Functions with Infinite domains, Automata, and Regular expressions

“An algorithm is a finite answer to an infinite number of questions.”, Attributed to Stephen Kleene.

The model of Boolean circuits (or equivalently, the NAND-CIRC programming language) has one very significant drawback: a Boolean circuit can only compute a finite function \( f \). In particular, since every gate has two inputs, a size \( s \) circuit can compute on an input of length at most \( 2^s \). Thus this model does not capture our intuitive notion of an algorithm as a single recipe to compute a potentially infinite function. For example, the standard elementary school multiplication algorithm is a single algorithm that multiplies numbers of all lengths. However, we cannot express this algorithm as a single circuit, but rather need a different circuit (or equivalently, a NAND-CIRC program) for every input length (see Fig. 6.1).

In this chapter, we extend our definition of computational tasks to consider functions with the unbounded domain of \( \{0, 1\}^* \). We focus on the question of defining what tasks to compute, mostly leaving the question of how to compute them to later chapters, where we will see Turing machines and other computational models for computing on unbounded inputs. However, we will see one example of a simple restricted model of computation - deterministic finite automata (DFAs).

This chapter: A non-mathy overview

In this chapter, we discuss functions that take as input strings of arbitrary length. We will often focus on the special case of Boolean functions, where the output is a single bit. These are still infinite functions since their inputs have unbounded

Learning Objectives:
- Define functions on unbounded length inputs, that cannot be described by a finite size table of inputs and outputs.
- Equivalence with the task of deciding membership in a language.
- Deterministic finite automatons (optional): A simple example for a model for unbounded computation.
- Equivalence with regular expressions.
6.1 FUNCTIONS WITH INPUTS OF UNBOUNDED LENGTH

Up until now, we considered the computational task of mapping some string of length \( n \) into a string of length \( m \). However, in general, computational tasks can involve inputs of unbounded length. For example, the following Python function computes the function \( \text{XOR} : \{0, 1\}^* \rightarrow \{0, 1\} \), where \( \text{XOR}(x) \) equals 1 iff the number of 1’s in \( x \) is odd. (In other words, \( \text{XOR}(x) = \sum_{i=0}^{\lfloor |x|/2 \rfloor} x_i \mod 2 \) for every \( x \in \{0, 1\}^* \).) As simple as it is, the \( \text{XOR} \) function cannot be computed by a Boolean circuit. Rather, for every \( n \), we can compute \( \text{XOR}_n \) (the restriction of \( \text{XOR} \) to \( \{0, 1\}^n \)) using a different circuit (e.g., see Fig. 6.2).

```python
def XOR(X):
    '''Takes list X of 0's and 1's
    Outputs 1 if the number of 1's is odd and outputs 0 otherwise'''
    result = 0
    for i in range(len(X)):
        result = (result + X[i]) % 2
    return result
```

Previously in this book, we studied the computation of finite functions \( f : \{0, 1\}^n \rightarrow \{0, 1\}^m \). Such a function \( f \) can always be described by listing all the \( 2^n \) values it takes on inputs \( x \in \{0, 1\}^n \). In this chapter, we consider functions such as \( \text{XOR} \) that take inputs of unbounded size. While we can describe \( \text{XOR} \) using a finite number of symbols (in fact, we just did so above), it takes infinitely many possible inputs, and so we cannot just write down all of its values. The same is true for many other functions capturing important computational tasks, including addition, multiplication, sorting, finding paths in
graphs, fitting curves to points, and so on. To contrast with the finite case, we will sometimes call a function $F : \{0, 1\}^* \to \{0, 1\}$ (or $F : \{0, 1\}^* \to \{0, 1\}^*$) infinite. However, this does not mean that $F$ takes as input strings of infinite length! It just means that $F$ can take as input a string of that can be arbitrarily long, and so we cannot simply write down a table of all the outputs of $F$ on different inputs.

**Big Idea 8** A function $F : \{0, 1\}^* \to \{0, 1\}^*$ specifies the computational task mapping an input $x \in \{0, 1\}^*$ into the output $F(x)$.

As we have seen before, restricting attention to functions that use binary strings as inputs and outputs does not detract from our generality, since other objects, including numbers, lists, matrices, images, videos, and more, can be encoded as binary strings.

As before, it is essential to differentiate between specification and implementation. For example, consider the following function:

$$TWINP(x) = \begin{cases} 1 & \exists p \in \mathbb{N} \text{ s.t. } p, p + 2 \text{ are primes and } p > |x| \\ 0 & \text{otherwise} \end{cases}$$

This is a mathematically well-defined function. For every $x$, $TWINP(x)$ has a unique value which is either 0 or 1. However, at the moment, no one knows of a Python program that computes this function. The Twin prime conjecture posits that for every $n$ there exists $p > n$ such that both $p$ and $p + 2$ are primes. If this conjecture is true, then $T$ is easy to compute indeed - the program

```python
def T(x):
    return 1
```

will do the trick. However, mathematicians have tried unsuccessfully to prove this conjecture since 1849. That said, whether or not we know how to implement the function $TWINP$, the definition above provides its specification.

6.1.1 Varying inputs and outputs

Many of the functions that interest us take more than one input. For example, the function

$$MULT(x, y) = x \cdot y$$

takes the binary representation of a pair of integers $x, y \in \mathbb{N}$, and outputs the binary representation of their product $x \cdot y$. However, since we can represent a pair of strings as a single string, we will consider functions such as $MULT$ as mapping $\{0, 1\}^*$ to $\{0, 1\}^*$. We will typically not be concerned with low-level details such as the precise way to represent a pair of integers as a string, since virtually all choices will be equivalent for our purposes.
Another example of a function we want to compute is

$$PALINDROME(x) = \begin{cases} 1 & \forall i \in |x|: x_i = x_{|x|-i} \\ 0 & \text{otherwise} \end{cases}$$

$PALINDROME$ has a single bit as output. Functions with a single bit of output are known as Boolean functions. Boolean functions are central to the theory of computation, and we will discuss them often in this book. Note that even though Boolean functions have a single bit of output, their input can be of arbitrary length. Thus they are still infinite functions that cannot be described via a finite table of values.

