Syntactic sugar, and computing every function

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# Syntactic sugar, and computing every function

### 

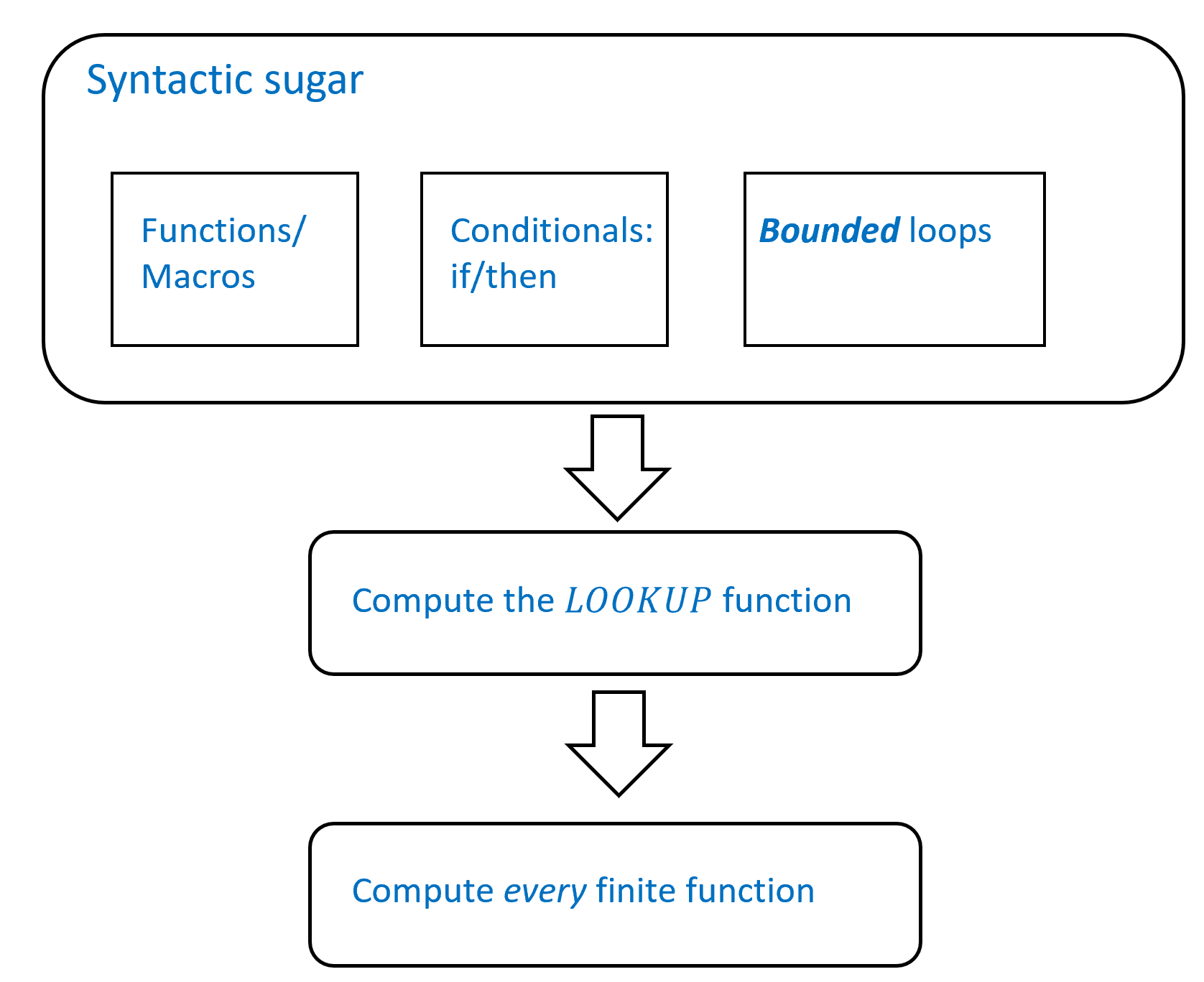
* Get comfortable with syntactic sugar or automatic translation of higher level logic to low level gates.
* Learn proof of major result: every finite function can be computed by a Boolean circuit.
* Start thinking *quantitatively* about number of lines required for computation.

*“[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 can be computed by a Boolean circuit, see circuit-univ-thm below. While the syntactic sugar toolkit is important in its own right, circuit-univ-thm can also be proven directly without using this toolkit. We present this alternative proof in seccomputalternative. See computefuncoverviewfig for an outline of the results of this chapter.



An outline of the results of this chapter. In secsyntacticsugar 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 seclookupfunc to give a NAND-CIRC program (or alternatively a Boolean circuit) to compute the function. We then build on this result to show in seccomputeallfunctions that NAND-CIRC programs (or equivalently, Boolean circuits) can compute *every* finite function. An alternative direct proof of the same result is given in seccomputalternative.

## 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 equivalencemodelsthm, 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… :)

### 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:

### 

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 , there exists a standard (i.e., “sugar free”) NAND-CIRC program that computes the same function as .

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 functionsynsugarthm would not be true.

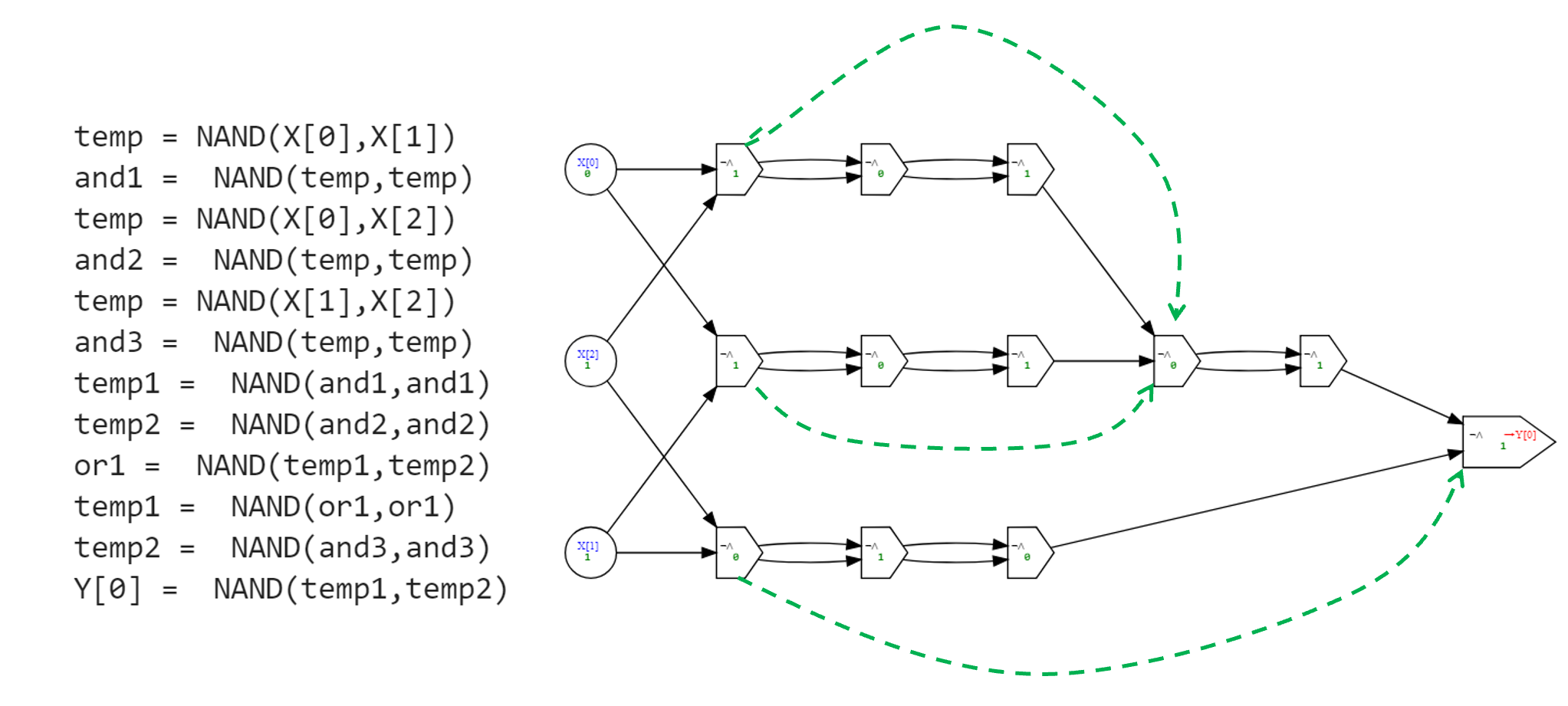
functionsynsugarthm can be proven using the transformation above, but since the formal proof is somewhat long and tedious, we omit it here.

