7

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

As the quote of Chapter 6 says, an algorithm is “a finite answer to an infinite number of questions”. To express an algorithm, we need to write down a finite set of instructions that will enable us to compute on arbitrarily long inputs. To describe and execute an algorithm we need the following components (see Fig. 7.1):

- The finite set of instructions to be performed.
- Some “local variables” or finite state used in the execution.
- A potentially unbounded working memory to store the input and any other values we may require later.
- While the memory is unbounded, at every single step we can only read and write to a finite part of it, and we need a way to address which are the parts we want to read from and write to.
- If we only have a finite set of instructions but our input can be arbitrarily long, we will need to repeat instructions (i.e., loop back). We need a mechanism to decide when we will loop and when we will halt.

This chapter: A non-mathy overview

In this chapter, we give a general model of an algorithm, which (unlike Boolean circuits) is not restricted to a fixed

Learning Objectives:

- Learn the model of Turing machines, which can compute functions of arbitrary input lengths.
- See a programming-language description of Turing machines, using NAND-TM programs, which add loops and arrays to NAND-CIRC.
- See some basic syntactic sugar and equivalence of variants of Turing machines and NAND-TM programs.
input length, and (unlike finite automata) is not restricted to a finite amount of working memory. We will see two ways to model algorithms:

- **Turing machines**, invented by Alan Turing in 1936, are hypothetical abstract devices that yield finite descriptions of algorithms that can handle arbitrarily long inputs.

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

It turns out that these two models are equivalent. In fact, they are equivalent to many other computational models, including programming languages such as C, Lisp, Python, JavaScript, etc. This notion, known as Turing equivalence or Turing completeness, will be discussed in Chapter 8. See Fig. 7.2 for an overview of the models presented in this chapter and Chapter 8.

**Figure 7.2:** Overview of our models for finite and unbounded computation. In the previous chapters we study the computation of finite functions, which are functions \( f : \{0,1\}^n \rightarrow \{0,1\}^m \) for some fixed \( n, m \), and modeled computing these functions using circuits or straight-line programs. In this chapter we study computing unbounded functions of the form \( F : \{0,1\}^* \rightarrow \{0,1\}^m \) or \( F : \{0,1\}^* \rightarrow \{0,1\}^* \). We model computing these functions using Turing Machines or (equivalently) NAND-TM programs, which add the notion of loops to the NAND-CIRC programming language. In Chapter 8 we will show that these models are equivalent to many other models, including RAM machines, the \( \lambda \) calculus, and all the common programming languages including C, Python, Java, JavaScript, etc.

### 7.1 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

_”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 “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. 7.4) 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). The paper is divided into “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. Based on the contents of this cell, the person can update their finite mental state, and/or move to the cell immediately to the left or right of the current one.

Turing modeled such a computation by a “machine” that maintains one of \( k \) states. At each point in time the machine reads from its “work tape” a single symbol from a finite alphabet \( \Sigma \) and uses that to update its state, write to tape, and possibly move to an adjacent cell (see Fig. 7.7). To compute a function \( F \) using this machine, we initialize the tape with the input \( x \in \{0, 1\}^* \) and our goal is to ensure that the tape will contain the value \( F(x) \) at the end of the computation. 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 \( \triangleright, x_0, \ldots, x_{n-1}, \emptyset, \emptyset, \ldots \). We use the symbol \( \triangleright \) to denote the beginning of the tape, and the symbol \( \emptyset \) to denote an empty cell. We will always assume that the alphabet \( \Sigma \) is a (potentially strict) superset of \( \{\triangleright, \emptyset, 0, 1\} \).
- 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. Based on this symbol and its state \( s \), the machine 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, its output is the binary string obtained by reading the tape from the beginning until the first location in which it contains a $\varnothing$ symbol, and then outputting all 0 and 1 symbols in sequence, dropping the initial $\triangleright$ symbol if it exists, as well as the final $\varnothing$ symbol.

