Hardware

The finite-state tools are provided in object form, compiled for Solaris (SUN workstations), for Microsoft, for Linux and for Mac OS X. Although interesting experiments can be performed on modest machines, large-scale linguistic development can often require serious computer crunch and large amounts of RAM. Although the final results of most morphological analyzers, for example, are under five megabytes in size, and while the Xerox finite-state algorithms are the fastest and most memory-efficient available, intermediate stages of various operations can often explode in size before minimization operations can be invoked. While 24 or 36 MBytes of RAM might be considered a minimum for full-scale work on most languages, XRCE maintains SUN workstations with up to 2 GBytes of RAM to handle particularly intensive finite-state computations.

Other Documentation

This book replaces previous documentation, some of which is terse, lacking in examples, and increasingly out of date. Where appropriate, sections of the following documents have been recycled:


Online documentation is also available at www.fsmbook.com.

Chapter 1

A Gentle Introduction

1.1 The Beginning

This book shows how to use the Xerox finite-state programming languages `xfst` and `lex` to create finite-state networks that perform morphological analysis and related tasks. So let’s begin at the beginning: What are FINITE-STATE NETWORKS and why should anyone care about them?

The following gentle introduction will attempt to paint the big picture, avoiding technical vocabulary and mathematical definitions, but conveying the core concepts necessary to make sense of all the formalism that must follow. We’ll try to present a vision of where we’re headed, showing how finite-state networks, created using various Xerox tools, are combined together into practical working applications, and in particular into morphological analyzers. Finally, we will try to convey the general notion that finite-state networks are Good Things, useful for doing many kinds of linguistic processing. Compared to the alternatives, applications based on finite-state networks are usually smaller, faster, more elegant, and easier to maintain and modify.

Analogies and explanations in this gentle introduction are not to be taken too far or too seriously; some will certainly not stand up to rigorous scrutiny. Experienced computational linguists and computer scientists can safely skip this section and move on to the more formal introduction in the next chapter.

1.2 Some Unavoidable Terminology

First of all, what is meant by the terms `FINITE`, `STATE` and `NETWORK`? Let’s start with the term STATE, which has everyday meanings that are closely related to the formal one we will need.
1.2.1 State

Substances, people, and physical machines are often said to be in one state or another. For example, $H_2O$ is a substance that can exist in a frozen state (ice), a liquid state (water), or a gaseous state (water vapor). Under the right conditions, $H_2O$ will change from one state to another. Similarly, a person may at any given time be in a happy state, a depressed state, an excited state, a bored state, an amorous state, etc.; and like $H_2O$, people usually do a lot of changing from one state to another until they reach a final dead state. Finally, many physical machines in the real world can be described in terms of the states they can assume and the way that they change from one state to another.

While it is perhaps most common to talk about machine states when discussing complex machines like computers, let’s start with some simpler mechanical machines that are easier to grasp. Consider first a common light switch, which is always in one of two states, which we will label **ON** and **OFF**. By convention, we will represent or model the states of this and other machines with circles as in Figure 1.1.

A common light switch changes from the **ON** state to the **OFF** state, and vice versa, when we humans physically manipulate it from one position to the other. Let us assume that the **OFF** state corresponds to the switch lever being down and that the **ON** state corresponds to the switch lever being up. The things that we can do to a light switch are pretty limited: when the switch is down (OFF state) we can push it up; and when the switch is up (ON state) we can push it down. The possible transitions from one state to another are modeled by conventional, with an arc leading from one state to another. Each arc is labeled with the action or input that causes the particular change of state.

Note that our little graphic model of the light-switch machine in Figure 1.2 ignores details of physical material, shape, electrical contacts and wiring that can vary greatly and don’t usually concern us. What our model does capture is the state behavior of the light switch, showing all the possible states it can assume, all the possible inputs, and how the inputs cause transitions from one state to another. Other inputs, such as hitting the switch with a hammer, are conceivable but illegal—they would result in the destruction of the machine. Such illegal inputs are simply absent from the model.