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

progcircmajfig 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.

Once we show that a computational model is equivalent to a model that has feature , we can assume we have when showing that a function is computable by .



A standard (i.e., “sugar free”) NAND-CIRC program that is obtained by expanding out the procedure definitions in the program for Majority of majcircnand. 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 computes and then a gate computes (as indicated by the dashed green arrows in the above figure).

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 has an -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 majcircnand is a -line program for computing the function, even though it can be written in fewer lines using NAND-CIRC-PROC.

### Proof by Python (optional)

We can write a Python program that implements the proof of functionsynsugarthm. This is a Python program that takes a NAND-CIRC-PROC program that includes procedure definitions and uses simple “search and replace” to transform into a standard (i.e., “sugar free”) NAND-CIRC program that computes the same function as without using any procedures. The idea is simple: if the program 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 ; for simplicity we ignore this issue in the code below though it can be easily added.

The code in desugarcode achieves such a transformation.[[1]](#footnote-35)

def inline\_proc(code, proc\_name, proc\_args,proc\_body):  
 '''Takes code of a program and name, arguments, body of a procedure.  
 Returns new code where all lines in program of the  
 form "foo = proc\_name(bar,blah,..)" are replaced with  
 the body of the procedure with arguments instantiated  
 with the variables bar, blah, etc.'''  
 arglist = ",".join([r"([a-zA-Z0-9\\_\[\]]+)" for i in range(len(proc\_args))])  
 regexp = fr'([a-zA-Z0-9\\_\[\]]+)\s\*=\s\*{proc\_name}\({arglist}\)\s\*$'  
 # captures "variable = func\_name(arguments)"   
 while True:  
 m = re.search(regexp, code, re.MULTILINE)  
 if not m: break  
 newcode = proc\_body  
 for i in range(len(proc\_args)):  
 newcode = newcode.replace(proc\_args[i], m.group(i+2))  
 newcode = newcode.replace('return', m.group(1) + " = ")  
 code = code[:m.start()] + newcode + code[m.end()+1:]  
 return code

progcircmajfig shows the result of applying the code of desugarcode to the program of majcircnand 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.

The function desugar in desugarcode 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\_procs(code):  
 """Parse code that contain procedure definitions into a list of  
 triples (name, arguments, body)"""  
 lines = [l for l in code.split('\n') if l ]  
 regexp = r'def\s+([a-zA-Z\\_0-9]+)\(([\sa-zA-Z0-9\\_,]+)\)\s\*:\s\*'  
 procs = []  
 current\_line = 0  
 rest = ""  
 while current\_line < len(lines):  
 m = re.match(regexp,lines[current\_line])  
 if m:  
 current\_line+= 1  
 code = ""  
 while current\_line < len(lines) and lines[current\_line][0]==' ':  
 code += lines[current\_line].strip()+'\n'  
 current\_line += 1  
 procs.append((m.group(1) , m.group(2).split(','), code))  
 else:  
 rest += lines[current\_line]+'\n'  
 current\_line += 1  
 return rest, procs

### 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 such that equals if and if .

### 

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

The function can be implemented from NANDs as follows (see mux-ex):

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

The function is also known as a *multiplexing* function, since can be thought of as a switch that controls whether the output is connected to or . Once we have a procedure for computing the 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 , 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 functionsynsugarthmpython for some hints on how to obtain it.

### 

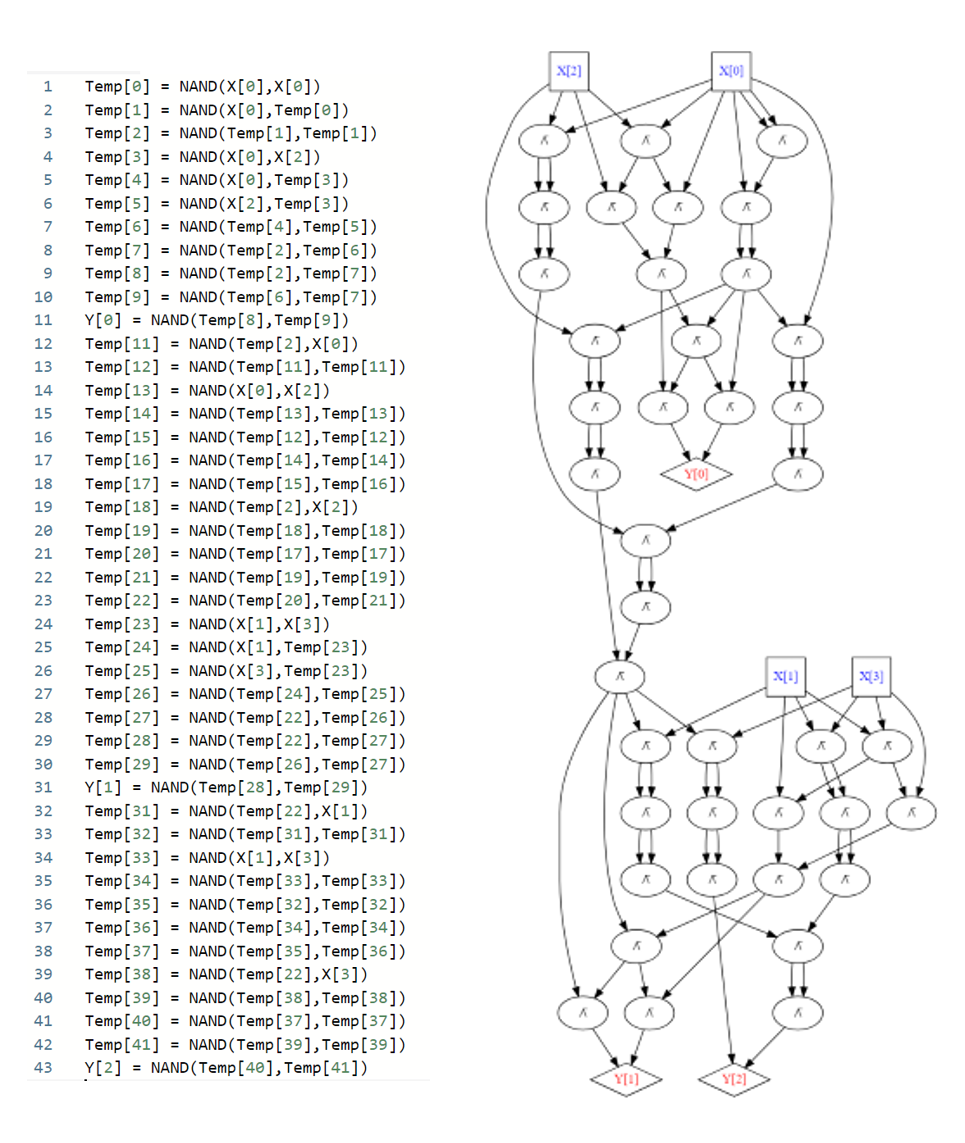
Let NAND-CIRC-IF be the programming language NAND-CIRC augmented with if/then/else statements for allowing code to be conditionally executed based on whether a variable is equal to or .  
Then for every NAND-CIRC-IF program , there exists a standard (i.e., “sugar free”) NAND-CIRC program that computes the same function as .

