Learning Objectives:

- Understand one of the most important concepts in computing: duality between code and data.
- Build up comfort in moving between different representations of programs.
- Follow the construction of a "universal NAND program" that can evaluate other NAND programs given their representation.
- See and understand the proof of a major result that compliments the result last lecture: some functions require an *exponential* number of NAND lines to compute.

5

Code as data, data as code

> *"The term code script is, of course, too narrow. The chromosomal structures are at the same time instrumental in bringing about the development they foreshadow. They are law-code and executive power - or, to use another simile, they are architect's plan and builder's craft - in one."* , Erwin Schrödinger, 1944.

> *"The importance of the universal machine is clear. We do not need to have an infinity of different machines doing different jobs. ... The engineering problem of producing various machines for various jobs is replaced by the office work of 'programming' the universal machine"*, Alan Turing, 1948

A NAND program can be thought of as simply a sequence of symbols, each of which can be encoded with zeros and ones using (for example) the ASCII standard. Thus we can represent every NAND program as a binary string. This statement seems obvious but it is actually quite profound. It means that we can treat a NAND program both as instructions to carrying computation and also as *data* that could potentially be input to other computations.

This correspondence between *code* and *data* is one of the most fundamental aspects of computing. It underlies the notion of *general purpose* computers, that are not pre-wired to compute only one task, and it is also the basis of our hope for obtaining *general* artificial intelligence. This concept finds immense use in all areas of computing, from scripting languages to machine learning, but it is fair to say that we haven't yet fully mastered it. Indeed many security exploits involve cases such as "buffer overflows" when attackers manage to

inject code where the system expected only "passive" data. The idea of code as data reaches beyond the realm of electronic computers. For example, DNA can be thought of as both a program and data (in the words of Schrödinger, who wrote before DNA's discovery a book that inspired Watson and Crick, it is both "architect's plan and builder's craft").

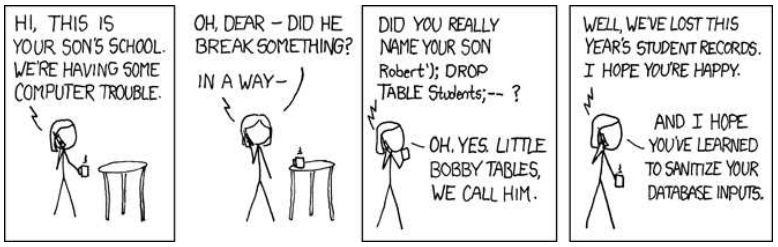


Figure 5.1: As illustrated in this xkcd cartoon, many exploits, including buffer overflow, SQL injections, and more, utilize the blurry line between "active programs" and "static strings".

5.1 A NAND interpreter in NAND

One of the most interesting consequences of the fact that we can represent programs as strings is the following theorem:

Theorem 5.1 — Bounded Universality of NAND programs. For every $S, n, m \in \mathbb{N}$ there is a NAND program that computes the function

$$EVAL_{S,n,m} : \{0,1\}^{S+n} \to \{0,1\}^m \tag{5.1}$$

defined as follows: For every string (P, x) where $P \in \{0,1\}^S$ and $x \in \{0,1\}^n$, if P describes a NAND program with n input bits and m outputs bits, then $EVAL_{S,n,m}(P, x)$ is the output of this program on input x. [1]

[1] If P does not describe a program then we don't care what $EVAL_{S,n,m}(P, x)$ is, but for concreteness we will set it to be 0^m. Note that in this theorem we use S to denote the number of bits describing the program, rather than the number of lines in it. However, these two quantities are very closely related.

Of course to fully specify $EVAL_{S,n,m}$, we need to fix a precise representation scheme for NAND programs as binary strings. We can simply use the ASCII representation, though we will use a more convenient representation. But regardless of the choice of representation, Theorem 5.1 is an immediate corollary of the fact that *every* finite function, and so in particular the function $EVAL_{S,n,m}$ above, can be computed by *some* NAND program.

Theorem 5.1 can be thought of as providing a "NAND interpreter in NAND". That is, for a particular size bound, we give a *single* NAND program that can evaluate all NAND programs of that size. We call this NAND program U that computes $EVAL_{S,n,m}$ a *bounded*

universal program. "Universal" stands for the fact that this is a *single program* that can evaluate *arbitrary* code, where "bounded" stands for the fact that U only evaluates programs of bounded size. Of course this limitation is inherent for the NAND programming language where an N-line program can never compute a function with more than N inputs. (We will later on introduce the concept of *loops*, that allows to escape this limitation.)

It turns out that we don't even need to pay that much of an overhead for universality

Theorem 5.2 — Efficient bounded universality of NAND programs. For every $S, n, m \in \mathbb{N}$ there is a NAND program of at most $O(S \log S)$ lines that computes the function $EVAL_{S,n,m} : \{0,1\}^{S+n} \to \{0,1\}^m$ defined above.