“Booleanizing” functions. Sometimes it might be convenient to obtain a Boolean variant for a non-Boolean function. For example, the following is a Boolean variant of $MULT$.

$$BMULT(x, y, i) = \begin{cases} i^{th} \text{ bit of } x \cdot y & i < |x \cdot y| \\ 0 & \text{otherwise} \end{cases}$$

If we can compute $BMULT$ via any programming language such as Python, C, Java, etc., we can compute $MULT$ as well, and vice versa.

**Solved Exercise 6.1 — Booleanizing general functions.** Show that for every function $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$, there exists a Boolean function $BF : \{0, 1\}^* \rightarrow \{0, 1\}$ such that a Python program to compute $BF$ can be transformed into a program to compute $F$ and vice versa.

**Solution:**

For every $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$, we can define

$$BF(x, i, b) = \begin{cases} F(x)_i & i < |F(x)|, b = 0 \\ 1 & i < |F(x)|, b = 1 \\ 0 & i \geq |F(x)| \end{cases}$$

to be the function that on input $x \in \{0, 1\}^*, i \in \mathbb{N}, b \in \{0, 1\}$ outputs the $i^{th}$ bit of $F(x)$ if $b = 0$ and $i < |F(x)|$. If $b = 1$, then $BF(x, i, b)$ outputs 1 iff $i < |F(x)|$ and hence this allows to compute the length of $F(x)$.

Computing $BF$ from $F$ is straightforward. For the other direction, given a Python function $BF$ that computes $BF$, we can compute $F$ as follows:

```python
def F(x):
    res = []
    i = 0
    while BF(x, i, 1):
```
6.1.2 Formal Languages

For every Boolean function \( F : \{0, 1\}^* \rightarrow \{0, 1\} \), we can define the set \( L_F = \{ x | F(x) = 1 \} \) of strings on which \( F \) outputs 1. Such sets are known as languages. This name is rooted in formal language theory as pursued by linguists such as Noam Chomsky. A formal language is a subset \( L \subseteq \{0, 1\}^* \) (or more generally \( L \subseteq \Sigma^* \) for some finite alphabet \( \Sigma \)). The membership or decision problem for a language \( L \), is the task of determining, given \( x \in \{0, 1\}^* \), whether or not \( x \in L \). If we can compute the function \( F \), then we can decide membership in the language \( L_F \) and vice versa. Hence, many texts such as [Sip97] refer to the task of computing a Boolean function as “deciding a language”. In this book, we mostly describe computational tasks using the function notation, which is easier to generalize to computation with more than one bit of output. However, since the language terminology is so popular in the literature, we will sometimes mention it.

6.1.3 Restrictions of functions

If \( F : \{0, 1\}^* \rightarrow \{0, 1\} \) is a Boolean function and \( n \in \mathbb{N} \) then the restriction of \( F \) to inputs of length \( n \), denoted as \( F_n \), is the finite function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \) such that \( f(x) = F(x) \) for every \( x \in \{0, 1\}^n \). That is, \( F_n \) is the finite function that is only defined on inputs in \( \{0, 1\}^n \), but agrees with \( F \) on those inputs. Since \( F_n \) is a finite function, it can be computed by a Boolean circuit, implying the following theorem:

**Theorem 6.1 — Circuit collection for every infinite function.** Let \( F : \{0, 1\}^* \rightarrow \{0, 1\} \). Then there is a collection \( \{ C_n \}_{n \in \{1, 2, \ldots\}} \) of circuits such that for every \( n > 0 \), \( C_n \) computes the restriction \( F_n \) of \( F \) to inputs of length \( n \).

\[
\text{Proof.} \quad \text{This is an immediate corollary of the universality of Boolean circuits. Indeed, since } F_n \text{ maps } \{0, 1\}^n \text{ to } \{0, 1\}, \text{ Theorem 4.15 implies that there exists a Boolean circuit } C_n \text{ to compute it. In fact, the size of this circuit is at most } c \cdot 2^n/n \text{ gates for some constant } c \leq 10.
\]

In particular, Theorem 6.1 implies that there exists such a circuit collection \( \{ C_n \} \) even for the TWINP function we described before, even though we do not know of any program to compute it. Indeed, this is not that surprising: for every particular \( n \in \mathbb{N} \), TWINP\(_n\) is either the constant zero function or the constant one function, both of which
can be computed by very simple Boolean circuits. Hence a collection of circuits \( \{C_n\} \) that computes \( TWINP \) certainly exists. The difficulty in computing \( TWINP \) using Python or any other programming language arises from the fact that we do not know for each particular \( n \) what is the circuit \( C_n \) in this collection.

6.2 DETERMINISTIC FINITE AUTOMATA (OPTIONAL)

All our computational models so far - Boolean circuits and straight-line programs - were only applicable for finite functions.

In Chapter 7, we will present Turing machines, which are the central models of computation for unbounded input length functions. However, in this section we present the more basic model of deterministic finite automata (DFA). Automata can serve as a good stepping-stone for Turing machines, though they will not be used much in later parts of this book, and so the reader can feel free to skip ahead to Chapter 7.

DFAs turn out to be equivalent in power to regular expressions: a powerful mechanism to specify patterns, which is widely used in practice. Our treatment of automata is relatively brief. There are plenty of resources that help you get more comfortable with DFAs. 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).

At a high level, an algorithm is a recipe for computing an output from an input via a combination of the following steps:

1. Read a bit from the input
2. Update the state (working memory)
3. Stop and produce an output

For example, recall the Python program that computes the XOR function:

```python
def XOR(X):
    '''Takes list X of 0's and 1's
    Outputs 1 if the number of 1's is odd and outputs 0 otherwise'''
    result = 0
    for i in range(len(X)):
        result = (result + X[i]) % 2
    return result
```

In each step, this program reads a single bit \( X[i] \) and updates its state \( \text{result} \) based on that bit (flipping \( \text{result} \) if \( X[i] \) is 1 and keeping it the same otherwise). When it is done transversing the input,
functions with infinite domains, automata, and regular expressions

the program outputs result. In computer science, such a program is called a single-pass constant-memory algorithm since it makes a single pass over the input and its working memory is finite. (Indeed, in this case, result can either be 0 or 1.) Such an algorithm is also known as a Deterministic Finite Automaton or DFA (another name for DFAs is a finite state machine). 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 $\sigma$ and its prior state. The output of the machine is based on the final state. Every single-pass constant-memory algorithm corresponds to such a machine. If an algorithm uses $c$ bits of memory, then the contents of its memory can be represented as a string of length $c$. Therefore such an algorithm can be in one of at most $2^c$ states at any point in the execution.

