Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition. Daniel Jurafsky & James H. Martin.

2 REGULAR EXPRESSIONS AND AUTOMATA

User$_1$: *Men are all alike.*
ELIZA$_1$: *IN WHAT WAY*
User$_2$: *They're always bugging us about something or other.*
ELIZA$_2$: *CAN YOU THINK OF A SPECIFIC EXAMPLE*
User$_3$: *Well, my boyfriend made me come here.*
ELIZA$_3$: *YOUR BOYFRIEND MADE YOU COME HERE*
User$_4$: *He says I'm depressed much of the time.*
ELIZA$_4$: *I AM SORRY TO HEAR YOU ARE DEPRESSED.*

Weizenbaum (1966)

Imagine that you have become a passionate fan of woodchucks. Desiring more information on this celebrated woodland creature, you turn to your favorite Web browser and type in *woodchuck*. Your browser returns a few sites. You have a flash of inspiration and type in *woodchucks*. This time you discover "interesting links to woodchucks and lemurs" and "all about Vermont's unique, endangered species". Instead of having to do this search twice, you would have rather typed one search command specifying something like *woodchuck with an optional final s*. Or perhaps you might want to search for all the prices in some document; you might want to see all strings that look like *$199* or *$25* or *$24.99*. In this chapter we introduce the **regular expression**, the standard notation for characterizing text sequences. The regular expression is used for specifying text strings in situations like this Web-search example, and in other information retrieval applications, but also plays an important role in word-processing, computation of frequencies from corpora, and other such tasks.

After we have defined regular expressions, we show how they can be implemented via the **finite-state automaton**. The finite-state automaton is not only the mathematical device used to implement regular expressions, but also one of the most significant tools of computational linguistics. Variations of automata such as finite-state transducers, Hidden Markov Models, and *N*-gram grammars are important components of applications that we will introduce in later chapters, including speech recognition and synthesis, machine translation, spell-checking, and information-extraction.

2.1 Regular Expressions

SIR ANDREW: Her C's, her U's and her T's: why that?
Shakespeare, *Twelfth Night*

REGULAR EXPRESSION

One of the unsung successes in standardization in computer science has been the **regular expression** (**RE**), a language for specifying text search strings. The regular expression languages used for searching texts in UNIX (vi, Perl, Emacs, grep), Microsoft Word (version 6 and beyond), and WordPerfect are almost identical, and many RE features exist in the various Web search engines. Besides this practical use, the regular expression is an important theoretical tool throughout computer science and linguistics.

STRINGS

A regular expression (first developed by Kleene (1956) but see the History section for more details) is a formula in a special language that is used for specifying simple classes of **strings**. A string is a sequence of symbols; for the purpose of most text-based search techniques, a string is any sequence of alphanumeric characters (letters, numbers, spaces, tabs, and punctuation). For these purposes a space is just a character like any other, and we represent it with the symbol ␣.

Formally, a regular expression is an algebraic notation for characterizing a set of strings. Thus they can be used to specify search strings as well as to define a language in a formal way. We will begin by talking about regular expressions as a way of specifying searches in texts, and proceed to other uses. Section 2.3 shows that the use of just three regular expression operators is sufficient to characterize strings, but we use the more convenient and commonly-used regular expression syntax of the Perl language throughout this section. Since common text-processing programs agree on most of the syntax of regular expressions, most of what we say extends to all UNIX, Microsoft Word, and WordPerfect regular expressions. Appendix A shows the few areas where these programs differ from the Perl syntax.

CORPUS

Regular expression search requires a **pattern** that we want to search for, and a **corpus** of texts to search through. A regular expression search function will search through the corpus returning all texts that contain the pattern. In an information retrieval (IR) system such as a Web search engine, the texts might be entire documents or Web pages. In a word-processor, the texts might be individual words, or lines of a document. In the rest of this chapter, we will use this last paradigm. Thus when we give a search pattern, we will assume that the search engine returns the *line of the document* returned. This is what the UNIX `grep` command does. We will underline the exact part of the pattern that matches the regular expression. A search can be designed to return all matches to a regular expression or only the first match. We will show only the first match.

2.1.1 Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters. For example, to search for *woodchuck*, we type `/woodchuck/`. So the regular expression `/Buttercup/` matches any string containing the substring *Buttercup*, for example the line *I'm called little Buttercup*) (recall that we are assuming a search application that returns entire lines). From here on we will put slashes around each regular expression to make it clear what is a regular expression and what is a pattern. We use the slash since this is the notation used by Perl, but the slashes are *not* part of the regular expressions.

The search string can consist of a single character (like `/!/`) or a sequence of characters (like `/urgl/`); The *first* instance of each match to the regular expression is underlined below (although a given application might choose to return more than just the first instance):

RE	Example Patterns Matched
`/woodchucks/`	“interesting links to woodchucks and lemurs”
`/a/`	“Mary Ann stopped by Mona’s”
`/Claire␣says,/`	“Dagmar, my gift please,” Claire says,”
`/song/`	“all our pretty songs”
`/!/`	“You’ve left the burglar behind again!” said Nori

Regular expressions are **case sensitive**; lowercase `/s/` is distinct from uppercase `/S/` (`/s/` matches a lower case *s* but not an uppercase *S*). This means that the pattern `/woodchucks/` will not match the string *Woodchucks*. We can solve this problem with the use of the square braces `[` and `]`. The string of characters inside the braces specify a **disjunction** of characters to match. For example Fig. 2.1 shows that the pattern `/[wW]/` matches patterns containing either *w* or *W*.

RE	Match	Example Patterns
`/[wW]oodchuck/`	Woodchuck or woodchuck	“Woodchuck”
`/[abc]/`	‘a’, ‘b’, *or* ‘c’	“In uomini, in soldati”
`/[1234567890]/`	any digit	“plenty of 7 to 5”

Figure 2.1 The use of the brackets `[]` to specify a disjunction of characters.

The regular expression `/[1234567890]/` specified any single digit. While classes of characters like digits or letters are important building blocks in expressions, they can get awkward (e.g., it’s inconvenient to specify

`/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/`

RANGE

to mean “any capital letter”). In these cases the brackets can be used with the dash (`-`) to specify any one character in a **range**. The pattern `/[2-5]/` specifies any one of the characters *2*, *3*, *4*, or *5*. The pattern `/[b-g]/` specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples:

RE	Match	Example Patterns Matched
`/[A-Z]/`	an uppercase letter	“we should call it ‘Drenched Blossoms’”
`/[a-z]/`	a lowercase letter	“my beans were impatient to be hoed!”
`/[0-9]/`	a single digit	“Chapter 1: Down the Rabbit Hole”

Figure 2.2 The use of the brackets `[]` plus the dash `-` to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret `ˆ`. If the caret `ˆ` is the first symbol after the open square brace `[`, the resulting pattern is negated. For example, the pattern `/[^a]/` matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Fig. 2.3 shows some examples.

RE	Match (single characters)	Example Patterns Matched
`[^A-Z]`	not an uppercase letter	"Oyfn pripetchik"
`[^Ss]`	neither 'S' nor 's'	"I have no exquisite reason for't"
`[^\.]`	not a period	"our resident Djinn"
`[e^]`	either 'e' or '^'	"look up ^ now"
`a^b`	the pattern 'a^b'	"look up a^ b now"

Figure 2.3 Uses of the caret ^ for negation or just to mean ^ .

The use of square braces solves our capitalization problem for *woodchucks*. But we still haven't answered our original question; how do we specify both *woodchuck* and *woodchucks*? We can't use the square brackets, because while they allow us to say "s or S", they don't allow us to say "s or nothing". For this we use the question-mark `/?/`, which means "the preceding character or nothing", as shown in Fig. 2.4.

RE	Match	Example Patterns Matched
`woodchucks?`	woodchuck or woodchucks	"woodchuck"
`colou?r`	color or colour	"colour"

Figure 2.4 The question-mark `?` marks optionality of the previous expression.

We can think of the question-mark as meaning "zero or one instances of the previous character". That is, it's a way of specifying how many of something that we want. So far we haven't needed to specify that we want more than one of something. But sometimes we need regular expressions that allow repetitions of things. For example, consider the language of (certain) sheep, which consists of strings that look like the following:

baa!
baaa!
baaaa!
baaaaa!
baaaaaa!
...

KLEENE *

This language consists of strings with a *b*, followed by at least two *a*s, followed by an exclamation point. The set of operators that allow us to say things like "some number of *a*s" are based on the asterisk or `*`, commonly called the **Kleene *** (pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So `/a*/` means "any string of zero or more *a*s". This will match *a* or *aaaaaa* but it will also match *Off Minor*, since the string *Off Minor* has zero *a*s. So the regular expression for matching one or more *a* is `/aa*/`, meaning one *a* followed by zero or more *a*s. More complex patterns can also be repeated. So `/[ab]*/` means “zero or more *a*s or *b*s” (not “zero or more right square braces”). This will match strings like *aaaa* or *ababab* or *bbbb*.

We now know enough to specify part of our regular expression for prices: multiple digits. Recall that the regular expression for an individual digit was `/[0-9]/`. So the regular expression for an integer (a string of digits) is `/[0-9][0-9]*/`. (Why isn’t it just `/[0-9]*/`?)

KLEENE +

Sometimes it’s annoying to have to write the regular expression for digits twice, so there is a shorter way to specify “at least one” of some character. This is the **Kleene +**, which means “one or more of the previous character”. Thus the expression `/[0-9]+/` is the normal way to specify “a sequence of digits”. There are thus two ways to specify the sheep language: `/baaa*!/` or `/baa+!/`.

