9
Restricted computational models

Learning Objectives:

- 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 λ calculus and even the game of life. The flip side of this is that for all these models, Rice's theorem (Theorem 8.16) 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.

Big Idea 13 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.

9.1 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 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,

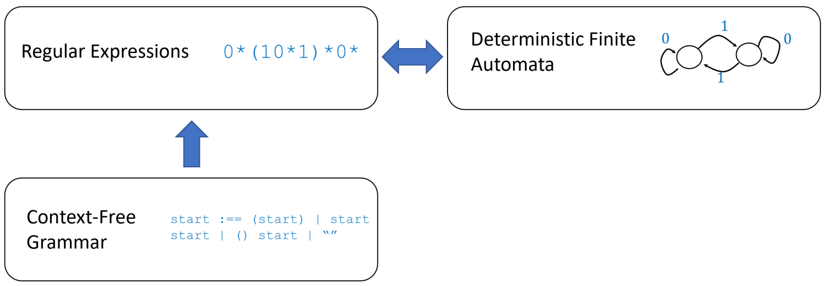


Figure 9.1: Some restricted computational models we study in this chapter. We show two equivalent models of computation: regular expressions and deterministic finite automata. We show a more powerful model: context-free grammars. We also present tools to demonstrate that some functions *can not* be computed in these models.

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 ranging from using infinite loops as a denial of service attack, to accessing the printer’s file system.

■ **Example 9.1 — The DAO Hack.** An interesting recent example of the pitfalls of Turing-completeness arose in the context of the cryptocurrency Ethereum. 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 P 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 which has a syntax similar to JavaScript. The flagship of Ethereum was an experiment known as The “Decentralized Autonomous Organization” or The DAO. 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 Chapter 8) 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 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 was created that preserved the original history.

9.2 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 $X = \{x_0, \ldots, x_k\}$ 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 $x \in X$ 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* $F : \{0,1\}^* \rightarrow \{0,1\}$, where $F(x) = 1$ corresponds to the pattern matching x. That is, the user provides a *program* P in some Turing-complete programming language such as *Python*, and the system will return all the $x \in X$ such that $P(x) = 1$. For example, one could search for all text files that contain the string `important document` or perhaps (letting P 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 P!

Because the Halting problem for Turing-complete computational models is uncomputable, we cannot in general verify that a given program P 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. If you ever used an advanced text editor, a command line shell, or have done any kind of manipulations of text files, then you have probably come across regular expressions.

A *regular expression* over some alphabet Σ is obtained by combining elements of Σ 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 $\{0,1\}$ corresponds to the set of all strings $x \in \{0,1\}^*$ where every digit is repeated at least twice:

$$(00(0^*)|11(1^*))^* \; . \tag{9.1}$$

The following regular expression over the alphabet $\{a,\ldots,z,0,\ldots,9\}$ corresponds to the set of all strings that consist of a sequence of one or more of the letters a-d followed by a sequence of one or more digits (without a leading zero):

$$(a|b|c|d)(a|b|c|d)^*(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^* \; . \tag{9.2}$$

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

Definition 9.2 — Regular expression. A *regular expression* e over an alphabet Σ is a string over $\Sigma \cup \{(,),|,*,\emptyset,""\}$ that has one of the following forms:

1. $e = \sigma$ where $\sigma \in \Sigma$

2. $e = (e'|e'')$ where e', e'' are regular expressions.

3. $e = (e')(e'')$ where e', e'' are regular expressions. (We often drop the parenthesis when there is no danger of confusion and so write this as $e'\ e''$.)

4. $e = (e')^*$ where e' is a regular expression.

   Finally we also allow the following "edge cases": $e = \emptyset$ 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 $00^*|11$ instead of $((0)(0^*))|((1)(1))$.

Every regular expression e corresponds to a function $\Phi_e : \Sigma^* \rightarrow \{0,1\}$ where $\Phi_e(x) = 1$ if x *matches* the regular expression. For example, if $e = (00|11)^*$ then $\Phi_e(110011) = 1$ but $\Phi_e(101) = 0$ (can you see why?).

P

The formal definition of Φ_e 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.

Definition 9.3 — Matching a regular expression. Let e be a regular expression over the alphabet Σ. The function $\Phi_e : \Sigma^* \rightarrow \{0,1\}$ is defined as follows:

1. If $e = \sigma$ then $\Phi_e(x) = 1$ iff $x = \sigma$.

2. If $e = (e'|e'')$ then $\Phi_e(x) = \Phi_{e'}(x) \vee \Phi_{e''}(x)$ where $\vee$ is the OR operator.

3. If $e = (e')(e'')$ then $\Phi_e(x) = 1$ iff there is some $x', x'' \in \Sigma^*$ such that x is the concatenation of x' and x'' and $\Phi_{e'}(x') = \Phi_{e''}(x'') = 1$.

4. If $e = (e')*$ then $\Phi_e(x) = 1$ iff there are is $k \in \mathbb{N}$ and some $x_0, \ldots, x_{k-1} \in \Sigma^*$ such that x is the concatenation $x_0 \cdots x_{k-1}$ and $\Phi_{e'}(x_i) = 1$ for every $i \in [k]$.

5. Finally, for the edge cases $\Phi_\emptyset$ is the constant zero function, and $\Phi_{""}$ is the function that only outputs 1 on the empty string "".

We say that a regular expression e over Σ *matches* a string $x \in \Sigma^*$ if $\Phi_e(x) = 1$. We say that a function $F : \Sigma^* \to \{0,1\}$ is *regular* if $F = \Phi_e$ for some regular expression e. [1]

P

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.

■ **Example 9.4 — A regular function.** Let $\Sigma = \{a,b,c,d,0,1,2,3,4,5,6,7,8,9\}$ and $F : \Sigma^* \to \{0,1\}$ be the function such that $F(x)$ outputs 1 iff x consists of one or more of the letters a-d followed by a sequence of one or more digits (without a leading zero). Then F is a regular function, since $F = \Phi_e$ where

$$e = (a|b|c|d)(a|b|c|d)^*(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^* \quad (9.3)$$

is the expression we saw in (9.2).

If we wanted to verify, for example, that $\Phi_e(abc12078) = 1$, we can do so by noticing that the expression $(a|b|c|d)$ matches the string a, $(a|b|c|d)^*$ matches bc, $(0|1|2|3|4|5|6|7|8|9)$ matches the string 1, and the expression $(0|1|2|3|4|5|6|7|8|9)^*$ matches the string 2078. Each one of those boils down to a simpler expression. For example, the expression $(a|b|c|d)^*$ matches the string bc because both of the one-character strings b and c are matched by the expression $a|b|c|d$.

Regular expression can be defined over any finite alphabet Σ, but as usual, we will focus our attention on the *binary case*, where $\Sigma = \{0,1\}$. 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 e over the alphabet Σ as a program that computes the function $\Phi_e : \Sigma^* \to \{0,1\}$. (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

[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 $L \subseteq \Sigma^*$ is called *regular* if and only if the corresponding function F_L is regular, where $F_L : \Sigma^* \to \{0,1\}$ is the function that outputs 1 on x iff $x \in L$.

every regular expression e, the function Φ_e is computable (and so in particular can be evaluated by an always-halting Turing machine).

Theorem 9.5 — Regular expression always halt. For every regular expression e over $\{0,1\}$, the function $\Phi_e : \{0,1\}^* \rightarrow \{0,1\}$ is computable.

That is, there is a Turing machine M such that for every $x \in \{0,1\}^*$, on input x, M halts with the output $\Phi_e(x)$.

We state Theorem 9.5 for regular expressions over the binary alphabet $\{0,1\}$, but it generalizes to any finite alphabet Σ.

Proof Idea:

The proof relies on the observation that Definition 9.3 actually specifies a recursive algorithm for *computing* Φ_e. Specifically, each one of our operations -concatenation, OR, and star- can be thought of as reducing the task of testing whether an expression e matches a string x to testing whether some sub-expressions of e match substrings of x. Since these sub-expressions are always shorter than the original expression, this yields a recursive algorithm for checking if e matches x which will eventually terminate at the base cases of the expressions that correspond to a single symbol or the empty string.

⋆

Proof of Theorem 9.5. Definition 9.3 gives a way of recursively computing Φ_e. The key observation is that in our recursive definition of regular expressions, whenever e is made up of one or two expressions e', e'' then these two regular expressions are *smaller* than e, and eventually (when they have size 1) 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 m of e (i.e., the number of symbols in the string e, also denoted as $|e|$). For $m = 1$, e is either a single alphabet symbol, "" or $\emptyset$, and so computing the function Φ_e is straightforward. In the general case, for $m = |e|$ we assume by the induction hypothesis that we have proven the theorem for all expressions of length smaller than m. Now, such an expression of length larger than one can obtained one of three cases using the OR, concatenation, or star operations. We now show that Φ_e will be computable in all these cases:

Case 1: $e = (e'|e'')$ where e', e'' are shorter regular expressions.

In this case by the inductive hypothesis we can compute $\Phi_{e'}$ and $\Phi_{e''}$ and so can compute $\Phi_e(x)$ as $\Phi_{e'}(x) \vee \Phi_{e''}(x)$ (where $\vee$ is the OR operator).