## 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. In the above we used the *loop* for i in range(n) but we can expand this out by simply repeating the code times, replacing the value of i with . By expanding out all the features, for every value of we can translate the above program into a standard (“sugar free”) NAND-CIRC program. add2bitnumbersfig depicts what we get for .

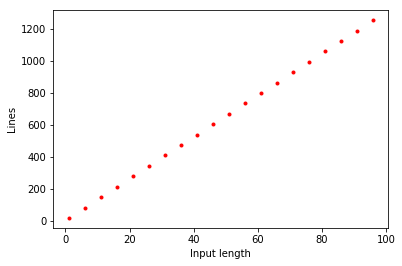


The NAND-CIRC program and corresponding NAND circuit for adding two-digit binary numbers that are obtained by “expanding out” all the syntactic sugar. The program/circuit has 43 lines/gates which is by no means necessary. It is possible to add bit numbers using NAND gates, see halffulladderex.

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 addnumoflinesfig):

### 

For every , let be the function that, given computes the representation of the sum of the numbers that and represent. Then there is a constant such that for every there is a NAND-CIRC program of at most lines computing .[[2]](#footnote-45)



The number of lines in our NAND-CIRC program to add two bit numbers, as a function of , for ’s between and . This is not the most efficient program for this task, but the important point is that it has the form .

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

### 

For every , let be the function that, given computes the representation of the product of the numbers that and represent. Then there is a constant such that for every , there is a NAND-CIRC program of at most that computes the function .

We omit the proof, though in multiplication-ex we ask you to supply a “constructive proof” in the form of a program (in your favorite programming language) that on input a number , outputs the code of a NAND-CIRC program of at most lines that computes the function. In fact, we can use Karatsuba’s algorithm to show that there is a NAND-CIRC program of lines to compute (and can get even further asymptotic improvements using better algorithms).

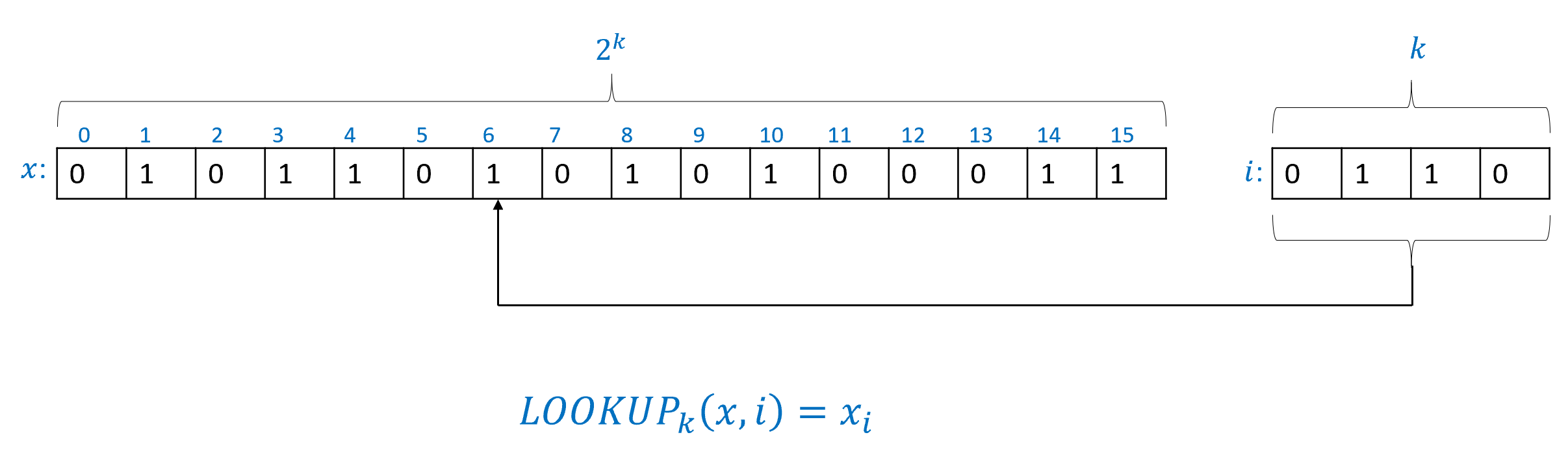
## The LOOKUP function

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

### 

For every , the *lookup* function of order , is defined as follows: For every and ,

where denotes the entry of , using the binary representation to identify with a number in .



The function takes an input in , which we denote by (with and ). The output is : the -th coordinate of , where we identify as a number in using the binary representation. In the above example and . Since is the binary representation of the number , the output of in this case is .

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

# 

For every , there is a NAND-CIRC program that computes the function . Moreover, the number of lines in this program is at most .

An immediate corollary of lookup-thm is that for every , can be computed by a Boolean circuit (with AND, OR and NOT gates) of at most gates.

### Constructing a NAND-CIRC program for

We prove lookup-thm by induction. For the case , maps to . In other words, if then it outputs and otherwise it outputs , which (up to reordering variables) is the same as the function presented in ifstatementsec, which can be computed by a 4-line NAND-CIRC program.

As a warm-up for the case of general , let us consider the case of . Given input for and an index , if the most significant bit of the index is then will equal if and equal if . Similarly, if the most significant bit is then will equal if and will equal if . Another way to say this is that we can write 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 using two invocations of and one invocation of :

### 

For every , is equal to

### 

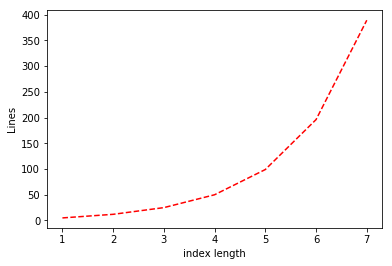
If the most significant bit of is zero, then the index is in and hence we can perform the lookup on the “first half” of and the result of will be the same as . On the other hand, if this most significant bit is equal to , then the index is in , in which case the result of is the same as . Thus we can compute by first computing and and then outputting .

