variables in (part of) its state. Specifically, if \(P\) has \(\ell\) local variables and \(t\) arrays, then the state space of \(M\) will be large enough to encode all \(2^\ell\) assignments to the local variables and the alphabet \(\Sigma\) of \(M\) will be large enough
to encode all $2^t$ assignments for the array variables at each location. The head location corresponds to the index variable $i$.

2. Recall that every line of the program $P$ corresponds to reading and writing either a scalar variable, or an array variable at the location $i$. In one iteration of $P$ the value of $i$ remains fixed, and so the machine $M$ can simulate this iteration by reading the values of all array variables at $i$ (which are encoded by the single symbol in the alphabet $\Sigma$ located at the $i$-th cell of the tape), reading the values of all scalar variables (which are encoded by the state), and updating both. The transition function of $M$ can output $L, S, R$ depending on whether the values given to the MODANDJMP operation are $01, 10$ or $11$ respectively.

3. When the program halts (i.e., MODANDJMP gets $00$) then the Turing machine will enter into a special loop to copy the results of the $Y$ array into the output and then halt. We can achieve this by adding a few more states.

The above is not a full formal description of a Turing Machine, but our goal is just to show that such a machine exists. One can see that $M_P$ simulates every step of $P$, and hence computes the same function as $P$.

---

**Remark 6.15** — Running time equivalence (optional). If we examine the proof of Theorem 6.14 then we can see that the every iteration of the loop of a NAND-TM program corresponds to one step in the execution of the Turing machine. We will come back to this question of measuring number of computation steps later in this course. For now the main take away point is that NAND-TM programs and Turing Machines are essentially equivalent in power even when taking running time into account.

### 6.3.1 Specification vs implementation (again)

Once you understand the definitions of both NAND-TM programs and Turing Machines, Theorem 6.14 is fairly straightforward. Indeed, NAND-TM programs are not as much a different model from Turing Machines as they are simply a reformulation of the same model using programming language notation. You can think of the difference between a Turing machine and a NAND-TM program as the difference between representing a number using decimal or binary notation. In contrast, the difference between a function $F$ and a Turing machine
that computes $F$ is much more profound: it is like the difference between an equation $E$ and a number that is a solution for $E$. For this reason, while we take special care in distinguishing functions from programs or machines, we will often identify the two latter concepts. We will move freely between describing an algorithm as a Turing machine or as a NAND-TM program (as well as some of the other equivalent computational models we will see in Chapter 7 and beyond).

<table>
<thead>
<tr>
<th>Setting</th>
<th>Specification</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite computation</td>
<td>$\text{Functions mapping } {0, 1}^n \text{ to } {0, 1}^m$</td>
<td>Circuits, Straightline programs</td>
</tr>
<tr>
<td>Infinite computation</td>
<td>$\text{Functions mapping } {0, 1}^* \text{ to } {0, 1}^<em>$ or to ${0, 1}^</em>$.</td>
<td>Algorithms, Turing Machines, Programs</td>
</tr>
</tbody>
</table>

6.4 NAND-TM SYNTACTIC SUGAR

Just like we did with NAND-CIRC in Chapter 4, we can use “syntactic sugar” to make NAND-TM programs easier to write. For starters, we can use all of the syntactic sugar of NAND-CIRC, and so have access to macro definitions and conditionals (i.e., if/then). But we can go beyond this and achieve for example:

- Inner loops such as the while and for operations common to many programming languages
- Multiple index variables (e.g., not just $i$ but we can add $j$, $k$, etc.).
- Arrays with more than one dimension (e.g., $\text{Foo}[i][j]$, $\text{Bar}[i][j][k]$ etc.)

In all of these cases (and many others) we can implement the new feature as mere “syntactic sugar” on top of standard NAND-TM, which means that the set of functions computable by NAND-TM with this feature is the same as the set of functions computable by standard NAND-TM. Similarly, we can show that the set of functions computable by Turing Machines that have more than one tape, or tapes of more dimensions than one, is the same as the set of functions computable by standard Turing machines.

