Learning Objectives:

- Learn about RAM machines and the λ calculus.
- Equivalence between these and other models and Turing machines.
- Cellular automata and configurations of Turing machines.
- Understand the Church-Turing thesis.

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 using Turing machines, which are not a close match 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 discuss the Church-Turing Thesis and the potential definitions of "reasonable" in Section 7.8.

Some of the main computational models we discuss in this chapter include:

- **RAM Machines:** Turing Machines do not correspond to standard computing architectures that have *Random Access Memory* (*RAM*). The mathematical model of RAM machines is much closer to actual computers, but we will see that it is equivalent in power to Turing Machines. We also discuss a programming language variant of RAM machines, which we call NAND-RAM. The equivalence of Turing Machines and RAM machines enables demonstrating the

Turing Equivalence of many popular programming languages, including all general-purpose languages used in practice such as C, Python,JavaScript, etc.

- **Cellular Automata:** Many natural and artificial systems can be modeled as collections of simple components, each evolving according to simple rules based on its state and the state of its immediate neighbors. One well-known such example is Conway's Game of Life. To prove that cellular automata are equivalent to Turing machines we introduce the tool of *configurations* of Turing Machines. These have other applications, and in particular are used in Chapter 10 to prove *Gödel's Incompleteness Theorem*: a central result in mathematics.
- λ **calculus:** The λ calculus is a model for expressing computation that originates from the 1930's, though it is closely connected to functional programming languages widely used today. Showing the equivalence of λ calculus to Turing Machines involves a beautiful technique to eliminate recursion known as the "Y Combinator".

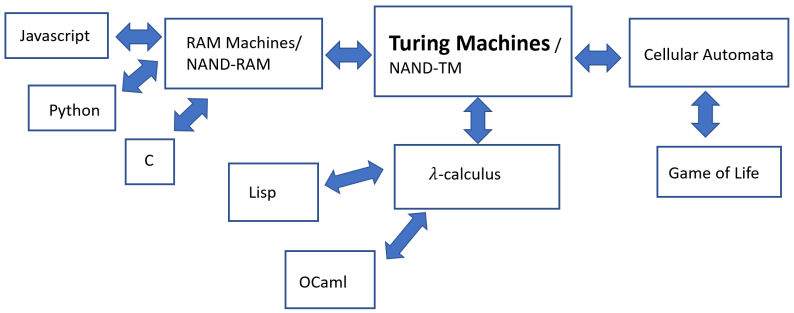

Figure 7.1: Some Turing-equivalent models. All of these are equivalent in power to Turing Machines (or equivalently NAND-TM programs) in the sense that they can compute exactly the same class of functions. All of these are models for computing *infinite* functions that take inputs of unbounded length. In contrast, Boolean circuits / NAND-CIRC programs can only compute *finite* functions and hence are not Turing complete.

7.1 RAM MACHINES AND NAND-RAM

One of the limitations of Turing Machines (and NAND-TM programs) is that we can only access one location of our arrays/tape at a time. If the head is at position 22 in the tape and we want to access the 957-th position 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 index p (i.e., memory address, or a *pointer*), we can read from and write to the p^{th} location of `Memory`. ("Random access memory" is quite a misnomer since it has nothing to do with probability, but since it is a standard term in both the theory and practice of computing, we will use it as well.)

The computational model that models access to such a memory is the *RAM machine* (sometimes also known as the *Word RAM model*), as depicted in Fig. 7.2. The memory of a RAM machine is an array of unbounded size where each cell can store a single *word*, which we think of as a string in $\{0,1\}^w$ and also (equivalently) as a number in $[2^w]$. For example, many modern computing architectures use 64 bit words, in which every memory location holds a string in $\{0,1\}^{64}$ which can also be thought of as a number between 0 and $2^{64}-1 = 9,223,372,036,854,775,807$. The parameter w is known as the *word size*. In practice often w is a fixed number such as 64, but when doing theory we model w as a parameter that can depend on the input length or number of steps. (You can think of 2^w as roughly corresponding to the largest memory address that we use in the computation.) In addition to the memory array, a RAM machine also contains a constant number of *registers* $r_0, \ldots, r_{k-1}$, each of which can also contain a single word.

The operations a RAM machine can carry out include:

- **Data movement:** Load data from a certain cell in memory into a register or store the contents of a register into a certain cell of memory. RAM machine can directly access any cell of memory without having to move the "head" (as Turing machines do) to that location. That is, in one step a RAM machine can load into register r_i the contents of the memory cell indexed by register r_j, or store into the memory cell indexed by register r_j the contents of register r_i.
- **Computation:** RAM machines can carry out computation on registers such as arithmetic operations, logical operations, and comparisons.
- **Control flow:** As in the case of Turing machines, the choice of what instruction to perform next can depend on the state of the RAM machine, which is captured by the contents of its register.

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 language* that models RAM 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

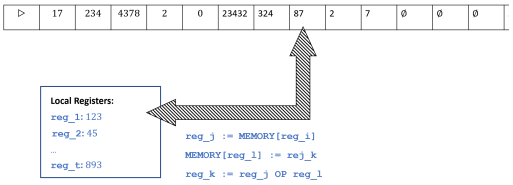

Figure 7.2: A *RAM Machine* contains a finite number of local registers, each of which holds an integer, and an unbounded memory array. It can perform arithmetic operations on its register as well as load to a register r the contents of the memory at the address indexed by the number in register r'.

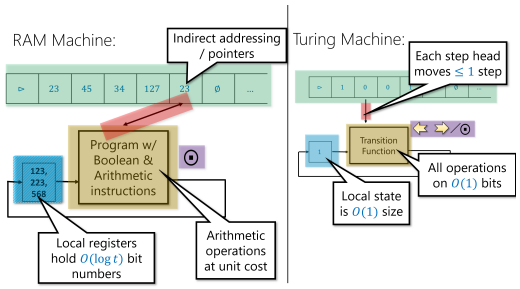

Figure 7.3: Different aspects of RAM machines and Turing machines. RAM machines can store integers in their local registers, and can read and write to their memory at a location specified by a register. In contrast, Turing machines can only access their memory in the head location, which moves at most one position to the right or left in each step.

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 $T - 1$, where T is the number of steps that have been executed by the program so far. (You can ignore this restriction for now: if we want to hold larger numbers, we can simply execute dummy instructions; it will be useful in later chapters.)

- 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 `NAND`, a zero valued integer is considered as *false*, and a nonzero valued integer is considered as *true*.
- In addition to `NAND`, NAND-RAM also includes all the basic arithmetic operations of addition, subtraction, multiplication, (integer) division, as well as comparisons (equal, greater than, less than, etc..).
- NAND-RAM includes conditional statements `if/then` as part of the language.
- NAND-RAM contains looping constructs such as `while` and `do` as part of the language.

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:

Theorem 7.1 — Turing Machines (aka NAND-TM programs) and RAM machines (aka NAND-RAM programs) are equivalent. 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-RAM program.

Since NAND-TM programs are equivalent to Turing machines, and NAND-RAM programs are equivalent to RAM machines, Theorem 7.1 shows that all these four models are equivalent to one another.

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

Figure 7.4: Overview of the steps in the proof of Theorem 7.1 simulating NANDRAM with NANDTM. We first use the inner loop syntactic sugar of Section 6.4.1 to enable loading an integer from an array to the index variable `i` of NANDTM. Once we can do that, we can simulate *indexed access* in NANDTM. We then use an embedding of $\mathbb{N}^2$ in $\mathbb{N}$ to simulate two dimensional bit arrays in NANDTM. Finally, we use the binary representation to encode one-dimensional arrays of integers as two dimensional arrays of bits hence completing the simulation of NANDRAM with NANDTM.

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. (See also Fig. 7.4.) NAND-RAM generalizes NAND-TM in two main ways: (**a**) adding *indexed access* to the arrays (ie.., `Foo[bar]` syntax) and (**b**) moving from *Boolean valued* variables to *integer valued* ones. The transformation has two steps:

1. *Indexed access of bit arrays:* We start by showing how to handle (**a**). Namely, we show how we can implement in NAND-TM the operation `Setindex(Bar)` such that if `Bar` is an array that encodes some integer j, then after executing `Setindex(Bar)` the value of `i` will equal to j. This will allow us to simulate syntax of the form `Foo[Bar]` by `Setindex(Bar)` followed by `Foo[i]`.

