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.

9
Restricted computational models

> *"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 a great many models of computation are *Turing equivalent*, including our NAND++/NAND« programs and Turing machines, standard programming languages such as C/Python/-Javascript etc.., and 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 (see Section 8.4.1) holds as well, which means that deciding 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 coming up with **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 will discuss several such examples.

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, CCS, SQL, sendmail configuration, as well as games such as Minecraft, Super Mario, and the card game "Magic: The gathering". This 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 gives 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 the biggest crowdfunding success in history and at its height 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 Turing-complete 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 would cause the DAO code to essentially enter into an infinite recursive loop in which it continuously transferred funds into their 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. There was a lot of debate in the Ethereum community how to handle this, including some partially successful "Robin Hood" attempts to use the same loophole to drain the DAO funds into a

secure account. Eventually it turned out that the code is mutable, stoppable, and refutable after all, and the Ethereum community decided to do a "hard fork" (also known as a "bailout") to revert history to before this transaction. Some elements of the community 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. The system has a collection $X = \{x_0, \ldots, x_k\}$ of strings (for example filenames on a hard-drive, or names of students inside a database), and the user wants to find out the subset of all the $x \in X$ that are *matched* by some pattern. (For example all files that end with the string `.txt`.) In full generality, we could allow the user to specify the pattern by giving out an arbitrary function $F : \{0,1\}^* \to \{0,1\}$, where $F(x) = 1$ corresponds to the pattern matching x. For example, the user could give a *program* P in some Turing-complete programming language such as *Python*, and the system will return all the x_i's such that $P(x_i) = 1$. However, we don't want our system to get into an infinite loop just trying to evaluate this function!

Because the Halting problem for Turing-complete computational models is uncomputable, a system would not be able to verify that a given program P does not halt. For this reason, typical systems for searching files or databases do *not* allow users to specify functions in full-fledged programming languages. Rather, they use *restricted computational models* that 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 `(xxx) xxx-xxxx` inside a textfile), but restricted enough so that they cannot result in an finite loop. One of the most popular models for this application is regular expressions. You have probably come across regular expressions if you ever used an advanced text editor, a command line shell, or have done any kind of manipulations of text files.

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).[1] For example, the following regular expression over the alphabet $\{0,1\}$ corresponds to the set of all even length strings $x \in \{0,1\}^*$ where the digit at location $2i$ is the same as the one at location $2i+1$

[1] Common implementations of regular expressions in programming languages and shells typically include some extra operations on top of $|$ and $*$, but these can all be implemented as "syntactic sugar" using the operators $|$ and $*$.

for every i:

$$(00|11)* \tag{9.1}$$

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:[2]

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

1. $exp = \sigma$ where $\sigma \in \Sigma$
2. $exp = (exp'|exp'')$ where exp', exp'' are regular expressions.
3. $exp = (exp')(exp'')$ where exp', exp'' are regular expressions. (We often drop the parenthesis when there is no danger of confusion and so write this as $exp\ exp'$.)
4. $exp = (exp')*$ where exp' is a regular expression.

Finally we also allow the following "edge cases": $exp = \emptyset$ and $exp = \text{""}$. [3]

[2] We have seen a recursive definition before in the setting of λ expressions (Definition 7.4). Just like recursive functions, we can define a concept recursively. A definition of some class $\mathcal{C}$ of objects can be thought of as defining a function that maps an object o to either $VALID$ or $INVALID$ depending on whether $o \in \mathcal{C}$. Thus we can think of Definition 9.2 as defining a recursive function that maps a string exp over $\Sigma \cup \{\text{"("}, \text{")"}, \text{"|"}, \text{"}*\text{"}\}$ to $VALID$ or $INVALID$ depending on whether exp describes a valid regular expression.

[3] These are the regular expressions corresponding to accepting no strings, and accepting only the empty string respectively.

Every regular expression exp corresponds to a function $\Phi_{exp} : \Sigma^* \to \{0,1\}$ where $\Phi_{exp}(x) = 1$ if x *matches* the regular expression. The definition of "matching" is recursive as well. For example, if exp and exp' match the strings x and x', then the expression $exp\ exp'$ matches the concatenated string xx'. Similarly, if $exp = (00|11)^*$ then $\Phi_{exp}(x) = 1$ if and only if x is of even length and $x_{2i} = x_{2i+1}$ for every $i < |x|/2$. We now turn to the formal definition of Φ_{exp}.

P The formal definition of Φ_{exp} 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 exp be a regular expression. Then the function Φ_{exp} is defined as follows:

1. If $exp = \sigma$ then $\Phi_{exp}(x) = 1$ iff $x = \sigma$.
2. If $exp = (exp'|exp'')$ then $\Phi_{exp}(x) = \Phi_{exp'}(x) \vee \Phi_{exp''}(x)$ where $\vee$ is the OR operator.
3. If $exp = (exp')(exp'')$ then $\Phi_{exp}(x) = 1$ iff there is some $x', x'' \in \Sigma^*$ such that x is the concatenation of x' and x'' and $\Phi_{exp'}(x') = \Phi_{exp''}(x'') = 1$.
4. If $exp = (exp')*$ then $\Phi_{exp}(x) = 1$ 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 $\Phi_{exp'}(x_i) = 1$ for every $i \in [k]$.
5. Finally, for the edge cases $\Phi_{\emptyset}$ is the constant zero function, and $\Phi_{\text{""}}$ is the function that only outputs 1 on the empty string "".