7.1.1 Extended example: A Turing machine for palindromes
Let $PAL$ (for 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 $PAL$. To specify $M$ we need to specify (i) $M$’s tape alphabet $\Sigma$ which should contain at least the symbols $0, 1, \triangleright, \varnothing$, 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 $\{0, 1, \triangleright, \varnothing, \times\}$ and will have $k = 14$ states. Though the states are simply numbers between 0 and $k - 1$, we will give them the following labels for convenience:

<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>$0_{-AND} _BLANK$</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 START and goes right, looking for the first symbol that is 0 or 1. If it finds $\varnothing$ before it hits such a symbol then we it moves to the OUTPUT_1 state described below.
• Once $M$ finds such a symbol $b \in \{0, 1\}$, $M$ deletes $b$ from the tape by writing the $\times$ symbol, it enters either the RIGHT_0 or RIGHT_1 mode according to the value of $b$ and starts moving rightwards until it hits the first $\emptyset$ or $\times$ symbol.

• Once $M$ finds this symbol, it goes into the state LOOK_FOR_0 or LOOK_FOR_1 depending on whether it was in the state RIGHT_0 or RIGHT_1 and makes one left move.

• In the state LOOK_FOR_$b$, $M$ checks whether the value on the tape is $b$. If it is, then $M$ deletes it by changing its value to $\times$, and moves to the state RETURN. Otherwise, it changes to the OUTPUT_0 state.

• The RETURN state means that $M$ goes back to the beginning. Specifically, $M$ moves leftward until it hits the first symbol that is not 0 or 1, in which case it changes its state to START.

• The OUTPUT_$b$ states mean that $M$ will eventually output the value $b$. In both the OUTPUT_0 and OUTPUT_1 states, $M$ goes left until it hits $\triangleright$. Once it does so, it makes a right step, and changes to the 1_AND_BLANK or 0_AND_BLANK states respectively. In the latter states, $M$ writes the corresponding value, moves right and changes to the BLANK_AND_STOP state, in which it writes $\emptyset$ to the tape and halts.

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.

### 7.1.2 Turing machines: a formal definition

The formal definition of Turing machines is as follows:

**Rules / transition function:** $M : \mathcal{A} \times \Sigma \to \mathcal{A} \times \Sigma \times \{L, R, S, H\}$

- If read $0$ and state is 17 then change state to 29, write 1, and move left.
- If read $0$ and state is 23 then change state to 15, write 0, and move right.
- If read $\triangleright$ and state is 8 then change state to 12, write $\triangleright$, and halt.

...
Definition 7.1 — 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:

- 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 \( i > n \).)
- We also initialize \( i = 0 \) and \( s = 0 \).
- 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 \( i \leftarrow \max\{i - 1, 0\} \).
     (If \( D = S \) then we keep \( i \) the same.)
  4. If \( D = H \), then halt.
- If the process above halts, then \( M \)'s output, denoted by \( M(x) \), is the string \( y \in \{0, 1\}^* \) obtained by concatenating all the symbols in \( \{0, 1\} \) in positions \( T[0], \ldots, T[i] \) where \( i + 1 \) is the first location in the tape containing \( \emptyset \).
- If The Turing machine does not halt then we denote \( M(x) = \bot \).

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.

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. However, 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. However, these definitions describe the same computational process and have the same computational powers. Hence they are equivalent despite their superficial differences. See Section 7.7 for a comparison between Definition 7.1 and the way Turing Machines are defined in texts such as Sipser [Sip97].

### 7.1.3 Computable functions

We now turn to make one of the most important definitions in this book: computable functions.

**Definition 7.2 — Computable functions.** Let \( F : \{0, 1\}^* \to \{0, 1\}^* \) 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 8, being computable in the sense of Definition 7.2 is equivalent to being computable in virtually any reasonable model of computation. This statement is known as the **Church-Turing Thesis**. (Unlike the extended Church-Turing Thesis which we discussed in Section 5.6, the Church-Turing thesis itself is widely believed and there are no candidate devices that attack it.)