Case 2: $e = (e')(e'')$ where e', e'' are regular expressions.

In this case by the inductive hypothesis we can compute $\Phi_{e'}$ and $\Phi_{e''}$ and so can compute $\Phi_e(x)$ as

$$\bigvee_{i=0}^{|x|-1} (\Phi_{e'}(x_0 \cdots x_{i-1}) \wedge \Phi_{e''}(x_i \cdots x_{|x|-1})) \tag{9.4}$$

where $\wedge$ is the AND operator and for $i < j$, $x_j \cdots x_i$ refers to the empty string.

Case 3: $e = (e')*$ where e' is a regular expression.

In this case by the inductive hypothesis we can compute $\Phi_{e'}$ and so we can compute $\Phi_e(x)$ by enumerating over all k from 1 to $|x|$, and all ways to write x as the concatenation of k nonempty strings $x_0 \cdots x_{k-1}$ (we can do so by enumerating over all possible $k-1$ positions in which one string stops and the other begins). If for one of those partitions, $\Phi_{e'}(x_0) = \cdots = \Phi_{e'}(x_{k-1}) = 1$ then we output 1. Otherwise we output 0. We can restrict attention to partitions of x as $x = x_0 \cdots x_{k-1}$ where all the x_i's are nonempty since if some of the x_i'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.

■

9.3 DETERMINISTIC FINITE AUTOMATA, AND EFFICIENT MATCHING OF REGULAR EXPRESSIONS (OPTIONAL)

The proof of Theorem 9.5 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., $O(n)$) 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 $O(n)$ running time as is used in introduction to programming courses and whiteboard coding interviews. We will see a formal definition of time complexity in Chapter 12.

Theorem 9.6 — Matching regular expressions in linear time. Let e be a regular expression. Then there is an $O(n)$ time algorithm that computes Φ_e.

The implicit constant in the $O(n)$ term of Theorem 9.6 depends on the expression e. Thus, another way to state Theorem 9.6 is that for every expression e, there is some constant c and an algorithm A that computes Φ_e on n-bit inputs using at most $c \cdot n$ steps. This makes sense, since in practice we often want to compute $\Phi_e(x)$ for a small regular expression e and a large document x. Theorem 9.6 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.

Proof Idea:

The idea is to define a more efficient recursive algorithm, that determines whether e matches a string $x \in \{0,1\}^n$ by reducing this task to determining whether a related expression e' matches $x_0, \dots, x_{n-1}$. This will result in an expression for the running time of the form $T(n) = T(n-1) + O(1)$ which solves to $T(n) = O(n)$.

⋆

Proof of Theorem 9.6. The central definition for this proof is the notion of a *restriction* of a regular expression. The idea is is that for every regular expression e and symbol σ in its alphabet, it is possible to define a regular expresion $e[\sigma]$ such that $e[\sigma]$ matches a string x if and only if e matches the string $x\sigma$. For example, if e is the regular expression $01|(01)*(01)$ (i.e., one or more occurrences of 01) then $e[1]$ is equal to $0|(01)*0$ and $e[0]$ will be $\emptyset$. (Can you see why?)

For simplicity, from now on we fix our attention to the case that the alphabet Σ is $\{0,1\}$. Given a regular expression e and $\sigma \in \{0,1\}$, we can compute $e[\sigma]$ recursively as follows:

1. If e consists of a single symbol (i.e. $e = \tau$ for $\tau \in \{0,1\}$) then $e[\sigma] =$ "" if $\tau = \sigma$ and $e[\sigma] = \emptyset$ otherwise.
2. If $e = e'|e''$ then $e[\sigma] = e'[\sigma]|e''[\sigma]$.
3. If $e = e'\ e''$ then $e[\sigma] = e'\ e''[\sigma]$ if e'' can not match the empty string. Otherwise, $e[\sigma] = e'\ e''[\sigma]\ |\ e'[\sigma]$
4. If $e = (e')^*$ then $e[\sigma] = (e')^*(e'[\sigma])$.
5. If $e =$ "" or $e = \emptyset$ then $e[\sigma] = \emptyset$.

By checking all these cases, one can verify that it is indeed the case that for every regular expression e, $\sigma \in \{0,1\}$ and $x \in \{0,1\}^*$, $e[\sigma]$ matches x if and only if e matches $x\sigma$. We let $C(\ell)$ denote the time to compute $e[\sigma]$ for regular expressions of length at most ℓ. The value $C(\ell)$ can be shown to be polynomial in ℓ, though this is not important for this theorem, since we only care about the dependence of the time to compute $\Phi_e(x)$ on the length of x and not about the dependence of this time on the length of e.

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

Algorithm 9.7 — Regular expression matching in linear time.
Input: Regular expression e over $\{0,1\}$ and $x \in \{0,1\}^n$ for $n \in \mathbb{N}$.
Goal: Compute $\Phi_e(x)$
Operation:

1. If $x =$ "" then return 1 if and only if $\Phi_e("") = 1$. (This can be either computed directly or using the algorithm of Theorem 9.5 in time which is a constant depending only on the regular expression e.)
2. Otherwise, compute $\Phi_{e[x_{n-1}]}(x_0 \cdots x_{n-1})$ recursively and output the result.

By the definition of a restriction, for every $\sigma \in \{0,1\}$ and $x' \in \{0,1\}^*$, the expression e matches $x'\sigma$ if and only if $e[\sigma]$ matches x'. Hence for every e and $x \in \{0,1\}^n$, $\Phi_{e[x_{n-1}]}(x_0 \cdots x_{n-2}) = \Phi_e(x)$ and Algorithm 9.7 does return the correct answer. The only remaining task is to analyze its *running time*.

Algorithm 9.7 is a recursive algorithm that on input an expression e and a string $x \in \{0,1\}^n$, does some constant time computation and then calls itself on input some expression e' and a string x of length $n-1$. It will terminate after n steps when it reaches a string of length 0. So, to calculate the running time of Algorithm 9.7 we need to analyze the cost of each step.

Specifically, the running time $T(e,n)$ that it takes for Algorithm 9.7 to compute Φ_e for inputs of length n satisfies the recursive equation:

$$T(e,n) = \max\{T(e[0],n-1), T(e[1],n-1)\} + C(|e|) \tag{9.5}$$

where $C(\ell)$, as before, denotes the time to compute $e[\sigma]$ for expressions e of length at most ℓ. (In the base case $n = 0$, $T(e,0)$ is equal to some constant depending only on e.)

To get some intuition for the expression Eq. (9.5), let us open up the recursion for one level, writing $T(e,n)$ as

$$\begin{aligned} T(e,n) = \max\{&T(e[0][0],n-2) + C(|e[0]|), \\ &T(e[0][1],n-2) + C(|e[0]|), \\ &T(e[1][0],n-2) + C(|e[1]|), \\ &T(e[1][1],n-2) + C(|e[1]|)\} + C(|e|) \; . \end{aligned} \tag{9.6}$$

Continuing this way, we can see that $T(e,n) \leq n \cdot C(\ell) + O(1)$ where ℓ is the largest length of any expression e' that we encounter along the way. Therefore, the following claim suffices to show that Algorithm 9.7 runs in linear time:

Claim: Let e be a regular expression over $\{0,1\}$, then there is some constant c such that for every string $\alpha \in \{0,1\}^*$, if we restrict e to α_0, and then to α_1 and so on and so forth, the resulting expression has length at most c.

Proof of claim: For a regular expression e over $\{0,1\}$ and $\alpha \in \{0,1\}^m$, we denote by $e[\alpha]$ the expression $e[\alpha_0][\alpha_1]\cdots[\alpha_{m-1}]$ obtained by restricting e to α_0 and then to α_1 and so on. We let $S(e) = \{e[\alpha]|\alpha \in \{0,1\}^*\}$. We will prove the claim by showing that for every e, the set $S(e)$ is finite, and hence so is the number $c(e)$ which is the maximum length of e' for $e' \in S(e)$.

We prove this by induction on the structure of e. If e is a symbol, the empty string, or the empty set, then this is straightforward to show as the most expressions $S(e)$ can contain are the expression itself, "", and $\emptyset$. Otherwise we split to the two cases **(i)** $e = e'^*$ and **(ii)** $e = e'e''$, where e', e'' are smaller expressions (and hence by the induction hypothesis $S(e')$ and $S(e'')$ are finite). In the case **(i)**, if $e = (e')^*$ then $e[\alpha]$ is either equal to $(e')^*e'[\alpha]$ or it is simply the empty set if $e'[\alpha] = \emptyset$. Since $e'[\alpha]$ is in the set $S(e')$, the number of distinct expressions in $S(e)$ is at most $|S(e')| + 1$. In the case **(ii)**, if $e = e'e''$ then all the restrictions of e to strings α will either have the form $e'e''[\alpha]$ or the form $e'e''[\alpha]|e'[\alpha']$ where α' is some string such that $\alpha = \alpha'\alpha''$ and $e[\alpha'']$ matches the empty string. Since $e''[\alpha] \in S(e'')$ and $e'[\alpha'] \in S(e')$, the number of the possible distinct expressions of the form $e[\alpha]$ is at most $|S(e'')| + |S(e'')| \cdot |S(e')|$. This completes the proof of the claim.

