Learning Objectives:

- See the NAND« programming language.
- Understand how NAND« can be implemented as syntactic sugar on top of NAND++
- Understand the construction of a *universal* NAND« (and hence NAND++) program.

8

Indirection and universality

> *"All problems in computer science can be solved by another level of indirection"*, attributed to David Wheeler.

> "The programmer is in the unique position that ... he has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before.", Edsger Dijkstra, "On the cruelty of really teaching computing science", 1988.

One of the most significant results we showed for NAND programs is the notion of *universality*: that a NAND program can evaluate other NAND programs. However, there was a significant caveat in this notion. To evaluate a NAND program of s lines, we needed to use a bigger number of lines than s. It turns out that NAND++ allows us to "break out of this cycle" and obtain a truly *universal NAND++* program U that can evaluate all other programs, including programs that have more lines than U itself. (As we'll see in the next lecture, this is not something special to NAND++ but is a feature of many other computational models.) The existence of such a universal program has far reaching applications, and we will explore them in the rest of this course.

To describe the universal program, it will be convenient for us to introduce some extra "syntactic sugar" for NAND++. We'll use the name NAND« for the language of NAND++ with this extra syntactic sugar. The classes of functions computable by NAND++ and NAND« programs are identical, but NAND« can sometimes be more convenient to work with. Moreover, NAND« will be useful for

us later in the course when we will turn to modelling *running time* of algorithms.[1]

8.1 The NAND« programming language

We now define a seemingly more powerful programming language than NAND++: NAND« (pronounced *"NAND shift"*). NAND« has some additional operators, but as we will see, it can ultimately be implemented by applying certain "syntactic sugar" constructs on top of NAND++. Nonetheless, NAND« will still serve (especially later in the course) as a useful computational model.[2] There are two key differences between NAND« and NAND:

1. The NAND« programming language works with *integer valued* as opposed to *binary* variables.
2. NAND« allows *indirection* in the sense of accessing the `bar`-th location of an array `foo`. Specifically, since we use *integer valued* variables, we can assign the value of `bar` to the special index `i` and then use `foo_i`.

   We will allow the following operations on variables:[3]

- `foo := bar` or `i := bar` (assignment)
- `foo := bar + baz` (addition)
- `foo := bar - baz` (subtraction)
- `foo := bar » baz` (right shift: $foo \leftarrow \lfloor bar \cdots 2^{-baz} \rfloor$)
- `foo := bar « baz` (left shift: $foo \leftarrow bar \cdots 2^{baz}$)
- `foo := bar % baz` (modular reduction)
- `foo := bar * baz` (multiplication)
- `foo := bar / baz` (integer division: $foo \leftarrow \lfloor \frac{bar}{baz} \rfloor$)
- `foo := bar bAND baz` (bitwise AND)
- `foo := bar bXOR baz` (bitwise XOR)
- `foo := bar > baz` (greater than)
- `foo := bar < baz` (smaller than)
- `foo := bar == baz` (equality)

The semantics of these operations are as expected except that we maintain the invariant that all variables always take values between

[1] Looking ahead, as we will see in the next lecture, NAND++ programs are essentially equivalent to *Turing Machines* (more precisely, their single-tape, oblivious variant), while NAND« programs are equivalent to *RAM machines*. Turing machines are typically the standard model used in computability and complexity theory, while RAM machines are used in algorithm design. As we will see, their powers are equivalent up to polynomial factors in the running time.

[2] If you have encountered computability or computational complexity before, we can already "let you in on the secret". NAND++ is equivalent to the model known as *single tape oblivious Turing machines*, while NAND« is (essentially) equivalent to the model known as *RAM machines*. For the purposes of the current lecture, the NAND++/Turing-Machine model is indistinguishable from the NAND«/RAM-Machine model (due to a notion known as "Turing completeness") but the difference between them can matter if one is interested in a fine enough resolution of computational efficiency.

[3] Below `foo`, `bar` and `baz` are indexed or non-indexed variable identifiers (e.g., they can have the form `blah` or `blah_12` or `blah_i`), as usual, we identify an indexed identifier `blah` with `blah_0`. Except for the assignment, where `i` can be on the lefthand side, the special index variable `i` cannot be involved in these operations.

0 and the current value of the iteration counter (i.e., number of iterations of the program that have been completed). If an operation would result in assigning to a variable `foo` a number that is smaller than 0, then we assign 0 to `foo`, and if it assigns to `foo` a number that is larger than the iteration counter, then we assign the value of the iteration counter to `foo`. Just like C, we interpret any nonzero value as "true" or 1, and hence `foo := bar NAND baz` will assign to `foo` the value 0 if both `bar` and `baz` are not zero, and 1 otherwise.