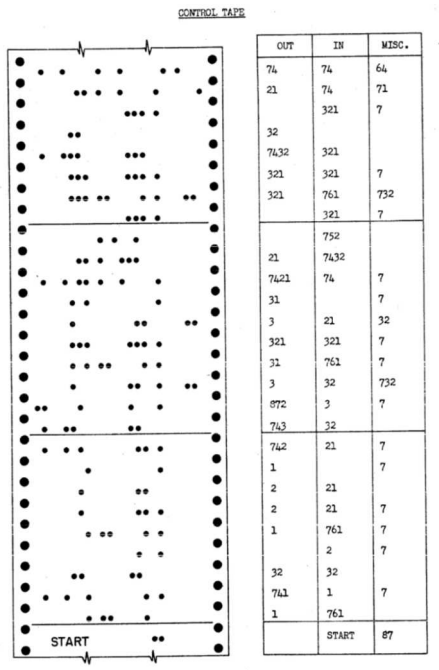
We will prove a weaker version of Theorem 5.2, that will use a large number of $O(S^2)$ lines instead of $O(S \log S)$ as stated in the theorem. We will sketch how we can improve this proof and get the $O(S \log S)$ bound in a future lecture. Unlike Theorem 5.1, Theorem 5.2 is not a trivial corollary of the fact that every function can be computed, and takes much more effort to prove. It requires us to present a concrete NAND program for the $EVAL_{S,n,m}$ function. We will do so in several stages.

First, we will spell out precisely how to represent NAND programs as strings. We can prove Theorem 5.2 using the ASCII representation, but a "cleaner" representation will be more convenient for us. Then, we will show how we can write a program to compute $EVAL_{S,n,m}$ in *Python*.[2] Finally, we will show how we can transform this Python program into a NAND program.

[2] We will not use much about Python, and a reader that has familiarity with programming in any language should be able to follow along.

5.2 *Concrete representation for NAND programs*

We can use the *canonical form* of NAND program (as per Definition 3.6) to represent it as a string. That is, if a NAND program has s lines and t distinct variables (where $t \leq 3s$) then we encode every a line of the program such as `foo_54 := baz NAND blah_22` as the triple (a, b, c) where a, b, c are the numbers corresponding to `foo_54`,`bar`,`blah_22` respectively. We choose the ordering such that the numbers $0, 1, \ldots, n-1$ encode the variables `x_0`,...,`x_`$\langle n-1 \rangle$ and the numbers $t-m, \ldots, t-1$ encode the variables `y_0`,...,`y_`$\langle m-1 \rangle$. Thus the representation of a program P of n inputes and m outputs is simply the list of triples of P in its canonical form. For example, the



OUT	IN	MISC.
74	74	64
21	74	71
	321	7
32		
7432	321	
321	321	7
321	761	732
	321	7
	752	
21	7432	
7421	74	7
31		7
3	21	32
321	321	7
31	761	7
3	32	732
872	3	7
743	32	
742	21	7
1		7
2	21	
2	21	7
1	761	7
	2	7
32	32	
741	1	7
1	761	
	START	87

Figure 5.2: In the Harvard Mark I computer, a program was represented as a list of triples of numbers, which were then encoded by perforating holes in a control card.

XOR program:

```
u_0 := x_0 NAND x_1
v_0 := x_0 NAND u_0
w_0 := x_1 NAND u_0
y_0 := v_0 NAND w_0
```

is represented by the following list of four triples:

```
[[2, 0, 1], [3, 0, 2], [4, 1, 2], [5, 3, 4]]
```

Note that even if we renamed `u`, `v` and `w` to `foo`, `bar` and `blah` then the representation of the program will remain the same (which is fine, since it does not change its semantics).

It is very easy to transform a string containing the program code to a the list-of-tuples representation; for example, it can be done in 15 lines of Python.[3]

[3] If you're curious what these 15 lines are, see the appendix or the website http://nandpl.org.

To evaluate a NAND program P given in this representation, on an input x, we will do the following:

- We create an array `avars` of t integers. The value of the variable with label j will be stored in the j-th location of this array.
- We initialize the value of the input variables. We set i to be the index corresponding to the label `x_`$\langle i \rangle$, and so set the i-th coordinate

of `avars` to x_i for every $i \in [n]$.

- For every triple (a,b,c) in the program's representation, we read from `avars` the values x, y of the variables b and c respectively, and then set the value of the variable indexed by a to $NAND(x,y) = 1 - x \cdot y$. That is, we set `avars[a] = 1-avars[b]*avars[c]`.
- The variables `y_0` till `y_`$\langle m-1 \rangle$ are given the indices $t-m, \ldots, t-1$ and so the output is `avars[t-m],...,avars[t-1]`.

