# *Appendix: The NAND* Programming Languages*

In this appendix we give a more formal specification of the NAND, NAND++ and NAND« programming languages. See the website http://nandpl.org for more information about these languages.

Note that the NAND programming language corresponds to Boolean circuits (with the NAND basis), the NAND++ programming language roughly corresponds to one-tape oblivious Turing machines, and the NAND« programming language roughly corresponds to RAM machines.

## *23.11 NAND programming language specification*

### *23.11.1 Syntax of NAND programs*

An *unindexed variable identifier* in NAND is a sequence of upper and lower case letters optionally ending with one or more occurrences of the prime symbol `'`, and that is not one of the following disallowed names: `loop`,`validx`, and `i`. For example, the following are valid unindexed variable identifiers: `foo`,`bar'`,`BaZ"'`. Invalid unindexed variable identifier include `foo33`,`bo'az`, `Hey!`,`bar_22blah` and `i`.

An *indexed variable identifier* has the form *var*`_`*num* where *var* is an unindexed variable identifier and *num* is a sequence of the digits `0-9`. For example, the following are valid indexed variable identifiers: `foo_19`, `bar'_73` and `Baz"'_195`. A *variable identifier* is either an indexed or an unidexed variable identifier. Thus both `foo` and `BaZ"_68` are valid variable identifiers.

A NAND program consists of a finite sequence of lines of the form

*vara* `:=` *varb* `NAND` *varc*

where *vara*, *varb*, *varc* are variable identifiers.

Variables of the form `x` or `x_`$\langle i \rangle$ can only appear on the righthand side of the `:=` operator and variables of the form `y` or `y_`$\langle i \rangle$ can only appear on the lefthand side of the `:=` operator. The *number of inputs* of a NAND program $P$ equals one plus the largest number $i$ such that a variable of the form `x_`$\langle i \rangle$ appears in the program, while the number of outputs of a NAND program equals one plus the largest number $j$ such that a variable of the form `y_`$\langle j \rangle$ appears in the program.

**Restrictions on indices:** If the variable identifiers are indexed, the index is always smaller number of lines in the program. If a variable of the form `y_`$\langle j \rangle$ appears in the program, then `y_`$\langle i \rangle$ must appear in it for all $i < j$. Similarly, we require that if a variable of the form `x_`$\langle j \rangle$ appears in the program, then then `x_`$\langle i \rangle$ must appear in it for all $i < j$.[16]

[16] I've been going back and forth on whether we should make this requirement. The main advantage in it is that it ensures that $Size(s)$ is a finite set, as the inputs and outputs are always smaller than $s$. It does however mean that we may need to add "dummy lines" of the form `one := x_i NAND zero` to the program to ensure that every input variable is touched. We could also simply require that $s \geq n$.

### 23.11.2 Semantics of NAND programs

To evaluate a NAND program $P$ with $n$ inputs and $m$ outputs on input $x_0, \ldots, x_{n-1}$ we initialize the all variables of the form `x_`$\langle i \rangle$ to $x_i$, and all other variables to zero. We then evaluate the program line by line, assigning to the variable on the lefthand side of the `:=` operator the value of the NAND of the variables on the righthand side. In this evaluation, we identify `foo` with `foo_0` and `bar_079` with `bar_79`. That is, we only care about the numerical value of the index of a variable (and so ignore leading zeros) and if an index is not specified, we assume that it equals zero.

The output is the value of the variables `y_0`, ..., `y_`$\langle m-1 \rangle$. (Recall that all variables of the form `y_`$\langle i \rangle$ must be assigned some value.)

Every NAND program $P$ on $n$ inputs and $m$ outputs can be associated with the function $F_P : \{0,1\}^n \rightarrow \{0,1\}^m$ such that for every $x \in \{0,1\}^n$, $F_P(x)$ equals the output of $P$ on input $x$. We say that the function $P$ computes $F_P$. The *NAND-complexity* of a function $F : \{0,1\}^n \rightarrow \{0,1\}^m$ is the length (i.e., number of lines) of the smallest program $P$ that computes $F$.

