Restricted computational models

Boaz Barak

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

# Restricted computational models

### 

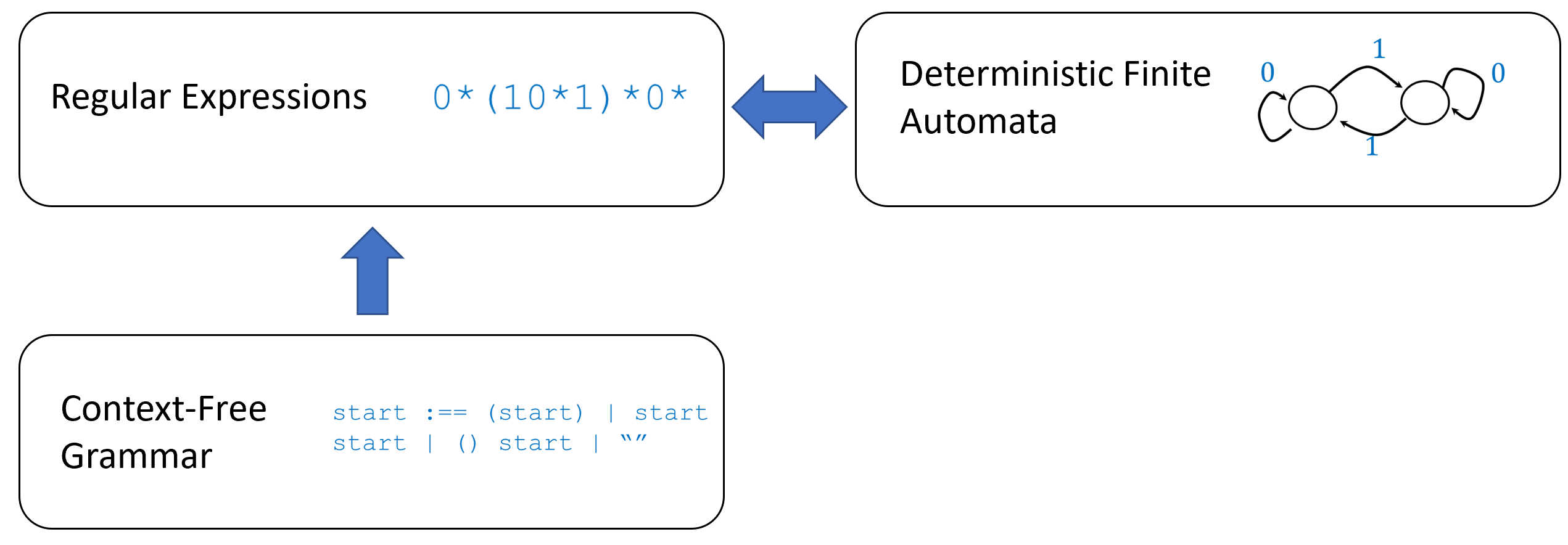
* See that Turing completeness is not always a good thing
* Two important examples of non-Turing-complete, always-halting formalisms: *regular expressions* and *context-free grammars*.
* The pumping lemmas for both these formalisms, and examples of non regular and non context-free functions.
* Examples of computable and uncomputable *semantic properties* of regular expressions and context-free grammars.

*“Happy families are all alike; every unhappy family is unhappy in its own way”*, Leo Tolstoy (opening of the book “Anna Karenina”).

We have seen that many models of computation are *Turing equivalent*, including Turing machines, NAND-TM/NAND-RAM programs, standard programming languages such as C/Python/Javascript, as well as other models such as the calculus and even the game of life. The flip side of this is that for all these models, Rice’s theorem (rice-thm) holds as well, which means that any semantic property of programs in such a model is *uncomputable*.

The uncomputability of halting and other semantic specification problems for Turing equivalent models motivates **restricted computational models** that are **(a)** powerful enough to capture a set of functions useful for certain applications but **(b)** weak enough that we can still solve semantic specification problems on them. In this chapter we discuss several such examples.

We can use *restricted computational models* to bypass limitations such as uncomputability of the Halting problem and Rice’s Theorem. Such models can compute only a restricted subclass of functions, but allow to answer at least some *semantic questions* on programs.



Some restricted computational models we study in this chapter. We show two equivalent models of computation: regular expressions and deterministic finite automata. We show a more powerful model: context-free grammars. We also present tools to demonstrate that some functions *can not* be computed in these models.

## Turing completeness as a bug

We have seen that seemingly simple computational models or systems can turn out to be Turing complete. The [following webpage](https://goo.gl/xRXq7p) lists several examples of formalisms that “accidentally” turned out to Turing complete, including supposedly limited languages such as the C preprocessor, CSS, (certain variants of) SQL, sendmail configuration, as well as games such as Minecraft, Super Mario, and the card game “Magic: The Gathering”. Turing completeness is not always a good thing, as it means that such formalisms can give rise to arbitrarily complex behavior. For example, the postscript format (a precursor of PDF) is a Turing-complete programming language meant to describe documents for printing. The expressive power of postscript can allow for short descriptions of very complex images, but it also gave rise to some nasty surprises, such as the attacks described in [this page](http://hacking-printers.net/wiki/index.php/PostScript) ranging from using infinite loops as a denial of service attack, to accessing the printer’s file system.

An interesting recent example of the pitfalls of Turing-completeness arose in the context of the cryptocurrency [Ethereum](https://www.ethereum.org/). The distinguishing feature of this currency is the ability to design “smart contracts” using an expressive (and in particular Turing-complete) programming language. In our current “human operated” economy, Alice and Bob might sign a contract to agree that if condition X happens then they will jointly invest in Charlie’s company. Ethereum allows Alice and Bob to create a joint venture where Alice and Bob pool their funds together into an account that will be governed by some program that decides under what conditions it disburses funds from it. For example, one could imagine a piece of code that interacts between Alice, Bob, and some program running on Bob’s car that allows Alice to rent out Bob’s car without any human intervention or overhead.

Specifically Ethereum uses the Turing-complete programming language [solidity](https://solidity.readthedocs.io/en/develop/index.html) which has a syntax similar to JavaScript. The flagship of Ethereum was an experiment known as The “Decentralized Autonomous Organization” or [The DAO](https://goo.gl/NegW77). The idea was to create a smart contract that would create an autonomously run decentralized venture capital fund, without human managers, where shareholders could decide on investment opportunities. The DAO was at the time the biggest crowdfunding success in history. At its height the DAO was worth 150 million dollars, which was more than ten percent of the total Ethereum market. Investing in the DAO (or entering any other “smart contract”) amounts to providing your funds to be run by a computer program. i.e., “code is law”, or to use the words the DAO described itself: *“The DAO is borne from immutable, unstoppable, and irrefutable computer code”*. Unfortunately, it turns out that (as we saw in chapcomputable) understanding the behavior of computer programs is quite a hard thing to do. A hacker (or perhaps, some would say, a savvy investor) was able to fashion an input that caused the DAO code to enter into an infinite recursive loop in which it continuously transferred funds into the hacker’s account, thereby [cleaning out about 60 million dollars](https://www.bloomberg.com/features/2017-the-ether-thief/) out of the DAO. While this transaction was “legal” in the sense that it complied with the code of the smart contract, it was obviously not what the humans who wrote this code had in mind. The Ethereum community struggled with the response to this attack. Some tried to the “Robin Hood” approach of using the same loophole to drain the DAO funds into a secure account, but it only had limited success. Eventually, the Ethereum community decided that the code can be mutable, stoppable, and refutable. Specifically, the Ethereum maintainers and miners agreed on a “hard fork” (also known as a “bailout”) to revert history to before the hacker’s transaction occurred. Some community members strongly opposed this decision, and so an alternative currency called [Ethereum Classic](https://ethereumclassic.github.io/) was created that preserved the original history.

## Regular expressions

*Searching* for a piece of text is a common task in computing. At its heart, the *search problem* is quite simple. We have a collection of strings (e.g., files on a hard-drive, or student records in a database), and the user wants to find out the subset of all the that are *matched* by some pattern (e.g., all files whose names end with the string .txt). In full generality, we can allow the user to specify the pattern by specifying a (computable) *function* , where corresponds to the pattern matching . That is, the user provides a *program* in some Turing-complete programming language such as *Python*, and the system will return all the such that . For example, one could search for all text files that contain the string important document or perhaps (letting correspond to a neural-network based classifier) all images that contain a cat. However, we don’t want our system to get into an infinite loop just trying to evaluate the program !

Because the Halting problem for Turing-complete computational models is uncomputable, we cannot in general verify that a given program will halt on a given input. For this reason, typical systems for searching files or databases do *not* allow users to specify the patterns using full-fledged programming languages. Rather, such systems use *restricted computational models* that on the one hand are *rich enough* to capture many of the queries needed in practice (e.g., all filenames ending with .txt, or all phone numbers of the form (617) xxx-xxxx), but on the other hand are *restricted* enough so that they cannot result in an infinite loop.

One of the most popular such computational models is [regular expressions](https://goo.gl/2vTAFU). If you ever used an advanced text editor, a command line shell, or have done any kind of manipulation of text files, then you have probably come across regular expressions.

