Learning Objectives:

- Learn about the Turing Machine and λ calculus, which are important models of computation.
- See the equivalence between these models and NAND++ programs.
- See how many other models turn out to be "Turing complete"
- Understand the Church-Turing thesis.

9

Equivalent models of computation

> *"Computing is normally done by writing certain symbols on paper. We may suppose that this paper is divided into squares like a child's arithmetic book.. The behavior of the [human] computer at any moment is determined by the symbols which he is observing, and of his 'state of mind' at that moment... We may suppose that in a simple operation not more than one symbol is altered.", "We compare a man in the process of computing ... to a machine which is only capable of a finite number of configurations... The machine is supplied with a 'tape' (the analogue of paper) ... divided into sections (called 'squares') each capable of bearing a 'symbol'"*, Alan Turing, 1936

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

We have defined the notion of computing a function based on the rather esoteric NAND++ programming language. In this lecture we justify this choice by showing that the definition of computable functions will remain the same under a wide variety of computational models. In fact, a widely believed claim known as the *Church-Turing Thesis* holds that *every* "reasonable" definition of computable function is equivalent to ours. We will discuss the Church-Turing Thesis and the potential definitions of "reasonable" at the end of this lecture.

9.1 *Turing machines*

The "granddaddy" of all models of computation is the *Turing Machine*, which is the standard model of computation in most textbooks.[1] Turing machines were defined in 1936 by Alan Turing in an attempt to formally capture all the functions that can be computed by human "computers" that follow a well-defined set of rules, such as the standard algorithms for addition or multiplication.

[1] This definitional choice does not make much difference since, as we will show, NAND++/NAND« programs are equivalent to Turing machines in their computing power.

Figure 9.1: Until the advent of electronic computers, the word "computer" was used to describe a person, often female, that performed calculations. These human computers were absolutely essential to many achievements including mapping the stars, breaking the Enigma cipher, and the NASA space mission. Two recent books about these "computers" and their important contributions are The Glass Universe (from which this photo is taken) and Hidden Figures.

Turing thought of such a person as having access to as much "scratch paper" as they need. For simplicity we can think of this scratch paper as a one dimensional piece of graph paper (commonly known as "work tape"), where in each box or "cell" of the tape holds a single symbol from some finite alphabet (e.g., one digit or letter). At any point in time, the person can read and write a single box of the paper, and based on the contents can update his/her finite mental state, and/or move to the box immediately to the left or right.

Thus, Turing modeled such a computation by a "machine" that maintains one of k states, and at each point can read and write a single symbol from some alphabet Σ (containing $\{0,1\}$) from its "work tape". To perform computation using this machine, we write

the input $x \in \{0,1\}^n$ on the tape, and the goal of the machine is to ensure that at the end of the computation, the value $F(x)$ will be written there. Specifically, a computation of a Turing Machine M with k states and alphabet Σ on input $x \in \{0,1\}^*$ proceeds as follows:

- Initially the machine is at state 0 (known as the "starting state") and the tape is initialized to $\triangleright, x_0, \ldots, x_{n-1}, \varnothing, \varnothing, \ldots$.[2]
- The location i to which the machine points to is set to 0.
- At each step, the machine reads the symbol $\sigma = T[i]$ that is in the i^{th} location of the tape, and based on this symbol and its state s decides on:
  - What symbol σ' to write on the tape
  - Whether to move **L**eft (i.e., $i \leftarrow i-1$) or **R**ight (i.e., $i \leftarrow i+1$)
  - What is going to be the new state $s \in [k]$
- When the machine reaches the state $s = k-1$ (known as the "halting state") then it halts. The output of the machine is obtained by reading off the tape from location 1 onwards, stopping at the first point where the symbol is not 0 or 1.

[2] We use the symbol $\triangleright$ to denote the beginning of the tape, and the symbol $\varnothing$ to denote an empty cell. Hence we will assume that Σ contains these symbols, along with 0 and 1.

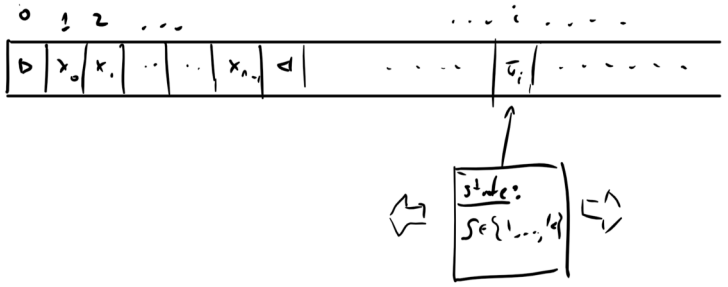

Figure 9.2: A Turing machine has access to a *tape* of unbounded length. At each point in the execution, the machine can read/write a single symbol of the tape, and based on that decide whether to move left, right or halt.

3

[3] TODO: update figure to $\{0,\ldots,k-1\}$.