## 23.12 NAND++ programming language specification

The NAND++ programming language adds to NAND the ability for loops and to access an unbounded amount of memomry.

### 23.12.1 Syntax of NAND++ programs

Every valid NAND program is also a valid NAND++ program, but the NAND++ programming language adds to NAND the following operations:

- An indexed variable identifier can also have the form *var*`_i` where *var* is an unindexed variable identifier.
- The special variables `loop` and `validx` can appear in NAND++ programs. However, `loop` can only appear without an index and only on the lefthand side of the `:=` operator and `validx` can only appear on the righthand side of the `:=` operator. The variable `i` can only appear as an index to an unindexed variable identifier.

[17]

[17] Add discussion of `invalidy` array for NAND++ and NAND«, see the loops lecture

### 23.12.2 Semantics of NAND++ programs

Unlike a NAND program, a NAND++ program can be evaluated on inputs of every length. To evaluate a NAND++ program $P$ on input $x \in \{0,1\}^*$ we do the following:

1. Set $ic = 0$, $r = 0$, $m = 0$, $i = 0$, $inc = +1$ ($ic$ stands for "iteration counter" and $r$ is the current "rounds" of the the index variable).
2. For every line in $P$ of the form *vara* `:=` *varb* `NAND` *varc*, assign to the variable denoted by *vara* the NAND of the values of the variables denoted by *varb* and *varc*. If a variable on the righthand side has not received a value then its value is set to 0. If a variable has the form `foo_i` then we treat it as if it was `foo_`$\langle i \rangle$ where $\langle i \rangle$ denotes the current value of `i`. If a variable has the form `x_`$\langle j \rangle$ then if $j < |x|$ it gets the value $x_j$ and otherwise it gets the value 0. If a variable has the form `validx_`$\langle j \rangle$ then if $j < |x|$ it gets the value 1 and otherwise it gets the value 0. If a variable on the lefthandside has the form `y_`$\langle j \rangle$ then we let $m = \max\{m, j+1\}$.
3. If the variable `loop` has the value 0 then halt with output $y_0, \ldots, y_{m-1}$ where $y_j$ equals the value of `y_`$\langle j \rangle$ if this variable has been assigned a value, and equals 0 if it hasn't been assigned a value. Otherwise (if `loop` has value 1) then do the following:
   - If $i = 0$ then set $r \leftarrow r + 1$ (in this case we have completed a "round") and $inc = +1$.
   - If $i = r$ then set $inc = -1$.
   - Then set $i \leftarrow i + inc$, $ic \leftarrow ic + 1$, and go back to step 2.

We say that a NAND++ program $P$ *halts* on input $x \in \{0,1\}^*$ if when initialized with $x$, $P$ will eventually reach the point where `loop` equals 0 in Step 3. above and will output some value, which we denote by $P(x)$. The *number of steps* that $P$ takes on input $x$ is defined as $(ic+1)\ell$ where $ic$ is the value of the iteration counter $ic$ at the time when the program halts and $\ell$ is the number of lines in $P$. If $F : \{0,1\}^* \to \{0,1\}^*$ is a (potentially partial) function and $P$ is a NAND++ program then we say that $P$ *computes* $F$ if for every $x \in \{0,1\}^*$ on which $F$ is defined, on input $x$ the program $P$ halts and outputs the value $F(x)$. If $P$ and $F$ are as above and $T : \mathbb{N} \to \mathbb{N}$ is some function, then we say that $P$ computes $F$ in time $T$ if for every $x \in \{0,1\}^*$ on which $F$ is defined, on input $x$ the program $P$ halts within $T(|x|)$ steps and outputs the value $F(x)$.

