IAS/PCMI Summer Session 2000
Clay Mathematics Undergraduate Program
Advanced Course on Computational Complexity

# Lecture 4: Non-Regular Languages and Uniformity

David Mix Barrington and Alexis Maciel
July 20, 2000

## 1. Overview

Now that we've characterized the regular and star-free languages in various ways, we're ready to move beyond them into the "real world" of complexity classes. In this lecture we will:

- Explore the relationship of the regular and star-free languages to the circuit classes we have defined in the basic lectures,
- Examine some non-regular languages and how they may be decided in the extensions of our models such as M-programs and first-order logic with more predicates,
- Examine a particular logical model, first-order logic with the BIT predicate, and demonstrate its computational power on some examples, and finally
- Relate first-order logic with BIT to log-time machines and log-time uniform $AC^0$ circuits.

## 2. Our New Models and Circuit Complexity

We have defined two robust complexity classes with our three new models of computation:

- The star-free regular languages, equal to the first-order definable languages and (we have asserted) the languages recognizable by aperiodic monoids, and

- The regular languages, equal to the recognizable languages.

We've proved that these two classes are different, because the language $(aa)^*$ is not first-order definable but is regular. We also showed one example of a language that is not in either class, the language $\{a^n b^n : n \geq 0\}$ that has infinitely many Myhill-Nerode classes.

In the Basic lectures we've defined various complexity classes in terms of machines and boolean circuits. The regular languages are equal, as we showed, to the machine-based class DSPACE(1). Thus they are contained in the machine-based classes L and P (the latter because their computations require linear time). But the relationship between our classes and the circuit-based classes is less obvious. The two relevant circuit-based classes are $\mathrm{AC}^0$ (unbounded fan-in, constant depth, polynomial size) and $\mathrm{NC}^1$ (bounded fan-in, $O(\log n)$ depth, polynomial size).

**Theorem 1** *Any first-order definable language is also in* $\mathrm{AC}^0$*. (We will for the moment ignore uniformity in the definition of circuit-based classes and be satisfied to show that the appropriate circuit families exist.*

**Proof** Let $\Phi$ be a formula (in prenex normal form, with all its $k$ quantifiers in the front) and fix the input length $n$. The major part of our circuit will be a tree of gates of depth $k$ and fan-in $n$ as follows. If the first quantifier is $\exists x$, the output gate will be an OR gate with $n$ children, each corresponding to one of the $n$ choices of $x$. If the first quantifier is $\forall x$, for example, the output gate is an AND gate with the same $n$ children.

Each of the succeeding quantifiers gives rise to a *level* of gates, AND gates for a $\forall$ quantifier and OR gates for an $\exists$. Each of these gates represents a choice of values for the variables *already bound* when their quantifier occurs, and each has $n$ children corresponding to the $n$ choices of values for the variable that *this quantifier* binds. If we call the output gate level 0, there are exactly $n^i$ gates on level $i$ of the tree.

Each of the $n^k$ leaves of this tree corresponds to a marked word of length $n$, obtained from $w$ by placing the $k$ marks for the $k$ variables according to the sequence of choices leading to that leaf. Our remaining task is to design a circuit for each leaf that will determine the value of the quantifier-free part of $\Phi$ (a boolean combination of atomic formulas) on that leaf's marked word.

We can certainly implement the boolean combination with a constant number of gates at each leaf. We thus need only determine the truth of each atomic formula on the marked word. Consider the three types of atomic formulas. For the order

and equality formulas, the answer is *determined* by the choice of values for the two variables concerned. We can thus place a constant 0 or 1 gate in the circuit for each leaf, depending on the choices leading to the leaf. For an input predicate $I_a(x)$, we must look at the letter in position $x$ of the input string $w$ and see whether it is an $a$. This can be done by one particular input gate (or perhaps $O(1)$ gates, depending on the coding of the input alphabet by booleans). Therefore with $O(1)$ gates we can evaluate the quantifier-free part on each leaf.

