7

Equivalent models of computation

“All problems in computer science can be solved by another level of indirection”, attributed to David Wheeler.

“Because we shall later compute with expressions for functions, we need a distinction between functions and forms and a notation for expressing this distinction. This distinction and a notation for describing it, from which we deviate trivially, is given by Church.”, John McCarthy, 1960 (in paper describing the LISP programming language)

So far we have defined the notion of computing a function based on Turing machines, which don’t really correspond to the way computation is done in practice. In this chapter we justify this choice by showing that the definition of computable functions will remain the same under a wide variety of computational models. This notion is known as Turing completeness or Turing equivalence and is one of the most fundamental facts of computer science. In fact, a widely believed claim known as the Church-Turing Thesis holds that every “reasonable” definition of computable function is equivalent to being computable by a Turing machine. We will discuss the Church-Turing Thesis and the potential definitions of “reasonable” in Section 7.8.

7.1 RAM MACHINES AND NAND-RAM

One of the limitations of NAND-TM (and Turing machines) is that we can only access one location of our arrays/tape at a time. If $i = 22$ and we want to access $\text{Foo}[957]$ then it will take us at least 923 steps to get there. In contrast, almost every programming language has a formalism for directly accessing memory locations. Actual physical computers also provide so called Random Access Memory (RAM) which can be thought of as a large array Memory, such that given an in-
Figure 7.1: Some Turing-equivalent models. All of these are equivalent in power to Turing Machines (or equivalently NAND-TM programs) in the sense that they can compute exactly the same class of functions. All of these are models for computing infinite functions that take inputs of unbounded length. In contrast, Boolean circuits / NAND-CIRC programs can only compute finite functions and hence are not Turing complete.

1 “Random access memory” is quite a misnomer, since it has nothing to do with probability. Indexed access would have been more appropriate. However, the term “random access” is standard in both the theoretical and practical literature, and hence we will use it as well.

Figure 7.2: A RAM Machine contains a finite number of local registers, each of which holds an integer, and an unbounded memory array. It can perform arithmetic operations on its register as well as load to a register the contents of the memory at the address indexed by the number in register $r'$.
We will not give a formal definition of RAM Machines, though the
bibliographical notes section (Section 7.10) contains sources for such
definitions. Just as the NAND-TM programming language models
Turing machines, we can also define a NAND-RAM programming lan-
guage that models Turing machines. The NAND-RAM programming
language extends NAND-TM by adding the following features:

- The variables of NAND-RAM are allowed to be (non negative)
  integer valued rather than only Boolean as is the case in NAND-TM.
  That is, a scalar variable foo holds an non negative integer in \( \mathbb{N} \)
  (rather than only a bit in \( \{0, 1\} \)), and an array variable Bar holds
  an array of integers. As in the case of RAM machines, we will not
  allow integers of unbounded size. Concretely, each variable holds a
  number between 0 and \( T \), where \( T \) is the number of steps that have
  been executed by the program so far.\(^2\)

- We allow indexed access to arrays. If foo is a scalar and Bar is an
  array, then Bar[foo] refers to the location of Bar indexed by the
  value of foo. (Note that this means we don’t need to have a special
  index variable i any more.)

- As is often the case in programming languages, we will assume
  that for Boolean operations such as \texttt{NAND}, a zero valued integer is
  considered as false, and a nonzero valued integer is considered as
  true.

- In addition to \texttt{NAND}, NAND-RAM also includes all the basic arith-
  metic operations of addition, subtraction, multiplication, (integer)
  division, as well as comparisons (equal, greater than, less than,
  etc.)

- We will also include as part of the language basic control flow
  structures such as \texttt{if} and \texttt{goto}.

A full description of the NAND-RAM programming language is
in the appendix. However, the most important fact you need to know
about NAND-RAM is that you actually don’t need to know much
about NAND-RAM at all, since it is equivalent in power to Turing
machines:

\textbf{Theorem 7.1 — Turing Machines (aka NAND-TM programs) and RAM ma-
chines (aka NAND-RAM programs) are equivalent.} For every function
\( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \), \( F \) is computable by a NAND-TM program if
and only if \( F \) is computable by a NAND-RAM program.

\textbf{Proof Idea:}
Clearly NAND-RAM is only more powerful than NAND-TM, and so if a function $F$ is computable by a NAND-TM program then it can be computed by a NAND-RAM program. The challenging direction is of course to transform a NAND-RAM program $P$ to an equivalent NAND-TM program $Q$. To describe the proof in full we will need to cover the full formal specification of the NAND-RAM language, and show how we can implement every one of its features as syntactic sugar on top of NAND-TM.

This can be done but going over all the operations in detail is rather tedious. Hence we will focus on describing the main ideas behind this transformation. The transformation has two steps:

1. **Indexed access of bit arrays**: NAND-RAM generalizes NAND-TM in two main ways: (a) adding indexed access to the arrays (i.e., `Foo[bar]` syntax) and (b) moving from *Boolean valued* variables to *integer valued* ones. We will start by showing how to handle (a). Namely, we will show how we can implement in NAND-TM the operation $\text{Setindex}(\text{Bar})$ such that if $\text{Bar}$ is an array that encodes some integer $j$, then after executing $\text{Setindex}(\text{Bar})$ the value of $i$ will equal to $j$. This will allow us to simulate syntax of the form `Foo[Bar]` by $\text{Setindex}(\text{Bar})$ followed by $\text{Foo}[i]$.

2. **Two dimensional bit arrays**: We will then show how we can use “syntactic sugar” to augment NAND-TM with *two dimensional arrays*. That is, have two indices $i$ and $j$ and two dimensional arrays, such that we can use the syntax $\text{Foo}[i][j]$ to access the $(i,j)$-th location of $\text{Foo}$.

3. **Arrays of integers**: Finally we will encode a one dimensional array $\text{Arr}$ of integers by a two dimensional $\text{Arrbin}$ of bits. The idea is simple: if $a_{i,0}, \ldots, a_{i,\ell}$ is a binary (prefix-free) representation of $\text{Arr}[i]$, then $\text{Arrbin}[i][j]$ will be equal to $a_{i,j}$.

Once we have arrays of integers, we can use our usual syntactic sugar for functions, $\text{GOTO}$ etc. to implement the arithmetic and control flow operations of NAND-RAM.

\begin{remark}
\textbf{Remark 7.2} — RAM machines / NAND-RAM and assembly language (optional). RAM machines correspond quite closely to actual microprocessors such as those in the Intel x86 series that also contains a large primary memory and a constant number of small registers. This is of course no accident: RAM machines aim at modeling more closely than Turing machines the architecture of actual computing systems, which largely follows the
 Interestingly, the fact that actual microprocessors also have a fixed number of registers (e.g., 14 general purpose registers in x86-64) does not make a big difference with RAM machines. It can be shown that RAM machines with as few as two registers are as powerful as full-fledged RAM machines that have an arbitrarily large constant number of register.

As a result, NAND-RAM is similar in its general outline to assembly languages such as x86 or NIPS. These assembly languages all have instructions to (1) move data from registers to memory, (2) perform arithmetic or logical computations on registers, and (3) conditional execution and loops (“if” and “goto”, commonly known as “branches” and “jumps” in the context of assembly languages).

The main difference between RAM machines and actual microprocessors (and correspondingly between NAND-RAM and assembly languages) is that actual microprocessors have a fixed word size \( w \) so that all registers and memory cells hold numbers in \([2^w]\) (or equivalently strings in \(\{0, 1\}^w\)). This number \( w \) can vary among different processors, but common values are either 32 or 64. As a theoretical model, RAM machines do not have this limitation, but we rather let \( w \) be the logarithm of our running time (which roughly corresponds to its value in practice as well). Of course actual microprocessors have many features not shared with RAM machines as well, including parallelism, memory hierarchies, and many others. However, RAM machines do capture actual computers to a first approximation and so (as we will see), the running time of an algorithm on a RAM machine (e.g., \( O(n) \) vs \( O(n^2) \)) is strongly correlated with its practical efficiency.

7.2 THE GORY DETAILS (OPTIONAL)

We do not show the full formal proof of Theorem 7.1 but focus on the most important parts: implementing indexed access, and simulating two dimensional arrays with one dimensional ones. Even these are already quite tedious to describe, as will not be surprising to anyone that has ever written a compiler. Hence you can feel free to merely skim this section. The important point is not for you to know all details by heart but to be convinced that in principle it is possible to transform a NAND-RAM program to an equivalent NAND-TM program, and even be convinced that, with sufficient time and effort, you could do it if you wanted to.

7.2.1 Indexed access in NAND-TM

Let us choose some prefix-free representation for the natural numbers (see Section 2.4.2). For example, if a natural number \( k \) is equal to \( \sum_{i=0}^{\ell} k_i \cdot 2^i \) for \( \ell = \lceil \log k \rceil \), then we can represent it as the string \((k_\ell, k_{\ell-1}, \ldots, k_1, k_0, 1, 0)\).

To implement indexed access in NAND-TM, we need to be able to do the following. Given an array \( \text{Bar} \), implement to operation
SetIndex(Bar) that will set i to the value encoded by Bar. This can be achieved as follows:

1. Set i to zero, by decrementing it until we reach the point where 
   \( \text{Atzero}[i] = 1 \) (where \( \text{Atzero} \) is an array that has 1 only in position 0).

2. Let Temp be an array encoding the number 0.

3. While the number encoded by Temp differs from the number encoded by Bar:
   
   a. Increment Temp
   
   b. GOTO the same line (but pass a NAND-TM iteration along the way, performing a MODANDJUMP operation that increments i by one).

   At the end of the loop, i is equal to the value at Bar, and so we can use this to read or write to arrays at the location corresponding to this value. In code, we can implement the above operations as follows:

```plaintext
# assume Atzero is an array such that Atzero[0]=1
# and Atzero[j]=0 for all j>0

# set i to 0.
LABEL("zero_idx")
dir0 = zero
dir1 = one
# corresponds to i <- i-1
GOTO("zero_idx",NOT(Atzero[i]))

...  

# zero out temp  
#(code below assumes a specific prefix-free encoding in which 10 is the "end marker")
Temp[0] = 1
Temp[1] = 0

# set i to Bar, assume we know how to increment, compare
LABEL("increment_temp")
cond = EQUAL(Temp,Bar)
dir0 = one
dir1 = one
# corresponds to i <- i+1
INC(Temp)
GOTO("increment_temp",cond)
```


# if we reach this point, i is number encoded by Bar

...