A *regular expression* over some alphabet is obtained by combining elements of with the operation of concatenation, as well as (corresponding to *or*) and (corresponding to repetition zero or more times). (Common implementations of regular expressions in programming languages and shells typically include some extra operations on top of and , but these operations can be implemented as “syntactic sugar” using the operators and .) For example, the following regular expression over the alphabet corresponds to the set of all strings where every digit is repeated at least twice:

The following regular expression over the alphabet corresponds to the set of all strings that consist of a sequence of one or more of the letters - followed by a sequence of one or more digits (without a leading zero):

Formally, regular expressions are defined by the following recursive definition:

A *regular expression* over an alphabet is a string over $\Sigma \cup \{ (,),|,\*,\emptyset, "" \}$ that has one of the following forms:

1. where
2. where are regular expressions.
3. where are regular expressions. (We often drop the parentheses when there is no danger of confusion and so write this as .)
4. where is a regular expression.

Finally we also allow the following “edge cases”: and $e = ""$. These are the regular expressions corresponding to accepting no strings, and accepting only the empty string respectively.

We will drop parenthesis when they can be inferred from the context. We also use the convention that OR and concatenation are left-associative, and give higher precedence to , then concatenation, and then OR. Thus for example we write instead of .

Every regular expression corresponds to a function where if *matches* the regular expression. For example, if then but (can you see why?).

The formal definition of is one of those definitions that is more cumbersome to write than to grasp. Thus it might be easier for you to first work it out on your own and then check that your definition matches what is written below.

Let be a regular expression over the alphabet . The function is defined as follows:

1. If then iff .
2. If then where is the OR operator.
3. If then iff there is some such that is the concatenation of and and .
4. If then iff there are is and some such that is the concatenation and for every .
5. Finally, for the edge cases is the constant zero function, and $\Phi\_{""}$ is the function that only outputs on the empty string $""$.

We say that a regular expression over *matches* a string if . We say that a function is *regular* if for some regular expression .[[1]](#footnote-38)

### 

The definitions above are not inherently difficult, but are a bit cumbersome. So you should pause here and go over it again until you understand why it corresponds to our intuitive notion of regular expressions. This is important not just for understanding regular expressions themselves (which are used time and again in a great many applications) but also for getting better at understanding recursive definitions in general.

Let and be the function such that outputs iff consists of one or more of the letters - followed by a sequence of one or more digits (without a leading zero). Then is a regular function, since where

is the expression we saw in regexpeq.

If we wanted to verify, for example, that , we can do so by noticing that the expression matches the string , matches , matches the string , and the expression matches the string . Each one of those boils down to a simpler expression. For example, the expression matches the string because both of the one-character strings and are matched by the expression .

Regular expression can be defined over any finite alphabet , but as usual, we will focus our attention on the *binary case*, where . Most (if not all) of the theoretical and practical general insights about regular expressions can be gleaned from studying the binary case.

We can think of regular expressions as a type of “programming language”. That is, we can think of a regular expression over the alphabet as a program that computes the function . (You can also think of regular expressions as *generative models*, since you can think of them as giving a recipe how to generate strings that match them.) This “regular expression programming language” is simpler than general programming languages, in the sense that for every regular expression , the function is computable (and so in particular can be evaluated by an always-halting Turing machine).

For every regular expression over , the function is computable.

That is, there is a Turing machine such that for every , on input , halts with the output .

We state regularexphalt for regular expressions over the binary alphabet , but it generalizes to any finite alphabet .

### 

The proof relies on the observation that matchingregexpdef actually specifies a recursive algorithm for *computing* . Specifically, each one of our operations -concatenation, OR, and star- can be thought of as reducing the task of testing whether an expression matches a string to testing whether some sub-expressions of match substrings of . Since these sub-expressions are always shorter than the original expression, this yields a recursive algorithm for checking if matches which will eventually terminate at the base cases of the expressions that correspond to a single symbol or the empty string.

matchingregexpdef gives a way of recursively computing . The key observation is that in our recursive definition of regular expressions, whenever is made up of one or two expressions then these two regular expressions are *smaller* than , and eventually (when they have size ) then they must correspond to the non-recursive case of a single alphabet symbol.

Therefore, we can prove the theorem by induction over the length of (i.e., the number of symbols in the string , also denoted as ). For , is either a single alphabet symbol, $""$ or , and so computing the function is straightforward. In the general case, for we assume by the induction hypothesis that we have proven the theorem for all expressions of length smaller than . Now, such an expression of length larger than one can obtained through one of three cases: OR, concatenation, or star operations. We now show that will be computable in all these cases:

**Case 1:** where are shorter regular expressions.

In this case by the inductive hypothesis we can compute and and so can compute as (where is the OR operator).

**Case 2:** where are regular expressions.

In this case by the inductive hypothesis we can compute and and so can compute as

where is the AND operator and for , refers to the empty string.

**Case 3:** where is a regular expression.

In this case by the inductive hypothesis we can compute and so we can compute by enumerating over all from to , and all ways to write as the concatenation of nonempty strings (we can do so by enumerating over all possible positions in which one string stops and the other begins). If for one of those partitions, then we output . Otherwise we output . We can restrict attention to partitions of as where all the ’s are nonempty since if some of the ’s are empty we can simply drop them and still be left with a valid partition.

These three cases exhaust all the possibilities for an expression of length larger than one, and hence this completes the proof.

## Deterministic finite automata, and efficient matching of regular expressions (optional)

The proof of regularexphalt gives a recursive algorithm to evaluate whether a given string matches or not a regular expression. But it is not a very efficient algorithm.

However, it turns out that there is a much more efficient algorithm that can match regular expressions in *linear* (i.e., ) time. Since we have not yet covered the topics of time and space complexity, we describe this algorithm in high level terms, without making the computational model precise, using the colloquial notion of running time as is used in introduction to programming courses and whiteboard coding interviews. We will see a formal definition of time complexity in chapmodelruntime.

Let be a regular expression. Then there is an time algorithm that computes .

The implicit constant in the term of reglintimethm depends on the expression . Thus, another way to state reglintimethm is that for every expression , there is some constant and an algorithm that computes on -bit inputs using at most steps. This makes sense, since in practice we often want to compute for a small regular expression and a large document . reglintimethm tells us that we can do so with running time that scales linearly with the size of the document, even if it has (potentially) worse dependence on the size of the regular expression.

The idea is to define a more efficient recursive algorithm, that determines whether matches a string by reducing this task to determining whether a related expression matches . This will result in an expression for the running time of the form which solves to .

The central definition for this proof is the notion of a *restriction* of a regular expression. The idea is that for every regular expression and symbol in its alphabet, it is possible to define a regular expression such that matches a string if and only if matches the string . For example, if is the regular expression (i.e., one or more occurrences of ) then is equal to and will be . (Can you see why?)

For simplicity, from now on we fix our attention to the case that the alphabet is . Given a regular expression and , we can compute recursively as follows:

1. If consists of a single symbol (i.e.  for ) then $e[\sigma]=""$ if and otherwise.
2. If then .
3. If then if can not match the empty string. Otherwise,
4. If then .
5. If $e = ""$ or then .

By checking all these cases, one can verify that it is indeed the case that for every regular expression , and , matches if and only if matches . We let denote the time to compute for regular expressions of length at most . The value can be shown to be polynomial in , though this is not important for this theorem, since we only care about the dependence of the time to compute on the length of and not about the dependence of this time on the length of .

Using this notion of restriction, we can define the following recursive algorithm for regular expression matching:

**Input:** Regular expression over and for .

**Goal:** Compute

**Operation:**

1. If $x=""$ then return if and only if $\Phi\_e("")=1$. (This can be either computed directly or using the algorithm of regularexphalt in time which is a constant depending only on the regular expression .)
2. Otherwise, compute recursively and output the result.

By the definition of a restriction, for every and , the expression matches if and only if matches . Hence for every and , and regexpmatchlinearalg does return the correct answer. The only remaining task is to analyze its *running time*.

regexpmatchlinearalg is a recursive algorithm that on input an expression and a string , does some constant time computation and then calls itself on input some expression and a string of length . It will terminate after steps when it reaches a string of length . So, to calculate the running time of regexpmatchlinearalg we need to analyze the cost of each step.

Specifically, the running time that it takes for regexpmatchlinearalg to compute for inputs of length satisfies the recursive equation:

where , as before, denotes the time to compute for expressions of length at most . (In the base case , is equal to some constant depending only on .)

To get some intuition for the expression matchregexprecursion, let us open up the recursion for one level, writing as

Continuing this way, we can see that where is the largest length of any expression that we encounter along the way. Therefore, the following claim suffices to show that regexpmatchlinearalg runs in linear time:

**Claim:** Let be a regular expression over , then there is some constant such that for every string , if we restrict to , and then to and so on and so forth, the resulting expression has length at most .

**Proof of claim:** For a regular expression over and , we denote by the expression obtained by restricting to and then to and so on. We let . We will prove the claim by showing that for every , the set is finite, and hence so is the number which is the maximum length of for .

We prove this by induction on the structure of . If is a symbol, the empty string, or the empty set, then this is straightforward to show as the most expressions can contain are the expression itself, $""$, and . Otherwise we split to the two cases **(i)** and **(ii)** , where are smaller expressions (and hence by the induction hypothesis and are finite). In the case **(i)**, if then is either equal to or it is simply the empty set if . Since is in the set , the number of distinct expressions in is at most . In the case **(ii)**, if then all the restrictions of to strings will either have the form or the form where is some string such that and matches the empty string. Since and , the number of the possible distinct expressions of the form is at most . This completes the proof of the claim.