**Proof of** **lookup-thm** **from** **lookup-rec-lem****.** Now that we have lookup-rec-lem, we can complete the proof of lookup-thm. We will prove by induction on that there is a NAND-CIRC program of at most lines for . For this follows by the four line program for we’ve seen before. For , 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 be the number of lines required for , then the above pseudo-code shows that

Since under our induction hypothesis , we get that which is what we wanted to prove. See lookuplinesfig for a plot of the actual number of lines in our implementation of .



The number of lines in our implementation of the LOOKUP\_k function as a function of (i.e., the length of the index). The number of lines in our implementation is roughly .

## 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:

### 

There exists some constant such that for every and function , there is a NAND-CIRC program with at most lines that computes the function .

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

### 

There exists some constant such that for every and function , there is a Boolean circuit with at most gates that computes the function .

*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 NAND-univ-thm and shave an extra factor of , as well as optimize the constant , and so prove that for every , and sufficiently large , if then can be computed by a NAND circuit of at most 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 in tight-upper-bound; see also the biographical notes.

### Proof of NAND’s Universality

To prove NAND-univ-thm, 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., ). mult-bit-ex asks you to extend the proof for all values of . A function can be specified by a table of its values for each one of the inputs. For example, the table below describes one particular function :[[3]](#footnote-64)

An example of a function . { .table #tablefunctiong }

|  |  |
| --- | --- |
| Input () | Output () |
|  | 1 |
|  | 1 |
|  | 0 |
|  | 0 |
|  | 1 |
|  | 0 |
|  | 0 |
|  | 1 |
|  | 0 |
|  | 0 |
|  | 0 |
|  | 0 |
|  | 1 |
|  | 1 |
|  | 1 |
|  | 1 |

For every , , and so the following is NAND-CIRC “pseudocode” to compute 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 and 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 , plugging in the appropriate inputs.

There was nothing about the above reasoning that was particular to the function of tablefunctiong. Given *every* function , we can write a NAND-CIRC program that does the following:

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

The total number of lines in the resulting program is lines for initializing the variables plus the lines that we pay for computing . This completes the proof of NAND-univ-thm.

### 

While NAND-univ-thm 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 can be represented by simply the list of its outputs for each one of the 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 or even fewer lines.

### Improving by a factor of (optional)

By being a little more careful, we can improve the bound of NAND-univ-thm and show that every function can be computed by a NAND-CIRC program of at most lines. In other words, we can prove the following improved version:

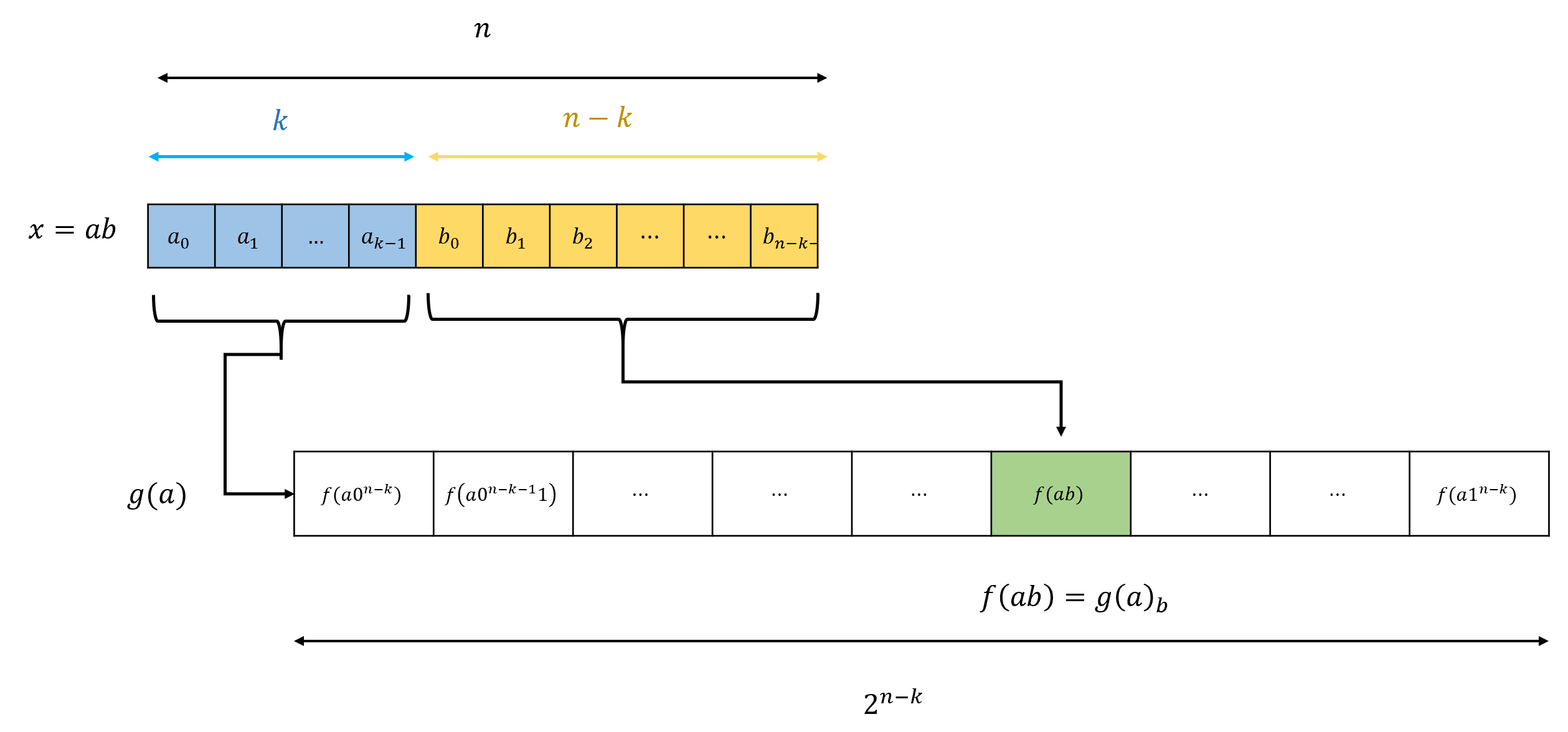
### 

There exists a constant such that for every and function , there is a NAND-CIRC program with at most lines that computes the function .[[4]](#footnote-68)

As before, it is enough to prove the case that . Hence we let , and our goal is to prove that there exists a NAND-CIRC program of lines (or equivalently a Boolean circuit of gates) that computes .