The bottom line is that while running Algorithm 9.7 on a regular expression e, all the expressions we ever encounter are in the finite set $S(e)$, no matter how large the input x is, and so the running time of Algorithm 9.7 satisfies the equation $T(n) = T(n-1) + C'$ for some constant C' depending on e. This solves to $O(n)$ where the implicit constant in the Oh notation can (and will) depend on e but crucially, not on the length of the input x.

■

9.3.1 Matching regular expressions using constant memory

Theorem 9.6 is already quite impressive, but we can do even better. Specifically, no matter how long the string x is, we can compute $\Phi_e(x)$ by maintaining only a constant amount of memory and moreover making a *single pass* over x. That is, the algorithm will scan the input x once from start to finish, and then determine whether or not x is matched by the expression e. 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) (see Section 9.3.2). There is a beautiful theory on the properties of DFA's and their connections with regular expressions. In particular, as we'll see in Theorem 9.12, a function is regular *if and only if* it can be computed by a DFA. We start with showing the "only if" direction:

> **Theorem 9.8 — DFA for regular expression matching.** Let e be a regular expression. Then there is an algorithm that on input $x \in \{0,1\}^*$ computes $\Phi_e(x)$ while making a single pass over x and maintaining a constant amount of memory.

Proof Idea:

The idea is to replace the recursive algorithm of Algorithm 9.7 with a dynamic program, using the technique of 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.

⋆

Proof of Theorem 9.8. We will replace the recursive Algorithm 9.7 with the following iterative algorithm:

Algorithm 9.9 — Constant memory regular expression matching.

```
Input: Regular expression $e$ over $\{0,1\}$, string $x \in \{0,1\}^n$.
Goals: Compute $\Phi_e(x)$.
Operation:

1. Let $S = S(e)$ be the set $\{e[\alpha] | \alpha \in \{0,1\}^*\}$ as defined
   in the proof of Theorem 9.6. Note that $S$ is finite and by
   definition, for every $e' \in S$ and $\sigma \in \{0,1\}$, $e'[\sigma]$ is in $S$ as
   well.
2. Define a Boolean variable $v_{e'}$ for every $e' \in S$. Initially
   we set $v_{e'} = 1$ if and only if $e'$ matches the empty string.
3. For $i = 0, \ldots, n-1$ do the following:
   a. Copy the variables $\{v_{e'}\}$ to temporary variables: For every
      $e' \in S$, we set $temp_{e'} = v_{e'}$.
   b. Update the variables $\{v_{e'}\}$ based on the i-th bit of $x$: Let
      $\sigma = x_i$ and set $v_{e'} = temp_{e'[\sigma]}$ for every $e' \in S$.
4. Output $v_e$.
```

Algorithm 9.9 maintains the invariant that at the end of step i, for every $e' \in S$, the variable $v_{e'}$ is equal if and only if e' matches the string $x_0 \cdots x_{i-1}$. In particular, at the very end, v_e is equal to 1 if and only if e matches the full string $x_0 \cdots x_{n-1}$. Algorithm 9.9 only maintains a constant number of variables (as S is finite), and that it proceeds in one linear scan over the input, and so this proves the theorem.

■

9.3.2 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 C states, for some constant C. The machine starts in some initial state, and then reads its input $x \in \{0,1\}^*$ one bit at a time. Whenever the machine reads a bit $\sigma \in \{0,1\}$, it transitions into a new state based on σ and its prior state. The output of the machine is based the final state. Every constant-memory one-pass algorithm corresponds to such a machine. If an algorithm uses c bits of memory, then the contents of its memory are a string of length c. Since there are 2^c such strings, at any point in the execution, such an algorithm can be in one of 2^c states.

■ **Example 9.10 — DFA for XOR.** Here is a DFA for computing the function $XOR : \{0,1\}^* \rightarrow \{0,1\}$ that maps x to $\sum_{i \in [|x|]} x_i \mod 2$.

We will have two states: 0 and 1. The set of accepting states is $\{1\}$, and if we are in a state $v \in \{0,1\}$ and read the bit σ, we will transition to the state v if $\sigma = 0$ and to the state $1 - v$ if $\sigma = 1$. In other words, we transition to the state $v \oplus \sigma$. Hence we can think of this algorithm's execution on input $x \in \{0,1\}^n$ as follows:

- Let v_t be the sate of the automaton at step t. We initialize $v_t = 0$.
- For every $i \in [n]$, let $v_i = v_{i+1} \oplus x_i$.
- Output v_n.

You can verify that the output of this algorithm is $x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1} = XOR(x)$. We can also describe this DFA graphically, see Fig. 9.2.

The formal definition of a DFA is the following:

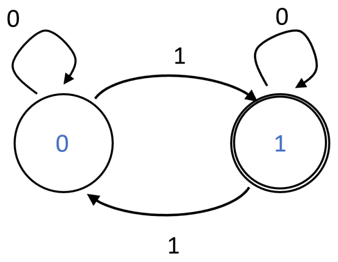


Figure 9.2: A deterministic finite automaton that computes the XOR function. It has two states 0 and 1, and when it observes σ it transitions from v to $v \oplus \sigma$.

Definition 9.11 — Deterministic Finite Automaton. A deterministic finite automaton (DFA) with C states over $\{0,1\}$ is a pair $(T, \mathcal{S})$ with $T : [C] \times \{0,1\} \rightarrow [C]$ and $\mathcal{S} \subseteq [C]$. The function T is known as the *transition function* of the DFA and the set $\mathcal{S}$ is known as the set of *accepting states*.

We say that $(T, \mathcal{S})$ *computes* a function $F : \{0,1\}^* \rightarrow \{0,1\}$ if for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$, if we define $v_0 = 0$ and $v_{i+1} = T(v_i, x_i)$ for every $i \in [n]$, then

$$v_n \in \mathcal{S} \Leftrightarrow F(x) = 1 \tag{9.7}$$

P

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 [Sip97] 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 and here).

Sipser defines a DFAs as a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is the alphabet, δ is the transition function, q_0 is the initial state, and F is the set of accepting states. In this book the set of states is always of the form $Q = \{0, \ldots, C-1\}$ and the initial state is always $q_0 = 0$, but this makes no difference to the computational power of these models. Also, we restrict our attention to the case that the alphabet Σ is equal to $\{0, 1\}$.

The following theorem is the central result of automata theory:

Theorem 9.12 — DFA and regular expression equivalency. Let $F : \{0,1\}^* \to \{0,1\}$. Then F is regular if and only if there exists a DFA $(T, \mathcal{S})$ that computes F.

Proof Idea:

One direction follows from Theorem 9.8, which shows that for every regular expression e, the function Φ_e can be computed by a DFA (see for example Fig. 9.3). For the other direction, we show that given a DFA $(T, \mathcal{S})$ for every $v, w \in [C]$ we can find a regular expression that would match $x \in \{0,1\}^*$ if an only if the DFA starting in state v, will end up in state w after reading x.

⋆

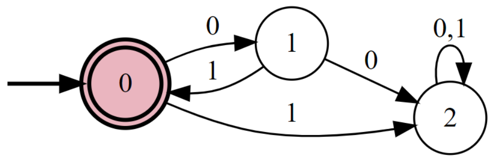


Figure 9.3: A deterministic finite automaton that computes the function $\Phi_{(01)^*}$.

Proof of Theorem 9.12. Since Theorem 9.8 proves the "only if" direction, we only need to show the "if" direction. Let $A = (T, \mathcal{S})$ be a DFA with C states that computes the function F. We need to show that F is regular.

For every $v, w \in [C]$, we let $F_{v,w} : \{0,1\}^* \to \{0,1\}$ be the function that maps $x \in \{0,1\}^*$ to 1 if and only if the DFA A, starting at the state v, will reach the state w if it reads the input x. We will prove that $F_{v,w}$ is regular for every v, w. This will prove the theorem, since by Definition 9.11, $F(x)$ is equal to the OR of $F_{0,w}(x)$ for every $w \in \mathcal{S}$. Hence if we have a regular expression for every function of the form $F_{v,w}$ then (using the $|$ operation) we can obtain a regular expression for F as well.

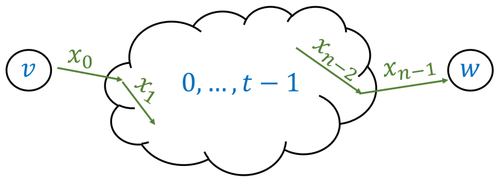


Figure 9.4: Given a DFA of C states, for every $v, w \in [C]$ and number $t \in \{0, \ldots, C\}$ we define the function $F^t_{v,w} : \{0,1\}^* \to \{0,1\}$ to output one on input $x \in \{0,1\}^*$ if and only if when the DFA is initialized in the state v and is given the input x, it will teach the state w while going only through the intermediate states $\{0, \ldots, t-1\}$.

