Learning Objectives:

- Get comfortable with syntactic sugar or automatic translation of higher level logic to low level gates.
- Learn proof of major result: every finite function can be computed by a Boolean circuit.
- Start thinking *quantitatively* about number of lines required for computation.

4

Syntactic sugar, and computing every function

"[In 1951] I had a running compiler and nobody would touch it because, they carefully told me, computers could only do arithmetic; they could not do programs.", Grace Murray Hopper, 1986.

"Syntactic sugar causes cancer of the semicolon.", Alan Perlis, 1982.

The computational models we considered thus far are as "bare bones" as they come. For example, our NAND-CIRC "programming language" has only the single operation `foo = NAND(bar,blah)`. In this chapter we will see that these simple models are actually *equivalent* to more sophisticated ones. The key observation is that we can implement more complex features using our basic building blocks, and then use these new features themselves as building blocks for even more sophisticated features. This is known as "syntactic sugar" in the field of programming language design since we are not modifying the underlying programming model itself, but rather we merely implement new features by syntactically transforming a program that uses such features into one that doesn't.

This chapter provides a "toolkit" that can be used to show that many functions can be computed by NAND-CIRC programs, and hence also by Boolean circuits. We will also use this toolkit to prove a fundamental theorem: *every* finite function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a Boolean circuit, see Theorem 4.13 below. While the syntactic sugar toolkit is important in its own right, Theorem 4.13 can also be proven directly without using this toolkit. We present this alternative proof in Section 4.5. See Fig. 4.1 for an outline of the results of this chapter.

This chapter: A non-mathy overview

In this chapter, we will see our first major result: *every* finite function can be computed by some Boolean circuit (see Theorem 4.13 and Big Idea 5). This is sometimes known as

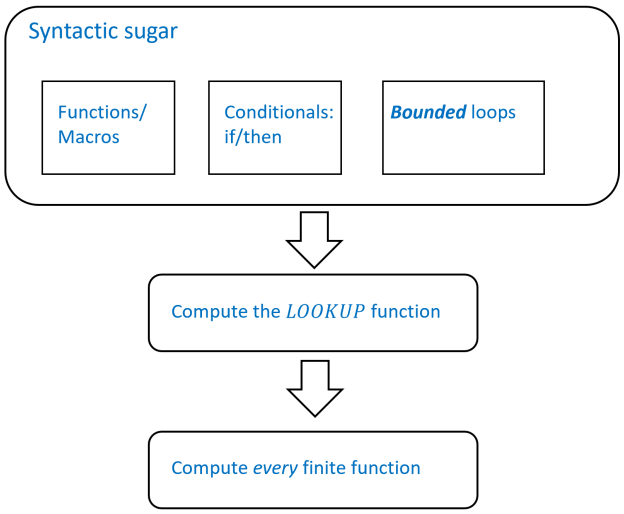

Figure 4.1: An outline of the results of this chapter. In Section 4.1 we give a toolkit of "syntactic sugar" transformations showing how to implement features such as programmer-defined functions and conditional statements in NAND-CIRC. We use these tools in Section 4.3 to give a NAND-CIRC program (or alternatively a Boolean circuit) to compute the *LOOKUP* function. We then build on this result to show in Section 4.4 that NAND-CIRC programs (or equivalently, Boolean circuits) can compute *every* finite function. An alternative direct proof of the same result is given in Section 4.5.

the "universality" of *AND*, *OR*, and *NOT* (and, using the equivalence of Chapter 3, of *NAND* as well)

Despite being an important result, Theorem 4.13 is actually not that hard to prove. Section 4.5 presents a relatively simple direct proof of this result. However, in Section 4.1 and Section 4.3 we derive this result using the concept of "syntactic sugar" (see Big Idea 4). This is an important concept for programming languages theory and practice. The idea behind "syntactic sugar" is that we can extend a programming language by implementing advanced features from its basic components. For example, we can take the AON-CIRC and NAND-CIRC programming languages we saw in Chapter 3, and extend them to achieve features such as user-defined functions (e.g., `def Foo(...)`), conditional statements (e.g., `if blah ...`), and more. Once we have these features, it is not that hard to show that we can take the "truth table" (table of all inputs and outputs) of any function, and use that to create an AON-CIRC or NAND-CIRC program that maps each input to its corresponding output.

We will also get our first glimpse of *quantitative measures* in this chapter. While Theorem 4.13 tells us that every function can be computed by *some* circuit, the number of gates in this circuit can be exponentially large. (We are not using here "exponentially" as some colloquial term for "very very big" but in a very precise mathematical sense, which also happens to coincide with being very very big.) It turns out that *some functions* (for example, integer addition and multi-

plication) can be in fact computed using far fewer gates. We will explore this issue of "gate complexity" more deeply in Chapter 5 and following chapters.

4.1 SOME EXAMPLES OF SYNTACTIC SUGAR

We now present some examples of "syntactic sugar" transformations that we can use in constructing straightline programs or circuits. We focus on the *straight-line programming language* view of our computational models, and specifically (for the sake of concreteness) on the NAND-CIRC programming language. This is convenient because many of the syntactic sugar transformations we present are easiest to think about in terms of applying "search and replace" operations to the source code of a program. However, by Theorem 3.19, all of our results hold equally well for circuits, whether ones using NAND gates or Boolean circuits that use the AND, OR, and NOT operations. Enumerating the examples of such syntactic sugar transformations can be a little tedious, but we do it for two reasons:

1. To convince you that despite their seeming simplicity and limitations, simple models such as Boolean circuits or the NAND-CIRC programming language are actually quite powerful.

2. So you can realize how lucky you are to be taking a theory of computation course and not a compilers course... :)

4.1.1 User-defined procedures

One staple of almost any programming language is the ability to define and then execute *procedures* or *subroutines*. (These are often known as *functions* in some programming languages, but we prefer the name *procedures* to avoid confusion with the function that a program computes.) The NAND-CIRC programming language does not have this mechanism built in. However, we can achieve the same effect using the time-honored technique of "copy and paste". Specifically, we can replace code which defines a procedure such as

```
def Proc(a,b):
    proc_code
    return c
some_code
f = Proc(d,e)
some_more_code
```

with the following code where we "paste" the code of `Proc`

```
some_code
proc_code'
some_more_code
```

and where `proc_code'` is obtained by replacing all occurrences of `a` with `d`, `b` with `e`, and `c` with `f`. When doing that we will need to ensure that all other variables appearing in `proc_code'` don't interfere with other variables. We can always do so by renaming variables to new names that were not used before. The above reasoning leads to the proof of the following theorem:

Theorem 4.1 — Procedure definition synctatic sugar. Let NAND-CIRC-PROC be the programming language NAND-CIRC augmented with the syntax above for defining procedures. Then for every NAND-CIRC-PROC program P, there exists a standard (i.e., "sugar-free") NAND-CIRC program P' that computes the same function as P.

R

Remark 4.2 — No recursive procedure. NAND-CIRC-PROC only allows *non-recursive* procedures. In particular, the code of a procedure `Proc` cannot call `Proc` but only use procedures that were defined before it. Without this restriction, the above "search and replace" procedure might never terminate and Theorem 4.1 would not be true.

Theorem 4.1 can be proven using the transformation above, but since the formal proof is somewhat long and tedious, we omit it here.

■ **Example 4.3 — Computing Majority from NAND using syntactic sugar.** Procedures allow us to express NAND-CIRC programs much more cleanly and succinctly. For example, because we can compute AND, OR, and NOT using NANDs, we can compute the *Majority* function as follows:

```
def NOT(a):
    return NAND(a,a)
def AND(a,b):
    temp = NAND(a,b)
    return NOT(temp)
def OR(a,b):
    temp1 = NOT(a)
    temp2 = NOT(b)
```

```
    return NAND(temp1,temp2)

def MAJ(a,b,c):
    and1 = AND(a,b)
    and2 = AND(a,c)
    and3 = AND(b,c)
    or1 = OR(and1,and2)
    return OR(or1,and3)

print(MAJ(0,1,1))
# 1
```

Fig. 4.2 presents the "sugar-free" NAND-CIRC program (and the corresponding circuit) that is obtained by "expanding out" this program, replacing the calls to procedures with their definitions.