We say that a regular expresion exp over Σ *matches* a string $x \in \Sigma^*$ if $\Phi_{exp}(x) = 1$. We say that a function $F : \Sigma^* \to \{0,1\}$ is *regular* if $F = \Phi_{exp}$ for some regular expression exp. [4]

[4] We use *function notation* in this book, but other texts often use the notion of *languages*, which are sets of string. We say that a language $L \subseteq \Sigma^*$ is *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 $x \in L$.

■ **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). As shown by Eq. (9.2), the function F is regular. Specifically, $F = \Phi_{(a|b|c|d)(a|b|c|d)*(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*}$. [5]

If we wanted to verify, for example, that $\Phi_{(a|b|c|d)(a|b|c|d)*(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*}$ does output 1 on the string $abc12078$, 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 $a|b|c|d$.

[5] Formally we should write $((a|b)|c)|d)$ instead of $a|b|c|d$ but for clarity we will use using the convention that OR and concatenation are left-associative, and that we give precedence to $*$, then concatenation, and then OR. Most of the time we will only explicitly write down the parenthesis that are not implied by these rules.

P The definitions above are not inherently difficult, but are a bit cumbersom. 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.

We can think of regular expressions as a type of "programming language". That is, we can think of a regular expression exp over the alphabet Σ as a program that computes the function $\Phi_{exp} : \Sigma^* \to \{0,1\}$.[6] It turns out that this "regular expression programming language" is simple in the sense that for every regular expression exp, we can compute the function Φ_{exp} by a Turing Machine / NAND++ program that always halts:

Theorem 9.5 — Regular expression always halt. For every finite set Σ and $exp \in (\Sigma \cup \{(,),|,*,\emptyset,\text{""}\})^*$, if exp is a valid regular expression over Σ then Φ_{exp} is a total computable function from Σ^* to $\{0,1\}$. That is, there is an always halting NAND++ program P_{exp} that computes Φ_{exp}.[7]

Proof Idea: The main idea behind the proof is to see that Definition 9.3 actually specifies a recursive algorithm for *computing* $\Phi_e xp$. Specifically, each one of our operations -concatenation, OR, and star- can be thought of as reducing the task of testing whether an expression exp matches a string x to testing whether some sub-expressions of exp match substrings of x. Since these sub-expressions are always shorter than the original expression, this yields a recursive algorithm for checking if exp matches x which will eventually terminate at the base cases of the expressions that correspond to a single symbol or the empty string. The details are specified below. ⋆

Proof of Theorem 9.5. Definition 9.3 gives a way of recursively computing Φ_{exp}. The key observation is that in our recursive definition of regular expressions, whenever exp is made up of one or two expressions exp', exp'' then these two regular expressions are *smaller* than exp, 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 exp (i.e., the number of symbols in the string exp, also denoted as $|exp|$). For $m = 1$, exp is either a single alphabet symbol, "" or $\emptyset$, and so computing the function Φ_{exp} is straightforward. In the general case, for $m = |exp|$ 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_{exp} will be computable in all these cases:

[6] Regular expressions (and context free grammars, which we'll see below) are often thought of as *generative models* rather than computational ones, since their definition does not immediately give rise to a way to *decide* matches but rather to a way to generate matching strings by repeatedly choosing which rules to apply.

[7] Formally, we only defined the notion of NAND++ programs that compute functions whose inputs are *binary* strings, but as usual we can represent non-binary strings over the binary alphabet. Specifically, since Σ is a finite set, we can always represent an element of it by a binary string of length $\lceil \log |\Sigma| \rceil$, and so can represent a string $x \in \Sigma^*$ as a string $\hat{x} \in \{0,1\}^*$ of length $\lceil \log |\Sigma| \rceil |x|$.

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

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

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

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

$$\bigvee_{i=0}^{|x|-1} (\Phi_{exp'}(x_0 \cdots x_{i-1}) \wedge \Phi_{exp''}(x_i \cdots x_{|x|-1})) \tag{9.3}$$

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

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

In this case by the inductive hypothesis we can compute $\Phi_{exp'}$ and so we can compute $\Phi_{exp}(x)$ by enumerating over all k from 1 to $|x|$, and all ways to write x as the concatenation of k 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, $\Phi_{exp'}(x_0) = \cdots = \Phi_{exp'}(x_{k-1}) = 1$ then we output 1. Otherwise we output 0.

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

9.2.1 Efficient matching of regular expressions (advanced, 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 to match regular expressions. In particular, for every regular expression exp there is an algorithm that on input $x \in \{0,1\}^n$, computes $\Phi_{exp}(x)$ in "$O(n)$ running time".[8] One way to obtain such an algorithm is to replace this recursive algorithm with dynamic programming, using the technique of memoization.[9]

It turns out that the resulting dynamic program not only runs in $O(n)$ time, but in fact uses only a constant amount of memory, and makes a single pass over its input. Such an algorithm is also known as a deterministic finite automaton (DFA). It is also known that *every* function that can be computed by a deterministic finite automaton is regular. The relation of regular expressions with finite automata is a beautiful topic, on which we only touch upon in this texts. See books such as Sipser's, Hopcroft, Motwani and Ullman, and Kozen's.

We now prove the algorithmic result that regular expression match-

[8] For now we use the colloquial notion of running time as is used in introduction to programming courses and white-board coding interviews. We will see in Chapter 12 how to formally define running time.