The following is a *Python* function `EVAL` that on input L, n, m, x where L is a list of triples representing an n-input m-output program, and x is list of $0/1$ values, returns the result of the execution of the NAND program represented by P on x:[4]

```
# Evaluates an n-input, m-output NAND program L on input x
# L is given in the canonical list of triples representation
# (first n variables are inputs and last m variables are
    outputs)
def EVAL(L,n,m,x):
   s = len(L)
   avars = [0]*(3*s) # initialize variable array to zeroes,
       3s is large enough to hold all variables
   avars[:n] = x # set first n vars to x

   for (a,b,c) in L: # evaluate each triple
      u = avars[b]
      v = avars[c]
      val = 1-u*v # i.e., the NAND of u and v
      avars[a] = val

   t = max([max(triple) for triple in L])+1 # num of vars in
        L

   return avars[t-m:] # output last m variables
```

For example, if we run

```
EVAL(
[[2, 0, 1], [3, 0, 2], [4, 1, 2], [5, 3, 4]],
2,
1,
[0,1]
)
```

then this corresponds to running our XOR program on the input $(0,1)$ and hence the resulting output is `[1]`.

[4] To keep things simple, we will not worry about the case that L does not represent a valid program of n inputs and m outputs. Also, there is nothing special about Python. We could have easily presented a corresponding function in JavaScript, C, OCaml, or any other programming language.

Accessing an element of the array `avars` at a given index takes a constant number of basic operations.[5] Hence (since $n, m \leq s$ and $t \leq 3s$), the program above will use $O(s)$ basic operations.

[5] Python does not distinguish between lists and arrays, but allows constant time random access to an indexed elements to both of them. One could argue that if we allowed programs of truly unbounded length (e.g., larger than 2^{64}) then the price would not be constant but logarithmic in the length of the array/lists, but the difference between $O(1)$ and $O(\log s)$ will not be important for our discussions.

5.3 A NAND interpreter in NAND

We now turn to actually proving Theorem 5.2. To do this, it is of course not enough to give a Python program. We need to **(a)** give a precise representation of programs as binary strings, and **(b)** show how we compute the $EVAL_{S,n,m}$ function on this representation by a NAND program.

First, if a NAND program has s lines, then since it can have at most $3s$ distinct variables, it can be represented by a string of size $S = 3s\lambda$ where $\lambda = \lceil \log(3s) \rceil$, by simply concatenating the binary representations of all the $3s$ numbers (adding leading zeroes as needed to make each number represented by a string of exactly λ bits). So, our job is to transform, for every s, n, m, the Python code above to a NAND program $U_{s,n,m}$ that computes the function $EVAL_{S,n,m}$ for $S = 3s\lambda$. That is, given any representation $r \in \{0,1\}^S$ of an s-line n-input m-output NAND program P, and string $w \in \{0,1\}^n$, $U_{s,n,m}(rw)$ outputs $P(w)$.

> P Before reading further, try to think how *you* could give a "constructive proof" of Theorem 5.2. That is, think of how you would write, in the programming language of your choice, a function `universal(s,n,m)` that on input s, n, m outputs the code for the NAND program $U_{s,n,m}$ such that $U_{s,n,m}$ computes $EVAL_{S,n,m}$. Note that there is a subtle but crucial difference between this function and the Python `EVAL` program described above. Rather than actually evaluating a given program P on some input w, the function `universal` should output the *code* of a NAND program that computes the map $(P, w) \mapsto P(w)$.

Let $n, m, s \in \mathbb{N}$ be some numbers satisfying $s \geq n$ and $s \geq m$. We now describe the NAND program $U_{n,m,s}$ that computes $EVAL_{S,n,m}$ for $S = 3s\lambda$ and $\lambda = \lceil \log(3s) \rceil$. Our construction will follow very closely the Python implementation of `EVAL` above:[6]

1. $U_{s,n,m}$ will contain variables `avars_0`,...,`avars_`$\langle 2^\lambda - 1 \rangle$. (This corresponds to the line `avars = [0]*t` in the Python function `EVAL`.)

[6] We allow ourselves use of syntactic sugar in describing the program. We can always "unsweeten" the program later.

2. For $i = 0, \ldots, n-1$, we add the line `avars_`$\langle i \rangle$ `:= x_`$\langle 3s\lambda + i \rangle$ to $U_{s,n,m}$. Recall that the input to $EVAL_{S,n,m}$ is a string $rw \in \{0,1\}^{3s\lambda+n}$ where $r \in \{0,1\}^{3s\lambda}$ is the representation of the program P and $w \in \{0,1\}^n$ is the input that the program should be applied on. Hence this step copies the input to the variables `avars_0`,...,`avars_`$\langle n-1 \rangle$. (This corresponds to the line `avars[:n] = x` in `EVAL`.)