We can specify a DFA of $C$ states by a list of $C \cdot 2$ rules. Each rule will be of the form "If the DFA is in state $v$ and the bit read from the input is $\sigma$ then the new state is $v'$". At the end of the computation, we will also have a rule of the form "If the final state is one of the following ... then output 1, otherwise output 0". For example, the Python program above can be represented by a two-state automaton for computing XOR of the following form:

- Initialize in the state 0.
- For every state $s \in \{0, 1\}$ and input bit $\sigma$ read, if $\sigma = 1$ then change to state $1 - s$, otherwise stay in state $s$.
- At the end output 1 iff $s = 1$.

We can also describe a $C$-state DFA as a labeled graph of $C$ vertices. For every state $s$ and bit $\sigma$, we add a directed edge labeled with $\sigma$ between $s$ and the state $s'$ such that if the DFA is at state $s$ and reads $\sigma$ then it transitions to $s'$. (If the state stays the same then this edge will be a self-loop; similarly, if $s$ transitions to $s'$ in both the case $\sigma = 0$ and $\sigma = 1$ then the graph will contain two parallel edges.) We also label the set $S$ of states on which the automaton will output 1 at the end of the computation. This set is known as the set of accepting states. See Fig. 6.3 for the graphical representation of the XOR automaton.

Formally, a DFA is specified by (1) the table of the $C \cdot 2$ rules, which can be represented as a transition function $T$ that maps a state $s \in [C]$ and bit $\sigma \in \{0, 1\}$ to the state $s' \in [C]$ which the DFA will transition to from state $s$ on input $\sigma$ and (2) the set $S$ of accepting states. This leads to the following definition.

Figure 6.3: A deterministic finite automaton that computes the XOR function. It has two states 0 and 1, and when it observes $\sigma$ it transitions from $v$ to $v \oplus \sigma$. 
**Definition 6.2 — Deterministic Finite Automaton.** A deterministic finite automaton (DFA) with \( C \) states over \( \{0, 1\} \) is a pair \((T, S)\) with 
\[
T : [C] \times \{0, 1\} \rightarrow [C] \text{ and } S \subseteq [C].
\]
The finite function \( T \) is known as the *transition function* of the DFA. The set \( S \) is known as the set of *accepting states*.

Let \( F : \{0, 1\}^* \rightarrow \{0, 1\} \) be a Boolean function with the infinite domain \( \{0, 1\}^* \). We say that \((T, S)\) 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 \( s_0 = 0 \) and \( s_{i+1} = T(s_i, x_i) \) for every \( i \in [n] \), then 
\[
s_n \in S \iff F(x) = 1.
\]

**Remark 6.3 — Definitions in other texts.** Deterministic finite automata can be defined in several equivalent ways. In particular Sipser [Sip97] defines a DFA as a five-tuple \((Q, \Sigma, \delta, q_0, F)\) where \( Q \) is the set of states, \( \Sigma \) is the alphabet, \( \delta \) 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 \( \Sigma \) is equal to \( \{0, 1\} \).

**Solved Exercise 6.2 — DFA for \((010)^*\).** Prove that there is a DFA that computes the following function \( F \):

\[
F(x) = \begin{cases} 
1 & \text{3 divides } |x| \text{ and } \forall i \in [x/3] : x_{3i} x_{3i+1} x_{3i+2} = 010 \\
0 & \text{otherwise}
\end{cases}
\]

**Solution:**
When asked to construct a deterministic finite automaton, it is often useful to start by constructing a single-pass constant-memory
algorithm using a more general formalism (for example, using pseudocode or a Python program). Once we have such an algorithm, we can mechanically translate it into a DFA. Here is a simple Python program for computing $F$:

```python
def F(X):
    '''Return 1 iff X is a concatenation of zero/more copies of [0,1,0]'''
    if len(X) % 3 != 0:
        return False
    ultimate = 0
    penultimate = 1
    antepenultimate = 0
    for idx, b in enumerate(X):
        antepenultimate = penultimate
        penultimate = ultimate
        ultimate = b
        if idx % 3 == 2 and ((antepenultimate, penultimate, ultimate) != (0,1,0)):
            return False
    return True
```

Since we keep three Boolean variables, the working memory can be in one of $2^3 = 8$ configurations, and so the program above can be directly translated into an 8 state DFA. While this is not needed to solve the question, by examining the resulting DFA, we can see that we can merge some states and obtain a 4 state automaton, described in Fig. 6.4. See also Fig. 6.5, which depicts the execution of this DFA on a particular input.

### 6.2.1 Anatomy of an automaton (finite vs. unbounded)

Now that we are considering computational tasks with unbounded input sizes, it is crucial to distinguish between the components of our algorithm that have fixed length and the components that grow with the input size. For the case of DFAs these are the following:

**Constant size components:** Given a DFA $A$, the following quantities are fixed independent of the input size:

- The number of states $|C|$ in $A$.
- The transition function $T$ (which has $2|C|$ inputs, and so can be specified by a table of $2|C|$ rows, each entry in which is a number in $[C]$).
- The set $\mathcal{S} \subseteq [C]$ of accepting states. This set can be described by a string in $\{0,1\}^C$ specifying which states are in $\mathcal{S}$ and which are not.

![Figure 6.4: A DFA that outputs 1 only on inputs $x \in \{0,1\}^*$ that are a concatenation of zero or more copies of 010. The state 0 is both the starting state and the only accepting state. The table denotes the transition function of $T$, which maps the current state and symbol read to the new symbol.](image)
Together the above means that we can fully describe an automaton using finitely many symbols. This is a property we require out of any notion of “algorithm”: we should be able to write down a complete specification of how it produces an output from an input.

**Components of unbounded size:** The following quantities relating to a DFA are not bounded by any constant. We stress that these are still finite for any given input.

- The length of the input \( x \in \{0, 1\}^* \) that the DFA is provided. The input length is always finite, but not a priori bounded.
- The number of steps that the DFA takes can grow with the length of the input. Indeed, a DFA makes a single pass on the input and so it takes precisely \(|x|\) steps on an input \( x \in \{0, 1\}^* \).