2. *Two dimensional bit arrays:* We 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 `Foo[i][j]` to access the `(i,j)`-th location of `Foo`.

3. *Arrays of integers:* Finally we will encode a one dimensional array `Arr` of *integers* by a two dimensional `Arrbin` of *bits*. The idea is simple: if $a_{i,0}, \ldots, a_{i,\ell}$ is a binary (prefix-free) representation of `Arr[`i`]`, then `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, `GOTO` etc. to implement the arithmetic and control flow operations of NAND-RAM.

⋆

The above approach is not the only way to obtain a proof of Theorem 7.1, see for example Exercise 7.1

R

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 so called von Neumann architecture as described in the report [Neu45]. As a result, NAND-RAM is sim-

ilar 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). Actual microprocessors also have a fixed number of registers (e.g., 14 general purpose registers in x86-64) but this 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 registers.

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

In NAND-TM we can only access our arrays in the position of the index variable `i`, while NAND-RAM has integer-valued variables and

can use them for *indexed access* to arrays, of the form `Foo[bar]`. To implement indexed access in NAND-TM, we will encode integers in our arrays using some prefix-free representation (see Section 2.4.2)), and then have a procedure `Setindex(Bar)` that sets `i` to the value encoded by `Bar`. We can simulate the effect of `Foo[Bar]` using `Setindex(Bar)` followed by `Foo[i]`.

Implementing `Setindex(Bar)` can be achieved as follows:

1. We initialize an array `Arzero` such that `Atzero[`0`]`$= 1$ and `Atzero[`j`]`$= 0$ for all $j > 0$. (This can be easily done in NAND-TM as all uninitialized variables default to zero.)

2. Set `i` to zero, by decrementing it until we reach the point where `Atzero[i]`$= 1$.

3. Let `Temp` be an array encoding the number 0.

4. We use `GOTO` to simulate an inner loop of of the form: **while** `Temp` $\neq$ `Bar`, increment `Temp`.

5. At the end of the loop, `i` is equal to the value encoded by `Bar`.

In NAND-TM code (using some syntactic sugar), we can implement the above operations as follows:

```
# 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 that we come up with a *one to one* function $embed : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and so embed the location (i, j) of the two dimensional array `Two` in the location $embed(i, j)$ of the array `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. 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 $embed$ would do (see Fig. 7.5):

$$embed(x, y) = \tfrac{1}{2}(x + y)(x + y + 1) + x \quad . \tag{7.1}$$

Exercise 7.3 asks you to prove that $embed$ is indeed one to one, as well as computable by a NAND-TM program. (The latter can be done by simply following the grade-school algorithms for multiplication, addition, and division.) This means that we can replace code of the form `Two[Foo][Bar] = something` (i.e., access the two dimensional array `Two` at the integers encoded by the one dimensional arrays `Foo` and `Bar`) by code of the form:

```
Blah = embed(Foo,Bar)
Setindex(Blah)
Two[i] = something
```

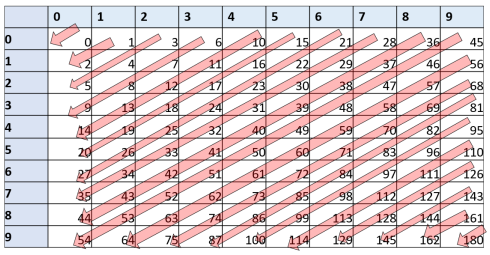

Figure 7.5: Illustration of the map $embed(x, y) = \frac{1}{2}(x + y)(x + y + 1) + x$ for $x, y \in [10]$, one can see that for every distinct pairs (x, y) and (x', y'), $embed(x, y) \neq embed(x', y')$.

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 comparisions 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.

R

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

```
Stack[stackpointer]=foo
stackpointer += one
```

and implement `bar = pop()` by

```
bar = Stack[stackpointer]
stackpointer -= 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 online tutorials on how recursion is implemented via stack in your favorite programming language, whether it's Python , JavaScript, or Lisp/Scheme.

7.3 TURING EQUIVALENCE (DISCUSSION)

Any of the standard programming language 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 6.12, Turing machines) us-

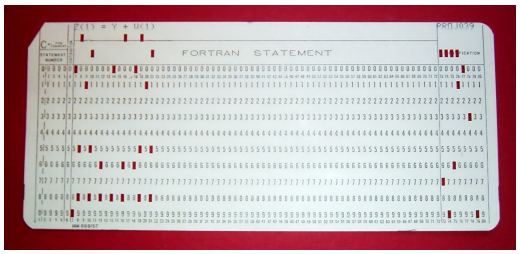

Figure 7.6: A punched card corresponding to a Fortran statement.

ing 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.[1]

[1] 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 *can not* 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 9 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.

Figure 7.7: By having the two equivalent languages NAND-TM and NAND-RAM, we can "have our cake and eat it too", using NAND-TM when we want to prove that programs *can't* do something, and using NAND-RAM or other high level languages when we want to prove that programs *can* do something.

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, Algorithm 7.4 is a high level description of the breadth first search algorithm.

Algorithm 7.4 — Breadth First Search.

Input: Graph G, vertices u, v
Output: "connected" when u is connected to v in G, "disconnected"

```
Initialize empty queue Q.
Put u in Q
while Q is not empty do
    Remove top vertex w from Q
    if w = v then
        return "connected"
    end if
    Mark w
    Add all unmarked neighbors of w to Q.
end while
return "disconnected"
```

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 mark vertices. 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.

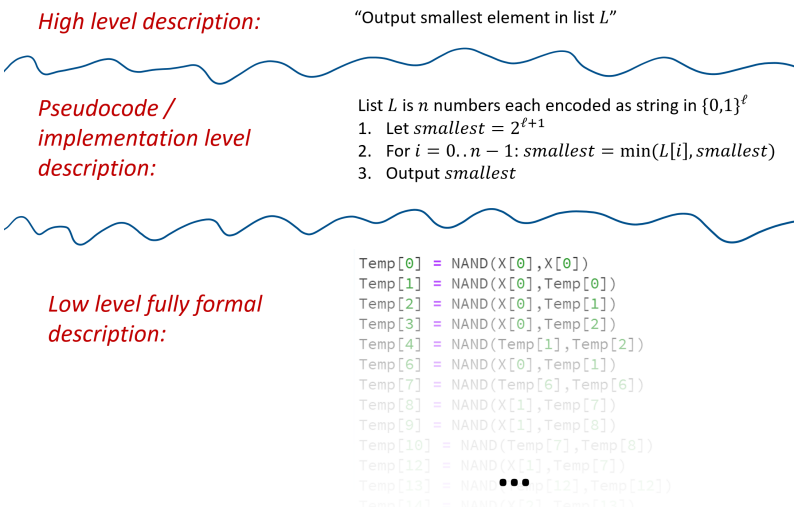

Figure 7.8: We can describe an algorithm at different levels of granularity/detail and precision. At the highest level we just write the idea in words, omitting all details on representation and implementation. In the intermediate level (also known as *implementation* or *pseudocode*) we give enough 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.

After all, our goal is not to use these models for actual computation, 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 $\overrightarrow{i\ j}$ 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 X (that can be a graph, a vector, a program, etc.) without specifying how X is encoded as a string.

Defining Algorithms. Up until now we have used the term "algorithm" informally. However, Turing Machines and the range of equivalent models yield 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.

7.3.3 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* $\mathcal{M}$ is *Turing complete*, if we can map every Turing machine (or equivalently NAND-TM program) N into a program P for $\mathcal{M}$ 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 $\mathcal{M}$ to a Turing machine that computes the same function). We can define this notion formally as follows. (This formal definition is not crucial for the remainder of this book so feel 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 admissable numbering.)

Definition 7.5 — 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 $\mathcal{M} : \{0,1\}^* \to \mathcal{F}$.

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

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

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

Some examples of Turing equivalent models (some of which we have already seen, and some are discussed below) include:

- Turing machines
- NAND-TM programs
- NAND-RAM programs
- 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)

7.4 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.