**Big Idea 9** We can precisely define what it means for a function to be computable by *any possible algorithm*.

This is a good point to remind the reader that **functions are not** the same as **programs**:\[
\text{Functions } \neq \text{ Programs} . \tag{7.1}
\]

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 \( F : \{0, 1\}^* \to \{0, 1\} \) that have a single bit of output. Hence we give a special name for the set of computable functions of this form.
A partial function \( F \) from a set \( A \) to a set \( B \) is a function that is only defined on a subset of \( A \), (see Section 1.4.3). We can also think of such a function as mapping \( A \) to \( B \cup \{\bot\} \) where \( \bot \) is a special “failure” symbol such that \( F(a) = \bot \) indicates the function \( F \) is not defined on \( a \).

**Definition 7.3 — The class \( R \).** We define \( R \) be the set of all computable functions \( F : \{0, 1\}^* \to \{0, 1\} \).

**Remark 7.4 — Functions vs. languages.** As discussed in Section 6.1.2, many texts use the terminology of “languages” rather than functions to refer to computational tasks. 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 \( F : \{0, 1\}^* \to \{0, 1\} \) defined as \( F(x) = 1 \) if \( x \in L \). A language \( L \) is decidable if there is a Turing machine \( M \) that decides it. For historical reasons, some texts also call such languages 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 7.3.

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 \( F : \{0, 1\}^* \to \{0, 1\} \) and the language \( L = \{x \in \{0, 1\}^* \mid F(x) = 1\} \).

### 7.1.4 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 \( P \) 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 a Turing machine \( M \), we cannot determine a priori the length of \( M \)’s output. In fact, we don’t even know if an output would be produced at all! For example, it is straightforward 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 an input \( x \), then it cannot compute any total function \( F \), since clearly on input \( x \), \( M \) will fail to output \( F(x) \). However, \( M \) can still compute a partial function.\(^1\)

For example, consider the partial function \( \text{DIV} \) 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 \( \text{DIV} \) 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 \( \text{DIV} \) is undefined on such inputs. If \( a = 0 \) and \( b = 0 \), the machine \( M \)

\(^1\) A partial function \( F \) from a set \( A \) to a set \( B \) is a function that is only defined on a subset of \( A \), (see Section 1.4.3). We can also think of such a function as mapping \( A \) to \( B \cup \{\bot\} \) where \( \bot \) is a special “failure” symbol such that \( F(a) = \bot \) indicates the function \( F \) is not defined on \( a \).
will output 0, which is also OK, since we don’t care about what the program outputs on inputs on which \( \text{DIV} \) is undefined. Formally, we define computability of partial functions as follows:

**Definition 7.5 — 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, \( M(x) = F(x) \). We say that a (partial or total) function \( F \) is computable if there is a Turing machine that computes it.

Note that if \( F \) is a total function, then it is defined on every \( x \in \{0, 1\}^* \) and hence in this case, **Definition 7.5** is identical to **Definition 7.2**.

**Remark 7.6 — Bot symbol.** We often use \( \bot \) 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 \( M(x) = \bot \). This does not mean that \( M \) outputs some encoding of the symbol \( \bot \) but rather that \( M \) enters into an infinite loop when given \( x \) as input.

If a partial function \( F \) is undefined on \( x \) then we can also write \( F(x) = \bot \). Therefore one might think that **Definition 7.5** 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 \( \$ \)s’s on which \( F \) is defined: on other inputs it is OK for \( M \) to output an arbitrary value such as 0, 1, or anything else, or not to halt at all. To borrow a term from the C programming language, on inputs \( x \) on which \( F \) is not defined, what \( M \) does is “undefined behavior”.

### 7.2 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 7.1.1** that computes the function \( \text{PAL} \) such that \( \text{PAL}(x) = 1 \) iff \( x \) is a palindrome.