One very important special character is the period (`/./`), a **wildcard** expression that matches any single character (*except* a carriage return):

RE	Match	Example Patterns
`/beg.n/`	any character between *beg* and *n*	begin, beg’n, begun

Figure 2.5 The use of the period `.` to specify any character.

The wildcard is often used together with the Kleene star to mean “any string of characters”. For example suppose we want to find any line in which a particular word, for example *aardvark*, appears twice. We can specify this with the regular expression `/aardvark.*aardvark/`.

ANCHORS

Anchors are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret `^` and the dollar-sign `$`. The caret `^` matches the start of a line. The pattern `/^The/` matches the word *The* only at the start of a line. Thus there are three uses of the caret `^`: to match the start of a line, as a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow Perl to know which function a given caret is supposed to have?) The dollar sign `$` matches the end of a line. So the pattern `␣$` is a useful pattern for matching a space at the end of a line, and `/^The dog\.$/` matches a line that contains only the phrase *The dog.* (We have to use the backslash here since we want the `.` to mean “period” and not the wildcard.)

There are also two other anchors: `\b` matches a word boundary, while `\B` matches a non-boundary. Thus `/\bthe\b/` matches the word *the* but not the word *other*. More technically, Perl defines a word as any sequence of digits, underscores or letters; this is based on the definition of “words” in programming languages like Perl or C. For example, `/\b99\b/` will match the string *99* in *There are 99 bottles of beer on the wall* (because 99 follows a space) but not *99* in *There are 299 bottles of beer on the wall* (since 99 follows a number). But it will match *99* in *$99* (since *99* follows a dollar sign ($), which is not a digit, underscore, or letter).

2.1.2 Disjunction, Grouping, and Precedence

DISJUNCTION

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case we might want to search for either the string *cat* or the string *dog*. Since we can't use the square-brackets to search for "cat or dog" (why not?) we need a new operator, the **disjunction** operator, also called the **pipe** symbol `|`. The pattern `/cat|dog/` matches either the string `cat` or the string `dog`.

PRECEDENCE

Sometimes we need to use this disjunction operator in the midst of a larger sequence. For example, suppose I want to search for information about pet fish for my cousin David. How can I specify both *guppy* and *guppies*? We cannot simply say `/guppy|ies/`, because that would match only the strings *guppy* and *ies*. This is because sequences like `guppy` take **precedence** over the disjunction operator `|`. In order to make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators `(` and `)`. Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe `|` and the Kleene`*`. So the pattern `/gupp(y|ies)/` would specify that we meant the disjunction only to apply to the suffixes `y` and `ies`.

The parenthesis operator `(` is also useful when we are using counters like the Kleene`*`. Unlike the `|` operator, the Kleene`*` operator applies by default only to a single character, not a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form *Column 1 Column 2 Column 3*. The expression `/Column␣[0-9]+␣*/` will not match any column; instead, it will match a column followed by any number of spaces! The star here applies only to the space ␣ that precedes it, not the whole sequence. With the parentheses, we could write the expression `/(Column␣[0-9]+␣*)*/` to match the word *Column*, followed by a number and optional spaces, the whole pattern repeated any number of times.

OPERATOR PRECEDENCE

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the **operator precedence hierarchy** for regular expressions. The following table gives the order of RE operator precedence, from highest precedence to lowest precedence:

Parenthesis	`()`
Counters	`* + ? {}`
Sequences and anchors	`the ^my end$`
Disjunction	`|`

Thus, because counters have a higher precedence than sequences, `/the*/` matches *theeeee* but not *thethe*. Because sequences have a higher precedence than disjunction, `/the|any/` matches *the* or *any* but not *theny*.

GREEDY

Patterns can be ambiguous in another way. Consider the expression `/[a-z]*/` when matching against the text *once upon a time*. Since `/[a-z]*/` matches zero or more letters, this expression could match nothing, or just the first letter *o*, or *on*, or *onc*, or *once*. In these cases regular expressions always match the *largest* string they can; we say that patterns are **greedy**, expanding to cover as much of a string as they can.

2.1.3 A Simple Example

Suppose we wanted to write a RE to find cases of the English article *the*. A simple (but incorrect) pattern might be:

```
/the/
```

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

```
/[tT]he/
```

But we will still incorrectly return texts with `the` embedded in other words (e.g., *other* or *theology*). So we need to specify that we want instances with a word boundary on both sides:

```
/\b[tT]he\b/
```

Suppose we wanted to do this without the use of `/\b/`? We might want this since `/\b/` won't treat underscores and numbers as word boundaries; but we might want to find *the* in some context where it might also have underlines or numbers nearby (*the_* or *the25*). We need to specify that we want instances in which there are no alphabetic letters on either side of the *the*:

```
/[^a-zA-Z][tT]he[^a-zA-Z]/
```

But there is still one more problem with this pattern: it won't find the word *the* when it begins a line. This is because the regular expression `[^a-zA-Z]`, which we used to avoid embedded *the*s, implies that there must be some single (although non-alphabetic) character before the `the`. We can avoid this by specifying that before the *the* we require *either* the beginning-of-line or a non-alphabetic character, and the same at the end of the line:

```
/(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/
```

FALSE POSITIVES
FALSE NEGATIVES

The process we just went through was based on fixing two kinds of errors: **false positives**, strings that we incorrectly matched like *other* or *there*, and **false negatives**, strings that we incorrectly missed, like *The*. Addressing these two kinds of errors comes up again and again in building and improving speech and language processing systems. Reducing the error rate for an application thus involves two antagonistic efforts:

- Increasing **accuracy** (minimizing false positives)
- Increasing **coverage** (minimizing false negatives).

2.1.4 A More Complex Example

Let's try out a more significant example of the power of REs. Suppose we want to build an application to help a user buy a computer on the Web. The user might want "any PC with more than 500 MHz and 32 Gb of disk space for less than $1000". In order to do this kind of retrieval we will first need to be able to look for expressions like *500 MHz* or *32 Gb* or *Compaq* or *Mac* or *$999.99*. In the rest of this section we'll work out some simple regular expressions for this task.

First, let's complete our regular expression for prices. Here's a regular expression for a dollar sign followed by a string of digits. Note that Perl is smart enough to realize that `$` here doesn't mean end-of-line; how might it know that?

```
/$[0-9]+/
```

Now we just need to deal with fractions of dollars. We'll add a decimal point and two digits afterwards:

```
/$[0-9]+\.[0-9][0-9]/
```

This pattern only allows *$199.99* but not *$199*. We need to make the cents optional, and make sure we're at a word boundary:

```
/\b$[0-9]+(\.[0-9][0-9])?\b/
```

How about specifications for processor speed (in megahertz = MHz or gigahertz = GHz)? Here's a pattern for that:

```
/\b[0-9]+␣*(MHz|[Mm]egahertz|GHz|[Gg]igahertz)\b/
```

Note that we use `/␣*/` to mean "zero or more spaces", since there might always be extra spaces lying around. Dealing with disk space (in Gb = gigabytes), or memory size (in Mb = megabytes or Gb = gigabytes), we need to allow for optional gigabyte fractions again (*5.5 Gb*). Note the use of `?` for making the final `s` optional:

```
/\b[0-9]+␣*(Mb|[Mm]egabytes?)\b/
/\b[0-9](\.[0-9]+)?␣*(Gb|[Gg]igabytes?)\b/
```

Finally, we might want some simple patterns to specify operating systems and vendors:

```
/\b(Win95|Win98|WinNT|Windows␣*(NT|95|98|2000)?)\b/
/\b(Mac|Macintosh|Apple)\b/
```

2.1.5 Advanced Operators

RE	Expansion	Match	Example Patterns
`\d`	`[0-9]`	any digit	Party␣of␣5
`\D`	`[^0-9]`	any non-digit	Blue␣moon
`\w`	`[a-zA-Z0-9_]`	any alphanumeric or underscore	Daiyu
`\W`	`[^\w]`	a non-alphanumeric	!!!!
`\s`	`[␣\r\t\n\f]`	whitespace (space, tab)	
`\S`	`[^\s]`	Non-whitespace	in␣Concord

Figure 2.6 Aliases for common sets of characters.

There are also some useful advanced regular expression operators. Fig. 2.6 shows some useful aliases for common ranges, which can be used mainly to save typing. Besides the Kleene * and Kleene +, we can also use explicit numbers as counters, by enclosing them in curly brackets. The regular expression `/{3}/` means "exactly 3 occurrences of the previous character or expression". So `/a\.{24}z/` will match *a* followed by 24 dots followed by *z* (but not *a* followed by 23 or 25 dots followed by a *z*).

A range of numbers can also be specified; so `/{n,m}/` specifies from n to m occurrences of the previous char or expression, while `/{n,}/` means at least n occurrences of the previous expression. REs for counting are summarized in Figure 2.7.

RE	Match
`*`	zero or more occurrences of the previous char or expression
`+`	one or more occurrences of the previous char or expression
`?`	exactly zero or one occurrence of the previous char or expression
`{n}`	*n* occurrences of the previous char or expression
`{n,m}`	from *n* to *m* occurrences of the previous char or expression
`{n,}`	at least *n* occurrences of the previous char or expression

Figure 2.7 Regular expression operators for counting.