Big Idea 4 Once we show that a computational model X is equivalent to a model that has feature Y, we can assume we have Y when showing that a function f is computable by X.

```
temp = NAND(X[0],X[1])
and1 =  NAND(temp,temp)
temp = NAND(X[0],X[2])
and2 =  NAND(temp,temp)
temp = NAND(X[1],X[2])
and3 =  NAND(temp,temp)
temp1 =  NAND(and1,and1)
temp2 =  NAND(and2,and2)
or1 =  NAND(temp1,temp2)
temp1 =  NAND(or1,or1)
temp2 =  NAND(and3,and3)
Y[0] =  NAND(temp1,temp2)
```

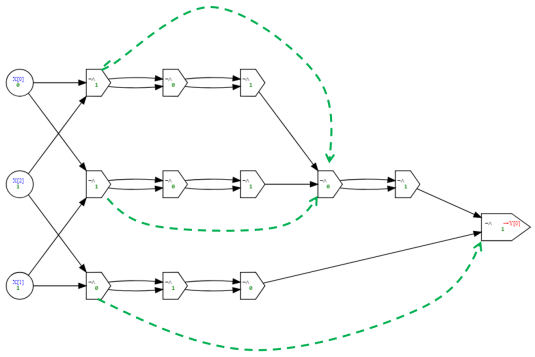

Figure 4.2: A standard (i.e., "sugar-free") NAND-CIRC program that is obtained by expanding out the procedure definitions in the program for Majority of Example 4.3. The corresponding circuit is on the right. Note that this is not the most efficient NAND circuit/program for majority: we can save on some gates by "short cutting" steps where a gate u computes $NAND(v,v)$ and then a gate w computes $NAND(u,u)$ (as indicated by the dashed green arrows in the above figure).

R

Remark 4.4 — Counting lines. While we can use syntactic sugar to *present* NAND-CIRC programs in more readable ways, we did not change the definition of the language itself. Therefore, whenever we say that some function f has an s-line NAND-CIRC program we mean a standard "sugar-free" NAND-CIRC program, where all syntactic sugar has been expanded out. For example, the program of Example 4.3 is a 12-line program for computing the *MAJ* function,

even though it can be written in fewer lines using NAND-CIRC-PROC.

4.1.2 Proof by Python (optional)

We can write a Python program that implements the proof of Theorem 4.1. This is a Python program that takes a NAND-CIRC-PROC program P that includes procedure definitions and uses simple "search and replace" to transform P into a standard (i.e., "sugar-free") NAND-CIRC program P' that computes the same function as P without using any procedures. The idea is simple: if the program P contains a definition of a procedure `Proc` of two arguments `x` and `y`, then whenever we see a line of the form `foo = Proc(bar,blah)`, we can replace this line by:

1. The body of the procedure `Proc` (replacing all occurrences of `x` and `y` with `bar` and `blah` respectively).
2. A line `foo = exp`, where `exp` is the expression following the `return` statement in the definition of the procedure `Proc`.

To make this more robust we add a prefix to the internal variables used by `Proc` to ensure they don't conflict with the variables of P; for simplicity we ignore this issue in the code below though it can be easily added.

The code of the Python function `desugar` below achieves such a transformation.

Fig. 4.2 shows the result of applying `desugar` to the program of Example 4.3 that uses syntactic sugar to compute the Majority function. Specifically, we first apply `desugar` to remove usage of the OR function, then apply it to remove usage of the AND function, and finally apply it a third time to remove usage of the NOT function.

Remark 4.5 — Parsing function definitions (optional). The function `desugar` in Fig. 4.3 assumes that it is given the procedure already split up into its name, arguments, and body. It is not crucial for our purposes to describe precisely how to scan a definition and split it up into these components, but in case you are curious, it can be achieved in Python via the following code:

```
def parse_func(code):
    """Parse a function definition into name,
    ↪ arguments and body"""
    lines = [l.strip() for l in code.split('\n')]
    regexp = r'def\s+([a-zA-Z\_0-9]+)\(([\sa-zA-
    ↪ Z0-9\_,]+)\)\s*:\s*'
```

Figure 4.3: Python code for transforming NAND-CIRC-PROC programs into standard sugar-free NAND-CIRC programs.

```
def desugar(code, func_name, func_args,func_body):
    """
    Replaces all occurences of
       foo = func_name(func_args)
    with
       func_body[x->a,y->b]
       foo = [result returned in func_body]
    """
    # Uses Python regular expressions to simplify the search and replace,
    # see https://docs.python.org/3/library/re.html and Chapter 9 of the book

    # regular expression for capturing a list of variable names separated by commas
    arglist = ",".join([r"([a-zA-Z0-9\_\[\]]+)" for i in range(len(func_args))])
    # regular expression for capturing a statement of the form
    # "variable = func_name(arguments)"
    regexp = fr'([a-zA-Z0-9\_\[\]]+)\s*=\s*{func_name}\({arglist}\)\s*$'
    while True:
        m = re.search(regexp, code, re.MULTILINE)
        if not m: break
        newcode = func_body
        # replace function arguments by the variables from the function invocation
        for i in range(len(func_args)):
            newcode = newcode.replace(func_args[i], m.group(i+2))
        # Splice the new code inside
        newcode = newcode.replace('return', m.group(1) + " = ")
        code = code[:m.start()] + newcode + code[m.end()+1:]
    return code
```

```
        m = re.match(regexp,lines[0])
        return m.group(1), m.group(2).split(','),
         ↪  '\n'.join(lines[1:])
```

4.1.3 Conditional statements

Another sorely missing feature in NAND-CIRC is a conditional statement such as the `if/then` constructs that are found in many programming languages. However, using procedures, we can obtain an ersatz if/then construct. First we can compute the function $IF : \{0,1\}^3 \to \{0,1\}$ such that $IF(a,b,c)$ equals b if $a = 1$ and c if $a = 0$.

Before reading onward, try to see how you could compute the *IF* function using *NAND*'s. Once you do that, see how you can use that to emulate `if/then` types of constructs.

The *IF* function can be implemented from NANDs as follows (see Exercise 4.2):

```
def IF(cond,a,b):
    notcond = NAND(cond,cond)
    temp = NAND(b,notcond)
    temp1 = NAND(a,cond)
    return NAND(temp,temp1)
```

The *IF* function is also known as a *multiplexing* function, since $cond$ can be thought of as a switch that controls whether the output is connected to a or b. Once we have a procedure for computing the *IF* function, we can implement conditionals in NAND. The idea is that we replace code of the form

```
if (condition):  assign blah to variable foo
```

with code of the form

```
foo   = IF(condition, blah, foo)
```

that assigns to `foo` its old value when `condition` equals 0, and assign to `foo` the value of `blah` otherwise. More generally we can replace code of the form

```
if (cond):
    a = ...
    b = ...
    c = ...
```

with code of the form

```
temp_a = ...
temp_b = ...
temp_c = ...
a = IF(cond,temp_a,a)
b = IF(cond,temp_b,b)
c = IF(cond,temp_c,c)
```

Using such transformations, we can prove the following theorem. Once again we omit the (not too insightful) full formal proof, though see Section 4.1.2 for some hints on how to obtain it.

Theorem 4.6 — Conditional statements syntactic sugar. Let NAND-CIRC-IF be the programming language NAND-CIRC augmented with `if/then/else` statements for allowing code to be conditionally executed based on whether a variable is equal to 0 or 1.

Then for every NAND-CIRC-IF program P, there exists a standard (i.e., "sugar-free") NAND-CIRC program P' that computes the same function as P.