The bottom line is that while running regexpmatchlinearalg on a regular expression , all the expressions we ever encounter are in the finite set , no matter how large the input is, and so the running time of regexpmatchlinearalg satisfies the equation for some constant depending on . This solves to where the implicit constant in the Oh notation can (and will) depend on but crucially, not on the length of the input .

### Matching regular expressions using constant memory

reglintimethm is already quite impressive, but we can do even better. Specifically, no matter how long the string is, we can compute by maintaining only a constant amount of memory and moreover making a *single pass* over . That is, the algorithm will scan the input once from start to finish, and then determine whether or not is matched by the expression . This is important in the common case of trying to match a short regular expression over a huge file or document that might not even fit in our computer’s memory. A single-pass constant-memory algorithm is also known as a [deterministic finite automaton (DFA)](https://goo.gl/SG6DS7) (see secdfa). There is a beautiful theory on the properties of DFA’s and their connections with regular expressions. In particular, as we’ll see in dfaregequivthm, a function is regular *if and only if* it can be computed by a DFA. We start with showing the “only if” direction:

### 

Let be a regular expression. Then there is an algorithm that on input computes while making a single pass over and maintaining a constant amount of memory.

### 

The idea is to replace the recursive algorithm of regexpmatchlinearalg with a [dynamic program](https://goo.gl/kgLdX1), using the technique of [memoization](https://en.wikipedia.org/wiki/Memoization). If you haven’t taken yet an algorithms course, you might not know these techniques. This is OK; while this more efficient algorithm is crucial for the many practical applications of regular expressions, it is not of great importance for this book.

We will replace the recursive regexpmatchlinearalg with the following iterative algorithm:

**Input:** Regular expression over , string .

**Goals:** Compute .

**Operation:**

1. Let be the set as defined in the proof of reglintimethm. Note that is finite and by definition, for every and , is in as well.
2. Define a Boolean variable for every . Initially we set if and only if matches the empty string.
3. For do the following:
   1. *Copy the variables to temporary variables:* For every , we set .
   2. *Update the variables based on the -th bit of :* Let and set for every .
4. Output .

iterregexpmatchlinearalg maintains the invariant that at the end of step , for every , the variable is equal if and only if matches the string . In particular, at the very end, is equal to if and only if matches the full string . iterregexpmatchlinearalg only maintains a constant number of variables (as is finite), and that it proceeds in one linear scan over the input, and so this proves the theorem.

### Deterministic Finite Automata

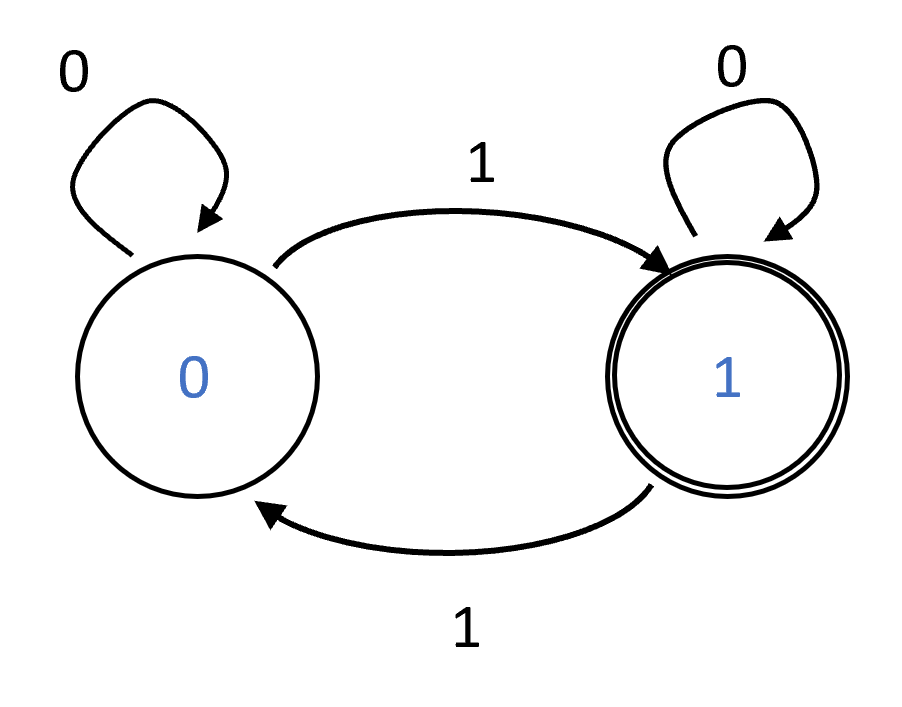
In Computer Science, a single-pass constant-memory algorithm is also known as a *Deterministic Finite Automaton* or *DFA* (another name for DFA’s is a *finite state machine*). That is, we can think of such an algorithm as a “machine” that can be in one of states, for some constant . The machine starts in some initial state, and then reads its input one bit at a time. Whenever the machine reads a bit , it transitions into a new state based on and its prior state. The output of the machine is based on the final state. Every constant-memory one-pass algorithm corresponds to such a machine. If an algorithm uses bits of memory, then the contents of its memory are a string of length . Since there are such strings, at any point in the execution, such an algorithm can be in one of states.

Here is a DFA for computing the function that maps to .

We will have two states: and . The set of accepting states is , and if we are in a state and read the bit , we will transition to the state if and to the state if . In other words, we transition to the state . Hence we can think of this algorithm’s execution on input as follows:

* Let be the state of the automaton at step . We initialize .
* For every , let .
* Output .

You can verify that the output of this algorithm is . We can also describe this DFA graphically, see xorautomatonfig.



A deterministic finite automaton that computes the function. It has two states and , and when it observes it transitions from to .

The formal definition of a DFA is the following:

A deterministic finite automaton (DFA) with states over is a pair with and . The function is known as the *transition function* of the DFA and the set is known as the set of *accepting states*.

We say that *computes* a function if for every and , if we define and for every , then

Our treatment of automata in this book is quite brief. If you find this definition confusing, there are plenty of resources that help you get more comfortable with DFA’s. In particular, Chapter 1 of Sipser’s book [@SipserBook] contains an excellent exposition of this material. There are also many websites with online simulators for automata, as well as translators from regular expressions to automata and vice versa (see for example [here](http://ivanzuzak.info/noam/webapps/fsm2regex/) and [here](https://cyberzhg.github.io/toolbox/nfa2dfa)).

Sipser defines a DFAs as a five-tuple where is the set of states, is the alphabet, is the transition function, is the initial state, and is the set of accepting states. In this book the set of states is always of the form and the initial state is always , but this makes no difference to the computational power of these models. Also, we restrict our attention to the case that the alphabet is equal to .

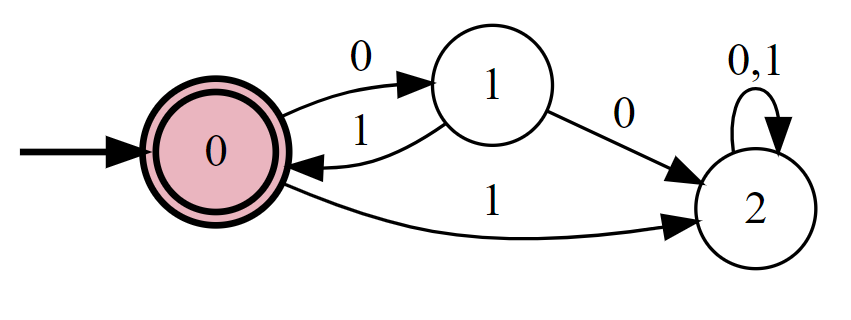
The following theorem is the central result of automata theory:

### 

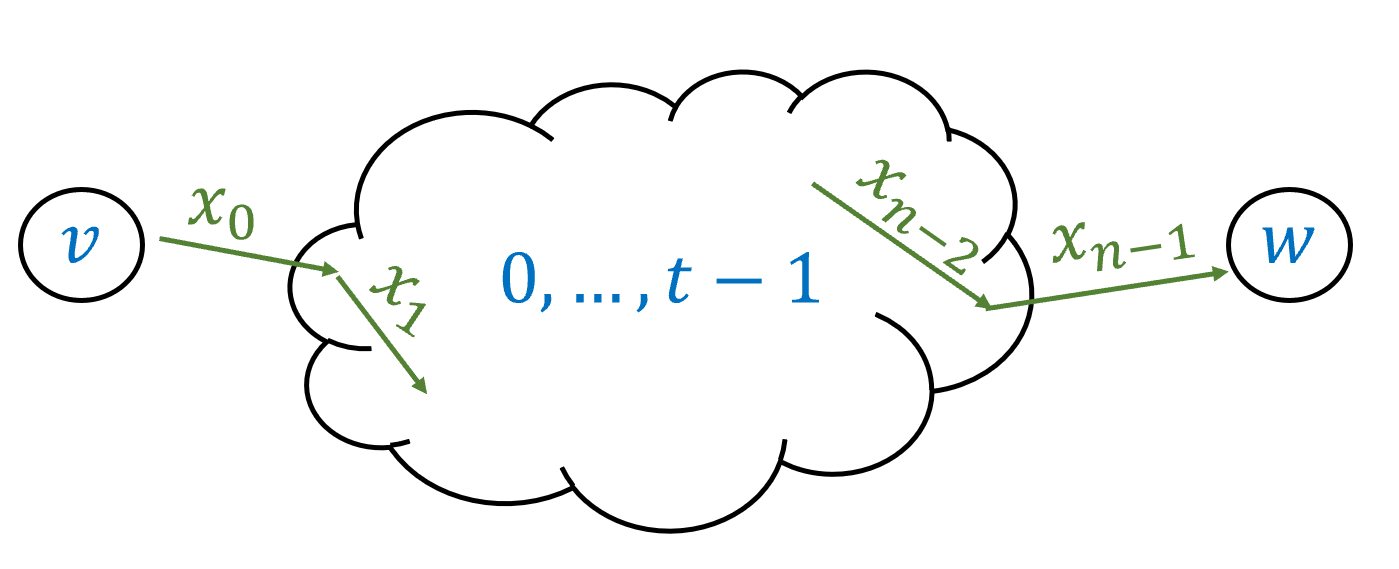
Let . Then is regular if and only if there exists a DFA that computes .