[9] If you haven't taken yet an algorithms course, you might not know these techniques. This is OK; while the more efficient algorithm is crucial for the many practical applications of regular expressions, it is not of great importance to this course.

ing can be done by a linear (i.e., $O(n)$) time algorithm, and moreover one that uses a constant amount of memory, and makes a single pass over its input.[10] Since we have not yet covered the topics of time and space complexity, we we describe the algorithm in high level terms, without making the computational model precise. In Chapter 16 we will define space complexity formally, and prove the equivalence of deterministic finite automata and regular expression (see also Theorem 9.7 below), which will also imply Theorem 9.6.

[10] We say that an algorithm A for matching regular expressions uses a constant, or $O(1)$, memory, if for every regular expression exp there exists some number C such that for every input $x \in \{0,1\}^*$, A utilizes at most C bits of working memory to compute $\Phi_{exp}(x)$, no matter how long x is.

Theorem 9.6 — DFA for regular expression matching. Let exp be a regular expression. Then there is an $O(n)$ time algorithm that computes Φ_{exp}.

Moreover, this algorithm only makes a single pass over the input, and utilizes only a constant amount of working memory. That is, it is a deterministic finite automaton.

We note that this theorem is very interesting even if one ignores the part following the "moreover". Hence, the reader is very welcome to ignore this part in the first pass over the theorem and its proof.

Proof Idea: The idea is to first obtain a more efficient recursive algorithm for computing Φ_{exp} and then turning this recursive algorithm into a constant-space single-pass algorithm using the technique of *memoization*. In this technique we record in a table the results of every call to a function, and then if we make future calls with the same input, we retrieve the result from the table instead of re-computing it. This simple optimization can sometimes result in huge savings in running time.

In this case, we can define a recursive algorithm that on input a regular expression exp and a string $x \in \{0,1\}^n$, computes $\Phi_{exp}(x)$ as follows:

- If $n = 0$ (i.e., x is the empty string) then we output 1 iff exp contains "".
- If $n > 0$, we let $\sigma = x_{n-1}$ and let $exp' = exp[\sigma]$ to be the regular expression that matches a string x iff exp matches the string $x\sigma$. (It can be shown that such a regular expression exp' exists and is in fact of equal or smaller "complexity" to exp for some appropriate notion of complexity.) We use a recursive call to return $\Phi_{exp'}(x_0 \cdots x_{n-1})$.

The running time of this recursive algorithm can be computed by the formula $T(n) = T(n-1) + O(1)$ which solves to $O(n)$ (where the constant in the running time can depend on the length of the regular expression exp).

If we want to get the stronger result of a *constant space algorithm* (i.e., DFA) then we can use *memoization*. Specifically, we will store a table of the (constantly many) expressions of length at most $|exp|$ that we need to deal with in the course of this algorithm, and iteratively for $i = 0, 1, \ldots, n-1$, compute whether or not each one of those expressions matches $x_0 \cdots x_{i-1}$. ⋆

Proof of Theorem 9.6. The central definition for this proof is the notion of a *restriction* of a regular expression. For a regular expression exp over an alphabet Σ and symbol $\sigma \in \Sigma$, we will define $exp[\sigma]$ to be a regular expression such that $exp[\sigma]$ matches a string x if and only if exp matches the string $x\sigma$. For example, if exp is the regular expression $01|(01)*(01)$ (i.e., one or more occurences of 01) then $exp[1]$ will be $0|(01)*0$ and $exp[0]$ will be $\emptyset$.

Given an expression exp and $\sigma \in \{0,1\}$, we can compute $exp[\sigma]$ recursively as follows:

1. If $exp = \tau$ for $\tau \in \Sigma$ then $exp[\sigma] =$ "" if $\tau = \sigma$ and $exp[\sigma] = \emptyset$ otherwise.
2. If $exp = exp'|exp''$ then $exp[\sigma] = exp'[\sigma]|exp''[\sigma]$.
3. If $exp = exp'\ exp''$ then $exp[\sigma] = exp'\ exp''[\sigma]$ if exp'' can not match the empty string. Otherwise, $exp[\sigma] = exp'\ exp''[\sigma]|exp'[\sigma]$
4. If $exp = (exp')^*$ then $exp[\sigma] = (exp')^*(exp'[\sigma])$.
5. If $exp =$ "" or $exp = \emptyset$ then $exp[\sigma] = \emptyset$.

We leave it as an exercise to prove the following claim: (which can be shown by induction following the recusrive definition of Φ_{exp})

Claim: For every $x \in \{0,1\}^*$, $\Phi_{exp}(x\sigma) = 1$ if and only if $\Phi_{exp[\sigma]}(x) = 1$

The claim above suggests the following algorithm:

A recursive linear time algorithm for regular expression matching: We can now define a recursive algorithm for computing Φ_{exp}:

> **Algorithm** $MATCH(exp, x)$**:**
>
> **Inputs:** exp is normal form regular expression, $x \in \Sigma^n$ for some $n \in \mathbb{N}$.
>
> 1. If $x =$ "" then return 1 iff exp has the form $exp =$ ""$|exp'$ for some exp'. (For a normal-form expression, this is the only way it matches the empty string.)
> 2. Otherwise, return $MATCH(exp[x_{n-1}], x_0 \cdots x_{n-1})$.