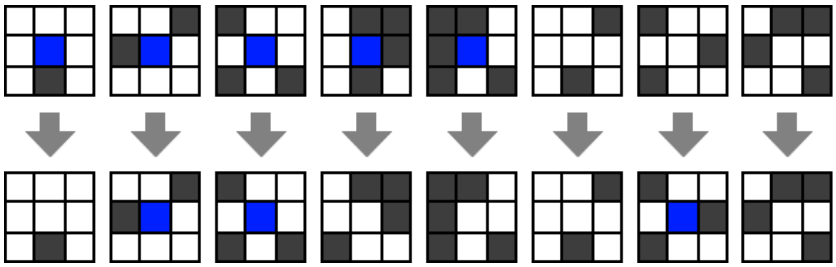

Figure 7.9: Rules for Conway's Game of Life. Image from this blog post.

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 $\varnothing$) 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 (see Fig. 7.9). 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 encode the state using a finite-length string 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. The Wikipedia page for the Game of Life contains some beautiful figures and animations of configurations that produce very interesting evolutions.

Since the cells in the game of life are are arranged in an infinite two-dimensional grid, it is an example of a *two dimensional cellular automaton*. We can also consider the even simpler setting of a *one dimensional cellular automaton*, where the cells are arranged in an infinite line, see Fig. 7.10. It turns out that even this simple model is enough to achieve

2 dimensional cellular automaton:

⋱	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋰
⋯	∅	∅	∅	∅	∅	∅	∅	⋯
⋯	∅	a	c	a	b	a	∅	⋯
⋯	∅	b	a	∅	c	a	∅	⋯
⋯	∅	a	b	c	a	b	∅	⋯
⋯	∅	∅	∅	∅	∅	∅	∅	⋯
⋰	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

1 dimensional cellular automaton:

⋯	∅	a	c	a	b	a	∅	⋯

Figure 7.10: In a *two dimensional cellular automaton* every cell is in position i, j for some integers $i, j \in \mathbb{Z}$. The *state* of a cell is some value $A_{i,j} \in \Sigma$ for some finite alphabet Σ. At a given time step, the state of the cell is adjusted according to some function applied to the state of (i, j) and all its neighbors $(i \pm 1, j \pm 1)$. In a *one dimensional cellular automaton* every cell is in position $i \in \mathbb{Z}$ and the state A_i of i at the next time step depends on its current state and the state of its two neighbors $i - 1$ and $i + 1$.

Turing-completeness. We will now formally define one-dimensional cellular automata and then prove their Turing completeness.

Definition 7.6 — One dimensional cellular automata. Let Σ be a finite set containing the symbol $\varnothing$. A *one dimensional cellular automation* over alphabet Σ is described by a *transition rule* $r : \Sigma^3 \to \Sigma$, which satisfies $r(\varnothing, \varnothing, \varnothing) = \varnothing$.

A *configuration* of the automaton r is a function $A : \mathbb{Z} \to \Sigma$. If an automaton with rule r is in configuration A, then its next configuration, denoted by $A' = NEXT_r(A)$. Is the function A' such that $A'(i) = r(A(i-1), A(i), A(i+1))$ for every $i \in \mathbb{Z}$. In other words, the next state of the automaton r at point i obtained by applying the rule r to the values of A at i and its two neighbors.

Finite configuration. We say that a configuration of an automaton r is *finite* if there is only some finite number $i_0, \ldots, i_{j-1}$ of indices in $\mathbb{Z}$ such that $A(i_j) \neq \varnothing$. (That is, for every $i \notin \{i_0, \ldots, i_{j-1}\}$, $A(i) = \varnothing$.) Such a configuration can be represented using a finite string that encodes the indices $i_0, \ldots, i_{n-1}$ and the values $A(i_0), \ldots, A(i_{n-1})$. Since $R(\varnothing, \varnothing, \varnothing) = \varnothing$, if A is a finite configuration then $NEXT_r(A)$ is finite as well. We will only be interested in studying cellular automata that are initialized in finite configurations, and hence remain in a finite configuration throughout their evolution.

7.4.1 One dimensional cellular automata are Turing complete

We can write a program (for example using NAND-RAM) that simulates the evolution of any cellular automaton from an initial finite configuration by simply storing the values of the cells with state not

equal to $\varnothing$ and repeatedly applying the rule r. Hence cellular automata can be simulated by Turing Machines. What is more surprising that the other direction holds as well. For example, as simple as its rules seem, we can simulate a Turing machine using the game of life (see Fig. 7.11).

In fact, even one dimensional cellular automata can be Turing complete:

Theorem 7.7 — 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.

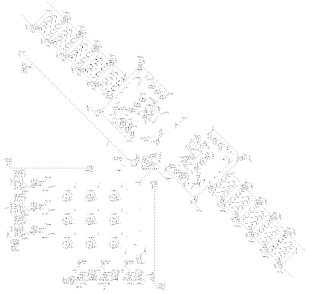

Figure 7.11: A Game-of-Life configuration simulating a Turing Machine. Figure by Paul Rendell.

To make the notion of "simulating a Turing machine" more precise we will need to define *configurations* of Turing machines. We will do so in Section 7.4.2 below, but at a high level a *configuration* of a Turing machine is a string that encodes its full state at a given step in its computation. That is, the contents of all (non empty) cells of its tape, its current state, as well as the head position.

The key idea in the proof of Theorem 7.7 is that at every point in the computation of a Turing machine M, the only cell in M's tape that can change is the one where the head is located, and the value this cell changes to is a function of its current state and the finite state of M. This observation allows us to encode the configuration of a Turing machine M as a finite configuration of a cellular automaton r, and ensure that a one-step evolution of this encoded configuration under the rules of r corresponds to one step in the execution of the Turing machine M.

7.4.2 Configurations of Turing machines and the next-step function

To turn the above ideas into a rigorous proof (and even statement!) of Theorem 7.7 we will need precisely define the notion of *configurations* of Turing machines. This notion will be useful for us in later chapters as well.

Definition 7.8 — Configuration of Turing Machines.. Let M be a Turing machine with tape alphabet Σ and state space $[k]$. A *configuration of* M is a string $\alpha \in \overline{\Sigma}^*$ where $\overline{\Sigma} = \Sigma \times (\{\cdot\} \cup [k])$ 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 $\alpha \in \overline{\Sigma}^*$ of M corresponds to the following state of its execution:

- M's tape contains $\alpha_{j,0}$ for all $j < |\alpha|$ and contains $\varnothing$ for all positions that are at least $|\alpha|$, where we let $\alpha_{j,0}$ be the value σ such

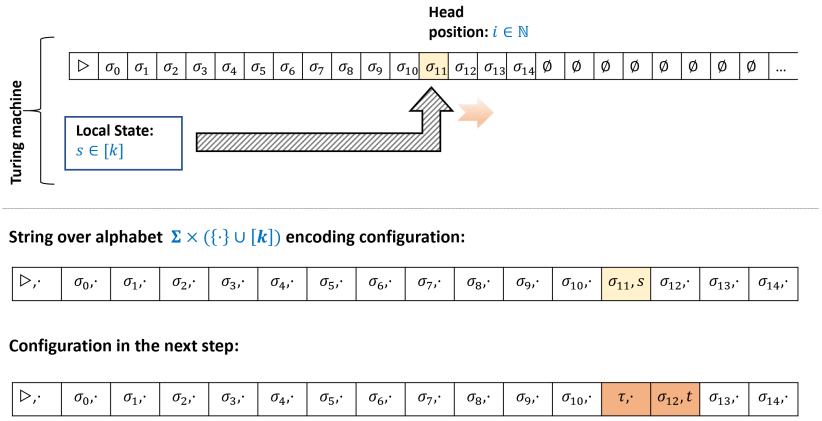

Figure 7.12: A *configuration* of a Turing machine M with alphabet Σ and state space $[k]$ encodes the state of M at a particular step in its execution as a string α over the alphabet $\overline{\Sigma} = \Sigma \times (\{\cdot\} \times [k])$. The string is of length t where t is such that M's tape contains $\varnothing$ 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 $j \neq i$, α_j encodes the value of the j-th cell of M's tape, while α_i encodes both this value as well as the current state of M. If the machine writes the value τ, changes state to t, and moves right, then in the next configuration will contain at position i the value $(\tau, \cdot)$ and at position $i+1$ the value (α_{i+1}, t).

that $\alpha_j = (\sigma, t)$ with $\sigma \in \Sigma$ and $t \in \{\cdot\} \cup [k]$. (In other words, since α_j is a pair of an alphabet symbol σ and either a state in $[k]$ or the symbol $\cdot$, $\alpha_{j,0}$ is the first component σ of this pair.)