4.2 EXTENDED EXAMPLE: ADDITION AND MULTIPLICATION (OPTIONAL)

Using "syntactic sugar", we can write the integer addition function as follows:

```
# Add two n-bit integers
# Use LSB first notation for simplicity
def ADD(A,B):
    Result = [0]*(n+1)
    Carry  = [0]*(n+1)
    Carry[0] = zero(A[0])
    for i in range(n):
        Result[i] = XOR(Carry[i],XOR(A[i],B[i]))
        Carry[i+1] = MAJ(Carry[i],A[i],B[i])
    Result[n] = Carry[n]
    return Result

ADD([1,1,1,0,0],[1,0,0,0,0]);;
# [0, 0, 0, 1, 0, 0]
```

where `zero` is the constant zero function, and `MAJ` and `XOR` correspond to the majority and XOR functions respectively. While we use Python syntax for convenience, in this example n is some *fixed integer* and so for every such n, `ADD` is a *finite* function that takes as input $2n$

bits and outputs $n+1$ bits. In particular for every n we can remove the loop construct `for i in range(n)` by simply repeating the code n times, replacing the value of `i` with $0, 1, 2, \ldots, n-1$. By expanding out all the features, for every value of n we can translate the above program into a standard ("sugar-free") NAND-CIRC program. Fig. 4.4 depicts what we get for $n = 2$.

```
Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(Temp[1],Temp[1])
Temp[3] = NAND(X[0],X[2])
Temp[4] = NAND(X[0],Temp[3])
Temp[5] = NAND(X[2],Temp[3])
Temp[6] = NAND(Temp[4],Temp[5])
Temp[7] = NAND(Temp[2],Temp[6])
Temp[8] = NAND(Temp[2],Temp[7])
Temp[9] = NAND(Temp[6],Temp[7])
Y[0] = NAND(Temp[8],Temp[9])
Temp[11] = NAND(Temp[2],X[0])
Temp[12] = NAND(Temp[11],Temp[11])
Temp[13] = NAND(X[0],X[3])
Temp[14] = NAND(Temp[13],Temp[13])
Temp[15] = NAND(Temp[12],Temp[12])
Temp[16] = NAND(Temp[14],Temp[14])
Temp[17] = NAND(Temp[15],Temp[16])
Temp[18] = NAND(Temp[2],X[2])
Temp[19] = NAND(Temp[18],Temp[18])
Temp[20] = NAND(Temp[17],Temp[17])
Temp[21] = NAND(Temp[19],Temp[19])
Temp[22] = NAND(Temp[20],Temp[21])
Temp[23] = NAND(X[1],X[3])
Temp[24] = NAND(X[1],Temp[23])
Temp[25] = NAND(X[3],Temp[23])
Temp[26] = NAND(Temp[24],Temp[25])
Temp[27] = NAND(Temp[22],Temp[26])
Temp[28] = NAND(Temp[22],Temp[27])
Temp[29] = NAND(Temp[26],Temp[27])
Y[1] = NAND(Temp[28],Temp[29])
Temp[31] = NAND(Temp[22],X[1])
Temp[32] = NAND(Temp[31],Temp[31])
Temp[33] = NAND(X[1],X[3])
Temp[34] = NAND(Temp[33],Temp[33])
Temp[35] = NAND(Temp[32],Temp[32])
Temp[36] = NAND(Temp[34],Temp[34])
Temp[37] = NAND(Temp[35],Temp[36])
Temp[38] = NAND(Temp[22],X[3])
Temp[39] = NAND(Temp[38],Temp[38])
Temp[40] = NAND(Temp[37],Temp[37])
Temp[41] = NAND(Temp[39],Temp[39])
Y[2] = NAND(Temp[40],Temp[41])
```

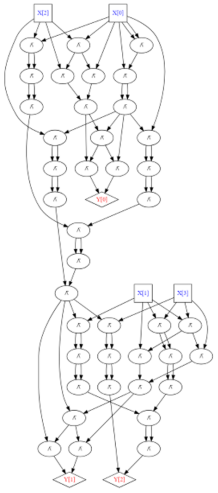

Figure 4.4: The NAND-CIRC program and corresponding NAND circuit for adding two-digit binary numbers that are obtained by "expanding out" all the syntactic sugar. The program/circuit has 43 lines/-gates which is by no means necessary. It is possible to add n bit numbers using $9n$ NAND gates, see Exercise 4.5.

By going through the above program carefully and accounting for the number of gates, we can see that it yields a proof of the following theorem (see also Fig. 4.5):

Theorem 4.7 — Addition using NAND-CIRC programs. For every $n \in \mathbb{N}$, let $ADD_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$ be the function that, given $x, x' \in \{0,1\}^n$ computes the representation of the sum of the numbers that x and x' represent. Then there is a constant $c \leq 30$ such that for every n there is a NAND-CIRC program of at most cn lines computing ADD_n. [1]

Once we have addition, we can use the grade-school algorithm to obtain multiplication as well, thus obtaining the following theorem:

Theorem 4.8 — Multiplication using NAND-CIRC programs. For every n, let $MULT_n : \{0,1\}^{2n} \to \{0,1\}^{2n}$ be the function that, given $x, x' \in \{0,1\}^n$ computes the representation of the product of the numbers that x and x' represent. Then there is a constant c such that for every n, there is a NAND-CIRC program of at most cn^2 lines that computes the function $MULT_n$.

We omit the proof, though in Exercise 4.7 we ask you to supply a "constructive proof" in the form of a program (in your favorite

[1] The value of c can be improved to 9, see Exercise 4.5.

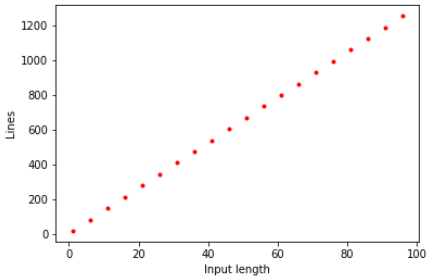

Figure 4.5: The number of lines in our NAND-CIRC program to add two n bit numbers, as a function of n, for n's between 1 and 100. This is not the most efficient program for this task, but the important point is that it has the form $O(n)$.

programming language) that on input a number n, outputs the code of a NAND-CIRC program of at most $1000n^2$ lines that computes the $MULT_n$ function. In fact, we can use Karatsuba's algorithm to show that there is a NAND-CIRC program of $O(n^{\log_2 3})$ lines to compute $MULT_n$ (and can get even further asymptotic improvements using better algorithms).

4.3 THE LOOKUP FUNCTION

The *LOOKUP* function will play an important role in this chapter and later. It is defined as follows:

Definition 4.9 — Lookup function. For every k, the *lookup* function of order k, $LOOKUP_k : \{0,1\}^{2^k+k} \rightarrow \{0,1\}$ is defined as follows: For every $x \in \{0,1\}^{2^k}$ and $i \in \{0,1\}^k$,

$$LOOKUP_k(x,i) = x_i \tag{4.1}$$

where x_i denotes the i^{th} entry of x, using the binary representation to identify i with a number in $\{0,\ldots,2^k-1\}$.

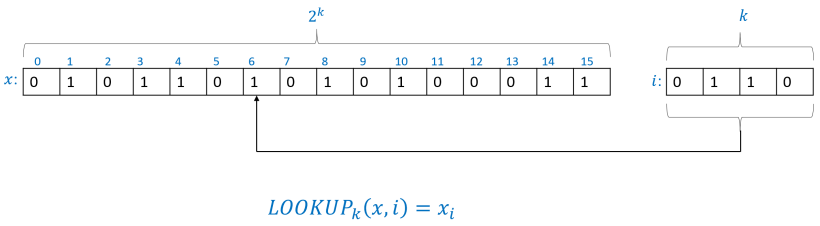

Figure 4.6: The $LOOKUP_k$ function takes an input in $\{0,1\}^{2^k+k}$, which we denote by x, i (with $x \in \{0,1\}^{2^k}$ and $i \in \{0,1\}^k$). The output is x_i: the i-th coordinate of x, where we identify i as a number in $[k]$ using the binary representation. In the above example $x \in \{0,1\}^{16}$ and $i \in \{0,1\}^4$. Since $i = 0110$ is the binary representation of the number 6, the output of $LOOKUP_4(x,i)$ in this case is $x_6 = 1$.

See Fig. 4.6 for an illustration of the LOOKUP function. It turns out that for every k, we can compute $LOOKUP_k$ using a NAND-CIRC program:

Theorem 4.10 — Lookup function. For every $k > 0$, there is a NAND-CIRC program that computes the function $LOOKUP_k : \{0,1\}^{2^k+k} \rightarrow \{0,1\}$. Moreover, the number of lines in this program is at most $4 \cdot 2^k$.

An immediate corollary of Theorem 4.10 is that for every $k > 0$, $LOOKUP_k$ can be computed by a Boolean circuit (with AND, OR and NOT gates) of at most $8 \cdot 2^k$ gates.

4.3.1 Constructing a NAND-CIRC program for *LOOKUP*

We prove Theorem 4.10 by induction. For the case $k = 1$, $LOOKUP_1$ maps $(x_0, x_1, i) \in \{0,1\}^3$ to x_i. In other words, if $i = 0$ then it outputs

x_0 and otherwise it outputs x_1, which (up to reordering variables) is the same as the *IF* function presented in Section 4.1.3, which can be computed by a 4-line NAND-CIRC program.

