4.2-5 Define a function f from regular expressions to natural numbers:

$$\begin{aligned} f(\mathtt{a}) &= 1 \\ f(\mathtt{b}) &= 0 \\ f(\Lambda) &= 0 \\ f((r_1) \cup (r_2)) &= \min(f(r_1), f(r_2)) \\ f((r_1)(r_2)) &= f(r_1) + f(r_2) \\ f((r)^*) &= 0. \end{aligned}$$

Prove that $f(r)$ is equal to the smallest number of a's in any string generated by r, i.e.,

$$f(r) = \min_{x \in L(r)} \#_{\mathtt{a}}(x).$$

4.2-6 Give a recursive algorithm to determine whether the language generated by a regular expression is empty, finite but nonempty, or infinite.

Solution: Knowing whether r_1 and r_2 are empty, finite, or infinite is enough to determine whether $r_1 r_2$ and $r_1 \cup r_2$ are empty, finite, or infinite. However, knowing whether r is empty, finite, or infinite is not enought to determine whether r^* is finite. Why? Because $\{\Lambda\}$ and $\{\mathtt{a}\}$ are finite languages, but $\{\Lambda\}^*$ is finite whereas $\{\mathtt{a}\}^*$ is infinite. Thus we will also need to determine whether $L(r) = \{\Lambda\}$.

Fix an alphabet Σ. Let

$$f(r) = \begin{cases} \emptyset & \text{if } L(r) = \emptyset, \\ \Lambda & \text{if } L(r) = \{\Lambda\}, \\ F & \text{if } L(r) \text{ is any other finite language,} \\ I & \text{if } L(r) \text{ is infinite.} \end{cases}$$

Then $L(r)$ is empty if $f(r) = \emptyset$, finite but nonempty if $f(r) = \Lambda$ or $f(r) = F$, infinite if $f(r) = I$. We compute $f(\cdot)$ by a recursive algorithm whose structure mimics the definition of $L(\cdot)$.

$$\begin{aligned} f(\emptyset) &= \emptyset, \\ f(\Lambda) &= \Lambda, \\ f(c) &= F \qquad \text{for each } c \in \Sigma, \\ f((r_1) \cup (r_2)) &= \max(f(r_1), f(r_2)) \qquad \text{using the order } \emptyset < \Lambda < F < I, \\ f((r_1)(r_2)) &= \max(f(r_1), f(r_2)) \qquad \text{using the order } \Lambda < F < I < \emptyset, \\ f((r)^*) &= \begin{cases} \Lambda & \text{if } f(r) = \emptyset \text{ or } f(r) = \Lambda, \\ I & \text{if } f(r) = F \text{ or } f(r) = I. \end{cases} \end{aligned}$$

4.2-7 How would you represent a regular expression as a string?

Solution: Suppose that the regular expression's alphabet is Σ. When we write down the regular expression, we use the characters in Σ and the special characters $\emptyset$, Λ, $\cup$, *, (, and). That is, we represent the regular expression as a string over the alphabet Σ', where Σ' consists of the characters in Σ and the six special characters mentioned above.

*4.3 REGULAR EXPRESSIONS IN THE REAL WORLD: EGREP

(This section is optional; it is intended only for students with access to Unix®.) If you have used the Unix operating system, you may have used the program fgrep, grep, or egrep[1] in order to search for a string or a more complex pattern in a file. Regular expressions provide a convenient, compact way of expressing patterns. This section describes the use of egrep. (The internal workings of egrep are based on finite machines; although we will

[1] The name "egrep" is an acronym for "extended global regular expression print."

FROM: FLOYD + BEIGEL, THE LANGUAGE OF THE MACHINE, COMPUTER SCIENCE PRESS, NY, 1994.

develop some of the relevant theory later in this chapter, the algorithmic details are beyond the scope of this book.)

We say that a string s *matches* a regular expression r if s belongs to the language generated by r, i.e., if $s \in L(r)$. Given a regular expression and a file name, egrep will print out every line (of the file) that contains a substring that matches the regular expression. The command format is

`grep '`*regexp*`'` *file*

where *file* is a file name and *regexp* is a regular expression whose format we will describe below. (The single quotes around the regular expression are not strictly necessary, but they prevent undesired preprocessing by the Unix command shell.)