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

Definition 7.8 below has some technical details, but is not actually that deep or complicated. Try to 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 finite symbols. 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 7.8 is a little cumbersome, but ultimately a configuration is simply a string that encodes a *snapshot* of the Turing machine at a given point in the execution. (In operating-systems lingo, it is 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 7.9 Let M be a Turing machine and let $NEXT_M : \overline{\Sigma}^* \to \overline{\Sigma}^*$ 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 $NEXT_M(\alpha)_i$ only depends on the coordinates $\alpha_{i-1}, \alpha_i, \alpha_{i+1}$.

(For simplicity of notation, above we use the convention that if i is "out of bounds", such as $i < 0$ or $i > |\alpha|$, then we assume that $\alpha_i = (\varnothing, \cdot)$.) We leave proving Lemma 7.9 as Exercise 7.7. The idea behind the proof is simple: if the head is neither in position i nor positions $i-1$ and $i+1$, then the next-step configuration at i will be the same as it was before. Otherwise, we can "read off" the state of the Turing machine and the value of the tape at the head location from the configuration at i or one of its neighbors and use that to update what the new state at i should be. Completing the full proof is not hard, but doing it is a great way to ensure that you are comfortable with the definition of configurations.

Completing the proof of Theorem 7.7. We can now restate Theorem 7.7 more formally, and complete its proof:

Theorem 7.10 — One dimensional automata are Turing complete (formal statement). For every Turing Machine M, if we denote by $\overline{\Sigma}$ the alphabet of its configuration strings, then there is a one-dimensional cellular automaton r over the alphabet $\overline{\Sigma}^*$ such that

$$(NEXT_M(\alpha)) = NEXT_r(\alpha) \tag{7.2}$$

for every configuration $\alpha \in \overline{\Sigma}^*$ of M (again using the convention that we consider $\alpha_i = \varnothing$ if i is "out of bounds").

Proof. We consider the element $(\varnothing, \cdot)$ of $\overline{\Sigma}$ to correspond to the $\varnothing$ element of the automaton r. In this case, by Lemma 7.9, the function $NEXT_M$ that maps a configuration of M into the next one is in fact a valid rule for a one dimensional automata.

■

The automaton arising from the proof of Theorem 7.10 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. 7.13 for an example of such an Turing-complete automaton.

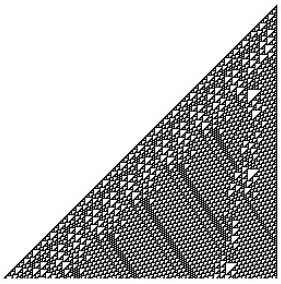

Figure 7.13: Evolution of a one dimensional automata. Each row in the figure corresponds to the configuration. The initial configuration corresponds to the top row and contains only a single "live" cell. This figure corresponds to the "Rule 110" automaton of Stefan Wolfram which is Turing Complete. Figure taken from Wolfram MathWorld.

R

Remark 7.11 — Configurations of NAND-TM programs. We can use the same approach as Definition 7.8 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.

7.5 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 giving a name f to a function and defining it as

$$f(x) = x \times x \tag{7.3}$$

we can write it as

$$\lambda x.x \times x \tag{7.4}$$

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 $x \mapsto exp(x)$. 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*x` while in *JavaScript* we can use `x => x*x` or

`(x) => x*x`. In *Scheme* we would define it as `(lambda (x) (* x 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.

Dropping parenthesis. To reduce notational clutter, when writing λ calculus expressions 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 $f\ x$. Therefore we can write $(\lambda x.x \times x)7 = 49$. In this chapter, we will use both the $f(x)$ and $f\ x$ notations for function application. Function evaluations are associative and bind from left to right, and hence $f\ g\ h$ is the same as $(fg)h$.

7.5.1 Applying functions to functions

A key feature of the λ calculus is that functions are "first-class objects" in the sense that we can use functions as arguments to other functions. For example, can you guess what number is the following expression equal to?

$$(((\lambda f.(\lambda y.(f\ (f\ y))))(\lambda x.x \times x))\ 3) \tag{7.5}$$

The expression (7.5) 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 λ calculus.

Let's evaluate (7.5) one step at a time. As nice as it is for the λ calculus to allow anonymous functions, adding names can be very helpful for understanding complicated expressions. So, let us write $F = \lambda f.(\lambda y.(f(fy)))$ and $g = \lambda x.x \times x$.

Therefore (7.5) becomes

$$((F\ g)\ 3)\ . \tag{7.6}$$

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

Solved Exercise 7.1 What number does the following expression equal to?

$$((\lambda x.(\lambda y.x))\ 2)\ 9)\ . \tag{7.7}$$

■

Solution:

$\lambda y.x$ is the function that on input y ignores its input and outputs x. Hence $(\lambda x.(\lambda y.x))2$ yields the function $y \mapsto 2$ (or, using λ notation, the function $\lambda y.2$). Hence (7.7) is equivalent to $(\lambda y.2)9 = 2$. ■

7.5.2 Obtaining multi-argument functions via Currying

In a λ expression of the form $\lambda x.e$, the expression e can itself involve the λ operator. Thus for example the function

$$\lambda x.(\lambda y.x + y) \tag{7.8}$$

maps x to the function $y \mapsto x + y$.

In particular, if we invoke the function (7.8) on a to obtain some function f, and then invoke f on b, we obtain 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 $(a, b) \mapsto a + b$. Generally, we can use the λ 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 $f(a, b)$.

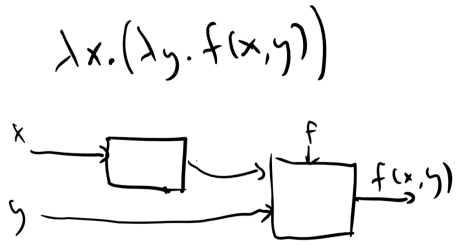

Figure 7.14: In the "currying" transformation, we can create the effect of a two parameter function $f(x, y)$ with the λ expression $\lambda x.(\lambda y.f(x, y))$ which on input x outputs a one-parameter function f_x that has x "hardwired" into it and such that $f_x(y) = f(x, y)$. This can be illustrated by a circuit diagram; see Chelsea Voss's site.

7.5.3 Formal description of the λ calculus.

We now provide a formal description of the λ calculus. We start with "basic expressions" that contain a single variable such as x or y and build more complex expressions of the form $(e\ e')$ and $\lambda x.e$ where e, e' are expressions and x is a variable idenifier. Formally λ expressions are defined as follows:

Definition 7.12 — λ expression.. A *λ expression* is either a single variable identifier or an expression e of the one of the following forms:

- **Application:** $e = (e'\ e'')$, where e' and e'' are λ expressions.
- **Abstraction:** If $e = \lambda x.(e')$ where e' is a λ expression.

Definition 7.12 is a *recursive definition* since we defined the concept of λ expressions in terms of itself. This 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.

Free and bound variables. Variables in a λ expression can either be *free* or *bound* to a λ operator (in the sense of Section 1.4.7). In a single-variable λ expression var, the variable var is free. The set of free and bound variables in an application expression $e = (e' \ e'')$ is the same as that of the underlying expressions e' and e''. In an abstraction expression $e = \lambda var.(e')$, all free occurences of var in e' are bound to the λ operator of e/ If you find the notion of free and bound variables confusing, you can avoid all these issues by using unique identifiers for all variables.

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 λ 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 Section 7.5.2, 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 λ expressions.