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

```
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
```

The precise details of this program are not important. What matters 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 holds 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 similar to the following:

```
def M(Tape):
    state = 0
    i = 0  # holds head location
    while (True):
        # Move head, modify state, write to tape
        # based on current state and cell at head
        # below are just examples for how program looks
        # for a particular transition function
        if Tape[i]=='0' and state==7:  #
            δ_M(7,0)=(19,1",R")
            i += 1
            Tape[i]="1"
            state = 19
        elif Tape[i]==">" and state == 13:  #
            δ_M(13,">")=(15,0",S")
```

2 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 $\varnothing$ 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.
Tape[i]="0"
state = 15
elif ... 
... 
elif Tape[i]==">" and state == 29: #
   δ_M(29,">")=(.,.,"H")
   break # Halt

If we wanted to use only Boolean (i.e., 0/1-valued) variables, then we can encode the state variables using \(\lceil \log k \rceil\) bits. Similarly, we can represent each element of the alphabet \(\Sigma\) using \(\ell = \lceil \log |\Sigma| \rceil\) bits and hence we can replace the \(\Sigma\)-valued array Tape[] with \(\ell\) Boolean-valued arrays Tape0[], ..., Tape(\(\ell - 1\))[].

7.2.1 The NAND-TM Programming language
We now introduce the \(NAND-TM\) programming language, which captures the power of a Turing machine with a programming-language formalism. 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 use a fixed-length program to encode 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 Foo_17 or Bar[22] in NAND-CIRC, they are not true arrays, since the number in the identifier is a constant that is “hardwired” into the program. NAND-TM contains actual arrays that can have a length that is not a priori bounded.

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

\[
NAND-TM = NAND-CIRC + loops + arrays \tag{7.2}
\]

**Remark 7.7** — NAND-CIRC + loops + arrays = everything.. As we will see, adding loops and arrays to NAND-CIRC is enough to capture the full power of
Figure 7.8: A NAND-TM program has scalar variables that can take a Boolean value, array variables that hold a sequence of Boolean values, and a special index variable $i$ that can be used to index the array variables. We refer to the $i$-th value of the array variable $Spam$ using $Spam[i]$. At each iteration of the program the index variable can be incremented or decremented by one step using the MODANDJMP operation.

Concretely, the NAND-TM programming language adds the following features on top of NAND-CIRC (see Fig. 7.8):

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

- Apart from $i$ NAND-TM has two kinds of variables: scalars and arrays. Scalar variables hold one bit (just as in NAND-CIRC). Array variables hold an unbounded number of bits. At any point in the computation we can access the array variables at the location indexed by $i$ using $\text{Foo}[i]$. We cannot access the arrays at locations other than the one pointed to by $i$.

- 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 $\text{Foo}$ is an array and $\text{bar}$ is a scalar variable.

- The input and output $X$ and $Y$ are now considered arrays with values of zeroes and ones. (There are also two other special arrays $X_{\text{nonblank}}$ and $Y_{\text{nonblank}}$, see below.)

- 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 $\text{MODANDJUMP}(a, b)$ increments $i$ by one and jumps to the first line of the program.
  - If $a = 0$ and $b = 1$ then $\text{MODANDJUMP}(a, b)$ 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.)

all programming languages! Hence we could replace “NAND-TM” with any of Python, C, Javascript, OCaml, etc. in the left-hand side of (7.2). But we’re getting ahead of ourselves: this issue will be discussed in Chapter 8.
- 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.

- The MODANDJUMP instruction always appears in the last line of a NAND-TM program and nowhere else.

**Default values.** We need one more convention to handle “default values”. Turing machines have the special symbol $\square$ to indicate that tape location is “blank” or “uninitialized”. In NAND-TM there is no such symbol, and all variables are Boolean, containing either 0 or 1. All variables and locations of arrays 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 initialized. In particular, we will use this convention for the input and output arrays $X$ and $Y$. A NAND-TM program has four special arrays $X, X_{\text{nonblank}}, Y,$ and $Y_{\text{nonblank}}$. 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 7.8 — NAND-TM programs.** A NAND-TM program consists of a sequence of lines of the form $\text{foo} = \text{NAND(bar,blah)}$ ending with a line of the form MODANDJMP(foo, bar), where foo, bar, blah are either scalar variables (sequences of letters, digits, and underscores) or array variables of the form Foo[i] (starting with capital letters and indexed by i). The program has the array variables $X, X_{\text{nonblank}}, Y, Y_{\text{nonblank}}$ and the index variable $i$ built in, and can use additional array and scalar variables.