Note also that particular inputs may be legal only when the machine is in particular states. Thus when the switch is in the **ON** state, pushing down is legal; but if the switch is already in the **OFF** state, pushing down is illegal as it would break the machine. Illegal inputs and transitions are simply left out of the model.

Now let’s model another kind of on-off light switch that has a single button to push rather than a lever to manipulate. Pushing the single button toggles the switch from **ON** to **OFF** and also from **OFF** to **ON**. Such a toggling switch has but a single legal input, pushing the button, as shown in Figure 1.3. Notice here that the effect of an input to a machine can differ radically depending on the current state of the machine.

Before we push these examples too far, let’s move on to a slightly more complex physical machine and model its behavior in a similar way. The fan in Ken’s old car had exactly four settings or states in which it could be placed, **OFF**, **LOW**, **MED** (medium), and **HIGH**. One could change the state by turning a dial Left (counterclockwise) or Right (clockwise). Figure 1.4 is a state model of this particular fan machine.

The way the fan control was built, it allowed no direct transition from **OFF** to **MED** or **HIGH**, from **LOW** to **HIGH**, from **MED** to **OFF**, or from **HIGH** to **LOW** or **OFF**; one had to move the machine, even if rather quickly, through the
intermediate states. From the OFF state, turning the dial Left again would break
the machine; the result of turning the dial Right from the HIGH state would be
similarly unfortunate. Such illegal inputs are simply not shown in the model.

We can easily imagine, and find, similar machines with dials that turn com-
pletely around in either direction, or perhaps only in one direction, allowing transi-
tions between the OFF and HIGH states. So-called 3-way lamps in the USA have
four states—OFF, LOW, MED, and HIGH—controlled by a dial that turns all the
way around, but only to the right, and these machines are modeled in Figure 1.5.
Comparing this machine with the one in Figure 1.4, we see that two machines can
have exactly the same states but different inputs and transitions.

1.2.2 Finite

Another inescapable technical term within finite-state networks is FINITE, which
refers to the number of states and can be satisfactorily defined, for present purposes,
as simply being “not infinite”.

So far we have looked at some very simple physical machines and have mod-
eled them in terms of possible states and transitions from one state to another. The
examples have been restricted to those with a finite number of states. Consider

one more kind of light control, the dimmer or rheostat, that allows an infinite num-
ber of gradations (states) between fully OFF and fully ON; such a machine is not
finite-state, and we cannot model a light dimmer with a finite number of circles and
arcs. Similarly, human beings display mixtures and infinite gradations of emo-
tional states that preclude a finite-state modeling.

1.2.3 Networks

The final term to examine is NETWORK, which will need to be reexamined with
more rigor in the following chapter. For now, let us accept that networks are graph-
like structures of nodes linked together with directed arc-transitions as shown in
Figures 1.4 and 1.5.

Other terms for finite-state network commonly used in more formal discus-
sion are FINITE-STATE MACHINE and FINITE-STATE AUTOMATA. Other more complicated networks are called TRANSDUCERS. The word
automaton is also commonly used to describe robots, especially those that seem to
move about and act under their own will. The finite-state networks that we will be
defining and manipulating may not be as cute as little metal rodents negotiating a
maze, but we will see that they are indeed abstract machines that can perform some
interesting linguistic tasks for us.

As a stepping stone to the linguistic examples, let us examine and model one
last mechanical machine called the Cola Machine. It is based on the familiar coin-
operated soft-drink machines, and we will limit our modeling to that part of the
machine that accepts our coin inputs and decides when we have entered enough
money to deserve a drink. We will not try to model the refrigerator, the drink-
selection system, the drink-delivery system, or anything else. To further simplify
the example, we specify that our cola machine has the following characteristics:

2. The only coins accepted by the machine are
   - The quarter (abbreviated Q) which is worth 25 cents,
   - The dime (abbreviated D) which is worth 10 cents, and
   - The nickel (abbreviated N) which is worth 5 cents.
3. The machine accepts any combination of these coins, in any order, that add
   up to 25 cents.
4. The machine requires exact change.