Algorithm $MATCH$ is a recursive algorithm that on input an expression exp and a string $x \in \{0,1\}^n$, does some constant time computation and then calls itself on input some expression exp' and a string x of length $n-1$. It will terminate after n steps when it reaches a string of length 0.

There is one subtle issue and that is that to bound the running time, we need to show that if we let exp_i be the regular expression that this algorithm obtains at step i, then exp_i does not become itself much larger than the original expression exp. If exp is a regular expression, then for every $n \in \mathbb{N}$ and string $\alpha \in \{0,1\}^n$, we will denote by $exp[\alpha]$ the expression $(((exp[\alpha_0])[\alpha_1])\cdots)[\alpha_{n-1}]$. That is, $exp[\alpha]$ is the expression obtained by considering the restriction $exp_0 = exp[\alpha_0]$, and then considering the restriction $x_1 = exp_0[\alpha_1]$ and so on and so forth. We can also think of $exp[\alpha]$ as the regular expression that matches x if and only if exp matches $x\alpha_{n-1}\alpha_{n-2}\cdots\alpha_0$.

The expressions considered by Algorithm $MATCH$ all have the form $exp[\alpha]$ for some string α where exp is the original input expression. Thus the following claim will help us bound our algorithms complexity:[11]

Claim: For every regular expresion exp, the set $S(exp) = \{exp[\alpha] | \alpha \in \{0,1\}^*\}$ is finite.

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

The bottom line is that while running our algorithm on a regular expression exp, all the expressions we will ever encounter will be in the finite set $S(exp)$, no matter how large the input x is. Therefore, the running time of $MATCH$ is $O(n)$ where the implicit constant in the Oh notation can (and will) depend on exp but crucially, not on the length of the input x.

Proving the "moreover" part: At this point, we have already

[11] This claim is strongly related to the Myhill-Nerode Theorem. One direction of this theorem can be thought of as saying that if exp is a regular expression then there is at most a finite number of strings $z_0, \ldots, z_{k-1}$ such that $\Phi_{exp[z_i]} \neq \Phi_{exp[z_j]}$ for every $0 \leq i \neq j < k$.

proven a highly non-trivial statement: the existence of a linear-time algorithm for matching regular expressions. The reader may well be content with this, and stop reading the proof at this point. However, as mentioned above, we can do even more and in fact have a *constant space* algorithm for this. To do so, we will turn our recursive algorithm into an iterative *dynamic program*. Specifically, we replace our recursive algorithm $MATCH$ with the following iterative algorithm $MATCH'$:

Algorithm $MATCH'(exp, x)$**:**

Inputs: exp is normal form regular expression, $x \in \Sigma^n$ for some $n \in \mathbb{N}$.

Operation:

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

Algorithm $MATCH'$ maintains the invariant that at the end of step i, for every $exp' \in S$, the variable $v_{exp'}$ is equal if and only if exp' matches the string $x_0 \cdots x_{i-1}$. In particular, at the very end, v_{exp} is equal to 1 if and only if exp matches the full string $x_0 \cdots x_{n-1}$. Note that $MATCH'$ 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. ■

9.2.2 Equivalence of DFA's and regular expressions (optional)

Surprisingly, regular expressions and constant-space algorithms turn out to be *equivalent* in power. That is, the following theorem is known

Theorem 9.7 — Regular expressions are equivalent to constant-space algorithms. Let Σ be a finite set and $F : \Sigma^* \to \{0,1\}$. Then F is regular if and only if there exists a $O(1)$-space algorithm to compute F. Moreover, if F can be computed by a $O(1)$-space algorithm, then

it can also be computed by such an algorithm that makes a single pass over its input, i.e., a *deterministic finite automaton*.

One direction of Theorem 9.7 (namely that if F is regular then it is computable by a constant-space one-pass algorithm) follows from Theorem 9.6. The other direction can be shown using similar ideas. We defer the full proof of Theorem 9.7 to Chapter 16, where we will formally define *space complexity*. However, we do state here an important corollary:

Lemma 9.8 — Regular expressions closed under complement. If $F : \Sigma^* \to \{0,1\}$ is regular then so is the function $\overline{F}$, where $\overline{F}(x) = 1 - F(x)$ for every $x \in \Sigma^*$.

Proof. If F is regular then by Theorem 9.6 it can be computed by a constant-space algorithm A. But then the algorithm $\overline{A}$ which does the same computation and outputs the negation of the output of A also utilizes constant space and computes $\overline{F}$. By Theorem 9.7 this implies that $\overline{F}$ is regular as well. ■

9.3 LIMITATIONS OF REGULAR EXPRESSIONS

The fact that functions computed by regular expressions always halt is of course one of the reasons why they are so useful. When you make a regular expression search, you are guaranteed that you will get a result. This is why operating systems, for example, restrict you for searching a file via regular expressions and don't allow searching by specifying an arbitrary function via a general-purpose programming language. But this always-halting property comes at a cost. Regular expressions cannot compute every function that is computable by NAND++ programs. In fact there are some very simple (and useful!) functions that they cannot compute, such as the following:

Lemma 9.9 — Matching parenthesis. Let $\Sigma = \{\langle, \rangle\}$ and $MATCHPAREN : \Sigma^* \to \{0,1\}$ 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 9.9 is a consequence of the following result known as the *pumping lemma*:

Theorem 9.10 — Pumping Lemma. Let exp be a regular expression. Then there is some number n_0 such that for every $w \in \{0,1\}^*$ with $|w| > n_0$ and $\Phi_{exp}(w) = 1$, it holds that we can write $w = xyz$ where $|y| \geq 1$, $|xy| \leq n_0$ and such that $\Phi_{exp}(xy^k z) = 1$ for every $k \in \mathbb{N}$.