To give regular expressions for the functions $F_{v,w}$, we start by defining the following functions $F^t_{v,w}$: for every $v, w \in [C]$ and $0 \leq t \leq C$, $F^t_{v,w}(x) = 1$ if and only if starting from v and observing x, the automata reaches w *with all intermediate states being in the set* $[t] = \{0, \ldots, t-1\}$ (see Fig. 9.4). That is, while v, w themselves might be outside $[t]$, $F^t_{v,w}(x) = 1$ if and only if throughout the execution of the automaton on the input x (when initiated at v) it never enters any of the states outside $[t]$ and still ends up at w. If $t = 0$ then $[t]$ is the empty set, and hence $F^0_{v,w}(x) = 1$ if and only if the automaton reaches w from v directly on x, without any intermediate state. If $t = C$ then all states are in $[t]$, and hence $F^t_{v,w} = F_{v,w}$.

We will prove the theorem by induction on t, showing that $F^t_{v,w}$ is regular for every v, w and t. For the **base case** of $t = 0$, $F^0_{v,w}$ is regular for every v, w since it can be described one of the expressions "", $\emptyset$, 0, 1 or $0|1$. Specifically, if $v = w$ then $F^0_{v,w}(x) = 1$ if and only if x is the empty string. If $v \neq w$ then $F^0_{v,w}(x) = 1$ if and only if x consists of a single symbol $\sigma \in \{0,1\}$ and $T(v,\sigma) = w$. Therefore in this case $F^0_{v,w}$ corresponds to one of the four regular expressions $0|1$, 0, 1 or $\emptyset$, depending on whether A transitions to w from v when it reads either 0 or 1, 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 $v', w' \in [C]$, we have a regular expression $R^t_{v,w}$ that computes $F^t_{v',w'}$. We need to prove that $F^{t+1}_{v,w}$ is regular for every v, w. If the automaton arrives from v to w using the intermediate states $[t+1]$, then it visits the t-th state zero or more times. If the path labeled by x causes the automaton to get from v to w without visiting the t-th state at all, then x is matched by the regular expression $R^t_{v,w}$. If the path labeled by x causes the automaton to get from v to w while visiting the t-th state $k > 0$ times then we can think of this path as:

- First travel from v to t using only intermediate states in $[t-1]$.
- Then go from t back to itself $k-1$ using only intermediate states in $[t-1]$
- Then go from t to w using only intermediate states in $[t-1]$.

Therefore in this case the string x is matched by the regular expression $R^t_{v,t}(R^t_{t,t})^* R^t_{t,w}$. (See also Fig. 9.5.)

Therefore we can compute $F^{t+1}_{v,w}$ using the regular expression

$$R^t_{v,w} \mid R^t_{v,t}(R^t_{t,t})^* R^t_{t,w} \; . \tag{9.8}$$

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

■

9.3.3 Regular functions are closed under complement

Here is an important corollary of Theorem 9.12:

Lemma 9.13 — Regular expressions closed under complement. If $F : \{0,1\}^* \to \{0,1\}$ is regular then so is the function $\overline{F}$, where $\overline{F}(x) = 1 - F(x)$ for every $x \in \{0,1\}^*$.

Proof. If F is regular then by Theorem 9.6 it can be computed by a constant-space one-pass algorithm A. But then the algorithm $\overline{A}$ which does the same computation and outputs the negation of the output

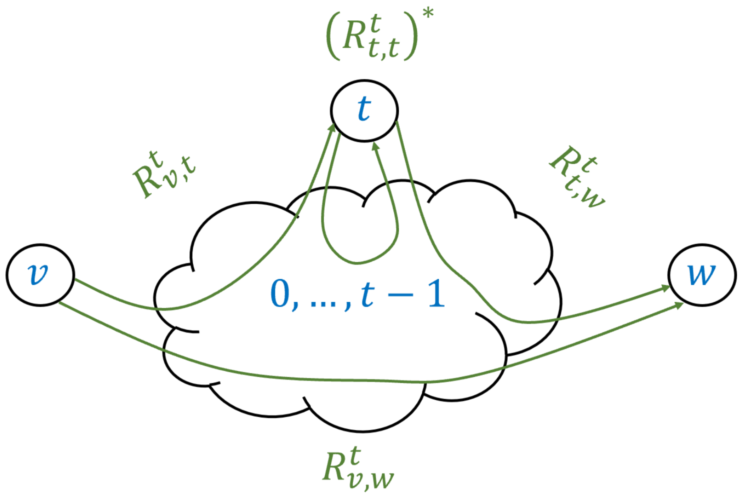


Figure 9.5: If we have regular expressions $R^t_{v',w'}$ corresponding to $F^t_{v',w'}$ for every $v', w' \in [C]$, we can obtain a regular expression $R^{t+1}_{v,w}$ corresponding to $F^{t+1}_{v,w}$. The key observation is that a path from v to w using $\{0,\ldots,t\}$ either does not touch t at all, in which case it is captured by the expression $R^t_{v,w}$, or it goes from v to t, comes back to t zero or more times, and then goes from t to w, in which case it is captured by the expression $R^t_{v,t}(R^t_{t,t})^*R^t_{t,w}$.

of A also utilizes constant space and one pass and computes $\overline{F}$. By Theorem 9.12 this implies that $\overline{F}$ is regular as well.

■

9.4 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:

Lemma 9.14 — Matching parenthesis. Let $\Sigma = \{\langle, \rangle\}$ and $MATCHPAREN : \Sigma^* \rightarrow \{0,1\}$ be the function that given a string of parenthesis, outputs 1 if and only if every opening parenthesis is matched by a corresponding closed one. Then there is no regular expression over Σ that computes $MATCHPAREN$.

Lemma 9.14 is a consequence of the following result, which is known as the *pumping lemma*:

Theorem 9.15 — Pumping Lemma. Let e be a regular expression over some alphabet Σ. Then there is some number n_0 such that for every $w \in \{0,1\}^*$ with $|w| > n_0$ and $\Phi_e(w) = 1$, we can write $w = xyz$ for strings $x, y, z \in \Sigma^*$ satisfying the following conditions:

1. $|y| \geq 1$.
2. $|xy| \leq n_0$.
3. $\Phi_e(xy^kz) = 1$ for every $k \in \mathbb{N}$.

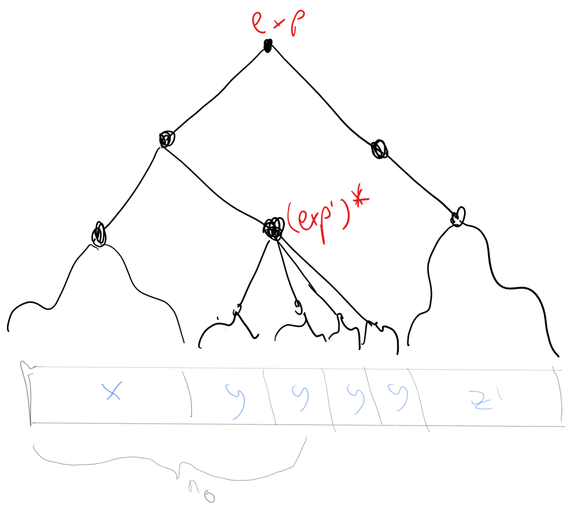


Figure 9.6: To prove the "pumping lemma" we look at a word w that is much larger than the regular expression e that matches it. In such a case, part of w must be matched by some sub-expression of the form $(e')^*$, since this is the only operator that allows matching words longer than the expression. If we look at the "leftmost" such sub-expression and define y^k to be the string that is matched by it, we obtain the partition needed for the pumping lemma.

Proof Idea:

The idea behind the proof the following. Let n_0 be twice the number of symbols that are used in the expression e, then the only way that there is some w with $|w| > n_0$ and $\Phi_e(w) = 1$ is that e contains the $*$ (i.e. star) operator and that there is a nonempty substring y of w that was matched by $(e')^*$ for some sub-expression e' of e. We can now repeat y any number of times and still get a matching string. See also Fig. 9.6.

★

P

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"*.

Proof of Theorem 9.15. 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 n length expression, $n_0 = 2n$ satisfies the conditions of the lemma. The **base case** is when the expression is a single symbol $\sigma \in \Sigma$ or that the expression is $\emptyset$ or "". In all these cases the conditions of the lemma are satisfied simply because there $n_0 = 2$ and there is no string x of length larger than n_0 that is matched by the expression.

We now prove the **inductive step**. Let e be a regular expression with $n > 1$ symbols. We set $n_0 = 2n$ and let $w \in \Sigma^*$ be a string satisfying $|w| > n_0$. Since e has more than one symbol, it has one of the the forms **(a)** $e'|e''$, **(b)**, $(e')(e'')$, or **(c)** $(e')^*$ where in all these cases the subexpressions e' and e'' have fewer symbols than e and hence satisfy the induction hypothesis.

In the case **(a)**, every string w matched by e must be matched by either e' or e''. If e' matches w then, since $|w| > 2|e'|$, by the induction hypothesis there exist x, y, z with $|y| \geq 1$ and $|xy| \leq 2|e'| < n_0$ such that e' (and therefore also $e = e'|e''$) matches xy^kz for every k. The same arguments works in the case that e'' matches w.