We let (the reasoning behind this choice will become clear later on). We define the function as follows:

In other words, if we use the usual binary representation to identify the numbers with the strings , then for every and

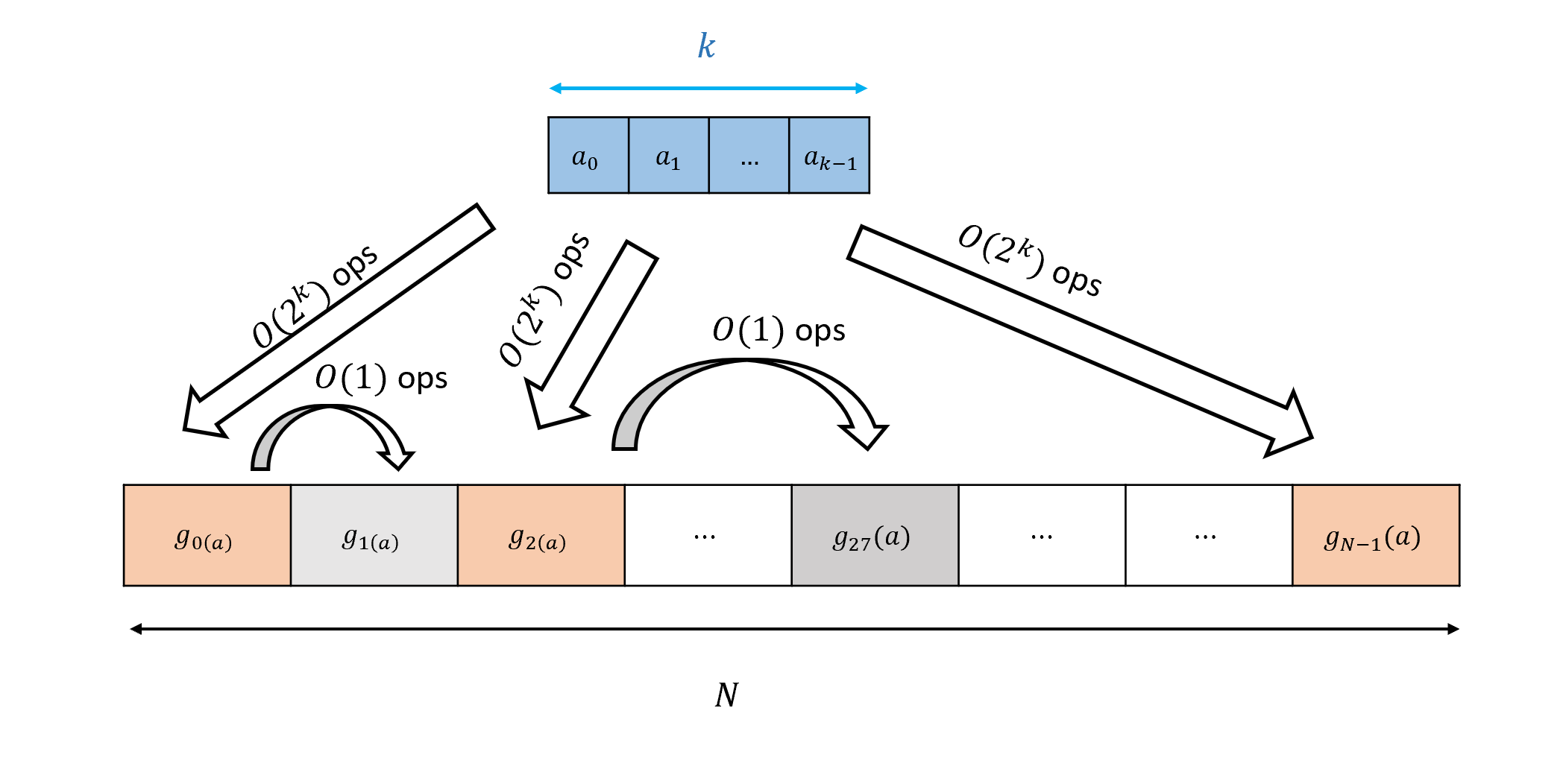


We can compute on input where and by first computing the long string that corresponds to all ’s values on inputs that begin with , and then outputting the -th coordinate of this string.

eqcomputefusinggeffcircuit means that for every , if we write with and then we can compute by first computing the string of length , and then computing to retrieve the element of at the position corresponding to (see efficient\_circuit\_allfuncfig). The cost to compute the is lines/gates and the cost in NAND-CIRC lines (or Boolean gates) to compute is at most

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

To complete the proof we need to give a bound on . Since is a function mapping to , we can also think of it as a collection of functions , where for every and . (That is, is the -th bit of .) Naively, we could use NAND-univ-thm to compute each in lines, but then the total cost is which does not save us anything. However, the crucial observation is that there are only *distinct functions* mapping to . For example, if is an identical function to that means that if we already computed then we can compute using only a constant number of operations: simply copy the same value! In general, if you have a collection of functions mapping to , of which at most are distinct then for every value we can compute the values using at most operations (see computemanyfunctionsfig).



If is a collection of functions each mapping to such that at most of them are distinct then for every , we can compute all the values using at most operations by first computing the distinct functions and then copying the resulting values.

In our case, because there are at most distinct functions mapping to , we can compute the function (and hence by eqcomputefusinggeffcircuit also ) using at most

operations. Now all that is left is to plug into eqboundoncostg our choice of . By definition, , which means that eqboundoncostg can be bounded

which is what we wanted to prove. (We used above the fact that for sufficiently large .)

Using the connection between NAND-CIRC programs and Boolean circuits, an immediate corollary of NAND-univ-thm-improved is the following improvement to circuit-univ-thm:

### 

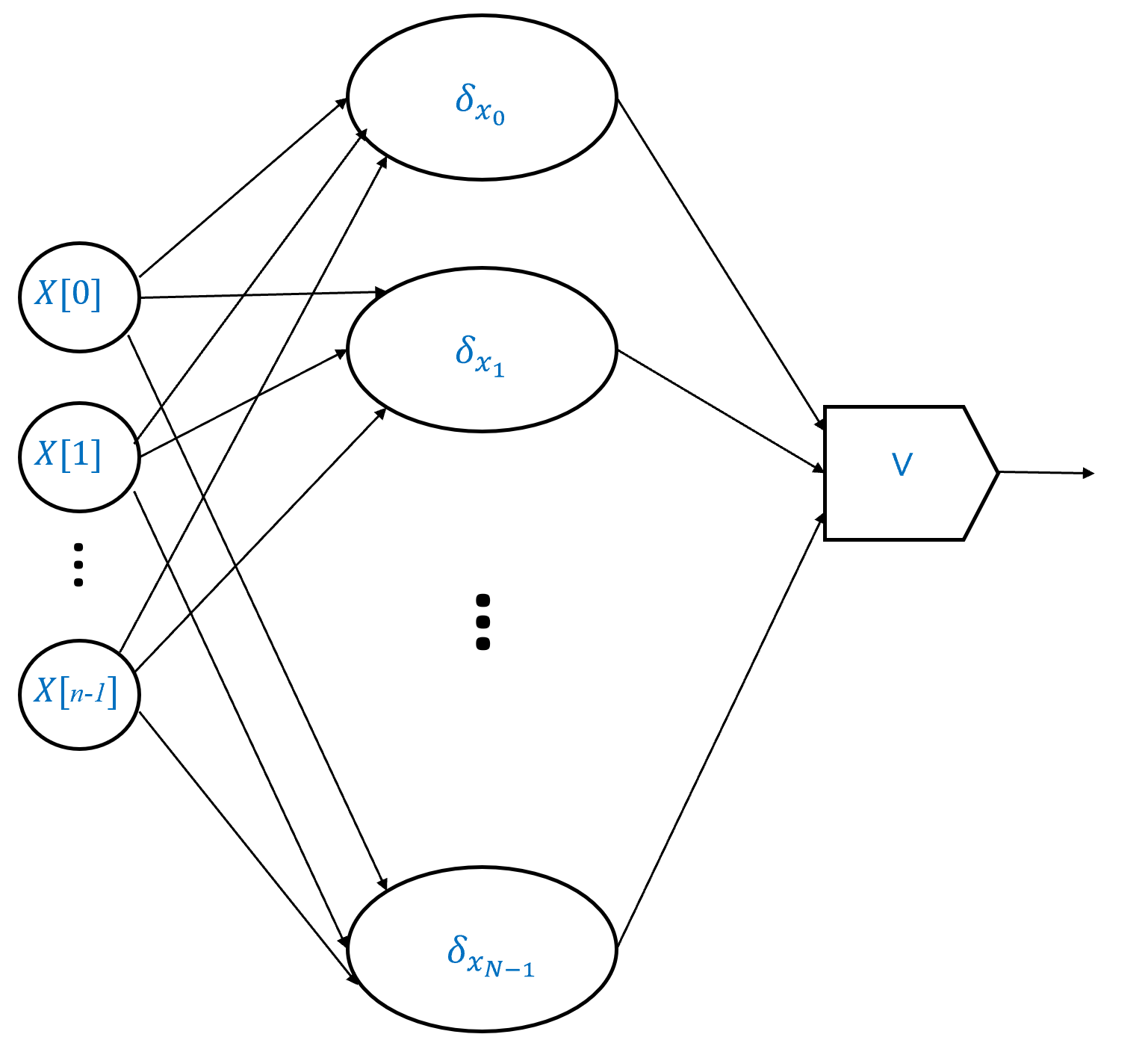
There exists some constant such that for every and function , there is a Boolean circuit with at most gates that computes the function .