Equivalence of λ expressions. As we have seen in Solved Exercise 7.1, the rule that $(\lambda x.exp)exp'$ is equivalent to $exp[x \rightarrow exp']$ enables us to modify λ 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 λ expressions:

Definition 7.13 — Equivalence of λ expressions. Two λ expressions are *equivalent* if they can be made into the same expression by repeated applications of the following rules:

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

If exp is a λ 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 λ calculus naturally implies a computational model. Since in the λ

calculus the inputs can themselves be functions, we need to decide in what order we evaluate an expression such as

$$(\lambda x.f)(\lambda y.gz) \ . \tag{7.9}$$

There are two natural conventions for this:

- *Call by name* (aka *"lazy evaluation"*): We evaluate (7.9) by first plugging in the righthand expression $(\lambda y.gz)$ as input to the lefthand side function, obtaining $f[x \rightarrow (\lambda y.gz)]$ and then continue from there.
- *Call by value* (aka *"eager evaluation"*): We evaluate (7.9) by first evaluating the righthand side and obtaining $h = g[y \rightarrow z]$, and then plugging this into the lefthandside to obtain $f[x \rightarrow h]$.

Because the λ 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, for concreteness we will always use the "call by name" (i.e., lazy evaluation) order. (The same choice is made in the programming language Haskell, though many other programming languages use eager evaluation.) Formally, the evaluation of a λ expression using "call by name" is captured by the following process:

Definition 7.14 — Simplification of λ expressions. Let e be a λ expression. The *simplification* of e is the result of the following recursive process:

1. If e is a single variable x then the simplification of e is e.
2. If e has the form $e = \lambda x.e'$ then the simplification of e is $\lambda x.f'$ where f' is the simplification of f'.
3. (*Evaluation / β reduction.*) If e has the form $e = (\lambda x.e' \ e'')$ then the simplification of e is the simplification of $e'[x \rightarrow e'']$, which denotes replacing all copies of x in e' bound to the λ operator with e''
4. (*Renaming / α conversion.*) The *canonical simplification* of e is obtained by taking the simplification of e and renaming 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.

We say that two λ expressions e and e' are *equivalent*, denoted by $e \cong e'$, if they have the same canonical simplification.

Solved Exercise 7.2 — Equivalence of λ expressions. Prove that the following two expressions e and f are equivalent:

$$e = \lambda x.x \tag{7.10}$$

$$f = (\lambda a.(\lambda b.b))(\lambda z.zz) \tag{7.11}$$

■

Solution:

The canonical simplification of e is simply $\lambda v_0.v_0$. To do the canonical simplification of f we first use β reduction to plug in $\lambda z.zz$ instead of a in $(\lambda b.b)$ but since a is not used in this function at all, we simply obtained $\lambda b.b$ which simplifies to $\lambda v_0.v_0$ as well.

■

7.5.4 Infinite loops in the λ calculus

Like Turing machines and NAND-TM programs, the simplification process in the λ calculus can also enter into an infinite loop. For example, consider the λ expression

$$\lambda x.xx\ \lambda x.xx \tag{7.12}$$

If we try to simplify (7.12) by invoking the lefthand function on the righthand one, then we get another copy of (7.12) and hence this never ends. There are examples where the order of evaluation can matter for whether or not an expression can be simplified, see Exercise 7.9.

7.6 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" on top of the "pure" (i.e., non enhanced) λ calculus. Hence the pure 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:** There are λ expressions 0, 1 and *IF* that satisfy the following conditions: for every λ expression

e and f, $IF\ 1\ e\ f = e$ and $IF\ 0\ e\ f = f$. That is, *IF* is the function that given three arguments a, e, f outputs e if $q = 1$ and f if$a = 0$.

- **Pairs:** There is a λ expression *PAIR* which we will think of as the *pairing* function. For every λ expressions e, f $PAIR\ e\ f$ if the pair $\langle e, f \rangle$ that contains e as its first member and f as its second member. We also have λ expressions *HEAD* and *TAIL* that extract the first and second member of a pair respectively. Hence, for every λ expressions e, f, $HEAD\ (PAIR\ e\ f) = e$ and $TAIL\ (PAIR\ e\ f) = f$..[2]

[2] In Lisp, the *PAIR*, *HEAD* and *TAIL* functions are traditionally called `cons`, `car` and `cdr`.

- **Lists and strings:** There is λ expression *NIL* that corresponds to the *empty list*, which we also denote by $\langle \bot \rangle$. Using *PAIR* and *NIL* we construct *lists*. The idea is that if L is a k element list of the form $\langle e_1, e_2, \ldots, e_k, \bot \rangle$ then for every λ expression e_0 we can obtain the $k+1$ element list $\langle e_0, e_1, e_2, \ldots, e_k, \bot \rangle$ using the expression $PAIR e_0 L$. For example, for every three λ expressions e, f, g, the following corresponds to the three element list $\langle e, f, g, \bot \rangle$:

$$PAIR\ e\ (PAIR\ f\ (PAIR\ g\ NIL)) \ . \tag{7.13}$$

The λ expression *ISEMPTY* 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 $L = \langle x_0, \ldots, x_{n-1}, \bot \rangle$ and a function f, $MAP\ L\ f$ applies f on every member of the list to obtain the new list $L' = \langle f(x_0), \ldots, f(x_{n-1}), \bot \rangle$. Given a list L as above and an expression f whose output is either 0 or 1, $FILTER\ L\ f$ returns the list $\langle x_i \rangle_{fx_i=1}$ 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. More generally, *REDUCE* takes a list L, an operation f (which we think of as taking two arguments) and a λ 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

$$REDUCE\ L\ f\ z = \begin{cases} z & L = NIL \\ f\ (HEAD L)\ (REDUCE\ (TAIL L)\ f\ z) & \text{otherwise} \end{cases} . \tag{7.14}$$

See Fig. 7.16 for an illustration of the three list-processing operations.

- **Recursion:** Finally, we want to be able to execute *recursive functions*. Since in λ calculus functions are *anonymous*, we can't write

a definition of the form $f(x) = blah$ where $blah$ includes calls to f. Instead we use functions f that take an additional input me as a parameter. The operator *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 *RECURSE* F will be the function f taking one parameter x such that $f(x) = F(f,x)$ for every x.

Solved Exercise 7.3 — Compute NAND using λ calculus. Give a λ expression N such that $N\ x\ y = NAND(x,y)$ for every $x, y \in \{0,1\}$.

■

Solution:

The *NAND* of x, y is equal to 1 unless $x = y = 1$. Hence we can write

$$N = \lambda x, y.IF(x, IF(y,0,1),1) \tag{7.15}$$

■

Solved Exercise 7.4 — Compute XOR using λ calculus. Give a λ expression *XOR* such that for every list $L = \langle x_0, \ldots, x_{n-1}, \perp \rangle$ where $x_i \in \{0,1\}$ for $i \in [n]$, $XORL$ evaluates to $\sum x_i \mod 2$.

■

Solution:

First, we note that we can compute XOR of two bits as follows:

$$NOT = \lambda a.IF(a,0,1) \tag{7.16}$$

and

$$XOR_2 = \lambda a, b.IF(b, NOT(a), a) \tag{7.17}$$

(We are using here a bit of syntactic sugar to describe the functions. To obtain the λ expression for XOR we will simply replace the expression (7.16) in (7.17).) Now recursively we can define the XOR of a list as follows:

$$XOR(L) = \begin{cases} 0 & L \text{ is empty} \\ XOR_2(HEAD(L), XOR(TAIL(L))) & \text{otherwise} \end{cases} \tag{7.18}$$

This means that *XOR* is equal to

$$RECURSE\ (\lambda me, L.IF(ISEMPTY(L), 0, XOR_2(HEAD\ L\ \ ,\ \ me(TAIL\ L)))) \ . \tag{7.19}$$

That is, *XOR* is obtained by applying the *RECURSE* operator to the function that on inputs me, L, returns 0 if $ISEMPTY(L)$ and otherwise returns XOR_2 applied to $HEAD(L)$ and to $me(TAIL(L))$.