### 23.12.3 *Interpreting NAND programs*

The NAND programming language is sufficiently simple so that writing an interpreter for it is an easy exercise. The website http://nandpl.org contains an OCaml implementation that is more general and can handle many "syntactic sugar" transformation, but interpreting or compiling "plain vanilla" NAND can be done in few lines. For example, the following Python function parses a NAND program to the "list of tuples" representation:

```
# Converts a NAND program from text to the list of tuples
    representation
# prog: code of program
# n: number of inputs
# m: number of outputs
# t: number of variables
def triples(prog,n,m,t):

   varsidx = {}

   def varindex(var): # index of variable with name var
      if var[:2]=='x_': return int(var[2:])
      if var[:2]=='y_': return t-m+int(var[2:])
      if var in varsidx: return varsidx[var]
      return varsidx.setdefault(var,len(varsidx)+n)

   result = []

   for line in prog.split('\n'):
```

```
      if not line or line[0]=='#' or line[0]=='//': continue
           # ignore empty and commented out lines
      (var1,assign,var2,op,var3) = line.split()
      result.append([varindex(var1),varindex(var2),varindex(
          var3)])

   return result
```

The function above assumes we know some parameters of the program, such as its input and output length, and the number of distinct variables, but this is easy to get as well.

```
# compute the parameters of a given program code
# returns: n = number of inputs, m = number of outputs, t =
    number of distinct variables
def params(prog):

   varnames = set()
   for line in prog.split('\n'):
      if not line or line[0]=='#' or line[0]=='//': continue
           # ignore empty and commented out lines
      (var1,assign,var2,op,var3) = line.split()
      varnames.update([var1,var2,var3])

   t = len(varnames)
   n = len([var for var in varnames if var[:2]=='x_'])
   m = len([var for var in varnames if var[:2]=='y_'])

   return [n,m,t]
```

As we discuss in the "code and data" lecture, we can execute a program given in the list of tuples representations as follows

```
# Evaluates an n-input, m-output NAND program L on input x
# L is given in the list of tuples representation
def EVAL(L,n,m,x):
   t = max([max(triple) for triple in L])+1 # num of vars in
        L
   vars = [0]*t # initialize variable array to zeroes
   vars[:n] = x # set first n vars to x

   for (a,b,c) in L: # evaluate each triple
      vars[a] = 1-vars[b]*vars[c]

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

Moreover, if we want to *compile* NAND programs, we can easily transform them to C code using the following `NAND2C` function:[18]

[18] The function is somewhat more complex than the minimum needed, since it uses an array of *bits* to store the variables.

```
# Transforms a NAND program to a C function
# prog: string of program code
# n: number of inputs
# m: number of outputs
# code has not been tested
def NAND2C(prog,n,m):
   avars = { } # dictionary of indices of "workspace"
       variables
   for i in range(n):
      avars['x_'+str(i)] = i
   for j in range(m):
      avars['y_'+str(j)] = n+j

   def var_idx(vname): # helper local function to return
       index of named variable
      vname = vname.split('_')
      name = vname[0]
      idx = int(vname[1]) if len(vname)>1 else 0
      return avars.setdefault(name+'_'+str(idx),len(avars))

   main = "\n"

   for line in prog.split('\n'):
      if not line or line[0]=='#' or line[0]=='//': continue
           # ignore empty and commented out lines
      (var1,assign,var2,op,var3) = line.split()
      main += ' setbit(vars,{idx1}, ~(getbit(vars,{idx2}) &
          (getbit(vars,{idx2}));\n'.format(
         idx1= var_idx(var1), idx2 = var_idx(var2), idx3 =
             var_idx(var3))

   Cprog = '''
   #include <stdbool.h>

   typedef unsigned long bfield_t[ (sizeof(long)-1+{numvars}
       )/sizeof(long) ];
   // See Stackoverflow answer https://stackoverflow.com/
       questions/2525310/how-to-define-and-work-with-an-array
       -of-bits-in-c

   unsigned long getval(bfield_t vars, int idx) {{
      return 1 & (vars[idx / (8 * sizeof(long) )] >> (idx %
```

```
        (8 * sizeof(long))));
}}

void setval(bfield_t vars, int idx, unsigned long v) {{
   vars[idx / (8 * sizeof(long) )] = ((vars[idx / (8 *
       sizeof(long) )] & ~(1<<b)) | (v<<b);
}}

unsigned long int *eval(unsigned long int *x) {{
  bfield_t vars = {{0}};
  int i;
  int j;
  unsigned long int y[{m}] = {{0}};
  for(i=0;i<{n};++i) {{
     setval(vars,i,x[i])
  }}
'''.format(n=n,m=m,numvars=len(avars))