![Figure 6.5: Execution of the DFA of Fig. 6.4. The number of states and the transition function size are bounded, but the input can be arbitrarily long. If the DFA is at state \( s \) and observes the value \( \sigma \), then it moves to the state \( T(s, \sigma) \). At the end of the execution, the DFA accepts if the final state is in \( S \).](image)

**6.2.2 DFA-computable functions**

We say that a function \( F : \{0, 1\}^* \rightarrow \{0, 1\} \) is **DFA computable** if there exists some DFA that computes \( F \). In Chapter 4 we saw that every finite function is computable by some Boolean circuit. Thus, at this point, you might expect that every infinite function is computable by some DFA. However, this is very much not the case. We will soon see some simple examples of infinite functions that are not computable by DFAs, but for starters, let us prove that such functions exist.

**Theorem 6.4 — DFA-computable functions are countable.** Let \( DFACOMP \) be the set of all Boolean functions \( F : \{0, 1\}^* \rightarrow \{0, 1\} \) such that there exists a DFA computing \( F \). Then \( DFACOMP \) is countable.

**Proof Idea:**
Every DFA can be described by a finite length string, which yields an onto map from \( \{0, 1\}^* \) to DFACOMP: namely, the function that maps a string describing an automaton \( A \) to the function that it computes.

\[ \text{Proof of Theorem 6.4.} \] Every DFA can be described by a finite string, representing the transition function \( T \) and the set of accepting states, and every DFA \( A \) computes some function \( F : \{0, 1\}^* \rightarrow \{0, 1\} \). Thus we can define the following function \( \text{StDC} : \{0, 1\}^* \rightarrow \text{DFACOMP} \):

\[
\text{StDC}(a) = \begin{cases} 
F & \text{a represents automaton } A \text{ and } F \text{ is the function } A \text{ computes} \\
\text{ONE} & \text{otherwise}
\end{cases}
\]

where \( \text{ONE} : \{0, 1\}^* \rightarrow \{0, 1\} \) is the constant function that outputs 1 on all inputs (and is a member of DFACOMP). Since by definition, every function \( F \) in DFACOMP is computable by some automaton, \( \text{StDC} \) is an onto function from \( \{0, 1\}^* \) to DFACOMP, which means that DFACOMP is countable (see Section 2.4.2).

\[ \]

Since the set of all Boolean functions is uncountable, we get the following corollary:

**Theorem 6.5 — Existence of DFA-uncomputable functions.** There exists a Boolean function \( F : \{0, 1\}^* \rightarrow \{0, 1\} \) that is not computable by any DFA.

**Proof.** If every Boolean function \( F \) is computable by some DFA, then DFACOMP equals the set ALL of all Boolean functions, but by Theorem 2.12, the latter set is uncountable, contradicting Theorem 6.4.

\[ \]

### 6.3 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 a programming language such as Python, and the system returns all \( x \in X \) 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
$P$! 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 queries can be evaluated very efficiently on huge files
and in particular 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 manipulation of text files, then you
have probably come across regular expressions.

A regular expression over some alphabet $\Sigma$ is obtained by combin-
ing elements of $\Sigma$ 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 oper-
ations on top of $|$ and *, but these operations can be implemented as
“syntactic sugar” using the operators $|$ and $*$.) For example, the fol-
lowing 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:

$$
(00(0^*)|11(1^*))^*.
$$

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):

$$
(a|b|c|d)(a|b|c|d)^*(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*.
$$

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

**Definition 6.6 — Regular expression.** A regular expression $e$ over an
alphabet $\Sigma$ is a string over $\Sigma \cup \{\epsilon, |, *, \emptyset, "\}$ 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 parentheses 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 parentheses when they can be inferred from the context. We also use the convention that OR and concatenation are left-associative, and we give highest 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 \( \Phi_e : \Sigma^* \to \{0, 1\} \) 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?).

**Definition 6.7 — Matching a regular expression.** Let \( e \) be a regular expression over the alphabet \( \Sigma \). The function \( \Phi_e : \Sigma^* \to \{0, 1\} \) is defined as follows:

1. If \( e = \sigma \) then \( \Phi_e(x) = 1 \) iff \( x = \sigma \).
2. If \( e = (e'|e'') \) then \( \Phi_e(x) = \Phi_{e'}(x) \lor \Phi_{e''}(x) \) where \( \lor \) 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 is some \( k \in \mathbb{N} \) 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 $\Sigma$ matches a string $x \in \Sigma^*$ if $\Phi_e(x) = 1$.

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.

A Boolean function is called “regular” if it outputs 1 on precisely the set of strings that are matched by some regular expression. That is,

**Definition 6.8 — Regular functions / languages.** Let $\Sigma$ be a finite set and $F : \Sigma^* \to \{0, 1\}$ be a Boolean function. We say that $F$ is regular if $F = \Phi_e$ for some regular expression $e$.

Similarly, for every formal language $L \subseteq \Sigma^*$, we say that $L$ is regular if and only if there is a regular expression $e$ such that $x \in L$ iff $e$ matches $x$.

**Example 6.9 — 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

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

is the expression we saw in (6.1).

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 $\Sigma$, but as usual, we will mostly 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.

6.3.1 Algorithms for matching regular expressions

Regular expressions would not be very useful for search if we could not evaluate, given a regular expression $e$, whether a string $x$ is matched by $e$. Luckily, there is an algorithm to do so. Specifically, there is an algorithm (think “Python program” though later we will formalize the notion of algorithms using Turing machines) that on input a regular expression $e$ over the alphabet $\{0, 1\}$ and a string $x \in \{0, 1\}^*$, outputs 1 iff $e$ matches $x$ (i.e., outputs $\Phi_e(x)$).

Indeed, Definition 6.7 actually specifies a recursive algorithm for computing $\Phi_e$. 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.
We assume above that we have a procedure \texttt{MatchEmpty} that on input a regular expression \(e\) outputs 1 if and only if \(e\) matches the empty string \(\texttt{"\text{"}}\).