# final instruction of program

MODANDJUMP(dir0, dir1)

## 7.2.2 Two-dimensional arrays in NAND-TM

To implement two-dimensional arrays, we want to embed them in a one-dimensional array. The idea is to come up with a one-to-one function \( \text{embed} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \), and so embed the location \((i, j)\) of the two-dimensional array \( \text{Two} \) in the location \( \text{embed}(i, j) \) of the array \( \text{One} \).

Since the set \( \mathbb{N} \times \mathbb{N} \) seems “much bigger” than the set \( \mathbb{N} \), a priori it might not be clear that such a one-to-one mapping exists. However, once you think about it more, it is not that hard to construct. For example, you could ask a child to use scissors and glue to transform a 10” by 10” piece of paper into a 1” by 100” strip. If you think about it, this is essentially a one-to-one map from \([10] \times [10]\) to \([100]\). We can generalize this to obtain a one-to-one map from \([n] \times [n]\) to \([n^2]\) and more generally a one-to-one map from \(\mathbb{N} \times \mathbb{N}\) to \(\mathbb{N}\). Specifically, the following map \( \text{embed} \) would do (see Fig. 7.3):

\[
\text{embed}(x, y) = \frac{1}{2}(x + y)(x + y + 1) + x .
\] (7.1)

We ask you to prove that \( \text{embed} \) is indeed one-to-one, as well as computable by a NAND-TM program, in Exercise 7.2.

So, we can replace code of the form \( \text{Two}[\text{Foo}][\text{Bar}] = \text{something} \) (i.e., access the two-dimensional array \( \text{Two} \) at the integers encoded by the one-dimensional arrays \( \text{Foo} \) and \( \text{Bar} \)) by code of the form:

\[
\begin{align*}
\text{Blah} &= \text{embed}(\text{Foo}, \text{Bar}) \\
\text{Setindex}(\text{Blah}) \\
\text{Two}[i] &= \text{something}
\end{align*}
\]

Computing \( \text{embed} \) is left for you the reader as Exercise 7.2, but let us hint that this can be done by simply following the grade-school algorithms for multiplication, addition, and division.

## 7.2.3 All the rest

Once we have two-dimensional arrays and indexed access, simulating NAND-RAM with NAND-TM is just a matter of implementing the standard algorithms for arithmetic operations and comparators in NAND-TM. While this is cumbersome, it is not difficult, and the end result is to show that every NAND-RAM program \( P \) can be simulated by an equivalent NAND-TM program \( Q \), thus completing the proof of Theorem 7.1.
7.3 TURING EQUIVALENCE (DISCUSSION)

Any of the standard programming languages such as C, Java, Python, Pascal, Fortran have very similar operations to NAND-RAM. (Indeed, ultimately they can all be executed by machines which have a fixed number of registers and a large memory array.) Hence using Theorem 7.1, we can simulate any program in such a programming language by a NAND-TM program. In the other direction, it is a fairly easy programming exercise to write an interpreter for NAND-TM in any of the above programming languages. Hence we can also simulate NAND-TM programs (and so by Theorem 6.14, Turing machines) using these programming languages. This property of being equivalent in power to Turing Machines / NAND-TM is called Turing Equivalent (or sometimes Turing Complete). Thus all programming languages we are familiar with are Turing equivalent.4

Remark 7.3 — Recursion in NAND-RAM (advanced). One concept that appears in many programming languages but we did not include in NAND-RAM programs is recursion. However, recursion (and function calls in general) can be implemented in NAND-RAM using the stack data structure. A stack is a data structure containing a sequence of elements, where we can “push” elements into it and “pop” them from it in “first in last out” order.

We can implement a stack using an array of integers Stack and a scalar variable stackpointer that will be the number of items in the stack. We implement push(foo) by

\[
\text{Stack[stackpointer]} = \text{foo} \\
\text{stackpointer} += \text{one}
\]

and implement bar = pop() by

\[
\text{bar} = \text{Stack[stackpointer]} \\
\text{stackpointer} -= \text{one}
\]

We implement a function call to \( F \) by pushing the arguments for \( F \) into the stack. The code of \( F \) will “pop” the arguments from the stack, perform the computation (which might involve making recursive or non recursive calls) and then “push” its return value into the stack. Because of the “first in last out” nature of a stack, we do not return control to the calling procedure until all the recursive calls are done.

The fact that we can implement recursion using a non-recursive language is not surprising. Indeed, machine languages typically do not have recursion (or function calls in general), and hence a compiler implements function calls using a stack and GOTO. You can find

4 Some programming language have fixed (even if extremely large) bounds on the amount of memory they can access, which formally prevent them from being applicable to computing infinite functions and hence simulating Turing machines. We ignore such issues in this discussion and assume access to some storage device without a fixed upper bound on its capacity.
7.3.1 The “Best of both worlds” paradigm

The equivalence between Turing Machines and RAM machines allows us to choose the most convenient language for the task at hand:

- When we want to prove a theorem about all programs/algorithms, we can use Turing machines (or NAND-TM) since they are simpler and easier to analyze. In particular, if we want to show that a certain function cannot be computed, then we will use Turing machines.

- When we want to show that a function can be computed we can use RAM machines or NAND-RAM, because they are easier to program in and correspond more closely to high level programming languages we are used to. In fact, we will often describe NAND-RAM programs in an informal manner, trusting that the reader can fill in the details and translate the high level description to the precise program. (This is just like the way people typically use informal or “pseudocode” descriptions of algorithms, trusting that their audience will know to translate these descriptions to code if needed.)

Our usage of Turing Machines / NAND-TM and RAM Machines / NAND-RAM is very similar to the way people use in practice high and low level programming languages. When one wants to produce a device that executes programs, it is convenient to do so for very simple and “low level” programming language. When one wants to describe an algorithm, it is convenient to use as high level a formalism as possible.

**Big Idea 8** Using equivalence results such as those between Turing and RAM machines, we can “have our cake and eat it too”. We can use a simpler model such as Turing machines when we want to prove something can’t be done, and use a feature-rich model such as RAM machines when we want to prove something can be done.
7.3.2 Let’s talk about abstractions.

“The programmer is in the unique position that ... he has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before.”, Edsger Dijkstra, “On the cruelty of really teaching computing science”, 1988.

At some point in any theory of computation course, the instructor and students need to have the talk. That is, we need to discuss the level of abstraction in describing algorithms. In algorithms courses, one typically describes algorithms in English, assuming readers can “fill in the details” and would be able to convert such an algorithm into an implementation if needed. For example, we might describe the breadth first search algorithm to find if two vertices $u, v$ are connected as follows:

Algorithm 7.4 — Breadth First Search.

Input: Graph $G$, vertices $u, v$

Operation:

1. Put $u$ in queue $Q$.
2. While $Q$ is not empty:
   - Remove the top vertex $w$ from $Q$
   - If $w = v$ then declare “connected” and exit.
   - Mark $w$ and add all unmarked neighbors of $w$ to $Q$.
3. Declare “unconnected”.

We call such a description a high level description.

If we wanted to give more details on how to implement breadth first search in a programming language such as Python or C (or NAND-RAM / NAND-TM for that matter), we would describe how we implement the queue data structure using an array, and similarly how we would use arrays to implement the marking. We call such an “intermediate level” description an implementation level or pseudocode description. Finally, if we want to describe the implementation precisely, we would give the full code of the program (or another fully precise representation, such as in the form of a list of tuples). We call this a formal or low level description.

While we started off by describing NAND-CIRC, NAND-TM, and NAND-RAM programs at the full formal level, as we progress in this book we will move to implementation and high level description. After all, our goal is not to use these models for actual computation,
Figure 7.6: We can describe an algorithm at different levels of granularity/detail and precision. At the highest level we just write the idea in words, omitting all details on representation and implementation. In the intermediate level (also known as implementation or pseudocode) we give enough details of the implementation that would allow someone to derive it, though we still fall short of providing the full code. The lowest level is where the actual code or mathematical description is fully spelled out. These different levels of detail all have their uses, and moving between them is one of the most important skills for a computer scientist.

but rather to analyze the general phenomenon of computation. That said, if you don’t understand how the high level description translates to an actual implementation, going “down to the metal” is often an excellent exercise. (One of the most important skills for a computer scientist is the ability to move up and down hierarchies of abstractions.)

A similar distinction applies to the notion of representation of objects as strings. Sometimes, to be precise, we give a low level specification of exactly how an object maps into a binary string. For example, we might describe an encoding of \( n \) vertex graphs as length \( n^2 \) binary strings, by saying that we map a graph \( G \) over the vertices \( [n] \) to a string \( x \in \{0, 1\}^{n^2} \) such that the \( n \cdot i + j \)-th coordinate of \( x \) is 1 if and only if the edge \( i \to j \) is present in \( G \). We can also use an intermediate or implementation level description, by simply saying that we represent a graph using the adjacency matrix representation.

Finally, because we are translating between the various representations of graphs (and objects in general) can be done via a NAND-RAM (and hence a NAND-TM) program, when talking in a high level we also suppress discussion of representation altogether. For example, the fact that graph connectivity is a computable function is true regardless of whether we represent graphs as adjacency lists, adjacency matrices, list of edge-pairs, and so on and so forth. Hence, in cases where the precise representation doesn’t make a difference, we would often talk about our algorithms as taking as input an object \( O \) (that can be a graph, a vector, a program, etc.) without specifying how \( O \) is encoded as a string.
7.3.3 Definition of “Algorithm”

Up until now we have use the term “algorithm” informally. However, Turing Machines and the range of equivalent models give a way to precisely and formally define algorithms. Hence whenever we refer to an algorithm in this book, we will mean that it is an instance of one of the Turing equivalent models, such as Turing machines, NAND-TM, RAM machines, etc. Because of the equivalence of all these models, in many contexts, it will not matter which of these we use.