6.4.1 “GOTO” and inner loops

We can implement more advanced looping constructs than the simple MODANDJUMP. For example, we can implement GOTO. A GOTO statement
corresponds to jumping to a certain line in the execution. If our code has the form

```
"line1": do foo
  GOTO("line2")
"line2": do bar
"line3": do blah
```

then the program will only do foo and blah as when it reaches the line `GOTO("gohere")` it will jump to the line labeled with "gohere". We can achieve the effect of `GOTO` in NAND-TM using conditionals. In the code below, we assume that we have a variable `pc` that can take strings of some constant length. This can be encoded using a finite number of Boolean variables `pc_0`, `pc_1`, ..., `pc_{k - 1}`, and so when we write below `pc = "line1"` what we mean is something like `pc_0 = 0, pc_1 = 1, ...` (where the bits 0, 1, ... correspond to the encoding of the finite string "line1" as a string of length $k$). We also assume that we have access to conditional (i.e., if statements), which we can emulate using syntactic sugar in the same way as we did in NAND-CIRC.

To emulate a `GOTO` statement, we will first modify a program $P$ of the form

```
do foo
do bar
do blah
```

into the following form (using syntactic sugar for if):

```
pc = "line1"
if (pc=="line1"):
  do foo
  pc = "line2"
if (pc=="line2"):
  do bar
  pc = "line3"
if (pc=="line3"):
  do blah
```

These two programs do the same thing. The variable `pc` corresponds to the "program counter" and tells the program which line to execute next. We can see that if we wanted to emulate a `GOTO("line3")` then we could simply modify the instruction `pc = "line2"` to be `pc = "line3"`.

In NAND-CIRC we could only have `GOTO`s that go forward in the code, but since in NAND-TM everything is encompassed within a large outer loop, we can use the same ideas to implement `GOTO`'s that can go backwards, as well as conditional loops.
Other loops. Once we have GOTO, we can emulate all the standard loop constructs such as while, do, .. until or for in NAND-TM as well. For example, we can replace the code

```python
while foo:
    do blah
do bar

with

"loop":
    if NOT(foo): GOTO("next")
    do blah
    GOTO("loop")

"next":
    do bar
```

**Remark 6.16 — GOTO’s in programming languages.** The GOTO statement was a staple of most early programming languages, but has largely fallen out of favor and is not included in many modern languages such as Python, Java, Javascript. In 1968, Edsger Dijkstra wrote a famous letter titled “Go to statement considered harmful.” (see also Fig. 6.9). The main trouble with GOTO is that it makes analysis of programs more difficult by making it harder to argue about invariants of the program.

When a program contains a loop of the form:

```python
for j in range(100):
    do something
do blah
```

you know that the line of code do blah can only be reached if the loop ended, in which case you know that j is equal to 100, and might also be able to argue other properties of the state of the program. In contrast, if the program might jump to do blah from any other point in the code, then it’s very hard for you as the programmer to know what you can rely upon in this code. As Dijkstra said, such invariants are important because “our intellectual powers are rather geared to master static relations and .. our powers to visualize processes evolving in time are relatively poorly developed” and so “we should … do … our utmost best to shorten the conceptual gap between the static program and the dynamic process.”

That said, GOTO is still a major part of lower level languages where it is used to implement higher level looping constructs such as while and for loops.
For example, even though Java doesn’t have a GOTO statement, the Java Bytecode (which is a lower level representation of Java) does have such a statement. Similarly, Python bytecode has instructions such as POP_JUMP_IF_TRUE that implement the GOTO functionality, and similar instructions are included in many assembly languages. The way we use GOTO to implement a higher level functionality in NAND-TM is reminiscent of the way these various jump instructions are used to implement higher level looping constructs.

6.4.2 Extended example: well formed programs (optional)

The notion of passing between different variants of programs can be extremely useful, as often, given a program $P$ that we want to analyze, it would be simpler for us to first modify it to an equivalent program $P'$ that has some convenient properties. You can think of this as the NAND-TM equivalent of enforcing “coding conventions” that are often used for programming languages. For example, while this is not part of the Python language, Google’s Python style guide stipulates that variables that are initialized to a value and never changed (i.e., constants) are typed with all capital letters. (Similar requirements are used in other style guides.) Of course this does not really restrict the power of Google-conforming Python programs, since every Python program can be transformed to an equivalent one that satisfies this requirement. In fact, many programming languages have automatic programs known as linters that can detect and sometimes modify the program to fit certain standards.

The following solved exercise is an example of that. We will define the notion of a well-formed program and show that every NAND-TM program can be transformed into an equivalent one that is well formed.