This coin-accepting machine is in fact a finite-state machine, and our task is to
model it as a network of states and transitions. If we walk up this machine, before
we put any money into it, it will be in a START STATE that we can label helpfully
as 0 (zero). In the 0 state, it will steadfastly refuse to deliver a drink, and our job is
to enter appropriate coins that change the state of the machine until it is in a special **final or accepting state**, which we will label as 25, that allows a drink to be selected. By convention, we will use a double circle to represent such a final state.

The first obvious way to change the machine from the start state to the final state is to input a quarter as in Figure 1.6. We can then select our preferred beverage, the coin-accepting machine will reset to state zero (magically for now), and it will be ready for the next thirsty customer.

![Figure 1.6: The Cola Machine Accepts a Quarter and Transitions Directly to the Final State](image)

There are of course other ways to get a drink, by adding various combinations and permutations of nickels and dimes. If we start by adding a nickel (N), the resulting new state of the machine will be closer to the final goal, but the machine still won't give us a drink; let's label the new non-final state 5 as in Figure 1.7. If we continue to add four more nickels, we will reach three more non-final states and, at last, the final state. The complete model of our simple Cola Machine is shown in Figure 1.8. Adding dimes will, in this particular machine, cause a jump of two states at a time. For example, adding a nickel (N), a dime (D) and another dime (D), in that order, will move the machine from the 0 state to the 5 state to the 15 state and finally to the final 25 state.

![Figure 1.7: Inputting a Nickel to the Cola Machine](image)

Now that we have our complete coin-accepting machine modeled, we list all the possible sequences of coins that it will accept, where acceptance means reaching the final state and allowing us to get a drink.

\[
\begin{align*}
Q &\rightarrow QD \rightarrow QDN \rightarrow QDNN \\
D &\rightarrow DD \rightarrow DDD \rightarrow DNNN \\
N &\rightarrow NN \rightarrow NNNN
\end{align*}
\]

These nine sequences of coins are the only valid inputs, to this particular cola machine, if you want to get a drink. Any other sequence of coins, including too few coins, or too many coins (in our simple machine that requires exact change) simply won't work. Other kinds of coins, like pennies (worth one cent) or any kind of foreign coins including Mexican pesos or Canadian quarters, are simply illegal inputs and will cause the mechanism to jam. As usual, illegal inputs are simply left out of the model.

Here is where we can make the transition from our mechanical machines to the linguistic ones that we will learn about and build in this book:

- Think of the inputs to the machine not as coins (quarters, dimes and nickels) but as letter **symbols** like Q, D and N.
- The set of valid symbols that the machine will accept is its **alphabet**.
- The sequences of symbols that the machine will accept are **words**.
- The entire set of words that the machine accepts or recognizes is its **language**.

In this case, the nine words listed above would constitute the entire Cola Machine Language.
This technical use of the word language, to denote just a collection of symbol strings (words), is somewhat odd, but we are stuck with it in formal language theory. In what follows, we will see that the task of the computational linguist is usually to make the formal language accepted by our finite-state network correspond as closely as possible to a natural language, such as French and Spanish. Our task, in short, is to model natural languages.

1.3 A Few Intuitive Examples

1.3.1 Finite-State Languages and Natural Languages

All of our technical terms, especially the odd usage of ALPHABET and LANGUAGE, will be made more precise as we progress. What we will do now is present a few small but fairly typical finite-state networks that linguists might build, and we will suggest ways that they could be genuinely useful in various kinds of natural-language processing.

Figure 1.9 shows a very small finite-state network that accepts the single word “canto”, which happens to look like a word in Spanish. We can also say that the language of this machine consists of the single word “canto”. If we think of the real Spanish language, rather perversely, as just the set of all its possible written words, then our new network models a very small subset of Spanish. The alphabet of the machine consists of just five symbols: c, a, n, t and o. The machine itself consists of a start state (if not overtly labeled, the start state is the leftmost one in our diagrams), a final state (marked with a double circle), and non-final states in between linked by arcs labeled c, a, n, t and o, in that order.