3. For $\ell = 0, \ldots, s-1$ we add the following code to $U_{s,n,m}$:

   (a) For all $j \in [\lambda]$, add the code `a_`$\langle j \rangle$ `:= x_`$\langle 3\ell\lambda + j \rangle$, `b_`$\langle j \rangle$ `:= x_`$\langle 3\ell\lambda + \lambda + j \rangle$ and `c_`$\langle j \rangle$ `:= x_`$\langle 3\ell\lambda + 2\lambda + j \rangle$. In other words, we add the code to copy to `a`, `b`, `c` the three λ-bit long strings containing the binary representation the ℓ-th triple (a, b, c) in the input program. (This corresponds to the line `for (a,b,c) in L:` in `EVAL`.)

   (b) Add the code `u := LOOKUP(avars_0 ,...,avars_`$\langle 2^\lambda - 1 \rangle$`,b_0,...,b_`$\langle \lambda - 1 \rangle$`)` and `v := LOOKUP(avars_0 ,...,avars_`$\langle 2^\lambda - 1 \rangle$`,c_0,...,c_`$\langle \lambda - 1 \rangle$`)` where `LOOKUP` is the macro that computes $LOOKUP_\lambda : \{0,1\}^{2^\lambda+\lambda} \to \{0,1\}$. Recall that we defined $LOOKUP_\lambda(A, i) = A_i$ for every $A \in \{0,1\}^{2^\lambda}$ and $i \in \{0,1\}^\lambda$ (using the binary representation to identify i with an index in $[2^\lambda]$). Hence this code means that `u` gets the value of `avars_`$\langle b \rangle$ and `v` gets the value of `avars_`$\langle c \rangle$. (This corresponds to the lines `u = avars[b]` and `v = avars[c]` in `EVAL`.)

   (c) Add the code `val := u NAND v` (i.e., `w` gets the value that should be stored in `avars_`$\langle a \rangle$). (This corresponds to the line `val = 1-u*v` in `EVAL`.)

   (d) Add the code `newvars_0`,...,`newvars_`$\langle 2^\lambda - 1 \rangle$ `:= UPDATE(avars_0`,...,`avars_`$\langle 2^\lambda - 1 \rangle$`, a_0`,...,`a_`$\langle \lambda - 1 \rangle$`,val“)`, where `UPDATE` is a macro that computes the function $UPDATE_\lambda : \{0,1\}^{2^\lambda+\lambda+1} \to \{0,1\}^{2^\lambda}$ defined as follows: for every $A \in \{0,1\}^{2^\lambda}$, $i \in \{0,1\}^\lambda$ and $v \in \{0,1\}$, $UPDATE_\lambda(A, i, v) = A'$ such that $A'_j = A_j$ for all $j \neq i$ and $A'_i = v$ (identifying i with an index in $[2^\lambda]$). See below for discussions on how to implement `UPDATE` and other macros.

   (e) Add the code `avars_`$\langle j \rangle$ `:= newvars_`$\langle j \rangle$ for every $j \in [2^\lambda]$ (i.e., update `avars` to `newvars`). (Steps 3.c and 3.d together correspond to the line `avars[a] = val` in `EVAL`.)

   After adding all the s snippets above in Step 3, we add to the program the code `t_0`,...,`t_`$\langle \lambda - 1 \rangle$ `:= INC(MAX(avars_0`,...,`avars_`2^λ`))`

where `MAX` is a macro that computes the function $MAX_{2^\lambda,\lambda}$ and we define $MAX_{s,\lambda} : \{0,1\}^{s\lambda} \rightarrow \{0,1\}^\lambda$ to take the concatenation of the representation of s numbers in $[2^\lambda]$ and output the representation of the maximum number, and `INC` is a macro that computes the function INC_λ that increments a given number in $[2^\lambda]$ by one. (This corresponds to the line `t = max([max(triple) for triple in L])+1` in `EVAL`.) We leave coming up with NAND programs for computing $MAX_{s,\lambda}$ and INC_λ as an exercise for the reader.

5. Finally we add for every $j \in [m]$:

   (a) The code `idx_0,...,idx_`$\langle \lambda - 1 \rangle$ `:= SUBTRACT(t_0,...,t_`$\langle \lambda \rangle$`,`z_0`,...,`$z_{\lambda-1}$`)` where `SUBTRACT` is the code for subtracting two numbers in $[2^\lambda]$ given in their binary representation, and each z_h is equal to either `zero` or `one` depending on the h-th digit in the binary representation of the number $m - j$.