The formal definition of Turing machines is as follows:

Definition 9.1 — Turing Machine. A (one tape) *Turing machine* with k states and alphabet $\Sigma \supseteq \{0,1,\triangleright,\varnothing\}$ is a function $M : [k] \times \Sigma \to [k] \times \Sigma \times \{L,R\}$. We say that the Turing machine M *computes* a (partial) function $F : \{0,1\}^* \to \{0,1\}^*$ if for every $x \in \{0,1\}^*$ on which F is defined, the result of the following process is $F(x)$:

- Initialize the array T of symbols in Σ as follows: $T[0] = \triangleright$,

$T[i] = x_i$ for $i = 1, \ldots, |x|$

- Let $s = 1$ and $i = 0$ and repeat the following while $s \neq k - 1$:

1. Let $\sigma = T[i]$. If $T[i]$ is not defined then let $\sigma = \varnothing$
2. Let $(s', \sigma', D) = M(s, \sigma)$
3. Modify $T[i]$ to equal σ'
4. If $D = L$ and $i > 0$ then set $i \leftarrow i - 1$. If $D = R$ then set $i \leftarrow i + 1$.
5. Set $s \leftarrow s'$

- Let n be the first index larger than 0 such that $T[i] \notin \{0, 1\}$. We define the output of the process to be $T[1], \ldots, T[n-1]$. The number of steps that the Turing Machine M takes on input x is the number of times that the while loop above is executed. (If the process never stops then we say that the machine did not halt on x.)

Turing Machine configurations Just as we did for NAND++ programs, we can also define the notion of *configurations* for Turing machines and define computation in terms of iterations of their *next step function*. However, we will not follow this approach in this course.

9.2 *Turing Machines and NAND++ programs*

As mentioned, Turing machines turn out to be equivalent to NAND++ programs:

Theorem 9.2 — Turing machines and NAND++ programs. For every $F : \{0,1\}^* \to \{0,1\}^*$, F is computable by a NAND++ program if and only if there is a Turing Machine M that computes F.

9.2.1 *Simulating Turing machines with NAND++ programs*

We now prove the "if" direction of Theorem 9.2, namely we show that given a Turing machine M, we can find a NAND++ program P_M such that for every input x, if M halts on input x with output y then $P_M(x) = y$. Because NAND« and NAND++ are equivalent in power, it is sufficient to construct a NAND« program that has

this property. Moreover, we can take advantage of the syntactic sugar transformations we have seen before for NAND«, including conditionals, loops, and function calls.

If $M : [k] \times \Sigma \to \Sigma \times [k] \times \{L, R\}$ then there is a finite length NAND program `ComputeM` that computes the finite function M (representing the finite sets $[k]$,Σ, $\{L, R\}$ appropriately by bits). The NAND« program simulating M will be the following:

```
// tape is an array with the alphabet Sigma
// we use ">" for the start-tape marker and "." for the
    empty cell
// in the syntactic sugar below, we assume some binary
    encoding of the alphabet.
// we also assume we can index an array with a variable
    other than i,
// and with some simple expressions of it (e.g., foo_{j+1})
// this can be easily simulated in NAND<<
//
// we assume an encoding such that the default value for
    tape_j is "."

// below k is the number of states of the machine M
// ComputeM is a function that maps a symbol in Sigma and a
    state in [k]
// to the new state, symbol to write, and direction

tape_0 := ">"
j = 0
while (validx_j) { // copy input to tape
   tape_{j+1} := x_j
   j++
}

state := 0
head := 0 // location of the head
while NOT EQUAL(state,k-1) { // not ending state
  state', symbol, dir := ComputeM(tape_head,state)
  tape_head := symbols
  if EQUAL(dir,`L`) AND NOT(EQUAL(tape_head,">")) {
     head--
  }
  if EQUAL(dir,'R') {
     head++
  }
```

```
  state' := state
}

// after halting, we copy the contents of the tape
// to the output variables
// we stop when we see a non-boolean symbol
j := 1
while EQUAL(tape_j,0) OR EQUAL(tape_j,1) {
  y_{j-1} := tape_j
}
```

In addition to the standard syntactic sugar, we are also assuming in the above code that we can make function calls to the function `EQUAL` that checks equality of two symbols as well as the finite function `ComputeM` that corresponds to the transition function of the Turing machine. Since these are *finite* functions (whose input and output length only depends on the number of states and symbols of the machine M, and not the input length), we can compute them using a NAND (and hence in particular a NAND++ or NAND«) program.

9.2.2 *Simulating NAND++ programs with Turing machines*

To prove the second direction of Theorem 9.2, we need to show that for every NAND++ program P, there is a Turing machine M that computes the same function as P. The idea behind the proof is that the TM M will *simulate* the program P as follows: (see also Fig. 9.3)

- The *head position* of M will correspond to the position of the index `i` in the current execution of the program P.
- The *alphabet* of M will be large enough so that in position i of the tape will store the contents of all the variables of P of the form `foo_`$\langle i \rangle$. (In particular this means that the alphabet of M will have at least 2^t symbols when t is the number of variables of P.)
- The *states* of M will be large enough to encode the current line number that is executed by P, as well as the contents of all variables that are indexed in the program by an absolute numerical index (e.g., variables of the form `foo` or `bar_17` that are not indexed with `i`.)

The key point is that the number of lines of P and the number of variables are constants that do not depend on the length of the input and so can be encoded in the alphabet and state size. More specifically, if V is the set of variables of P, then the alphabet of M

will contain (in addition to the symbols $\{0,1,\triangleright,\varnothing\}$) the finite set of all functions from V to $\{0,1\}$, with the semantics that if $\sigma : V \to \{0,1\}$ appears in the i-th position of the tape then for every variable $v \in V$, the value of $v\_\langle i \rangle$ equals $\sigma(v)$. Similarly, we will think of the state space of M as containing all pairs of the form (ℓ, τ) where ℓ is a number between 0 and the number of lines in P and τ is a function from $V \times [c]$ to $\{0,1\}$ where $c-1$ is the largest absolute numerical index that appears in the program.[4] The semantics are that ℓ encodes the line of P that is about to be executed, and $\tau(v,j)$ encodes the value of the variable $v\_\langle j \rangle$.