![Figure 1.9: A Network that Accepts a One-Word Language]

If we walk up to our new machine, like we walked up to the Cola Machine with our coins, and if we enter the symbols c, a, n, t and o, in that order, the machine will transition through a series of states, ending up in the final state, and the word will be accepted. We don’t get a drink as a reward this time, but the finite-state machine will in essence tell us “I accept this string”, which means “This string is in my language”. If we enter any other string, such as “Libro”, it will be rejected.

Now let’s imagine a slightly larger machine, shown in Figure 1.10, that accepts a language consisting of three strings, “canto”, “tiare” and “mesa”. Again, all of these words happen to look like words of Spanish. Again, each valid word corresponds to the symbols on a path from the start state (by convention the leftmost state in the diagram) to a final state (indicated with a double circle). This new machine recognizes a language with a bigger alphabet and vocabulary, but it is not different in type or usage from the first. If you enter the symbols m, e, s and a, in that order, the word “mesa” will be accepted, as will any other string of symbols representing a word in the language of the network. Enter any word not in the language, such as “panes”, and it will be rejected.

Now imagine the same network expanded to include three million words, all of them happening to correspond to words of the real Spanish language. At this point, we have a potentially valuable basis for several natural-language-processing systems, the most obvious being a spelling checker. Simple spell-checking involves somehow looking up each word from a document to see if it is a valid word, and flagging all the words that are not found. Given that we have a large finite-state network that accepts over 3,000,000 Spanish-like words, all we have to do is to enter each word from the document and flag all the words that the network rejects. The quality of the spell-checker will depend largely on the coverage and accuracy of the network, i.e. the degree to which the formal language that it accepts corresponds to the real Spanish language. Xerox linguists routinely build and manipulate networks of this size, modeling real natural languages like Spanish and French as closely as humanly possible.

1.3.2 Symbol Matching and Analysis

It’s time to introduce some slightly different metaphors for the word-entering and accepting process, those of ANALYSIS (also called LOOKUP) and SYMBOL MATCHING. When we enter a word (as a string of symbols) into a network, to see if it is contained in the language of the network, we often talk of LOOKING UP the word. From this point of view, the network is a kind of dictionary. The analysis (lookup) will be successful if and only if the word is in the language of the network. Such analysis, viewed as a process, involves matching the symbols of the input word one-by-one against the symbols on the arcs of a path through the network. Let’s return to our 3-word language and analyze (look up) the string “mesa” as shown in Figure 1.11.
1.3 A Few Intuitive Examples

1.3.3 Getting More Back from Analysis: Transducers

So far the analysis of words in a network has simply yielded one of two responses, either an ACCEPT, indicating that the word is in the language of the network, or a REJECT, indicating that the word is not in the language. While this can be valuable, as for instance in spell-checking, finite-state networks are capable of storing and returning much more interesting information. The first step in understanding this capability is to imagine our three-word network with a pair of labels on each arc rather than just a single label. Such a two-level network is shown in Figure 1.12.

![Figure 1.12: A Two-Level Network or Transducer](image)

The analysis process is now modified slightly:

- Start at the start state and at the beginning of the input string.
- Match the input symbols of the input string against the lower-side symbols on the arcs, consuming input symbols and finding a path from the start state to a final state.
- If successful, return the string of upper-side symbols on the path as the result.
- If the analysis is not successful, return nothing.

If we analyze the word “mesa”, the successful result will now be an output string “mesa” as shown in Figure 1.13. The dotted arrows in the figure indicate where input symbols are matched along the lower side of a path through the network from the start state to the final state. The output is the string of symbols on the upper side of this successful path. Granted, this output is not especially interesting, being exactly the same as the input, but in fact the upper-side symbols in a network do not need to be the same as the lower-side symbols.