NEWLINE

Finally, certain special characters are referred to by special notation based on the backslash (`\`). The most common of these are the **newline** character `\n` and the **tab** character `\t`. To refer to characters that are special themselves (like `.`, `*`, `[`, and `\`), precede them with a backslash, (i.e., `/\./`, `/\*/`, `/\[/`, and `/\\/`).

RE	Match	Example Patterns Matched
`\*`	an asterisk "*"	"K*A*P*L*A*N"
`\.`	a period "."	"Dr. Livingston, I presume"
`\?`	a question mark	"Would you light my candle?"
`\n`	a newline	
`\t`	a tab	

Figure 2.8 Some characters that need to be backslashed.

The reader should consult Appendix A for further details of regular expressions, and especially for the differences between regular expressions in Perl, UNIX, and Microsoft Word.

2.1.6 Regular Expression Substitution, Memory, and ELIZA

SUBSTITUTION

An important use of regular expressions is in **substitutions**. For example, the Perl substitution operator `s/regexp1/pattern/` allows a string characterized by a regular expression to be replaced by another string:

```
s/colour/color/
```

It is often useful to be able to refer to a particular subpart of the string matching the first pattern. For example, suppose we wanted to put angle brackets around all integers in a text, changing e.g., *the 35 boxes* to *the <35> boxes*. We'd like a way to refer back to the integer we've found so that we can easily add the brackets. To do this, we put parentheses `(` and `)` around the first pattern, and use the **number** operator `\1` in the second pattern to refer back. Here's how it looks:

```
s/([0-9]+)/<\1>/
```

The parenthesis and number operators can also be used to specify that a certain string or expression must occur twice in the text. For example, suppose we are looking for the pattern "the Xer they were, the Xer they will be", where we want to constrain the two X's to be the same string. We do this by surrounding the first X with the parenthesis operator, and replacing the second X with the number operator `\1`, as follows:

```
/the (.*)er they were, the \1er they will be/
```

Here the `\1` will be replaced by whatever string matched the first item in parentheses. So this will match *The bigger they were, the bigger they will be* but not *The bigger they were, the faster they will be*.

The number operator can be used with other numbers: if you match two different sets of parenthesis, `\2` means whatever matched the *second* set. For example

```
/the (.*)er they (.*), the \1er they \2/
```

REGISTERS

will match *The bigger they were, the bigger they were* but not *The bigger they were, the bigger they will be*. These numbered memories are called **registers** (e.g. register 1, register 2, register 3, etc). This memory feature is not part of every regular expression language and is often considered an "extended" feature of regular expressions.

Substitutions using memory are very useful in implementing a simple natural-language understanding program like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulated a Rogerian psychologist and could carry on conversations with the user like the following:

User$_1$:	Men are all alike.
ELIZA$_1$:	IN WHAT WAY
User$_2$:	They're always bugging us about something or other.
ELIZA$_2$:	CAN YOU THINK OF A SPECIFIC EXAMPLE
User$_3$:	Well, my boyfriend made me come here.
ELIZA$_3$:	YOUR BOYFRIEND MADE YOU COME HERE
User$_4$:	He says I'm depressed much of the time.
ELIZA$_4$:	I AM SORRY TO HEAR YOU ARE DEPRESSED.

Eliza worked by having a cascade of regular expression substitutions that each matched some part of the input lines and changed them. The first substitutions changed all instances of *my* to *YOUR*, and *I'm* to *YOU ARE*, and so on. The next set of substitutions looked for relevant patterns in the input and created an appropriate output; here are some examples:

```
s/.* YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/.* YOU ARE (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
```

```
s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
```

Since multiple substitutions could apply to a given input, substitutions were assigned a rank and were applied in order. Creation of such patterns is addressed in Exercise 2.2.

2.2 FINITE-STATE AUTOMATA

FINITE-STATE AUTOMATON

FSA

REGULAR LANGUAGE

The regular expression is more than just a convenient metalanguage for text searching. First, a regular expression is one way of describing a **finite-state automaton** (**FSA**). Finite-state automata are the theoretical foundation of a good deal of the computational work we will describe in this book. Any regular expression can be implemented as a finite-state automaton (except regular expressions that use the memory feature; more on this later). Symmetrically, any finite-state automaton can be described with a regular expression. Second, a regular expression is one way of characterizing a particular kind of formal language called a **regular language**. Both regular expressions and finite-state automata can be used to describe regular languages. A third equivalent method of characterizing the regular languages, the **regular grammar**, will be introduced in Ch. 15. The relation among these four theoretical constructions is sketched out in Fig. 2.9.

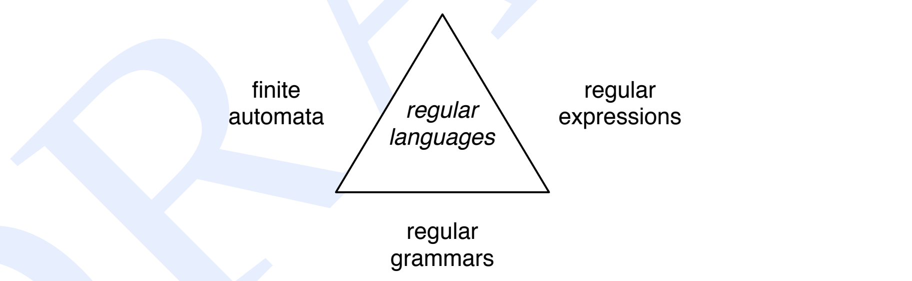


Figure 2.9 Finite automata, regular expressions, and regular grammars are all equivalent ways of describing regular languages.

This section will begin by introducing finite-state automata for some of the regular expressions from the last section, and then suggest how the mapping from regular expressions to automata proceeds in general. Although we begin with their use for implementing regular expressions, FSAs have a wide variety of other uses that we will explore in this chapter and the next.

2.2.1 Using an FSA to Recognize Sheeptalk

> *After a while, with the parrot's help, the Doctor got to learn the language of the animals so well that he could talk to them himself and understand everything they said.*

Hugh Lofting, *The Story of Doctor Dolittle*

Let's begin with the "sheep language" we discussed previously. Recall that we defined the sheep language as any string from the following (infinite) set:

baa!
baaa!
baaaa!
baaaaa!
baaaaaa!
...

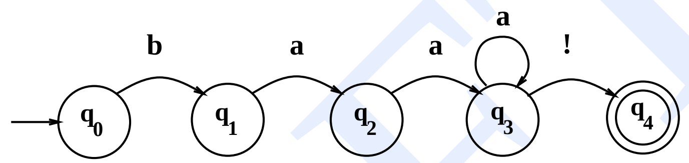


Figure 2.10 A finite-state automaton for talking sheep.

AUTOMATON

STATE

START STATE

The regular expression for this kind of "sheeptalk" is `/baa+!/`. Fig. 2.10 shows an **automaton** for modeling this regular expression. The automaton (i.e., machine, also called **finite automaton**, **finite-state automaton**, or **FSA**) recognizes a set of strings, in this case the strings characterizing sheep talk, in the same way that a regular expression does. We represent the automaton as a directed graph: a finite set of vertices (also called nodes), together with a set of directed links between pairs of vertices called arcs. We'll represent vertices with circles and arcs with arrows. The automaton has five **state**s, which are represented by nodes in the graph. State 0 is the **start state** which we represent by the incoming arrow. State 4 is the **final state** or **accepting state**, which we represent by the double circle. It also has four **transitions**, which we represent by arcs in the graph.

The FSA can be used for recognizing (we also say **accepting**) strings in the following way. First, think of the input as being written on a long tape broken up into cells, with one symbol written in each cell of the tape, as in Fig. 2.11.

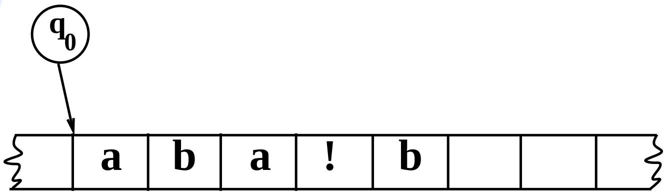


Figure 2.11 A tape with cells.

The machine starts in the start state (q_0), and iterates the following process: Check the next letter of the input. If it matches the symbol on an arc leaving the current state, then cross that arc, move to the next state, and also advance one symbol in the input. If we are in the accepting state (q_4) when we run out of input, the machine has successfully recognized an instance of sheeptalk. If the machine never gets to the final state, either because it runs out of input, or it gets some input that doesn't match an arc (as in Fig. 2.11), or if it just happens to get stuck in some non-final state, we say the machine **rejects** or fails to accept an input.

REJECTS

STATE-TRANSITION TABLE

We can also represent an automaton with a **state-transition table**. As in the graph notation, the state-transition table represents the start state, the accepting states, and what transitions leave each state with which symbols. Here's the state-transition table for the FSA of Figure 2.10.

	Input		
State	b	a	!
0	1	∅	∅
1	∅	2	∅
2	∅	3	∅
3	∅	3	4
4:	∅	∅	∅

Figure 2.12 The state-transition table for the FSA of Figure 2.10.

We've marked state 4 with a colon to indicate that it's a final state (you can have as many final states as you want), and the ∅ indicates an illegal or missing transition. We can read the first row as "if we're in state 0 and we see the input **b** we must go to state 1. If we're in state 0 and we see the input **a** or **!**, we fail".