In the case **(b)**, if w is matched by $(e')(e'')$ then we can write $w = w'w''$ where e' matches w' and e'' matches w''. We split to subcases. If $|w'| > 2|e'|$ then by the induction hypothesis there exist x, y, z' with $|y| \leq 1$, $|xy| \leq 2|e'| < n_0$ such that $w' = xyz'$ and e' matches xy^kz' for every $k \in \mathbb{N}$. This completes the proof since if we set $z = z'w''$ then we see that $w = w'w'' = xyz$ and $e = (e')(e'')$ matches xy^kz for every $k \in \mathbb{N}$. Otherwise, if $|w'| \leq 2|e'|$ then since $|w| = |w'| + |w''| > n_0 = 2(|e'| + |e''|)$, it must be that $|w''| > 2|e''|$. Hence by the induction hypothesis there exist x', y, z such that $|y| \geq 1$, $|x'y| \leq 2|e''|$ and e'' matches $x'y^kz$ for every $k \in \mathbb{N}$. But now if we set $x = w'x'$ we see that $|xy| \leq |w'| + |x'y| \leq 2|e'| + 2|e''| = n_0$ and on the other hand the expression $e = (e')(e'')$ matches $xy^kz = w'x'y^kz$ for every $k \in \mathbb{N}$.

In case **(c)**, if w is matched $(e')^*$ then $w = w_0 \cdots w_t$ where for every $i \in [t]$, w_i is a nonempty string matched by e'. If $|w_0| > 2|e'|$ then we can use the same approach as in the concatenation case above. Otherwise, we simply note that if x is the empty string, $y = w_0$, and $z = w_1 \cdots w_t$ then $|xy| \leq n_0$ and xy^kz is matched by $(e')^*$ for every $k \in \mathbb{N}$.

■

Remark 9.16 — Recursive definitions and inductive proofs. 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 P, then it is natural to use an inductive steps that says that if o', o'', o''' etc have property P then so is an object o that is obtained by composing them.

Using the pumping lemma, we can easily prove Lemma 9.14 (i.e., the non-regularity of the "matching parenthesis" function):

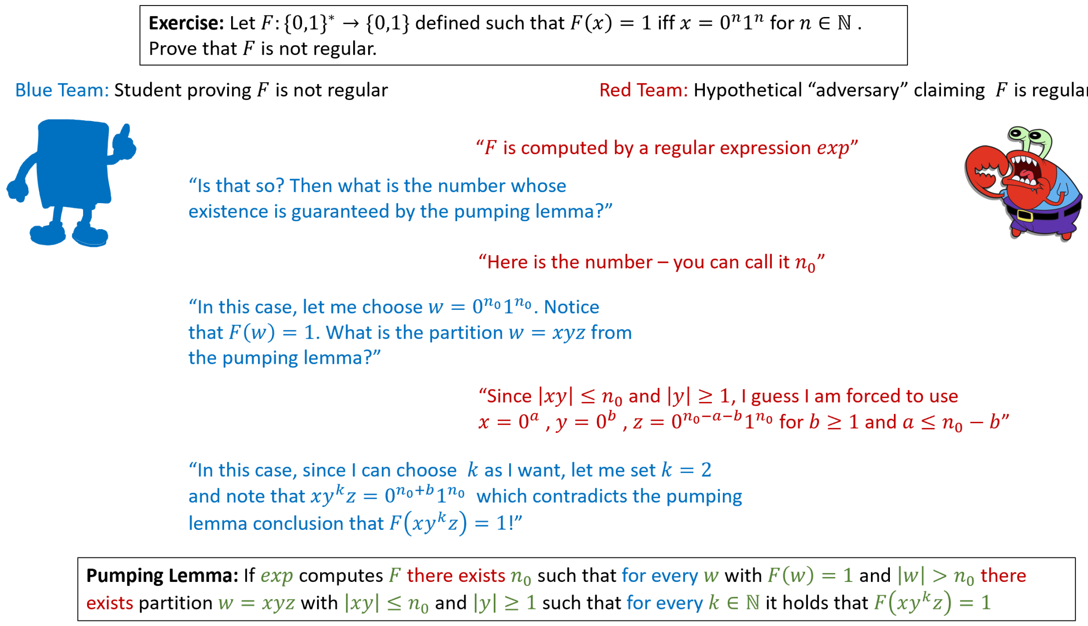
Proof of Lemma 9.14. Suppose, towards the sake of contradiction, that there is an expression e such that $\Phi_e = MATCHPAREN$. Let n_0 be the number from Lemma 9.14 and let $w = \langle^{n_0}\rangle^{n_0}$ (i.e., n_0 left parenthesis followed by n_0 right parenthesis). Then we see that if we write $w = xyz$ as in Lemma 9.14, the condition $|xy| \leq n_0$ implies that y consists solely of left parenthesis. Hence the string xy^2z will contain more left parenthesis than right parenthesis. Hence $MATCHPAREN(xy^2z) = 0$ but by the pumping lemma $\Phi_e(xy^2z) = 1$, contradicting our assumption that $\Phi_e = MATCHPAREN$.

■

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 Theorem 9.15. In particular, the number n_0 in the statement of Theorem 9.15 depends on the regular expression (in the proof we chose n_0 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 e computing some function F, we need to be able to choose an appropriate input $w \in \{0,1\}^*$ that can be arbitrarily large and satisfies $F(w) = 1$. This makes sense if you think about the intuition behind the pumping lemma: we need w to be large enough as to force the use of the star operator.

Solved Exercise 9.1 — Palindromes is not regular. Prove that the following function over the alphabet $\{0,1,;\}$ is not regular: $PAL(w) = 1$ if and only if $w = u;u^R$ where $u \in \{0,1\}^*$ and u^R denotes u "reversed": the string $u_{|u|-1} \cdots u_0$. (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.)

■



Exercise: Let $F:\{0,1\}^* \to \{0,1\}$ defined such that $F(x) = 1$ iff $x = 0^n 1^n$ for $n \in \mathbb{N}$. Prove that F is not regular.

Blue Team: Student proving F is not regular

Red Team: Hypothetical "adversary" claiming F is regular

"F is computed by a regular expression exp"

"Is that so? Then what is the number whose existence is guaranteed by the pumping lemma?"

"Here is the number – you can call it n_0"

"In this case, let me choose $w = 0^{n_0}1^{n_0}$. Notice that $F(w) = 1$. What is the partition $w = xyz$ from the pumping lemma?"

"Since $|xy| \le n_0$ and $|y| \ge 1$, I guess I am forced to use $x = 0^a$, $y = 0^b$, $z = 0^{n_0-a-b}1^{n_0}$ for $b \ge 1$ and $a \le n_0 - b$"

"In this case, since I can choose k as I want, let me set $k = 2$ and note that $xy^kz = 0^{n_0+b}1^{n_0}$ which contradicts the pumping lemma conclusion that $F(xy^kz) = 1$!"

Pumping Lemma: If exp computes F there exists n_0 such that for every w with $F(w) = 1$ and $|w| > n_0$ there exists partition $w = xyz$ with $|xy| \le n_0$ and $|y| \ge 1$ such that for every $k \in \mathbb{N}$ it holds that $F(xy^kz) = 1$

Figure 9.7: A cartoon of a proof using the pumping lemma that a function F is not regular. The pumping lemma states that if F is regular then *there exists* a number n_0 such that *for every* large enough w with $F(w) = 1$, *there exists* a partition of w to $w = xyz$ satisfying certain conditions such that *for every* $k \in \mathbb{N}$, $F(xy^kz) = 1$. 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 k that would result in $F(xy^ky) = 0$, hence violating the conclusion of the pumping lemma.

Solution:

We use the pumping lemma. Suppose towards the sake of contradiction that there is a regular expression e computing PAL, and let n_0 be the number obtained by the pumping lemma (Theorem 9.15). Consider the string $w = 0^{n_0};0^{n_0}$. Since the reverse of the all zero string is the all zero string, $PAL(w) = 1$. Now, by the pumping lemma, if PAL is computed by e, then we can write $w = xyz$ such that $|xy| \leq n_0$, $|y| \geq 1$ and $PAL(xy^kz) = 1$ for every $k \in \mathbb{N}$. In particular, it must hold that $PAL(xz) = 1$, but this is a contradiction, since $xz = 0^{n_0-|y|};0^{n_0}$ 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 Fig. 9.7 which illustrates a cartoon of the proof of the non-regularity of the function $F : \{0,1\}^* \rightarrow \{0,1\}$ which is defined as $F(x) = 1$ iff $x = 0^n1^n$ for some $n \in \mathbb{N}$ (i.e., x consists of a string of consecutive zeroes, followed by a string of consecutive ones of the same length).

9.5 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. 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 (Theorem 8.16). For example, we can tell whether two regular expressions are *equivalent*, as well as whether a regular expression computes the constant zero function.

Theorem 9.17 — Emptiness of regular languages is computable. There is an algorithm that given a regular expression e, outputs 1 if and only if Φ_e is the constant zero function.