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 \(e\). Eventually (when they have size 1) then they must correspond to the non-recursive case of a single alphabet symbol. Correspondingly, the recursive calls made in Algorithm 6.10 always correspond to a shorter expression or (in the case of an expression of the form \((e')^*\) a shorter input string. Thus, we can prove the correctness of Algorithm 6.10 on inputs of the form \((e, x)\) by induction over \(\min\{|e|, |x|\}\). The base case is when either \(x = \texttt{"\text{"}}\) or \(e\) is a single alphabet symbol, \(\texttt{"\text{"}}\) or \(\emptyset\). In the case the expression is of the form \(e = (e'|e'')\) or \(e = (e')^*\), we make recursive calls with the shorter expressions \(e', e''\). In the case the expression is of the form \(e = (e')^*\), we make recursive calls with either a shorter string.
functions with infinite domains, automata, and regular expressions

\[ x \text{ and the same expression, or with the shorter expression } e' \text{ and a string } x' \text{ that is equal in length or shorter than } x. \]

**Solved Exercise 6.3 — Match the empty string.** Give an algorithm that on input a regular expression \( e \), outputs 1 if and only if \( \Phi_e(\epsilon) = 1 \).

**Solution:**

We can obtain such a recursive algorithm by using the following observations:

1. An expression of the form \( \epsilon \) or \( (e')^* \) always matches the empty string.
2. An expression of the form \( \sigma \), where \( \sigma \in \Sigma \) is an alphabet symbol, never matches the empty string.
3. The regular expression \( \emptyset \) does not match the empty string.
4. An expression of the form \( e'|e'' \) matches the empty string if and only if one of \( e' \) or \( e'' \) matches it.
5. An expression of the form \( (e')(e'') \) matches the empty string if and only if both \( e' \) and \( e'' \) match it.

Given the above observations, we see that the following algorithm will check if \( e \) matches the empty string:

```plaintext
Algorithm 6.11 — Check for empty string.
Input: Regular expression \( e \) over \( \Sigma^* \), \( x \in \Sigma^* \)
Output: 1 iff \( e \) matches the empty string.
1: procedure MatchEmpty(e)
2: if \( e = \epsilon \) then return 1 ;
3: if \( e = \emptyset \) or \( e \in \Sigma \) then return 0 ;
4: if \( e = (e'|e'') \) then return MatchEmpty(e') or MatchEmpty(e'') ;
5: if \( e = (e')(e'') \) then return MatchEmpty(e') and MatchEmpty(e'') ;
6: if \( e = (e')^* \) then return 1 ;
7: end procedure
```

**6.4 EFFICIENT MATCHING OF REGULAR EXPRESSIONS (OPTIONAL)**

Algorithm 6.10 is not very efficient. For example, given an expression involving concatenation or the “star” operation and a string of length
It can be shown that in the worst case Algorithm 6.10 can take time exponential in the length of the input string \( x \). Fortunately, 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. Rather we will use the colloquial notion of \( O(n) \) running time as used in introduction to programming courses and whiteboard coding interviews. We will see a formal definition of time complexity in Chapter 13.

**Theorem 6.12** — Matching regular expressions in linear time. Let \( e \) be a regular expression. Then there is an \( O(n) \) time algorithm that computes \( \Phi_e \).

The implicit constant in the \( O(n) \) term of Theorem 6.12 depends on the expression \( e \). Thus, another way to state Theorem 6.12 is that for every expression \( e \), there is some constant \( c \) and an algorithm \( A \) that computes \( \Phi_e \) 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 6.12 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.

We prove Theorem 6.12 by obtaining 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 \( x_0, \ldots, x_{n-2} \). This will result in an expression for the running time of the form \( T(n) = T(n - 1) + O(1) \) which solves to \( T(n) = O(n) \).

**Restrictions of regular expressions.** The central definition for the algorithm behind Theorem 6.12 is the notion of a restriction of a regular expression. The idea is that for every regular expression \( e \) and symbol \( \sigma \) in its alphabet, it is possible to define a regular expression \( e[\sigma] \) 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 \((01)^*101\) (i.e., one or more occurrences of 01) then \( e[1] \) is equal to \((01)^*0 \) and \( e[0] \) will be \( \emptyset \). (Can you see why?)

Algorithm 6.13 computes the restriction \( e[\sigma] \) given a regular expression \( e \) and an alphabet symbol \( \sigma \). It always terminates, since the recursive calls it makes are always on expressions smaller than the input expression. Its correctness can be proven by induction on the length of the regular expression \( e \), with the base cases being when \( e \) is "", \( \emptyset \), or a single alphabet symbol \( \tau \).
functions with infinite domains, automata, and regular expressions

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

By the definition of a restriction, for every \( \sigma \in \Sigma \) and \( x' \in \Sigma^* \), the expression \( e \) matches \( x' \sigma \) if and only if \( e[\sigma] \) matches \( x' \). Hence for every \( e \) and \( x \in \Sigma^n \), \( \Phi_e(x_0 \cdots x_{n-2}) = \Phi_e(x) \) and Algorithm 6.14 does return the correct answer. The only remaining task is to analyze its running time. Note that Algorithm 6.14 uses the MatchEmpty procedure of Solved Exercise 6.3 in the base case that \( x = "" \). However, this is OK since this procedure’s running time depends only on \( e \) and is independent of the length of the original input.

For simplicity, let us restrict our attention to the case that the alphabet \( \Sigma \) is equal to \( \{0, 1\} \). Define \( C(\ell) \) to be the maximum number of operations that Algorithm 6.13 takes when given as input a regular expression \( e \) over \( \{0, 1\} \) of at most \( \ell \) symbols. The value \( C(\ell) \) can be shown to be polynomial in \( \ell \), though this is not important for this theorem, since we only care about the dependence of the time to compute
Algorithm 6.14 is a recursive algorithm that input an expression \( e \) and a string \( x \in \{0, 1\}^n \), does computation of at most \( C(|e|) \) steps and then calls itself with 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, the running time \( T(e, n) \) that it takes for Algorithm 6.14 to compute \( \Phi_e \) for inputs of length \( n \) satisfies the recursive equation:

\[
T(e, n) = \max\{T(e[0], n - 1), T(e[1], n - 1)\} + C(|e|) \tag{6.2}
\]

(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. (6.2), let us open up the recursion for one level, writing \( T(e, n) \) as

\[
T(e, n) = \max\{T(e[0][0], n - 2) + C(|e[0]|), T(e[1][0], n - 2) + C(|e[1]|), T(e[1][1], n - 2) + C(|e[1]|)\} + C(|e|) .
\]

Continuing this way, we can see that \( T(e, n) \leq n \cdot C(L) + O(1) \) 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 6.14 runs in \( O(n) \) time:

Claim: Let \( e \) be a regular expression over \( \{0, 1\} \), then there is a number \( L(e) \in \mathbb{N} \), such that for every sequence of symbols \( \alpha_0, \ldots, \alpha_{n-1} \), if we define \( e' = e[\alpha_0][\alpha_1] \cdots [\alpha_{n-1}] \) (i.e., restricting \( e \) to \( \alpha_0 \), and then \( \alpha_1 \) and so on and so forth), then \( |e'| \leq L(e) \).