More formally, a finite automaton is defined by the following five parameters:

$Q = q_0 q_1 q_2 \ldots q_{N-1}$	a finite set of N **states**
Σ	a finite **input alphabet** of symbols
q_0	the **start state**
F	the set of **final states**, $F \subseteq Q$
$\delta(q,i)$	the **transition function** or transition matrix between states. Given a state $q \in Q$ and an input symbol $i \in \Sigma$, $\delta(q,i)$ returns a new state $q' \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q;

For the sheeptalk automaton in Fig. 2.10, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b, !\}$, $F = \{q_4\}$, and $\delta(q,i)$ is defined by the transition table in Fig. 2.12.

Figure 2.13 presents an algorithm for recognizing a string using a state-transition table. The algorithm is called D-RECOGNIZE for "deterministic recognizer". A **deterministic** algorithm is one that has no choice points; the algorithm always knows what to do for any input. The next section will introduce non-deterministic automata that must make decisions about which states to move to.

DETERMINISTIC

D-RECOGNIZE takes as input a tape and an automaton. It returns *accept* if the string it is pointing to on the tape is accepted by the automaton, and *reject* otherwise. Note that since D-RECOGNIZE assumes it is already pointing at the string to be checked, its task is only a subpart of the general problem that we often use regular expressions for, finding a string in a corpus. (The general problem is left as an exercise to the reader in Exercise 2.9.)

D-RECOGNIZE begins by setting the variable *index* to the beginning of the tape, and *current-state* to the machine's initial state. D-RECOGNIZE then enters a loop that drives the rest of the algorithm. It first checks whether it has reached the end of its input. If so, it either accepts the input (if the current state is an accept state) or rejects the input (if not).

If there is input left on the tape, D-RECOGNIZE looks at the transition table to decide which state to move to. The variable *current-state* indicates which row of the table to consult, while the current symbol on the tape indicates which column of the table to consult. The resulting transition-table cell is used to update the variable *current-state* and *index* is incremented to move forward on the tape. If the transition-table cell is empty then the machine has nowhere to go and must reject the input.

```
function D-RECOGNIZE(tape, machine) returns accept or reject

  index ← Beginning of tape
  current-state ← Initial state of machine
  loop
    if End of input has been reached then
      if current-state is an accept state then
        return accept
      else
        return reject
    elsif transition-table[current-state,tape[index]] is empty then
      return reject
    else
      current-state ← transition-table[current-state,tape[index]]
      index ← index + 1
  end
```

Figure 2.13 An algorithm for deterministic recognition of FSAs. This algorithm returns *accept* if the entire string it is pointing at is in the language defined by the FSA, and *reject* if the string is not in the language.

Figure 2.14 traces the execution of this algorithm on the sheep language FSA given the sample input string *baaa!*.

Before examining the beginning of the tape, the machine is in state q_0. Finding a *b* on input tape, it changes to state q_1 as indicated by the contents of *transition-table*$[q_0,b]$ in Fig. 2.12 on page 13. It then finds an *a* and switches to state q_2, another *a* puts it in state q_3, a third *a* leaves it in state q_3, where it reads the "!", and switches to state q_4. Since there is no more input, the `End of input` condition at the beginning of the loop is satisfied for the first time and the machine halts in q_4. State q_4 is an accepting state, and so the machine has accepted the string *baaa!* as a sentence in the sheep language.

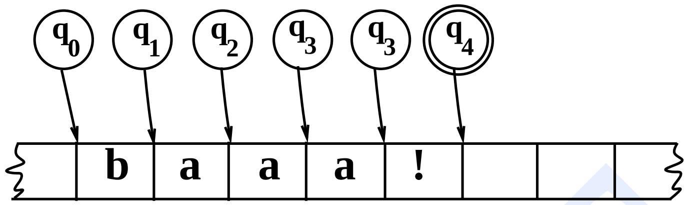


Figure 2.14 Tracing the execution of FSA #1 on some sheeptalk.

The algorithm will fail whenever there is no legal transition for a given combination of state and input. The input *abc* will fail to be recognized since there is no legal transition out of state q_0 on the input a, (i.e., this entry of the transition table in Fig. 2.12 on page 13 has a $\emptyset$). Even if the automaton had allowed an initial *a* it would have certainly failed on *c*, since *c* isn't even in the sheeptalk alphabet! We can think of these "empty" elements in the table as if they all pointed at one "empty" state, which we might call the **fail state** or **sink state**. In a sense then, we could view any machine with empty transitions *as if* we had augmented it with a fail state, and drawn in all the extra arcs, so we always had somewhere to go from any state on any possible input. Just for completeness, Fig. 2.15 shows the FSA from Figure 2.10 with the fail state q_F filled in.

FAIL STATE

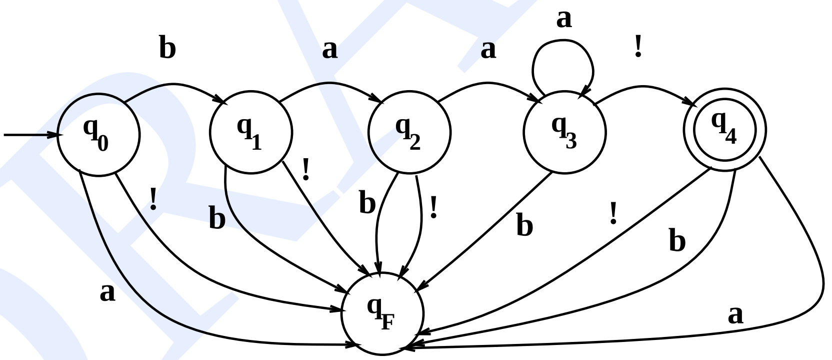


Figure 2.15 Adding a fail state to Fig. 2.10.

2.2.2 Formal Languages

We can use the same graph in Fig. 2.10 as an automaton for GENERATING sheeptalk. If we do, we would say that the automaton starts at state q_0, and crosses arcs to new states, printing out the symbols that label each arc it follows. When the automaton gets to the final state it stops. Notice that at state 3, the automaton has to chose between printing out a **!** and going to state 4, or printing out an **a** and returning to state 3. Let's say for now that we don't care how the machine makes this decision; maybe it flips a coin. For now, we don't care which exact string of sheeptalk we generate, as long as it's a string captured by the regular expression for sheeptalk above.

> **Formal Language:** A model which can both generate and recognize all and only the strings of a formal language acts as a *definition* of the formal language.

FORMAL LANGUAGE
ALPHABET

A **formal language** is a set of strings, each string composed of symbols from a finite symbol-set called an **alphabet** (the same alphabet used above for defining an automaton!). The alphabet for the sheep language is the set $\Sigma = \{a, b, !\}$. Given a model m (such as a particular FSA), we can use $L(m)$ to mean "the formal language characterized by m". So the formal language defined by our sheeptalk automaton m in Fig. 2.10 (and Fig. 2.12) is the infinite set:

$$L(m) = \{baa!, baaa!, baaaa!, baaaaa!, baaaaaa!, \ldots\} \tag{2.1}$$

NATURAL LANGUAGES

The usefulness of an automaton for defining a language is that it can express an infinite set (such as this one above) in a closed form. Formal languages are not the same as **natural languages**, which are the kind of languages that real people speak. In fact, a formal language may bear no resemblance at all to a real language (e.g., a formal language can be used to model the different states of a soda machine). But we often use a formal language to model part of a natural language, such as parts of the phonology, morphology, or syntax. The term **generative grammar** is sometimes used in linguistics to mean a grammar of a formal language; the origin of the term is this use of an automaton to define a language by generating all possible strings.

2.2.3 Another Example

In the previous examples our formal alphabet consisted of letters; but we can also have a higher level alphabet consisting of words. In this way we can write finite-state automata that model facts about word combinations. For example, suppose we wanted to build an FSA that modeled the subpart of English dealing with amounts of money. Such a formal language would model the subset of English consisting of phrases like *ten cents*, *three dollars*, *one dollar thirty-five cents* and so on.

We might break this down by first building just the automaton to account for the numbers from 1 to 99, since we'll need them to deal with cents. Fig. 2.16 shows this.

We could now add *cents* and *dollars* to our automaton. Fig. 2.17 shows a simple version of this, where we just made two copies of the automaton in Fig. 2.16 and appended the words *cents* and *dollars*.

We would now need to add in the grammar for different amounts of dollars; including higher numbers like *hundred*, *thousand*. We'd also need to make sure that the nouns like *cents* and *dollars* are singular when appropriate (*one cent*, *one dollar*), and plural when appropriate (*ten cents*, *two dollars*). This is left as an exercise for the reader (Exercise 2.3). We can think of the FSAs in Fig. 2.16 and Fig. 2.17 as simple grammars of parts of English. We will return to grammar-building in Part II of this book, particularly in Ch. 12.

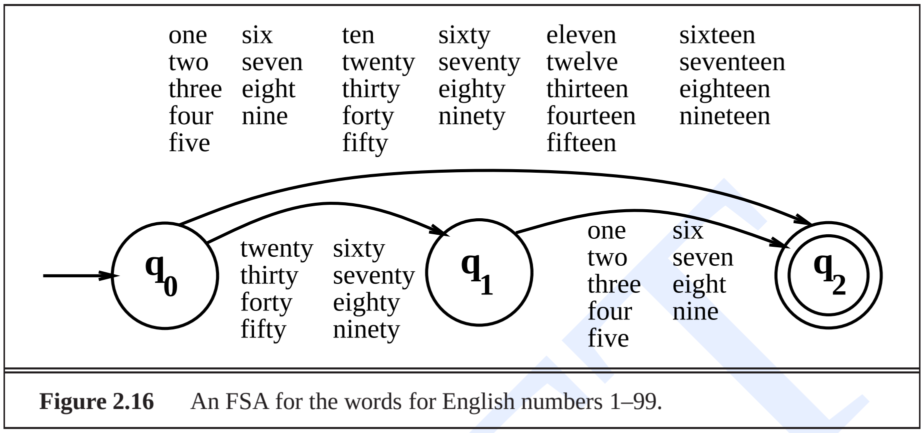

one two three four five
six seven eight nine
ten twenty thirty forty fifty
sixty seventy eighty ninety
eleven twelve thirteen fourteen fifteen
sixteen seventeen eighteen nineteen
q_0
twenty thirty forty fifty
sixty seventy eighty ninety
q_1
one two three four five
six seven eight nine
q_2

Figure 2.16 An FSA for the words for English numbers 1–99.