7.4 Lambda Calculus and Functional Programming Languages

The λ calculus is another way to define computable functions. It was proposed by Alonzo Church in the 1930’s around the same time as Alan Turing’s proposal of the Turing Machine. Interestingly, while Turing Machines are not used for practical computation, the λ calculus has inspired functional programming languages such as LISP, ML and Haskell, and indirectly the development of many other programming languages as well. In this section we will present the λ calculus and show that its power is equivalent to NAND-TM programs (and hence also to Turing machines). Our Github repository contains a Jupyter notebook with a Python implementation of the λ calculus that you can experiment with to get a better feel for this topic.

The λ operator. At the core of the λ calculus is a way to define “anonymous” functions. For example, instead of defining the squaring function as

\[
square(x) = x \times x
\]  

(7.2)

we can write it as

\[
\lambda x. x \times x
\]  

(7.3)

and so \((\lambda x. x \times x)(7) = 49\). That is, you can think of \(\lambda x. exp(x)\), where \(exp\) is some expression as a way of specifying the anonymous function \(x \mapsto exp(x)\). Clearly, the name of the argument to a function doesn’t matter, and so \(\lambda y. y \times y\) is the same as \(\lambda x. x \times x\), as both correspond to the squaring function.

Remark 7.5 — Dropping parenthesis. To reduce notational clutter, when writing a calculus expression we often drop the parenthesis for function evaluation. Hence instead of writing \(f(x)\) for the result of applying the function \(f\) to the input \(x\), we can also write this as simply \(fx\). Therefore we can write \((\lambda x. x \times x)7 = 49\). In this chapter, we will use both the \(f(x)\) and \(fx\) no-

5 Anonymous functions, using either \(\lambda x. f(x), x \mapsto f(x)\) or other closely related notation, appear in many programming languages. For example, in Python we can define the squaring function using \(\lambda x: x**2\); while in JavaScript we can use \(x \mapsto x**2\) or \((x) \Rightarrow x**2\). In Scheme we would define it as \((\lambda x: (* x x))\).
7.4.1 Applying functions to functions

A key feature of the $\lambda$ calculus is that we can use functions as arguments to other functions. For example, can you guess what number is the following expression equal to?

$$(((\lambda f.(\lambda y.(f (f y))))((\lambda x.x \times x)) 3)$$

(7.4)

The expression (7.4) might seem daunting, but before you look at the solution below, try to break it apart to its components, and evaluate each component at a time. Working out this example would go a long way toward understanding the $\lambda$ calculus.

Let’s evaluate (7.4) one step at a time. As nice as it is for the $\lambda$ calculus to allow anonymous functions, adding names can be very helpful for understanding complicated expressions. So, let us write $F = \lambda f.(\lambda y.(f (f y)))$ and $g = \lambda x.x \times x$.

Therefore (7.4) becomes

$$((F g) 3).$$

(7.5)

On input a function $f$, $F$ outputs the function $\lambda y.(f (f y))$, or in other words $F f$ is the function $y \mapsto f(f(y))$. Our function $g$ is simply $g(x) = x^2$ and so $(F g)$ is the function that maps $y$ to $(y^2)^2$ or in other words to $y^4$. Hence $((F g) 3) = 3^4 = 81$.

Example 7.6 — Simplifying a $\lambda$ expression. Here is another example of a $\lambda$ expression:

$$(((\lambda x. (\lambda y.x)) 2) 9).$$

(7.6)

Let us denote $(\lambda y.x)$ by $F$. Then (7.6) has the form

$$((\lambda x.F) 2) 9).$$

(7.7)

Now $(\lambda x.F) 2$ is equal to $F[x \mapsto 2]$. Since $F$ is $\lambda y.x$ this means that $(\lambda x.F) 2$ is the function $\lambda y.2$ that ignores its input and outputs 2 no matter what it is equal to. Hence (7.6) is equivalent to $(\lambda y.2) 9$. 

which is the result of applying the function \( y \mapsto 2 \) on the input 9, which is simply the number 2.

### 7.4.2 Obtaining multi-argument functions via Currying

As we’ve seen, in a \( \lambda \) expression of the form \( \lambda x.e \), the expression \( e \) can itself involve the \( \lambda \) operator. Thus for example the function

\[
\lambda x. (\lambda y. x + y)
\]

maps \( x \) to the function \( y \mapsto x + y \).

In particular, if we invoke the function (7.8) on \( a \), and then invoke the result of this invocation on \( b \), we will get the value \( a + b \). We can see that the one-argument function (7.8) corresponding to \( a \mapsto (b \mapsto a + b) \) can also be thought of as the two-argument function \( (a, b) \mapsto a + b \).

In general, we will use the \( \lambda \) expression \( \lambda x. (\lambda y. f(x, y)) \) to simulate the effect of a two argument function \( (x, y) \mapsto f(x, y) \). This technique is known as Currying. We will use the shorthand \( \lambda x, y.e \) for \( \lambda x. (\lambda y. e) \).

If \( f = \lambda x. (\lambda y.e) \) then \( (fa)b \) corresponds to applying \( fa \) and then invoking the resulting function on \( b \), obtaining the result of replacing in \( e \) the occurrences of \( x \) with \( a \) and occurrences of \( b \) with \( y \). By our rules of associativity, this is the same as \( (fab) \) which we’ll sometimes also write as \( f(a, b) \).

### 7.4.3 Formal description of the \( \lambda \) calculus.

We now provide a formal description of the \( \lambda \) calculus. We start with “basic expressions” that contain a single variable such as \( x \) or \( y \) and build more complex expressions using the following two rules:

- **Application:** If \( e \) and \( e' \) are \( \lambda \) expressions, then the \( \lambda \) expression \( (e e') \) corresponds to applying the function described by \( e \) to the input \( e' \).

- **Abstraction:** If \( e \) is a \( y \) expression and \( x \) is a variable, then the \( \lambda \) expression \( \lambda x.(e) \) corresponds to the function that on any input \( z \) returns the expression \( e[x \rightarrow z] \) replacing all (free) occurrences of \( x \) in \( e \). \(^{6}\)

Formally \( \lambda \) expressions are defined as follows:

**Definition 7.7 — \( \lambda \) expression.** A \( \lambda \) expression is either a single variable identifier or an expression that is built from other expressions using the application and abstraction operations.

Definition 7.7 is a recursive definition since we defined the concept of \( \lambda \) expressions in terms of itself. Specifically, \( ¥1 ¥2 \) expression can either be the “base case” of the form \( foo \) for a variable identifier \( foo \),

\[ \lambda x. (\lambda y. f(x, y)) \]

![Currying Transformation Diagram](image-url)

Figure 7.7: In the “currying” transformation, we can create the effect of a two parameter function \( f(x, y) \) with the \( \lambda \) expression \( \lambda x. (\lambda y. f(x, y)) \) which on input \( x \) outputs a one-parameter function \( f_x \) that has \( x \) “hardwired” into it and such that \( f_x(y) = f(x, y) \). This can be illustrated by a circuit diagram; see Chelsea Voss’s site.

\(^{6}\) Strictly speaking we should replace only the free and not the ones that are bound by some other \( \lambda \) operator. For example, if we have the \( \lambda \) expression \( \lambda x. (\lambda x. x + 1)(x) \) and invoke it on the number 7 then we get \( (\lambda x. x + 1)(7) = 8 \) and not the nonsensical expression \( (\lambda 7.7 + 1)(7) \). To avoid such annoyances, we can adopt the convention that every instance of \( \lambda \text{var}.e \) uses a unique variable identifier \( \text{var} \). See Section 1.5.8 for more discussion on bound and free variables.
or it can be of the form \((ee')\) or \(\lambda \text{bar.}(e)\) where \(e\) and \(e'\) are other \(\lambda\) expressions and \(\text{bar}\) is a variable identifier. Such a recursive definition might seem confusing at first, but in fact you have known recursive definitions since you were an elementary school student. Consider how we define an arithmetic expression: it is an expression that is either just a number, or has one of the forms \((e + e')\), \((e - e')\), \((e \times e')\), or \((e \div e')\), where \(e\) and \(e'\) are other arithmetic expressions.

**Remark 7.8 — Precedence and parenthesis.** We will use the following rules to allow us to drop some parenthesis. Function application associates from left to right, and so \(fgh\) is the same as \((fg)h\). Function application has a higher precedence than the \(\lambda\) operator, and so \(\lambda x. fgx\) is the same as \(\lambda x.((fg)x)\).

This is similar to how we use the precedence rules in arithmetic operations to allow us to use fewer parenthesis and so write the expression \((7 \times 3) + 2\) as \(7 \times 3 + 2\).

As mentioned in ??, we also use the shorthand \(\lambda x. y.e\) for \(\lambda x. (\lambda y.e)\) and the shorthand \(f(x, y)\) for \((f x) y\).

This plays nicely with the “Currying” transformation of simulating multi-input functions using \(\lambda\) expressions.

As we have seen in Eq. (7.6), the rule that \((\lambda x.\exp)e\exp'\) is equivalent to \(\exp[x \rightarrow e\exp']\) enables us to modify \(\lambda\) expressions and obtain simpler equivalent form for them. Another rule that we can use is that the parameter does not matter and hence for example \(\lambda y. y\) is the same as \(\lambda z. z\). Together these rules define the notion of equivalence of \(\lambda\) expressions:

**Definition 7.9 — Equivalence of \(\lambda\) expressions.** Two \(\lambda\) expressions are equivalent if they can be made into the same expression by repeated applications of the following rules:

1. **Evaluation (aka \(\beta\) reduction):** The expression \((\lambda x.\exp)e\exp'\) is equivalent to \(\exp[x \rightarrow e\exp']\).
2. **Variable renaming (aka \(\alpha\) conversion):** The expression \(\lambda x.\exp\) is equivalent to \(\lambda y.\exp[x \rightarrow y]\).

If \(\exp\) is a \(\lambda\) expression of the form \(\lambda x.\exp'\) then it naturally corresponds to the function that maps any input \(z\) to \(\exp'[x \rightarrow z]\). Hence the \(\lambda\) calculus naturally implies a computational model. Since in the

---

7 These two rules are commonly known as “\(\beta\) reduction” and “\(\alpha\) conversion” in the literature on the \(\lambda\) calculus.
\( \lambda \) calculus the inputs can themselves be functions, we need to decide how to evaluate an expression such as

\[(\lambda x.f)(\lambda y.gz) . \quad (7.9)\]

There are two natural conventions for this:

- **Call by name**: We evaluate (7.9) by first plugging in the righthand expression \((\lambda y.gz)\) as input to the lefthand side function, obtaining \(f[x \to (\lambda y.gz)]\) and then continue from there.