Cprog = Cprog + main + '''
   for(j=0;j<{m};++j) {{
     y[j] = getval(vars,{n}+j)
  }}
  return y;
}}
'''.format(n=n,m=m)

return Cprog
```

## 23.13 *NAND« programming language specification*

The NAND« (pronounced "NAND shift") programming language allows *indirection*, hence using every variable as a pointer or index variable. Unlike the case of NAND++ vs NAND, NAND« cannot compute functions that NAND++ can not (and indeed any NAND« program can be "compiled to a NAND++ program) but it can be polynomially faster.

### 23.13.1 *Syntax of NAND« programs*

Like in NAND++, an indexed variable identifier in NAND« has the form `foo_num` where `num` is some absolute numerical constant or or `foo_i` where `i` is the special index variable. Every NAND++ program

is also a valid NAND« program but in addition to lines of the form `foo := bar NAND blah`, in NAND« we allow the following operations as well:

- `foo := bar` (assignment)
- `foo := bar + baz` (addition)
- `foo := bar - baz` (subtraction)
- `foo := bar » baz` (right shift: $idx \leftarrow \lfloor foo 2^{-bar} \rfloor$)
- `foo := bar « baz` (left shift: $idx \leftarrow foo 2^{bar}$)
- `foo := bar % baz` (modular reduction)
- `foo := bar * baz` (multiplication)
- `foo := bar / baz` (integer division: $idx \leftarrow \lfloor \frac{foo}{bar} \rfloor$)
- `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)

Where `foo`, `bar` or `baz` are indexed or non-indexed variable identifiers but not the special variable `i`. However, we do allow `i` to be on the left hand side of the assignment operation, and hence can write code such as `i := foo`. As in NAND++, we only allow variables of the form `x_..` and `validx_..` and to be on the righthand side of the assignment operator and only allow variables of the form `y_..` to be on the lefthand side of the operator. In fact, by default we will only allow *bit string valued* NAND« programs which means that we only allow variables of the form `y_..` to be on the lefthand side of a line of the form `y_..  := foo NAND bar`, hence guaranteeing that they are either 0 or 1. We might however sometimes consider drop the bit-string valued restriction and consider programs that can output integers as well.

### 23.13.2 *Semantics of NAND« programs*

Semantics of NAND« are obtained by generalizing to integer valued variables. Arithmetic operations are defined as expected except that we maintain the invariant that all variables always take values between 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. More formally, to evaluate a NAND++ program on inputs $x_0, x_1, x_2, \ldots$ (which we will typically assume to be bits, but could be integers as well) we do the following:

1. Set $ic = 0$, $m = 0$, $i = 0$, ($ic$ stands for "iteration counter" and $r$ is the current "rounds" of the the index variable).

2. For every line in $P$, we do the following:

   (a) If the line has the form *vara* `:=` *varb* `NAND` *varc*, assign to the variable denoted by *vara* the NAND of the values of the variables denoted by *varb* and *varc* (interpreting 0 as false and nonzero as true). If a variable on the righthand side has not received a value then its value is set to 0.