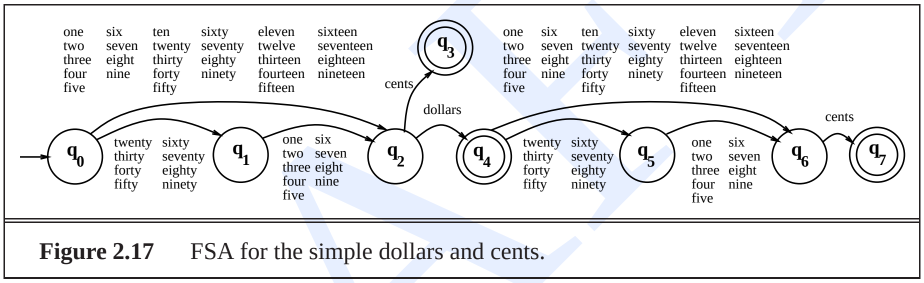

one two three four five
six seven eight nine
ten twenty thirty forty fifty
sixty seventy eighty ninety
eleven twelve thirteen fourteen fifteen
sixteen seventeen eighteen nineteen
q_3
cents
dollars
q_0
twenty thirty forty fifty
sixty seventy eighty ninety
q_1
one two three four five
six seven eight nine
q_2
q_4
one two three four five
six seven eight nine
ten twenty thirty forty fifty
sixty seventy eighty ninety
eleven twelve thirteen fourteen fifteen
sixteen seventeen eighteen nineteen
twenty thirty forty fifty
sixty seventy eighty ninety
q_5
one two three four five
six seven eight nine
q_6
cents
q_7

Figure 2.17 FSA for the simple dollars and cents.

2.2.4 Non-Deterministic FSAs

Let's extend our discussion now to another class of FSAs: **non-deterministic FSA**s (or **NFSA**s). Consider the sheeptalk automaton in Figure 2.18, which is much like our first automaton in Figure 2.10:

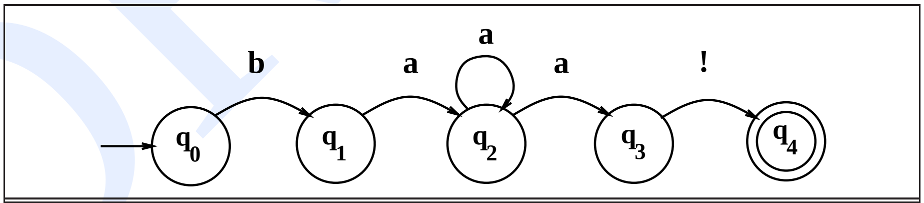


Figure 2.18 A non-deterministic finite-state automaton for talking sheep (NFSA #1). Compare with the deterministic automaton in Fig. 2.10.

The only difference between this automaton and the previous one is that here in Figure 2.18 the self-loop is on state 2 instead of state 3. Consider using this network as an automaton for recognizing sheeptalk. When we get to state 2, if we see an **a** we don't know whether to remain in state 2 or go on to state 3. Automata with decision points like this are called **non-deterministic FSA**s (or **NFSA**s). Recall by contrast that Figure 2.10 specified a **deterministic** automaton, i.e., one whose behavior during recognition is fully *determined* by the state it is in and the symbol it is looking at. A

NON-DETERMINISTIC
NFSA

DFSA

deterministic automaton can be referred to as a **DFSA**. That is not true for the machine in Figure 2.18 (NFSA #1).

ε-TRANSITION

There is another common type of non-determinism, caused by arcs that have no symbols on them (called **ε-transitions**). The automaton in Fig. 2.19 defines the exact same language as the last one, or our first one, but it does it with an ε-transition.

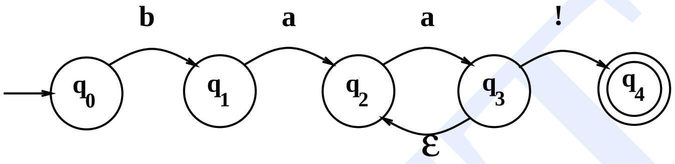


Figure 2.19 Another NFSA for the sheep language (NFSA #2). It differs from NFSA #1 in Fig. 2.18 in having an ε-transition.

We interpret this new arc as follows: If we are in state 3, we are allowed to move to state 2 *without* looking at the input, or advancing our input pointer. So this introduces another kind of non-determinism — we might not know whether to follow the ε-transition or the **!** arc.

2.2.5 Using an NFSA to Accept Strings

If we want to know whether a string is an instance of sheeptalk or not, and if we use a non-deterministic machine to recognize it, we might follow the wrong arc and reject it when we should have accepted it. That is, since there is more than one choice at some point, we might take the wrong choice. This problem of choice in non-deterministic models will come up again and again as we build computational models, particularly for parsing. There are three standard **solutions to the problem of non-determinism**:

BACKUP

- **Backup:** Whenever we come to a choice point, we could put a *marker* to mark where we were in the input, and what state the automaton was in. Then if it turns out that we took the wrong choice, we could back up and try another path.

LOOK-AHEAD

- **Look-ahead:** We could look ahead in the input to help us decide which path to take.

PARALLELISM

- **Parallelism:** Whenever we come to a choice point, we could look at every alternative path in parallel.

We will focus here on the backup approach and defer discussion of the look-ahead and parallelism approaches to later chapters.

The backup approach suggests that we should blithely make choices that might lead to deadends, knowing that we can always return to unexplored alternative choices. There are two keys to this approach: we need to remember all the alternatives for each choice point, and we need to store sufficient information about each alternative so that we can return to it when necessary. When a backup algorithm reaches a point in its processing where no progress can be made (because it runs out of input, or has no

	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Figure 2.20 The transition table from NFSA #1 in Fig. 2.18.

SEARCH-STATE

legal transitions), it returns to a previous choice point, selects one of the unexplored alternatives, and continues from there. Applying this notion to our non-deterministic recognizer, we need only remember two things for each choice point: the state, or node, of the machine that we can go to and the corresponding position on the tape. We will call the combination of the node and position the **search-state** of the recognition algorithm. To avoid confusion, we will refer to the state of the automaton (as opposed to the state of the search) as a **node** or a **machine-state**. Figure 2.21 presents a recognition algorithm based on this approach.

Before going on to describe the main part of this algorithm, we should note two changes to the transition table that drives it. First, in order to represent nodes that have outgoing ε-transitions, we add a new **ε-column** to the transition table. If a node has an ε-transition, we list the destination node in the ε-column for that node's row. The second addition is needed to account for multiple transitions to different nodes from the same input symbol. We let each cell entry consist of a list of destination nodes rather than a single node. Fig. 2.20 shows the transition table for the machine in Figure 2.18 (NFSA #1). While it has no ε-transitions, it does show that in machine-state q_2 the input *a* can lead back to q_2 or on to q_3.

Fig. 2.21 shows the algorithm for using a non-deterministic FSA to recognize an input string. The function ND-RECOGNIZE uses the variable *agenda* to keep track of all the currently unexplored choices generated during the course of processing. Each choice (search state) is a tuple consisting of a node (state) of the machine and a position on the tape. The variable *current-search-state* represents the branch choice being currently explored.

ND-RECOGNIZE begins by creating an initial search-state and placing it on the agenda. For now we don't specify what order the search-states are placed on the agenda. This search-state consists of the initial machine-state of the machine and a pointer to the beginning of the tape. The function NEXT is then called to retrieve an item from the agenda and assign it to the variable *current-search-state*.

As with D-RECOGNIZE, the first task of the main loop is to determine if the entire contents of the tape have been successfully recognized. This is done via a call to ACCEPT-STATE?, which returns *accept* if the current search-state contains both an accepting machine-state and a pointer to the end of the tape. If we're not done, the machine generates a set of possible next steps by calling GENERATE-NEW-STATES, which creates search-states for any ε-transitions and any normal input-symbol transitions from the transition table. All of these search-state tuples are then added to the current agenda.

Finally, we attempt to get a new search-state to process from the agenda. If the agenda is empty we've run out of options and have to reject the input. Otherwise, an unexplored option is selected and the loop continues.