- **Call by value**: We evaluate (7.9) by first evaluating the righthand side and obtaining \(h = g[y \to z]\), and then plugging this into the lefthandside to obtain \(f[x \to h]\).

Because the \( \lambda \) calculus has only pure functions, that do not have “side effects”, in many cases the order does not matter. In fact, it can be shown that if we obtain an definite irreducible expression (for example, a number) in both strategies, then it will be the same one. However, there could be situations where “call by value” goes into an infinite loop while “call by name” does not. Hence we will use “call by name” henceforth.

**7.4.4 Functions as first class objects**

The key property of the \( \lambda \) calculus (and functional languages in general) is that functions are “first-class citizens” in the sense that they can be used as parameters and return values of other functions. Thus, we can invoke one \( \lambda \) expression on another. For example if \( \text{DOUBLE} \) is the \( \lambda \) expression \( \lambda f.(\lambda x.f(fx))\), then for every function \( f \), \( \text{DOUBLE } f \) corresponds to the function that invokes \( f \) twice on \( x \) (i.e., first computes \( fx \) and then invokes \( f \) on the result). In particular, if \( f = \lambda y.(y + 1) \) then \( \text{DOUBLE } f = \lambda x.(x + 2) \).

**Remark 7.10 — (Lack of) types.** Unlike most programming languages, the pure \( \lambda \)-calculus doesn’t have the notion of types. Every object in the \( \lambda \) calculus can also be thought of as a \( \lambda \) expression and hence as a function that takes one input and returns one output. All functions take one input and return one output, and if you feed a function an input of a form it didn’t expect, it still evaluates the \( \lambda \) expression via “search and replace”, replacing all instances of its parameter with copies of the input expression you fed it.
7.5 THE “ENHANCED” λ CALCULUS

We now discuss the λ calculus as a computational model. We will start by describing an “enhanced” version of the λ calculus that contains some “superfluous features” but is easier to wrap your head around. We will first show how the enhanced λ calculus is equivalent to Turing machines in computational power. Then we will show how all the features of “enhanced λ calculus” can be implemented as “syntactic sugar” and even the “pure” (i.e., non enhanced) λ calculus is equivalent in power to Turing machines (and hence also to RAM machines and all other Turing-equivalent models).

The enhanced λ calculus includes the following set of objects and operations:

• **Boolean constants and IF function:** The enhanced λ calculus has the constants 0 and 1 and the IF function such that for every cond ∈ \{0, 1\} and λ expressions a, b, IF\(cond\) \(a\ b\) outputs \(a\) if \(cond = 1\) and outputs \(b\) if \(cond = 0\). Using IF and the constants 0, 1 we can also compute logical operations such as AND, OR, NOT, NAND. For example, NOT = \(λa.\ IFa01\) and AND = \(λa, b.\ IFab0\).

• **Pairs:** We have the function PAIR such that PAIR \(a\ b\) returns the pair \((a, b)\) that holds \(a\) and \(b\). We also have the functions HEAD and TAIL to extract the first and second member of a pair respectively. Hence, HEAD(PAIR\(ab\)) = \(a\) and TAIL(PAIR\(ab\)) = \(b\).

• **Lists and strings:** Using PAIR we can also construct lists. The idea is that PAIR \(a\ L\) corresponds to the list obtained by adding the element \(a\) to the beginning of a list \(L\). By repeating this operation, we can construct lists of any length. Specifically, we will have a special λ expression NIL that corresponds to the empty list, which we also denote by \(\langle\rangle\). If \(c\) is some λ expression, then PAIR \(c\ NIL\) corresponds to the single-element list \(\langle c\rangle\). Now for every λ expressions \(b, c\), the expression PAIR \(b\ (PAIR\ NIL\ c)\) corresponds to the two-element list \(\langle b, c\rangle\). Similarly the expression PAIR \(a(PAIR\ b\ (PAIR\ c\ NIL))\) corresponds to the list \(\langle a, b, c\rangle\) and so on and so forth. The function ISEMPYRTY returns 1 on NIL and returns 0 on every other list. A string is simply a list of bits.

• **List operations:** The enhanced λ calculus also contains the list-processing functions MAP, REDUCE, and FILTER. Given a list \(L = \langle x_0, ..., x_{n-1}\rangle\) and a function \(f\), MAP \(L\ f\) applies \(f\) on every member of the list to obtain the new list \(L’ = \langle f(x_0), ..., f(x_{n-1})\rangle\). Given a list \(L\) as above and a function \(f\) whose output is either 0 or 1, FILTER \(L\ f\) returns the list \(\langle x_i\rangle\ f(x_i) = 1\) containing all the elements of \(L\) for which \(f\) outputs 1. The function REDUCE applies a “combining” operation to a list. For example, REDUCE \(L\ + 0\) will return...
the sum of all the elements in the list \( L \). The sum of a list is defined recursively as follows: the sum of the empty list is 0, and the sum of a non-empty list \( L \) is obtained by recursively summing \( \text{TAIL} L \) (i.e., all elements of \( L \) except the first) and adding the result to \( \text{HEAD} L \) (which is the first element of \( L \)). More generally, \( \text{REDUCE} \) takes a list \( L \), an operation \( f \) (which we think of as taking two arguments) and a \( \lambda \) expression \( z \) (which we think of as the “neutral element” for the operation \( f \), such as 0 for addition and 1 for multiplication). The output is defined via

\[
\text{REDUCE} \; L \; f \; z = \begin{cases} 
    z & \text{if } L = \text{NIL} \\
    f(\text{HEAD} L) \; (\text{REDUCE} \; (\text{TAIL} L) \; f \; z) & \text{otherwise}
\end{cases}
\]

(7.10)

See Fig. 7.9 for an illustration of the three list-processing operations.

- **Recursion:** Finally, we want to be able to execute recursive functions. Since in \( \lambda \) calculus functions are anonymous, we can’t write a definition of the form \( f(x) = \text{blah} \) where \( \text{blah} \) includes calls to \( f \). Instead we use functions \( f \) that take an additional input \( me \) as a parameter. The operator \( \text{RECURSE} \) will take such a function \( f \) as input and return a “recursive version” of \( f \) where all the calls to \( me \) are replaced by recursive calls to this function. That is, if we have a function \( F \) taking two parameters \( me \) and \( x \), then \( \text{RECURSE} \; F \) will be the function \( f \) taking one parameter \( x \) such that \( f(x) = F(f, x) \) for every \( x \).

**Solved Exercise 7.1** — **Compute NAND using \( \lambda \) calculus.** Give a \( \lambda \) expression \( N \) such that \( N \; x \; y = \text{NAND}(x, y) \) for every \( x, y \in \{0, 1\} \).

**Solution:**

This can be done in a similar way to how we computed XOR.

The \( \text{NAND} \) of \( x, y \) is equal to 1 unless \( x = y = 1 \). Hence we can write

\[
N = \lambda x, y. \text{IF}(x, \text{IF}(y, 0, 1), 1)
\]

(7.11)

**Example 7.11** — **Computing XOR.** Let us see how we can compute the XOR of a list in the enhanced \( \lambda \) calculus. First, we note that we can compute XOR of two bits as follows:

\[
\text{NOT} = \lambda a. \text{IF}(a, 0, 1)
\]

(7.12)
and
\[ \text{XOR}_2 = \lambda a, b. \text{IF}(b, \text{NOT}(a), a) \] (7.13)

(We are using here a bit of syntactic sugar to describe the functions. To obtain the \( \lambda \) expression for XOR we will simply replace the expression (7.12) in (7.13).)

Now recursively we can define the XOR of a list as follows:

\[
\text{XOR}(L) = \begin{cases} 
0 & \text{if } L \text{ is empty} \\
\text{XOR}_2(\text{HEAD}(L), \text{XOR}(\text{TAIL}(L))) & \text{otherwise}
\end{cases}
\] (7.14)

This means that XOR is equal to

\[
\text{RECURSE} \left( \lambda me, L. \text{IF}(\text{ISEMPTY}(L), 0, \text{XOR}_2(\text{HEAD} L, me(\text{TAIL} L))) \right).
\] (7.15)

That is, XOR is obtained by applying the \text{RECURSE} operator to the function that on inputs \( me, L \), returns 0 if \( \text{ISEMPTY}(L) \) and otherwise returns \( \text{XOR}_2 \) applied to \( \text{HEAD}(L) \) and to \( me(\text{TAIL}(L)) \).

We could have also computed XOR using the \text{REDUCE} operation, we leave working this out as an exercise to the reader.

Figure 7.8: A list \( \langle x_0, x_1, x_2 \rangle \) in the \( \lambda \) calculus is constructed from the tail up, building the pair \( \langle x_2, \text{NIL} \rangle \), then the pair \( \langle x_1, \langle x_2, \text{NIL} \rangle \rangle \) and finally the pair \( \langle x_0, \langle x_1, \langle x_2, \text{NIL} \rangle \rangle \rangle \). That is, a list is a pair where the first element of the pair is the first element of the list and the second element is the rest of the list. The figure on the left renders this “pairs inside pairs” construction, though it is often easier to think of a list as a “chain”, as in the figure on the right, where the second element of each pair is thought of as a \textit{link}, \textit{pointer} or \textit{reference} to the remainder of the list.

### 7.5.1 Enhanced \( \lambda \) expressions

An enhanced \( \lambda \) expression is obtained by composing the objects above with the \textit{application} and \textit{abstraction} rules. We can now define the notion of computing a function using the \( \lambda \) calculus. We will define the \textit{simplification} of a \( \lambda \) expression as the following recursive process:

1. \textit{(Evaluation / \( \beta \) reduction.)} If the expression has the form \( (\text{exp}_L, \text{exp}_R) \) then replace the expression with \( \text{exp}_L'[x \rightarrow \text{exp}_R] \).

2. \textit{(Renaming / \( \alpha \) conversion.)} When we cannot simplify any further, rename the variables so that the first bound variable in the expression is \( v_0 \), the second one is \( v_1 \), and so on and so forth.
The result of simplifying a \( \lambda \) expression is an equivalent expression, and hence if two expressions have the same simplification then they are equivalent.