It should be clear, from the definition of the meaning of formulas and of the value of gates, that the output value of this circuit represents the truth value of the formula $\Phi$ on $w$. The size of the circuit is $O(n^k)$, and the depth is $k$, plus perhaps $O(1)$ for the bounded fan-in circuits at each leaf. We have thus placed the arbitrary first-order definable language in (non-uniform) $\mathrm{AC}^0$. □

**Theorem 2** *Any recognizable language is in* $\mathrm{NC}^1$ *(again, ignoring uniformity for the time being).*

**Proof** Let $L = \phi^{-1}(X)$ for some homomorphism $\phi$ from $\Sigma^*$ to a finite monoid $M$ and some set $X \subseteq M$. Choose an arbitrary encoding of the elements of $M$ as sequences of $O(1)$ booleans. For each input letter $w_i$ in $w$, we can make a circuit of size $O(1)$ and bounded fan-in that reads $w_1$ and outputs the encoding of the monoid element $\phi(w_i)$. By a lookup table, we can make another circuit of size $O(1)$ and bounded fan-in that inputs two encodings of monoid elements and outputs the encoding of their product. By a binary tree of copies of the latter circuit, we can take the $n$ monoid elements $\phi(w_i)$ and find the monoid element $\phi(w)$. A final circuit of $O(1)$ size and bounded fan-in determines whether $\phi(w)$ is in $X$ and thus whether $w$ is in $L$.

The entire circuit has bounded fan-in throughout, and its depth is $O(\log n)$ because the tree has $\log n$ levels of $O(1)$ depth each. Its size is $O(n)$ because there are $n-1$ multipliers, each of size $O(1)$, plus $n$ letter converters, each of size $O(1)$, and the output converter of size $O(1)$. We have thus constructed an $\mathrm{NC}^1$ circuit. □

Thus $\mathrm{NC}^1$ contains both $\mathrm{AC}^0$ and the regular languages, and both $\mathrm{AC}^0$ and the regular languages contain the star-free languages. In fact all four of these classes are different, and $\mathrm{AC}^0$ and the regular languages are incomparable, though we have not yet proved all these facts. (In the next section we exhibit a language in $\mathrm{AC}^0$ but not regular, and in Basic Lecture 10 we exhibit a regular language that is not in $\mathrm{AC}^0$ (even in non-uniform $\mathrm{AC}^0$).

# 3. Extending the Models to Non-Regular Languages

Consider our canonical non-regular language $\{a^i b^i : i \geq 0\}$. It is not first-order definable as we have so far understood the term, but consider the following formula (with the input positions considered to be numbered from 0 to $n-1$):

$$\begin{aligned} \exists x : \forall y : \quad & [(y \leq x) \rightarrow I_a(y)] \wedge \\ & [(y > x) \rightarrow I_b(y)] \wedge \\ & [(y \leq x) \leftrightarrow \exists z : (z = y + y)] \wedge \\ & [\exists z : \exists w : (z > w) \wedge (w = x + x)] \end{aligned}$$

This formula is a first-order formula all right, but it uses an additional atomic predicate $x = y + z$ along with order, equality, and the input predicate. Let's see what it means. The string $w$ must be in $a^*b^*$, with $x$ denoting the position of the last $a$. The last two clauses determine the value of $x$. The double of a number exists (is less than $n$) iff it is at most $x$, so $2x$ must be $n-1$ or $n-2$. The last clause forces $n$ to be even, so that $z = n-1$ can be greater than $w = 2x = 2(n/2-1)$. Thus a string can satisfy this formula iff it is in $\{a^i b^i : i \geq 1\}$. If we OR in the clause $\forall x : \neg(x = x)$, for example, we can include the empty string as well and get exactly our example of a non-regular language.

Many similar non-regular languages can be defined if we allow ourselves suitable new atomic predicates:

- $\{a^n b^{n^2} : n \geq 0\}$ can be defined if we have an atomic predicate for *multiplication* of input position numbers.
- Consider the set of strings $w$ such that $w_i = a$ iff the $i$'th machine in some standard list halts on blank input. This language is easily seen to be *undecidable* by any machine *at all*, using basic computability theory. But a first-order formula $\forall x : I_a(x) \leftrightarrow T(x)$ *exists* to describe this language, if we are allowed to use the appropriate uncomputable new atomic predicate $T$.

This is our first example of a general phenomenon, that of *non-uniform* computation. Boolean circuits are an example of a *combinatorial* model of computation, where a large number of simple computing elements are arranged so as to solve a problem. We've already talked about the *combinatorial* resources a circuit uses, such as the number of gates (size) and the length of the longest path (depth). Non-uniformity

is the resource that allows us to arrange the circuit in complicated ways. (It gets its name from the prior use of the term "uniform" to describe circuit families put together in very simple ways, particularly those where the different circuits in the family are very similar.)