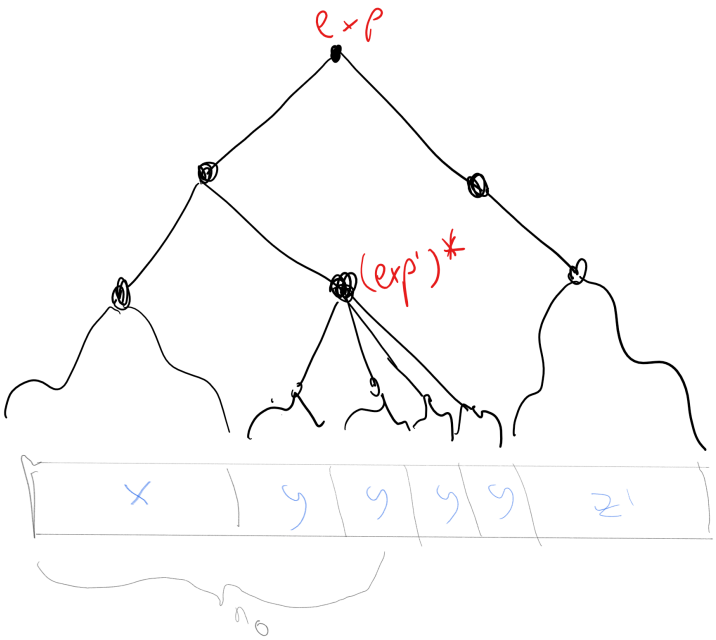

Figure 9.1: To prove the "pumping lemma" we look at a word w that is much larger than the regular expression exp that matches it. In such a case, part of w must be matched by some sub-expression of the form $(exp')*$, since this is the only operator that allows matching words longer than the expression. 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 is very simple (see Fig. 9.1). If we let n_0 be, say, twice the number of symbols that are used in the expression exp, then the only way that there is some w with $|w| > n_0$ and $\Phi_{exp}(w) = 1$ is that exp contains the $*$ (i.e. star) operator and that there is a nonempty substring y of w that was matched by $(exp')^*$ for some sub-expression exp' of exp. We can now repeat y any number of times and still get a matching string. ⋆

> 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 9.10. 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 comfort 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 or that it is $\emptyset$ or "" in which case the condition is satisfied just because there is no matching string of length more than one. Otherwise, exp is of the form **(a)** $exp'|exp''$, **(b)**, $(exp')(exp'')$, **(c)** or $(exp')^*$ where in all these cases the subexpressions have fewer symbols than exp and hence satisfy the induction hypothesis.

In case **(a)**, every string w matching exp must match either exp' or exp''. In the former case, since exp' satisfies the induction hypothesis, if $|w| > n_0$ then we can write $w = xyz$ such that xy^kz matches exp' for every k, and hence this is matched by exp as well.

In case **(b)**, if w matches $(exp')(exp'')$. then we can write $w = w'w''$ where w' matches exp' and w'' matches exp''. Again we split to subcases. If $|w'| > 2|exp'|$, then by the induction hypothesis we can write $w' = xyz$ of the form above such that xy^kz matches exp' for every k and then xy^kzw'' matches $(exp')(exp'')$. This completes the proof since $|xy| \leq 2|exp'|$ and so in particular $|xy| \leq 2(|exp'| + |exp''|) \leq 2|exp$, and hence zw'' can be play the role of z in the proof. Otherwise, if $|w'| \leq 2|exp'|$ then since $|w|$ is larger than $2|exp|$ and $w = w'w''$ and $exp = exp'exp''$, we get that $|w'| + |w''| > 2(|exp'| + |exp''|)$. Thus, if $|w'| \leq 2|exp'|$ it must be that $|w''| > 2|exp''|$ and hence by the induction hypothesis we can write $w'' = xyz$ such that xy^kz matches exp'' for every k and $|xy| \leq 2|exp''|$. Therefore we get that

$w'xy^kz$ matches $(exp')(exp'')$ for every k and since $|w'| \leq 2|exp'|$, $|w'xy| \leq 2(|exp'| + |exp'|)$ and this completes the proof since $w'x$ can play the role of x in the statement.

Now in the case **(c)**, if w matches $(exp')^*$ then $w = w_0 \cdots w_t$ where w_i is a nonempty string that matches exp' for every i. If $|w_0| > 2|exp'|$ 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^kz will match $(exp')^*$ for every k. ■

> **R** **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.

Given the pumping lemma, we can easily prove Lemma 9.9:

Proof of Lemma 9.9. Suppose, towards the sake of contradiction, that there is an expression exp such that $\Phi_{exp} = MATCHPAREN$. Let n_0 be the number from Lemma 9.9 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 9.9, 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 $\Phi_{exp}(xy^2z) = 1$, contradicting our assumption that $\Phi_{exp} = MATCHPAREN$.