As a warm-up for the case of general k, let us consider the case of $k = 2$. Given input $x = (x_0, x_1, x_2, x_3)$ for $LOOKUP_2$ and an index $i = (i_0, i_1)$, if the most significant bit i_0 of the index is 0 then $LOOKUP_2(x, i)$ will equal x_0 if $i_1 = 0$ and equal x_1 if $i_1 = 1$. Similarly, if the most significant bit i_0 is 1 then $LOOKUP_2(x, i)$ will equal x_2 if $i_1 = 0$ and will equal x_3 if $i_1 = 1$. Another way to say this is that we can write $LOOKUP_2$ as follows:

```
def LOOKUP2(X[0],X[1],X[2],X[3],i[0],i[1]):
    if i[0]==1:
        return LOOKUP1(X[2],X[3],i[1])
    else:
        return LOOKUP1(X[0],X[1],i[1])
```

or in other words,

```
def LOOKUP2(X[0],X[1],X[2],X[3],i[0],i[1]):
    a = LOOKUP1(X[2],X[3],i[1])
    b = LOOKUP1(X[0],X[1],i[1])
    return IF( i[0],a,b)
```

More generally, as shown in the following lemma, we can compute $LOOKUP_k$ using two invocations of $LOOKUP_{k-1}$ and one invocation of *IF*:

Lemma 4.11 — Lookup recursion. For every $k \geq 2$, $LOOKUP_k(x_0, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1})$ is equal to

$$IF\,(i_0, LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_1, \ldots, i_{k-1}), LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_1, \ldots, i_{k-1})) \tag{4.2}$$

Proof. If the most significant bit i_0 of i is zero, then the index i is in $\{0, \ldots, 2^{k-1} - 1\}$ and hence we can perform the lookup on the "first half" of x and the result of $LOOKUP_k(x, i)$ will be the same as $a = LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_1, \ldots, i_{k-1})$. On the other hand, if this most significant bit i_0 is equal to 1, then the index is in $\{2^{k-1}, \ldots, 2^k - 1\}$, in which case the result of $LOOKUP_k(x, i)$ is the same as $b = LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_1, \ldots, i_{k-1})$. Thus we can compute $LOOKUP_k(x, i)$ by first computing a and b and then outputting $IF(i_{k-1}, a, b)$.

■

Proof of Theorem 4.10 from Lemma 4.11. Now that we have Lemma 4.11, we can complete the proof of Theorem 4.10. We will prove by induction on k that there is a NAND-CIRC program of at most $4 \cdot (2^k - 1)$

lines for $LOOKUP_k$. For $k = 1$ this follows by the four line program for *IF* we've seen before. For $k > 1$, we use the following pseudocode:

```
a = LOOKUP_(k-1)(X[0],...,X[2^(k-1)-1],i[1],...,i[k-1])
b = LOOKUP_(k-1)(X[2^(k-1)],...,Z[2^(k-1)],i[1],...,i[k-
 ↪  1])
return IF(i[0],b,a)
```

If we let $L(k)$ be the number of lines required for $LOOKUP_k$, then the above pseudo-code shows that

$$L(k) \leq 2L(k-1) + 4 \,. \tag{4.3}$$

Since under our induction hypothesis $L(k-1) \leq 4(2^{k-1} - 1)$, we get that $L(k) \leq 2 \cdot 4(2^{k-1} - 1) + 4 = 4(2^k - 1)$ which is what we wanted to prove. See Fig. 4.7 for a plot of the actual number of lines in our implementation of $LOOKUP_k$.

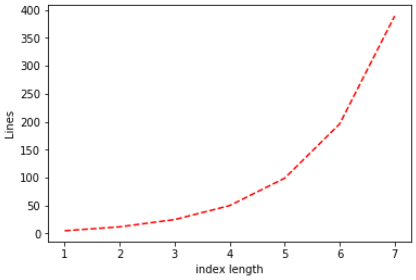

Figure 4.7: The number of lines in our implementation of the LOOKUP_k function as a function of k (i.e., the length of the index). The number of lines in our implementation is roughly $3 \cdot 2^k$.

4.4 COMPUTING *EVERY* FUNCTION

At this point we know the following facts about NAND-CIRC programs (and so equivalently about Boolean circuits and our other equivalent models):

1. They can compute at least some non-trivial functions.
2. Coming up with NAND-CIRC programs for various functions is a very tedious task.

Thus I would not blame the reader if they were not particularly looking forward to a long sequence of examples of functions that can be computed by NAND-CIRC programs. However, it turns out we are not going to need this, as we can show in one fell swoop that NAND-CIRC programs can compute *every* finite function:

Theorem 4.12 — Universality of NAND. There exists some constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0,1\}^n \to \{0,1\}^m$, there is a NAND-CIRC program with at most $c \cdot m 2^n$ lines that computes the function f.

By Theorem 3.19, the models of NAND circuits, NAND-CIRC programs, AON-CIRC programs, and Boolean circuits, are all equivalent to one another, and hence Theorem 4.12 holds for all these models. In particular, the following theorem is equivalent to Theorem 4.12:

Theorem 4.13 — Universality of Boolean circuits. There exists some constant $c > 0$ such that for every $n, m > 0$ and function

> $f : \{0,1\}^n \to \{0,1\}^m$, there is a Boolean circuit with at most $c \cdot m2^n$ gates that computes the function f.

Big Idea 5 *Every* finite function can be computed by a large enough Boolean circuit.

Improved bounds. Though it will not be of great importance to us, it is possible to improve on the proof of Theorem 4.12 and shave an extra factor of n, as well as optimize the constant c, and so prove that for every $\epsilon > 0$, $m \in \mathbb{N}$ and sufficiently large n, if $f : \{0,1\}^n \to \{0,1\}^m$ then f can be computed by a NAND circuit of at most $(1+\epsilon)\frac{m \cdot 2^n}{n}$ gates. The proof of this result is beyond the scope of this book, but we do discuss how to obtain a bound of the form $O(\frac{m \cdot 2^n}{n})$ in Section 4.4.2; see also the biographical notes.

4.4.1 Proof of NAND's Universality

To prove Theorem 4.12, we need to give a NAND circuit, or equivalently a NAND-CIRC program, for *every* possible function. We will restrict our attention to the case of Boolean functions (i.e., $m = 1$). Exercise 4.9 asks you to extend the proof for all values of m. A function $F : \{0,1\}^n \to \{0,1\}$ can be specified by a table of its values for each one of the 2^n inputs. For example, the table below describes one particular function $G : \{0,1\}^4 \to \{0,1\}$:[2]

[2] In case you are curious, this is the function on input $i \in \{0,1\}^4$ (which we interpret as a number in $[16]$), that outputs the i-th digit of π in the binary basis.

Table 4.1: An example of a function $G : \{0,1\}^4 \to \{0,1\}$.

Input (x)	Output ($G(x)$)
0000	1
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	1
1000	0
1001	0
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

For every $x \in \{0,1\}^4$, $G(x) = LOOKUP_4(1100100100001111, x)$, and so the following is NAND-CIRC "pseudocode" to compute G using syntactic sugar for the `LOOKUP_4` procedure.

```
G0000 = 1
G1000 = 1
G0100 = 0
...
G0111 = 1
G1111 = 1
Y[0] = LOOKUP_4(G0000,G1000,...,G1111,
                X[0],X[1],X[2],X[3])
```

We can translate this pseudocode into an actual NAND-CIRC program by adding three lines to define variables `zero` and `one` that are initialized to 0 and 1 respectively, and then replacing a statement such as `Gxxx = 0` with `Gxxx = NAND(one,one)` and a statement such as `Gxxx = 1` with `Gxxx = NAND(zero,zero)`. The call to `LOOKUP_4` will be replaced by the NAND-CIRC program that computes $LOOKUP_4$, plugging in the appropriate inputs.