It is important to understand why ND-RECOGNIZE returns a value of reject only when the agenda is found to be empty. Unlike D-RECOGNIZE, it does not return reject when it reaches the end of the tape in a non-accept machine-state or when it finds itself unable to advance the tape from some machine-state. This is because, in the non-deterministic case, such roadblocks only indicate failure down a given path, not overall failure. We can only be sure we can reject a string when all possible choices have been examined and found lacking.

```
function ND-RECOGNIZE(tape, machine) returns accept or reject

 agenda ← {(Initial state of machine, beginning of tape)}
 current-search-state ← NEXT(agenda)
 loop
   if ACCEPT-STATE?(current-search-state) returns true then
      return accept
   else
      agenda ← agenda ∪ GENERATE-NEW-STATES(current-search-state)
   if agenda is empty then
      return reject
   else
      current-search-state ← NEXT(agenda)
 end

function GENERATE-NEW-STATES(current-state) returns a set of search-states

 current-node ← the node the current search-state is in
 index ← the point on the tape the current search-state is looking at
 return a list of search states from transition table as follows:
   (transition-table[current-node,ε], index)
   ∪
   (transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

 current-node ← the node search-state is in
 index ← the point on the tape search-state is looking at
 if index is at the end of the tape and current-node is an accept state of machine
 then
   return true
 else
   return false
```

Figure 2.21 An algorithm for NFSA recognition. The word *node* means a state of the FSA, while *state* or *search-state* means "the state of the search process", i.e., a combination of *node* and *tape-position*.

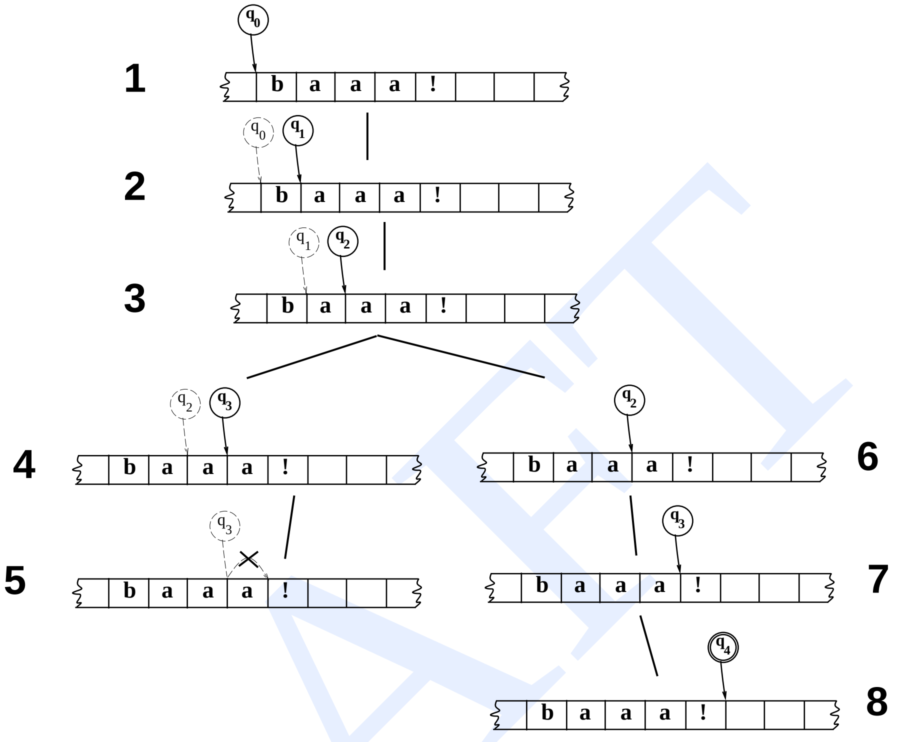


Figure 2.22 Tracing the execution of NFSA #1 (Fig. 2.18) on some sheeptalk.

Figure 2.22 illustrates the progress of ND-RECOGNIZE as it attempts to handle the input `baaa!`. Each strip illustrates the state of the algorithm at a given point in its processing. The *current-search-state* variable is captured by the solid bubbles representing the machine-state along with the arrow representing progress on the tape. Each strip lower down in the figure represents progress from one *current-search-state* to the next.

Little of interest happens until the algorithm finds itself in state q_2 while looking at the second a on the tape. An examination of the entry for transition-table[q_2,a] returns both q_2 and q_3. Search states are created for each of these choices and placed on the agenda. Unfortunately, our algorithm chooses to move to state q_3, a move that results in neither an accept state nor any new states since the entry for transition-table[q_3, a] is empty. At this point, the algorithm simply asks the agenda for a new state to pursue. Since the choice of returning to q_2 from q_2 is the only unexamined choice on the agenda it is returned with the tape pointer advanced to the next a. Somewhat diabolically, ND-RECOGNIZE finds itself faced with the same choice. The entry for transition-table[q_2,a] still indicates that looping back to q_2 or advancing to q_3 are valid choices. As before, states representing both are placed on the agenda. These search states are not the same as the previous ones since their tape index values have advanced. This time the agenda provides the move to q_3 as the next move. The move to q_4, and success, is then uniquely determined by the tape and the transition-table.

2.2.6 Recognition as Search

ND-RECOGNIZE accomplishes the task of recognizing strings in a regular language by providing a way to systematically explore all the possible paths through a machine. If this exploration yields a path ending in an accept state, it accepts the string, otherwise it rejects it. This systematic exploration is made possible by the agenda mechanism, which on each iteration selects a partial path to explore and keeps track of any remaining, as yet unexplored, partial paths.

STATE-SPACE SEARCH

Algorithms such as ND-RECOGNIZE, which operate by systematically searching for solutions, are known as **state-space search** algorithms. In such algorithms, the problem definition creates a space of possible solutions; the goal is to explore this space, returning an answer when one is found or rejecting the input when the space has been exhaustively explored. In ND-RECOGNIZE, search states consist of pairings of machine-states with positions on the input tape. The state-space consists of all the pairings of machine-state and tape positions that are possible given the machine in question. The goal of the search is to navigate through this space from one state to another looking for a pairing of an accept state with an end of tape position.

The key to the effectiveness of such programs is often the *order* in which the states in the space are considered. A poor ordering of states may lead to the examination of a large number of unfruitful states before a successful solution is discovered. Unfortunately, it is typically not possible to tell a good choice from a bad one, and often the best we can do is to insure that each possible solution is eventually considered.

Careful readers may have noticed that the ordering of states in ND-RECOGNIZE has been left unspecified. We know only that unexplored states are added to the agenda as they are created and that the (undefined) function NEXT returns an unexplored state from the agenda when asked. How should the function NEXT be defined? Consider an ordering strategy where the states that are considered next are the most recently created ones. Such a policy can be implemented by placing newly created states at the front of the agenda and having NEXT return the state at the front of the agenda when called. Thus the agenda is implemented by a **stack**. This is commonly referred to as a **depth-first search** or **Last In First Out** (**LIFO**) strategy.

DEPTH-FIRST

Such a strategy dives into the search space following newly developed leads as they are generated. It will only return to consider earlier options when progress along a current lead has been blocked. The trace of the execution of ND-RECOGNIZE on the string `baaa!` as shown in Fig. 2.22 illustrates a depth-first search. The algorithm hits the first choice point after seeing `ba` when it has to decide whether to stay in q_2 or advance to state q_3. At this point, it chooses one alternative and follows it until it is sure it's wrong. The algorithm then backs up and tries another older alternative.

Depth first strategies have one major pitfall: under certain circumstances they can enter an infinite loop. This is possible either if the search space happens to be set up in such a way that a search-state can be accidentally re-visited, or if there are an infinite number of search states. We will revisit this question when we turn to more complicated search problems in parsing in Ch. 13.

The second way to order the states in the search space is to consider states in the order in which they are created. Such a policy can be implemented by placing newly created states at the back of the agenda and still have NEXT return the state at the front of the agenda. Thus the agenda is implemented via a **queue**. This is commonly referred to as a **breadth-first search** or **First In First Out** (**FIFO**) strategy. Consider a different trace of the execution of ND-RECOGNIZE on the string baaa! as shown in Fig. 2.23. Again, the algorithm hits its first choice point after seeing ba when it had to decide whether to stay in q_2 or advance to state q_3. But now rather than picking one choice and following it up, we imagine examining all possible choices, expanding one ply of the search tree at a time.

BREADTH-FIRST

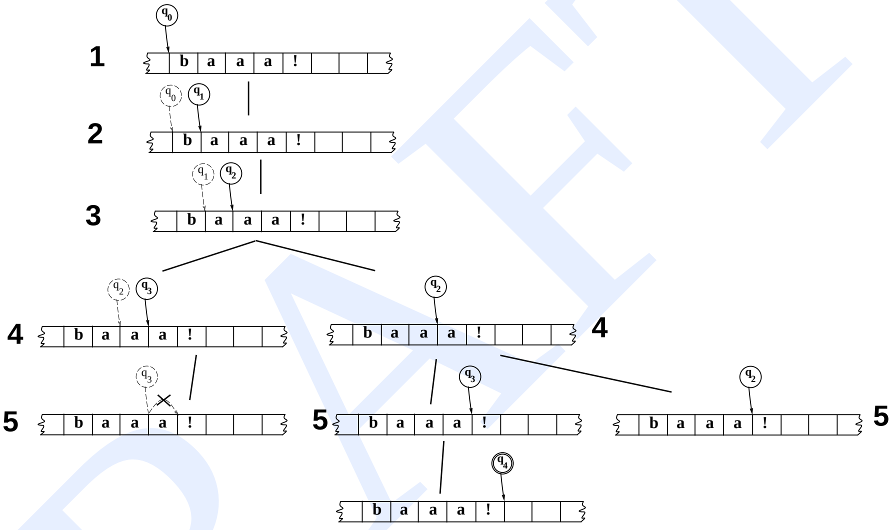


Figure 2.23 A breadth-first trace of FSA #1 on some sheeptalk.