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

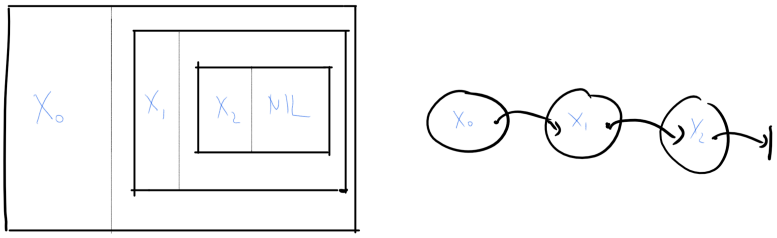

Figure 7.15: A list $\langle x_0, x_1, x_2 \rangle$ in the λ calculus is constructed from the tail up, building the pair $\langle x_2, NIL \rangle$, then the pair $\langle x_1, \langle x_2, NIL \rangle \rangle$ and finally the pair $\langle x_0, \langle x_1, \langle x_2, NIL \rangle \rangle \rangle$. 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 *link*, *pointer* or *reference* to the remainder of the list.

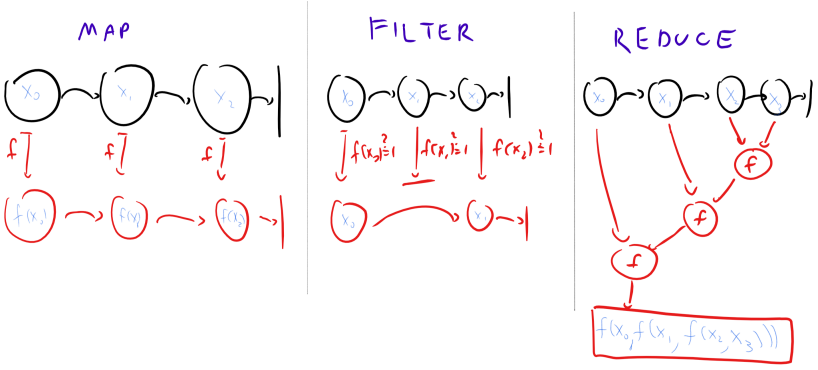

Figure 7.16: Illustration of the *MAP*, *FILTER* and *REDUCE* operations.

7.6.1 Computing a function in the enhanced λ calculus

An *enhanced λ expression* is obtained by composing the objects above with the *application* and *abstraction* rules. The result of simplifying a λ expression is an equivalent expression, and hence if two expressions have the same simplification then they are equivalent.

Definition 7.15 — Computing a function via λ calculus. Let $F : \{0,1\}^* \to \{0,1\}^*$

We say that exp *computes* F if for every $x \in \{0,1\}^*$,

$$exp\langle x_0, \ldots, x_{n-1}, \bot \langle \cong \langle y_0, \ldots, y_{m-1}, \bot \rangle \tag{7.20}$$

where $n = |x|$, $y = F(x)$, and $m = |y|$, and the notion of equivalence is defined as per Definition 7.14.

7.6.2 Enhanced λ calculus is Turing-complete

The basic operations of the enhanced λ calculus more or less amount to the Lisp or Scheme programming languages Given that, it is perhaps not surprising that the enhanced λ-calculus is equivalent to Turing machines:

Theorem 7.16 — Lambda calculus and NAND-TM. For every function $F : \{0,1\}^* \to \{0,1\}^*$, F is computable in the enhanced λ 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 λ 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 λ calculus expression.

Showing **(1)** is fairly straightforward. Applying the simplification rules to a λ 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. We give the details below but how this can be done is a good exercise in mastering some functional programming techniques that are useful in their own right.

⋆

Proof of Exercise 7.11. We only sketch the proof. The "if" direction is simple. As mentioned above, evaluating λ 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 using a λ expression. We will do so by first showing that showing for every Turing machine M a λ expression to compute the next-step

function $NEXT_M : \overline{\Sigma}^* \rightarrow \overline{\Sigma}^*$ that maps a configuration of M to the next one (see Section 7.4.2).

A configuration of M is a string $\alpha \in \overline{\Sigma}^*$ for a finite set $\overline{\Sigma}$. We can encode every symbol $\sigma \in \overline{\Sigma}$ by a finite string $\{0,1\}^\ell$, and so we will encode a configuration α in the λ calculus as a list $\langle \alpha_0, \alpha_1, \ldots, \alpha_{m-1}, \bot \rangle$ where α_i is an ℓ-length string (i.e., an ℓ-length list of 0's and 1's) encoding a symbol in $\overline{\Sigma}$.

By Lemma 7.9, for every $\alpha \in \overline{\Sigma}^*$, $NEXT_M(\alpha)_i$ is equal to $r(\alpha_{i-1}, \alpha_i, \alpha_{i+1})$ for some finite function $r : \overline{\Sigma}^3 \rightarrow \overline{\Sigma}$. Using our encoding of $\overline{\Sigma}$ as $\{0,1\}^\ell$, we can also think of r as mapping $\{0,1\}^{3\ell}$ to $\{0,1\}^\ell$. By Solved Exercise 7.3, we can compute the *NAND* function, and hence *every* finite function, including r, using the λ calculus. Using this insight, we can compute $NEXT_M$ using the λ calculus as follows. Given a list L encoding the configuration $\alpha_0 \cdots \alpha_{m-1}$, we define the lists L_{prev} and L_{next} encoding the configuration α shifted by one step to the right and left respectively. The next configuration α' is defined as $\alpha'_i = r(L_{prev}[i], L[i], L_{next}[i])$ where we let $L'[i]$ denote the i-th element of L'. This can be computed by recursion (and hence using the enhanced λ calculus' *RECURSE* operator) as follows:

Algorithm 7.17 — $NEXT_M$ using the λ calculus.

Input: List $L = \langle \alpha_0, \alpha_1, \ldots, \alpha_{m-1}, \bot \rangle$ encoding a configuration α.

Output: List L' encoding $NEXT_M(\alpha)$

```
1:  procedure ComputeNext(L_prev, L, L_next)
2:      if then ISEMPTY L_prev
3:          return NIL
4:      end if
5:      a ← HEAD L_prev
6:      if then ISEMPTY L
7:          b ← ∅                          # Encoding of ∅ in {0,1}^ℓ
8:      else
9:          b ← HEAD L
10:     end if
11:     if then ISEMPTY L_next
12:         c ← ∅
13:     else
14:         c ← HEAD L_next
15:     end if
16:     return PAIR r(a,b,c) ComputeNext(TAIL L_prev , TAIL L , TAIL L_next)
17: end procedure
18: L_prev ← PAIR ∅ L          # L_prev = ⟨∅, α_0, ..., α_{m-1}, ⊥⟩
19: L_next ← TAIL L            # L_next = ⟨α_1, ..., α_{m-1}, ⊥}
20: return ComputeNext(L_prev, L, L_next)
```

Once we can compute $NEXT_M$, we can simulate the execution of M on input x using the following recursion. Define $FINAL(\alpha)$ to be the final configuration of M when initialized at configuration α. The function $FINAL$ can be defined recursively as follows:

$$FINAL(\alpha) = \begin{cases} \alpha & \alpha \text{ is halting configuration} \\ NEXT_M(\alpha) & \text{otherwise} \end{cases} . \tag{7.21}$$

Checking whether a configuration is halting (i.e., whether it is one in which the transition function would output Halt) can be easily implemented in the λ calculus, and hence we can use the *RECURSE* to compute *FINAL*. If we let α^0 be the *initial configuration* of M on input x then we can obtain the output $M(x)$ from $FINAL(\alpha^0)$, hence completing the proof.

■

7.7 FROM ENHANCED TO PURE λ CALCULUS

While the collection of "basic" functions we allowed for the enhanced λ 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?

It turns out that there is in fact a proper subset of the operations of the enhanced λ calculus 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 λ formalism only, even without using 0's and 1's. It's λ's all the way down!

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 *MAP* using *REDUCE*, and then *REDUCE* using *RECURSE* combined with $0, 1, IF, PAIR, HEAD, TAIL, NIL, ISEMPTY$. You can also *PAIR*, *HEAD* and *TAIL* based on $0, 1, IF$. The most challenging part is to implement *RECURSE* using only the operations of the pure λ calculus.