[4] While formally the state space of M is a number between 0 and $k-1$ for some $k \in \mathbb{N}$, by making k large enough, we can encode all pairs (ℓ, τ) of the form above as elements of the state space of the amchine, and so we can think of the state as having this form.

To simulate the execution of one step in P's computation, the machine M will do the following:

1. The contents of the symbol at the current head position and its state encode all information about the current line number to be executed, as well as the contents of all variables of the program of the form `foo_`$\langle i \rangle$ where i is the current value of the index variable in the simulated program. Hence we can use that to compute the new line to be executed.

2. M will write to the tape and update its state to reflect the update to the variable that is assigned a new value in the execution of this line.

3. If the current line was the last one in the program, and the `loop` variable (which is encoded in M's state) is equal to 1 then M will decide whether to move the head left or right based on whether the index is going to increase and decrease. We can ensure this is is a function of the current variables of P by adding to the program the variables `breadcrumbs_i`, `atstart_i` and `indexincreasing` and the appropriate code so that `indexincreasing` is set to 1 if and only if the index will increase in the next iteration.[5]

[5] While formally we did not allow the head of the Turing machine to stay in place, we can simulate this by adding a "dummy step" in which M's head moves right and goes into a special state that will move left in the next step.

The simulation will also contain an *initialization* and a *finalization* phases. In the *initialization phase*, M will scan the input and modify the tape so it contains the proper encoding of `x_i`'s `validx_i`'s variables as well as load into its state all references to these variables with absolute numerical indices. In the *finalization phase*, M will scan its tape to copy the contents of the `y_i`'s (i.e., output) variables to the beginning of the tape, and write a non $\{0,1\}$ value at their end. We leave verifying the (fairly straightforward) details of implementing these steps to the reader. Note that we can add a finite number of states to M or symbols to its alphabet to make this implementation easier. Writing down the full description of M from the above "pseu-

docode" is straightforward, even if somewhat painful, exercise, and hence this completes the proof of Theorem 9.2.

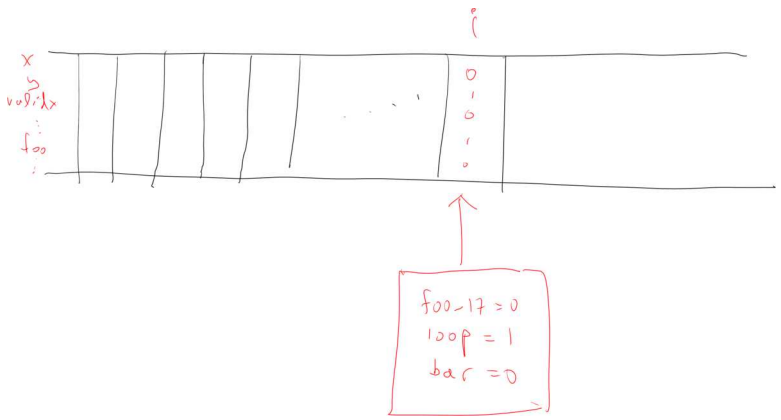

Figure 9.3: To simulate a NAND++ program P using a machine M we introduce a large alphabet Σ such that each symbol in Σ can be thought of as a "mega symbol" that encodes the value of all the variables indexed at i, where i is the current tape location. Similarly each state of M can be thought of as a "mega state" that encodes the value of all variables of P that have an absolute numerical index, as well as the current line that is about to be executed.

R **Polynomial equivalence** If we examine the proof of Theorem 9.2 then we can see that the equivalence between NAND++ programs and Turing machines is up to polynomial overhead in the number of steps. Specifically, our NAND« program to simulate a Turing Machine M, has two loops to copy the input and output and then one loop that iterates once per each step of the machine M. In particular this means that if the Turing machine M halts on every n-length input x within $T(n)$ steps, then the NAND« program computes the same function within $O(T(n) + n + m)$ steps. Moreover, the transformation of NAND« to NAND++ has polynomial overhead, and hence for any such machine M there is a NAND++ program P' that computes the same function within $poly(T(n) + n + m)$ steps (where $poly(f(n))$ is shorthand for $f(n)^{O(1)}$ as defined in the mathematical background section). In the other direction, our Turing machine M simulates each step of a NAND++ program P in a constant number of steps. The initialization and finalization phases involve scanning over $O(n + m)$ symbols and copying them around. Since the cost to move a symbol to a point that is of distance d in the tape can be $O(d)$ steps, the total cost of these phases will be $O((n + m)^2)$ Thus, for every NAND++ program P, if P halts on input x after T steps, then the corresponding Turing machine M will halt after $O(T + (n + m)^2)$ steps.