The pumping lemma is a very useful tool to show that certain functions are *not* computable by a regular language. However, it is *not* an "if and only if" condition for regularity. There are non regular functions which still satisfy the conditions of the pumping lemma. To understand the pumping lemma, it is important to follow the order of quantifiers in Theorem 9.10. In particular, the number n_0 in the statement of Theorem 9.10 depends on the regular expression (in particular we can choose 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 exp computing some function F, we need to be able to choose an appropriate w 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. ■

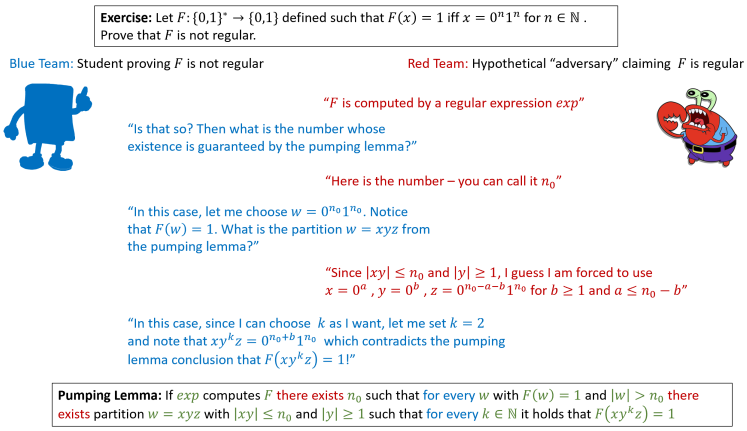

Figure 9.2: A cartoon of a proof using the pumping lemma that a function F is not regular. The pumping lemma states that if F is regular then *there exists* a number n_0 such that *for every* large enough w with $F(w)=1$, *there exists* a partition of w to $w=xyz$ satisfying certain conditions such that *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.

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 $u_{|u|-1}\cdots u_0$.[12] ■

[12] 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.

Solution: We use the pumping lemma. Suppose towards the sake of contradiction that there is a regular expression exp computing PAL, and let n_0 be the number obtained by the pumping lemma (Theorem 9.10). 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 exp, 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. 9.2 which illustrates a cartoon of the proof of the non-regularity of the function $F:\{0,1\}^* \to \{0,1\}$ 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).

9.4 OTHER SEMANTIC PROPERTIES OF REGULAR EXPRESSIONS

Regular expressions are widely used beyond just searching. First, they are typically used to define *tokens* in various formalisms such as programming data description languages. But they are also used beyond it. 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", that are run by programmable switches that can implement policies such as "if packet is 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 that where 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 that.

Such applications use the fact that, due to their restrictions, we can solve not just the halting problem for them, but also answer several other semantic questions as well, all of whom would not be solvable for Turing complete models due to Rice's Theorem (Theorem 8.7). For example, we can tell whether two regular expressions are *equivalent*, as well as whether a regular expression computes the constant zero function.

Theorem 9.11 — Emptiness of regular languages is computable. There is an algorithm that given a regular expression exp, outputs 1 if and only if Φ_{exp} 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 it will output the constant zero function is if it has the form $\emptyset$ or is obtained by concatenating $\emptyset$ with other expressions. ⋆

Proof of Theorem 9.11. Define a regular expression to be "empty" if it computes the constant zero function. The algorithm simply follows the following rules:

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

- σ and "" are not empty.

Using these rules it is straightforward to come up with a recursive algorithm to determine emptiness. We leave verifying the details to the reader. ■

Theorem 9.12 — Equivalence of regular expressions is computable. There is an efficient algorithm that on input two regular expressions exp, exp', outputs 1 if and only if $\Phi_{exp} = \Phi_{exp'}$.

Proof. Theorem 9.11 above is actually a special case of Theorem 9.12, since emptiness is the same as checking equivalence with the expression $\emptyset$. However we prove Theorem 9.12 from Theorem 9.11. The idea is that given exp and exp', we will compute an expression exp'' such that $\Phi_{exp''}(x) = (\Phi_{exp}(x) \wedge \overline{\Phi_{exp'}(x)}) \vee (\overline{\Phi_{exp}(x)} \wedge \Phi_{exp'}(x))$ (where $\overline{y}$ denotes the negation of y, i.e., $\overline{y} = 1 - y$). One can see that exp is equivalent to exp' if and only if exp'' is empty. To construct this expression, we need to show how given expressions exp and exp', we can construct expressions $exp \wedge exp'$ and $\overline{exp}$ that compute the functions $\Phi_{exp} \wedge \Phi_{exp'}$ and $\overline{\Phi_{exp}}$ respectively. (Computing the expression for $exp \vee exp'$ is straightforward using the $|$ operation of regular expressions.)

Specifically, by Lemma 9.8, regular functions are closed under negation, which means that for every regular expression exp over the alphabet Σ there is an expression $\overline{exp}$ such that $\Phi_{\overline{exp}}(x) = 1 - \Phi_{exp}(x)$ for every $x \in \Sigma^*$. For every two expressions exp and exp' we can define $exp \vee exp'$ to be simply the expression $exp|exp'$ and $exp \wedge exp'$ as $\overline{\overline{exp} \vee \overline{exp'}}$. Now we can define

$$exp'' = (exp \wedge \overline{exp'}) \vee (\overline{exp} \wedge exp') \tag{9.4}$$

and verify that $\Phi_{exp''}$ is the constant zero function if and only if $\Phi_{exp}(x) = \Phi_{exp'}(x)$ for every $x \in \Sigma^*$. Since by Theorem 9.11 we can verify emptiness of exp'', we can also verify equivalence of exp and exp'. ■

9.5 CONTEXT FREE GRAMMARS

If you have ever written a program, you've experienced a *syntax error*. You might also have 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 when trying to figure out if your program has a syntax error.