## Computing every function: An alternative proof

circuit-univ-thm 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.)

### 

There exists some constant such that for every and function , there is a Boolean circuit with at most gates that computes the function .



Given a function , we let be the set of inputs such that , and note that . We can express as the OR of for where the function (for ) is defined as follows: iff . We can compute the OR of values using two-input OR gates. Therefore if we have a circuit of size to compute for every , we can compute using a circuit of size .

### 

The idea of the proof is illustrated in computeallfuncaltfig. As before, it is enough to focus on the case that (the function has a single output), since we can always extend this to the case of by looking at the composition of circuits each computing a different output bit of the function . We start by showing that for every , there is an sized circuit that computes the function defined as follows: iff (that is, outputs on all inputs except the input ). We can then write any function as the OR of at most functions for the ’s on which .

We prove the theorem for the case . The result can be extended for as before (see also mult-bit-ex). Let . We will prove that there is an -sized Boolean circuit to compute in the following steps:

1. We show that for every , there is an sized circuit that computes the function , where iff .
2. We then show that this implies the existence of an -sized circuit that computes , by writing as the OR of for all such that .

We start with Step 1:

**CLAIM:** For , define as follows:

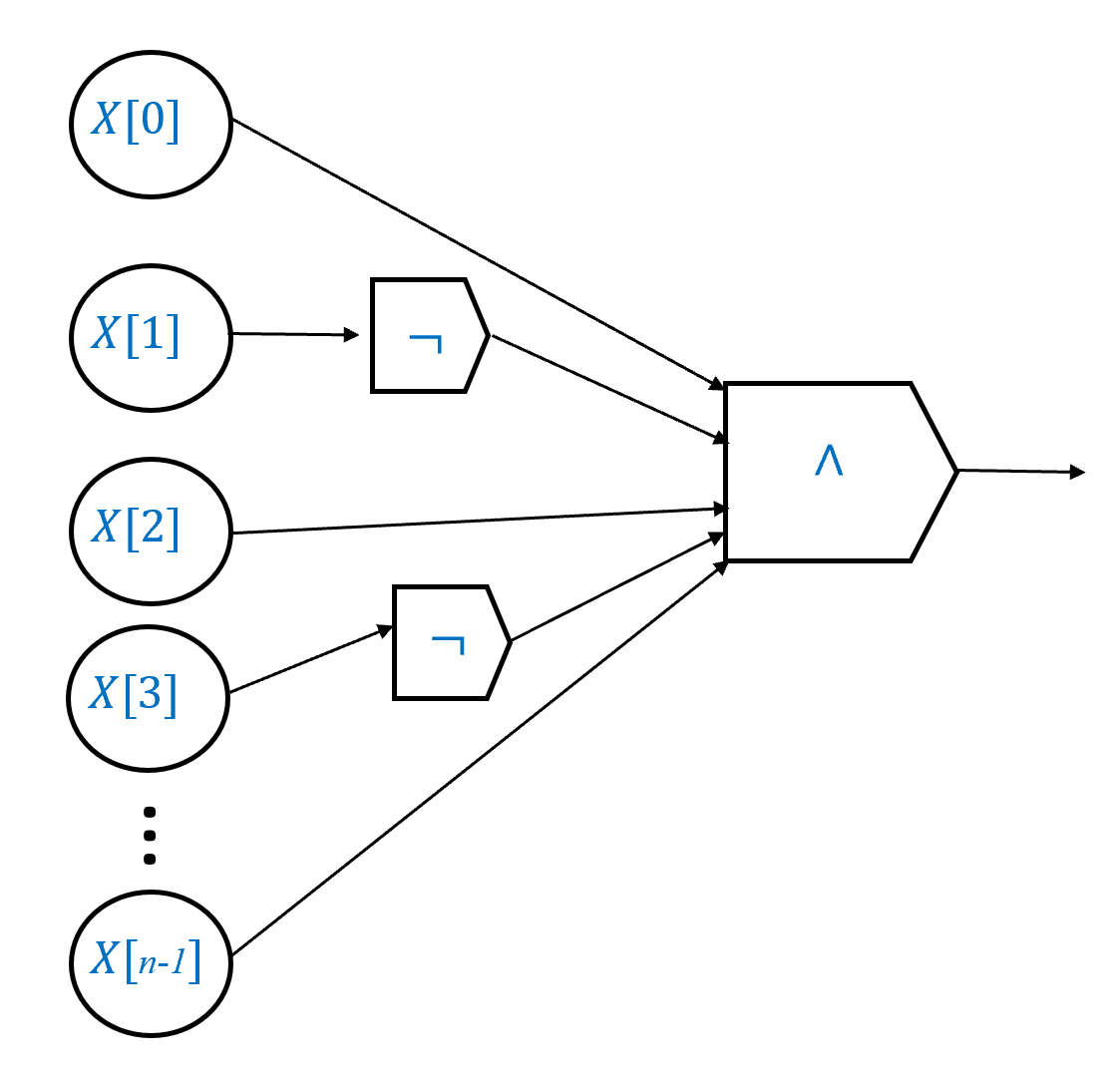
then there is a Boolean circuit using at most gates that computes .

**PROOF OF CLAIM:** The proof is illustrated in deltafuncfig. As an example, consider the function . This function outputs on if and only if , and , and so we can write , which translates into a Boolean circuit with one NOT gate and two AND gates. More generally, for every , we can express as , where if we replace with and if we replace by simply . This yields a circuit that computes using AND gates and at most NOT gates, so a total of at most gates.

Now for every function , we can write

where is the set of inputs on which outputs . (To see this, you can verify that the right-hand side of eqorofdeltafunc evaluates to on if and only if is in the set .)

Therefore we can compute using a Boolean circuit of at most gates for each of the functions and combine that with at most OR gates, thus obtaining a circuit of at most gates. Since , its size is at most and hence the total number of gates in this circuit is .



For every string , there is a Boolean circuit of gates to compute the function such that if and only if . The circuit is very simple. Given input we compute the AND of where if and if . While formally Boolean circuits only have a gate for computing the AND of two inputs, we can implement an AND of inputs by composing two-input ANDs.