### 

One direction follows from DFAforREGthm, which shows that for every regular expression , the function can be computed by a DFA (see for example automatonregfig). For the other direction, we show that given a DFA for every we can find a regular expression that would match if and only if the DFA starting in state , will end up in state after reading .



A deterministic finite automaton that computes the function .



Given a DFA of states, for every and number we define the function to output one on input if and only if when the DFA is initialized in the state and is given the input , it will teach the state while going only through the intermediate states .

Since DFAforREGthm proves the “only if” direction, we only need to show the “if” direction. Let be a DFA with states that computes the function . We need to show that is regular.

For every , we let be the function that maps to if and only if the DFA , starting at the state , will reach the state if it reads the input . We will prove that is regular for every . This will prove the theorem, since by DFAdef, is equal to the OR of for every . Hence if we have a regular expression for every function of the form then (using the operation) we can obtain a regular expression for as well.

To give regular expressions for the functions , we start by defining the following functions : for every and , if and only if starting from and observing , the automata reaches *with all intermediate states being in the set*  (see dfatoregonefig). That is, while themselves might be outside , if and only if throughout the execution of the automaton on the input (when initiated at ) it never enters any of the states outside and still ends up at . If then is the empty set, and hence if and only if the automaton reaches from directly on , without any intermediate state. If then all states are in , and hence .

We will prove the theorem by induction on , showing that is regular for every and . For the **base case** of , is regular for every since it can be described as one of the expressions $""$, , , or . Specifically, if then if and only if is the empty string. If then if and only if consists of a single symbol and . Therefore in this case corresponds to one of the four regular expressions , , or , depending on whether transitions to from when it reads either or , only one of these symbols, or neither.

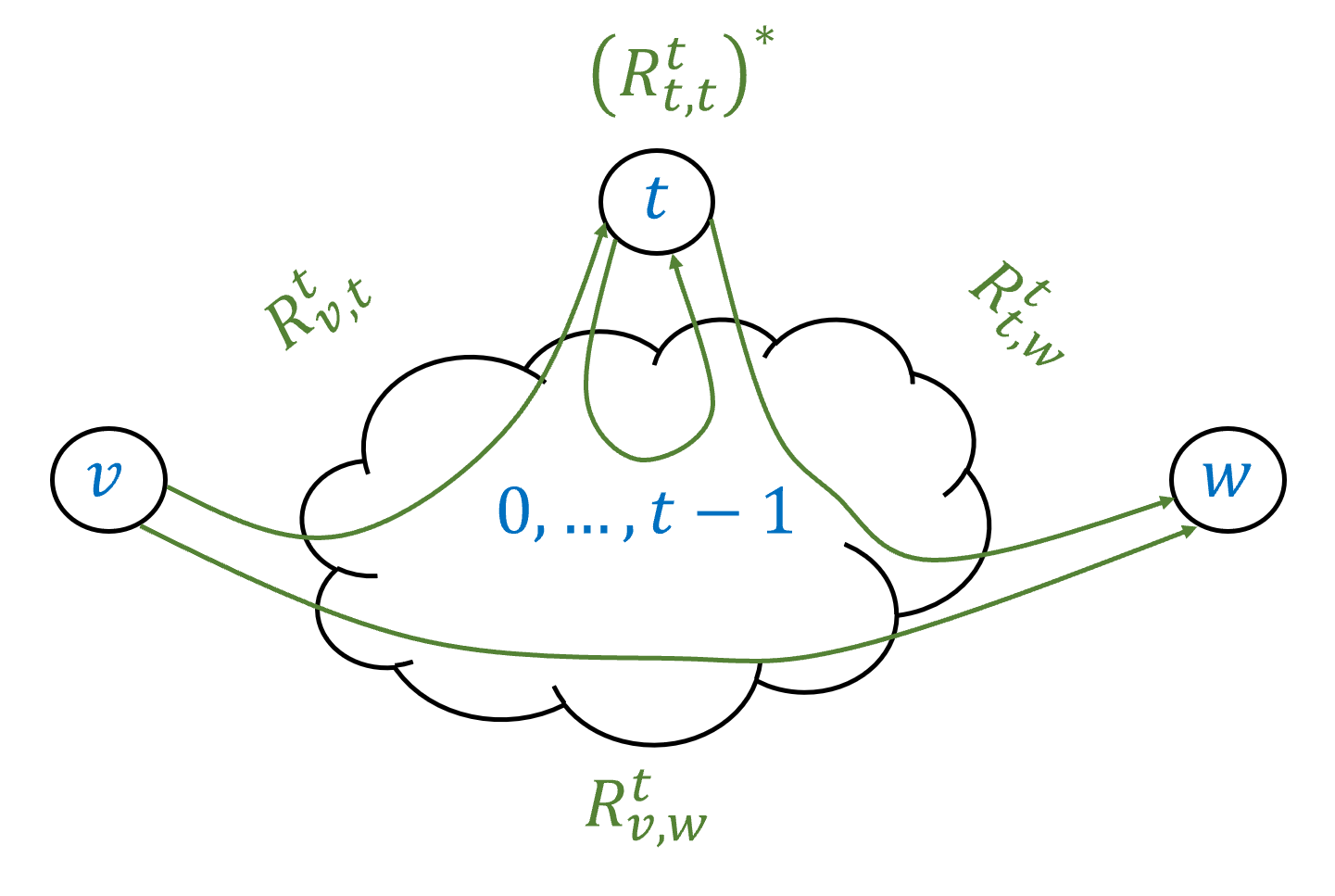
**Inductive step:** Now that we’ve seen the base case, let’s prove the general case by induction. Assume, via the induction hypothesis, that for every , we have a regular expression that computes . We need to prove that is regular for every . If the automaton arrives from to using the intermediate states , then it visits the -th state zero or more times. If the path labeled by causes the automaton to get from to without visiting the -th state at all, then is matched by the regular expression . If the path labeled by causes the automaton to get from to while visiting the -th state times then we can think of this path as:

* First travel from to using only intermediate states in .
* Then go from back to itself using only intermediate states in
* Then go from to using only intermediate states in .

Therefore in this case the string is matched by the regular expression . (See also dfatoreginductivefig.)

Therefore we can compute using the regular expression

This completes the proof of the inductive step and hence of the theorem.



If we have regular expressions corresponding to for every , we can obtain a regular expression corresponding to . The key observation is that a path from to using either does not touch at all, in which case it is captured by the expression , or it goes from to , comes back to zero or more times, and then goes from to , in which case it is captured by the expression .

### Regular functions are closed under complement

Here is an important corollary of dfaregequivthm:

### 

If is regular then so is the function , where for every .

If is regular then by reglintimethm it can be computed by a constant-space one-pass algorithm . But then the algorithm which does the same computation and outputs the negation of the output of also utilizes constant space and one pass and computes . By dfaregequivthm this implies that is regular as well.