When a person designs a programming language, they need to come up with a function $VALID : \{0,1\}^* \to \{0,1\}$ that determines

the strings that correspond to valid programs in this language. The compiler or interpreter computes $VALID$ on the string corresponding to your source code to determine if there is a syntax error. To ensure that the compiler will always halt in this computation, language designers typically *don't* use a general Turing-complete mechanism to express the function $VALID$, but rather a restricted computational model. One of the most popular choices for such a model is *context free grammar*.

To explain context free grammars, let's begin with a canonical example. Let us try to define a function $ARITH : \Sigma^* \to \{0,1\}$ 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 to smaller expressions, or enclosing them in parenthesis, where the "base case" corresponds to expressions that are simply numbers. A bit 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.[13]
- An *operation* is one of $+,-,\times,\div$
- An *expression* has either the form "*number*" or the form "*subexpression1 operation subexpression2*" or "(*subexpression*)".

[13] 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.

A context free grammar (CFG) is a formal way of specifying such conditions. We can think of a CFG as a set of rules to *generate* valid expressions. In the example above, the rule $expression \Rightarrow expression \times expression$ tells us that if we have built two valid expressions $exp1$ and $exp2$, then the expression $exp1 \times exp2$ is valid above.

We can divide our rules to "base rules" and "recursive rules". The "base rules" are rules such as $number \Rightarrow 0$, $number \Rightarrow 1$, $number \Rightarrow 2$ and so on, that tell us that a single digit is a number. The "recursive rules" are rules such as $number \Rightarrow number\,0$, $number \Rightarrow number\,1$ and so on, that tell us that if we add a digit to a valid number then we still have a valid number. We now make the formal definition of context-free grammars:

Definition 9.13 — Context Free Grammar. Let Σ be some finite set. A *context free grammar (CFG) over* Σ is a triple (V,R,s) where V is a set disjoint from Σ of *variables*, R is a set of *rules*, which are pairs (v,z) (which we will write as $v \Rightarrow z$) where $v \in V$ and $z \in (\Sigma \cup V)^*$,

and $s \in V$ is the starting rule.

■ **Example 9.14 — 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 correspond the set R containing the following pairs:
  - $operation \Rightarrow +$, $operation \Rightarrow -$, $operation \Rightarrow \times$, $operation \Rightarrow \div$
  - $digit \Rightarrow 0$,..., $digit \Rightarrow 9$
  - $number \Rightarrow digit$
  - $number \Rightarrow digit\ number$
  - $expression \Rightarrow number$
  - $expression \Rightarrow expression\ operation\ expression$
  - $expression \Rightarrow (expression)$
- The starting variable is $expression$