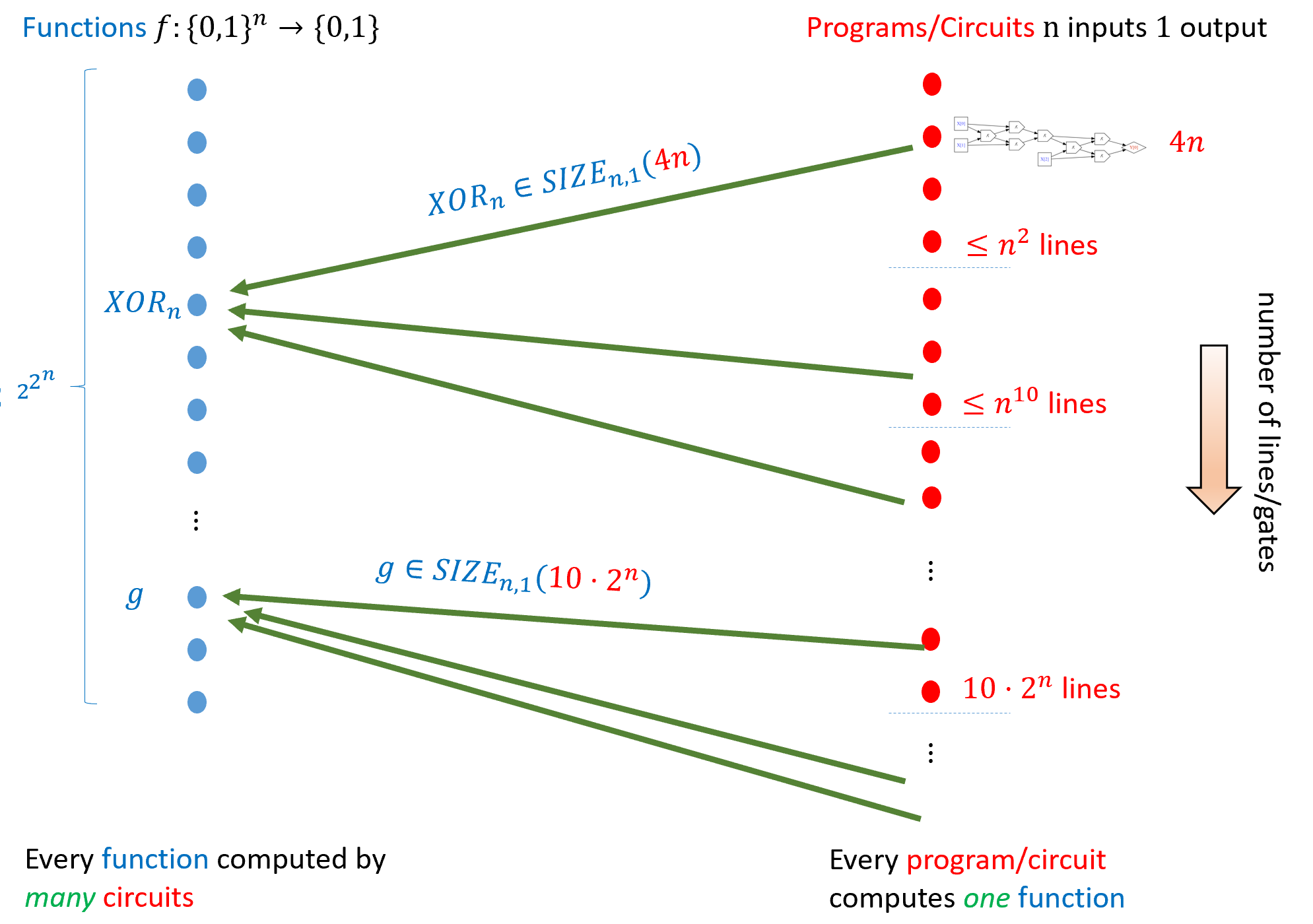
## The class

We have seen that *every* function can be computed by a circuit of size , and *some* functions (such as addition and multiplication) can be computed by much smaller circuits. We define to be the set of functions that can be computed by NAND circuits of at most gates (or equivalently, by NAND-CIRC programs of at most lines). Formally, the definition is as follows:

### 

For every , we let set denotes the set of all functions such that .[[5]](#footnote-83) We denote by the set . For every integer , we let be the set of all functions for which there exists a NAND circuit of at most gates that compute .

funcvscircfig depicts the sets . Note that is a set of *functions*, not of *programs!* (asking if a program or a circuit is a member of is a *category error* as in the sense of cucumberfig). As we discussed in specvsimplrem (and secimplvsspec), 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.



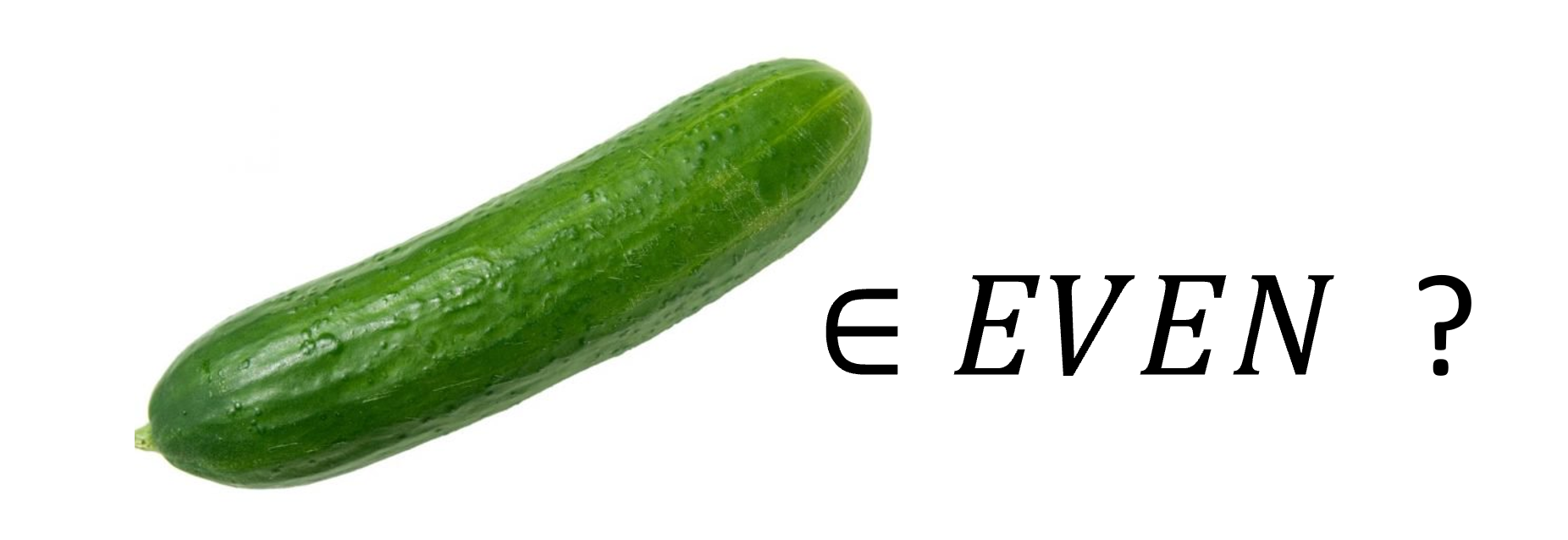
There are functions mapping to , and an infinite number of circuits with bit inputs and a single bit of output. Every circuit computes one function, but every function can be computed by many circuits. We say that if the smallest circuit that computes has or fewer gates. For example . NAND-univ-thm shows that *every* function is computable by some circuit of at most gates, and hence corresponds to the set of *all* functions from to .