Theorem 7.18 — Enhanced λ calculus equivalent to pure λ calculus.. There are λ expressions that implement the functions 0,1,*IF*,*PAIR*, *HEAD*, *TAIL*, *NIL*, *ISEMPTY*, *MAP*, *REDUCE*, and *RECURSE*.

The idea behind Theorem 7.18 is that we encode 0 and 1 themselves as λ 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 λ calculus can be a basis for all computation. We will not write the full formal proof of Theorem 7.18 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. We use Currying to achieve the effect of two-input functions and hence $0 = \lambda x.\lambda y.y$ and $1 = \lambda x.\lambda y.x$. (This representation scheme is the common convention for representing `false` and `true` but there are many other alternative representations for 0 and 1 that would have worked just as well.)
- The above implementation makes the *IF* function trivial: $IF(cond, a, b)$ is simply $cond\ a\ b$ since $0ab = b$ and $1ab = a$. We can write $IF = \lambda x.x$ to achieve $IF(cond, a, b) = (((IF cond)a)b) = cond\ a\ b$.

- 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, $PAIR = \lambda x, y.\, (\lambda g.gxy)$. We can extract the first element of a pair p by writing $p1$ and the second element by writing $p0$, and so $HEAD = \lambda p.p1$ and $TAIL = \lambda p.p0$.

- We define *NIL* to be the function that ignores its input and always outputs 1. That is, $NIL = \lambda x.1$. The *ISEMPTY* function checks, given an input p, whether we get 1 if we apply p to the function $zero = \lambda x, y.0$ that ignores both its inputs and always outputs 0. For every valid pair of the form $p = PAIRxy$, $pzero = pxy = 0$ while $NILzero = 1$. Formally, $ISEMPTY = \lambda p.p(\lambda x, y.0)$.

R

Remark 7.19 — 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 $ITER_n$ that on input a function f outputs the function $x \mapsto f(f(\cdots f(x)))$ (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 $PLUS(n, m)$ as $\lambda f.\lambda x.(nf)((mf)x)$ and $TIMES(n, m)$ as $\lambda f.n(mf)$. Subtraction and division are trickier, but can be achieved using recursion. (Working this out is a great exercise.)

7.7.1 List processing

Now we come to a bigger hurdle, which is how to implement *MAP*, *FILTER*, *REDUCE* and *RECURSE* in the pure λ calculus. It turns out that we can build *MAP* and *FILTER* from *REDUCE*, and *REDUCE* from *RECURSE*. For example $MAP(L, f)$ is the same as $REDUCE(L, g)$ where g is the operation that on input x and y, outputs $PAIR(f(x), NIL)$ if y is NIL and otherwise outputs $PAIR(f(x), y)$. (I leave checking this as a (recommended!) exercise for you, the reader.)

We can define $REDUCE(L, g)$ recursively, by setting $REDUCE(NIL, g) = NIL$ and stipulating that given a non-empty list L, which we can think of as a pair $(head, rest)$,

$REDUCE(L,g) = g(head, REDUCE(rest,g)))$. Thus, we might try to write a recursive λ expression for *REDUCE* as follows

$$REDUCE = \lambda L,g.IF(ISEMPTY(L), NIL, gHEAD(L)REDUCE(TAIL(L),g)) \; . \tag{7.22}$$

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 *RECURSE* operator to solve this problem. We will replace the recursive call to "*REDUCE*" with a call to a function me that is given as an extra argument, and then apply *RECURSE* to this. Thus $REDUCE = RECURSE\ myREDUCE$ where

$$myREDUCE = \lambda me,L,g.IF(ISEMPTY(L), NIL, gHEAD(L)me(TAIL(L),g)) \; . \tag{7.23}$$

7.7.2 The Y combinator, or recursion without recursion

Eq. (7.23) means that implementing *MAP*, *FILTER*, and *REDUCE* boils down to implementing the *RECURSE* operator in the pure λ calculus. This is what we do now.

How can we implement recursion without recursion? We will illustrate this using a simple example - the *XOR* function. As shown in Solved Exercise 7.4, we can write the *XOR* function of a list recursively as follows:

$$XOR(L) = \begin{cases} 0 & L \text{ is empty} \\ XOR_2(HEAD(L), XOR(TAIL(L))) & \text{otherwise} \end{cases} \tag{7.24}$$

where $XOR_2 : \{0,1\}^2 \rightarrow \{0,1\}$ is the XOR on two bits. In *Python* we would write this as

```
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

print(xor([0,1,1,0,0,1]))
# 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 λ calculus of course) to give a function *itself* as input. So, our idea is to try to come up with a *non recursive* function `tempxor` that takes *two inputs*: a function and a list, and such that `tempxor(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 `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([1,1,0,0,1,1,1,1]))
# 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 $me\ \ me$ and similarly the function $x \mapsto tempf(tempf, x)$ is simply $tempf\ \ tempf$. (Can you see why?) Therefore the function `tempf` defined above can be written as `λ me. myf(me me)`. This means that if we denote the input of `RECURSE` by f, then $RECURSE\ myf = tempf\ tempf$ where $tempf = \lambda m.f(m\ m)$ or in other words

$$RECURSE = \lambda f.((\lambda m.f(m\ m))\ \ (\lambda m.f(m\ m))) \tag{7.25}$$

The online appendix contains an implementation of the λ calculus using Python. Here is an implementation of the recursive XOR function from that appendix:[3]

```
# XOR of two bits
XOR2 = λ(a,b)(IF(a,IF(b,_0,_1),b))

# Recursive XOR with recursive calls replaced by m
 ↪  parameter
myXOR = λ(m,l)(IF(ISEMPTY(l),_0,XOR2(HEAD(l),m(TAIL(l)))))

# Recurse operator (aka Y combinator)
RECURSE = λf((λm(f(m*m)))(λm(f(m*m))))

# XOR function
```

[3] Because of specific issues of Python syntax, in this implementation we use `f * g` for applying `f` to `g` rather than `fg`, and use `λx(exp)` rather than `λx.exp` for abstraction. We also use `_0` and `_1` for the λ terms for 0 and 1 so as not to confuse with the Python constants.