There are various notations to write context free grammars in the literature, with one of the most common being Backus–Naur form where 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` (and this similarly extends for the case of more rules). For example, the Backus-Naur description for the context free grammar above 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)
```

You can verify that 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.

9.5.1 Context-free grammars as a computational model

We can think of a CFG 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 9.15 — 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 occurence 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 define the *function computed by* (V, R, s) to be the map $\Phi_{V,R,s} : \Sigma^* \to \{0,1\}$ such that $\Phi_{V,R,s}(x) = 1$ iff $s \Rightarrow^*_G x$.

We say that $F : \Sigma^* \to \{0,1\}$ is *context free* if $F = \Phi_{V,R,s}$ for some CFG (V, R, s). [14]

[14] 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 we can in fact compute it. That is, the "halting problem" for context free grammars is trivial:

Theorem 9.16 — Context-free grammars always halt. For every CFG (V, R, s) over Σ, the function $\Phi_{V,R,s} : \Sigma^* \to \{0,1\}$ is computable.

Proof. We only sketch the proof. It turns out that we can convert every CFG to an equivalent version that has the so called *Chomsky normal form*, where all rules either have the form $u \to vw$ for variables u, v, w or the form $u \to \sigma$ for a variable u and symbol $\sigma \in \Sigma$, plus potentially the rule $s \to$ "" 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 $v \to u\sigma w$ into the three rules $v \to ur$, $r \to tw$ and $t \to \sigma$.)

Using this 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 $s \to uv$ 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 **Parse trees** While we present CFGs as merely *deciding* whether the syntax is correct or not, 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 connected directed acylic graph (i.e., a *tree*) with s being a source (or *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 (often known in this context 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.)

9.5.2 The power of context free grammars

While we can (and people do) talk about context free grammars over any alphabet Σ, in the following we will restrict ourselves to $\Sigma = \{0,1\}$. This is of course not a big restriction, as any finite alphabet Σ can be encoded as strings of some finite size. It turns out that context free grammars can capture every regular expression:

Theorem 9.17 — Context free grammars and regular expressions. Let exp be a regular expression over $\{0,1\}$, then there is a CFG (V,R,s) over $\{0,1\}$ such that $\Phi_{V,R,s} = \Phi_{exp}$.

Proof. We will prove this by induction on the length of exp. If exp is an expression of one bit length, then $exp = 0$ or $exp = 1$, 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: $exp = exp'exp''$, **case 2:** $exp = exp'|exp''$ or **case 3:** $exp = (exp')^*$ where in all cases exp', exp'' are shorter regular expressions. By the induction hypothesis have grammars (V', R', s') and (V'', R'', s'') that compute $\Phi_{exp'}$ and $\Phi_{exp''}$ 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 (again 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 9.9, it cannot be computed by regular expressions. Here is another example:

Solved Exercise 9.2 — Context free grammar for palindromes. Let $PAL : \{0,1,;\}^* \to \{0,1\}$ 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 $NPAL : \{0,1,;\}^* \to \{0,1\}$ 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

$$w = \alpha b u ; u^R b' \beta \tag{9.5}$$

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). ■

9.5.3 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 9.10):

Theorem 9.18 — Context-free pumping lemma. Let (V, R, s) be a CFG over Σ, then there is some $n_0 \in \mathbb{N}$ 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 $k \in \mathbb{N}$.

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 9.18. We only sketch the proof. The idea is that if the total number of symbols in the rules R is k_0, then the only way to get $|x| > k_0$ 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. If we try to take the minimal such v, then we can ensure that $|bcd|$ is at most some constant depending on k_0 and we can set n_0 to be that constant ($n_0 = 10 \cdot |R| \cdot k_0$

will do, since we will not need more than $|R|$ applications of rules, and each such application can grow the string by at most k_0 symbols). Thus by the definition of the grammar, we can repeat the derivation to replace the substring bcd in x with $b^k c d^k$ for every $k \in \mathbb{N}$ while retaining the property that the output of $\Phi_{V,R,s}$ is still one. ■

Using Theorem 9.18 one can show that even the simple function $F(x) = 1$ iff $x = ww$ for some $w \in \{0,1\}^*$ is not context free. (In contrast, the function $F(x) = 1$ iff $x = ww^R$ for $w \in \{0,1\}^*$ where for $w \in \{0,1\}^n$, $w^R = w_{n-1}w_{n-2}\cdots w_0$ 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 $F(x) = 1$ 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 9.18. 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 9.18, with $|bcd| \leq n_0$ and with $|b| + |d| \geq 1$. By Theorem 9.18, it should hold that $EQ(ace) = 1$. However, by case analysis this can be shown to be a contradiction. First of all, 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. ■

9.6 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 9.19 — 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 9.16. 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 9.19. We assume that the grammar G in Chomsky Normal Form as in Theorem 9.16. 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". ■

9.6.1 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 9.20 — 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 9.20 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 $s \Rightarrow$ ""$|s\sigma_0|\cdots|s\sigma_{k-1}$. Note that Theorem 9.20 and Theorem 9.19 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 9.20 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++ programs, as defined in Definition 7.12. 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. $\star$

Proof of Theorem 9.20. 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 8.4. Let P be an input program for $HALTONZERO$. We will use the notion of *configurations* of a NAND++ program, as defined in Definition 7.12. Recall that a configuration of a NAND++ program P and input x captures the full state of P (contents of all the variables) at some iteration 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 P on the empty input is some fixed string.
- A *halting configuration* will have the value of the variable `loop` (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 $NEXT_P(\sigma)$ as the configuration at the next step. $NEXT_P(\sigma)$ is a string that agrees with σ on all but a constant number of coordinates (those encoding the position corresponding to the variable `i` and the two adjacent ones). On those coordinates, the value of $NEXT_P(\sigma)$ can be computed by some finite function.

We will let the alphabet $\Sigma = \{0,1\} \cup \{\|, \#\}$. A *computation history* of P 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.)[15]

[15] Reversing the odd-numbered block is a technical trick to help with making the function $INVALID_P$ we'll define below context free.

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

$$INVALID_P(L) = \begin{cases} 0 & L \text{ is a valid computation history of } P \text{ on } 0 \\ 1 & \text{otherwise} \end{cases} \tag{9.6}$$

We will show the following claim:

CLAIM: $INVALID_P$ is context-free.

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

We now turn to the proof of the claim. We will not show all the details, but the main point $INVALID_P(L) = 1$ if 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 this format can be done using a regular expression, and 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 $NEXT_P$ function only modifies its input in a constant number of places. We leave filling out the details as an exercise to the reader. Since $INVALID_P(L) = 1$ 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. ■

9.7 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.

Model	**Halting**	**Emptiness**	**Equivalence**
Regular Expressions	Decidable	Decidable	Decidable
Context Free Grammars	Decidable	Decidable	Undecidable
Turing complete models	Undecidable	Undecidable	Undecidable

Unrestricted Grammars (optional) The reason we call context free grammars "context free" is because if we have a rule of the form $v \mapsto a$ it means that we can always replace v with the string a, no matter the *context* in which v appears. More generally, we might want to consider cases where our replacement rules depend on the context.

This gives rise to the notion of *general 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 - these general grammars are Turing complete and hence their halting problem is undecidable.

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.

9.8 EXERCISES

Disclaimer Most of the exercises have been written in the summer of 2018 and haven't yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

9.9 BIBLIOGRAPHICAL NOTES

16

[16] TODO: Add letter of Christopher Strachey to the editor of The Computer Journal. Explain right order of historical achievements. Talk about intuitionistic, logicist, and formalist approaches for the foudnations of mathematics. Perhaps analogy to veganism. State the full Rice's Theorem and say that it follows from the same proof as in the exercise.

9.10 FURTHER EXPLORATIONS

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

9.11 ACKNOWLEDGEMENTS