   - If a variable has the form `foo` without an index then we treat it as if it was `foo_0`.
   - If a variable has the form `foo_i` then we treat it as if it is `foo_`$\langle j \rangle$ where $j$ denotes the current value of the index variable `i`.
   - If a variable has the form `x_`$\langle j \rangle$ then if $j < |x|$ it gets the value $x_j$ and otherwise it gets the value 0.
   - If a variable has the form `validx_`$\langle j \rangle$ then if $j < |x|$ it gets the value 1 and otherwise it gets the value 0.
   - If a variable on the lefthand side has the form `y_`$\langle j \rangle$ then we let $m = \max\{m, j+1\}$.

   (b) If a line correspond to an index operation of the form `foo := bar OP baz` where `OP` is one of the operations listed above then we evaluate `foo` and `bar` as in the step above and compute the value $v$ to be the operation `OP` applied to the values of `foo` and `bar`. We assign to the index variable corresponding to `idx` the value $\max\{0, \min\{ic, v\}\}$.

   If the variable `loop` has the value 0 then halt with output $y_0, \ldots, y_{m-1}$ where $y_j$ equals the value of `y_`$\langle j \rangle$ if this variable has been assigned a value, and equals 0 if it hasn't been assigned a value. Otherwise (if `loop` has value 1) then do the following. Set

$px \leftarrow ic + 1$, set $i = INDEX(ic)$ where $INDEX$ is the function that maps the iteration counter value to the current value of `i`, and go back to step 2.

Like a NAND++ program, the number of steps which a NAND« program $P$ takes on input $x$ is defined as $(ic + 1)\ell$ where $ic$ is the value of the iteration counter at the point in which it halts and $\ell$ is the number of lines in $P$. Just like in NAND++, we say that a NAND« program $P$ computes a partial function $F : \{0,1\}^* \to \{0,1\}^*$ in time $T : \mathbb{N} \to \mathbb{N}$ if for every $x \in \{0,1\}^*$ on which $F$ is defined, $P(x)$ outputs $F(x)$ within $T(|x|)$ steps. Note that any NAND« program can be transformed into a NAND++ program that computes the same function, albeit at a polynomial loss in the number of steps.

## 23.14 The "standard library"

Additional features of NAND/NAND++/NAND« can be implemented via "syntactic sugar" transformations. The following is a "standard library" of features that can be assumed in writing programs for NAND/NAND++/NAND«. Whenever counting number of steps/lines in a program, or when feeding it as input to other programs, we assume that it is first transformed into its "sugar free" version that does not use any of these features.

### 23.14.1 Syntactic sugar for NAND

The following constructs are applicable in all the languages NAND/-NAND++/NAND«:

**Variable assignment:** `foo := bar` where `foo`,`bar` are variable identifiers corresponds to assigning the value of `foo` to `bar`.

**Constants:** We can assume that `zero` and `one` are assigned the values 0 and 1 respectively.

**Multidimensional arrays:** We can use multidimensional indices such as `foo_12,24` or `foo_i,j,k`. These can be transformed into a one-dimensional array via one-to-one embeddings of $\mathbb{N}^k$ in $\mathbb{N}$.

**Conditionals:** We can write code such as