9.3 *"Turing Completeness" and other Computational models*

A computational model M is *Turing complete* if every partial function $F : \{0,1\}^* \to \{0,1\}^*$ that is computable by a Turing Machine is also computable in M. A model is *Turing equivalent* if the other direction holds as well; that is, for every partial function $F : \{0,1\}^* \to \{0,1\}^*$, F is computable by a Turing machine if and only if it is computable in M. Another way to state Theorem 9.2 is that NAND++ is Turing equivalent. Since we can simulate any NAND« program by a NAND++ program (and vice versa), NAND« is Turing equivalent as well. It turns out that there are many other Turing-equivalent models of computation.

We now discuss a few examples.

9.3.1 *RAM Machines*

The *Word RAM model* is a computational model that is arguably more similar to real-world computers than Turing machines or NAND++ programs. In this model the memory 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 as a number in $[2^w]$. The parameter w is known as the *word size* and is chosen as some function of the input length n. A typical choice is that $w = c \log n$ for some constant c. This is sometimes known as the "transdichotomous RAM model". In this model there are a constant number of *registers* $r_1, \ldots, r_k$ that also contain a single word. The operations in this model include loops, arithmetic on registers, and reading and writing from a memory location addressed by the register. See this lecture for a more precise definition of this model.

We will not show all the details but it is not hard to show that the word RAM model is equivalent to NAND« programs. Every NAND« program can be easily simulated by a RAM machine as long as w is larger than the logarithm of its running time. In the other direction, a NAND« program can simulate a RAM machine, using its variables as the registers.

9.3.2 *Imperative languages*

As we discussed before, any function computed by a standard programming language such as `C`, `Java`, `Python`, `Pascal`, `Fortran` etc. can be computed by a NAND++ program. Indeed, a *compiler* for such

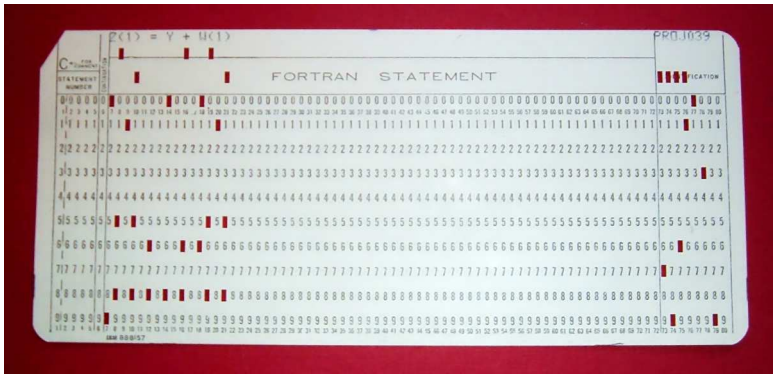

Figure 9.4: A punched card corresponding to a Fortran statement.

languages translates programs into machine languages that are quite similar to NAND« programs or RAM machines. We can also translate NAND++ programs to such programming languages. Thus all these programming languages are Turing equivalent.[6]

[6] Some programming language have hardwired fixed (even if extremely large) bounds on the amount of memory they can access. We ignore such issues in this discussion and assume access to some storage device without a fixed upper bound on its capacity.

9.4 *Lambda calculus and functional programming languages*

The λ calculus is another way to define computable functions. It was proposed by Alonzo Church in the 1930's around the same time as Alan Turing's proposal of the Turing Machine. Interestingly, while Turing Machines are not used for practical computation, the λ calculus has inspired functional programming languages such as LISP, ML and Haskell, and so indirectly, the development of many other programming languages as well.

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

$$square(x) = x \cdot x \tag{9.1}$$

we write it as

$$\lambda x.x \cdot x \tag{9.2}$$

Generally, an expression of the form

$$\lambda x.e \tag{9.3}$$

corresponds to the function that maps any expression z into the expression $e[x \to z]$ which is obtained by replacing every occurrence of x in e with z.[7]

Currying. The expression e can itself involve λ, and so for example the function

$$\lambda x.(\lambda y.x + y) \tag{9.4}$$

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

In particular, if we invoke this function on a and then invoke the result on b then we get $a + b$. We can use this approach to achieve the effect of functions with more than one input and so we will use the shorthand $\lambda x, y.e$ for $\lambda x.(\lambda y.e)$.[8]

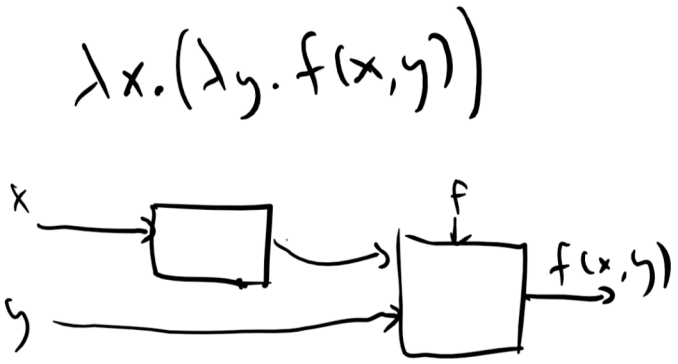

Figure 9.5: In the "currying" transformation, we can create the effect of a two parameter function $f(x, y)$ with the λ expression $\lambda x.(\lambda y.f(x, y))$ which on input x outputs a one-parameter function f_x that has x "hardwired" into it and such that $f_x(y) = f(x, y)$. This can be illustrated by a circuit diagram; see Chelsea Voss's site.