With circuits we measure non-uniformity by the computational complexity of either *describing* each circuit or of *answering questions* about it. A completely *non-uniform* circuit has no restriction of this kind — the circuits in the family simply have to exist. In a *Turing uniform* family there must be some machine (with no bound on its resources) that inputs an input size as a number and outputs a description of a circuit of that size. In a *poly-time uniform* family this machine is restricted to use only time polynomial in the input size, a *log-space uniform* family restricts this to space logarithmic in the input size, and so on. Later in this lecture we'll see one of the most restrictive uniformity conditions in common use, that of *log-time uniformity*. A log-time machine could never describe the whole circuit, so we require it only to be able to answer questions about one or two gates at a time, referred to by their numbers. Since these numbers have $O(\log n)$ bits in a poly-size circuit, the machine has time to read them and process them in some way.

In the logical model we introduce non-uniformity in a different way, by adding new atomic predicates to the first-order formalism. We still want to access the input in the same way, so the input predicates like $I_a(x)$ will still be the only predicates that refer to the input. Our new predicates must be *numerical*, meaning that they can refer only to the *numbers* of input positions. We saw several examples of numerical predicates above, such as those for addition and multiplication of input positions.

Non-uniformity allows us to define new complexity classes in the logical model. We have been working with FO, the languages definable by first-order formulas whose only numerical predicates are $=$ and $<$. We get particular classes by adding particular predicates, and we get the completely non-uniform class FO + ARB by adding *arbitrary* numerical predicates. (Of course a single formula will contain only some finite number of predicates, each of a fixed arity.) Perhaps not surprisingly, completely non-uniform first-order formulas correspond in computing power to completely non-uniform $\mathrm{AC}^0$ circuits:

**Theorem 3** *A language is in* FO + ARB *iff it is in non-uniform* $\mathrm{AC}^0$.

**Proof** We have seen above the basic argument to show that formulas may be simulated by circuits, when we showed that FO is contained in non-uniform $\mathrm{AC}^0$. For FO + ARB formulas, we can still create an $n$-ary tree of gates, AND gates for $\forall$ quantifiers and OR gates for $\exists$ quantifiers. For each leaf of this tree, we must construct

a circuit that determines the truth value of the quantifier-free part of the formula, evaluated on a marked word corresponding to a particular choice of a value for each of the variables in the formula. Since this quantifier-free part is a boolean combination of atomic formulas, we need only show how to evaluate an atomic formula given choices of all the variable values. For an input predicate, we use an input gate to look up the appropriate input value as we did before. But now note that for *any* numerical predicate, if the values of the variables are fixed, so is the value of the predicate (even if it is difficult or impossible to compute). There thus *exists* a circuit containing a constant 0 or 1 gate in the right position corresponding to this numerical predicate value. Just as before, once we know that the leaves evaluate to the correct value of the quantifier-free part on each marked word, the tree of gates simulates the quantifiers and the output gate contains the truth value of the entire formula on the input word.

For the other half, we need a new argument that takes an arbitrary circuit of polynomial size and constant depth and produces an FO + ARB formula that simulates the circuit. We first number the gates of the circuit with vectors of $O(1)$ variables, and declare numerical predicates:

- $INP(\mathbf{x})$ meaning that the gate labelled by the vector $\mathbf{x}$ is an input gate,
- $OUT(\mathbf{x})$ meaning that gate $\mathbf{x}$ is an output gate,
- $AND(\mathbf{x})$, $OR(\mathbf{x})$, and $NOT(\mathbf{x})$ to represent the gate type of gate $\mathbf{x}$, and
- $CHILD(\mathbf{x}, \mathbf{y})$ meaning that gate $\mathbf{x}$ is a child of gate $\mathbf{y}$.