## Limitations of regular expressions

The fact that functions computed by regular expressions always halt is one of the reasons why they are so useful. When you make a regular expression search, you are guaranteed that that it will terminate with a result. This is why operating systems and text editors often restrict their search interface to regular expressions and don’t allow searching by specifying an arbitrary function. But this always-halting property comes at a cost. Regular expressions cannot compute every function that is computable by Turing machines. In fact there are some very simple (and useful!) functions that they cannot compute. Here is one example:

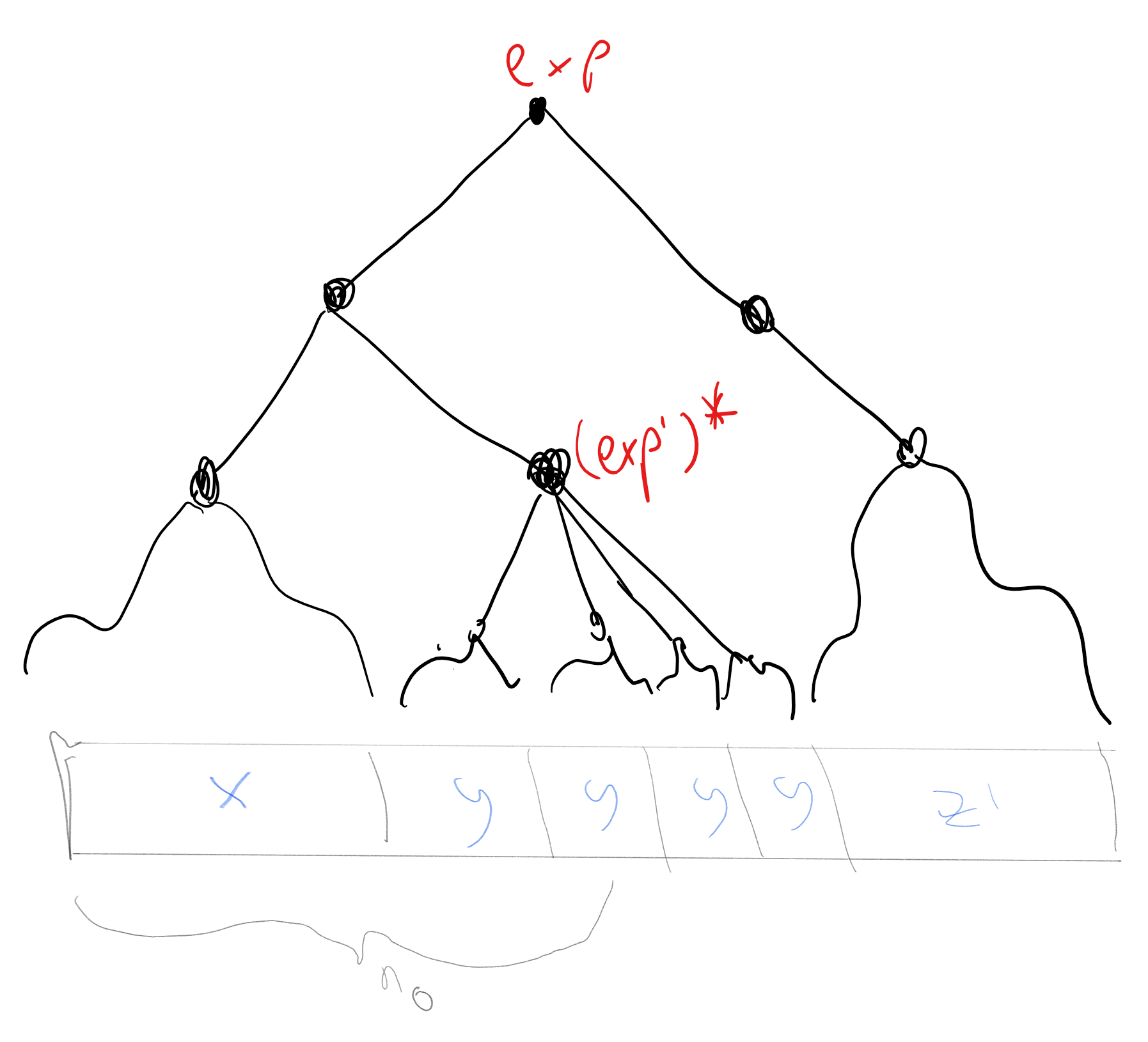
### 

Let and be the function that given a string of parentheses, outputs if and only if every opening parenthesis is matched by a corresponding closed one. Then there is no regular expression over that computes .

regexpparn is a consequence of the following result, which is known as the *pumping lemma*:

Let be a regular expression over some alphabet . Then there is some number such that for every with and , we can write for strings satisfying the following conditions:

1. .
2. .
3. for every .



To prove the “pumping lemma” we look at a word that is much larger than the regular expression that matches it. In such a case, part of must be matched by some sub-expression of the form , since this is the only operator that allows matching words longer than the expression. If we look at the “leftmost” such sub-expression and define to be the string that is matched by it, we obtain the partition needed for the pumping lemma.

### 

The idea behind the proof the following. Let be twice the number of symbols that are used in the expression , then the only way that there is some with and is that contains the (i.e. star) operator and that there is a nonempty substring of that was matched by for some sub-expression of . We can now repeat any number of times and still get a matching string. See also pumpinglemmafig.

The pumping lemma is a bit cumbersome to state, but one way to remember it is that it simply says the following: *“if a string matching a regular expression is long enough, one of its substrings must be matched using the operator”*.

To prove the lemma formally, we use induction on the length of the expression. Like all induction proofs, this is going to be somewhat lengthy, but at the end of the day it directly follows the intuition above that *somewhere* we must have used the star operation. Reading this proof, and in particular understanding how the formal proof below corresponds to the intuitive idea above, is a very good way to get more comfortable with inductive proofs of this form.

Our inductive hypothesis is that for an length expression, satisfies the conditions of the lemma. The **base case** is when the expression is a single symbol or that the expression is or $""$. In all these cases the conditions of the lemma are satisfied simply because there and there is no string of length larger than that is matched by the expression.

We now prove the **inductive step**. Let be a regular expression with symbols. We set and let be a string satisfying . Since has more than one symbol, it has one of the the forms **(a)** , **(b)**, , or **(c)** where in all these cases the subexpressions and have fewer symbols than and hence satisfy the induction hypothesis.

In the case **(a)**, every string matched by must be matched by either or . If matches then, since , by the induction hypothesis there exist with and such that (and therefore also ) matches for every . The same arguments works in the case that matches .

In the case **(b)**, if is matched by then we can write where matches and matches . We split to subcases. If then by the induction hypothesis there exist with , such that and matches for every . This completes the proof since if we set then we see that and matches for every . Otherwise, if then since , it must be that . Hence by the induction hypothesis there exist such that , and matches for every . But now if we set we see that and on the other hand the expression matches for every .

In case **(c)**, if is matched then where for every , is a nonempty string matched by . If then we can use the same approach as in the concatenation case above. Otherwise, we simply note that if is the empty string, , and then and is matched by for every .

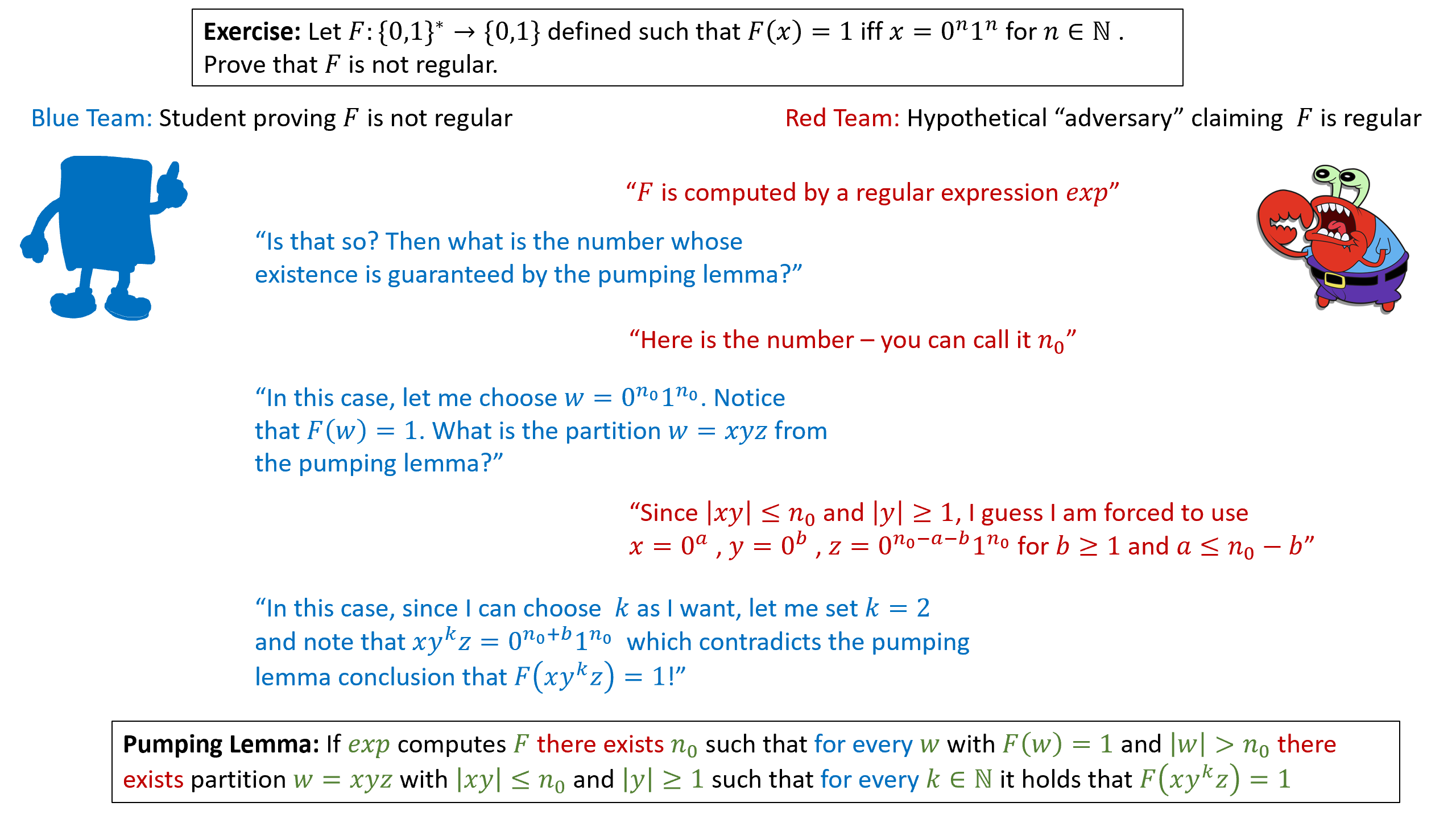
### 

When an object is *recursively defined* (as in the case of regular expressions) then it is natural to prove properties of such objects by *induction*. That is, if we want to prove that all objects of this type have property , then it is natural to use an inductive steps that says that if etc have property then so is an object that is obtained by composing them.