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 MODAND-
JMP(\texttt{foo, bar}) is executed we do as follows:

a. If \texttt{foo} = 1 and \texttt{bar} = 0, jump to the first line without modifying the value of \texttt{i}.

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

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

d. If \texttt{foo} = 0 and \texttt{bar} = 0, halt and output \texttt{Y[0]}, \ldots, \texttt{Y[m - 1]} where \( m \) is the smallest integer such that \texttt{Y\_nonblank[m]} = 0.

7.2.2 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 7.2: Turing Machine and NAND-TM analogs

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

### 7.2.3 Examples

We now present some examples of NAND-TM programs.

#### Example 7.9 — Increment in NAND-TM.

The following is a 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 describing the program using “syntactic sugar” for NAND-CIRC for the \( \text{IF} \), \( \text{XOR} \) and \( \text{AND} \) functions (as well as the constant one function, and the function \( \text{COPY} \) that just maps a bit to itself).

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

Since we used syntactic sugar, the above is not, strictly speaking, a valid NAND-TM program. However, by “opening up” all the syntactic sugar, we get the following “sugar free” valid program to compute the same function.

\[
\begin{align*}
\text{temp}_0 &= \text{NAND}(\text{started}, \text{started}) \\
\text{temp}_1 &= \text{NAND}(\text{started}, \text{temp}_0) \\
\text{temp}_2 &= \text{NAND}(\text{started}, \text{started}) \\
\text{temp}_3 &= \text{NAND}(\text{temp}_1, \text{temp}_2) \\
\text{temp}_4 &= \text{NAND}(\text{carry}, \text{started}) \\
\text{carry} &= \text{NAND}(\text{temp}_3, \text{temp}_4) \\
\text{temp}_6 &= \text{NAND}(\text{started}, \text{started}) \\
\text{started} &= \text{NAND}(\text{started}, \text{temp}_6) \\
\text{temp}_8 &= \text{NAND}(X[i], \text{carry}) \\
\text{temp}_9 &= \text{NAND}(X[i], \text{temp}_8)
\end{align*}
\]
Example 7.10 — 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 \( \text{XOR}(x) = \sum_{i=0}^{\|x\|-1} x_i \mod 2 \) for every \( x \in \{0, 1\}^* \). Once again, we use a certain “syntactic sugar”. Specifically, we access the arrays \( X \) and \( Y \) at their zero-th entry, while NAND-TM only allows access to arrays in the coordinate of the variable \( i \).

To transform the program above to a valid NAND-TM program, we can transform references such as \( X[0] \) and \( Y[0] \) to scalar variables \( x_0 \) and \( y_0 \) (similarly we can transform any reference of the form Foo[17] or Bar[15] to scalars such as foo_17 and bar_15). We then need to add code to load the value of \( X[0] \) to \( x_0 \) and similarly to write to \( Y[0] \) the value of \( y_0 \), but this is not hard to do. Using the fact that variables are initialized to zero by default, we can create a variable init which will be set to 1 at the end of the first iteration and not changed since then. We can then add an array Atzero and code that will modify Atzero[i] to 1 if init is 0 and otherwise leave it as it is. This will ensure that Atzero[i] is equal to 1 if and only if \( i \) is set to zero, and allow the program to know when we are at the zeroth location. Thus we can add code to read and write to the corresponding scalars \( x_0, y_0 \) when we are at the zeroth location, and also code to move \( i \) to zero and then halt at the end. Working this out fully is somewhat tedious, but can be a good exercise.
7.3 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 a significant result, and the first of many other such equivalence results we will see in this book.