   (b) `y_`$\langle j \rangle$ `:= LOOKUP( avars_0,..., avars_`$\langle 2^\lambda - 1 \rangle$`, idx_0,..., idx_`$\langle \lambda - 1 \rangle$ `)`. (Steps 5.a and 5.b together correspond to the line `return avars[t-m:]` in `EVAL`.)

To complete the description of this program, we need to show that we can implement the macros for `LOOKUP`,`UPDATE`,`MAX`,`INC` and `SUBTRACT`:

- We have already seen the implementation of `LOOKUP`
- We leave the implementation of the arithmetic macros `MAX`, `INC`, and `SUBTRACT` as exercises for the reader. All of those can be done using a number of lines that is linear in the size of their input. That is $MAX_{s,\lambda}$ can be computed in $O(s\lambda)$ lines, and INC_λ and $SUBTRACT_\lambda$ can be computed in $O(\lambda)$ lines.
- For implementing the function `UPDATE`, note that for every indices i, $UPDATE_\lambda(A,i,v)_j = A_j$ unless $j = i$ in which case $UPDATE_\lambda(A,i,v)_i = v$. Since we can use the syntactic sugar for `if` statements, computing $UPDATE$ boils down to the function $EQUAL_\lambda : \{0,1\}^{2\lambda} \rightarrow \{0,1\}$ such that $EQUAL_\lambda(i_0,\ldots,i_{\lambda-1},j_0,\ldots,j_{\lambda-1}) = 1$ if and only if $i_k = j_k$ for every $k \in [\lambda]$. $EQUAL_\lambda$ is equivalent to the AND of λ invocations of the function $EQUAL_1 : \{0,1\}^2 \rightarrow \{0,1\}$ that checks if two bits are equal. Since each $EQUAL_1$ (as a function on two inputs) can be computed in a constant number of lines, we can compute $EQUAL_\lambda$ using $O(\lambda)$ lines.

The total number of lines in $U_{s,n,m}$ is dominated by the cost of step

3 above,[7] where we repeat s times the following:

1. Copying the ℓ-th triple to the variables `a`,`b`,`c`. Cost: $O(\lambda)$ lines.
2. Perform `LOOKUP` on a $2^\lambda = O(s)$ variables `avars_0`,..., `avars_`$\langle 2^\lambda - 1 \rangle$. Cost: $O(2^\lambda) = O(s)$ lines.
3. Perform the `UPDATE` to update the 2^λ variables `avar_0`,..., `avars_`$\langle 2^\lambda - 1 \rangle$ to `newvars_0`,..., `newvars_`$\langle 2^\lambda - 1 \rangle$. Since `UPDATE` makes $O(2^\lambda)$ calls to $EQUAL_\lambda$, and each such call costs $O(\lambda)$ lines, the total cost for $UPDATE$ is $O(2^\lambda \lambda) = O(s \log s)$ lines.
4. Copy `newvars_0`,..., `newvars_`$\langle 2^\lambda - 1 \rangle$ to `avar_0`,..., `avars_`$\langle 2^\lambda - 1 \rangle$. Cost: $O(2^\lambda)$ lines.

[7] It is a good exercise to verify that steps 1,2,4 and 5 above can be implemented in $O(s \log s)$ lines.

Since the loop of step 3 is repeated s times, the total number of lines in $U_{s,n,m}$ is $O(s^2 \log s)$ which (since $S = \Omega(s \log s)$) is $O(S^2)$.[8] The NAND program above is less efficient that its Python counterpart, since NAND does not offer arrays with efficient random access. Hence for example the `LOOKUP` operation on an array of s bits takes $\Omega(s)$ lines in NAND even though it takes $O(1)$ steps (or maybe $O(\log s)$ steps, depending how we count) in *Python*. We might see in a future lecture how to improve this to $O(s \log s)$.

[8] The website http://nandpl.org will (hopefully) eventually contain the implementation of the NAND program $U_{s,n,m}$ where you can also play with it by feeding it various other programs as inputs.

5.4 A Python interpreter in NAND

To prove Theorem 5.2 we essentially translated every line of the Python program for `EVAL` into an equivalent NAND snippet. It turns out that none of our reasoning was specific to the particular function $EVAL$. It is possible to translate *every* Python program into an equivalent `NAND` program of comparable efficiency.[9] Actually doing so requires taking care of many details and is beyond the scope of this course, but let me convince you why you should believe it is possible in principle. We can use CPython (the reference implementation for Python), to evaluate every Python program using a `C` program. We can combine this with a C compiler to transform a Python program to various flavors of "machine language".

[9] More concretely, if the Python program takes $T(n)$ operations on inputs of length at most n then we can find a NAND program of $O(T(n) \log T(n))$ lines that agrees with the Python program on inputs of length n.