Proof Idea:

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

⋆

Proof of Theorem 9.17. Define a regular expression to be "empty" if it computes the constant zero function. Given a regular expression e, we can determine if e is empty using the following rules:

- If e has the form σ or "" then it is not empty.
- If e is not empty then $e|e'$ is not empty for every e'.
- If e is not empty then e^* is not empty.
- If e and e' are both not empty then $e\ e'$ is not empty.
- $\emptyset$ is empty.

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

■

Theorem 9.18 — Equivalence of regular expressions is computable. Let $REGEQ : \{0,1\}^* \rightarrow \{0,1\}$ be the function that on input (a string representing) a pair of regular expressions e, e', $REGEQ(e,e') = 1$ if and only if $\Phi_e = \Phi_{e'}$. Then $REGEQ$ is computable.

Proof Idea:

The idea is to show that given a pair of regular expression e and e' we can find an expression e'' such that $\Phi_{e''}(x) = 1$ if and only if $\Phi_e(x) \neq \Phi_(e'')(x)$. Therefore $\Phi_{e''}$ is the constant zero function if and only if e and e' are equivalent, and thus we can test for emptiness of e'' to determine equivalence of e and e'.

⋆

Proof of Theorem 9.18. We will prove Theorem 9.18 from Theorem 9.17. (The two theorems are in fact equivalent: it is easy to prove Theorem 9.17 from Theorem 9.18, since checking for emptiness is the same as checking equivalence with the expression $\emptyset$.) Given two regular expressions e and e', we will compute an expression e'' such that $\Phi_{e''}(x) = 1$ if and only if $\Phi_e(x) \neq \Phi_{e'}(x)$. One can see that e is equivalent to e' if and only if e'' is empty.

We start with the observation that for every bits $a, b \in \{0,1\}$, $a \neq b$ if and only if

$$(a \wedge \overline{b}) \;\vee\; (\overline{a} \wedge b)\;. \tag{9.9}$$

Hence we need to construct e'' such that for every x,

$$\Phi_{e''}(x) = (\Phi_e(x) \wedge \overline{\Phi_{e'}(x)}) \;\vee\; (\overline{\Phi_e(x)} \wedge \Phi_{e'}(x))\;. \tag{9.10}$$

To construct the expression e'', we will show how given any pair of expressions e and e', we can construct expressions $e \wedge e'$ and $\overline{e}$ that compute the functions $\Phi_e \wedge \Phi_{e'}$ and $\overline{\Phi_e}$ respectively. (Computing the expression for $e \vee e'$ is straightforward using the $|$ operation of regular expressions.)

Specifically, by Lemma 9.13, regular functions are closed under negation, which means that for every regular expression e, there is an expression $\overline{e}$ such that $\Phi_{\overline{e}}(x) = 1 - \Phi_e(x)$ for every $x \in \{0,1\}^*$. Now, for every two expression e and e', the expression

$$e \wedge e' = \overline{(\overline{e}|\overline{e'})} \tag{9.11}$$

computes the AND of the two expressions. Given these two transformations, we see that for every regular expressions e and e' we can find a regular expression e'' satisfying (9.10) such that e'' is empty if and only if e and e' are equivalent.

■

9.6 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 determines 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 $ARITH : \Sigma^* \rightarrow \{0,1\}$ that takes as input a string x over the alphabet $\Sigma = \{(,),+,-,\times,\div,0,1,2,3,4,5,6,7,8,9\}$ and returns 1 if and only if the string x represents a valid arithmetic expression. Intuitively, we build expressions by applying an operation such as $+$,$-$,$\times$ or $\div$ 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 $0,1,2,3,4,5,6,7,8,9$.
- 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 $+,-,\times,\div$
- 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 $exp1$ and $exp2$ are valid expressions, then $exp1 \times exp2$ is also a valid expression"; we can also write this rule using the shorthand $expression \Rightarrow expression \times expression$. As in the above example, the rules of a context-free grammar are often *recursive*: the rule $expression \Rightarrow expression \times expression$ defines valid expressions in terms of itself. We now formally define context-free grammars:

Definition 9.19 — Context Free Grammar. Let Σ be some finite set. A *context free grammar (CFG) over* Σ is a triple (V,R,s) such that:

- V, known as the *variables*, is a set disjoint from Σ.
- $v \in V$ is known as the *initial variable*.
- R is a set of *rules*. Each rule is a pair (v,z) with $v \in V$ and $z \in (\Sigma \cup V)^*$. We often write the rule (v,z) as $v \Rightarrow z$ and say that the string z *can be derived* from the variable v.

■ **Example 9.20 — Context free grammar for arithmetic expressions.** The example above of well-formed arithmetic expressions can be captured formally by the following context free grammar:

- The alphabet Σ is $\{(,),+,-,\times,\div,0,1,2,3,4,5,6,7,8,9\}$
- The variables are $V = \{expression\ ,\ number\ ,\ digit\ ,\ operation\}$.
- The rules are the set R containing the following 19 rules:
  - The 4 rules *operation* $\Rightarrow +$, *operation* $\Rightarrow -$, *operation* $\Rightarrow \times$, and *operation* $\Rightarrow \div$.
  - The 10 rules $digit \Rightarrow 0$,…, $digit \Rightarrow 9$.
  - The rule $number \Rightarrow digit$.
  - The rule $number \Rightarrow digit\ number$.
  - The rule $expression \Rightarrow number$.
  - The rule $expression \Rightarrow expression\ operation\ expression$.
  - The rule $expression \Rightarrow (expression)$.
- The starting variable is $expression$

People use many different notations to write context free grammars. One of the most common notations is the Backus–Naur form. In this notation we write a rule of the form $v \Rightarrow a$ (where v is a variable and a is a string) in the form `<v> := a`. If we have several rules of the form $v \mapsto a$, $v \mapsto b$, and $v \mapsto c$ then we can combine them as `<v> := a|b|c`. (In words we say that v can derive either a, b, or c.) For example, the Backus-Naur description for the context free grammar of Example 9.20 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 Lemma 9.14 there is no regular expression that matches a string x if and only if x contains a valid sequence of matching parenthesis.

9.6.1 Context-free grammars as a computational model

We can think of a context-free grammar over the alphabet Σ as defining a function that maps every string x in Σ^* to 1 or 0 depending on whether x can be generated by the rules of the grammars. We now make this definition formally.

Definition 9.21 — Deriving a string from a grammar. If $G = (V, R, s)$ is a context-free grammar over Σ, then for two strings $\alpha, \beta \in (\Sigma \cup V)^*$ we say that β *can be derived in one step* from α, denoted by $\alpha \Rightarrow_G \beta$, if we can obtain β from α by applying one of the rules of G. That is, we obtain β by replacing in α one occurrence of the variable v with the string z, where $v \Rightarrow z$ is a rule of G.

We say that β *can be derived* from α, denoted by $\alpha \Rightarrow_G^* \beta$, if it can be derived by some finite number k of steps. That is, if there are $\alpha_1, \ldots, \alpha_{k-1} \in (\Sigma \cup V)^*$, so that $\alpha \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \cdots \Rightarrow_G \alpha_{k-1} \Rightarrow_G \beta$.

We say that $x \in \Sigma^*$ is *matched* by $G = (V, R, s)$ if x can be derived from the starting variable s (i.e., if $s \Rightarrow_G^* x$). We define the *function computed by* (V, R, s) to be the map $\Phi_{V,R,s} : \Sigma^* \rightarrow \{0,1\}$ such that $\Phi_{V,R,s}(x) = 1$ iff x is matched by (V, R, s). A function $F : \Sigma^* \rightarrow \{0,1\}$ is *context free* if $F = \Phi_{V,R,s}$ for some CFG (V, R, s). [2]

[2] As in the case of Definition 9.3 we can also use *language* rather than *function* notation and say that a language $L \subseteq \Sigma^*$ is *context free* if the function F such that $F(x) = 1$ iff $x \in L$ is context free.

A priori it might not be clear that the map $\Phi_{V,R,s}$ is computable, but it turns out that this is the case.

Theorem 9.22 — Context-free grammars always halt. For every CFG (V, R, s) over $\{0,1\}$, the function $\Phi_{V,R,s} : \{0,1\}^* \rightarrow \{0,1\}$ is computable.

As usual we restrict attention to grammars over $\{0,1\}$ although the proof extends to any finite alphabet Σ.

Proof. 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 $u \rightarrow vw$ for variables u, v, w or the form $u \rightarrow \sigma$ for a variable u and symbol $\sigma \in \Sigma$, plus potentially the rule $s \rightarrow$ "" where s 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 $v \rightarrow u\sigma w$ into the three rules $v \rightarrow ur$, $r \rightarrow tw$ and $t \rightarrow \sigma$.

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

■

R

Remark 9.23 — Parse trees. While we focus on the task of *deciding* whether a CFG matches a string, the algorithm to compute $\Phi_{V,R,s}$ actually gives more information than that. That is, on input a string x, if $\Phi_{V,R,s}(x) = 1$ then the algorithm yields the sequence of rules that one can apply from the starting vertex s to obtain the final string x. We can think of these rules as determining a *tree* with s being the *root* vertex and the sinks (or *leaves*) corresponding to the substrings of x 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 x, and often yields very useful information about the structure of x.