Proof of claim: For a regular expression \( e \) over \( \{0, 1\} \) and \( \alpha \in \{0, 1\}^m \), we denote by \( e[\alpha] \) the expression \( e[\alpha_0][\alpha_1] \cdots [\alpha_m] \) obtained by restricting \( e \) to \( \alpha_0 \) and then to \( \alpha_1 \) 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 \( L(e) \) 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 \( (e')'[\alpha] \) 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 \( \alpha \) will either have the form \( e'e''[\alpha] \) or the form \( e''[\alpha][e'[\alpha']] \) where \( \alpha' \) is some string such that \( \alpha = \alpha'\alpha'' \) and \( e''[\alpha'] \)
matches the empty string. Since \( e^\prime[\alpha] \in S(e^\prime) \) and \( e^\prime[\alpha'] \in S(e') \), the number of the possible distinct expressions of the form \( e[\alpha] \) is at most \( |S(e^\prime)| + |S(e^\prime)| \cdot |S(e')| \). This completes the proof of the claim.

The bottom line is that while running Algorithm 6.14 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 6.14 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 \( O \) notation can (and will) depend on \( e \) but crucially, not on the length of the input \( x \).

6.4.1 Matching regular expressions using DFAs

Theorem 6.12 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. Of course, as we have seen before, a single-pass constant-memory algorithm is simply a deterministic finite automaton. As we will see in Theorem 6.17, a function can be computed by regular expression if and only if it can be computed by a DFA. We start with showing the “only if” direction:

**Theorem 6.15 — 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 single-pass constant-memory for checking if a string matches a regular expression is presented in Algorithm 6.16. The idea is to replace the recursive algorithm of Algorithm 6.14 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.
Algorithm 6.16 — Regular expression matching by a DFA.

**Input:** Regular expression $e$ over $\Sigma^*$, $x \in \Sigma^n$ where $n \in \mathbb{N}$

**Output:** $\Phi_e(x)$

1: procedure DFAMatch($e, x$)
2: Let $S \leftarrow S(e)$ be the set $\{e[\alpha]|\alpha \in \{0, 1\}^*\}$ as defined in the proof of the linear-time matching theorem.
3: for $e' \in S$ do
4: Let $v_{e'} \leftarrow 1$ if $\Phi_{e'}(\epsilon) = 1$ and $v_{e'} \leftarrow 0$ otherwise
5: end for
6: for $i \in [n]$ do
7: Let $\text{last}_{e'} \leftarrow v_{e'}$ for all $e' \in S$
8: Let $v_{e'} \leftarrow \text{last}_{e' \mid x_i}$ for all $e' \in S$
9: end for
10: return $v_e$
11: end procedure

**Proof of Theorem 6.15.** Algorithm 6.16 checks if a given string $x \in \Sigma^*$ is matched by the regular expression $e$. For every regular expression $e$, this algorithm has a constant number of Boolean variables (specifically a variable $v_{e'}$ for every $e' \in S(e)$ and a variable $\text{last}_{e'}$ for every $e' \in S(e)$), using the fact that $e'[x_i]$ is in $S(e)$ for every $e' \in S(e)$. It makes a single pass over the input string. Hence it corresponds to a DFA. We prove its correctness by induction on the length $n$ of the input. Specifically, we will argue that before reading $x_i$, the variable $v_{e'}$ is equal to $\Phi_{e'}(x_0 \cdots x_{i-1})$ for every $e' \in S(e)$. In the case $i = 0$ this holds since we initialize $v_{e'} = \Phi_{e'}(\epsilon)$ for all $e' \in S(e)$. For $i > 0$ this holds by induction since the inductive hypothesis implies that $\text{last}_{e'} = \Phi_{e'}(x_0 \cdots x_{i-2})$ for all $e' \in S(e)$ and by the definition of the set $S(e')$, for every $e' \in S(e)$ and $x_{i-1} \in \Sigma$, $e'' = e'[x_{i-1}]$ is in $S(e)$ and $\Phi_{e''}(x_0 \cdots x_{i-1}) = \Phi_{e'}(x_0 \cdots x_i)$.

6.4.2 Equivalence of regular expressions and automata

Recall that a Boolean function $F : \{0, 1\}^* \rightarrow \{0, 1\}$ is defined to be regular if it is equal to $\Phi_e$ for some regular expression $e$. (Equivalently, a language $L \subseteq \{0, 1\}^*$ is defined to be regular if there is a regular expression $e$ such that $e$ matches $x$ if and only if $x \in L$.) The following theorem is the central result of automata theory:

**Theorem 6.17 — DFA and regular expression equivalency.** Let $F : \{0, 1\}^* \rightarrow \{0, 1\}$. Then $F$ is regular if and only if there exists a DFA $(T, S)$ that computes $F$.

**Proof Idea:**
One direction follows from Theorem 6.15, which shows that for every regular expression $e$, the function $\Phi_e$ can be computed by a DFA (see for example Fig. 6.6). For the other direction, we show that given a DFA $(T, S)$ for every $v, w \in [C]$ we can find a regular expression that would match $x \in \{0, 1\}^*$ if and only if the DFA starting in state $v$, will end up in state $w$ after reading $x$.

*Proof of Theorem 6.17.* Since Theorem 6.15 proves the "only if" direction, we only need to show the "if" direction. Let $A = (T, S)$ 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 6.2, $F(x)$ is equal to the OR of $F_{v,w}(x)$ for every $w \in S$.

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.

To give regular expressions for the functions $F_{v,w}$, we start by defining the following functions $F_{v,w}^t$: for every $v, w \in [C]$ and $0 \leq t \leq C$, $F_{v,w}^t(x) = 1$ if and only if starting from $v$ and observing $x$, the automaton reaches $w$ with all intermediate states being in the set $[t] = \{0, \ldots, t - 1\}$ (see Fig. 6.7). That is, while $v, w$ themselves might be outside $[t]$, $F_{v,w}^t(x) = 1$ 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 $F_{v,w}^0(x) = 1$ 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 $F_{v,w}^t = F_{v,w}$.