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

### 

Let denote the set of all functions that can be computed by an AND/OR/NOT Boolean circuit of at most gates. Then,

If can be computed by a NAND circuit of at most 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 gates that computes . On the other hand, if can be computed by a Boolean AND/OR/NOT circuit of at most gates, then by NANDuniversamthm it can be computed by a NAND circuit of at most gates.



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 is a circuit or program, then asking if is a category error, since is a set of *functions* and not programs or circuits.

The results we have seen in this chapter can be phrased as showing that and . NAND-univ-thm shows that for some constant , is equal to the set of all functions from to .

A NAND-CIRC program can only compute a function with a certain number of inputs and a certain number of outputs. Hence, for example, there is no single NAND-CIRC program that can compute the increment function that maps a string (which we identify with a number via the binary representation) to the string that represents . Rather for every , there is a NAND-CIRC program that computes the restriction of the function to inputs of length . Since it can be shown that for every such a program exists of length at most , for every .

If and , we will write (or sometimes slightly abuse notation and write simply ) to indicate that for every the restriction of to inputs in is in . Hence we can write . We will come back to this issue of finite vs infinite functions later in this course.

In this exercise we prove a certain “closure property” of the class . That is, we show that if is in this class then (up to some small additive term) so is the complement of , which is the function .

Prove that there is a constant such that for every and , if then .

If then there is an -line NAND-CIRC program that computes . We can rename the variable Y[0] in to a variable temp and add the line

Y[0] = NAND(temp,temp)

at the very end to obtain a program that computes .

### 

* We can define the notion of computing a function via a simplified “programming language”, where computing a function in steps would correspond to having a -line NAND-CIRC program that computes .
* While the NAND-CIRC programming only has one operation, other operations such as functions and conditional execution can be implemented using it.
* Every function can be computed by a circuit of at most gates (and in fact at most gates).
* Sometimes (or maybe always?) we can translate an *efficient* algorithm to compute into a circuit that computes with a number of gates comparable to the number of steps in this algorithm.

## Exercises

This exercise asks you to give a one-to-one map from to . 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 is a one-to-one map from to .
2. Show that there is a one-to-one map such that for every , .
3. For every , show that there is a one-to-one map such that for every , .

Prove that the NAND-CIRC program below computes the function (or ) where equals if and equals if :

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

Give a NAND-CIRC program of at most 6 lines to compute the function where iff .

In this exercise we will explore conditionalsugarthm: 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 conditionalsugarthm: a program in a programming language of your choice that transforms a NAND-CIRC-IF program into a “sugar free” NAND-CIRC program that computes the same function. See footnote for hint.[[6]](#footnote-96)
2. Prove the following statement, which is the heart of conditionalsugarthm: suppose that there exists an -line NAND-CIRC program to compute and an -line NAND-CIRC program to compute . Prove that there exist a NAND-CIRC program of at most lines to compute the function where equals if and equals otherwise. (All programs in this item are standard “sugar-free” NAND-CIRC programs.)

1. A *half adder* is the function that corresponds to adding two binary bits. That is, for every , where . Prove that there is a NAND circuit of at most five NAND gates that computes .
2. A *full adder* is the function that takes in two bits and a “carry” bit and outputs their sum. That is, for every , such that . Prove that there is a NAND circuit of at most nine NAND gates that computes .
3. Prove that if there is a NAND circuit of gates that computes , then there is a circuit of gates that computes where (as in addition-thm) is the function that outputs the addition of two input -bit numbers. See footnote for hint.[[7]](#footnote-98)
4. Show that for every there is a NAND-CIRC program to compute with at most lines.

### 

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

### 

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

### 

Write a program using your favorite programming language that on input of an integer , outputs a NAND-CIRC program that computes and has at most lines.[[8]](#footnote-103) What is the smallest number of lines you can use to multiply two 2048 bit numbers?

In the text NAND-univ-thm is only proven for the case . In this exercise you will extend the proof for every .

Prove that

1. If there is an -line NAND-CIRC program to compute and an -line NAND-CIRC program to compute then there is an -line program to compute the function such that .
2. For every function , there is a NAND-CIRC program of at most lines that computes . (You can use the case of NAND-univ-thm, as well as Item 1.)

Let 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 with at most three lines of code that uses both NAND as well as the syntactic sugar OR that computes the same function as .
2. Draw a circuit that computes the same function as and uses only and gates.

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

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

*and*

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

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 and , and the function such that equals if and equals if ). Compare the power of the NAND-CIRC programming language and the IF-CIRC programming language.

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 , and the function that maps to ). Compare the power of the NAND-CIRC programming language and the XOR-CIRC programming language. See footnote for hint.[[9]](#footnote-107)

Prove that there is some constant such that for every , where is the majority function on input bits. That is iff . See footnote for hint.[[10]](#footnote-109)

Prove that there is some constant such that for every , and integers , there is a NAND circuit with at most gates that computes the *threshold* function that on input outputs if and only if .

## Bibliographical notes

See Jukna’s and Wegener’s books [@Jukna12, @wegener1987complexity] for much more extensive discussion on circuits. Shannon showed that every Boolean function can be computed by a circuit of exponential size [@Shannon1938]. The improved bound of (with the optimal value of for many bases) is due to Lupanov [@Lupanov1958]. An exposition of this for the case of NAND (where ) is given in Chapter 4 of his book [@lupanov1984]. (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](https://babeljs.io/) 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](https://babeljs.io/docs/plugins/) architecture, that allows users to add their own syntactic sugar to the language.

1. This code uses *regular expressions* to make the search and replace parts a little easier. We will see the theoretical basis for regular expressions in restrictedchap. [↑](#footnote-ref-35)
2. The value of can be improved to , see halffulladderex. [↑](#footnote-ref-45)
3. In case you are curious, this is the function on input (which we interpret as a number in ), that outputs the -th digit of in the binary basis. [↑](#footnote-ref-64)
4. The constant in this theorem is at most and in fact can be arbitrarily close to , see computeeveryfunctionbibnotes. [↑](#footnote-ref-68)
5. The restriction that makes no difference; see nandcircsizeex. [↑](#footnote-ref-83)
6. You can start by transforming into a NAND-CIRC-PROC program that uses procedure statements, and then use the code of desugarcode to transform the latter into a “sugar free” NAND-CIRC program. [↑](#footnote-ref-96)
7. Use a “cascade” of adding the bits one after the other, starting with the least significant digit, just like in the elementary-school algorithm. [↑](#footnote-ref-98)
8. **Hint:** Use Karatsuba’s algorithm. [↑](#footnote-ref-103)
9. You can use the fact that . In particular it means that if you have the lines d = XOR(a,b) and e = XOR(d,c) then e gets the sum modulo of the variable a, b and c. [↑](#footnote-ref-107)
10. One approach to solve this is using recursion and the so-called [Master Theorem](https://en.wikipedia.org/wiki/Master%5Ftheorem%5F(analysis%5Fof%5Falgorithms)). [↑](#footnote-ref-109)