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). 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, and programming language designers often try to ensure their languages have these more efficient grammars.)

9.6.2 The power of context free grammars

Context free grammars can capture every regular expression:

Theorem 9.24 — Context free grammars and regular expressions. Let e be a regular expression over $\{0,1\}$, then there is a CFG (V,R,s) over $\{0,1\}$ such that $\Phi_{V,R,s} = \Phi_e$.

Proof. We prove the theorem by induction on the length of e. If e is an expression of one bit length, then $e = 0$ or $e = 1$, 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:** $e = e'e''$, **case 2:** $e = e'|e''$ or **case 3:** $e = (e')^*$ where in all cases e', e'' are shorter regular expressions. By the induction hypothesis have grammars (V',R',s') and (V'',R'',s'') that compute $\Phi_{e'}$ and $\Phi_{e''}$

respectively. By renaming of variables, we can also assume without loss of generality that V' and V'' are disjoint.

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

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 *MATCHPAREN* can be computed by a context free grammar, whereas, as shown in Lemma 9.14, it cannot be computed by regular expressions. Here is another example:

Solved Exercise 9.2 — Context free grammar for palindromes. Let $PAL : \{0,1,;\}^* \rightarrow \{0,1\}$ be the function defined in Solved Exercise 9.1 where $PAL(w) = 1$ iff w has the form $u;u^R$. Then PAL can be computed by a context-free grammar

■

Solution:

A simple grammar computing PAL 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 w such that $PAL(w) = 1$.

■

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

Solved Exercise 9.3 — Non palindromes. Prove that there is a context free grammar that computes $NPAL : \{0,1,;\}^* \rightarrow \{0,1\}$ where $NPAL(w) = 1$ if $w = u;v$ but $v \neq u^R$.

■

Solution:

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 w such that $NPAL(w) = 1$ as having the following form

$$w = \alpha b u; u^R b' \beta \tag{9.12}$$

where α, β, u are arbitrary strings and $b \neq b'$. Hence we can generate such a string by first generating a palindrome $u; u^R$ (`palindrome` variable), then adding either 0 on the right and 1 on the left to get something that is *not* a palindrome (`different` variable), and then we can add arbitrary number of 0's and 1's on either end (the `start` variable).

■

9.6.3 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" (Theorem 9.15):

Theorem 9.25 — Context-free pumping lemma. Let (V, R, s) be a CFG over Σ, then there is some numbers $n_0, n_1 in \mathbb{N}$ such that for every $x \in \Sigma^*$ with $|x| > n_0$, if $\Phi_{V,R,s}(x) = 1$ then $x = abcde$ such that $|b| + |c| + |d| \leq n_1$, $|b| + |d| \geq 1$, and $\Phi_{V,R,s}(ab^k cd^k e) = 1$ for every $k \in \mathbb{N}$.

P

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."*

Proof of Theorem 9.25. We only sketch the proof. The idea is that if the total number of symbols in the rules of the grammar is k_0, then the only way to get $|x| > n_0$ with $\Phi_{V,R,s}(x) = 1$ is to use *recursion*. That is, there must be some variable $v \in V$ such that we are able to derive from v the value bvd for some strings $b, d \in \Sigma^*$, and then further on derive from v some string $c \in \Sigma^*$ such that bcd is a substring of x (in other words, $x = abcde$ for some $a, e \in \{0,1\}^*$). If we take the variable v satisfying this requirement with a minimum number of derivation steps, then we can ensure that $|bcd|$ is at most some constant depending on n_0 and we can set n_1 to be that constant ($n_1 = 10 \cdot |R| \cdot n_0$ will do, since we will not need more than $|R|$ applications of rules, and each such application can grow the string by at most n_0 symbols).

Thus by the definition of the grammar, we can repeat the derivation to replace the substring bcd in x with $b^k cd^k$ for every $k \in \mathbb{N}$ while retaining the property that the output of $\Phi_{V,R,s}$ is still one. Since bcd is a substring of x, we can write $x = abcde$ and are guaranteed that $ab^k cd^k e$ is matched by the grammar for every k.

■

Using Theorem 9.25 one can show that even the simple function $F : \{0,1\}^* \to \{0,1\}$ defined as follows:

$$F(x) = \begin{cases} 1 & x = ww \text{ for some } w \in \{0,1\}^* \\ 0 & \text{otherwise} \end{cases} \tag{9.13}$$

is not context free. (In contrast, the function $G : \{0,1\}^* \to \{0,1\}$ defined as $G(x) = 1$ iff $x = w_0 w_1 \cdots w_{n-1} w_{n-1} w_{n-2} \cdots w_0$ for some $w \in \{0,1\}^*$ and $n = |w|$ is context free, can you see why?.)

Solved Exercise 9.4 — Equality is not context-free. Let $EQ : \{0,1,;\}^* \to \{0,1\}$ be the function such that $EQ(x) = 1$ if and only if $x = u;u$ for some $u \in \{0,1\}^*$. Then EQ is not context free.

■

Solution:

We use the context-free pumping lemma. Suppose towards the sake of contradiction that there is a grammar G that computes EQ, and let n_0 be the constant obtained from Theorem 9.25.

Consider the string $x = 1^{n_0}0^{n_0};1^{n_0}0^{n_0}$, and write it as $x = abcde$ as per Theorem 9.25, with $|bcd| \leq n_0$ and with $|b| + |d| \geq 1$. By Theorem 9.25, it should hold that $EQ(ace) = 1$. However, by case analysis this can be shown to be a contradiction.

Firstly, unless b is on the left side of the ; separator and d is on the right side, dropping b and d will definitely make the two parts different. But if it is the case that b is on the left side and d is on the right side, then by the condition that $|bcd| \leq n_0$ we know that b is a string of only zeros and d is a string of only ones. If we drop b and d 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 $EQ(ace) = 0$, obtaining the desired contradiction.

■

9.7 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:

Theorem 9.26 — Emptiness for CFG's is decidable. There is an algorithm that on input a context-free grammar G, outputs 1 if and only if Φ_G is the constant zero function.

Proof Idea:

The proof is easier to see if we transform the grammar to Chomsky Normal Form as in Theorem 9.22. Given a grammar G, we can recursively define a non-terminal variable v to be *non empty* if there is either a rule of the form $v \Rightarrow \sigma$, or there is a rule of the form $v \Rightarrow uw$ where both u and w are non empty. Then the grammar is non empty if and only if the starting variable s is non-empty.

⋆

Proof of Theorem 9.26. We assume that the grammar G in Chomsky Normal Form as in Theorem 9.22. We consider the following procedure for marking variables as "non empty":

1. We start by marking all variables v that are involved in a rule of the form $v \Rightarrow \sigma$ as non empty.
2. We then continue to mark v as non empty if it is involved in a rule of the form $v \Rightarrow uw$ where u, w 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 s has not been marked. To see why this is a valid algorithm, note that if a variable v has been marked as "non empty" then there is some string $\alpha \in \Sigma^*$ that can be derived from v. On the other hand, if v has not been marked, then every sequence of derivations from v will always have a variable that has not been replaced by alphabet symbols. Hence in particular Φ_G is the all zero function if and only if the starting variable s is not marked "non empty".

■

9.7.1 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:

Theorem 9.27 — Fullness of CFG's is uncomputable. For every set Σ, let $CFGFULL_\Sigma$ be the function that on input a context-free grammar G over Σ, outputs 1 if and only if G computes the constant 1 function. Then there is some finite Σ such that $CFGFULL_\Sigma$ is uncomputable.

Theorem 9.27 immediately implies that equivalence for context-free grammars is uncomputable, since computing "fullness" of a grammar G over some alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{k-1}\}$ corresponds to checking whether G is equivalent to the grammar $s \Rightarrow \text{""}|s\sigma_0|\cdots|s\sigma_{k-1}$. Note that Theorem 9.27 and Theorem 9.26 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 Σ using $\lceil \log |\Sigma| \rceil$ bits (and this finite encoding can be easily carried out within a grammar) Theorem 9.27 implies that fullness is also uncomputable for grammars over the binary alphabet.

Proof Idea:

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 Definition 7.8. Recall that a *configuration* of a program P is a binary string s that encodes all the information about the program in the current iteration.

We define Σ to be $\{0,1\}$ plus some separator characters and define $INVALID_P : \Sigma^* \rightarrow \{0,1\}$ to be the function that maps every string $L \in \Sigma^*$ to 1 if and only L does *not* encode a sequence of configurations that correspond to a valid halting history of the computation of P on the empty input.

The heart of the proof is to show that $INVALID_P$ is context-free. Once we do that, we see that P halts on the empty input if and only if $INVALID_P(L) = 1$ for *every* L. 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.

⋆

Proof of Theorem 9.27. We only sketch the proof. We will show that if we can compute $CFGFULL$ then we can solve $HALTONZERO$, which has been proven uncomputable in Theorem 8.10. Let M be an input Turing machine for $HALTONZERO$. We will use the notion of *configurations* of a Turing machine, as defined in Definition 7.8.

Recall that a *configuration* of Turing machine M and input x captures the full state of M 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 $\sigma \in \{0,1\}^*$.