Apart from those operations, NAND« is identical to NAND++. For consistency, we still treat the variable `i` as special, in the sense that we only allow it to be used as an index, even though the other variables contain integers as well, and so we don't allow variables such as `foo_bar` though we can simulate it by first writing `i := bar` and then `foo_i`. We also maintain the invariant that at the beginning of each iteration, the value of `i` is set to the same value that it would have in a NAND++ program (i.e., the function of the iteration counter stated in Exercise 7.1), though this can be of course overwritten by explicitly assigning a value to `i`. Once again, see the appendix for a more formal specification of NAND«.

> **R** **Computing on integers** Most of the time we will be interested in applying NAND« programs on bits, and hence we will assume that both inputs and outputs are bits. We can enforce the latter condition by not allowing `y_` variables to be on the lefthand side of any operation other than NAND. However, the same model can be used to talk about functions that map tuples of integers to tuples of integers, and so we may very occasionally abuse notation and talk about NAND« programs that compute on integers.

8.1.1 Simulating NAND« in NAND++

The most important fact we need to know about NAND« is that it can be implemented by mere "syntactic sugar" and hence does not give us more computational power than NAND++, as stated in the following theorem:

Theorem 8.1 — NAND++ and NAND« are equivalent. For every (partial) function $F : \{0,1\}^* \to \{0,1\}^*$, F is computable by a NAND++ program if and only if F is computable by a NAND« program.

The rest of this section is devoted to outlining the proof of Theo-

rem 8.1. The "only if" direction of the theorem is immediate. After all, every NAND++ program P is in particular also a NAND« program, and hence if F is computable by a NAND++ program then it is also computable by a NAND« program. To show the "if" direction, we need to show how we can implement all the operations of NAND« in NAND++. In other words, we need to give a "NAND« to NAND++ compiler".

Writing a compiler in full detail, and then proving that it is correct, is possible (and has been done) but is quite a time consuming enterprise, and not very illuminating. For our purposes, we need to convince ourselves that Theorem 8.1 and that such a transformation exists, and we will do so by outlining the key ideas behind it.[4]

[4] The webpage nandpl.org should eventually contain a program that transforms a NAND« program into an equivalent NAND++ program.

Let P be a NAND« program, we need to transform P into a NAND++ program P' that computes the same function as P. The idea will be that P' will simulate P "in its belly". We will use the variables of P' to encode the state of the simulated program P, and every single NAND« step of the program P will be translated into several NAND++ steps by P'. We will do so in several steps:

Step 1: Controlling the index and inner loops. We have seen that we can add syntactic sugar for inner loops and incrementing/decrementing the index (i.e., operations such as `i++ (foo)` and `i- (bar)`) to NAND++. Hence we can assume access to these operations in constructing P'. In particular, we can use such an inner loop to perform tasks such as copying the contents of one array (e.g., variables `foo_0`, ..., `foo_`$\langle k-1 \rangle$ for some k) to another.

Step 2: Operations on integers. We can use some standard prefix free encoding to represent an integer as an array of bits. For example, we can use the map $pf : \mathbb{Z} \to \{0,1\}^*$ defined as follows. Given an integer $z \in \mathbb{Z}$, if $z \geq 0$ then we define the string $pf(z)$ as $z_0 z_0 z_1 z_1 \ldots z_{n-1} z_{n-1} 01$ (where n is the smallest number s.t. $2^n > z$ and z_i is the binary digit of x corresponding to 2^i). If $z < 0$ then we define $pf(z) = 10 pf(|z|)$. We can then use the standard gradeschool algorithms to define NAND++ macros that perform arithmetic operations on the representation of integers (e.g., addition, multiplication, division, etc.).

Step 3: Move index to specified location. We can use the above operations to move the index to a location encoded by an integer. To do so, we can first move `i` to the zero location by decreasing it until the `atstart_i` equals 1 (where, as we've seen before, `atstart` is an array we can set up so `attstart_0` is 1 and `atstart_`$\langle j \rangle$ is zero for $j \neq 0$). Now, if the variables `foo_0`,`foo_1`,... encode the number

$k \in \mathbb{N}$ we can set the value of `i` to k as follows. We'll set `bar` to 1 and an inner loop that will proceed as long as `bar` is not zero. In this loop we will do the following: **(1)** If `foo` encodes 0 then set `bar` to zero. **(2)** Otherwise, we use a nested inner loop to decrement the number represented by `foo` by 1, and perform the operation `i++ (bar)`.