There was nothing about the above reasoning that was particular to the function G above. Given *every* function $F : \{0,1\}^n \to \{0,1\}$, we can write a NAND-CIRC program that does the following:

1. Initialize 2^n variables of the form `F00...0` till `F11...1` so that for every $z \in \{0,1\}^n$, the variable corresponding to z is assigned the value $F(z)$.
2. Compute $LOOKUP_n$ on the 2^n variables initialized in the previous step, with the index variable being the input variables `X[0]`,...,`X[`2^n-1`]`. That is, just like in the pseudocode for `G` above, we use `Y[0] = LOOKUP(F00..00,...,F11..1,X[0],..,x[`$n-1$`])`

The total number of lines in the resulting program is $3 + 2^n$ lines for initializing the variables plus the $4 \cdot 2^n$ lines that we pay for computing $LOOKUP_n$. This completes the proof of Theorem 4.12.

Remark 4.14 — Result in perspective. While Theorem 4.12 seems striking at first, in retrospect, it is perhaps not that surprising that every finite function can be computed with a NAND-CIRC program. After all, a finite function $F : \{0,1\}^n \to \{0,1\}^m$ can be represented by simply the list of its outputs for each one of the 2^n input values. So it makes sense that we could write a NAND-CIRC program of similar size to compute it. What is more interesting is that *some* functions, such as addition and multiplication, have

a much more efficient representation: one that only requires $O(n^2)$ or even fewer lines.

4.4.2 Improving by a factor of n (optional)

By being a little more careful, we can improve the bound of Theorem 4.12 and show that every function $F : \{0,1\}^n \to \{0,1\}^m$ can be computed by a NAND-CIRC program of at most $O(m2^n/n)$ lines. In other words, we can prove the following improved version:

Theorem 4.15 — Universality of NAND circuits, improved bound. There exists a constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0,1\}^n \to \{0,1\}^m$, there is a NAND-CIRC program with at most $c \cdot m2^n/n$ lines that computes the function f. [3]

[3] The constant c in this theorem is at most 10 and in fact can be arbitrarily close to 1, see Section 4.8.

Proof. As before, it is enough to prove the case that $m = 1$. Hence we let $f : \{0,1\}^n \to \{0,1\}$, and our goal is to prove that there exists a NAND-CIRC program of $O(2^n/n)$ lines (or equivalently a Boolean circuit of $O(2^n/n)$ gates) that computes f.

We let $k = \log(n - 2\log n)$ (the reasoning behind this choice will become clear later on). We define the function $g : \{0,1\}^k \to \{0,1\}^{2^{n-k}}$ as follows:

$$g(a) = f(a0^{n-k})f(a0^{n-k-1}1)\cdots f(a1^{n-k}) \; . \tag{4.4}$$

In other words, if we use the usual binary representation to identify the numbers $\{0,\ldots,2^{n-k}-1\}$ with the strings $\{0,1\}^{n-k}$, then for every $a \in \{0,1\}^k$ and $b \in \{0,1\}^{n-k}$

$$g(a)_b = f(ab) \; . \tag{4.5}$$

(4.5) means that for every $x \in \{0,1\}^n$, if we write $x = ab$ with $a \in \{0,1\}^k$ and $b \in \{0,1\}^{n-k}$ then we can compute $f(x)$ by first computing the string $T = g(a)$ of length 2^{n-k}, and then computing $LOOKUP_{n-k}(T\ ,\ b)$ to retrieve the element of T at the position corresponding to b (see Fig. 4.8). The cost to compute the $LOOKUP_{n-k}$ is $O(2^{n-k})$ lines/gates and the cost in NAND-CIRC lines (or Boolean gates) to compute f is at most

$$cost(g) + O(2^{n-k}) \; , \tag{4.6}$$

where $cost(g)$ is the number of operations (i.e., lines of NAND-CIRC programs or gates in a circuit) needed to compute g.

To complete the proof we need to give a bound on $cost(g)$. Since g is a function mapping $\{0,1\}^k$ to $\{0,1\}^{2^{n-k}}$, we can also think of it as a collection of 2^{n-k} functions $g_0,\ldots,g_{2^{n-k}-1} : \{0,1\}^k \to \{0,1\}$, where

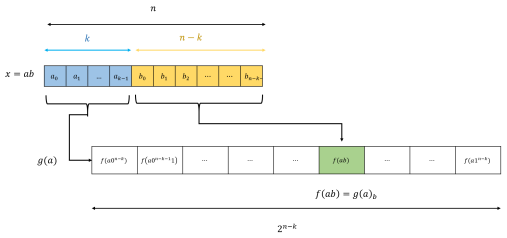

Figure 4.8: We can compute $f : \{0,1\}^n \to \{0,1\}$ on input $x = ab$ where $a \in \{0,1\}^k$ and $b \in \{0,1\}^{n-k}$ by first computing the 2^{n-k} long string $g(a)$ that corresponds to all f's values on inputs that begin with a, and then outputting the b-th coordinate of this string.

$g_i(x) = g(a)_i$ for every $a \in \{0,1\}^k$ and $i \in [2^{n-k}]$. (That is, $g_i(a)$ is the i-th bit of $g(a)$.) Naively, we could use Theorem 4.12 to compute each g_i in $O(2^k)$ lines, but then the total cost is $O(2^{n-k} \cdot 2^k) = O(2^n)$ which does not save us anything. However, the crucial observation is that there are only 2^{2^k} *distinct functions* mapping $\{0,1\}^k$ to $\{0,1\}$. For example, if g_{17} is an identical function to g_{67} that means that if we already computed $g_{17}(a)$ then we can compute $g_{67}(a)$ using only a constant number of operations: simply copy the same value! In general, if you have a collection of N functions $g_0, \ldots, g_{N-1}$ mapping $\{0,1\}^k$ to $\{0,1\}$, of which at most S are distinct then for every value $a \in \{0,1\}^k$ we can compute the N values $g_0(a), \ldots, g_{N-1}(a)$ using at most $O(S \cdot 2^k + N)$ operations (see Fig. 4.9).

In our case, because there are at most 2^{2^k} distinct functions mapping $\{0,1\}^k$ to $\{0,1\}$, we can compute the function g (and hence by (4.5) also f) using at most

$$O(2^{2^k} \cdot 2^k + 2^{n-k}) \tag{4.7}$$

operations. Now all that is left is to plug into (4.7) our choice of $k = \log(n - 2\log n)$. By definition, $2^k = n - 2\log n$, which means that (4.7) can be bounded

$$O\left(2^{n-2\log n} \cdot (n - 2\log n) + 2^{n-\log(n-2\log n)}\right) \leq \tag{4.8}$$

$$O\left(\tfrac{2^n}{n^2} \cdot n + \tfrac{2^n}{n-2\log n}\right) \leq O\left(\tfrac{2^n}{n} + \tfrac{2^n}{0.5n}\right) = O\left(\tfrac{2^n}{n}\right) \tag{4.9}$$

which is what we wanted to prove. (We used above the fact that $n - 2\log n \geq 0.5\log n$ for sufficiently large n.)

■

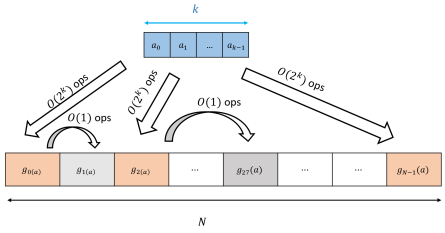

Figure 4.9: If $g_0, \ldots, g_{N-1}$ is a collection of functions each mapping $\{0,1\}^k$ to $\{0,1\}$ such that at most S of them are distinct then for every $a \in \{0,1\}^k$, we can compute all the values $g_0(a), \ldots, g_{N-1}(a)$ using at most $O(S \cdot 2^k + N)$ operations by first computing the distinct functions and then copying the resulting values.

Using the connection between NAND-CIRC programs and Boolean circuits, an immediate corollary of Theorem 4.15 is the following improvement to Theorem 4.13:

Theorem 4.16 — Universality of Boolean circuits, improved bound. There exists some constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0,1\}^n \to \{0,1\}^m$, there is a Boolean circuit with at most $c \cdot m2^n/n$ gates that computes the function f.

4.5 COMPUTING EVERY FUNCTION: AN ALTERNATIVE PROOF

Theorem 4.13 is a fundamental result in the theory (and practice!) of computation. In this section, we present an alternative proof of this basic fact that Boolean circuits can compute every finite function. This alternative proof gives a somewhat worse quantitative bound on the number of gates but it has the advantage of being simpler, working

directly with circuits and avoiding the usage of all the syntactic sugar machinery. (However, that machinery is useful in its own right, and will find other applications later on.)