Precedence and parenthesis. The above is a complete description of the λ calculus. However, to avoid clutter, we will allow to drop parenthesis for function invocation, and so if f is a λ expression and z is some other expression, then we can write fz instead of $f(z)$ for the expression corresponding to invoking f on z.[9] That is, if f has the form $\lambda x.e$ then fz is the same as $f(z)$, which corresponds to the expression $e[x \to z]$ (i.e., the expression obtained by invoking f on z via replacing all copies of the x parameter with z).

We can still use parenthesis for grouping and so $f(gh)$ means invoking g on h and then invoking f on the result, while $(fg)h$ means invoking f on g and then considering the result as a function which then is invoked on h. We will associate from left to right and

[7] More accurately, we replace every expression of x that is *bound* by the λ operator. For example, if we have the λ expression $\lambda x.(\lambda x.x + 1)(x)$ and invoke it on the number 7 then we get $(\lambda x.x + 1)(7) = 8$ and not the nonsensical expression $(\lambda 7.7 + 1)(7)$. To avoid such annoyances, we can always ensure that every instance of $\lambda var.e$ uses a unique variable identifier var. See the "logical operators" section in the math background lecture for more discussion on bound variables.

[8] This technique of simulating multiple-input functions with single-input functions is known as Currying and is named after the logician Haskell Curry.

[9] When using identifiers with multiple letters for λ expressions, we'll separate them with spaces or commas.

so identify fgh with $(fg)h$. For example, if $f = \lambda x.(\lambda y.x + y)$ then $fzw = (fz)w = z + w$.

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

(Lack of) types. 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 expression you fed it.

9.4.1 The "basic" λ calculus objects

To calculate, it seems we need some basic objects such as 0 and 1, and so we will consider the following set of "basic" objects and operations:

- **Boolean constants:** 0 and 1. We also have the $IF(cond, a, b)$ functions that outputs a if $cond = 1$ and b otherwise. Using IF we can also compute logical operations such as $AND, OR, NOT, NAND$ etc.: can you see why?
- **The empty string:** The value NIL and the function $ISNIL(x)$ that returns 1 iff x is NIL.
- **Strings/lists:** The function $PAIR(x, y)$ that creates a pair from x and y. We will also have the function $HEAD$ and $TAIL$ to extract the first and second member of the pair. We can now create the list x, y, z by $PAIR(x, PAIR(y, PAIR(z, NIL)))$, see Fig. 9.6. A *string* is of course simply a list of bits.[10]
- **List operations:** The functions $MAP, REDUCE, FILTER$. Given a list $L = (x_0, \ldots, x_{n-1})$ and a function f, $MAP(L, f)$ applies f on every member of the list to obtain $L = (f(x_0), \ldots, f(x_{n-1}))$. The function $FILTER(L, f)$ returns the list of x_i's such that $f(x_i) = 1$,

[10] Note that if L is a list, then $HEAD(L)$ is its first element, but $TAIL(L)$ is not the last element but rather all the elements except the first. We use NIL to denote the empty list and hence $PAIR(x, NIL)$ denotes the list with the single element x.

and $REDUCE(L, f)$ "combines" the list by outputting

$$f(x_0, f(x_1, \cdots f(x_{n-3}, f(x_{n-2}, f(x_{n-1}, NIL)) \cdots) \tag{9.5}$$

For example $REDUCE(L, +)$ would output the sum of all the elements of the list L. See Fig. 9.7 for an illustration of these three operations.

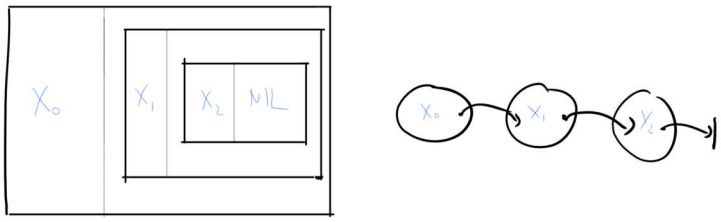

Figure 9.6: A list (x_0, x_1, x_2) in the λ calculus is constructed from the tail up, building the pair (x_2, NIL), then the pair $(x_1, (x_2, NIL))$ and finally the pair $(x_0, (x_1, (x_2, NIL)))$. That is, a list is a pair where the first element of the pair is the first element of the list and the 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.

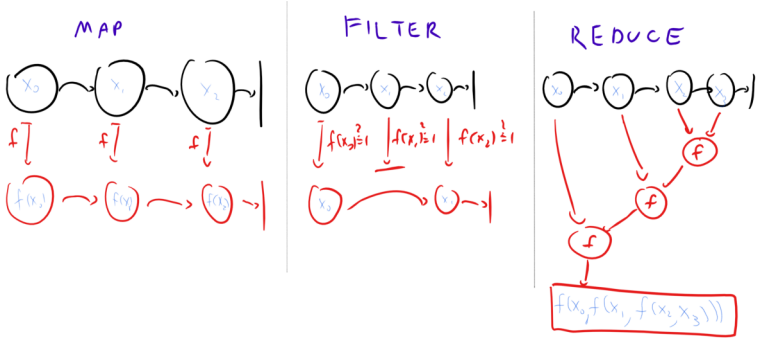

Figure 9.7: Illustration of the *MAP*, *FILTER* and *REDUCE* operations.