The Xerox Spanish Morphological Analyzer network includes over 3,000,000 paths, from the start state to a final state, that look like the path in Figure 1.14. When you analyze the word “canto”, as shown in Figure 1.15, one of the solutions returned is the string “cantar+Verb+FresInd+1P+Sg”, which is intended to be read as follows:

![Figure 1.11: Successful Analysis of the Word “mesa”](image)
1. The traditional baseform is *cantar* ("to sing").
2. The word is a verb.
3. The verb is conjugated in the present-indicative (Spanish combines the tense and mood into a single inflection paradigm).
4. First person.
5. Singular.

Thus the process of analysis identifies "*canto*" as the Spanish equivalent of the English "I sing", and yet it's all done via the simple language-independent process of matching symbols from the input string against paths of symbols in the network. The actual analysis step is trivial, language-independent and very fast when performed by computers. The hard part is defining and building the network itself.

Notice the new MULTICRCHARACTER SYMBOLS labeling some of the arcs in Figure 1.14: +Verb, +PresInd, +1P and +Sg are in fact single symbols, with multicharacter print names, that were chosen and defined by the linguists who built the system. The spelling and order of these symbol TAGS, and the choice of the infinitive as the baseform, were also determined by the linguists. Another special symbol in the network of Figure 1.14 is the EPSILON symbol (ε), which represents the empty string and fills in the gaps when the upper-level string of symbols and the lower-level string of symbols are not of the same length. During analysis, EPSILON ARCS on the bottom side are traversed freely without consuming any input symbols. You will also see the symbol 0 (zero) used for the epsilon in Xerox notations and compilers because the epsilon is not available on standard ASCII keyboards.

Another of the three million paths through the Spanish network looks like Figure 1.16. When you look up "*canto*", the analysis algorithm detects the multiple possibilities and automatically BACKTRACKS to find and return a second solution as well: "*canto+Noun+Masc+Sg*". Again, this solution is just another string of symbols, where +Noun and +Masc (masculine) are more multicharacter-symbol tags defined by the linguists who created the system. The Spanish noun *canto* means "song".

Two-sided networks, like the Spanish Morphological Analyzer, are called LEXICAL TRANSDUCERS. In more common discourse, a transducer is a device that converts energy from one form to another; thus a microphone is a transducer that converts physical vibrations of air into analogous electrical signals. Our finite-state transducers convert one string of symbols into another string or strings of symbols. When the transducer is constructed so that the lower-side language consists of valid written words in Spanish, and the upper-side language consists of strings showing baseforms and tags, and when the strings are properly related, the result is an ex-
1.3.4 Generation

The opposite of analysis is generation, and in fact we use the exact same Spanish network, applying it backwards, to generate surface strings from analysis strings. Assume that we want to generate the first-person plural, present-indicative form of the Spanish verb cantar. It happens that the Spanish network also contains the path shown in Figure 1.17.

![Figure 1.17: Another Verb Path](image)

The process of generation (sometimes called lookdown) is just the inverse of analysis (also called lookup). Let us assume that the input string, the string we want to generate from, is "cantar+Verb+PresInd+1P+Pl", i.e. the first-person plural, present-indicative form of the verb cantar.

- Start at the start state and at the beginning of the input string.
- Match the input symbols of the input string one-by-one with the upper-side symbols on the arcs, consuming input symbols and finding a path from the start state to a final state.
- If successful, return the string of lower-side symbols on the path as the result.
- If generation is not successful, return nothing.

The output of the generation in this case will be the surface string "cantanmos" ("we sing"). The epsilon symbols in the network represent the empty string and are ignored in the output. One can, of course, turn back around and enter "cantanmos" for analysis and get back the corresponding analysis string. Finite-state transducers are inherently bidirectional.

In a more formal sense, the set of analysis strings produced during analysis, or accepted during generation, constitutes the analysis language (or upper-side language) of the transducer. The term lexical language is also synonymous with upper-side language. Similarly, the set of surface strings constitutes the surface language (or lower-side language) of the transducer. The transducer maps between strings of the upper-side language and strings of the lower-side language. Each string in the upper-side language of a transducer is related to one or more strings in the lower-side language, and vice-versa.