Theorem 4.17 — Universality of Boolean circuits (alternative phrasing). There exists some constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0,1\}^n \to \{0,1\}^m$, there is a Boolean circuit with at most $c \cdot m \cdot n2^n$ gates that computes the function f.

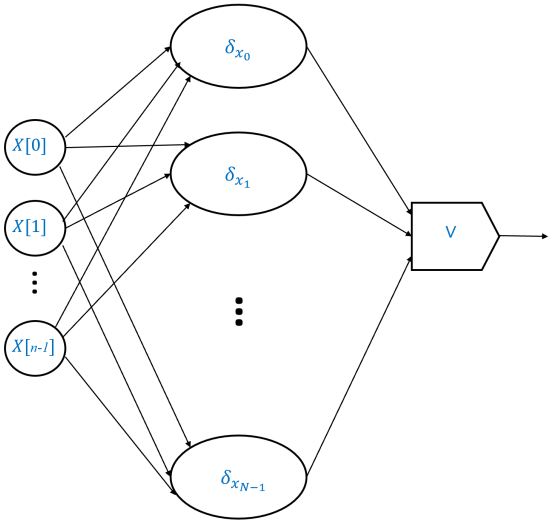

Figure 4.10: Given a function $f : \{0,1\}^n \to \{0,1\}$, we let $\{x_0, x_1, \ldots, x_{N-1}\} \subseteq \{0,1\}^n$ be the set of inputs such that $f(x_i) = 1$, and note that $N \leq 2^n$. We can express f as the OR of δ_{x_i} for $i \in [N]$ where the function $\delta_\alpha : \{0,1\}^n \to \{0,1\}$ (for $\alpha \in \{0,1\}^n$) is defined as follows: $\delta_\alpha(x) = 1$ iff $x = \alpha$. We can compute the OR of N values using N two-input OR gates. Therefore if we have a circuit of size $O(n)$ to compute δ_α for every $\alpha \in \{0,1\}^n$, we can compute f using a circuit of size $O(n \cdot N) = O(n \cdot 2^n)$.

Proof Idea:

The idea of the proof is illustrated in Fig. 4.10. As before, it is enough to focus on the case that $m = 1$ (the function f has a single output), since we can always extend this to the case of $m > 1$ by looking at the composition of m circuits each computing a different output bit of the function f. We start by showing that for every $\alpha \in \{0,1\}^n$, there is an $O(n)$-sized circuit that computes the function $\delta_\alpha : \{0,1\}^n \to \{0,1\}$ defined as follows: $\delta_\alpha(x) = 1$ iff $x = \alpha$ (that is, δ_α outputs 0 on all inputs except the input α). We can then write any function $f : \{0,1\}^n \to \{0,1\}$ as the OR of at most 2^n functions δ_α for the α's on which $f(\alpha) = 1$.

⋆

Proof of Theorem 4.17. We prove the theorem for the case $m = 1$. The result can be extended for $m > 1$ as before (see also Exercise 4.9). Let $f : \{0,1\}^n \to \{0,1\}$. We will prove that there is an $O(n \cdot 2^n)$-sized Boolean circuit to compute f in the following steps:

1. We show that for every $\alpha \in \{0,1\}^n$, there is an $O(n)$-sized circuit that computes the function $\delta_\alpha : \{0,1\}^n \to \{0,1\}$, where $\delta_\alpha(x) = 1$ iff $x = \alpha$.

2. We then show that this implies the existence of an $O(n \cdot 2^n)$-sized circuit that computes f, by writing $f(x)$ as the OR of $\delta_\alpha(x)$ for all $\alpha \in \{0,1\}^n$ such that $f(\alpha) = 1$. (If f is the constant zero function and hence there is no such α, then we can use the circuit $f(x) = x_0 \wedge \overline{x}_0$.)

We start with Step 1:

CLAIM: For $\alpha \in \{0,1\}^n$, define $\delta_\alpha : \{0,1\}^n$ as follows:

$$\delta_\alpha(x) = \begin{cases} 1 & x = \alpha \\ 0 & \text{otherwise} \end{cases} . \tag{4.10}$$

then there is a Boolean circuit using at most $2n$ gates that computes δ_α.

PROOF OF CLAIM: The proof is illustrated in Fig. 4.11. As an example, consider the function $\delta_{011} : \{0,1\}^3 \to \{0,1\}$. This function outputs 1 on x if and only if $x_0 = 0$, $x_1 = 1$ and $x_2 = 1$, and so we can write $\delta_{011}(x) = \overline{x_0} \wedge x_1 \wedge x_2$, which translates into a Boolean circuit with one NOT gate and two AND gates. More generally, for every $\alpha \in \{0,1\}^n$, we can express $\delta_\alpha(x)$ as $(x_0 = \alpha_0) \wedge (x_1 = \alpha_1) \wedge \cdots \wedge (x_{n-1} = \alpha_{n-1})$, where if $\alpha_i = 0$ we replace $x_i = \alpha_i$ with $\overline{x_i}$ and if $\alpha_i = 1$ we replace $x_i = \alpha_i$ by simply x_i. This yields a circuit that computes δ_α using n AND gates and at most n NOT gates, so a total of at most $2n$ gates.

Now for every function $f : \{0,1\}^n \to \{0,1\}$, we can write

$$f(x) = \delta_{x_0}(x) \vee \delta_{x_1}(x) \vee \cdots \vee \delta_{x_{N-1}}(x) \tag{4.11}$$

where $S = \{x_0, \ldots, x_{N-1}\}$ is the set of inputs on which f outputs 1. (To see this, you can verify that the right-hand side of (4.11) evaluates to 1 on $x \in \{0,1\}^n$ if and only if x is in the set S.)

Therefore we can compute f using a Boolean circuit of at most $2n$ gates for each of the N functions δ_{x_i} and combine that with at most N OR gates, thus obtaining a circuit of at most $2n \cdot N + N$ gates. Since $S \subseteq \{0,1\}^n$, its size N is at most 2^n and hence the total number of gates in this circuit is $O(n \cdot 2^n)$.

■

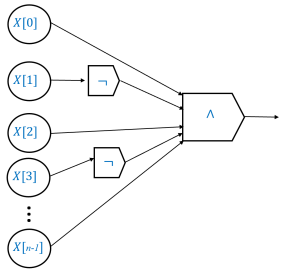

Figure 4.11: For every string $\alpha \in \{0,1\}^n$, there is a Boolean circuit of $O(n)$ gates to compute the function $\delta_\alpha : \{0,1\}^n \to \{0,1\}$ such that $\delta_\alpha(x) = 1$ if and only if $x = \alpha$. The circuit is very simple. Given input $x_0, \ldots, x_{n-1}$ we compute the AND of $z_0, \ldots, z_{n-1}$ where $z_i = x_i$ if $\alpha_i = 1$ and $z_i = NOT(x_i)$ if $\alpha_i = 0$. While formally Boolean circuits only have a gate for computing the AND of two inputs, we can implement an AND of n inputs by composing n two-input ANDs.

4.6 THE CLASS $SIZE(T)$

We have seen that *every* function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a circuit of size $O(m \cdot 2^n)$, and *some* functions (such as addition and multiplication) can be computed by much smaller circuits.

We define $SIZE(s)$ to be the set of functions that can be computed by NAND circuits of at most s gates (or equivalently, by NAND-CIRC programs of at most s lines). Formally, the definition is as follows:

Definition 4.18 — Size class of functions. For every $n, m \in \{1, \ldots, 2s\}$, we let $SIZE_{n,m}(s)$ denote the set of all functions $f : \{0,1\}^n \to \{0,1\}^m$ such that $f \in SIZE(s)$. [4] We denote by $SIZE_n(s)$ the set $SIZE_{n,1}(s)$. For every integer $s \geq 1$, we let $SIZE(s) = \cup_{n,m \leq 2s} SIZE_{n,m}(s)$ be the set of all functions f for which there exists a NAND circuit of at most s gates that compute f.

[4] The restriction that $m, n \leq 2s$ makes no difference; see Exercise 3.11.

Fig. 4.12 depicts the set $SIZE_{n,1}(s)$. Note that $SIZE_{n,m}(s)$ is a set of *functions*, not of *programs!* (Asking if a program or a circuit is a member of $SIZE_{n,m}(s)$ is a *category error* as in the sense of Fig. 4.13.) As we discussed in Section 3.6.2 (and Section 2.6.1), the distinction between *programs* and *functions* is absolutely crucial. You should always remember that while a program *computes* a function, it is not *equal* to a function. In particular, as we've seen, there can be more than one program to compute the same function.