Armed with these predicates, we can successively define more predicates using first-order quantifiers, meaning:

- "gate $\mathbf{x}$ is an input gate with value 1",
- "gate $\mathbf{x}$ is a gate at level 1",
- "gate $\mathbf{x}$ is a gate at level 1 with value 1",
- "gate $\mathbf{x}$ is a gate at level $d$",
- "gate $\mathbf{x}$ is a gate at level $d$ with value 1", and finally
- "gate $\mathbf{x}$ is an output gate with value 1"

(We leave to the exercises the exact formulas for each of these.) The circuit accepts the input iff there exists an $\mathbf{x}$ such that the last predicate is true. □

In Basic Lecture 10 we'll prove that the parity language (strings of $a$'s and $b$'s with an even number of $a$'s) is *not* in FO + ARB or non-uniform $AC^0$. At this point this should be a surprising result, because we have already seen that FO + ARB can do uncomputable things! But we will see how the combinatorial bounds themselves impose certain limits on the computational power of circuits.

The model of $M$-programs provides a non-uniform extension of the algebraic computational model. Fix a finite monoid $M$. An $M$-program is a sequence of *instructions*, where each instruction consists of an input position number and a function from $\Sigma$ to $M$. The *yield* of an instruction $\langle i, f\rangle$ is the monoid element $f(x_i)$, and the yield of the entire $M$-program is the product, in $M$, of the yields of the instructions. (Note that in general $M$ may not be abelian, so that the order of the instructions is important.) A fixed program defines a map $\phi$ from words in $\Sigma^n$ to $M$, and a *family* of $M$-programs, one for each input length, defines a map from $\Sigma^*$ to $M$. Just as for homomorphisms, a language is recognized by an $M$-program if it is $\phi^{-1}(X)$ for some set $X \subseteq M$.

Let's look at examples of $M$-programs for the non-regular languages above. For $\{a^n b^n : n \geq 0\}$, we let $M$ be the monoid $T_2$ ($\{0,1\}$ under addition with $1+1=1$) and let $X = \{0\}$. For even $n$, the program consists of $n/2$ instructions $\langle i; a \mapsto 0, b \mapsto 1\rangle$ for $i$ from 0 to $n/2-1$, followed by $n/2$ instructions $\langle i; a \mapsto 1, b \mapsto 0\rangle$ for $i$ from $n/2$ to $n-1$. (For odd $n$ we want to always reject, so one instruction $\langle 0; a \mapsto 1, b \mapsto 1\rangle$ will do.) This program yields 0 only if each of the letters has the correct value for the string to be in $\{a^i b^i : i \geq 0\}$.

Note something about this construction — it tests whether the input string is equal to one particular string of the right length. The program would look much the same if the target strings changed, so that our other two examples are also easily seen to have linear-length $M$-programs over the same monoid $T_2$. In the exercises you are asked to show that there are poly-length $M$-programs to decide any language in FO + ARB. The monoids for these programs are all aperiodic, but depend on the depth of the circuit being simulated. It turns out that FO + ARB (and thus non-uniform $AC^0$ as well) is exactly equal to the set of languages that can be decided by poly-length $M$-programs over aperiodics.

But what of $M$-programs over general finite monoids? They can certainly decide any regular language (including parity, which we have asserted to be outside of FO + ARB), and additional non-regular languages. (For example, consider the set of

strings that have an even number of $a$'s in prime-numbered positions. This is easy to decide with a program over the integers mod 2.)

Can we think of a language that cannot be decided by an $M$-program over any finite monoid? A natural candidate is the *majority language* — the set of strings of $a$'s and $b$'s with at least half the letters $a$'s. In the next lecture we'll characterize the power of general $M$-programs and determine whether they can decide the majority language.

# 4. First-Order Logic with BIT

FO + ARB is an attractively robust model of computation, but not a realistic one. As we have seen, there are languages that are not computable at all in the conventional sense but have first-order definitions using strange numerical predicates. To model actual computation, we want to limit our numerical predicates in some way to keep them easy to compute. And limiting them to $<$ and $=$ is too restrictive, because it doesn't allow us to define any non-regular languages.

In this section we will consider a single numerical predicate that is very easy to compute with a machine but greatly increases our ability to define languages. A fundamental tool in real machines is *indirect addressing*: the ability to use a number as an index to one of many memory locations. In first-order logic we capture this ability with the *bit predicate* BIT. If $x$ and $y$ are numbers representing input positions, $\mathrm{BIT}(x, y)$ is true iff the $x$'th bit of the binary expansion of $y$ is a one. (We define the $x$'th bit to be the one that records the number of $2^x$'s in $y$.) The complexity class FO + BIT consists of those languages definable by first-order formulas using the numerical predicates $<$, $=$, and BIT.