Using the pumping lemma, we can easily prove regexpparn (i.e., the non-regularity of the “matching parenthesis” function):

Suppose, towards the sake of contradiction, that there is an expression such that . Let be the number obtained from pumping and let (i.e., left parenthesis followed by right parenthesis). Then we see that if we write as in regexpparn, the condition implies that consists solely of left parenthesis. Hence the string will contain more left parenthesis than right parenthesis. Hence but by the pumping lemma , contradicting our assumption that .

The pumping lemma is a very useful tool to show that certain functions are *not* computable by a regular expression. However, it is *not* an “if and only if” condition for regularity: there are non regular functions that still satisfy the conditions of the pumping lemma. To understand the pumping lemma, it is important to follow the order of quantifiers in pumping. In particular, the number in the statement of pumping depends on the regular expression (in the proof we chose to be twice the number of symbols in the expression). So, if we want to use the pumping lemma to rule out the existence of a regular expression computing some function , we need to be able to choose an appropriate input that can be arbitrarily large and satisfies . This makes sense if you think about the intuition behind the pumping lemma: we need to be large enough as to force the use of the star operator.



A cartoon of a proof using the pumping lemma that a function is not regular. The pumping lemma states that if is regular then *there exists* a number such that *for every* large enough with , *there exists* a partition of to satisfying certain conditions such that *for every* , . You can imagine a pumping-lemma based proof as a game between you and the adversary. Every *there exists* quantifier corresponds to an object you are free to choose on your own (and base your choice on previously chosen objects). Every *for every* quantifier corresponds to an object the adversary can choose arbitrarily (and again based on prior choices) as long as it satisfies the conditions. A valid proof corresponds to a strategy by which no matter what the adversary does, you can win the game by obtaining a contradiction which would be a choice of that would result in , hence violating the conclusion of the pumping lemma.

Prove that the following function over the alphabet is not regular: if and only if where and denotes “reversed”: the string . (The *Palindrome* function is most often defined without an explicit separator character , but the version with such a separator is a bit cleaner and so we use it here. This does not make much difference, as one can easily encode the separator as a special binary string instead.)

We use the pumping lemma. Suppose towards the sake of contradiction that there is a regular expression computing , and let be the number obtained by the pumping lemma (pumping). Consider the string . Since the reverse of the all zero string is the all zero string, . Now, by the pumping lemma, if is computed by , then we can write such that , and for every . In particular, it must hold that , but this is a contradiction, since and so its two parts are not of the same length and in particular are not the reverse of one another.

For yet another example of a pumping-lemma based proof, see pumpingprooffig which illustrates a cartoon of the proof of the non-regularity of the function which is defined as iff for some (i.e., consists of a string of consecutive zeroes, followed by a string of consecutive ones of the same length).

## Other semantic properties of regular expressions

Regular expressions are widely used beyond just searching. For example, regular expressions are often used to define *tokens* (such as what is a valid variable identifier, or keyword) in programming languages. But they also have other uses. One nice example is the recent work on the [NetKAT network programming language](https://goo.gl/oeJNuw). In recent years, the world of networking moved from fixed topologies to “software defined networks”. These are run by programmable switches that can implement policies such as “if packet is secured by SSL then forward it to A, otherwise forward it to B”. By its nature, one would want to use a formalism for such policies that is guaranteed to always halt (and quickly!) and such that it is possible to answer semantic questions such as “does C see the packets moved from A to B” etc. The NetKAT language uses a variant of regular expressions to achieve precisely that.

Such applications use the fact that because regular expressions are so restricted, we can not only solve the halting problem for them, but also answer other *semantic questions*. Such semantic questions would not be solvable for Turing-complete models due to Rice’s Theorem (rice-thm). For example, we can tell whether two regular expressions are *equivalent*, as well as whether a regular expression computes the constant zero function.

### 

There is an algorithm that given a regular expression , outputs if and only if is the constant zero function.

### 

The idea is that we can directly observe this from the structure of the expression. The only way a regular expression computes the constant zero function is if has the form or is obtained by concatenating with other expressions.

Define a regular expression to be “empty” if it computes the constant zero function. Given a regular expression , we can determine if is empty using the following rules:

* If has the form or $""$ then it is not empty.
* If is not empty then is not empty for every .
* If is not empty then is not empty.
* If and are both not empty then is not empty.
* is empty.

Using these rules it is straightforward to come up with a recursive algorithm to determine emptiness.

### 

Let be the function that on input (a string representing) a pair of regular expressions , if and only if . Then is computable.

### 

The idea is to show that given a pair of regular expression and we can find an expression such that if and only if . Therefore is the constant zero function if and only if and are equivalent, and thus we can test for emptiness of to determine equivalence of and .

We will prove regequivalencethm from regemptynessthm. (The two theorems are in fact equivalent: it is easy to prove regemptynessthm from regequivalencethm, since checking for emptiness is the same as checking equivalence with the expression .) Given two regular expressions and , we will compute an expression such that if and only if . One can see that is equivalent to if and only if is empty.

We start with the observation that for every bit , if and only if

Hence we need to construct such that for every ,

To construct the expression , we will show how given any pair of expressions and , we can construct expressions and that compute the functions and respectively. (Computing the expression for is straightforward using the operation of regular expressions.)

Specifically, by regcomplementlem, regular functions are closed under negation, which means that for every regular expression , there is an expression such that for every . Now, for every two expression and , the expression

computes the AND of the two expressions. Given these two transformations, we see that for every regular expressions and we can find a regular expression satisfying eqemptyequivreg such that is empty if and only if and are equivalent.

## Context free grammars

If you have ever written a program, you’ve experienced a *syntax error*. You probably also had the experience of your program entering into an *infinite loop*. What is less likely is that the compiler or interpreter entered an infinite loop while trying to figure out if your program has a syntax error.

When a person designs a programming language, they need to determine its *syntax*. That is, the designer decides which strings corresponds to valid programs, and which ones do not (i.e., which strings contain a syntax error). To ensure that a compiler or interpreter always halts when checking for syntax errors, language designers typically *do not* use a general Turing-complete mechanism to express their syntax. Rather they use a *restricted* computational model. One of the most popular choices for such models is *context free grammars*.

To explain context free grammars, let us begin with a canonical example. Consider the function that takes as input a string over the alphabet and returns if and only if the string represents a valid arithmetic expression. Intuitively, we build expressions by applying an operation such as ,, or to smaller expressions, or enclosing them in parenthesis, where the “base case” corresponds to expressions that are simply numbers. More precisely, we can make the following definitions:

* A *digit* is one of the symbols .
* A *number* is a sequence of digits. (For simplicity we drop the condition that the sequence does not have a leading zero, though it is not hard to encode it in a context-free grammar as well.)
* An *operation* is one of
* An *expression* has either the form “*number*”, the form “*sub-expression1 operation sub-expression2*”, or the form “(*sub-expression1*)”, where “sub-expression1” and “sub-expression2” are themselves expressions. (Note that this is a *recursive* definition.)

A context free grammar (CFG) is a formal way of specifying such conditions. A CFG consists of a set of *rules* that tell us how to generate strings from smaller components. In the above example, one of the rules is “if and are valid expressions, then is also a valid expression”; we can also write this rule using the shorthand . As in the above example, the rules of a context-free grammar are often *recursive*: the rule defines valid expressions in terms of itself. We now formally define context-free grammars:

Let be some finite set. A *context free grammar (CFG) over*  is a triple such that:

* , known as the *variables*, is a set disjoint from .
* is known as the *initial variable*.
* is a set of *rules*. Each rule is a pair with and . We often write the rule as and say that the string *can be derived* from the variable .

The example above of well-formed arithmetic expressions can be captured formally by the following context free grammar:

* The alphabet is
* The variables are .
* The rules are the set containing the following rules:
  + The rules , , , and .
  + The rules ,, .
  + The rule .
  + The rule .
  + The rule .
  + The rule .
  + The rule .
* The starting variable is

People use many different notations to write context free grammars. One of the most common notations is the [Backus–Naur form](https://goo.gl/R4qZji). In this notation we write a rule of the form (where is a variable and is a string) in the form <v> := a. If we have several rules of the form , , and then we can combine them as <v> := a|b|c. (In words we say that can derive either , , or .) For example, the Backus-Naur description for the context free grammar of cfgarithmeticex is the following (using ASCII equivalents for operations):

operation := +|-|\*|/  
digit := 0|1|2|3|4|5|6|7|8|9  
number := digit|digit number  
expression := number|expression operation expression|(expression)

Another example of a context free grammar is the “matching parenthesis” grammar, which can be represented in Backus-Naur as follows:

match := ""|match match|(match)

A string over the alphabet (,) can be generated from this grammar (where match is the starting expression and "" corresponds to the empty string) if and only if it consists of a matching set of parenthesis. In contrast, by regexpparn there is no regular expression that matches a string if and only if contains a valid sequence of matching parenthesis.