Theorem 7.11 — 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 \( M \) 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. 7.9. To show (1), given a Turing machine \( M \), we will create a NAND-TM program \( P \) that will have an array \( \text{Tape} \) for the tape of \( M \) and scalar (i.e., non-array) variable(s) \( \text{state} \) 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 \( \text{state}_0, ..., \text{state}_{\lceil \log k \rceil} \) 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 \( \text{Tape}_0, ..., \text{Tape}_{\lceil \log |\Sigma| \rceil} \), 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 \( 2^a \) states to encode the values of scalar variables, and an alphabet of about \( 2^b \) so we can encode the arrays using our tape. (The reason the sizes are only “about” \( 2^a \) and \( 2^b \) is that we need to
add some symbols and steps for bookkeeping purposes.) The Turing machine \( M \) simulates each iteration of the program \( P \) by updating its state and tape accordingly.

*Figure 7.9: 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.*

**Proof of Theorem 7.11.** We start by proving the “if” direction of Theorem 7.11. 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 4.12 we can compute every finite function using a NAND-CIRC program. In particular, consider the transition function \( \delta_M : [k] \times \Sigma \rightarrow [k] \times \Sigma \times \{L, R, S, H\} \) of our Turing machine. We can encode 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 encoding \(L \mapsto 01, R \mapsto 11, S \mapsto 10, H \mapsto 00\). (This conveniently corresponds to the semantics of the MODANDJUMP operation.)

Hence we can identify \(\delta_M\) with a function \(\overline{M} : \{0, 1\}^{\ell+\ell'} \rightarrow \{0, 1\}^{\ell+\ell'+2}\), mapping strings of length \(\ell + \ell'\) to strings of length \(\ell + \ell' + 2\). By Theorem 4.12 there exists a finite length NAND-CIRC program \(\text{ComputeM}\) that computes this function \(\overline{M}\). The NAND-TM program to simulate \(M\) will essentially be the following:
Algorithm 7.12 — NAND-TM program to simulate TM $M$.

Input: $x \in \{0, 1\}^*$
Output: $M(x)$ if $M$ halts on $x$. Otherwise go into infinite loop

1: # We use variables $\text{state}_0 \ldots \text{state}_{\ell - 1}$ to encode $M$'s state
2: # We use arrays $\text{Tape}_0[\ldots]$ $\text{Tape}_{\ell'}[\ldots]$ to encode $M$’s tape
3: # We omit the initial and final "book keeping" to copy
Input: to $\text{Tape}$ and copy
Output: from $\text{Tape}$

4: # Use the fact that transition is finite and computable by NAND-CIRC program:
5: $\text{state}_0 \ldots \text{state}_{\ell - 1}$, $\text{Tape}_0[i] \ldots$ $\text{Tape}_{\ell'}[i]$, $\text{dir}_0, \text{dir}_1 \leftarrow \text{TRANSITION}(\text{state}_0 \ldots \text{state}_{\ell - 1}$, $\text{Tape}_0[i] \ldots$ $\text{Tape}_{\ell'}[i], \text{dir}_0, \text{dir}_1)$
6: MODANDJMP($\text{dir}_0, \text{dir}_1$)

Every step of the main loop of the above program perfectly mimics the computation of the Turing machine $M$, and so the program carries out exactly the definition of computation by a Turing machine as per Definition 7.1.

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 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 7.13 — Running time equivalence (optional).** If we examine the proof of Theorem 7.11 then we can see that 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 the 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.

### 7.3.1 Specification vs implementation (again)

Once you understand the definitions of both NAND-TM programs and Turing machines, Theorem 7.11 is 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 the equation $x^2 + x = 12$, and the number 3 that is a solution for this equation. 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 8 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 ${0,1}^n$ to ${0,1}^m$</td>
<td>Circuits, Straightline programs</td>
</tr>
<tr>
<td>Infinite computation</td>
<td>$\text{Functions}$ mapping ${0,1}^\ast$ to ${0,1}$ or to ${0,1}^\ast$.</td>
<td>Algorithms, Turing Machines, Programs</td>
</tr>
</tbody>
</table>