We will prove the theorem by induction on $t$, showing that $F_{v,w}^t$ is regular for every $v, w$ and $t$. For the base case of $t = 0$, $F_{v,w}^0$ is regular for every $v, w$ since it can be described as one of the expressions "", $\emptyset$, $0$, $1$ or $0|1$. Specifically, if $v = w$ then $F_{v,w}^0(x) = 1$ if and only if $x$ is the empty string. If $v \neq w$ then $F_{v,w}^0(x) = 1$ if and only if $x$ consists of a single symbol $\sigma \in \{0, 1\}$ and $T(v, \sigma) = w$. Therefore in this case $F_{v,w}^0$ 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 us prove the general case by induction. Assume, via the induction hypothesis, that for every $v', w' \in [C]$, we have a regular expression $R_{v',w'}^t$ that computes $F_{v',w'}^t$. We need to prove that $F_{v,w}^{t+1}$ 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 $k-1$ times 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. 6.8.)

Therefore we can compute $F^t_{v,w}$ using the regular expression

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

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

---

**Figure 6.8:** 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 $R^t_{v,t}(R^t_{t,t})^* R^t_{t,w}$.

6.4.3 **Closure properties of regular expressions**

If $F$ and $G$ are regular functions computed by the expressions $e$ and $f$ respectively, then the expression $e|f$ computes the function $H = F \lor G$ defined as $H(x) = F(x) \lor G(x)$. Another way to say this is that the set of regular functions is closed under the OR operation. That is, if $F$ and $G$ are regular then so is $F \lor G$. An important corollary of **Theorem 6.17** is that this set is also closed under the NOT operation:
Lemma 6.18 — Regular expressions closed under complement. If $F : \{0, 1\}^* \rightarrow \{0, 1\}$ 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 6.12 it can be computed by a DFA $A$. But we can then construct a DFA $\overline{A}$ which does the same computation but flips the set of accepted states. The DFA $\overline{A}$ will compute $\overline{F}$. By Theorem 6.17 this implies that $\overline{F}$ is regular as well. □

Since $a \land b = \overline{\overline{a} \lor \overline{b}}$, Lemma 6.18 implies that the set of regular functions is closed under the AND operation as well. Moreover, since OR, NOT and AND are a universal basis, this set is also closed under NAND, XOR, and any other finite function. That is, we have the following corollary:

**Theorem 6.19 — Closure of regular expressions.** Let $f : \{0, 1\}^k \rightarrow \{0, 1\}$ be any finite Boolean function, and let $F_0, \ldots, F_{k-1} : \{0, 1\}^* \rightarrow \{0, 1\}$ be regular functions. Then the function $G(x) = f(F_0(x), F_1(x), \ldots, F_{k-1}(x))$ is regular.

**Proof.** This is a direct consequence of the closure of regular functions under OR and NOT (and hence AND), combined with Theorem 4.13, that states that every $f$ can be computed by a Boolean circuit (which is simply a combination of the AND, OR, and NOT operations). □

### 6.5 Limitations of Regular Expressions and the Pumping Lemma

The efficiency of regular expression matching makes them very useful. This is why operating systems and text editors often restrict their search interface to regular expressions and do not allow searching by specifying an arbitrary function. However, this efficiency comes at a cost. As we have seen, regular expressions cannot compute every function. In fact, there are some very simple (and useful!) functions that they cannot compute. Here is one example:

**Lemma 6.20 — Matching parentheses.** Let $\Sigma = \{\langle, \rangle\}$ and $MATCHPAREN : \Sigma^* \rightarrow \{0, 1\}$ be the function that given a string of parentheses, outputs 1 if and only if every opening parenthesis is matched by a corresponding closed one. Then there is no regular expression over $\Sigma$ that computes MATCHPAREN.

**Lemma 6.20** is a consequence of the following result, which is known as the **pumping lemma:**
Theorem 6.21 — Pumping Lemma. Let \( e \) be a regular expression over some alphabet \( \Sigma \). Then there is some number \( n_0 \) such that for every \( w \in \Sigma^* \) 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 6.9: 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 is 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 non-empty 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. 6.9.

\[ \star \]

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 6.21. To prove the lemma formally, we use induction on the length of the expression. Like all induction proofs, this will 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 the empty string \( \varepsilon \). In all these cases the conditions of the lemma are satisfied simply because \( n_0 = 2 \), and there exists no string \( x \) of length larger than \( n_0 \) that is matched by the expression.

We now prove the inductive step. Let \( \varepsilon \) 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 \( \varepsilon \) has more than one symbol, it has one of the forms \( (a) \varepsilon \varepsilon', (b) \varepsilon' \varepsilon', (c) \varepsilon'\varepsilon'\varepsilon' \) where in all these cases the subexpressions \( \varepsilon' \) and \( \varepsilon'' \) have fewer symbols than \( \varepsilon \) and hence satisfy the induction hypothesis.

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

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

In case \( (c) \), if \( w \) is matched by \( (\varepsilon')^* \) then \( w = w_0 \ldots w_l \) where for every \( i \in [l], w_i \) is a nonempty string matched by \( \varepsilon' \). If \( |w_0| > 2|\varepsilon'| \), 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 \( z = w_1 \ldots w_l \) then \( |xy| \leq n_0 \) and \( xy^kz \) is matched by \( (\varepsilon')^* \) for every \( k \in \mathbb{N} \).
Remark 6.22 — 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 step that says that if $a'$, $a''$, $a'''$ etc have property $P$ then so is an object $a$ that is obtained by composing them.

Using the pumping lemma, we can easily prove Lemma 6.20 (i.e., the non-regularity of the “matching parenthesis” function):

Proof of Lemma 6.20. Suppose, towards the sake of contradiction, that there is an expression $e$ such that $\Phi_e = MATCHPAREN$. Let $n_0$ be the number obtained from Theorem 6.21 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 Theorem 6.21, 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 pumping lemma conditions. To understand the pumping lemma, it is crucial to follow the order of quantifiers in Theorem 6.21. In particular, the number $n_0$ in the statement of Theorem 6.21 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 6.4 — 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
Exercise: Let $F : \{0,1\}^* \to \{0,1\}$ defined such that $F(x) = 1$ iff $x = 0^n1^n$ 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 $w = 0^n1^n$. Notice that $F(w) = 1$. What is the partition $w = xyz$ from the pumping lemma?”