Step 4: Maintaining an iteration counter and index. The NAND++ program P' will simulate execution of the NAND« program P. Every step of P will be simulated by several steps of P'. We can use the above operations to maintain a variable `itercounter` and `index` that will encode the current step of P that is being executed and the current value of the special index variable `i` in the simulated program P (which does not have to be the same as the value of `i` in the NAND++ program P').

Step 5: Embedding two dimensional arrays into one dimension. If `foo` and `bar` the encode the natural numbers $x, y \in \mathbb{N}$, then we can use NAND++ to compute the map $PAIR : \mathbb{N}^2 \rightarrow \mathbb{N}$ where $PAIR(x,y) = \frac{1}{2}(x+y)(x+y+1)+x$. In Exercise 8.1 we ask you to verify that $PAIR$ is a one-to-one map from $\mathbb{N}^2$ to $\mathbb{N}$ and that there are NAND++ programs P_0, P_1 such that for every $x_0, x_1 \in \mathbb{N}$ and $i \in \{0,1\}$, $P_i(PAIR(x_0,x_1)) = x_i$. Using this $PAIR$ map, we can assume we have access to two dimensional arrays in our NAND++ program.

Step 6: Embedding an array of integers into a two dimensional bit array. We can use the same encoding as above to embed a one-dimensional array `foo` of integers into a two-dimensional array `bar` of bits, where `bar_`$\langle i \rangle$`,` $\langle j \rangle$ will encode the j-th bit in the representation of the integer `foo_`$\langle i \rangle$. Thus we can simulate the integer arrays of the NAND« program P in the NAND++ program P'.

Step 7: Simulating P. Now we have all the components in place to simulate every operation of P in P'. The program P' will have a two dimensional bit array corresponding to any one dimensional array of P, as well as variables to store the iteration counter, index, as well as the `loop` variable of the simulated program P. Every step of P can now be translated into an inner loop that would perform the same operation on the representations of the state.

We omit the full details of all the steps above and their analysis, which are tedious but not that insightful.

8.1.2 Example

Here is a program that computes the function $PALINDROME : \{0,1\}^* \to \{0,1\}$ that outputs 1 on x if and only if $x_i = x_{|x|-i}$ for every $i \in \{0,\ldots,|x|-1\}$. This program uses NAND« with the syntactic sugar we described before, but as discussed above, we can transform it into a NAND++ program.

```
// A sample NAND<< program that computes the language of
    palindromes
// By Juan Esteller
def a := NOT(b) {
  a := b NAND b
}
o := NOT(z)
two := o + o
if(NOT(seen_0)) {
  cur := z
  seen_0 := o
}
i := cur
if(validx_i) {
 cur := cur + o
 loop := o
}
if(NOT(validx_i)) {
  computedlength := o
}
if(computedlength) {
  if(justentered) {
    justentered := o
    iter := z
  }
  i := iter
  left := x_i
  i := (cur - iter) - o
  right := x_i
  if(NOT(left == right)) {
    loop := z
    y_0 := z
  }
  halflength := cur / two
  if(NOT(iter < halflength)) {
   y_0 := o
```

```
  loop := z
 }
 iter := iter + o
}
```

8.2 The "Best of both worlds" paradigm

The equivalence between NAND++ and NAND« allows us to choose the most convenient language for the task at hand:

- When we want to give a theorem about all programs, we can use NAND++ because it is simpler and easier to analyze. In particular, if we want to show that a certain function *can not* be computed, then we will use NAND++.
- When we want to show the existence of a program computing a certain function, we can use NAND«, because it is higher level and easier to program in. In particular, if we want to show that a function *can* be computed then we can use NAND«. In fact, because NAND« has much of the features of high level programming languages, we will often describe NAND« programs in an informal manner, trusting that the reader can fill in the details and translate the high level description to the precise program. (This is just like the way people typically use informal or "pseudocode" descriptions of algorithms, trusting that their audience will know to translate these descriptions to code if needed.)

Our usage of NAND++ and NAND« is very similar to the way people use in practice high and low level programming languages. When one wants to produce a device that executes programs, it is convenient to do so for very simple and "low level" programming language. When one wants to describe an algorithm, it is convenient to use as high level a formalism as possible.