### Context-free grammars as a computational model

We can think of a context-free grammar over the alphabet as defining a function that maps every string in to or depending on whether can be generated by the rules of the grammars. We now make this definition formally.

If is a context-free grammar over , then for two strings we say that *can be derived in one step* from , denoted by , if we can obtain from by applying one of the rules of . That is, we obtain by replacing in one occurrence of the variable with the string , where is a rule of .

We say that *can be derived* from , denoted by , if it can be derived by some finite number of steps. That is, if there are , so that .

We say that is *matched* by if can be derived from the starting variable (i.e., if ). We define the *function computed by* to be the map such that iff is matched by . A function is *context free* if for some CFG .[[2]](#footnote-92)

A priori it might not be clear that the map is computable, but it turns out that this is the case.

### 

For every CFG over , the function is computable.

As usual we restrict attention to grammars over although the proof extends to any finite alphabet .

We only sketch the proof. We start with the observation we can convert every CFG to an equivalent version of *Chomsky normal form*, where all rules either have the form for variables or the form for a variable and symbol , plus potentially the rule $s \rightarrow ""$ where is the starting variable.

The idea behind such a transformation is to simply add new variables as needed, and so for example we can translate a rule such as into the three rules , and .

Using the Chomsky Normal form we get a natural recursive algorithm for computing whether for a given grammar and string . We simply try all possible guesses for the first rule that is used in such a derivation, and then all possible ways to partition as a concatenation . If we guessed the rule and the partition correctly, then this reduces our task to checking whether and , which (as it involves shorter strings) can be done recursively. The base cases are when is empty or a single symbol, and can be easily handled.

While we focus on the task of *deciding* whether a CFG matches a string, the algorithm to compute actually gives more information than that. That is, on input a string , if then the algorithm yields the sequence of rules that one can apply from the starting vertex to obtain the final string . We can think of these rules as determining a *tree* with being the *root* vertex and the sinks (or *leaves*) corresponding to the substrings of that are obtained by the rules that do not have a variable in their second element. This tree is known as the *parse tree* of , and often yields very useful information about the structure of .

Often the first step in a compiler or interpreter for a programming language is a *parser* that transforms the source into the parse tree (also known as the [abstract syntax tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree)). There are also tools that can automatically convert a description of a context-free grammars into a parser algorithm that computes the parse tree of a given string. (Indeed, the above recursive algorithm can be used to achieve this, but there are much more efficient versions, especially for grammars that have [particular forms](https://en.wikipedia.org/wiki/LR_parser), and programming language designers often try to ensure their languages have these more efficient grammars.)

### The power of context free grammars

Context free grammars can capture every regular expression:

### 

Let be a regular expression over , then there is a CFG over such that .

We prove the theorem by induction on the length of . If is an expression of one bit length, then or , in which case we leave it to the reader to verify that there is a (trivial) CFG that computes it. Otherwise, we fall into one of the following case: **case 1:** , **case 2:** or **case 3:** where in all cases are shorter regular expressions. By the induction hypothesis have grammars and that compute and respectively. By renaming of variables, we can also assume without loss of generality that and are disjoint.

In case 1, we can define the new grammar as follows: we add a new starting variable and the rule . In case 2, we can define the new grammar as follows: we add a new starting variable and the rules and . Case 3 will be the only one that uses *recursion*. As before we add a new starting variable , but now add the rules $s \mapsto ""$ (i.e., the empty string) and also add, for every rule of the form , the rule to .

We leave it to the reader as (a very good!) exercise to verify that in all three cases the grammars we produce capture the same function as the original expression.

It turns out that CFG’s are strictly more powerful than regular expressions. In particular, as we’ve seen, the “matching parenthesis” function can be computed by a context free grammar, whereas, as shown in regexpparn, it cannot be computed by regular expressions. Here is another example:

Let be the function defined in palindromenotreg where iff has the form . Then can be computed by a context-free grammar

A simple grammar computing can be described using Backus–Naur notation:

start := ; | 0 start 0 | 1 start 1

One can prove by induction that this grammar generates exactly the strings such that .

A more interesting example is computing the strings of the form that are *not* palindromes:

Prove that there is a context free grammar that computes where if but .

Using Backus–Naur notation we can describe such a grammar as follows

palindrome := ; | 0 palindrome 0 | 1 palindrome 1  
different := 0 palindrome 1 | 1 palindrome 0  
start := different | 0 start | 1 start | start 0 | start 1

In words, this means that we can characterize a string such that as having the following form

where are arbitrary strings and . Hence we can generate such a string by first generating a palindrome (palindrome variable), then adding either on the right and on the left to get something that is *not* a palindrome (different variable), and then we can add arbitrary number of ’s and ’s on either end (the start variable).

### Limitations of context-free grammars (optional)

Even though context-free grammars are more powerful than regular expressions, there are some simple languages that are *not* captured by context free grammars. One tool to show this is the context-free grammar analog of the “pumping lemma” (pumping):

### 

Let be a CFG over , then there is some numbers such that for every with , if then such that , , and for every .

The context-free pumping lemma is even more cumbersome to state than its regular analog, but you can remember it as saying the following: *“If a long enough string is matched by a grammar, there must be a variable that is repeated in the derivation.”*

We only sketch the proof. The idea is that if the total number of symbols in the rules of the grammar is , then the only way to get with is to use *recursion*. That is, there must be some variable such that we are able to derive from the value for some strings , and then further on derive from some string such that is a substring of (in other words, for some ). If we take the variable satisfying this requirement with a minimum number of derivation steps, then we can ensure that is at most some constant depending on and we can set to be that constant ( will do, since we will not need more than applications of rules, and each such application can grow the string by at most symbols).

Thus by the definition of the grammar, we can repeat the derivation to replace the substring in with for every while retaining the property that the output of is still one. Since is a substring of , we can write and are guaranteed that is matched by the grammar for every .

Using cfgpumping one can show that even the simple function defined as follows:

is not context free. (In contrast, the function defined as iff for some and is context free, can you see why?.)

Let be the function such that if and only if for some . Then is not context free.

We use the context-free pumping lemma. Suppose towards the sake of contradiction that there is a grammar that computes , and let be the constant obtained from cfgpumping.

Consider the string , and write it as as per cfgpumping, with and with . By cfgpumping, it should hold that . However, by case analysis this can be shown to be a contradiction.

Firstly, unless is on the left side of the separator and is on the right side, dropping and will definitely make the two parts different. But if it is the case that is on the left side and is on the right side, then by the condition that we know that is a string of only zeros and is a string of only ones. If we drop and then since one of them is non empty, we get that there are either less zeroes on the left side than on the right side, or there are less ones on the right side than on the left side. In either case, we get that , obtaining the desired contradiction.

## Semantic properties of context free languages

As in the case of regular expressions, the limitations of context free grammars do provide some advantages. For example, emptiness of context free grammars is decidable:

### 

There is an algorithm that on input a context-free grammar , outputs if and only if is the constant zero function.

### 

The proof is easier to see if we transform the grammar to Chomsky Normal Form as in CFGhalt. Given a grammar , we can recursively define a non-terminal variable to be *non empty* if there is either a rule of the form , or there is a rule of the form where both and are non empty. Then the grammar is non empty if and only if the starting variable is non-empty.

We assume that the grammar in Chomsky Normal Form as in CFGhalt. We consider the following procedure for marking variables as “non empty”:

1. We start by marking all variables that are involved in a rule of the form as non empty.
2. We then continue to mark as non empty if it is involved in a rule of the form where have been marked before.

We continue this way until we cannot mark any more variables. We then declare that the grammar is empty if and only if has not been marked. To see why this is a valid algorithm, note that if a variable has been marked as “non empty” then there is some string that can be derived from . On the other hand, if has not been marked, then every sequence of derivations from will always have a variable that has not been replaced by alphabet symbols. Hence in particular is the all zero function if and only if the starting variable is not marked “non empty”.

### Uncomputability of context-free grammar equivalence (optional)

By analogy to regular expressions, one might have hoped to get an algorithm for deciding whether two given context free grammars are equivalent. Alas, no such luck. It turns out that the equivalence problem for context free grammars is *uncomputable*. This is a direct corollary of the following theorem:

### 

For every set , let be the function that on input a context-free grammar over , outputs if and only if computes the constant function. Then there is some finite such that is uncomputable.

fullnesscfgdef immediately implies that equivalence for context-free grammars is uncomputable, since computing “fullness” of a grammar over some alphabet corresponds to checking whether is equivalent to the grammar $s \Rightarrow ""|s\sigma\_0|\cdots|s\sigma\_{k-1}$. Note that fullnesscfgdef and cfgemptinessthem together imply that context-free grammars, unlike regular expressions, are *not* closed under complement. (Can you see why?) Since we can encode every element of using bits (and this finite encoding can be easily carried out within a grammar) fullnesscfgdef implies that fullness is also uncomputable for grammars over the binary alphabet.