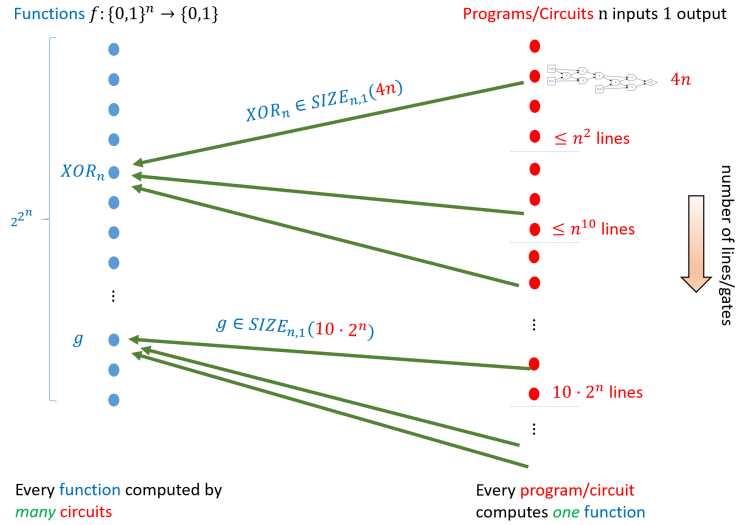

Figure 4.12: There are 2^{2^n} functions mapping $\{0,1\}^n$ to $\{0,1\}$, and an infinite number of circuits with n bit inputs and a single bit of output. Every circuit computes one function, but every function can be computed by many circuits. We say that $f \in SIZE_{n,1}(s)$ if the smallest circuit that computes f has s or fewer gates. For example $XOR_n \in SIZE_{n,1}(4n)$. Theorem 4.12 shows that *every* function g is computable by some circuit of at most $c \cdot 2^n/n$ gates, and hence $SIZE_{n,1}(c \cdot 2^n/n)$ corresponds to the set of *all* functions from $\{0,1\}^n$ to $\{0,1\}$.

While we defined $SIZE(s)$ with respect to NAND gates, we would get essentially the same class if we defined it with respect to AND/OR/NOT gates:

Lemma 4.19 Let $SIZE^{AON}_{n,m}(s)$ denote the set of all functions $f : \{0,1\}^n \to \{0,1\}^m$ that can be computed by an AND/OR/NOT Boolean circuit of at most s gates. Then,

$$SIZE_{n,m}(s/2) \subseteq SIZE^{AON}_{n,m}(s) \subseteq SIZE_{n,m}(3s) \tag{4.12}$$

Proof. If f can be computed by a NAND circuit of at most $s/2$ gates, then by replacing each NAND with the two gates NOT and AND, we can obtain an AND/OR/NOT Boolean circuit of at most s gates that computes f. On the other hand, if f can be computed by a Boolean AND/OR/NOT circuit of at most s gates, then by Theorem 3.12 it can be computed by a NAND circuit of at most $3s$ gates. ■

The results we have seen in this chapter can be phrased as showing that $ADD_n \in SIZE_{2n,n+1}(100n)$ and $MULT_n \in SIZE_{2n,2n}(10000n^{\log_2 3})$. Theorem 4.12 shows that for some constant c, $SIZE_{n,m}(cm2^n)$ is equal to the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$.

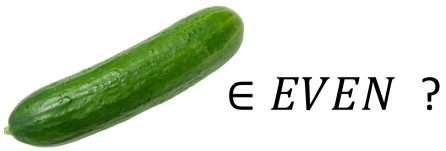

Figure 4.13: A "category error" is a question such as "is a cucumber even or odd?" which does not even make sense. In this book one type of category error you should watch out for is confusing *functions* and *programs* (i.e., confusing *specifications* and *implementations*). If C is a circuit or program, then asking if $C \in SIZE_{n,1}(s)$ is a category error, since $SIZE_{n,1}(s)$ is a set of *functions* and not programs or circuits.

R

Remark 4.20 — Finite vs infinite functions. A NAND-CIRC program P can only compute a function with a certain number n of inputs and a certain number m of outputs. Hence, for example, there is no single NAND-CIRC program that can compute the increment function $INC : \{0,1\}^* \to \{0,1\}^*$ that maps a string x (which we identify with a number via the binary representation) to the string that represents $x+1$. Rather for every $n > 0$, there is a NAND-CIRC program P_n that computes the restriction INC_n of the function INC to inputs of length n. Since it can be shown that for every $n > 0$ such a program P_n exists of length at most $10n$, $INC_n \in SIZE(10n)$ for every $n > 0$.

If $T : \mathbb{N} \to \mathbb{N}$ and $F : \{0,1\}^* \to \{0,1\}^*$, we will write $F \in SIZE_*(T(n))$ (or sometimes slightly abuse notation and write simply $F \in SIZE(T(n))$) to indicate that for every n the restriction $F_{\restriction n}$ of F to inputs in $\{0,1\}^n$ is in $SIZE(T(n))$. Hence we can write $INC \in SIZE_*(10n)$. We will come back to this issue of finite vs infinite functions later in this course.

Solved Exercise 4.1 — $SIZE$ closed under complement.. In this exercise we prove a certain "closure property" of the class $SIZE_n(s)$. That is, we show that if f is in this class then (up to some small additive term) so is the complement of f, which is the function $g(x) = 1 - f(x)$.

Prove that there is a constant c such that for every $f : \{0,1\}^n \to \{0,1\}$ and $s \in \mathbb{N}$, if $f \in SIZE_n(s)$ then $1 - f \in SIZE_n(s+c)$. ■

Solution:

If $f \in SIZE(s)$ then there is an s-line NAND-CIRC program P that computes f. We can rename the variable `Y[0]` in P to a variable `temp` and add the line

```
Y[0] = NAND(temp,temp)
```

at the very end to obtain a program P' that computes $1 - f$.

■

Chapter Recap

- We can define the notion of computing a function via a simplified "programming language", where computing a function F in T steps would correspond to having a T-line NAND-CIRC program that computes F.
- While the NAND-CIRC programming only has one operation, other operations such as functions and conditional execution can be implemented using it.
- Every function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a circuit of at most $O(m2^n)$ gates (and in fact at most $O(m2^n/n)$ gates).
- Sometimes (or maybe always?) we can translate an *efficient* algorithm to compute f into a circuit that computes f with a number of gates comparable to the number of steps in this algorithm.

4.7 EXERCISES

Exercise 4.1 — Pairing. This exercise asks you to give a one-to-one map from $\mathbb{N}^2$ to $\mathbb{N}$. This can be useful to implement two-dimensional arrays as "syntactic sugar" in programming languages that only have one-dimensional array.

1. Prove that the map $F(x,y) = 2^x 3^y$ is a one-to-one map from $\mathbb{N}^2$ to $\mathbb{N}$.
2. Show that there is a one-to-one map $F : \mathbb{N}^2 \to \mathbb{N}$ such that for every x, y, $F(x,y) \leq 100 \cdot \max\{x,y\}^2 + 100$.
3. For every k, show that there is a one-to-one map $F : \mathbb{N}^k \to \mathbb{N}$ such that for every $x_0, \ldots, x_{k-1} \in \mathbb{N}$, $F(x_0, \ldots, x_{k-1}) \leq 100 \cdot (x_0 + x_1 + \ldots + x_{k-1} + 100k)^k$.

■

Exercise 4.2 — Computing MUX. Prove that the NAND-CIRC program below computes the function MUX (or $LOOKUP_1$) where $MUX(a,b,c)$ equals a if $c = 0$ and equals b if $c = 1$:

```
t = NAND(X[2],X[2])
u = NAND(X[0],t)
v = NAND(X[1],X[2])
Y[0] = NAND(u,v)
```

Exercise 4.3 — At least two / Majority. Give a NAND-CIRC program of at most 6 lines to compute the function $MAJ : \{0,1\}^3 \to \{0,1\}$ where $MAJ(a,b,c) = 1$ iff $a + b + c \geq 2$.

Exercise 4.4 — Conditional statements. In this exercise we will explore Theorem 4.6: transforming NAND-CIRC-IF programs that use code such as `if .. then .. else ..` to standard NAND-CIRC programs.