Like depth-first search, breadth-first search has its pitfalls. As with depth-first if the state-space is infinite, the search may never terminate. More importantly, due to growth in the size of the agenda if the state-space is even moderately large, the search may require an impractically large amount of memory. For small problems, either depth-first or breadth-first search strategies may be adequate, although depth-first is normally preferred for its more efficient use of memory. For larger problems, more complex search techniques such as **dynamic programming** or **A*** must be used, as we will see in Chapters 7 and 10.

2.2.7 Relating Deterministic and Non-Deterministic Automata

It may seem that allowing NFSAs to have non-deterministic features like ε-transitions would make them more powerful than DFSAs. In fact this is not the case; for any NFSA, there is an exactly equivalent DFSA. In fact there is a simple algorithm for converting an NFSA to an equivalent DFSA, although the number of states in this equivalent deterministic automaton may be much larger. See Lewis and Papadimitriou

(1988) or Hopcroft and Ullman (1979) for the proof of the correspondence. The basic intuition of the proof is worth mentioning, however, and builds on the way NFSAs parse their input. Recall that the difference between NFSAs and DFSAs is that in an NFSA a state q_i may have more than one possible next state given an input i (for example q_a and q_b). The algorithm in Figure 2.21 dealt with this problem by choosing either q_a or q_b and then *backtracking* if the choice turned out to be wrong. We mentioned that a parallel version of the algorithm would follow both paths (toward q_a and q_b) simultaneously.

The algorithm for converting a NFSA to a DFSA is like this parallel algorithm; we build an automaton that has a deterministic path for every path our parallel recognizer might have followed in the search space. We imagine following both paths simultaneously, and group together into an equivalence class all the states we reach on the same input symbol (i.e., q_a and q_b). We now give a new state label to this new equivalence class state (for example q_{ab}). We continue doing this for every possible input for every possible group of states. The resulting DFSA can have as many states as there are distinct sets of states in the original NFSA. The number of different subsets of a set with N elements is 2^N, hence the new DFSA can have as many as 2^N states.

2.3 REGULAR LANGUAGES AND FSAS

As we suggested above, the class of languages that are definable by regular expressions is exactly the same as the class of languages that are characterizable by finite-state automata (whether deterministic or non-deterministic). Because of this, we call these languages the **regular languages**. In order to give a formal definition of the class of regular languages, we need to refer back to two earlier concepts: the alphabet Σ, which is the set of all symbols in the language, and the *empty string* ϵ, which is conventionally not included in Σ. In addition, we make reference to the *empty set* $\emptyset$ (which is distinct from ϵ). The class of regular languages (or **regular sets**) over Σ is then formally defined as follows: [1]

REGULAR LANGUAGES

1. $\emptyset$ is a regular language
2. $\forall a \in \Sigma \cup \epsilon$, $\{a\}$ is a regular language
3. If L_1 and L_2 are regular languages, then so are:
   (a) $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$, the **concatenation** of L_1 andL_2
   (b) $L_1 \cup L_2$, the **union** or **disjunction** of L_1 andL_2
   (c) L_1^*, the **Kleene closure** of L_1

Only languages which meet the above properties are regular languages. Since the regular languages are the languages characterizable by regular expressions, all the regular expression operators introduced in this chapter (except memory) can be implemented by the three operations which define regular languages: concatenation, disjunction/union (also called "|"), and Kleene closure. For example all the counters (`*`,`+`, `{n,m}`) are just a special case of repetition plus Kleene *. All the anchors can be

[1] Following van Santen and Sproat (1998), Kaplan and Kay (1994), and Lewis and Papadimitriou (1988).

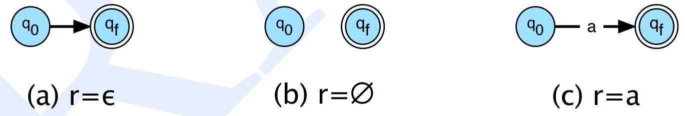
thought of as individual special symbols. The square braces `[]` are a kind of disjunction (i.e., `[ab]` means "*a* or *b*", or the disjunction of *a* and *b*). Thus it is true that any regular expression can be turned into a (perhaps larger) expression which only makes use of the three primitive operations.

Regular languages are also closed under the following operations (Σ^* means the infinite set of all possible strings formed from the alphabet Σ):

- **intersection**: if L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$, the language consisting of the set of strings that are in both L_1 and L_2.
- **difference**: if L_1 and L_2 are regular languages, then so is $L_1 - L_2$, the language consisting of the set of strings that are in L_1 but not L_2.
- **complementation**: If L_1 is a regular language, then so is $\Sigma^* - L_1$, the set of all possible strings that aren't in L_1.
- **reversal**: If L_1 is a regular language, then so is L_1^R, the language consisting of the set of reversals of all the strings in L_1.

The proof that regular expressions are equivalent to finite-state automata can be found in Hopcroft and Ullman (1979), and has two parts: showing that an automaton can be built for each regular language, and conversely that a regular language can be built for each automaton.

We won't give the proof, but we give the intuition by showing how to do the first part: take any regular expression and build an automaton from it. The intuition is inductive on the number of operators: for the base case we build an automaton to correspond to the regular expressions with no operators, i.e. the regular expressions $\emptyset$, ε, or any single symbol $a \in \Sigma$. Fig. 2.24 shows the automata for these three base cases.



(a) r=ϵ (b) r=∅ (c) r=a

Figure 2.24 Automata for the base case (no operators) for the induction showing that any regular expression can be turned into an equivalent automaton.

Now for the inductive step, we show that each of the primitive operations of a regular expression (concatenation, union, closure) can be imitated by an automaton:

- **concatenation**: We just string two FSAs next to each other by connecting all the final states of FSA_1 to the initial state of FSA_2 by an ε-transition.
- **closure**: We create a new final and initial state, connect the original final states of the FSA back to the initial states by ε-transitions (this implements the repetition part of the Kleene *), and then put direct links between the new initial and final states by ε-transitions (this implements the possibility of having *zero* occurrences). We'd leave out this last part to implement Kleene-plus instead.
- **union**: We add a single new initial state q_0', and add new ε-transitions from it to the former initial states of the two machines to be joined.

We will return to regular languages and their relationship to regular grammars in Ch. 15.

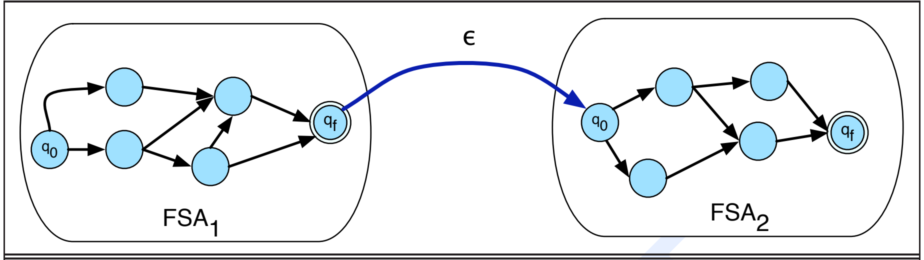


Figure 2.25 The concatenation of two FSAs.

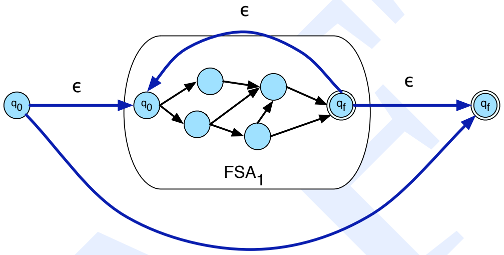


Figure 2.26 The closure (Kleene *) of an FSA.

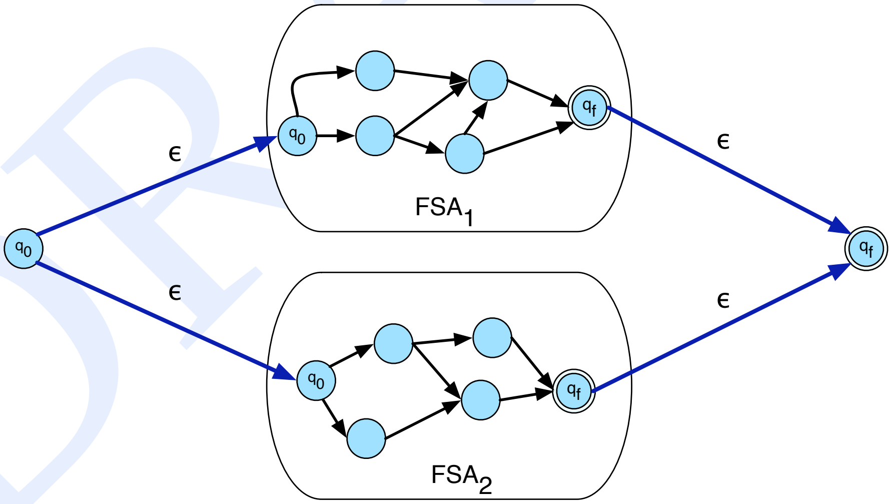


Figure 2.27 The union (|) of two FSAs.