We prove the theorem by reducing from the Halting problem. To do that we use the notion of *configurations* of NAND-TM programs, as defined in configtmdef. Recall that a *configuration* of a program is a binary string that encodes all the information about the program in the current iteration.

We define to be plus some separator characters and define to be the function that maps every string to if and only does *not* encode a sequence of configurations that correspond to a valid halting history of the computation of on the empty input.

The heart of the proof is to show that is context-free. Once we do that, we see that halts on the empty input if and only if for *every* . To show that, we will encode the list in a special way that makes it amenable to deciding via a context-free grammar. Specifically we will reverse all the odd-numbered strings.

We only sketch the proof. We will show that if we can compute then we can solve , which has been proven uncomputable in haltonzero-thm. Let be an input Turing machine for . We will use the notion of *configurations* of a Turing machine, as defined in configtmdef.

Recall that a *configuration* of Turing machine and input captures the full state of at some point of the computation. The particular details of configurations are not so important, but what you need to remember is that:

* A configuration can be encoded by a binary string .
* The *initial* configuration of on the input is some fixed string.
* A *halting configuration* will have the value a certain state (which can be easily “read off” from it) set to .
* If is a configuration at some step of the computation, we denote by as the configuration at the next step. is a string that agrees with on all but a constant number of coordinates (those encoding the position corresponding to the head position and the two adjacent ones). On those coordinates, the value of can be computed by some finite function.

We will let the alphabet . A *computation history* of on the input is a string that corresponds to a list (i.e., comes before an even numbered block, and comes before an odd numbered one) such that if is even then is the string encoding the configuration of on input at the beginning of its -th iteration, and if is odd then it is the same except the string is *reversed*. (That is, for odd , encodes the configuration of on input at the beginning of its -th iteration.) Reversing the odd-numbered blocks is a technical trick to ensure that the function we define below is context free.

We now define as follows:

We will show the following claim:

**CLAIM:** is context-free.

The claim implies the theorem. Since halts on if and only if there exists a valid computation history, is the constant one function if and only if does *not* halt on . In particular, this allows us to reduce determining whether halts on to determining whether the grammar corresponding to is full.

We now turn to the proof of the claim. We will not show all the details, but the main point if *at least one* of the following three conditions hold:

1. is not of the right format, i.e. not of the form .
2. contains a substring of the form such that
3. contains a substring of the form such that

Since context-free functions are closed under the OR operation, the claim will follow if we show that we can verify conditions 1, 2 and 3 via a context-free grammar.

For condition 1 this is very simple: checking that *is* of the correct format can be done using a regular expression. Since regular expressions are closed under negation, this means that checking that is *not* of this format can also be done by a regular expression and hence by a context-free grammar.

For conditions 2 and 3, this follows via very similar reasoning to that showing that the function such that iff is context-free, see nonpalindrome. After all, the function only modifies its input in a constant number of places. We leave filling out the details as an exercise to the reader. Since if and only if satisfies one of the conditions 1., 2. or 3., and all three conditions can be tested for via a context-free grammar, this completes the proof of the claim and hence the theorem.

## Summary of semantic properties for regular expressions and context-free grammars

To summarize, we can often trade *expressiveness* of the model for *amenability to analysis*. If we consider computational models that are *not* Turing complete, then we are sometimes able to bypass Rice’s Theorem and answer certain semantic questions about programs in such models. Here is a summary of some of what is known about semantic questions for the different models we have seen.

---  
caption: 'Computability of semantic properties'  
alignment: ''  
table-width: ''  
id: semantictable  
---  
\_Model\_, \*\*Halting\*\*, \*\*Emptiness\*\*, \*\*Equivalence\*\*  
\_Regular expressions\_, Computable, Computable,Computable  
\_Context free grammars\_, Computable, Computable, Uncomputable  
\_Turing-complete models\_, Uncomputable, Uncomputable, Uncomputable

### 

* The uncomputability of the Halting problem for general models motivates the definition of restricted computational models.
* In some restricted models we can answer *semantic* questions such as: does a given program terminate, or do two programs compute the same function?
* *Regular expressions* are a restricted model of computation that is often useful to capture tasks of string matching. We can test efficiently whether an expression matches a string, as well as answer questions such as Halting and Equivalence.
* *Context free grammars* is a stronger, yet still not Turing complete, model of computation. The halting problem for context free grammars is computable, but equivalence is not computable.

## Exercises

Suppose that are regular. For each one of the following definitions of the function , either prove that is always regular or give a counterexample for regular that would make not regular.

1. .
2. .
3. where is the reverse of : for .

One among the following two functions that map to can be computed by a regular expression, and the other one cannot. For the one that can be computed by a regular expression, write the expression that does it. For the one that cannot, prove that this cannot be done using the pumping lemma. \* if divides and otherwise.

* if and only if and otherwise.

1. Prove that the following function is not regular. For every , iff is of the form for some .
2. Prove that the following function is not regular. For every , iff for some .

Suppose that are context free. For each one of the following definitions of the function , either prove that is always context free or give a counterexample for regular that would make not context free.

1. .
2. .
3. where is the reverse of : for .

Prove that the function such that if and only if is a power of two is not context free.

Consider the following syntax of a “programming language” whose source can be written using the [ASCII](https://en.wikipedia.org/wiki/ASCII) character set:

* *Variables* are obtained by a sequence of letters, numbers and underscores, but can’t start with a number.
* A *statement* has either the form foo = bar; where foo and bar are variables, or the form IF (foo) BEGIN ... END where ... is list of one or more statements, potentially separated by newlines.

A *program* in our language is simply a sequence of statements (possibly separated by newlines or spaces).

1. Let be the function that given a string , outputs if and only if corresponds to an ASCII encoding of a valid variable identifier. Prove that is regular.
2. Let be the function that given a string , outputs if and only if is an ASCII encoding of a valid program in our language. Prove that is context free. (You do not have to specify the full formal grammar for , but you need to show that such a grammar exists.)
3. Prove that is not regular. See footnote for hint[[3]](#footnote-119)

## Bibliographical notes

The relation of regular expressions with finite automata is a beautiful topic, on which we only touch upon in this text. It is covered more extensively in [@SipserBook, @hopcroft , @kozen1997automata]. These texts also discuss topics such as *non deterministic finite automata* (NFA) and the relation between context-free grammars and pushdown automata.

Our proof of reglintimethm is closely related to the [Myhill-Nerode Theorem](https://goo.gl/mnKVMP). One direction of the Myhill-Nerode theorem theorem can be stated as saying that if is a regular expression then there is at most a finite number of strings such that for every .

As in the case of regular expressions, there are many resources available that cover context-free grammar in great detail. Chapter 2 of [@SipserBook] contains many examples of context-free grammars and their properties. There are also websites such as [Grammophone](https://mdaines.github.io/grammophone/) where you can input grammars, and see what strings they generate, as well as some of the properties that they satisfy.

The adjective “context free” is used for CFG’s because a rule of the form means that we can *always* replace with the string , no matter what is the *context* in which appears. More generally, we might want to consider cases where the replacement rules depend on the context. This gives rise to the notion of *general (aka “Type 0”) grammars* that allow rules of the form where both and are strings over . The idea is that if, for example, we wanted to enforce the condition that we only apply some rule such as when is surrounded by three zeroes on both sides, then we could do so by adding a rule of the form (and of course we can add much more general conditions). Alas, this generality comes at a cost - general grammars are Turing complete and hence their halting problem is uncomputable. That is, there is no algorithm that can determine for every general grammar and a string , whether or not the grammar generates .

The [Chomsky Hierarchy](https://en.wikipedia.org/wiki/Chomsky_hierarchy) is a hierarchy of grammars from the least restrictive (most powerful) Type 0 grammars, which correspond to *recursively enumerable* languages (see recursiveenumerableex) to the most restrictive Type 3 grammars, which correspond to regular languages. Context-free languages correspond to Type 2 grammars. Type 1 grammars are *context sensitive grammars*. These are more powerful than context-free grammars but still less powerful than Turing machines. In particular functions/languages corresponding to context-sensitive grammars are always computable, and in fact can be computed by a [linear bounded automatons](https://en.wikipedia.org/wiki/Linear_bounded_automaton) which are non-deterministic algorithms that take space. For this reason, the class of functions/languages corresponding to context-sensitive grammars is also known as the complexity class ; we discuss space-bounded complexity in spacechap). While Rice’s Theorem implies that we cannot compute any non-trivial semantic property of Type 0 grammars, the situation is more complex for other types of grammars: some semantic properties can be determined and some cannot, depending on the grammar’s place in the hierarchy.

1. We use *function notation* in this book, but other texts often use the notion of *languages*, which are sets of strings. In that notation a language is called *regular* if and only if the corresponding function is regular, where is the function that outputs on iff . [↑](#footnote-ref-38)
2. As in the case of matchingregexpdef we can also use *language* rather than *function* notation and say that a language is *context free* if the function such that iff is context free. [↑](#footnote-ref-92)
3. Try to see if you can “embed” in some way a function that looks similar to in , so you can use a similar proof. Of course for a function to be non-regular, it does not need to utilize literal parentheses symbols. [↑](#footnote-ref-119)