Together these operations more or less amount to the Lisp/Scheme programming language.[11]

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

Given that, it is perhaps not surprising that we can simulate NAND++ programs using the λ-calculus plus these basic elements, hence showing the following theorem:

Theorem 9.3 — Lambda calculus and NAND++. For every function $F : \{0,1\}^* \to \{0,1\}^*$, F is computable in the λ calculus with the above basic operations if and only if it is computable by a NAND++ program.

Proof. We only sketch the proof. The "only 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« as well, which we can then transform to a NAND++ program.

For the "if" direction, we start by showing that for every normal-form NAND++ program P, we can compute the next-step function $NEXT_P : \{0,1\}^* \to \{0,1\}^*$ using the above operations. It turns out not to be so hard. A configuration σ of P is a string of length TB where B is the (constant sized) block size, and so we can think of it as a list $\sigma = (\sigma^1, \ldots, \sigma^T)$ of T lists of bits, each of length B. There is some constant index $a \in [B]$ such that the i-th block is active if and only if $\sigma^i_a = 1$, and similarly there are also indices s, f that tell us if a block is the first or final block.

For every index c, we can extract from the configuration σ the B sized string corresponding to the block σ^i where $\sigma^i_c = 1$ using a single $REDUCE$ operation. Therefore, we can extract the first block σ^0, as well as the active block σ^i, and using similar ideas we can also extract a constant number of blocks that follow the first blocks $(\sigma^1, \sigma^2, \ldots, \sigma^{c'}$ where c' is the largest numerical index that appears in the program.)[12]

[12] It's also not hard to modify the program so that the largest numerical index is zero, without changing its functionality

Using the first blocks and active block, we can update the configuration, execute the corresponding line, and also tell if this is an operation where the index `i` stays the same, increases, or decreases. If it stays the same then we can compute $NEXT_P$ via a MAP operation, using the function that on input $C \in \{0,1\}^B$, keeps C the same if $C_a = 0$ (i.e., C is not active) and otherwise updates it to the value in its next step. If `i` increases, then we can update σ by a $REDUCE$ operation, with the function that on input a block C and a list S, we output $PAIR(C, S)$ unless $C_a = 1$ in which case we output $PAIR(C', PAIR(C'', TAIL(S)))$ where (C', C'') are the new values of the blocks i and $i+1$. The case for decreasing i is analogous.

Once we have a λ expression φ for computing $NEXT_P$, we can compute the final expression by defining

$$APPLY = \lambda f, \sigma . IF(HALT\sigma, \sigma, ff\varphi\sigma) \tag{9.6}$$

now for every configuration σ_0, $APPLY\ APPLY\sigma$ is the final configuration σ_t obtained after running the next-step function continuosly. Indeed, note that if σ_0 is not halting, then $APPLY\ APPLY\sigma_0$ outputs $APPLY\ APPLY\varphi\sigma_0$ which (since φ computes the $NEXT_P$) function

is the same as $APPLY\ \ APPLY\sigma_1$. By the same reasoning we see that we will eventually get $APPLY\ APPLY\sigma_t$ where σ_t is the halting configuration, but in this case we will get simply the output σ_t.[13] ■

[13] If this looks like recursion then this is not accidental- this is a special case of a general technique for simulating recursive functions in the λ calculus. See the discussion on the Y combinator below.

9.4.2 *How basic is "basic"?*

While the collection of "basic" functions we allowed for λ calculus is smaller than what's provided by most Lisp dialects, coming from NAND++ 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?

> P 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 try to continue and minimize things further, trying to implement $REDUCE$ with from $0, 1, IF, PAIR, HEAD, TAIL$ together with the λ operations. Remember that your functions can take functions as input and return functions as output.

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

We now outline how this can be done:

- We define 0 to be the function that on two inputs x, y outputs y, and 1 to be the function that on two inputs x, y outputs x. Of course we use Currying to achieve the effect of two inputs and hence $0 = \lambda x.\lambda y.y$ and $1 = \lambda x.\lambda y.x$.[14]
- 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 $IF\ cond\ a\ b = cond\ a\ b$.)
- To encode a pair (x, y) we will produce a function $f_{x,y}$ that has x and y "in its belly" and such that $f_{x,y}g = gxy$ for every function g. That is, we write $PAIR = \lambda x, y.\lambda g.gxy$. Note that now we can

[14] We could of course have flipped the definitions of 0 and 1, but we use the above because it is the common convention in the λ calculus, where people think of 0 and 1 as "false" and "true".

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 $ISNIL$ function checks, given an input p, whether we get 1 if we apply p to the function $0_{x,y}$ that ignores both its inputs and always outputs 0. For every valid pair $p0_{x,y} = 0$ while $NIL0_{x,y} = 1$. Formally, $ISNIL = \lambda p.p(\lambda x, y.0)$.

9.4.3 *List processing and recursion without recursion*

Now we come to the big hurdle, which is how to implement MAP, $FILTER$, and $REDUCE$ in the λ calculus. It turns out that we can build MAP and $FILTER$ from $REDUCE$. 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.) So, it all boils down to implementing $REDUCE$. 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 λ expression for $REDUCE$ as follows

$$REDUCE = \lambda L, g.IF(ISNIL(L), NIL, gHEAD(L)REDUCE(TAIL(L), g)) \; . \tag{9.7}$$

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. This seems like a very serious hurdle: if we don't have loops, and don't have recursion, how are we ever going to be able to compute a function like $REDUCE$?