In Basic Lecture 2, we showed that an $\mathrm{AC}^0$ circuit can compute the sum of two $n$-bit binary integers, by carry-lookahead addition. This operation cannot be in ordinary FO if the two inputs are given sequentially, because a finite-state machine cannot match corresponding bits of the two inputs in order to add them. (In an exercise you were asked to show that addition *is* regular if the format is chosen correctly.) The BIT predicate allows us to deal with any natural formatting of the input. For example, suppose the input size $n$ is equal to $2^k$ for some $k$, and we get the $n/2$-bit input $x$ in the first $n/2$ bits, followed by $y$. We can define $k$ as the smallest number for which $\mathrm{BIT}(k, i)$ is always 0, then distinguish between input bits from $x$ and $y$ by looking at bit $k-1$ of the address. Thus bit $i$ of $x$, for example, is bit $i$ of the input and bit $i$ of $y$ is bit $2^{k-1}+i$ of the input. (The number $2^{k-1}+i$ is the unique number whose bits are the same as $i$'s, except for bit $k-1$.) Let's work through the definition

of the addition formula — we eventually want $z(i)$ to be true iff the $i$'th bit of $x + y$ is one:

- First define (for each $i$) $e(i)$ to be the exclusive or $x(i) \oplus y(i)$ of the input bits $x(i)$ and $y(i)$,
- Then define $g(i)$ to be true if a carry is generated at bit $i$ ($g(i) = x(i) \wedge y(i)$), and $p(i)$ to be true if one is propagated ($p(i) = x(i) \oplus y(i)$).
- The carry bit $c(i)$ is then given by $\exists j : \forall k : (j \leq i) \wedge g(j) \wedge ((i > k > j) \rightarrow p(k))$,
- and finally the output bit $z(i)$ is just $e(i) \oplus c(i)$.

By applying this method to input positions themselves, we can define the predicate $x + y = z$ on input positions in FO + BIT. This PLUS predicate was the additional numerical predicate we used above to define the language $\{a^n b^n\}$. We'll now see that FO + BIT allows us quite a bit of arithmetic on input positions, once we develop some computational techniques.

There are several good ways to develop arithmetic in FO + BIT — here we choose one based on *sequences*. Let $t$ be the square root of $\log n$ (we can assume that $n$ is two to a perfect square without much loss of generality). If we have a sequence $x_0, \ldots, x_{t-1}$ where each $x_i$ is a number of at most $t$ bits, we can represent this sequence by the *single number* $\mathbf{x} = \sum_{i=0}^{t-1} x_i 2^{it}$. The $t^2$ bits of the binary expansion of $\mathbf{x}$ can be broken up into $t$ $t$-bit numbers. We need to devise a way to access the individual numbers by their indices (for example, the $j$'th bit of $x_i$) in FO + BIT.

Once we define the number $t$, we can define a number $\mathbf{y}$ corresponding to the sequence $1, \ldots, 1$ by saying that bit 0 is one and bit $i + t$ is one iff bit $i$ is. Then the number for the sequence $\mathbf{z} = 0, 1, 2, \ldots, t - 1$ has another particular property defined in terms of the positions where $\mathbf{y}$ has ones. (The substring of bits ending in one of these places is always one greater than the corresponding number in the next such place on the right.) Similarly $\mathbf{w} = 1, 2, 4, \ldots, 2^{t-1}$ can be defined by doubling each number in succession. These sequences can be defined in terms of arbitrary $t$, and the correct $t$ is the one for which $\mathbf{w}$ has a one in its leftmost position.

We can use sequences to solve a problem called BITSUM in FO + BIT. This problem is to take a $\log n$-bit number $x$ and determine the number of ones in its binary expansion — that is, to add up $\log n$ booleans to get an integer of $\log \log n$ bits. To do this we view $x$ as $t$ $t$-bit numbers. To get the bitsum of each of these numbers, we can guess a sequence of $t$ $t$-bit numbers, where the first is 0 and the $i + 1$'st is either equal to the $i$'th (if the $i$'th bit of the bitsummed number is zero)

or its successor (if the $i$'th bit of the bitsummed number is one). The last of these $t$ numbers is the bitsum. We then have to add the $t$ individual bitsums to get a grand total, which we can do similarly: guess a sequence where the first number is 0 and each successive one is greater than the previous one by the appropriate addend. None of the numbers involved here can exceed $\log \log n$ bits, so they fit in the $t$ bits available for each.