```
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
```

Remark 7.20 — 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 $f = Ff$. 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.

7.8 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 *efficient*, but they do not change the difference between what is finitely computable and what is not. (The *extended* Church Turing thesis, which we discuss in Section 12.3, 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 7.1: Different models for computing finite functions and functions with arbitrary input length.

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

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 Turing machines, we could just as well have done so using many other models, including not just NAND-TM programs but also RAM machines, NAND-RAM, the λ-calculus, cellular automata 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 — Alternative proof for TM/RAM equivalence. Let $SEARCH : \{0,1\}^* \rightarrow \{0,1\}^*$ be the following function. The input is a pair (L,k) where $k \in \{0,1\}^*$, L is an encoding of a list of *key value* pairs $(k_0,v_1),\ldots,(k_{m-1},v_{m-1})$ where $k_0,\ldots,k_{m-1}$, $v_0,\ldots,v_{m-1}$ are binary strings. The output is v_i for the smallest i such that $k_i = k$, if such i exists, and otherwise the empty string.

1. Prove that *SEARCH* is computable by a Turing machine.
2. Let $UPDATE(L,k,v)$ be the function whose input is a list L of pairs, and whose output is the list L' obtained by prepending the pair (k,v) to the beginning of L. Prove that *UPDATE* is computable by a Turing machine.
3. Suppose we encode the configuration of a NAND-RAM program by a list L of key/value pairs where the key is either the name of a scalar variable `foo` or of the form `Bar[<num>]` for some number `<num>` and it contains all the nonzero values of variables. Let $NEXT(L)$ be the function that maps a configuration of a NAND-RAM program at one step to the configuration in the next step. Prove that *NEXT* is computable by a Turing machine (you don't have to implement each one of the arithmetic operations: it is enough to implement addition and multiplication).
4. Prove that for every $F : \{0,1\}^* \rightarrow \{0,1\}^*$ that is computable by a NAND-RAM program, F is computable by a Turing machine.

■

Exercise 7.2 — 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 $LOOKUP : \{0,1\}^* \rightarrow \{0,1\}$ that is defined as follows. On input $pf(i)x$, where $pf(i)$ denotes a prefix-free encoding of an integer i, $LOOKUP(pf(i)x) = x_i$ if $i < |x|$

and $LOOKUP(pf(i)x) = 0$ otherwise. (We don't care what *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 7.3 — Pairing. Let $embed : \mathbb{N}^2 \to \mathbb{N}$ be the function defined as $embed(x_0, x_1) = \frac{1}{2}(x_0 + x_1)(x_0 + x_1 + 1) + x_1$.

1. Prove that for every $x^0, x^1 \in \mathbb{N}$, $embed(x^0, x^1)$ is indeed a natural number.
2. Prove that $embed$ is one-to-one
3. Construct a NAND-TM program P such that for every $x^0, x^1 \in \mathbb{N}$, $P(pf(x^0)pf(x^1)) = pf(embed(x^0, x^1))$, 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 for every $x^0, x^1 \in \mathbb{N}$ and $i \in N$, $P_i(pf(embed(x^0, x^1))) = pf(x^i)$. You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.

Exercise 7.4 — Shortest Path. Let $SHORTPATH : \{0,1\}^* \to \{0,1\}^*$ be the function that on input a string encoding a triple (G, u, v) outputs a string encoding ∞ if u and v are disconnected in G or a string encoding the length k of the shortest path from u to v. Prove that *SHORTPATH* is computable by a Turing machine. See footnote for hint.[4]

[4] You don't have to give a full description of a Turing machine: use our "eat the cake and have it too" paradigm to show the existence of such a machine by arguing from more powerful equivalent models.

Exercise 7.5 — Longest Path. Let $LONGPATH : \{0,1\}^* \to \{0,1\}^*$ be the function that on input a string encoding a triple (G, u, v) outputs a string encoding ∞ if u and v are disconnected in G or a string encoding the length k of the *longest simple path* from u to v. Prove that *LONGPATH* is computable by a Turing machine. See footnote for hint.[5]

[5] Same hint as Exercise 7.5 applies. Note that for showing that *LONGPATH* is computable you don't have to give an *efficient* algorithm.

Exercise 7.6 — Shortest path λ expression. Let *SHORTPATH* be as in Exercise 7.4. Prove that there exists a λ expression that computes *SHORTPATH*. You can use Exercise 7.4

Exercise 7.7 — Next-step function is local. Prove Lemma 7.9 and use it to complete the proof of Theorem 7.7.

Exercise 7.8 — λ calculus requires at most three variables. Prove that for every λ-expression e with no free variables there is an equivalent λ-expression f that only uses the variables x,y, and z.[6]

Exercise 7.9 — Evaluation order example in λ calculus.

1. Let $e = \lambda x.7\,((\lambda x.xx)(\lambda x.xx))$. Prove that the simplification process of e ends in a definite number if we use the "call by name" evaluation order while it never ends if we use the "call by value" order.

2. (bonus, challenging) Let e be any λ expression. Prove that if the simplification process ends in a definite number if we use the "call by value" order then it also ends in such a number if we use the "call by name" order. See footnote for hint.[7]

Exercise 7.10 — Zip function. Give an enhanced λ calculus expression to compute the function 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 zip "zips together" these two lists of elements into a single list of pairs.[8]

Exercise 7.11 — Next-step function without $RECURSE$. Let M be a Turing machine. Give an enhanced λ calculus expression to compute the next-step function $NEXT_M$ of M (as in the proof of Exercise 7.11) *without using RECURSE*. See footnote for hint.[9]

Exercise 7.12 — λ calculus to NAND-TM compiler (challenging). Give a program in the programming language of your choice that takes as input a λ expression e and outputs a NAND-TM program P that computes the same function as e. For partial credit you can use the `GOTO` and all NAND-CIRC syntactic sugar in your output program. You can use any encoding of λ expressions as binary string that is convenient for you. See footnote for hint.[10]

Exercise 7.13 — At least two in λ calculus. Let $1 = \lambda x, y.x$ and $0 = \lambda x, y.y$ as before. Define

$$ALT = \lambda a, b, c.(a(b1(c10))(bc0)) \tag{7.26}$$

Prove that ALT is a λ expression that computes the *at least two* function. That is, for every $a, b, c \in \{0, 1\}$ (as encoded above) $ALTabc = 1$ if and only at least two of $\{a, b, c\}$ are equal to 1.

[6] **Hint:** You can reduce the number of variables a function takes by "pairing them up". That is, define a λ expression *PAIR* such that for every x, y $PAIRxy$ is some function f such that $f0 = x$ and $f1 = y$. Then use *PAIR* to iteratively reduce the number of variables used.

[7] Use structural induction on the expression e.

[8] The name zip is a common name for this operation, for example in Python. It should not be be confused with the `zip` compression file format.

[9] Use *MAP* and *REDUCE* (and potentially *FILTER*). You might also find the function zip of Exercise 7.10 useful.

[10] Try to set up a procedure such that if array `Left` contains an encoding of a λ expression $\lambda x.e$ and array `Right` contains an encoding of another λ expression e', then the array `Result` will contain $e[x \to e']$.

Exercise 7.14 — Locality of next-step function. This question will help you get a better sense of the notion of *locality of the next step function* of Turing Machines. This locality plays an important role in results such as the Turing completeness of λ calculus and one dimensional cellular automata, as well as results such as Godel's Incompleteness Theorem and the Cook Levin theorem that we will see later in this course. Define `STRINGS` to be the a programming language that has the following semantics:

- A `STRINGS` program Q has a single string variable `str` that is both the input and the output of Q. The program has no loops and no other variables, but rather consists of a sequence of conditional search and replace operations that modify `str`.
- The operations of a `STRINGS` program are:
  - `REPLACE(pattern1,pattern2)` where `pattern1` and `pattern2` are fixed strings. This replaces the first occurrence of `pattern1` in `str` with `pattern2`
  - `if search(pattern) { code }` executes `code` if `pattern` is a substring of `str`. The code `code` can itself include nested `if`'s. (One can also add an `else { ... }` to execute if `pattern` is *not* a substring of condf).
  - the returned value is `str`
- A `STRING` program Q computes a function $F : \{0,1\}^* \rightarrow \{0,1\}^*$ if for every $x \in \{0,1\}^*$, if we initialize `str` to x and then execute the sequence of instructions in Q, then at the end of the execution `str` equals $F(x)$.

For example, the following is a `STRINGS` program that computes the function $F : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, if x contains a substring of the form $y = 11ab11$ where $a, b \in \{0,1\}$, then $F(x) = x'$ where x' is obtained by replacing the first occurrence of y in x with 00.

```
if search('110011') {
    replace('110011','00')
} else if search('110111') {
    replace('110111','00')
} else if search('111011') {
    replace('111011','00')
} else if search('111111') {
    replace('1111111','00')
}
```

Prove that for every Turing Machine program M, there exists a `STRINGS` program Q that computes the $NEXT_M$ function that maps every string encoding a valid *configuration* of M to the string encoding the configuration of the next step of M's computation. (We don't care what the function will do on strings that do not encode a valid configuration.) You don't have to write the `STRINGS` program fully, but you do need to give a convincing argument that such a program exists.

■

7.10 BIBLIOGRAPHICAL NOTES

Chapters 7 in the wonderful book of Moore and Mertens [MM11] contains a great exposition much of this material. .

The RAM model can be very useful in studying the concrete complexity of practical algorithms. Its theoretical study was initiated in [CR73]. However, the exact set of operations that are allowed in the RAM model and their costs 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]. 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 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.

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.

Unlike most programming languages, the pure λ-calculus doesn't have the notion of *types*. Every object in the λ calculus can also be thought of as a λ 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 λ expression via "search and replace", replacing all instances of its parameter with copies of the input expres-

sion you fed it. Typed variants of the λ calculus are objects of intense research, and are strongly related to type systems for programming language and computer-verifiable proof systems, see [Pie02]. Some of the typed variants of the λ calculus do not have infinite loops, which makes them very useful as ways of enabling static analysis of programs as well as computer-verifiable proofs. We will come back to this point in Chapter 9 and Chapter 21.

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.