The idea is to use the "self referential" properties of the λ calculus. Since we are able to work with λ expressions, we can possibly inside $REDUCE$ compute a λ expression that amounts to running $REDUCE$ itself. This is very much like the common exercise of a program that prints its own code. For example, suppose that you have some programming language with an `eval` operation that given a string `code` and an input `x`, evaluates `code` on `x`. Then, if you have a program P that can print its own code, you can use `eval` as an alternative to recursion: instead of using a recursive call on some input `x`, the program will compute its own code, store it in some string variable `str` and then use `eval(str,x)`. You might find this confusing. *I* definitely

find this confusing. But hopefully the following will make things a little more concrete.

[15]

[15] TODO: add a direct example how to implement *REDUCE* with *XOR* without using the *Y* combinator. Hopefully it can be done in a way that makes things more intuitive.

9.4.4 *The Y combinator*

The solution is to think of a recursion as a sort of "differential equation" on functions. For example, suppose that all our lists contain either 0 or 1 and consider $REDUCE(L, XOR)$ which simply computes the *parity* of the list elements. The ideas below will clearly generalize for implementing *REDUCE* with any other function, and in fact for implementing recursive functions in general. We can define the parity function *par* recursively as

$$par(x_0,\ldots,x_n) = \begin{cases} 0 & |x| = 0 \\ x_0 \oplus par(x_1,\ldots,x_n) & \text{otherwise} \end{cases} \tag{9.8}$$

where $\oplus$ denotes the XOR operator.

Our key insight would be to recast Eq. (9.8) not as a *definition* of the parity function but rather as an *equation* on it. That is, we can think of Eq. (9.8) as stating that

$$par = PAREQ(par) \tag{9.9}$$

where *PAREQ* is a *non-recursive* function that takes a function p as input, and returns the function q defined as

$$q(x_0,\ldots,x_n) = \begin{cases} 0 & |x| = 0 \\ x_0 \oplus p(x_1,\ldots,x_n) & \text{otherwise} \end{cases} \tag{9.10}$$

In fact, it's not hard to see that satisfying Eq. (9.9) is *equivalent* to satisfying Eq. (9.8), and hence *par* is the *unique* function that satisfies the condition Eq. (9.9). This means that to find a function *par* computing parity, all we need is a "magical function" *SOLVE* that given a function *PAREQ* finds "fixed point" of *PAREQ*: a function p such that $PAREQ(p) = p$. Given such a "magical function", we could give a non-recursive definition for *par* by writing $par = SOLVE(PAREQ)$.

It turns out that we *can* find such a "magical function" *SOLVE* in the λ calculus, and this is known as the Y combinator.

Theorem 9.4 — Y combinator. Let

$$Y = \lambda f.(\lambda x.f(xx))(\lambda y.f(yy)) \tag{9.11}$$

then for every λ expression F, if we let $h = YF$ then $h = Fh$.

Proof. Indeed, for every λ expression F of the form $\lambda t.e$, we can see that

$$YF = (\lambda x.F(xx))(\lambda y.F(yy)) \tag{9.12}$$

But this is the same as applying F to gg where $g = \lambda y.F(y,y)$, or in other words

$$YF = F\left((\lambda y.F(y,y))(\lambda y.F(y,y))\right) \tag{9.13}$$

but by a change of variables the RHS is the same as $F(YF)$. ■

Using the Y combinator we can implement recursion in the λ-calculus, and hence loops. This can be used to complete the "if" direction of Theorem 9.3.

For example, to compute parity we first give a recursive definition of parity using the λ-calculus as

$$parL = IF(ISNIL(L), 0, XORHEAD(L)par(TAIL(L))) \tag{9.14}$$

We then avoid the recursion by converting Eq. (9.14) to the operator $PAREQ$ defined as

$$PAREQ = \lambda p.\lambda L.IF(ISNIL(L), 0, XORHEAD(L)p(TAIL(L))) \tag{9.15}$$

and then we can define par as $YPAREQ$ since this will be the unique solution to $p = PAREQp$.

Infinite loops in λ-expressions. The fact that λ-expressions can simulate NAND++ programs means that, like them, it can also enter into an infinite loop. For example, consider the λ expression

$$(\lambda x.xxx)(\lambda x.xxx) \tag{9.16}$$

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

$$(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \tag{9.17}$$

To evaluate this, the next step would be to apply the second term on the third term,[16] which would result in

$$(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \tag{9.18}$$

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

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

9.5 Other models

There is a great variety of models that are computationally equivalent to Turing machines (and hence to NAND++/NAND« program). Chapter 8 of the book The Nature of Computation is a wonderful resource for some of those models. We briefly mention a few examples.

9.5.1 Parallel algorithms and cloud computing

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

9.5.2 Game of life, tiling and cellular automata

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

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

We can think of such a system as encoding a computation by starting it in some initial configuration, and then defining some halting condition (e.g., we halt if the cell at position $(0,0)$ becomes dead) and some way to define an output (e.g., we output the state of the cell at position $(1,1)$). Clearly, given any starting configuration x, we can simulate the game of life starting from x using a NAND« (or NAND++) program, and hence every "Game-of-Life computable" function is computable by a NAND« program. Surprisingly, it turns out that the other direction is true as well: as simple as its rules seem, we can simulate a NAND++ program using the game of life (see Fig. 9.8). The Wikipedia page for the Game of Life contains some beautiful figures and animations of configurations that produce some very interesting evolutions. See also the book The Nature of Computation. It turns out that even one dimensional cellular automata can be Turing complete, see Fig. 9.9.