**Definition 7.12** — Computing a function via \( \lambda \) calculus. Let \( F : \{0, 1\}^* \to \{0, 1\}^* \) be a function and \( \text{exp} \) a \( \lambda \) expression. For every \( x \in \{0, 1\}^n \), we denote by \( \text{LIST}(x) \) the list \( \text{PAIR}(x_0, \text{PAIR}(x_1, \text{PAIR}(\cdots \text{PAIR}(x_{n-1}, \text{NIL}))))) \) that corresponds to \( x \).

We say that \( \text{exp} \) computes \( F \) if for every \( x \in \{0, 1\}^* \), the expressions \( \text{expLIST}(x) \) and \( \text{LIST}(F(x)) \) are equivalent, and moreover they have the same simplification.

**7.5.2 Enhanced \( \lambda \) calculus is Turing-complete**

The basic operations of the enhanced \( \lambda \) calculus more or less amount to the Lisp or Scheme programming languages. It is perhaps not surprising that the enhanced \( \lambda \)-calculus is equivalent to Turing machines:

**Theorem 7.13** — Lambda calculus and NAND-TM. For every function \( F : \{0, 1\}^* \to \{0, 1\}^* \), \( F \) is computable in the enhanced \( \lambda \) calculus if and only if it is computable by a Turing machine.

**Proof Idea:**

To prove the theorem, we need to show that (1) if \( F \) is computable by a \( \lambda \) calculus expression then it is computable by a Turing machine,
and (2) if $F$ is computable by a Turing machine, then it is computable by an enhanced $\lambda$ calculus expression.

Showing (1) is fairly straightforward. Applying the simplification rules to a $\lambda$ expression basically amounts to “search and replace” which we can implement easily in, say, NAND-RAM, or for that matter Python (both of which are equivalent to Turing machines in power). Showing (2) essentially amounts to simulating a Turing machine (or writing a NAND-TM interpreter) in a functional programming language such as LISP or Scheme. Showing how this can be done is a good exercise in mastering some functional programming techniques that are useful in their own right.

$\star$

Proof of Theorem 7.13. We only sketch the proof. The “if” direction is simple. As mentioned above, evaluating $\lambda$ expressions basically amounts to “search and replace”. It is also a fairly straightforward programming exercise to implement all the above basic operations in an imperative language such as Python or C, and using the same ideas we can do so in NAND-RAM as well, which we can then transform to a NAND-TM program.

For the “only if” direction, we need to simulate a Turing machine, or equivalently a NAND-TM program, using a $\lambda$ expression. First, by Solved Exercise 7.1 we can compute the NAND function, and hence every finite function, using the $\lambda$ calculus. Thus proving the theory boils down to simulating the arrays of NAND-TM using the lists of the enhanced $\lambda$ calculus.

We will encode each array $A$ of NAND-TM program by a list $L$ of the NAND-CIRC program. We encode the index variable $i$ by a special list $I$ that has 1 in the location corresponding to the value of $i$ and 0’s everywhere else. To simulate moving $i$ to the left, we need to remove the first item from the list, while to simulate moving $i$ to the right, we add a zero to the head of list.\footnote{In fact, it will be convenient for us to make sure all lists are of the same length, and so at the end of each step we will add a sufficient number of zeroes to the end of each list. This can be done with a simple REDUCE operation.}

To extract the $i$-th bit of the array corresponding to $L$, we need to compute the following function $\text{get}$ that on input a pair of lists $I$ and $L$ of bits of the same length $n$, $\text{get}(I, L)$ outputs 1 if and only if there is some $j \in [n]$ such that the $j$-th element of both $I$ and $L$ is equal to 1. This turns out to be not so hard. The key is to implement the function $\text{zip}$ that on input a pair of lists $I$ and $L$ of the same length $n$, outputs a list of $n$ pairs $M$ such that the $j$-th element of $M$ (which we denote by $M_j$) is the pair $(I_j, L_j)$. Thus $\text{zip}$ “zips together” these two lists of elements into a single list of pairs.\footnote{The name $\text{zip}$ is a common name for this operation, for example in Python. It should not be be confused with the zip compression file format.} It is a good exercise to give a recursive implementation of $\text{zip}$, and so can implement it using the RECURSE operator. Once we have $\text{zip}$, we can implement $\text{get}$ by applying an appropriate REDUCE on the list $\text{zip}(I, L)$.
Setting the list \( L \) at the \( i \)-th location to a certain value requires computing the function \( \text{set}(I, L, v) \) that outputs a list \( L' \) such that \( L'_j = L_j \) if \( I_j = 0 \) and \( L'_j = v \) otherwise. The function \( \text{set} \) can be implemented by applying \( \text{MAP} \) with an appropriate operator to the list \( \text{zip}(I, L) \).

We omit the full details of implementing \( \text{set} \) and \( \text{get} \), but the bottom line is that for every NAND-TM program \( P \), we can obtain a \( \lambda \) expression \( \text{NEXT}_P \) such that, if we let \( \sigma = (\text{loop, foo, bar,..., I, X, X_{onblank}, Y, Y_{onblank}, Baz, Blah,...}) \) be the set of Boolean values and lists that encode the current state of \( P \) (with a list for each array and for the index variable \( i \)), then \( \text{NEXT}_P \sigma \) will encode the state after performing one iteration of \( P \).

Now we can use the following “pseudocode” to simulate the program \( P \). The function \( \text{SIM}_P \) will obtain an encoding \( \sigma_0 \) of the initial state of \( P \), and output the encoding \( \sigma^* \) of the state of \( P \) after it halts. It will be computed as follows:

Algorithm 7.14 — \( \text{SIM}_P \): Simulate NAND-TM program in \( \lambda \) calculus.

**Input:** \( \sigma \): encoding of the state of a NAND-TM program

**Operation:**

1. Let \( \sigma' = \text{NEXT}_P \sigma \).
2. If \( \text{halt}(\sigma') = 1 \) then return \( \sigma' \), where \( \text{halt}(\sigma') \) is equal to 1 if the configuration \( \sigma' \) is one in which the program will halt when executing the \text{MODANDJUMP} operation.
3. Otherwise return \( \text{SIM}_P(\sigma') \).

We can write this algorithm as the \( \lambda \) expression

\[
\text{RECURSE} \left( \lambda m, \sigma. \text{IF}(\text{loop}(\text{NEXT}_P \sigma), m(\text{NEXT}_P \sigma), \text{NEXT}_P \sigma) \right)
\]

(7.16)

Given \( \text{SIM}_P \), we can compute the function computed by \( P \) by writing expressions for encoding the input as the initial state, and decoding the output from the final state. We omit the details, though this is fairly straightforward.\(^{14}\)

### 7.6 THE PURE \( \lambda \) CALCULUS

While the collection of “basic” functions we allowed for \( \lambda \) calculus is smaller than what’s provided by most Lisp dialects, coming from NAND-TM it still seems a little “bloated”. Can we make do with less? In other words, can we find a subset of these basic operations that can implement the rest?

\(^{14}\) For example, if \( X \) is a list representing the input, then we can obtain a list \( X_{\text{nonblank}} \) of 1’s of the same length by simply writing \( X_{\text{nonblank}} = \text{MAP}(X, \lambda x. 1) \).
This is a good point to pause and think how you would implement these operations yourself. For example, start by thinking how you could implement \( \text{MAP} \) using \( \text{REDUCE} \), and then \( \text{REDUCE} \) using \( \text{RECURSE} \) combined with \( 0, 1, \text{IF}, \text{PAIR}, \text{HEAD}, \text{TAIL}, \text{NIL}, \text{ISEMPTY} \) together with the \( \lambda \) operations.

Now you can think how you could implement \( \text{PAIR}, \text{HEAD} \) and \( \text{TAIL} \) based on \( 0, 1, \text{IF} \). The idea is that we can represent a pair as function.

It turns out that there is in fact a proper subset of these basic operations that can be used to implement the rest. That subset is the empty set. That is, we can implement all the operations above using the \( \lambda \) formalism only, even without using 0’s and 1’s. It’s \( \lambda \)'s all the way down! The idea is that we encode 0 and 1 themselves as \( \lambda \) expressions, and build things up from there. This is known as Church encoding, as it was originated by Church in his effort to show that the \( \lambda \) calculus can be a basis for all computation.

**Theorem 7.15** — Enhanced \( \lambda \) calculus equivalent to pure \( \lambda \) calculus. There are \( \lambda \) expressions that implement the functions \( 0, 1, \text{IF}, \text{PAIR}, \text{HEAD}, \text{TAIL}, \text{NIL}, \text{ISEMPTY}, \text{MAP}, \text{REDUCE}, \) and \( \text{RECURSE} \).

We will not write the full formal proof of Theorem 7.15 but outline the ideas involved in it:

- We define 0 to be the function that on two inputs \( x, y \) outputs \( y \), and 1 to be the function that on two inputs \( x, y \) outputs \( x \). Of course we use Currying to achieve the effect of two-input functions and hence \( 0 = \lambda x. \lambda y. y \) and \( 1 = \lambda x. \lambda y. x \).\(^{15}\)

- The above implementation makes the IF function trivial: \( \text{IF}(\text{cond}, a, b) \) is simply \( \text{cond} \ a \ b \) since \( 0 a b = b \) and \( 1 a b = a \). We can write \( \text{IF} = \lambda x.x \) to achieve \( \text{IF}(\text{cond}, a, b) = (((\text{IF}\text{cond})a)b) = \text{cond} \ a \ b \).

- To encode a pair \( (x, y) \) we will produce a function \( f_{x,y} \) that has \( x \) and \( y \) “in its belly” and satisfies \( f_{x,y} g = gxy \) for every function \( g \). That is, we write \( \text{PAIR} = \lambda x. \lambda y. \lambda g. gxy \). Note that now we can extract the first element of a pair \( p \) by writing \( p1 \) and the second element by writing \( p0 \), and so \( \text{HEAD} = \lambda p. p1 \) and \( \text{TAIL} = \lambda p. p0 \).