So, to transform a Python program into an equivalent NAND program, it is enough to show how to transform a machine language program into an equivalent NAND program. One minimalistic (and hence convenient) family of machine languages is known as the *ARM architecture* which powers a great many mobile devices including essentially all Android devices.[10]

[10] ARM stands for "Advanced RISC Machine" where RISC in turn stands for "Reduced instruction set computer".

There are even simpler machine languages, such as the LEG

acrhitecture for which a backend for the LLVM compiler was implemented (and hence can be the target of compiling any of large and growing list of languages that this compiler supports). Other examples include the TinyRAM architecture (motivated by interactive proof systems that we will discuss much later in this course) and the teaching-oriented Ridiculously Simple Computer architecture.[11]

[11] The reverse direction of compiling NAND to C code, is much easier. We show code for a `NAND2C` function in the appendix.

Going one by one over the instruction sets of such computers and translating them to NAND snippets is no fun, but it is a feasible thing to do. In fact, ultimately this is very similar to the transformation that takes place in converting our high level code to actual silicon gates that (as we will see in the next lecture) are not so different from the operations of a NAND program.

Indeed, tools such as MyHDL that transform "Python to Silicon" can be used to convert a Python program to a NAND program.

The NAND programming language is just a teaching tool, and by no means do I suggest that writing NAND programs, or compilers to NAND, is a practical, useful, or even enjoyable activity. What I do want is to make sure you understand why it *can* be done, and to have the confidence that if your life (or at least your grade in this course) depended on it, then you would be able to do this. Understanding how programs in high level languages such as Python are eventually transformed into concrete low-level representation such as NAND is fundamental to computer science.

The astute reader might notice that the above paragraphs only outlined why it should be possible to find for every *particular* Python-computable function F, a *particular* comparably efficient NAND program P that computes F. But this still seems to fall short of our goal of writing a "Python interpreter in NAND" which would mean that for every parameter n, we come up with a *single* NAND program $UNIV_n$ such that given a description of a Python program P, a particular input x, and a bound T on the number of operations (where the length of P, x and the magnitude of T are all at most n) would return the result of executing P on x for at most T steps. After all, the transformation above would transform every Python program into a different NAND program, but would not yield "one NAND program to rule them all" that can evaluate every Python program up to some given complexity. However, it turns out that it is enough to show such a transformation for a single Python program. The reason is that we can write a Python interpreter *in Python*: a Python program U that takes a bit string, interprets it as Python code, and then runs that code. Hence, we only need to show a NAND program U^* that computes the same function as the particular Python program U, and

this will give us a way to evaluate *all* Python programs.

What we are seeing time and again is the notion of *universality* or *self reference* of computation, which is the sense that all reasonably rich models of computation are expressive enough that they can "simulate themselves". The importance of this phenomena to both the theory and practice of computing, as well as far beyond it, including the foundations of mathematics and basic questions in science, cannot be overstated.

5.5 Counting programs, and lower bounds on the size of NAND programs

One of the consequences of our representation is the following:

Theorem 5.3 — Counting programs.

$$|Size(s)| \leq 2^{O(s \log s)}. \tag{5.2}$$

That is, there are at most $2^{O(s \log s)}$ functions computed by NAND programs of at most s lines.

Moreover, the implicit constant in the $O(\cdot)$ notation in Theorem 5.3 is at most 10.[12] The idea behind the proof is that we can represent every s line program by a binary string of $O(s \log s)$ bits. Therefore the number of functions computed by s-line programs cannot be larger than the number of such strings, which is $2^{O(s \log s)}$. In the actual proof, given below, we count the number of representations a little more carefully, talking directly about triples rather than binary strings, although the idea remains the same.

[12] By this we mean that for all sufficiently large s, $|Size(s)| \leq 2^{10 s \log s}$.

Proof. Every NAND program P with s lines has at most $3s$ variables. Hence, using our canonical representation, P can be represented by the numbers n, m of P's inputs and outputs, as well as by the list L of s triples of natural numbers, each of which is smaller or equal to $3s$.

If two programs compute distinct functions then they have distinct representations. So we will simply count the number of such representations: for every $s' \leq s$, the number of s'-long lists of triples of numbers in $[3s]$ is $(3s)^{3s'}$, which in particular is smaller than $(3s)^{3s}$. So, for every $s' \leq s$ and n, m, the total number of representations of s'-line programs with n inputs and m outputs is smaller than $(3s)^{3s}$.