### 7.4 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, such as macro definitions and conditionals (i.e., if/then). However, 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. This 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.
7.4.1 “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 specific line in the execution. For example, if we have code of the form

```
"start": do foo
    GOTO("end")
"skip": do bar
"end": do blah
```

then the program will only do foo and blah as when it reaches the line GOTO("end") it will jump to the line labeled with "end". 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 \( pc_0, pc_1, \ldots, pc_{k-1} \), and so when we write below \( pc = \text{"label"} \) what we mean is something like \( pc_0 = 0, pc_1 = 1, \ldots \) (where the bits 0, 1, ... correspond to the encoding of the finite string "label" 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
```

to have 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 \( pc = \text{"line2"} \) to be \( pc = \text{"line3"} \).
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 backward, 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

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

### Remark 7.14 — 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. 7.10). 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
Introduction to Theoretical Computer Science

Figure 7.10: XKCD’s take on the GOTO statement.

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.

7.5 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 or Boolean circuits. NAND-CIRC programs, having no loops, are simply not applicable for computing functions with an unbounded 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. Hence we think of Turing machines / NAND-TM as uniform models of computation, as opposed to Boolean circuits or NAND-CIRC, which are non-uniform models, in which 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. The notion of “self replication”, and the related notion of “self reference” are crucial to many aspects of computation, and beyond that 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 non-uniform 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 non-uniform 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 9.

---

### 7.6 EXERCISES

**Exercise 7.1 — 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 \( \sum_{i=0}^{\left| x \right|} x_i > \left| x \right| / 2 \). 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 7.2 — Computable functions examples.** Prove that the following
functions are computable. For all of these functions, you do not have
to fully specify the Turing machine or the NAND-TM program that
computes the function, but rather only prove that such a machine or
program exists:

1. $INC : \{0, 1\}^* \to \{0, 1\}^*$ which takes as input a representation of a
natural number $n$ and outputs the representation of $n + 1$.
2. $ADD : \{0, 1\}^* \to \{0, 1\}^*$ which takes as input a representation of
a pair of natural numbers $(n, m)$ and outputs the representation of
$n + m$.
3. $MULT : \{0, 1\}^* \to \{0, 1\}^*$, which takes a representation of a pair of
natural numbers $(n, m)$ and outputs the representation of $nm$.
4. $SORT : \{0, 1\}^* \to \{0, 1\}^*$ which takes as input the representation of
a list of natural numbers $(a_0, \ldots, a_{n-1})$ and returns its sorted version
$(b_0, \ldots, b_{n-1})$ such that for every $i \in [n]$ there is some $j \in [n]$ with
$b_i = a_j$ and $b_0 \leq b_1 \leq \cdots \leq b_{n-1}$.

**Exercise 7.3 — Two index NAND-TM.** Define NAND-TM’ to be the variant of
NAND-TM where there are two index variables $i$ and $j$. Arrays can be
indexed by either $i$ or $j$. The operation $MODANDJMP$ takes four variables
$a, b, c, d$ and uses the values of $c, d$ to decide whether to increment $j$
decrement $j$ or keep it in the same value (corresponding to 01, 10, and
00 respectively). Prove that for every function $F : \{0, 1\}^* \to \{0, 1\}^*$, $F$
is computable by a NAND-TM program if and only if $F$ is computable
by a NAND-TM’ program.

**Exercise 7.4 — Two tape Turing machines.** Define a two tape Turing machine
to be a Turing machine which has two separate tapes and two separate
heads. At every step, the transition function gets as input the location
of the cells in the two tapes, and can decide whether to move each
head independently. Prove that for every function $F : \{0, 1\}^* \to \{0, 1\}^*$, $F$
is computable by a standard Turing machine if and only if $F$ is computable
by a two-tape Turing machine.

**Exercise 7.5 — Two dimensional arrays.** Define NAND-TM” to be the vari-
ant of NAND-TM where just like NAND-TM’ defined in Exercise 7.3
loops and infinity 279

You can use the sequence \( R, L, R, R, L, L, R, R, L, L, \ldots \).