- We define \( \text{NIL} \) to be the function that ignores its input and always outputs 1. That is, \( \text{NIL} = \lambda x. 1 \). The ISEMP erty function checks, given an input \( p \), whether we get 1 if we apply \( p \) to the function.
Remark 7.16 — Church numerals (optional). There is nothing special about Boolean values. You can use similar tricks to implement natural numbers using λ terms. The standard way to do so is to represent the number \( n \) by the function \( \text{ITER}_n \) that on input a function \( f \) outputs the function \( x \mapsto f(f(...f(x))) \) (\( n \) times). That is, we represent the natural number 1 as \( \lambda f.f \), the number 2 as \( \lambda f.(\lambda x.f(fx)) \), the number 3 as \( \lambda f.(\lambda x.f(f(fx))) \), and so on and so forth. (Note that this is not the same representation we used for 1 in the Boolean context: this is fine; we already know that the same object can be represented in more than one way.) The number 0 is represented by the function that maps any function \( f \) to the identity function \( \lambda x.x \). (That is, \( 0 = \lambda f.(\lambda x.x) \).)

In this representation, we can compute \( \text{PLUS}(n, m) \) as \( \lambda f.\lambda x.(n f)((m f)x) \) and \( \text{TIMES}(n, m) \) as \( \lambda f.n(m f) \). Subtraction and division are trickier, but can be achieved using recursion. (Working this out is a great exercise.)

7.6.1 List processing

Now we come to a bigger hurdle, which is how to implement \( \text{MAP} \), \( \text{FILTER} \), and \( \text{REDUCE} \) in the λ calculus. It turns out that we can build \( \text{MAP} \) and \( \text{FILTER} \) from \( \text{REDUCE} \). For example \( \text{MAP}(L, f) \) is the same as \( \text{REDUCE}(L, g) \) where \( g \) is the operation that on input \( x \) and \( y \), outputs \( \text{PAIR}(f(x), y) \) if \( y \) is NIL and otherwise outputs \( \text{PAIR}(f(x), \text{NIL}) \). (I leave checking this as a (recommended!) exercise for you, the reader.) So, it all boils down to implementing \( \text{REDUCE} \). We can define \( \text{REDUCE}(L, g) \) recursively, by setting \( \text{REDUCE}(\text{NIL}, g) = \text{NIL} \) and stipulating that given a non-empty list \( L \), which we can think of as a pair \( (\text{head}, \text{rest}) \), \( \text{REDUCE}(L, g) = g(\text{head}, \text{REDUCE}(\text{rest}, g)) \). Thus, we might try to write a λ expression for \( \text{REDUCE} \) as follows

\[
\text{REDUCE} = \lambda L, g.\text{IF}(\text{ISEMPTY}(L), \text{NIL}, g\text{HEAD}(L)\text{REDUCE}(\text{TAIL}(L), g)) .
\]

(7.17)

The only fly in this ointment is that the λ calculus does not have the notion of recursion, and so this is an invalid definition. But of course we can use our \text{RECURSE} operator to solve this problem. We will replace the recursive call to “\text{REDUCE}” with a call to a function \( \text{me} \)
that is given as an extra argument, and then apply RECURSE to this. Thus \( \text{REDUCE} = \text{RECURSE} \, \text{myREDUCE} \) where

\[
\text{myREDUCE} = \lambda me, L, g. \text{IF}(\text{ISEMPTY}(L), \text{NIL}, g \text{HEAD}(L) me(\text{TAIL}(L), g)) .
\]  

(7.18)

So everything boils down to implementing the RECURSE operator, which we now deal with.

### 7.6.2 The Y combinator, or recursion without recursion

How can we implement recursion without recursion? We will illustrate this using a simple example - the XOR function. As shown in Example 7.11, we can write the XOR function of a list recursively as follows:

\[
\text{XOR}(L) = \begin{cases} 0 & \text{if } L \text{ is empty} \\ \text{XOR}(\text{HEAD}(L), \text{XOR}(\text{TAIL}(L))) & \text{otherwise} \end{cases}
\]

(7.19)

where \( \text{XOR}_2 : \{0, 1\}^2 \to \{0, 1\} \) is the XOR on two bits. In Python we would write this as

```python
def xor2(a,b): return 1-b if a else b
def head(L): return L[0]
def tail(L): return L[1:]

def xor(L): return xor2(head(L),xor(tail(L))) if L else 0
```

```python
print(xor([0,1,0,1,0,1]))
# 1
```

Now, how could we eliminate this recursive call? The main idea is that since functions can take other functions as input, it is perfectly legal in Python (and the \( \lambda \) calculus of course) to give a function itself as input. So, our idea is to try to come up with a non recursive function \( \text{tempxor} \) that takes two inputs: a function and a list, and such that \( \text{tempxor}(\text{tempxor}, L) \) will output the XOR of \( L \)!

At this point you might want to stop and try to implement this on your own in Python or any other programming language of your choice (as long as it allows functions as inputs).

Our first attempt might be to simply use the idea of replacing the recursive call by \( me \). Let’s define this function as \( \text{myxor} \)
def myxor(me,L):
    return xor2(head(L),me(tail(L))) if L
    else 0

Let’s test this out:

myxor(myxor,[1,0,1])

If you do this, you will get the following complaint from the interpreter: TypeError: myxor() missing 1 required positional argument. The problem is that myxor expects two inputs - a function and a list- while in the call to me we only provided a list. To correct this, we modify the call to also provide the function itself:

def tempxor(me,L):
    return xor2(head(L),me(me,tail(L))) if L
    else 0

Note the call me(me,..) in the definition of tempxor: given a function me as input, tempxor will actually call the function me with itself as the first input. If we test this out now, we see that we actually get the right result!

tempxor(tempxor,[1,0,1])
# 0
tempxor(tempxor,[1,0,1,1])
# 1

and so we can define xor(L) as simply return tempxor(tempxor,L).

The approach above is not specific to XOR. Given a recursive function f that takes an input x, we can obtain a non recursive version as follows:

1. Create the function myf that takes a pair of inputs me and x, and replaces recursive calls to f with calls to me.
2. Create the function tempf that converts calls in myf of the form me(x) to calls of the form me(me,x).
3. The function f(x) will be defined as tempf(tempf,x)

Here is the way we implement the RECURSE operator in Python. It will take a function myf as above, and replace it with a function g such that g(x)=myf(g,x) for every x.

def RECURSE(myf):
    def tempf(me,x):
        return myf(lambda y: me(me,y),x)

    return lambda x: tempf(tempf,x)
xor = RECURSE(myxor)

print(xor([0,1,1,0,0,1]))
# 1

print(xor([1,1,0,0,1,1,1]))
# 0

From Python to the λ calculus. In the λ calculus, a two input function \( g \) that takes a pair of inputs \( me, y \) is written as \( \lambda me. (\lambda y.g) \). So the function \( y \mapsto me(me, y) \) is simply written as \( me \cdot me \) and similarly the function \( x \mapsto tempf(tempf, x) \) is simply \( tempf \cdot tempf \). (Can you see why?) Therefore in the λ calculus, the function \( tempf \) is \( \lambda me. f(me me) \) and similarly for \( \lambda x: tempf(tempf, x) \) is the same as \( tempf \cdot tempf \). Therefore, if we denote the input of \( \text{RECURSE} \) by \( f \), then the output is equal to \( FF \) where \( F = \lambda m.f(mm) \), which means that

\[
\text{RECURSE} = \lambda f.((\lambda m.f(mm))(\lambda m.f(mm))) \tag{7.20}
\]

The online appendix contains an implementation of the λ calculus using Python. Here is an implementation of the recursive XOR function from that appendix:16

# XOR of two bits
XOR2 = \( \lambda (a,b)(\text{IF}(a,\text{IF}(b,\_0,\_1),b)) \)

# Recursive XOR with recursive calls replaced by m
→ parameter
myXOR = \( \lambda (m,l)(\text{IF}(\text{ISEMPY}(l),\_0,\text{XOR2}(\text{HEAD}(l),m(\text{TAIL}(l)))))) \)

# Recurse operator (aka Y combinator)
RECURSE = \( \lambda f((\lambda m.f(mm))(\lambda m.f(mm))) \)

# XOR function
XOR = RECURSE(myXOR)

#TESTING:
XOR(PAIR(_1,NIL)) # List [1]
# equals 1
XOR(PAIR(_1,PAIR(_0,PAIR(_1,NIL)))) # List [1,0,1]
# equals 0

16 Because of specific issues of Python syntax, in this implementation we use \( f \cdot g \) for applying \( f \) to \( g \) rather than \( fg \), and use \( \lambda x.\exp \) rather than \( \lambda x.\exp \) for abstraction. We also use \( \_0 \) and \( \_1 \) for the \( \lambda \) terms for \( 0 \) and \( 1 \) so as not to confuse with the Python constants.
This assumes we use the “call by value” evaluation ordering which states that to evaluate a λ expression \( \lambda x. x x x \) we first evaluate the righthand expression \( x x x \) and then invoke \( \lambda x. x x x \) on it. The “Call by name” or “lazy evaluation” ordering would first evaluate the lefthand expression \( \lambda x. x x x \) and then invoke it on \( x x x \). In this case both strategies would result in an infinite loop. There are examples though when “call by name” would not enter an infinite loop while “call by value” would. The SML and OCaml programming languages use “call by value” while Haskell uses (a close variant of) “call by name”.

7.6.3 Infinite loops in the \( \lambda \) calculus
The fact that \( \lambda \)-expressions can simulate NAND-TM programs means that, like Turing machines and NAND-TM programs, the \( \lambda \) calculus can also enter into an infinite loop. For example, consider the \( \lambda \) expression

\[
(\lambda x. x x x)(\lambda x. x x x)
\]

If we try to evaluate it then the first step is to invoke the lefthand function on the righthand one and then obtain

\[
(\lambda x. x x x)(\lambda x. x x x)(\lambda x. x x x)
\]

To evaluate this, the next step would be to apply the second term on the third term,\(^{17}\) which would result in

\[
(\lambda x. x x x)(\lambda x. x x x)(\lambda x. x x x)(\lambda x. x x x)
\]

We can see that continuing in this way we get longer and longer expressions, and this process never concludes.

7.7 MORE TURING-COMPLETE COMPUTATIONAL MODELS
There is a great variety of models that are computationally equivalent to Turing machines (and hence to NAND-TM/NAND-RAM program). We briefly mention a few examples.