“Since $|xy| \leq n_0$ and $|y| \geq 1$, I guess I am forced to use $x = 0^a$, $y = 0^b$, $z = 0^{n_0-a-b}1^{n_0}$ for $b \geq 1$ and $a \leq n_0 - b$”

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

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 $|xy| \leq n_0$ and $|y| \geq 1$ such that for every $k \in \mathbb{N}$ it holds that $F(xy^kz) = 1$.

Figure 6.10: 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 $\exists$ quantifier corresponds to an object you are free to choose on your own (and base your choice on previously chosen objects). Every $\forall$ 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 $F(xy^kz) = 0$, hence violating the conclusion of the pumping lemma.
much difference, as one can easily encode the separator as a special binary string instead.

Solution:

We use the pumping lemma. Suppose toward the sake of contradiction that there is a regular expression $e$ computing $\text{PAL}$, and let $n_0$ be the number obtained by the pumping lemma (Theorem 6.21). Consider the string $w = 0^n_0; 0^n_0$. Since the reverse of the all zero string is the all zero string, $\text{PAL}(w) = 1$. Now, by the pumping lemma, if $\text{PAL}$ is computed by $e$, then we can write $w = xyz$ such that $|xy| \leq n_0, |y| \geq 1$ and $\text{PAL}(xy^kz) = 1$ for every $k \in \mathbb{N}$. In particular, it must hold that $\text{PAL}(xz) = 1$, 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. 6.10 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 $x = 0^n 1^n$ 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).

6.6 Answering Semantic Questions about Regular Expressions

Regular expressions have applications beyond search. For example, regular expressions are often used to define tokens (such as what is a valid variable identifier, or keyword) in the design of parsers, compilers and interpreters for programming languages. Regular expressions have other applications too: for example, in recent years, the world of networking moved from fixed topologies to “software defined networks”. Such networks are routed 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”. To represent such policies we need a language that is on one hand sufficiently expressive to capture the policies we want to implement, but on the other hand sufficiently restrictive so that we can quickly execute them at network speed and also be able to answer questions such as “can C see the packets moved from A to B?”. The NetKAT network programming language uses a variant of regular expressions to achieve precisely that. For this application, it is important that we are not merely able to answer whether an expression $e$ matches a string $x$ but also answer semantic questions about regular expressions such as “do expressions
functions with infinite domains, automata, and regular expressions 247

\(e\) and \(e'\) compute the same function?" and "does there exist a string \(x\) that is matched by the expression \(e\)?". The following theorem shows that we can answer the latter question:

**Theorem 6.23 — Emptiness of regular languages is computable.** There is an algorithm that given a regular expression \(e\), outputs 1 if and only if \(\Phi_e\) 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 6.23.* 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 \(\sigma\) 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.

Using Theorem 6.23, we can obtain an algorithm that determines whether or not two regular expressions \(e\) and \(e'\) are equivalent, in the sense that they compute the same function.

**Theorem 6.24 — 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 there is an algorithm that computes \(REGEQ\).

**Proof Idea:**
The idea is to show that given a pair of regular expressions \(e\) 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 6.24. We will prove Theorem 6.24 from Theorem 6.23. (The two theorems are in fact equivalent: it is easy to prove Theorem 6.23 from Theorem 6.24, 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 bit $a, b \in \{0, 1\}$, $a \neq b$ if and only if $(a \land b) \lor (\neg a \land b)$.

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

$$\Phi_{e''}(x) = (\Phi_e(x) \land \Phi_{e'}(x)) \lor (\neg \Phi_e(x) \land \Phi_{e'}(x)).$$

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

Specifically, by Lemma 6.18, regular functions are closed under negation, which means that for every regular expression $e$, there is an expression $\neg e$ such that $\Phi_{\neg e}(x) = 1 - \Phi_e(x)$ for every $x \in \{0, 1\}^*$. Now, for every two expressions $e$ and $e'$, the expression

$$e \land e' = (\neg e \lor \neg e')$$

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 (6.3) such that $e''$ is empty if and only if $e$ and $e'$ are equivalent.

Chapter Recap

- We model computational tasks on arbitrarily large inputs using infinite functions $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$.
- Such functions take an arbitrarily long (but still finite!) string as input, and cannot be described by a finite table of inputs and outputs.
- A function with a single bit of output is known as a Boolean function, and the task of computing it is equivalent to deciding a language $L \subseteq \{0, 1\}^*$. 
• **Deterministic finite automata** (DFAs) are one simple model for computing (infinite) Boolean functions.
• There are some functions that cannot be computed by DFAs.
• The set of functions computable by DFAs is the same as the set of languages that can be recognized by regular expressions.

### 6.7 EXERCISES

**Exercise 6.1** — **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. $H(x) = F(x) \lor G(x)$.
2. $H(x) = F(x) \land G(x)$
3. $H(x) = \text{NAND}(F(x), G(x))$.
4. $H(x) = F(x^R)$ where $x^R$ is the reverse of $x$: $x^R = x_{n-1}x_{n-2}\cdots x_0$ 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 6.2** 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 $\sum_{i=0}^{\lfloor |x|/4 \rfloor} x_i$ and $F(x) = 0$ otherwise.
- $G(x) = 1$ if and only if $\sum_{i=0}^{\lfloor |x|/4 \rfloor} x_i \geq |x|/4$ and $G(x) = 0$ otherwise.

**Exercise 6.3** — **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 $F : \{0, 1\}^* \to \{0, 1\}$ is not regular.
   For every $x \in \{0, 1\}^*$, $F(x) = 1$ iff $\sum_j x_j = 3^i$ for some $i > 0$.

6.8 BIBLIOGRAPHICAL NOTES

The relation of regular expressions with finite automata is a beautiful topic, on which we only touch upon in this text. It is covered more extensively in [Sip97; HMU14; Koz97]. These texts also discuss topics such as non-deterministic finite automata (NFA) and the relation between context-free grammars and pushdown automata.

The automaton of Fig. 6.4 was generated using the FSM simulator of Ivan Zuzak and Vedrana Jankovic. Our proof of Theorem 6.12 is closely related to the Myhill-Nerode Theorem. One direction of the Myhill-Nerode 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 $0 \leq i \neq j < k$. 