R **Recursion in NAND«** One high level tool we can use in describing NAND« programs is *recursion*. We can use the standard implementation of the stack data structure, which can be (and in fact is) used to implement recursion. A *stack* is a data structure containing a sequence of elements, where we can "push" elements into it and "pop" them from it in "first in last out" order. We can implement `stack` by an array of integers `stack_0`, ..., `stack_`$\langle k - 1 \rangle$ and `stackpointer` will be the number k of items in the stack. We implement `push(foo)` by doing `i`

Figure 8.1: By having the two equivalent languages NAND++ and NAND«, we can "have our cake and eat it too", using NAND++ when we want to prove that programs *can't* do something, and using NAND« or other high level languages when we want to prove that programs *can* do something.

`:= stackpointer` and `stack_i := foo` and `pop()` by letting `stackpointer := stackpointer - 1`. By encoding strings as integers, we can have allow strings in our stack as well.

Now we can implement recursion using the stack just as is done in most programming languages. The idea is that a (recursive or non recursive) call to a function F is implemented by pushing the arguments for F into the stack. The code of F will "pop" the arguments from the stack, perform the computation (which might involve making recursive or non recursive calls) and then "push" its return value into the stack. Because of the "first in last out" nature of a stack, we do not return control to the calling procedure until all the recursive calls are done.

Specifically, we note that using loops and conditionals, we can implement "goto" statements in NAND«. Moreover, we can even implement "dynamic gotos", in the sense that we can set integer labels for certain lines of codes, and have a `goto foo` operation that moves execution to the line labeled by `foo`. Now, if we want to make a call to a function F with parameter `bar` then we will push into the stack the label of the next line and `bar`, and then make a `goto` to the code of F. That code will pop its parameter from the

stack, do the computation of F, and when it needs to resume execution, will pop the label from the stack and `goto` there.

You can find online a tutorial on how recursion is implemented via stack in your favorite programming language, whether it's Python , JavaScript, or Lisp/Scheme.

8.2.1 Let's talk about abstractions.

At some point in any theory of computation course, the instructor and students need to have *the talk*. That is, we need to discuss the *level of abstraction* in describing algorithms. In algorithms courses, one typically describes algorithms in English, assuming readers can "fill in the details" and would be able to convert such an algorithm into an implementation if needed. For example, we might describe the breadth first search algorithm to find if two vertices u, v are connected as follows:

1. Put u in queue Q.
2. While Q is not empty:

- Remove the top vertex w from Q
- If $w = v$ then declare "connected" and exit.
- Mark w and add all unmarked neighbors of w to Q.

1. Declare "unconnected".

We call such a description a *high level description*.

If we wanted to give more details on how to implement breadth first search in a programming language such as Python or C (or NAND« / NAND++ for that matter), we would describe how we implement the queue data structure using an array, and similarly how we would use arrays to implement the marking. We call such a "intermediate level" description an *implementation level* or *pseudocode* description. Finally, if we want to describe the implementation precisely, we would give the full code of the program (or another fully precise representation, such as in the form of a list of tuples). We call this a *formal* or *low level* description.

While initially we might have described NAND, NAND++, and NAND« programs at the full formal level (and the NAND website contains more such examples), as the course continues we will move to implementation and high level description. After all, our focus is

typically not to use these models for actual computation, but rather to analyze the general phenomenon of computation. That said, if you don't understand how the high level description translates to an actual implementation, you should always feel welcome to ask for more details of your teachers and teaching fellows.

A similar distinction applies to the notion of *representation* of objects as strings. Sometimes, to be precise, we give a *low level specification* of exactly how an object maps into a binary string. For example, we might describe an encoding of n vertex graphs as length n^2 binary strings, by saying that we map a graph G over the vertex $[n]$ to a string $x \in \{0,1\}^{n^2}$ such that the $n \cdot i + j$-th coordinate of x is 1 if and only if the edge $\overrightarrow{i\,j}$ is present in G. We can also use an *intermediate* or *implementation level* description, by simply saying that we represent a graph using the adjacency matrix representation. Finally, because we translating between the various representations of graphs (and objects in general) can be done via a NAND« (and hence a NAND++) program, when talking in a high level we also suppress discussion of representation altogether. For example, the fact that graph connectivity is a computable function is true regardless of whether we represent graphs as adjacency lists, adjacency matrices, list of edge-pairs, and so on and so forth. Hence, in cases where the precise representation doesn't make a difference, we would often talk about our algorithms as taking as input an object O (that can be a graph, a vector, a program, etc.) without specifying how O is encoded as a string.