7.7.1 Parallel algorithms and cloud computing
The models of computation we considered so far are inherently sequential, but these days much computation happens in parallel, whether using multi-core processors or in massively parallel distributed computation in data centers or over the Internet. Parallel computing is important in practice, but it does not really make much difference for the question of what can and can’t be computed. After all, if a computation can be performed using \( m \) machines in \( t \) time, then it can be computed by a single machine in time \( mt \).

\(^{17}\) This assumes we use the “call by value” evaluation ordering which states that to evaluate a \( \lambda \) expression \( f \) \( g \) we first evaluate the righthand expression \( g \) and then invoke \( f \) on it. The “Call by name” or “lazy evaluation” ordering would first evaluate the lefthand expression \( f \) and then invoke it on \( g \). In this case both strategies would result in an infinite loop. There are examples though when “call by name” would not enter an infinite loop while “call by value” would. The SML and OCaml programming languages use “call by value” while Haskell uses (a close variant of) “call by name”.

Remark 7.17 — The Y combinator. The RECURSE operator above is better known as the Y combinator. It is one of a family of a fixed point operators that given a lambda expression \( F \), find a fixed point \( f \) of \( F \) such that \( f = F f \). If you think about it, XOR is the fixed point of \( myXOR \) above. XOR is the function such that for every \( x \), if plug in XOR as the first argument of \( myXOR \) then we get back XOR, or in other words \( XOR = myXOR XOR \). Hence finding a fixed point for \( myXOR \) is the same as applying RECURSE to it.
7.7.2 Game of life, tiling and cellular automata

Many physical systems can be described as consisting of a large number of elementary components that interact with one another. One way to model such systems is using cellular automata. This is a system that consists of a large number (or even infinite) cells. Each cell only has a constant number of possible states. At each time step, a cell updates to a new state by applying some simple rule to the state of itself and its neighbors.

A canonical example of a cellular automaton is Conway’s Game of Life. In this automata the cells are arranged in an infinite two dimensional grid. Each cell has only two states: “dead” (which we can encode as 0 and identify with ∅) or “alive” (which we can encode as 1). The next state of a cell depends on its previous state and the states of its 8 vertical, horizontal and diagonal neighbors. A dead cell becomes alive only if exactly three of its neighbors are alive. A live cell continues to live if it has two or three live neighbors. Even though the number of cells is potentially infinite, we can have a finite encoding for the state by only keeping track of the live cells. If we initialize the system in a configuration with a finite number of live cells, then the number of live cells will stay finite in all future steps.

We can think of such a system as encoding a computation by starting it in some initial configuration, and then defining some halting condition (e.g., we halt if the cell at position (0, 0) becomes dead) and some way to define an output (e.g., we output the state of the cell at position (1, 1)). Clearly, given any starting configuration \( x \), we can simulate the game of life starting from \( x \) using a NAND-RAM (or NAND-TM) program, and hence every “Game-of-Life computable” function is computable by a NAND-RAM program. Surprisingly, it turns out that the other direction is true as well: as simple as its rules seem, we can simulate a Turing machine using the game of life (see Fig. 7.10). The Wikipedia page for the Game of Life contains some beautiful figures and animations of configurations that produce some very interesting evolutions. See also the book The Nature of Computation.

7.7.3 Configurations of Turing machines and one dimensional cellular automata

It turns out that even one dimensional cellular automata can be Turing complete (see Fig. 7.12). In a one dimensional automata, the cells are laid out in one infinitely long line. The next state of each cell is only a function of its past state and the state of both its neighbors.

Figure 7.10: A Game-of-Life configuration simulating a Turing Machine. Figure by Paul Rendell.
**Definition 7.18** — One dimensional cellular automata. Let $\Sigma$ be a finite set containing the symbol $\emptyset$. A one dimensional cellular automaton over alphabet $\Sigma$ is described by a transition rule $r : \Sigma^3 \to \Sigma$, which satisfies $r(\emptyset, \emptyset, \emptyset) = \emptyset$.

An configuration of the automaton is specified by a string $\alpha \in \Sigma^*$. We can also think of $\alpha$ as the infinite sequence $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}, \emptyset, \emptyset, \emptyset, \ldots)$, where $n = |\alpha|$. If $\alpha$ is a configuration and $r$ is a transition rule, then the next step configuration, denoted by $\alpha' = \text{NEXT}_r(\alpha)$, is defined as follows:

$$\alpha'_i = \text{NEXT}_r(\alpha_{i-1}, \alpha_i, \alpha_{i+1})$$

(7.24)

for $i = 0, \ldots, n$. If $j$ is smaller than 0 or larger than $n - 1$ then we set $\alpha_j = \emptyset$.

In other words, the next state of the automaton $r$ at point $i$ obtained by applying the rule $r$ to the values of $\alpha$ at $i$ and its two neighbors.

**Theorem 7.19** — One dimensional automata are Turing complete. For every Turing machine $M$, there is a one dimension cellular automaton that can simulate $M$ on every input $x$.

To make this more precise, what Theorem 7.19 says is that for every Turing machine $M$, there is a one-dimensional cellular automaton $A$ over some alphabet $\Sigma$, such that:

- There are computable maps to encode and decode configurations of $M$ as configurations of $A$.
- If we initialize $A$ in the configuration encoding the starting state of $M$, then at every step $t$, the configuration of $A$ at step $t$ encodes the configuration of $M$ at the same step.

**Proof Idea:**

A configuration of $M$ contains its full state at after a particular iteration. That is, the contents of all (non empty) cells of its tape, its current state, as well as the head position. We can encode such a configuration as a string $\alpha$ over some large alphabet $\Sigma$. At position $j$, the symbol $\alpha_j$ will encode the value of the $j$-th symbol in $M$’s tape. If the head position is $i$ then $\alpha_i$ will encode this fact as well, and also contain an encoding of which state the machine is at.

Given this notion of an encoding, and the fact that the head moves only one position in each step, we can see that after one step of the machine $M$, the configuration largely stays the same except the locations $i, i-1, i+1$ corresponding to the location of the current variable $i$ and its immediate neighbors. Once we realize this, we can phrase the
progression from one configuration to the next as a one dimensional cellular automaton! From this observation, Theorem 7.19 follows in a fairly straightforward manner.

Before proving Theorem 7.19, let us formally define the notion of a configuration of a Turing machine (see also Fig. 7.11). We will come back to this notion in later chapters as well.

A configuration of a Turing machine $M$ with alphabet $\Sigma$ and state space $[k]$ encodes the state of $M$ at a particular step in its execution as a string $\alpha$ over the alphabet $\Sigma = \Sigma \times (\{\cdot\} \cup [k])$. The string is of length $t$ where $t$ is such that $M$’s tape contains $\emptyset$ in all positions $t$ and larger and $M$’s head is in a position smaller than $t$. If $M$’s head is in the $i$-th position, then for $j \neq i$, $\alpha_j$ encodes the value of the $j$-th cell of $M$’s tape, while $\alpha_i$ encodes both this value as well as the current state of $M$. If the machine writes the value $\tau$, changes state to $t$, and moves right, then in the next configuration will contain at position $i$ the value $\langle \tau, \cdot \rangle$ and at position $i + 1$ the value $\langle \alpha_{i+1}, t \rangle$.

Definition 7.20 below has some technical details, but is not actually that deep or complicated. You would probably understand it better if before starting to read it, you take a moment to stop and think how you would encode as a string the state of a Turing machine at a given point in an execution.

Think what are all the components that you need to know in order to be able to continue the execution from this point onwards, and what is a simple way to encode them using a list of strings (which in turn can be encoded as a string). In particular, with an eye towards our future applications, try to think of an encoding which will make it as simple as possible to map a configuration at step $t$ to the configuration at step $t + 1$.

**Definition 7.20** — Configuration of NAND-TM programs. Let $M$ be a Turing machine with tape alphabet $\Sigma$ and state space $[k]$. A configuration of $M$ is a string $\alpha \in \Sigma^*$ where $\Sigma = \Sigma \times (\{\cdot\} \cup [k])$ that satisfies that there is exactly one coordinate $i$ for which $\alpha_i = (\sigma, s)$ for some $\sigma \in \Sigma$ and $s \in [k]$. For all other coordinates $j$, $\alpha_j = (\sigma', \cdot)$ for some $\sigma' \in \Sigma$.
A configuration $\alpha \in \Sigma^*$ of $M$ corresponds to the following state of its execution:

- $M$’s tape contains $\alpha_{j,0}$ for all $j < |\alpha|$ and contains $\emptyset$ for all positions that are at least $|\alpha|$, where we let $\alpha_{j,0}$ be the value $\sigma$ such that $\alpha_j = (\sigma, t)$ with $\sigma \in \Sigma$ and $t \in \{\cdot\} \cup [k]$. (In other words, since $\alpha_j$ is a pair of an alphabet symbol $\sigma$ and either a state in $[k]$ or the symbol $\cdot$, $\alpha_{j,0}$ is the first component $\sigma$ of this pair.)

- $M$’s head is in the unique position $i$ for which $\alpha_i$ has the form $(\sigma, s)$ for $s \in [k]$, and $M$’s state is equal to $s$.

**Definition 7.20** is a little cumbersome, but ultimately a configuration is simply a string that encodes a snapshot of the state of the NAND-TM program at a given point in the execution. (In operating-systems lingo, it would be a “core dump”.) Such a snapshot needs to encode the following components:

1. The current head position.

2. The full contents of the large scale memory, that is the tape.

3. The contents of the “local registers”, that is the state of the machine.

The precise details of how we encode a configuration are not important, but we do want to record the following simple fact:

**Lemma 7.21** Let $M$ be a Turing machine and let $\text{NEXT}_M : \Sigma^* \rightarrow \Sigma^*$ be the function that maps a configuration of $M$ to the configuration at the next step of the execution. Then for every $i \in \mathbb{N}$, the value of $\text{NEXT}_M(\alpha)_i$ only depends on the coordinates $\alpha_{i-1}, \alpha_i, \alpha_{i+1}$.

We leave proving **Lemma 7.21** as Exercise 7.4. It is not a hard exercise, but doing it is a great way to ensure that you are comfortable with the definition of configurations.