**Definition 6.17 — Well-formed programs.** We say that a NAND-TM program $P$ is well formed if it satisfies the following properties:

- Every reference to a variable in $P$ either has the form `foo` or `foo_123` (a scalar variable: alphanumerical string starting with a lowercase letter and no brackets) or the form `Bar[i]` or `Bar_12[i]` (an array variable alphanumerical string starting with a capital letter and ending with `[i]`).

- $P$ contains the scalar variables `zero`, `one`, `dir0`, and `dir1` such that `zero` and `one` are always the constants 0 and 1 respectively, and MODANDJUMP is always invoked with the variables `dir0` and `dir1`. 
The following exercise shows that we can transform every NAND-TM program \( P \) into a well-formed program \( P' \) that is equivalent to it. Hence if we are given a NAND-TM program \( P \), we can (and will) often assume without loss of generality that it is well-formed.

\textbf{Lemma 6.18} For every NAND-TM program \( P \), there exists an NAND-TM program \( P' \) equivalent to \( P \) that is well formed as pre Definition 6.17. That is, for every input \( x \in \{0, 1\}^* \), either both \( P \) and \( P' \) do not halt on \( x \), or both \( P \) and \( P' \) halt on \( x \) and produce the same output \( y \in \{0, 1\}^* \).

\textbf{Solved Exercise 6.1 — Making an NAND-TM program well formed.} Prove Lemma 6.18

\begin{quote}
As usual, I would recommend you try to solve this exercise yourself before looking up the solution.
\end{quote}

\textbf{Solution:}

Since variable identifiers on their own have no meaning in (enhanced) NAND-TM (other than the special ones \( X, X_{\text{nonblank}}, Y, Y_{\text{nonblank}} \) that already have the desired properties), we can easily achieve the property that scalars variables start with lowercase and arrays with uppercase using “search and replace”. We just have to take care that we don’t make two distinct identifiers become the same. For example, we can do so by changing all scalar variable identifiers to lower case, and adding to them the prefix \texttt{scalar}_-, and adding the prefix \texttt{Array}_- to all array variable identifiers.

The property that an array variable is never references with a numerical index is more challenging. We need to remove all references to an array variable with an actual numerical index rather than \( i \). One thought might be to simply convert a a reference of the form \texttt{Arr[i]} to the scalar variable \texttt{arr_i}. However, this will not necessarily preserve the functionality of the program. The reason is that we want to ensure that when \( i = 17 \) then \texttt{Arr[i]} would give us the same value as \texttt{arr_17}.
Nevertheless, we can use the approach above with a slight twist. We will demonstrate the solution in a concrete case. (Needless to say, if you needed to solve this question in a problem set or an exam, such a demonstration of a special case would not be sufficient; but this example should be good enough for you to extrapolate a full solution.) Suppose that there are only three references to array variables with numerical indices in the program: Foo[5], Bar[12] and Blah[22]. We will include three scalar variables foo_5, bar_12 and blah_22 which will serve as a cache for the values of these arrays. We will change all references to Foo[5] to foo_5, Bar[12] to bar_12 and so on and so forth. But in addition to that, whenever in the code we refer to Foo[i] we will check if \(i = 5\) and if so use the value foo_5 instead, and similarly with Bar[i] or Blah[i].

Specifically, we will change our program as follows. We will create an array Is_5 such that \(Is_5[i] = 1\) if and only if \(i = 5\), and similarly create arrays Is_12, Is_22.

We can then change code of the following form

\[
\text{Foo}[i] = \text{something}
\]

to

\[
\text{temp} = \text{something} \\
\text{foo}_5 = \text{IF}(Is_5[i], \text{temp}, \text{foo}_5) \\
\text{Foo}[i] = \text{temp}
\]

and similarly code of the form

\[
\text{blah} = \text{NAND}(\text{Bar}[i], \text{baz})
\]

to

\[
\text{temp} = \text{IF}(Is_22[i], \text{bar}_22, \text{Bar}[i]) \\
\text{blah} = \text{NAND}(\text{temp}, \text{baz})
\]

To create the arrays we can add code of the following form in the beginning of the program (here we’re using enhanced NAND-TM syntax, GOTO, and the constant one but this syntactic sugar can of course be avoided):

\[
\# \text{ initialization of arrays} \\
\text{GOTO("program body", init_done)} \\
i += \text{one} \\
i += \text{one}
\]
i += one
i += one
i += one
Is_5[i] = one
i += one
...
# repeat i += one 6 more times
Is_12[i] = one
i += one
...
# repeat i += one 9 more times
Is_22[i] = one
i -= one
...
# repeat i -= one 21 more times
init_done = one

LABEL("program body")
original code of program...