2.4 Summary

This chapter introduced the most important fundamental concept in language processing, the **finite automaton**, and the practical tool based on automaton, the **regular ex-**

pression. Here's a summary of the main points we covered about these ideas:

- The **regular expression** language is a powerful tool for pattern-matching.
- Basic operations in regular expressions include **concatenation** of symbols, **disjunction** of symbols (`[]`, `|`, and `.`), **counters** (`*`, `+`, and `{n,m}`), **anchors** (`^`, `$`) and precedence operators (`(`,`)`).
- Any regular expression can be realized as a **finite state automaton** (**FSA**).
- Memory (`\1` together with `()`) is an advanced operation that is often considered part of regular expressions, but which cannot be realized as a finite automaton.
- An automaton implicitly defines a **formal language** as the set of strings the automaton **accepts**.
- An automaton can use any set of symbols for its vocabulary, including letters, words, or even graphic images.
- The behavior of a **deterministic** automaton (**DFSA**) is fully determined by the state it is in.
- A **non-deterministic** automaton (**NFSA**) sometimes has to make a choice between multiple paths to take given the same current state and next input.
- Any **NFSA** can be converted to a **DFSA**.
- The order in which a **NFSA** chooses the next state to explore on the agenda defines its **search strategy**. The **depth-first search** or **LIFO** strategy corresponds to the agenda-as-stack; the **breadth-first search** or **FIFO** strategy corresponds to the agenda-as-queue.
- Any regular expression can be automatically compiled into a **NFSA** and hence into a **FSA**.

Bibliographical and Historical Notes

Finite automata arose in the 1950s out of Turing's (1936) model of algorithmic computation, considered by many to be the foundation of modern computer science. The Turing machine was an abstract machine with a finite control and an input/output tape. In one move, the Turing machine could read a symbol on the tape, write a different symbol on the tape, change state, and move left or right. Thus the Turing machine differs from a finite-state automaton mainly in its ability to change the symbols on its tape.

MCCULLOCH-PITTS NEURON

Inspired by Turing's work, McCulloch and Pitts built an automata-like model of the neuron (see von Neumann, 1963, p. 319). Their model, which is now usually called the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), was a simplified model of the neuron as a kind of "computing element" that could be described in terms of propositional logic. The model was a binary device, at any point either active or not, which took excitatory and inhibitatory input from other neurons and fired if its activation passed some fixed threshold. Based on the McCulloch-Pitts neuron, Kleene (1951) and (1956) defined the finite automaton and regular expressions, and proved their equivalence. Non-deterministic automata were introduced by Rabin and Scott (1959), who also proved them equivalent to deterministic ones.

Ken Thompson was one of the first to build regular expressions compilers into editors for text searching (Thompson, 1968). His editor *ed* included a command "g/regular expression/p", or Global Regular Expression Print, which later became the UNIX `grep` utility.

There are many general-purpose introductions to the mathematics underlying automata theory, such as Hopcroft and Ullman (1979) and Lewis and Papadimitriou (1988). These cover the mathematical foundations of the simple automata of this chapter, as well as the finite-state transducers of Ch. 3, the context-free grammars of Ch. 12, and the Chomsky hierarchy of Ch. 15. Friedl (1997) is a very useful comprehensive guide to the advanced use of regular expressions.

The metaphor of problem-solving as search is basic to Artificial Intelligence (AI); more details on search can be found in any AI textbook such as Russell and Norvig (2002).

Exercises

2.1 Write regular expressions for the following languages: You may use either Perl notation or the minimal "algebraic" notation of Sec. 2.3, but make sure to say which one you are using. By "word", we mean an alphabetic string separated from other words by white space, any relevant punctuation, line breaks, and so forth.

a. the set of all alphabetic strings.

b. the set of all lowercase alphabetic strings ending in a *b*.

c. the set of all strings with two consecutive repeated words (e.g., "Humbert Humbert" and "the the" but not "the bug" or "the big bug").

d. the set of all strings from the alphabet a,b such that each a is immediately preceded and immediately followed by a *b*.

e. all strings which start at the beginning of the line with an integer (i.e., 1,2,3,...,10,...,10000,...) and which end at the end of the line with a word.

f. all strings which have both the word *grotto* and the word *raven* in them. (but not, for example, words like *grottos* that merely *contain* the word *grotto*).

g. write a pattern which places the first word of an English sentence in a register. Deal with punctuation.

2.2 Implement an ELIZA-like program, using substitutions such as those described on page 10. You may choose a different domain than a Rogerian psychologist, if you wish, although keep in mind that you would need a domain in which your program can legitimately do a lot of simple repeating-back.

2.3 Complete the FSA for English money expressions in Fig. 2.16 as suggested in the text following the figure. You should handle amounts up to $100,000, and make sure that "cent" and "dollar" have the proper plural endings when appropriate.

2.4 Design an FSA that recognizes simple date expressions like *March 15*, *the 22nd of November*, *Christmas*. You should try to include all such "absolute" dates, (e.g. not "deictic" ones relative to the current day like *the day before yesterday*). Each edge of the graph should have a word or a set of words on it. You should use some sort of shorthand for classes of words to avoid drawing too many arcs (e.g., furniture → desk, chair, table).

2.5 Now extend your date FSA to handle deictic expressions like *yesterday*, *tomorrow*, *a week from tomorrow*, *the day before yesterday*, *Sunday*, *next Monday*, *three weeks from Saturday*.

2.6 Write an FSA for time-of-day expressions like *eleven o'clock*, *twelve-thirty*, *midnight*, or *a quarter to ten* and others.

2.7 (Due to Pauline Welby; this problem probably requires the ability to knit.) Write a regular expression (or draw an FSA) which matches all knitting patterns for scarves with the following specification: *32 stitches wide, K1P1 ribbing on both ends, stockinette stitch body, exactly two raised stripes*. All knitting patterns must include a cast-on row (to put the correct number of stitches on the needle) and a bind-off row (to end the pattern and prevent unraveling). Here's a sample pattern for one possible scarf matching the above description:[2]

1.	Cast on 32 stitches.	*cast on; puts stitches on needle*
2.	K1 P1 across row (i.e. do (K1 P1) 16 times).	*K1P1 ribbing*
3.	Repeat instruction 2 seven more times.	*adds length*
4.	K32, P32.	*stockinette stitch*
5.	Repeat instruction 4 an additional 13 times.	*adds length*
6.	P32, P32.	*raised stripe stitch*
7.	K32, P32.	*stockinette stitch*
8.	Repeat instruction 7 an additional 251 times.	*adds length*
9.	P32, P32.	*raised stripe stitch*
10.	K32, P32.	*stockinette stitch*
11.	Repeat instruction 10 an additional 13 times.	*adds length*
12.	K1 P1 across row.	*K1P1 ribbing*
13.	Repeat instruction 12 an additional 7 times.	*adds length*
14.	Bind off 32 stitches.	*binds off row: ends pattern*

2.8 Write a regular expression for the language accepted by the NFSA in Fig. 2.28.

2.9 Currently the function D-RECOGNIZE in Fig. 2.13 only solves a subpart of the important problem of finding a string in some text. Extend the algorithm to solve the following two deficiencies: (1) D-RECOGNIZE currently assumes that it is already

[2] *Knit* and *purl* are two different types of stitches. The notation K*n* means do *n* knit stitches. Similarly for purl stitches. Ribbing has a striped texture—most sweaters have ribbing at the sleeves, bottom, and neck. Stockinette stitch is a series of knit and purl rows that produces a plain pattern— socks or stockings are knit with this basic pattern, hence the name.

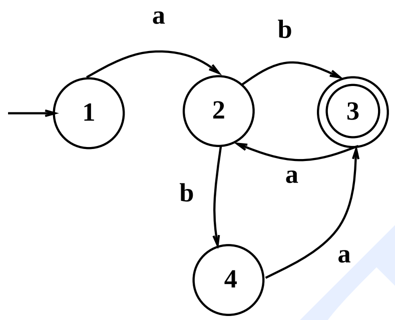


Figure 2.28 A mystery language

pointing at the string to be checked, and (2) D-RECOGNIZE fails if the string it is pointing includes as a proper substring a legal string for the FSA. That is, D-RECOGNIZE fails if there is an extra character at the end of the string.

2.10 Give an algorithm for negating a deterministic FSA. The negation of an FSA accepts exactly the set of strings that the original FSA rejects (over the same alphabet), and rejects all the strings that the original FSA accepts.

2.11 Why doesn't your previous algorithm work with NFSAs? Now extend your algorithm to negate an NFSA.

Friedl, J. E. F. (1997). *Master Regular Expressions*. O'Reilly.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, *20*(3), 331–378.

Kleene, S. C. (1951). Representation of events in nerve nets and finite automata. Tech. rep. RM-704, RAND Corporation. RAND Research Memorandum†.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. and McCarthy, J. (Eds.), *Automata Studies*, pp. 3–41. Princeton University Press, Princeton, NJ.

Lewis, H. and Papadimitriou, C. (1988). *Elements of the Theory of Computation*. Prentice-Hall. Second edition.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–133. Reprinted in *Neurocomputing: Foundations of Research, ed. by J. A. Anderson and E Rosenfeld. MIT Press 1988*.

Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, *3*(2), 114–125.

Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall. Second edition.

Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM*, *11*(6), 419–422.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *42*, 230–265. Read to the Society in 1936, but published in 1937. Correction in volume 43, 544–546.

van Santen, J. P. H. and Sproat, R. (1998). Methods and tools. In Sproat, R. (Ed.), *Multilingual Text-To-Speech Synthesis: The Bell Labs Approach*, pp. 7–30. Kluwer, Dordrecht.

von Neumann, J. (1963). *Collected Works: Volume V*. Macmillan Company, New York.

Weizenbaum, J. (1966). ELIZA – A computer program for the study of natural language communication between man and machine. *Communications of the ACM*, *9*(1), 36–45.