Now we can use BITSUM to calculate the product of two $\log n$-bit numbers. By the ordinary multiplication algorithm we get $\log n$ numbers to add, of $O(\log n)$ bits each. (The $i$'th number is $2^i y$ if $\mathrm{BIT}(i, x)$ is true and zero otherwise.) Think of each of these numbers $z_i$ in base $2^t$, then split it into two numbers, one ($u_i$) with the odd-numbered base-$2^t$ "digits" and one ($v_i$) with the even-numbered "digits". We will first add all the $u_i$ together, then add all the $v_i$ together, then add the two totals together. The final addition we already know to be in FO + BIT — the problem is to find the two subtotals.

Here we use the fact that adding all the "digits" in one column cannot cause a carry farther than the next column, since the total for the column is at most $2^t(\log n) < 2^{2t}$. Each column of *bits* within this column of digits can be added with BITSUM to get a number of $\log \log n$ bits. The sum for the column is the sum of shifted versions of these BITSUM's, that is, $t$ numbers of at most $t + \log \log n$ bits each. These can be added by guessing sequences as before, with a few extra tricks.

We've now shown that FO + BIT can simulate the PLUS and TIMES operations on input positions. It turns out that this is *exactly* what FO + BIT can do, in the sense that first-order logic with PLUS and TIMES can simulate the BIT predicate and thus is *equal* to FO + BIT. (This is yet more evidence for the robustness of this class.) We'll outline the proof of this fact in the exercises.

# 5. Log-Time Turing Machines and Log-Time Uniformity

In this last section we relate the computational power of FO + BIT to the machine model. The machine class relevant to FO + BIT turns out to be the languages decided in logarithmic *time*, a somewhat strange class because a machine as we have defined it can only access the first $O(\log n)$ input positions in time $O(\log n)$. There are two answers to this: first, we will apply DLOGTIME machines to inputs where we only care about these first positions, and secondly, in the exercises we will deal with an enhanced machine that has *random access* to its input.

First note that the BIT predicate itself, when used in first-order formulas that talk about strings of length $n$, has only $O(\log n)$ bits of relevant input. To compute $\mathrm{BIT}(x, y)$, a machine can place a head on the $2^0$ bit of $y$ and then move its head left, maintaining a *binary counter* in its working memory. When the number on the counter equals $x$, the machine need only output the bit of $y$ it now sees. The only technicality is to operate the counter in $O(|x|) = O(\log n)$ time despite the need to take time for carries — we deal with this in the exercises. (We also need to compare the number on the counter to $x$ at every step.)

We have simulated the BIT predicate in DLOGTIME, and we will now simulate a DLOGTIME machine in FO + BIT. The natural idea would be to represent the successive configurations of the machine by numbers, but unfortunately the $O(\log n)$ configurations have $O(\log n)$ bits each, and we cannot do this with only $O(1)$ variables. But we can do something similar. There *is* a number (or more precisely, a sequence of $O(1)$ numbers) that represents the sequence of *internal states* that the machine takes on in its $O(\log n)$ computation steps. There is a unique sequence of states that is correct for the machine and the input — our task is to define a formula that will test whether an alleged sequence is the correct one.

To be correct the sequence of states must have the property that each state follows, by the rules of the machine, from the previous state, the input character seen, and the memory character seen. To determine these two seen characters, we must know where on the input the machine is, where on the worktape it is, and what was written on the worktape the last time the machine was in that place. Fortunately, *all* these facts can be determined using a variant of the BITSUM predicate. At each time step, we have a move on the input tape and a move on the worktape, either left, right, or stationary. Finding the position means adding up a sequence of 1's, 0's, and $-1$'s. Finding the character written means saying something like "there is a time step such that the net movement on the worktape from then to now is zero, there is no intermediate time at which the net movement from then is zero, and the character written then was an $a$". This is expressible in FO + BIT.