Exercise 7.6 — Two dimensional Turing machines. Define a two-dimensional Turing machine to be a Turing machine in which the tape is two dimensional. At every step the machine can move Up, Down, Left, Right, or Stay. Prove that for every function \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \), \( F \) is computable by a standard Turing machine if and only if \( F \) is computable by a two-dimensional Turing machine.

Exercise 7.7 Prove the following closure properties of the set \( R \) defined in Definition 7.3:

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 \( x = w_0 w_1 \ldots w_{k-1} \) and \( F(w_i) = 1 \) for every \( i \in [k] \).

4. If \( F \in R \) then the function

\[
G(x) = \begin{cases} 
\exists y \in \{0, 1\}^n : F(xy) = 1 & \text{otherwise} \end{cases} \tag{7.3}
\]

is in \( R \).

Exercise 7.8 — Oblivious Turing Machines (challenging). Define a Turing machine \( M \) to be oblivious if its head movements are independent of its input. That is, we say that \( M \) is oblivious if there exists an infinite sequence \( MOVE \in \{L, R, S\}^\infty \) such that for every \( x \in \{0, 1\}^* \), the movements of \( M \) when given input \( x \) (up until the point it halts, if such point exists) are given by \( MOVE_0, MOVE_1, MOVE_2, \ldots \).

Prove that for every function \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \), if \( F \) is computable then it is computable by an oblivious Turing machine. See footnote for hint.\(^3\)

Exercise 7.9 — 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 func-
tion \( G : \{0, 1\}^* \rightarrow \{0, 1\} \) 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 7.10 — 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 7.3). Prove that \( R \) is countable. That is, prove that there exists a one-to-one map \( D_{TN} : R \rightarrow \mathbb{N} \). You can use the equivalence between Turing machines and NAND-TM programs.

Exercise 7.11 — Not every function is computable. Prove that the set of all total functions from \( \{0, 1\}^* \rightarrow \{0, 1\} \) is not countable. You can use the results of Section 2.4. (We will see an explicit uncomputable function in Chapter 9.)

7.7 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). The quote in the chapter’s beginning is taken from Nota A in this text. Lovelace’s notes contain several examples of programs for the analytical engine, and because of this she has been called “the world’s first computer programmer” though it is not clear whether they were written by Lovelace or Babbage himself [Hol01]. Regardless, Ada was clearly one of very few people (perhaps the only one outside of Babbage himself) to fully appreciate how important and revolutionary the idea of mechanizing computation truly is.

The books of Shetterly [She16] and Sobel [Sob17] discuss the history of human computers (who were female, more often than not) and their important contributions to scientific discoveries in astronomy and space exploration.

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.

Sipser’s text [Sip97] defines a Turing machine as a seven tuple consisting of the state space, input alphabet, tape alphabet, transition function, starting state, accepting state, and rejecting state. Superficially this looks like a very different definition than Definition 7.1 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.

One difference is that Sipser considers a general set of states $Q$ that is not necessarily of the form $Q = \{0, 1, 2, \ldots, k - 1\}$ for some natural number $k > 0$. Sipser also restricts his 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 presents the basic task of a Turing machine as that of deciding a language as opposed to computing a function, but these are equivalent, see Remark 7.4.)

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 $\log \lceil|\Sigma|\rceil$. 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 staying in place is very easy to emulate by simply moving right and then back left.

Another definition used in the literature is that a Turing machine $M$ recognizes a language $L$ if for every $x \in L$, $M(x) = 1$ and for every $x \notin L$, $M(x) \in \{0, \bot\}$. A language $L$ is recursively enumerable if there exists a Turing machine $M$ that recognizes it, and the set of all recursively enumerable languages is often denoted by $\text{RE}$. We will not use this terminology in this book.

One of the first programming-language formulations of Turing machines was given by Wang [Wan57]. Our formulation of NAND-TM 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 $\text{P} \subseteq \text{P/poly}$ and $\text{BPP} \subseteq \text{P/poly}$. The website esolangs.org features a large variety of esoteric Turing-complete programming languages. One of the most famous of them is Brain*f*ck.