1. Give a "proof by code" of Theorem 4.6: a program in a programming language of your choice that transforms a NAND-CIRC-IF program P into a "sugar-free" NAND-CIRC program P' that computes the same function. See footnote for hint.[5]

2. Prove the following statement, which is the heart of Theorem 4.6: suppose that there exists an s-line NAND-CIRC program to compute $f : \{0,1\}^n \to \{0,1\}$ and an s'-line NAND-CIRC program to compute $g : \{0,1\}^n \to \{0,1\}$. Prove that there exist a NAND-CIRC program of at most $s + s' + 10$ lines to compute the function $h : \{0,1\}^{n+1} \to \{0,1\}$ where $h(x_0, \ldots, x_{n-1}, x_n)$ equals $f(x_0, \ldots, x_{n-1})$ if $x_n = 0$ and equals $g(x_0, \ldots, x_{n-1})$ otherwise. (All programs in this item are standard "sugar-free" NAND-CIRC programs.)

[5] You can start by transforming P into a NAND-CIRC-PROC program that uses procedure statements, and then use the code of Fig. 4.3 to transform the latter into a "sugar-free" NAND-CIRC program.

Exercise 4.5 — Half and full adders. 1. A *half adder* is the function $HA : \{0,1\}^2 :\to \{0,1\}^2$ that corresponds to adding two binary bits. That is, for every $a, b \in \{0,1\}$, $HA(a,b) = (e,f)$ where $2e + f = a + b$. Prove that there is a NAND circuit of at most five NAND gates that computes HA.

2. A *full adder* is the function $FA : \{0,1\}^3 \to \{0,1\}^2$ that takes in two bits and a "carry" bit and outputs their sum. That is, for every $a, b, c \in \{0,1\}$, $FA(a,b,c) = (e,f)$ such that $2e + f = a + b + c$. Prove that there is a NAND circuit of at most nine NAND gates that computes FA.

3. Prove that if there is a NAND circuit of c gates that computes FA, then there is a circuit of cn gates that computes ADD_n where (as in Theorem 4.7) $ADD_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$ is the function that outputs the addition of two input n-bit numbers. See footnote for hint.[6]

4. Show that for every n there is a NAND-CIRC program to compute ADD_n with at most $9n$ lines.

[6] Use a "cascade" of adding the bits one after the other, starting with the least significant digit, just like in the elementary-school algorithm.

Exercise 4.6 — Addition. Write a program using your favorite programming language that on input of an integer n, outputs a NAND-CIRC program that computes ADD_n. Can you ensure that the program it outputs for ADD_n has fewer than $10n$ lines?

■

Exercise 4.7 — Multiplication. Write a program using your favorite programming language that on input of an integer n, outputs a NAND-CIRC program that computes $MULT_n$. Can you ensure that the program it outputs for $MULT_n$ has fewer than $1000 \cdot n^2$ lines?

■

Exercise 4.8 — Efficient multiplication (challenge). Write a program using your favorite programming language that on input of an integer n, outputs a NAND-CIRC program that computes $MULT_n$ and has at most $10000n^{1.9}$ lines.[7] What is the smallest number of lines you can use to multiply two 2048 bit numbers?

[7] **Hint:** Use Karatsuba's algorithm.

■

Exercise 4.9 — Multibit function. In the text Theorem 4.12 is only proven for the case $m = 1$. In this exercise you will extend the proof for every m.

Prove that

1. If there is an s-line NAND-CIRC program to compute $f : \{0,1\}^n \rightarrow \{0,1\}$ and an s'-line NAND-CIRC program to compute $f' : \{0,1\}^n \rightarrow \{0,1\}$ then there is an $s + s'$-line program to compute the function $g : \{0,1\}^n \rightarrow \{0,1\}^2$ such that $g(x) = (f(x), f'(x))$.

2. For every function $f : \{0,1\}^n \rightarrow \{0,1\}^m$, there is a NAND-CIRC program of at most $10m \cdot 2^n$ lines that computes f. (You can use the $m = 1$ case of Theorem 4.12, as well as Item 1.)

■

Exercise 4.10 — Simplifying using syntactic sugar. Let P be the following NAND-CIRC program:

```
Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[1],X[1])
Temp[2] = NAND(Temp[0],Temp[1])
Temp[3] = NAND(X[2],X[2])
Temp[4] = NAND(X[3],X[3])
Temp[5] = NAND(Temp[3],Temp[4])
Temp[6] = NAND(Temp[2],Temp[2])
Temp[7] = NAND(Temp[5],Temp[5])
Y[0] = NAND(Temp[6],Temp[7])
```

1. Write a program P' with at most three lines of code that uses both `NAND` as well as the syntactic sugar `OR` that computes the same function as P.

2. Draw a circuit that computes the same function as P and uses only *AND* and *NOT* gates.

■

In the following exercises you are asked to compare the *power* of pairs of programming languages. By "comparing the power" of two programming languages X and Y we mean determining the relation between the set of functions that are computable using programs in X and Y respectively. That is, to answer such a question you need to do both of the following:

1. Either prove that for every program P in X there is a program P' in Y that computes the same function as P, *or* give an example for a function that is computable by an X-program but not computable by a Y-program.

   and

2. Either prove that for every program P in Y there is a program P' in X that computes the same function as P, *or* give an example for a function that is computable by a Y-program but not computable by an X-program.

When you give an example as above of a function that is computable in one programming language but not the other, you need to *prove* that the function you showed is (1) computable in the first programming language and (2) *not computable* in the second programming language.

Exercise 4.11 — Compare IF and NAND. Let IF-CIRC be the programming language where we have the following operations `foo = 0`, `foo = 1`, `foo = IF(cond,yes,no)` (that is, we can use the constants 0 and 1, and the $IF : \{0,1\}^3 \rightarrow \{0,1\}$ function such that $IF(a,b,c)$ equals b if $a = 1$ and equals c if $a = 0$). Compare the power of the NAND-CIRC programming language and the IF-CIRC programming language.

■

Exercise 4.12 — Compare XOR and NAND. Let XOR-CIRC be the programming language where we have the following operations `foo = XOR(bar,blah)`, `foo = 1` and `bar = 0` (that is, we can use the constants 0, 1 and the *XOR* function that maps $a, b \in \{0,1\}^2$ to $a + b \mod 2$). Compare the power of the NAND-CIRC programming language and the XOR-CIRC programming language. See footnote for hint.[8]

Exercise 4.13 — Circuits for majority. Prove that there is some constant c such that for every $n > 1$, $MAJ_n \in SIZE(cn)$ where $MAJ_n : \{0,1\}^n \to \{0,1\}$ is the majority function on n input bits. That is $MAJ_n(x) = 1$ iff $\sum_{i=0}^{n-1} x_i > n/2$. See footnote for hint.[9]

[9] One approach to solve this is using recursion and the so-called Master Theorem.

Exercise 4.14 — Circuits for threshold. Prove that there is some constant c such that for every $n > 1$, and integers $a_0, \ldots, a_{n-1}, b \in \{-2^n, -2^n + 1, \ldots, -1, 0, +1, \ldots, 2^n\}$, there is a NAND circuit with at most n^c gates that computes the *threshold* function $f_{a_0,\ldots,a_{n-1},b} : \{0,1\}^n \to \{0,1\}$ that on input $x \in \{0,1\}^n$ outputs 1 if and only if $\sum_{i=0}^{n-1} a_i x_i > b$.

4.8 BIBLIOGRAPHICAL NOTES

See Jukna's and Wegener's books [Juk12; Weg87] for much more extensive discussion on circuits. Shannon showed that every Boolean function can be computed by a circuit of exponential size [Sha38]. The improved bound of $c \cdot 2^n/n$ (with the optimal value of c for many bases) is due to Lupanov [Lup58]. An exposition of this for the case of NAND (where $c = 1$) is given in Chapter 4 of his book [Lup84]. (Thanks to Sasha Golovnev for tracking down this reference!)

The concept of "syntactic sugar" is also known as "macros" or "meta-programming" and is sometimes implemented via a preprocessor or macro language in a programming language or a text editor. One modern example is the Babel JavaScript syntax transformer, that converts JavaScript programs written using the latest features into a format that older Browsers can accept. It even has a plug-in architecture, that allows users to add their own syntactic sugar to the language.