1.3.5 Further Finite-State Applications

A morphological analyzer/generator, like the Spanish Lexical Transducer discussed above, is often the first goal when a linguist applies finite-state tools and techniques to a new language. A lexical transducer does morphological analysis or generation, depending on which way it is applied to the input, and it is often a vital component of larger systems, such as syntactic parsers, or a crucial starting point for making derivative systems like spelling checkers or part-of-speech taggers.

Only the smallest example networks can be diagrammed, so we typically view a network as a black-box component, as in Figure 1.18, keeping in mind that it consists of a finite number of states and labeled arcs as in the examples shown above. From the start state to a final state, there may be millions (or even an infinite number) of paths in the network; but however large the lexical transducer may become, analysis (lookup) and generation (lookdown) are performed by the same language-independent symbol-matching techniques.

![Figure 1.18: A Transducer Viewed as a Black Box](image)

Besides morphological analysis and generation, finite-state networks can do many other jobs for us, including tokenization, which is the dividing up of a running text into individual tokens. Tokens usually, but not always, correspond to our everyday notion of words, but there are complicating exceptions, including contractions like couldn't, joined words like cannot, and indivisible multiword tokens like to and from that often require special attention.

It is possible and often quite useful to define finite-state networks that tokenize strings of written natural language. Such an English tokenizer might be defined so that the lower-side language consists of strings that look like English text. When such surface strings are looked up, the output string is the input string plus defined multicharacter symbols, e.g. "TB", for "Token Boundary", inserted between tokens as shown in Figure 1.19. By convention, feature-like multicharacter symbols in Xerox finite-state systems are usually spelled with an initial circumflex (') character.

Other common applications of finite-state techniques include part-of-speech guessers, which are typically defined to guess categories of words based on...
characteristic affixes, and systems that perform phonological/orthographical alternations, based on rule types that have been used by linguists for centuries. More advanced and experimental applications include shallow syntactic parsing or "chunking", certain kinds of counting and perfect hashing. Suffice it to say that while systems of finite-state power cannot do everything necessary in linguistics, they can do a great deal; and more practical applications are being found all the time.

### 1.4 Sharing Structure

All of the networks we have seen in the previous section have a single initial state and a single final state. Every word or pair of words in these examples has its own unique path sharing only the common initial and final state.

In fact a network can have any number of final states, and an arc may be part of multiple paths. Figure 1.20 shows a simple example. This network recognizes two words, "fat" and "father". Because the words begin with the same three letters, the first three arcs of the network can be shared. The path for "father" goes through the final state terminating "fat" and continues to the second final state.

![Figure 1.20: A Network for "fat" and "father"](image)

In an optimally configured network, all words that begin or end the same way as some other word share states and arcs that encode the common beginnings and endings. Figure 1.21 shows an optimally encoded network that accepts the four words "clear", "clever", "ear" and "ever". Because all the words end in "r", they can all share the last arc leading to the final state. The paths for "clear" and "clever" go together up to the point where the words diverge and join again for the common ending. Only the first letter of "ear" and "ever" needs its own arc.

![Figure 1.21: A Network for "clear", "clever", "ear" and "ever"](image)

The network in Figure 1.21 is minimal in the sense that it is impossible to encode the same four paths using fewer states and arcs. Minimality is an important property for many practical applications. For a typical natural language, a minimal network can encode in the space of a few hundred kilobytes a list of words that takes several megabytes of disk space in text format. The compression ratio is often better than zip or any other general compression can offer.\(^1\) We will discuss minimality and other formal properties of networks in Section 2.5.

Because of structure sharing, removing paths from a network may actually increase its size. For example, if we remove "ear" from the network in 1.21, one new state and arc have to be added, as shown in Figure 1.22.

![Figure 1.22: A Network for "clear", "clever" and "ever"](image)

### 1.5 Some Background in Set Theory

Formal language theory assumes some concepts and terminology from set theory, including membership, union, intersection, subtraction and complementation. In the same spirit as the rest of this chapter, we will introduce these concepts more intuitively than formally, always trying to illustrate them with linguistically oriented

---

\(^1\)For example, the 25K Unix word list /usr/dict/words takes up 206K bytes of disk space as a text file, 81K as a gziped file, and just 75K as a finite-state network.
examples showing how sets and operations on them are represented by finite-state networks. Some of these concepts will be examined more formally in the next chapter.