8.3 Universality: A NAND++ interpreter in NAND++

Like a NAND program, a NAND++ or a NAND« program is ultimately a sequence of symbols and hence can obviously be represented as a binary string. We will spell out the exact details of representation later, but as usual, the details are not so important (e.g., we can use the ASCII encoding of the source code). What is crucial is that we can use such representation to evaluate any program. That is, we prove the following theorem:

Theorem 8.2 — Universality of NAND++. There is a NAND++ program U that computes the partial function $EVAL : \{0,1\}^* \to \{0,1\}^*$ defined as follows:

$$EVAL(P,x) = P(x) \tag{8.1}$$

for strings P, x such that P is a valid representation of a NAND++

program which produces an output on x. Moreover, for every input $x \in \{0,1\}^*$ on which P does not halt, $U(P,x)$ does not halt as well.

This is a stronger notion than the universality we proved for NAND, in the sense that we show a *single* universal NAND++ program U that can evaluate *all* NAND programs, including those that have more lines than the lines in U. In particular, U can even be used to evaluate itself! This notion of *self reference* will appear time and again in this course, and as we will see, leads to several counterintuitive phenomena in computing.

Because we can easily transform a NAND« program into a NAND++ program, this means that even the seemingly "weaker" NAND++ programming language is powerful enough to simulate NAND« programs. Indeed, as we already alluded to before, NAND++ is powerful enough to simulate also all other standard programming languages such as Python, C, Lisp, etc.

8.3.1 Representing NAND++ programs as strings

Before we can prove Theorem 8.2, we need to make its statement precise by specifying a representation scheme for NAND++ programs. As mentioned above, simply representing the program as a string using ASCII or UTF-8 encoding will work just fine, but we will use a somewhat more convenient and concrete representation, which is the natural generalization of the "list of triples" representation for NAND programs. We will assume that all variables are of the form `foo_##` where `foo` is an identifier and `##` is some number or the index `i`. If a variable `foo` does not have an index then we add the index zero to it. We represent an instruction of the form

`foo_`$\langle j \rangle$ `:= bar_`$\langle k \rangle$ `NAND baz_`$\langle \ell \rangle$

as a 6 tuple (a, j, b, k, c, ℓ) where a, b, c are numbers corresponding to the labels `foo`,`bar`,and `baz` respectively, and j, k, ℓ are the corresponding indices. For variables that indexed by the special index `i`, we will encode the index by s, where s ithe number of lines in the program. (There is no risk of conflict since we did not allow numerical indices larger or equal to the number of lines in the program.) We will set the identifiers of `x`,`y`,`validx` and `loop` to $0, 1, 2, 3$ respectively. Therefore the representation of the parity program

```
tmp_1 := seen_i NAND seen_i
tmp_2 := x_i NAND tmp_1
```

```
val := tmp_2 NAND tmp_2
ns := s NAND s
y_0 := ns NAND ns
u := val NAND s
v := s NAND u
w := val NAND u
s := v NAND w
seen_i := z NAND z
stop := validx_i NAND validx_i
loop := stop NAND stop
```

will be

```
[[4, 1, 5, 11, 5, 11],
 [4, 2, 0, 11, 4, 1],
 [6, 0, 4, 2, 4, 2],
 [7, 0, 8, 0, 8, 0],
 [1, 0, 7, 0, 7, 0],
 [9, 0, 6, 0, 8, 0],
 [10, 0, 8, 0, 9, 0],
 [11, 0, 6, 0, 9, 0],
 [8, 0, 10, 0, 11, 0],
 [5, 11, 12, 0, 12, 0],
 [13, 0, 2, 11, 2, 61],
 [3, 0, 13, 0, 13, 0]]
```

Binary encoding: The above is a way to represent any NAND++ program as a list of numbers. We can of course encode such a list as a binary string in a number of ways. For concreteness, since all the numbers involved are between 0 and s (where s is the number of lines), we can simply use a string of length $6\lceil \log(s+1) \rceil$ to represent them, starting with the prefix $0^{s+1}1$ to encode s. For convenience we will assume that any string that is not formatted in this way encodes the single line program `y_0 := x_0 NAND x_0`. This way we can assume that every string $P \in \{0,1\}^*$ represents *some* NAND++ program.