Using IF statements (which can easily be replaced with syntactic sugar) we can handle the conditions that loop, Y, and Y_nonblank are not written to once loop is set to 0. We leave completing all the details as an exercise to the reader (see ??).

6.5 UNIFORMITY, AND NAND VS NAND-TM (DISCUSSION)

While NAND-TM adds extra operations over NAND-CIRC, it is not exactly accurate to say that NAND-TM programs or Turing machines are “more powerful” than NAND-CIRC programs. NAND-CIRC programs, having no loops, are simply not applicable for computing functions with a bounded number of inputs. Thus, to compute a function \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \) using NAND-CIRC (or equivalently, Boolean circuits) we need a collection of programs/circuits: one for every input length.

The key difference between NAND-CIRC and NAND-TM is that NAND-TM allows us to express the fact that the algorithm for computing parities of length-100 strings is really the same one as the algorithm for computing parities of length-5 strings (or similarly the fact that the algorithm for adding \( n \)-bit numbers is the same for every \( n \), etc.). That is, one can think of the NAND-TM program for general parity as the “seed” out of which we can grow NAND-CIRC programs for length 10, length 100, or length 1000 parities as needed.

This notion of a single algorithm that can compute functions of all input lengths is known as uniformity of computation and hence we think of Turing machines / NAND-TM as uniform model of computation, as opposed to Boolean circuits or NAND-CIRC which is a
nonuniform model, where we have to specify a different program for every input length.

Looking ahead, we will see that this uniformity leads to another crucial difference between Turing machines and circuits. Turing machines can have inputs and outputs that are longer than the description of the machine as a string and in particular there exists a Turing machine that can “self replicate” in the sense that it can print its own code. This notion of “self replication”, and the related notion of “self reference” is crucial to many aspects of computation, as well of course to life itself, whether in the form of digital or biological programs.

For now, what you ought to remember is the following differences between uniform and nonuniform computational models:

- **Non uniform computational models**: Examples are NAND-CIRC programs and Boolean circuits. These are models where each individual program/circuit can compute a finite function \( f : \{0, 1\}^n \rightarrow \{0, 1\}^m \). We have seen that every finite function can be computed by some program/circuit. To discuss computation of an infinite function \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \) we need to allow a sequence \( \{P_n\}_{n \in \mathbb{N}} \) of programs/circuits (one for every input length), but this does not capture the notion of a single algorithm to compute the function \( F \).

- **Uniform computational models**: Examples are Turing machines and NAND-TM programs. These are models where a single program/machine can take inputs of arbitrary length and hence compute an infinite function \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \). The number of steps that a program/machine takes on some input is not a priori bounded in advance and in particular there is a chance that it will enter into an infinite loop. Unlike the nonuniform case, we have not shown that every infinite function can be computed by some NAND-TM program/Turing Machine. We will come back to this point in Chapter 8.

**Lecture Recap**

- Turing machines capture the notion of a single algorithm that can evaluate functions of every input length.
- They are equivalent to NAND-TM programs, which add loops and arrays to NAND-CIRC.
- Unlike NAND-CIRC or Boolean circuits, the number of steps that a Turing machine takes on a given input is not fixed in advance. In fact, a Turing machine or a NAND-TM program can enter into an infinite loop on certain inputs, and not halt at all.
6.6 EXERCISES

Exercise 6.1 — Explicit NAND TM programming. Produce the code of a (syntactic-sugar free) NAND-TM program $P$ that computes the (unbounded input length) Majority function $Maj : \{0, 1\}^* \rightarrow \{0, 1\}$ where for every $x \in \{0, 1\}^*$, $Maj(x) = 1$ if and only if $\sum_{i=0}^{\lvert x \rvert} x_i > \lvert x \rvert / 2$. We say “produce” rather than “write” because you do not have to write the code of $P$ by hand, but rather can use the programming language of your choice to compute this code.