9.6 *Our models vs standard texts*

We can summarize the models we use versus those used in other texts in the following table:

Model	These notes	Other texts
Nonuniform	NAND programs	Boolean circuits, straightline programs
Uniform (random access)	NAND« programs	RAM machines
Uniform (sequential access)	NAND++ programs	Oblivious one-tape Turing machines

\

Later on in this course we may study *memory bounded* computation. It turns out that NAND++ programs with a constant amount of memory are equivalent to the model of *finite automata* (the adjectives

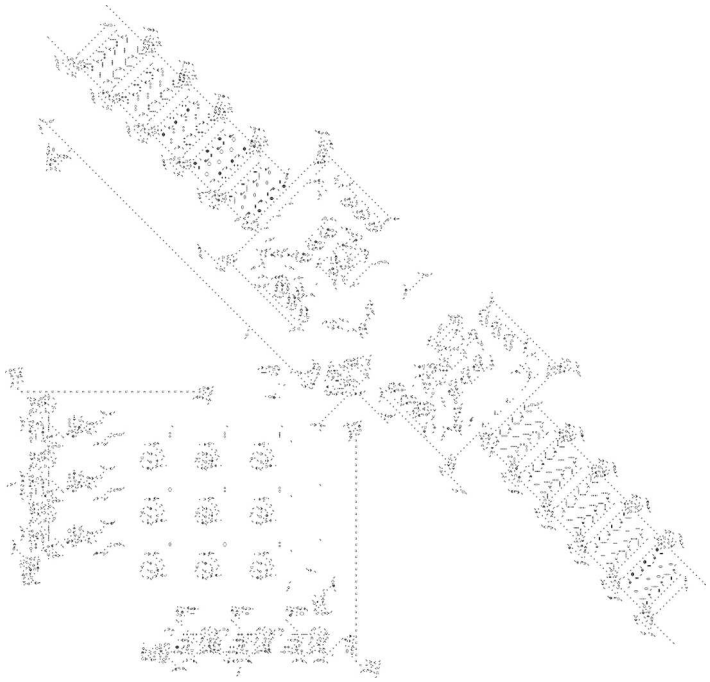

Figure 9.8: A Game-of-Life configuration simulating a Turing Machine. Figure by Paul Rendell.

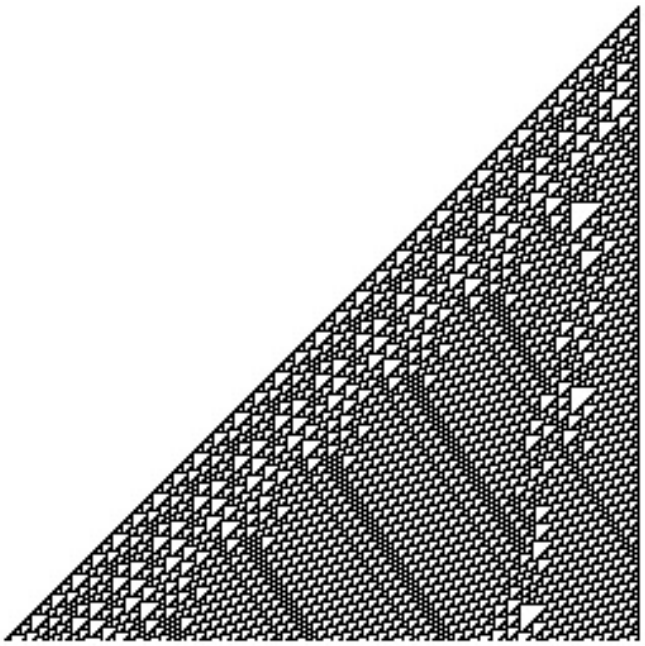

Figure 9.9: Evolution of a one dimensional automata. Each row in the figure corresponds to the configuration. The initial configuration corresponds to the top row and contains only a single "live" cell. This figure corresponds to the "Rule 110" automaton of Stefan Wolfram which is Turing Complete. Figure taken from Wolfram MathWorld.

"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).

9.7 The Church-Turing Thesis

We have defined functions to be *computable* if they can be computed by a NAND++ program, and we've seen that the definition would remain the same if we replaced NAND++ 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.[17]

[17] The *extended* Church Turing thesis, which we'll discuss later in this course, is that NAND++ programs even capture the limit of what can be *efficiently* computable. Just like the PECTT, quantum computing presents the main challenge to this thesis.

9.8 Lecture summary

- While we defined computable functions using NAND++ programs, we could just as well have done so using many other models, including not just NAND« but also Turing machines, RAM machines, the λ-calculus and many other models.
- Very simple models turn out to be "Turing complete" in the sense that they can simulate arbitrarily complex computation.

9.9 Exercises

[18]

Exercise 9.1 — lambda calculus requires three variables. Prove that for every λ-expression e with no free variables there is an equivalent λ-expression f using only the variables x, y, z.[19] ■

9.10 Bibliographical notes

[20]

9.11 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

- 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

9.12 Acknowledgements

[18] TODO: Add an exercise showing that NAND++ programs where the integers are represented using the *unary* basis are equivalent up to polylog terms with multi-tape Turing machines.

[19] **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.

[20] TODO: Recommend Chapter 7 in the nature of computation