Since a program of at most s lines has at most s inputs and outputs, the total number of representations of all programs of at most s

lines is smaller than

$$s \times s \times s \times (3s)^{3s} = (3s)^{3s+3} \tag{5.3}$$

(the factor $s \times s$ *timess* arises from taking all of the at most s options for the number of inputs n, all of the at most s options for the number of outputs m, and all of the at most s options for the number of lines s'). We claim that for s large enough, the righthand side of Eq. (5.3) (and hence the total number of representations of programs of at most s lines) is smaller than $2^{4s \log s}$. Indeed, we can write $3s = 2^{\log(3s)} = 2^{\log 3 + \log s} \leq 2^{2+\log s}$, and hence the righthand side of Eq. (5.3) is at most $\left(2^{2+\log s}\right)^{3s+3} = 2^{(2+\log s)(3s+3)} \leq 2^{4s \log s}$ for s large enough.

For every function $F \in Size(s)$ there is a program P of at most s lines that computes it, and we can map F to its representation as a tuple (n, m, L). If $F \neq F'$ then a program P that computes F must have an input on which it disagrees with any program P' that computes F', and hence in particular P and P' have distinct representations. Thus we see that the map of $Size(s)$ to its representation is one to one, and so in particular $|Size(s)|$ is at most the number of distinct representations which is it at most $2^{4s \log s}$. ■

R **Counting by ASCII representation** We can also establish Theorem 5.3 directly from the ASCII representation of the source code. Since an s-line NAND program has at most $3s$ distinct variables, we can change all the workspace variables of such a program to have the form `work_`$\langle i \rangle$ for i between 0 and $3s - 1$ without changing the function that it computes. This means that after removing comments and extra whitespaces, every line of such a program (which will the form `var := var' NAND var"` for variable identifiers which will be either `x_###`,`y_###` or `work_###` where `###` is some number smaller than $3s$) will require at most, say, $20 + 3\log_{10}(3s) \leq O(\log s)$ characters. Since each one of those characters can be encoded using seven bits in the ASCII representation, we see that the number of functions computed by s-line NAND programs is at most $2^{O(s \log s)}$.

A function mapping $\{0,1\}^2$ to $\{0,1\}$ can be identified with the table of its four values on the inputs $00, 01, 10, 11$; a function mapping $\{0,1\}^3$ to $\{0,1\}$ can be identified with the table of its eight values on the inputs $000, 001, 010, 011, 100, 101, 110, 111$. More generally, every function $F : \{0,1\}^n \to \{0,1\}$ can be identified with the table of its 2^n

values on the inputs $\{0,1\}^n$. Hence the number of functions mapping $\{0,1\}^n$ to $\{0,1\}$ is equal to the number of such tables which (since we can choose either 0 or 1 for every row) is exactly 2^{2^n}. Note that this is *double exponential* in n, and hence even for small values of n (e.g., $n = 10$) the number of functions from $\{0,1\}^n$ to $\{0,1\}$ is truly astronomical.[13] This has the following interesting corollary:

[13] "Astronomical" here is an understatement: there are much fewer than 2^{210} stars, or even particles, in the observable universe.

Theorem 5.4 — Counting argument lower bound. There is a function $F : \{0,1\}^n \to \{0,1\}$ such that the shortest NAND program to compute F requires $2^n/(100n)$ lines.

Proof. Suppose, towards the sake of contradiction, that every function $F : \{0,1\}^n \to \{0,1\}$ can be computed by a NAND program of at most $s = 2^n/(100n)$ lines. Then the by Theorem 5.3 the total number of such functions would be at most $2^{10s \log s} \leq 2^{10 \log s \cdot 2^n/(100n)}$. Since $\log s = n - \log(100n) \leq n$ this means that the total number of such functions would be at most $2^{2^n/10}$, contradicting the fact that there are 2^{2^n} of them. ■

We have seen before that *every* function mapping $\{0,1\}^n$ to $\{0,1\}$ can be computed by an $O(2^n/n)$ line program. We now see that this is tight in the sense that some functions do require such an astronomical number of lines to compute. In fact, as we explore in the exercises below, this is the case for *most* functions. Hence functions that can be computed in a small number of lines (such as addition, multiplication, finding short paths in graphs, or even the *EVAL* function) are the exception, rather than the rule.

R **Advanced note: more efficient representation** The list of triples is not the shortest representation for NAND programs. As we will see in the next lecture, every NAND program of s lines and n inputs can be represented by a directed graph of $s + n$ vertices, of which n have in-degree zero, and the s others have in-degree at most two. Using the adjacency list representation, such a graph can be represented using roughly $2s \log(s + n) \leq 2s(\log s + O(1))$ bits. Using this representation we can reduce the implicit constant in Theorem 5.3 arbitrarily close to 2.