```
if (foo) {
   code
}
```

to indicate that `code` will only be executed if `foo` is equal to 1.

**Functions:** We can define (non recursive) functions as follows:

```
def foo1,...,fook := Func(bar1,...,barl) {
   function_code
}
```

denotes code for a function that takes $l$ inputs `bar1`,...,`barl` and returns $k$ outputs `foo1`,...,`fook`. We can then invoke such a function by writing

```
blah1,...,blahk := Func(baz1,...,bazl)
```

this will be implemented by copy-pasting the code of `Func` and doing the appropriate search-and-replace of the variables, adding a unique prefix to workspace variables to ensure there are no namespace conflicts.

We can use Functions also inside expressions, and so write code such as

```
foo := XOR(bar,baz)
```

or

```
if AND(foo,bar) {
   code
}
```

where `XOR`,`AND` have been defined above.

**NAND for loops:** We can introduce syntactic sugar for loops in NAND, as long as the number of times the loop executes is fixed and independent of the input size. Hence we will use

More generally, we will replace code of the form

```
for i in RANGE do {
   code
}
```

where `RANGE` specifies a finite set $I = \{i_0, \ldots, i_{k-1}\}$ of natural numbers, as syntactic sugar for $|R|$ copies of `code`, where for $j \in [k]$, we replace all occurences of `_<expr(i)>` in the $j$-th copy with $\_\langle expr(i_j)\rangle$ where `expr(i)` denotes an arithmetic expression in `i` (involving `i`, constants, parenthesis, and the operators `+`,`-`,`*`,`mod`,`/`) and for every $x \in \mathbb{N}$, $expr(c)$ denotes the result of of applying this expression to the value $c$.

We specify the set $I = \{i_0, \ldots, i_{k-1}\}$ by simply writing `[` $\langle i_0 \rangle$, $\langle i_1 \rangle$, ..., $\langle i_{k-1} \rangle$ `]`. We will also use the $\langle beg \rangle : \langle end \rangle$ notation so specify the interval $\{beg, beg+1, \ldots, end-1\}$. For example, `[ 2:4, 10:13 ]` specifies the set $\{2, 3, 10, 11, 12\}$.

**Operator overloading and infix notation:** For convenience of notation, we can define functions corresponding to standard operators `*`,`+`,`-`, etc.. and then code such as `foo * bar` will correspond to `operator*(foo,bar)`.

[19]

[19] TODO: potentially add lists and lists operations

### 23.14.2 *Encoding of integers, strings, lists.*

While NAND/NAND++/NAND« natively only supports values of 0 and 1 for variables (and integers for indices in NAND++/NAND«), in our standard library We will use the following default 8-symbol alphabet for specifying constants and strings in NAND/NAND++/NAND«:

| Symbol | Encoding | Default semantics |
|---|---|---|
| `.` | 000 | End of string/integer/list marker |
| `0` | 001 | The bit 0 |
| `1` | 010 | The bit 1 |
| `[` | 011 | Begin list |
| `]` | 100 | End list |
| `,` | 101 | Separate items |
| `:` | 110 | Define mapping |
| `_` | 111 | Space / "no op" / to be ignored |

When we write code such as `foo:= "str"` where `str` is a length $n$ sequence from this alphabet that doesn't contain `.`, then we mean that we store in `var_`$\langle 0 \rangle$ till `var_`$\langle 3n-1 \rangle$ the encoding for the $n$ symbols of `str`, while we store in `var_`$\langle 3n \rangle$,`var_`$\langle 3n+1 \rangle$,`var_`$\langle 3n+2 \rangle$ the encoding for `.` (which is 000).

We will store integers in this encoding using their represnetation binary basis ending with `.`. So, for example, the integer $13 = 2^3 + 2^2 + 2^0$ will be stored by encoding `1011.` and hence the shorthands `foo := 13` and `foo := "1011"` will be transformed into identical NAND code. Arithmetic operations will be shorthand for the standard algorithms and so we can use code such as

```
foo := 12
bar := 1598
```

```
baz := foo * bar
```

Using multidimensional arrays we can also use code such as

```
foo_0 := 124
foo_1 := 57
foo_2 := 5459
```

**Lists:** We store lists and nested lists using the separators `[,],`, with the binary encoding of each element. Thus code such as

```
foo := [ 13 , 1, 4 ]
```

will be the same as

```
foo := "[1011,01,001]"
```

We can store in lists not just integers but any other object for which we have some canonical way to encode it into bits.

We can use the union operation on lists and so

```
foo := [17,8] + [24,9]
```

will be the same as

```
foo := [17,8,24,9]
```

we will use `in` as shorthand for the operation that scans a list to see if it contains an element and so in the code

```
foo := [12,127,8]
bar := 8 in foo
```

`bar` is assigned the value 1.

The `length` macro computes the length of a list.

Within lists the no-op character `_` will be ignored, and so replacing characters with `_` can be a way of removing items from the lists.

**Iterating:** We use the construct

```
for foo in bar {
   code
}
```

to execute code `length(bar)` times where each time `foo` will get the current element.

20

21

[20] TODO: check if we should add map and filter for lists

### 23.14.3 *Syntactic sugar for NAND++ / NAND«*

In addition to the syntactic sugar above, in NAND++ and NAND« we can also use the following constructs:

**Indirect indexing:** If we use code such as `foo_bar` where `bar` is not an index variable then this is shorthand for copying the integer encoded in `bar` into some index variable of the form `i"'` (with an appropriate number of primes to make it unique) and then use `foo_i"'`. Similarly, we will also use code such as `idx := bar` where `idx` is an index variable and `bar` is a non-index variable to denote that we copy to `idx` the integer that is encoded by `bar`. If `bar` does not represent a number then we consider it as representing 0.

**Inner loops:** We can use the following loop construct

```
while (foo) {
   code
}
```

to indicate the loop will execute as long as `foo` equals 1. We can nest loops inside one another, and also replace `foo` with another expression such as a call to a function.

[22]

[22] TODO: potentially have a loop that iterates over a list