8.3.2 A NAND++ interpreter in NAND«

Here is the "pseudocode"/"sugar added" version of an interpreter for NAND++ programs (given in the list of 6 tuples representation) in NAND«. We assume below that the input is given as integers `x_0`,...,`x_`$\langle 6 \cdot lines - 1 \rangle$ where *lines* is the number of lines in the program. We also assume that `NumberVariables` gives some upper

bound on the total number of distinct non-indexed identifiers used in the program (we can also simply use *lines* as this bound).

```
simloop := 3
totalvars := NumberVariables(x)
maxlines := Length(x) / 6
currenti := 0
currentround := 0
increasing := 1
pc := 0
while (true) {
   line := 0
   foo := x_{6*line + 0}
   fooidx := x_{6*line + 1}
   bar := x_{6*line + 2}
   baridx := x_{6*line + 3}
   baz := x_{6*line + 4}
   bazidx := x_{6*line + 5}
   if (fooidx == maxlines) {
      fooidx := currenti
   }
   ... // similar for baridx, bazidx

   vars_{totalvars*fooidx+foo} := vars_{totalvars*baridx+bar
       } NAND vars_{totalvars*bazidx+baz}
   line++

   if line==maxlines {
      if not avars[simloop] {
         break
      }
      pc := pc+1
      if (increasing) {
         i := i + 1
      } else
      {
         i := i - 1
      }
      if i>r {
         increasing := 0
         r := r+1
      }
      if i==0 {
         increasing := 1
      }
```

```
    }
    // keep track in loop above of largest m that y_{m-1} was
         assigned a value
    // add code to move vars[0*totalvars+1]...vars[(m-1)*
        totalvars+1] to y_0..y_{m-1}
}
```

Since we can transform *every* NAND« program to a NAND++ one, we can also implement this interpreter in NAND++, hence completing the proof of Theorem 8.2.

8.3.3 A Python interpreter in NAND++

At this point you probably can guess that it is possible to write an interpreter for languages such as C or Python in NAND« and hence in NAND++ as well. After all, with NAND++ / NAND« we have access to an unbounded array of memory, which we can use to simulate memory allocation and access, and can do all the basic computation steps offered by modern CPUs. Writing such an interpreter is nobody's idea of a fun afternoon, but the fact it can be done gives credence to the belief that NAND++ *is* a good model for general-purpose computing.

8.4 Lecture summary

- NAND++ programs introduce the notion of *loops*, and allow us to capture a single algorithm that can evaluate functions of any length.
- NAND« programs include more operations, including the ability to use indirection to obtain random access to memory, but they are computationally equivalent to NAND++ program.
- We can translate many (all?) standard algorithms into NAND« and hence NAND++ programs.
- There is a *universal* NAND++ program U such that on input a description of a NAND++ program P and some input x, $U(P,x)$ halts and outputs $P(x)$ if (and only if) P halts on input x.

8.5 Exercises

Exercise 8.1 — Pairing. Let $PAIR : \mathbb{N}^2 \to \mathbb{N}$ be the function defined as $PAIR(x_0, x_1) = \frac{1}{2}(x_0 + x_1)(x_0 + x_1 + 1) + x_1$.

1. Prove that for every $x^0, x^1 \in \mathbb{N}$, $PAIR(x^0, x^1)$ is indeed a natural number.

2. Prove that $PAIR$ is one-to-one

3. Construct a NAND++ program P such that for every $x^0, x^1 \in \mathbb{N}$, $P(pf(x^0)pf(x^1)) = pf(PAIR(x^0, x^1))$, where pf is the prefix-free encoding map defined above. You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.

4. Construct NAND++ programs P_0, P_1 such that for for every $x^0, x^1 \in \mathbb{N}$ and $i \in N$, $P_i(pf(PAIR(x^0, x^1))) = pf(x^i)$. You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter. ■

Exercise 8.2 — Single vs multiple bit. Prove that for every $F : \{0,1\}^* \to \{0,1\}^*$, the function F is computable if and only if the following function $G : \{0,1\}^* \to \{0,1\}$ is computable, where G is defined as follows: $G(x, i, \sigma) = \begin{cases} F(x)_i & i < |F(x)|, \sigma = 0 \\ 1 & i < |F(x)|, \sigma = 1 \\ 0 & i \geq |F(x)| \end{cases}$ ■

8.6 Bibliographical notes

The notion of "NAND++ programs" we use is nonstandard but (as we will see) they are equivalent to standard models used in the literature. Specifically, NAND++ programs are closely related (though not identical) to *oblivious one-tape Turing machines*, while NAND« programs are essentially the same as RAM machines. As we've seen in these lectures, in a qualitative sense these two models are also equivalent to one another, though the distinctions between them matter if one cares (as is typically the case in algorithms research) about polynomial factors in the running time.

8.7 Further explorations

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

8.8 Acknowledgements