Exercise 6.2 Prove the following closure properties of the set $R$ defined in Definition 6.5:

1. If $F \in R$ then the function $G(x) = 1 - F(x)$ is in $R$.
2. If $F, G \in R$ then the function $H(x) = F(x) \lor G(x)$ is in $R$.
3. If $F \in R$ then the function $F^*$ in in $R$ where $F^*$ is defined as follows: $F^*(x) = 1$ iff there exist some strings $w_0, \ldots, w_{k-1}$ such that $x = w_0 w_1 \cdots w_{k-1}$ and $F(w_i) = 1$ for every $i \in [k]$.
4. If $F \in R$ then the function $G(x) = \begin{cases} \exists y \in \{0, 1\} \mid F(xy) = 1 & 0 \\ 0 & \text{otherwise} \end{cases}$ (6.3) is in $R$.

Exercise 6.3 — Single vs multiple bit. Prove that for every $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$, the function $F$ is computable if and only if the following function $G : \{0, 1\}^* \rightarrow \{0, 1\}$ is computable, where $G$ is defined as follows:

$$G(x, i, \sigma) = \begin{cases} F(x)_i & i < |F(x)|, \sigma = 0 \\ 1 & i < |F(x)|, \sigma = 1 \\ 0 & i \geq |F(x)| \end{cases}$$

Exercise 6.4 — Uncomputability via counting. Recall that $R$ is the set of all total functions from $\{0, 1\}^*$ to $\{0, 1\}$ that are computable by a Turing machine (see Definition 6.5). Prove that $R$ is countable. That is, prove that there exists a one-to-one map $DiN : R \rightarrow \mathbb{N}$. You can use the equivalence between Turing machines and NAND-TM programs.

Exercise 6.5 — Not every function is computable. Prove that the set of all total functions from $\{0, 1\}^*$ to $\{0, 1\}$ is not countable. You can use the results of Section 2.3.1. (We will see an explicit uncomputable function in Chapter 8.)
6.7 BIBLIOGRAPHICAL NOTES

Augusta Ada Byron, countess of Lovelace (1815-1852) lived a short but turbulent life, though is today most well known for her collaboration with Charles Babbage (see [Ste87] for a biography). Ada took an immense interest in Babbage’s analytical engine, which we mentioned in Chapter 3. In 1842-3, she translated from Italian a paper of Menabrea on the engine, adding copious notes (longer than the paper itself) including some examples of programs. Because of these programs, some call Ada “the first computer programmer” though recent evidence shows they were likely written by Babbage himself [Hol01]. Regardless, Ada was clearly one of very few people (perhaps the only one outside of Babbage himself) to fully appreciate how significant and revolutionary the idea of mechanizing computation truly is.

The books of Shetterly [She16] and Sobel [Sob17] discuss the history of human computers (which were more often than not women) and their important contributions to scientific discoveries.

Alan Turing was one of the intellectual giants of the 20th century. He was not only the first person to define the notion of computation, but also invented and used some of the world’s earliest computational devices as part of the effort to break the Enigma cipher during World War II, saving millions of lives. Tragically, Turing committed suicide in 1954, following his conviction in 1952 for homosexual acts and a court-mandated hormonal treatment. In 2009, British prime minister Gordon Brown made an official public apology to Turing, and in 2013 Queen Elizabeth II granted Turing a posthumous pardon. Turing’s life is the subject of a great book and a mediocre movie.

One of the first programming-language formulations of Turing machines was given by Wang [Wan57]. Our formulation is aimed at making the connection with circuits more direct, with the eventual goal of using it for the Cook-Levin Theorem, as well as results such as $\mathbf{P} \subseteq \mathbf{P} / \text{poly}$ and $\mathbf{BPP} \subseteq \mathbf{P} / \text{poly}$. The website esolangs.org features a large variety of esoteric Turing-complete programming languages. One of the most famous of them is Brainf*ck.

In this book, we use the terminology of functions as opposed to languages. These are easily related by identifying a language $L \subseteq \{0, 1\}^*$ with its characteristic function $1_L : \{0, 1\} \rightarrow \{0, 1\}$ defined as $1_L(x) = 1$ iff $x \in L$. The characteristic function of a language is always a total function. The language-like object that corresponds to a partial function is known as a promise problem.