- The *initial* configuration of M on the input 0 is some fixed string.
- A *halting configuration* will have the value a certain state (which can be easily "read off" from it) set to 1.
- If σ is a configuration at some step i of the computation, we denote by $NEXT_M(\sigma)$ as the configuration at the next step. $NEXT_M(\sigma)$ is a string that agrees with σ 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 $NEXT_M(\sigma)$ can be computed by some finite function.

We will let the alphabet $\Sigma = \{0,1\} \cup \{\|, \#\}$. A *computation history* of M on the input 0 is a string $L \in \Sigma$ that corresponds to a list $\|\sigma_0\#\sigma_1\|\sigma_2\#\sigma_3 \cdots \sigma_{t-2}\|\sigma_{t-1}\#$ (i.e., $\|$ comes before an even numbered block, and $\|$ comes before an odd numbered one) such that if i is even then σ_i is the string encoding the configuration of P on input 0 at the beginning of its i-th iteration, and if i is odd then it is the same except the string is *reversed*. (That is, for odd i, $rev(\sigma_i)$ encodes the configuration of P on input 0 at the beginning of its i-th iteration.) Reversing the odd-numbered blocks is a technical trick to ensure that the function $INVALID_M$ we define below is context free.

We now define $INVALID_M : \Sigma^* \rightarrow \{0,1\}$ as follows:

$$INVALID_M(L) = \begin{cases} 0 & L \text{ is a valid computation history of } M \text{ on } 0 \\ 1 & \text{otherwise} \end{cases} \tag{9.14}$$

We will show the following claim:

CLAIM: $INVALID_M$ is context-free.

The claim implies the theorem. Since M halts on 0 if and only if there exists a valid computation history, $INVALID_M$ is the constant one function if and only if M does *not* halt on 0. In particular, this allows us to reduce determining whether M halts on 0 to determining whether the grammar G_M corresponding to $INVALID_M$ is full.

We now turn to the proof of the claim. We will not show all the details, but the main point $INVALID_M(L) = 1$ if *at least one* of the following three conditions hold:

1. L is not of the right format, i.e. not of the form $\langle$binary-string$\rangle\#\langle$binary-string$\rangle\|\langle$binary-string$\rangle\#\cdots$.
2. L contains a substring of the form $\|\sigma\#\sigma'\|$ such that $\sigma' \neq rev(NEXT_P(\sigma))$
3. L contains a substring of the form $\#\sigma\|\sigma'\#$ such that $\sigma' \neq NEXT_P(rev(\sigma))$

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 L *is* of the correct format can be done using a regular expression. Since regular expressions are closed under negation, this means that checking that L 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 F such that $F(u\#v) = 1$ iff $u \neq rev(v)$ is context-free, see Solved Exercise 9.3. After all, the $NEXT_M$ function only modifies its input in a constant number of places. We leave filling out the details as an exercise to the reader. Since $INVALID_M(L) = 1$ if and only if L 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.

■

9.8 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.

Table 9.1: Computability of semantic properties

Model	**Halting**	**Emptiness**	**Equivalence**
Regular expressions	Computable	Computable	Computable
Context free grammars	Computable	Computable	Uncomputable
Turing-complete models	Uncomputable	Uncomputable	Uncomputable

✓ **Lecture Recap**

- 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.

9.9 EXERCISES

Exercise 9.1 — Closure properties of regular functions. Suppose that $F, G : \{0,1\}^* \rightarrow \{0,1\}$ are regular. For each one of the following definitions of the function H, either prove that H is always regular or give a counterexample for regular F, G that would make H not regular.

1. $H(x) = F(x) \vee G(x)$.
2. $H(x) = F(x) \wedge G(x)$
3. $H(x) = NAND(F(x), G(x))$.
4. $H(x) = F(x^R)$ where x^R is the reverse of x: $x^R = x_{n-1}x_{n-2} \cdots x_o$ for $n = |x|$.
5. $H(x) = \begin{cases} 1 & x = uv \text{ s.t. } F(u) = G(v) = 1 \\ 0 & \text{otherwise} \end{cases}$
6. $H(x) = \begin{cases} 1 & x = uu \text{ s.t. } F(u) = G(u) = 1 \\ 0 & \text{otherwise} \end{cases}$
7. $H(x) = \begin{cases} 1 & x = uu^R \text{ s.t. } F(u) = G(u) = 1 \\ 0 & \text{otherwise} \end{cases}$

■

Exercise 9.2 One among the following two functions that map $\{0,1\}^*$ to $\{0,1\}$ 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. * $F(x) = 1$ if 4 divides $\sum_{i=0}^{|x|-1} x_i$ and $F(x) = 0$ otherwise.

- $G(x) = 1$ if and only if $\sum_{i=0}^{|x|-1} x_i \geq |x|/4$ and $G(x) = 0$ otherwise.

■

Exercise 9.3 — Non regularity. 1. Prove that the following function $F : \{0,1\}^* \rightarrow \{0,1\}$ is not regular. For every $x \in \{0,1\}^*$, $F(x) = 1$ iff x is of the form $x = 1^{3^i}$ for some $i > 0$.

2. Prove that the following function $F : \{0,1\}^* \rightarrow \{0,1\}$ is not regular. For every $x \in \{0,1\}^*$, $F(x) = 1$ iff $\sum_j x_j = 3^i$ for some $i > 0$.

■

Exercise 9.4 — Closure properties of context-free functions. Suppose that $F, G : \{0,1\}^* \rightarrow \{0,1\}$ are context free. For each one of the following definitions of the function H, either prove that H is always context free or give a counterexample for regular F, G that would make H not context free.

1. $H(x) = F(x) \vee G(x)$.
2. $H(x) = F(x) \wedge G(x)$
3. $H(x) = NAND(F(x), G(x))$.
4. $H(x) = F(x^R)$ where x^R is the reverse of x: $x^R = x_{n-1}x_{n-2}\cdots x_o$ for $n = |x|$.
5. $H(x) = \begin{cases} 1 & x = uv \text{ s.t. } F(u) = G(v) = 1 \\ 0 & \text{otherwise} \end{cases}$
6. $H(x) = \begin{cases} 1 & x = uu \text{ s.t. } F(u) = G(u) = 1 \\ 0 & \text{otherwise} \end{cases}$
7. $H(x) = \begin{cases} 1 & x = uu^R \text{ s.t. } F(u) = G(u) = 1 \\ 0 & \text{otherwise} \end{cases}$

■

Exercise 9.5 Prove that the function $F : \{0,1\}^* \rightarrow \{0,1\}$ such that $F(x) = 1$ if and only if $|x|$ is a power of two is not context free.

■

Exercise 9.6 — Syntax for programming languages. Consider the following syntax of a "programming language" whose source can be written using the 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 $VAR : \{0,1\}^* \rightarrow \{0,1\}$ be the function that given a string $x \in \{0,1\}^*$, outputs 1 if and only if x corresponds to an ASCII encoding of a valid variable identifier. Prove that VAR is regular.

2. Let $SYN : \{0,1\}^* \rightarrow \{0,1\}$ be the function that given a string $s \in \{0,1\}^*$, outputs 1 if and only if s is an ASCII encoding of a valid program in our language. Prove that SYN is context free. (You do not have to specify the full formal grammar for SYN, but you need to show that such a grammar exists.)

3. Prove that SYN is not regular. See footnote for hint[3]

■

[3] Try to see if you can "embed" in some way a function that looks similar to *MATCHPAREN* in *SYN*, 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.

9.10 BIBLIOGRAPHICAL NOTES

The relation of regular expressions with finite automata is a beautiful topic, on which we only touch upon in this texts. It is covered more extensively in [**hopcroft** ; Sip97; Koz97]. 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 Theorem 9.6 is closely related to the Myhill-Nerode Theorem. One direction of the Myhill-Nerode theorem theorem can be stated as saying that if e is a regular expression then there is at most a finite number of strings $z_0, \ldots, z_{k-1}$ such that $\Phi_{e[z_i]} \neq \Phi_{e[z_j]}$ for every $0 \leq i \neq j < k$.

As in the case of regular expressions, there are many resources available that cover context-free grammar in great detail. Chapter 2 of [Sip97] contains many examples of context-free grammars and their properties. There are also websites such as 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 $v \mapsto a$ means that we can *always* replace v with the string a, no matter what is the *context* in which v 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 $a \Rightarrow b$ where both a and b are strings over $(V \cup \Sigma)^*$. The idea is that if, for example, we wanted to enforce the condition that we only apply some rule such as $v \mapsto 0w1$ when v is surrounded by three zeroes on both sides, then we could do so by adding a rule of the form $000v000 \mapsto 0000w1000$ (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 A that can determine for every general grammar G and a string x, whether or not the grammar G generates x.

The Chomsky Hierarchy is a hierarchy of grammars from the least restrictive (most powerful) Type 0 grammars, which correspond to *recursively enumerable* languages (see Exercise 8.10) 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 which are non-deterministic algorithms that take $O(n)$ space. For this reason, the class of functions/languages corresponding to context-sensitive grammars is also known as the complexity class **NSPACE**$O(n)$; we discuss space-bounded complexity in Chapter 16). 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.