Now that we have related DLOGTIME machines to FO + BIT and vice versa, we can prove our characterization of FO + BIT by uniform circuits:

**Theorem 4** *A language is in* FO + BIT *iff it is in DLOGTIME-uniform* $\mathrm{AC}^0$*, meaning that the child and gate-type predicates on the circuit taking $n$ inputs can be decided by a machine in time $O(\log n)$.*

**Proof** We follow the proof above that non-uniform $\mathrm{AC}^0$ equals FO + ARB, with suitable attention to uniformity. Given an FO + BIT formula of depth $d$, we form a

tree of depth $d$ and fan-in $n$ as before. The numbers of the gates in this tree encode the sequence of choices taken to get to the gate, as a sequence of numbers from 0 to $n-1$ with their binary representations concatenated. The child and gate-type predicates can then be computed with simple string comparisons on these $O(\log n)$ bit strings, in $O(\log n)$ time.

For the circuit segments on the leaves of the tree, the gate numbers have a prefix indicating the choice of values leading to the leaf, and an additional $O(1)$ bits to indicate which gate of the segment this gate is. These gates are boolean operators, input gates, or constant gates representing numerical predicates evaluated on the values chosen for the variables. The child predicate on gate numbers depends only on which gates in the segment we have (and are false if the two gates are in different segments). The gate type predicate may depend on the sequence of values — to compute it we may need to look up one of the values (to determine which input we want) or evaluate the order, equality, or BIT predicate on two of these values. The latter operation, and hence the entire gate-type predicate, is computable in DLOGTIME as we saw above how to evaluate BIT.

For the other half of this proof, we simply follow the development of the predicates on gates that indicate their depth and their output value. These predicates have first-order definitions explicitly in terms of the child and gate-type predicates. If, by hypothesis, these two predicates are computable in DLOGTIME, by the argument above they are also in FO + BIT. Thus all the depth and value predicates, and finally the output of the circuit, are in FO + BIT. □

Most of the natural machine-based complexity classes also have equivalent definitions in terms of uniform circuits and in terms of first-order formulas. The relationships among these classes are clearest when we use the most restrictive uniformity conditions that still allow them to be proved. In the case of circuits this generally turns out to be DLOGTIME uniformity, and in the case of formulas this generally turns out to be the BIT predicate.

We've defined three versions of $AC^0$ in terms of first-order logic:

- FO itself, equal to the star-free languages,
- FO + BIT, equal to log-time uniform $AC^0$ and to FO augmented with PLUS and TIMES, and finally
- FO + ARB, equal to non-uniform $AC^0$.

# 6. Exercises

1. Show that a language is in poly-time uniform $\mathrm{AC}^0$ iff it can be described by a first-order sentence in which each numerical predicate can be computed (with input size $n$) in time polynomial in $n$.

2. Find a series of aperiodic monoids $M_d$ such that depth-$d$ first-order formulas (with arbitrary numerical predicates, say) can be simulated by polynomial length $M$-programs over $M_d$.

3. Show that with PLUS and TIMES, there is a first-order predicate POWER($x$) that is true iff $x$ is a power of two.

4. Show that with PLUS and TIMES, we can express the predicate INSUM($y, x$) that is true iff $y$ is a power of two and there is a one in the $y$'s place of the binary expansion of $x$. Note how this differs from the BIT predicate.

5. Show that if $y$ is a power of two, with PLUS and TIMES, we can define the numbers whose base-$y$ representations are $1, 1, \ldots, 1$ and $1, 2, 4, \ldots, 2^k$. Show how with the right condition on these two numbers we can force $n$ to be $2^{(\log y)^2}$. (Hint: use TIMES to simulate addition on bit positions).

6. Now show that with this $y$, we can also define (with PLUS and TIMES) the number whose base-$y$ representation is $0, 1, 2, \ldots, \log y$. Show that by comparing components between this number and $1, 2, 4, \ldots, y$ we can define BIT($i, j$) whenever $i \leq \log y$. Complete the argument that we can define the entire BIT predicate with PLUS and TIMES.

7. Show that a machine with a tape can operate a binary counter, counting from 0 to $t$, in $O(t)$ time. (Hint: the problem is that carries take time to update all the carried bits on the tape. But carries don't happen all that often, at least big carries. Add up the total time taken by all the carries to show that the *average* update takes only $O(1)$ time).

8. A *random-access machine* has a special section of its memory called its *index string*. It has no input read head — instead, on a given time step it sees the bit of the input whose index is currently written as the index string. If the index is out of range, the machine gets indication of this. Show that a random-access machine can determine the length of its input in $O(\log n)$ time. (Assume that the index tape starts as the empty string.)

9. Show that a DLOGTIME random access machine can also be simulated in FO + BIT.