Once we have **Lemma 7.21** in place, we see that the function $\text{NEXT}_M$ that maps a configuration of $P$ into the next one is in fact a valid rule for a one dimensional automata, hence completing the proof of **Theorem 7.19**. The automaton arising from the proof of **Theorem 7.19** has a large alphabet, and furthermore one whose size that depends on the machine $M$ that is being simulated. It turns out that one can obtain an automaton with an alphabet of fixed size that is independent of the program being simulated, and in fact the alphabet of the automaton can be the minimal set $\{0, 1\}$! See Fig. 7.12 for an example of such an Turing-complete automaton.

---

**Figure 7.12**: Evolution of a one dimensional automata. Each row in the figure corresponds to the configuration. The initial configuration corresponds to the top row and contains only a single “live” cell. This figure corresponds to the “Rule 110” automaton of Stefan Wolfram which is Turing Complete. Figure taken from Wolfram MathWorld.
The formal definition is very cumbersome to state, and not crucial for the remainder of this book. Feel free to skip it as long as you understand the general concept of Turing equivalence. This notion is sometimes referred to in the literature as Gödel numbering or admissible numbering.

Remark 7.22 — Configurations as binary strings. We can represent a configuration $\alpha \in \Sigma^*$ by simply encoding each coordinate of $\alpha$ using $\log |\Sigma|$ bits. When we refer to a configuration as a binary string (for example when feeding it as input to other programs) we will assume that this string represents the configuration via the above encoding.

Remark 7.23 — Configurations of NAND-TM programs. We can use the same approach to define configurations of a NAND-TM program. Such a configuration will need to encode:

1. The current value of the variable $i$.
2. For every scalar variable $foo$, the value of $foo$.
3. For every array variable $Bar$, the value $Bar[j]$ for every $j \in \{0, \ldots, t - 1\}$ where $t - 1$ is the largest value that the index variable $i$ ever achieved in the computation.

7.7.4 Turing completeness and equivalence, a formal definition (optional)

A computational model is some way to define what it means for a program (which is represented by a string) to compute a (partial) function. A computational model $M$ is Turing complete, if we can map every Turing machine (or equivalently NAND-TM program) $N$ into a program $P$ for $M$ that computes the same function as $Q$. It is Turing equivalent if the other direction holds as well (i.e., we can map every program in $M$ to a Turing machine that computes the same function). Formally, we can define this notion as follows:\[19\]

**Definition 7.24 — Turing completeness and equivalence (optional).** Let $\mathcal{F}$ be the set of all partial functions from $\{0, 1\}^*$ to $\{0, 1\}^*$. A computational model is a map $M : \{0, 1\}^* \rightarrow \mathcal{F}$.

We say that a program $P \in \{0, 1\}^*$ $M$-computes a function $F \in \mathcal{F}$ if $M(P) = F$.

A computational model $M$ is Turing complete if there is a computable map $ENCODE_M : \{0, 1\}^* \rightarrow \{0, 1\}^*$ for every Turing machine $N$ (represented as a string), $M(ENCODE_M(N))$ is equal to the partial function computed by $P$.

A computational model $M$ is Turing equivalent if it is Turing complete and there exists a computable map $DECODE_M : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for every string $P \in \{0, 1\}^*$, $N = M(P)$.
DECODEₘ(𝑃) is a string representation of a Turing machine that computes the function ₘ(𝑃).

Some examples of Turing equivalent models include:

- Turing machines
- NAND-TM programs
- NAND-RAM programs
- Python, JavaScript, C, Lisp, and other programming languages.
- λ calculus
- Game of life (mapping programs and inputs/outputs to starting and ending configurations)
- Programming languages such as Python/C/JavaScript/OCaml... (allowing for unbounded storage)

### 7.8 THE CHURCH-TURING THESIS (DISCUSSION)

“[In 1934], Church had been speculating, and finally definitely proposed, that the λ-definable functions are all the effectively calculable functions. … When Church proposed this thesis, I sat down to disprove it … but, quickly realizing that [my approach failed], I became overnight a supporter of the thesis.”, Stephen Kleene, 1979.

“[The thesis is] not so much a definition or to an axiom but ... a natural law.”, Emil Post, 1936.

We have defined functions to be computable if they can be computed by a NAND-TM program, and we’ve seen that the definition would remain the same if we replaced NAND-TM programs by Python programs, Turing machines, λ calculus, cellular automata, and many other computational models. The Church-Turing thesis is that this is the only sensible definition of “computable” functions. Unlike the "Physical Extended Church Turing Thesis" (PECTT) which we saw before, the Church Turing thesis does not make a concrete physical prediction that can be experimentally tested, but it certainly motivates predictions such as the PECTT. One can think of the Church-Turing Thesis as either advocating a definitional choice, making some prediction about all potential computing devices, or suggesting some laws of nature that constrain the natural world. In Scott Aaronson’s words, “whatever it is, the Church-Turing thesis can only be regarded as extremely successful”. No candidate computing device (including quantum computers, and also much less reasonable models such as the hypothetical “closed time curve” computers we mentioned before) has so far mounted a serious challenge to the Church Turing thesis. These devices might potentially make some computations more effi-
cient, but they do not change the difference between what is finitely computable and what is not.\footnote{The extended Church Turing thesis, which we discuss in Section 12.6, stipulates that Turing machines capture also the limit of what can be efficiently computable. Just like its physical version, quantum computing presents the main challenge to this thesis.}

### 7.8.1 Different models of computation

We can summarize the models we have seen in the following table:

<table>
<thead>
<tr>
<th>Computational problems</th>
<th>Type of model</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite functions $f : {0, 1}^n \rightarrow {0, 1}^m$</td>
<td>Non uniform computation (algorithm depends on input length)</td>
<td>Boolean circuits, NAND circuits, straight-line programs (e.g., NAND-CIRC)</td>
</tr>
<tr>
<td>Functions with unbounded inputs $F : {0, 1}^* \rightarrow {0, 1}^*$</td>
<td>Sequential access to memory</td>
<td>Turing machines, NAND-TM programs</td>
</tr>
<tr>
<td></td>
<td>Indexed access / RAM</td>
<td>RAM machines, NAND-RAM, modern programming languages</td>
</tr>
<tr>
<td></td>
<td>Other</td>
<td>Lambda calculus, cellular automata</td>
</tr>
</tbody>
</table>

Later on in Chapter 16 we will study memory bounded computation. It turns out that NAND-TM programs with a constant amount of memory are equivalent to the model of finite automata (the adjectives “deterministic” or “nondeterministic” are sometimes added as well, this model is also known as finite state machines) which in turns captures the notion of regular languages (those that can be described by regular expressions), which is a concept we will see in Chapter 9.

### Lecture Recap

- While we defined computable functions using NAND-TM programs, we could just as well have done so using many other models, including not just NAND-RAM but also Turing machines, RAM machines, the $\lambda$-calculus and many other models.
- Very simple models turn out to be “Turing complete” in the sense that they can simulate arbitrarily complex computation.
7.9 EXERCISES

Exercise 7.1 — NAND-TM lookup. This exercise shows part of the proof that NAND-TM can simulate NAND-RAM. Produce the code of a NAND-TM program that computes the function \( \text{LOOKUP} : \{0, 1\}^* \to \{0, 1\} \) that is defined as follows. On input \( pf(i)x \), where \( pf(i) \) denotes a prefix-free encoding of an integer \( i \), \( \text{LOOKUP}(pf(i)x) = x_i \) if \( i < |x| \) and \( \text{LOOKUP}(pf(i)x) = 0 \) otherwise. (We don’t care what \( \text{LOOKUP} \) outputs on inputs that are not of this form.) You can choose any prefix-free encoding of your choice, and also can use your favorite programming language to produce this code.

Exercise 7.2 — Pairing. Let \( \text{embed} : \mathbb{N}^2 \to \mathbb{N} \) be the function defined as \( \text{embed}(x_0, x_1) = \frac{1}{2}(x_0 + x_1)(x_0 + x_1 + 1) + x_1 \).

1. Prove that for every \( x_0, x_1 \in \mathbb{N}, \text{embed}(x_0, x_1) \) is indeed a natural number.

2. Prove that \( \text{embed} \) is one-to-one

3. Construct a NAND-TM program \( P \) such that for every \( x_0, x_1 \in \mathbb{N}, P(pf(x_0)pf(x_1)) = pf(\text{embed}(x_0, x_1)) \), where \( pf \) is the prefix-free encoding map defined above. You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.

4. Construct NAND-TM programs \( P_0, P_1 \) such that for every \( x_0, x_1 \in \mathbb{N} \) and \( i \in \mathbb{N}, P_i(pf(\text{embed}(x_0, x_1))) = pf(x^i) \). You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.

Exercise 7.3 — lambda calculus requires three variables. Prove that for every \( \lambda \)-expression \( e \) with no free variables there is an equivalent \( \lambda \)-expression \( f \) that only uses the variables \( x, y, \) and \( z \).

Exercise 7.4 — Next-step function is local. Prove Lemma 7.21 and use it to complete the proof of Theorem 7.19.

7.10 BIBLIOGRAPHICAL NOTES

Chapters 7 in the wonderful book of Moore and Mertens [MM11] contains a great exposition much of this material. Chapter 3 in Savage’s book [Sav98] contains a more formal description of RAM machines,
see also the paper [Hag98]. A study of RAM algorithms that are independent of the input size (known as the “transdichotomous RAM model”) was initiated by [FW93].

The RAM model can be very useful in studying the concrete complexity of practical algorithms. However, the exact set of operations that are allowed in the RAM model at unit cost can vary between texts and contexts. One needs to be careful in making such definitions, especially if the word size grows, as was already shown by Shamir [Sha79].

The λ-calculus was described by Church in [Chu41]. Pierce’s book [Pie02] is a canonical textbook, see also [Bar84]. The “Currying technique” is named after the logician Haskell Curry (the Haskell programming language is named after Haskell Curry as well). Curry himself attributed this concept to Moses Schönfinkel, though for some reason the term “Schönfinkeling” never caught on.

Tao has proposed showing the Turing completeness of fluid dynamics (a “water computer”) as a way of settling the question of the behavior of the Navier-Stokes equations, see this popular article.