5.6 *Lecture summary*

- We can think of programs both as describing a *process*, as well as simply a list of symbols that can be considered as *data* that can be

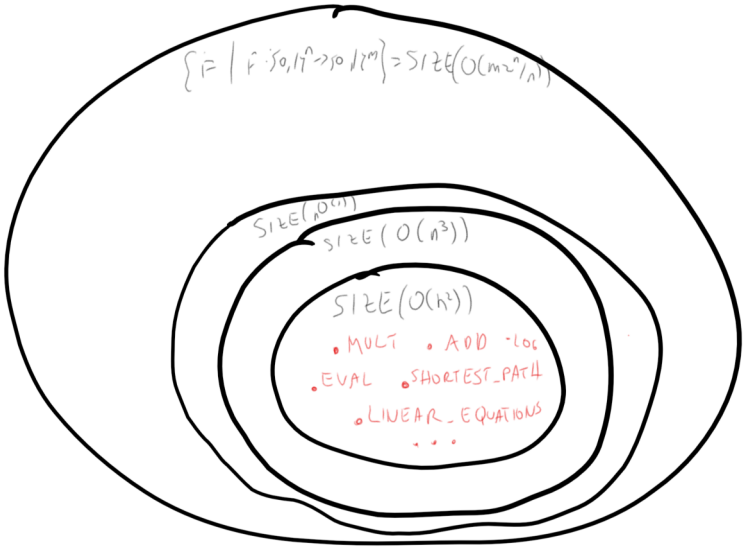


Figure 5.3: All functions mapping n bits to m bits can be computed by NAND programs of $O(m2^n/n)$ lines, but most functions cannot be computed using much smaller programs. However there are many important exceptions which are functions such as addition, multiplication, program evaluation, and many others, that can be computed in polynomial time with a small exponent.

fed as input to other programs.

- We can write a NAND program that evaluates arbitrary NAND programs. Moreover, the efficiency loss in doing so is not too large.
- We can even write a NAND program that evaluates programs in other programming languages such as Python, C, Lisp, Java, Go, etc.

5.7 *Exercises*

Exercise 5.1 Which one of the following statements is false:

a. There is an $O(s^3)$ line NAND program that given as input program P of s lines in the list-of-tuples representation computes the output of P when all its input are equal to 1.

b. There is an $O(s^3)$ line NAND program that given as input program P of s characters encoded as a string of $7s$ bits using the ASCII encoding, computes the output of P when all its input are equal to 1.

c. There is an $O(\sqrt{s})$ line NAND program that given as input program P of s lines in the list-of-tuples representation computes the

output of P when all its input are equal to 1. ■

Exercise 5.2 — Equals function. For every k, show that there is an $O(k)$ line NAND program that computes the function $EQUALS_k : \{0,1\}^{2k} \to \{0,1\}$ where $EQUALS(x,x') = 1$ if and only if $x = x'$. ■

Exercise 5.3 — Random functions are hard (challenge). Suppose $n > 1000$ and that we choose a function $F : \{0,1\}^n \to \{0,1\}$ at random, choosing for every $x \in \{0,1\}^n$ the value $F(x)$ to be the result of tossing an independent unbiased coin. Prove that the probability that there is a $2^n/(1000n)$ line program that computes F is at most 2^{-100}.[14] ■

Exercise 5.4 — Circuit hierarchy theorem (challenge). Prove that there is a constant c such that for every n, there is some function $F : \{0,1\}^n \to \{0,1\}$ s.t. **(1)** F *can* be computed by a NAND program of at most cn^5 lines, but **(2)** F can *not* be computed by a NAND program of at most n^4/c lines.[15] ■

[16]

5.8 Bibliographical notes

[17]

5.9 Further explorations

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

- Lower bounds. While we've seen the "most" functions mapping n bits to one bit require NAND programs of exponential size $\Omega(2^n/n)$, we actually do not know of any *explicit* function for which we can *prove* that it requires, say, at least n^{100} or even $100n$ size. At the moment, strongest such lower bound we know is that there are quite simple and explicit n-variable functions that require at least $(5-o(1))n$ lines to compute, see this paper of Iwama et al as well as this more recent work of Kulikov et al. Proving lower bounds for restricted models of straightline programs (more often described as *circuits*) is an extremely interesting research area, for which Jukna's book provides very good introduction and overview.

[14] **Hint:** An equivalent way to say this is that you need to prove that the set of functions that can be computed using at most $2^n/(1000n)$ has fewer than $2^{-100}2^{2^n}$ elements. Can you see why?

[15] **Hint:** Find an approriate value of t and a function $G : \{0,1\}^t \to \{0,1\}$ that can be computed in $O(2^t/t)$ lines but *can't* be computed in $\Omega(2^t/t)$ lines, and then extend this to a function mapping $\{0,1\}^n$ to $\{0,1\}$.

[16] TODO: add exercise to do evaluation of T line programs in $\tilde{O}(T^{1.5})$ time.

[17] TODO: *EVAL* is known as *Circuit Evaluation* typically. More references regarding oblivious RAM etc..

5.10 Acknowledgements