EXAMPLE 4.3. The command

```
% grep 'depend' /usr/dict/words
```

searches the dictionary for all words containing **depend** as a substring. The result:

```
depend
dependent
independent
```

■■■

Because of the limitations of standard keyboards, we must write regular expressions slightly differently when using egrep (Table 4.1). The Kleene-closure operator (*) is denoted by * (not a superscript), union ($\cup$) is denoted by a vertical bar (|), and concatenation is denoted by adjacency in the ordinary way. Parentheses have their usual meaning as grouping operators. The alphabet (Σ) is denoted by a period (.), i.e., . matches any character.

For example, the regular expression $(ab \cup c)^*$ would be denoted `(ab|c)*`, and Σ^* would be denoted `.*`.

There is no way to express the empty string in egrep; e.g., `()` is not permitted. Instead, r? denotes zero or one occurrences of the regular expression r, so the empty string is not needed in practice.

Egrep understands other special characters, of which we will mention only two more. Imagine that a line starts with an invisible character, which we call *beginning-of-line*, and ends with an invisible character, which we call

end-of-line. These are not standard characters, though they are loosely related to the carriage return/line feed pairs that separate lines of text in real files. In egrep, beginning-of-line is denoted by ^, and end-of-line is denoted by $. Thus ^$r$ matches an entire line of text rather than a substring, though ^.*r.*$ is equivalent to r.

EXAMPLE 4.4. To find words of two or more letters that begin and end with y, we could type

```
% egrep '^y.*y$' /usr/dict/words
yeasty
yeomanry
yesterday
```

In contrast, the command

```
% egrep 'y.*y' /usr/dict/words
```

Expression	*Egrep Notation*
r^*	r*
r^+	r+
$r \cup \Lambda$	r?
$r \cup s$	r|s
rs	rs
(r)	(r)
Σ	.
c	c
d	\d
beginning-of-line	^
end-of-line	$

TABLE 4.1: Egrep notation (r and s denote nonempty regular expressions; c denotes an ordinary character; d denotes a special character).

results in a list of 113 words, going all the way from `alleyway` to `yesteryear`. ■■■

EXAMPLE 4.5. For help with spelling we might type

```
% egrep '^rec(ei|ie)ve$' /usr/dict/words
receive
```
■■■

EXAMPLE 4.6. For help with a crossword puzzle

s			u		t			e

we might type

```
% egrep '^s..u.t..e$' /usr/dict/words
seductive
structure
```
■■■

If one wants to specify the character (rather than use it as a grouping operator, one types \(. A character with a special meaning can be quoted by preceding it with \.

EXAMPLE 4.7. To search for all occurrences of the string `f(x)` in a file called ch4.tex, we could type

```
% egrep 'f\(x\)' ch4.tex
string \verb:f(x): in a file called ch4.tex, we could type
```
■■■

EXAMPLE 4.8. Egrep is also useful in checking files for common grammatical errors. The following command looks for two consecutive occurrences of the word `the` separated by one or more spaces:

```
% egrep '(^| )the +the( |$)' myfile.txt
```
■■■

This short section explains only a few of the capabilities of egrep. For additional information consult your Unix documentation or type the Unix command

```
% man egrep
```


Exercises

4.3-1 A common grammatical error is to write the word a followed by a word beginning with a vowel. Write a Unix command using egrep that checks myfile.txt for this error, assuming that the error occurs inside a single line.

Solution:

```
egrep '(^| )(a|A) +(a|e|i|o|u)' myfile.txt
```

4.3-2 Write a command using egrep that searches myfile.txt for all lines that contain the letter a and the letter b.

Solution:

```
egrep 'a.*b|b.*a' myfile.txt
```

4.3-3 Write a command using egrep that searches myfile.txt for all lines that contain at least two y's or at least two z's.

4.3-4 Write a command using egrep that searches myfile.txt for all lines that contain the string Bob and do not start with %. Hint: [^%] matches each character other than %.

4.4 KLEENE'S THEOREM

In our diagrams, we have represented programs as labeled digraphs. This metaphor for programs is very powerful. In this section, we present a fundamental theorem about directed graphs. Using the relationship between digraphs and programs, we then show that every NFA language is a regular language.

Recall that the first element of a path is called its origin, and the last element is called its destination. A path of length 0, such as $\langle\langle v \rangle\rangle$, will be identified with the vertex v. A path of length 1, such as $\langle\langle v_0, v_1 \rangle\rangle$, will be identified with the edge (v_0, v_1).

The *catenation* operation (denoted $\vec{\otimes}$) on paths is very similar to the *con*catenation operation on sequences. If one path's destination is another path's origin, then the catenation operation joins the two paths at that vertex; if one path ends at a different vertex from where the other path starts, then it