1.5.1 Sets

What is a Set?

A set corresponds to the everyday notions of a collection or group. If you have objects A, B, C, D, E and F, they might be grouped into two sets as shown in Figure 1.23. We use ellipses, arbitrarily, to show set groupings of objects; they should not be confused with the circles that represent states in our network diagrams.

![Figure 1.23: Two Sets](image)

We say that the first set has four members or elements, A, B, C and D. The second set has two members, E and F. For sets that have a finite number of members, as in Figure 1.23, we can explicitly list or enumerate the sets as {A, B, C, D} and {E, F}. The order in which the elements of a set are listed has no significance. {A, B} and {B, A} denote the same set. There is only one instance of each element in a set, thus {A, A, B} is a redundant listing of the set {A, B}.

In finite-state linguistics, we often talk about a network encoding or accepting a particular set of strings; and we call this set a language. We usually think of these strings as words, and we use the two terms string and word interchangeably. For example, Figure 1.22 shows a network whose language is a set of three strings. Each word accepted by a network is a member of the set of words that constitutes the language of the network. For a small language like this one, we can enumerate it easily as {"clear", "clever", "ever"}.

The Empty Set

Some sets are empty, like the set of all living Neandertals or the set of all transparent elephants. Empty sets have no members. The empty language is the language that contains no strings at all. This notion of the empty language must be distinguished from the notion of the empty string, a string of length zero. Thus if "dog" is a string of length 3, and "tiger" is a string of length 5, then "" is a string of length 0 (zero), the empty string. The empty string is a string and may be a member of a language; but the empty language contains no strings at all, not even the empty string.

![Figure 1.24: The (Empty) Set of All Living Neandertals](image)

In finite-state terms the empty language is an empty set, represented by a network that consists of one non-final state (see Figure 1.25).

![Figure 1.25: Network for the Empty Language {}](image)

A network consisting of a single final state with no arcs represents the language that contains just the empty string (see Figure 1.26). The empty-string language (i.e. the language that contains just the empty string) is not an empty set; it contains one element, which is the empty string.

![Figure 1.26: Network for the Empty-String Language {}](image)

Infinite Sets

Many sets, such as the set of all integers, contain an infinite number of elements. Real human languages that allow productive compounding, like German, contain an infinite number of words, at least theoretically. Obviously, it is impossible to enumerate all the members of an infinite set, but as we shall see in the next chapter, we can easily specify many infinite languages, using the finite metalanguage of regular expressions.

Some infinite languages can be represented by a very small finite-state network. For example, the language {"", "a", "aa", "aaa", \ldots} that contains all strings of zero or more as is represented by the network in Figure 1.27.
Universal Set

The infinite set that contains all strings of any length, including the empty string, is called the UNIVERSAL LANGUAGE. If the question mark (?) represents any symbol, then the network shown in Figure 1.28 accepts the universal language.

Ordered Set

An ordered set is a set whose members are ordered in some way. We will only be concerned with one very specific type of ordered set. An ORDERED PAIR is a set that has two members: a first and a second element. To distinguish an ordered pair from an ordinary set we list it in angle brackets. While \{A, B\} and \{B, A\} denote the same set, \langle A, B \rangle and \langle B, A \rangle are distinct ordered pairs. One is the inverse of the other.

1.5.2 Relations

A RELATION is a set whose members are ordered pairs. Words like father, husband, wife, spouse, etc. denote relations in this technical sense because they involve pairs of individuals. For example, consider the British royal family tree in Figure 1.29.

A genealogical tree implicitly defines certain relations over a set of people. For example, the father relation in Figure 1.29 is the set \{\langle Philip, Charles \rangle, \langle John, Diana \rangle\}. Here we chose the ordering that reflects the word order in the corresponding English sentences: Philip is the father of Charles, etc. By the same principle, the wife relation is the set \{\langle Elisabeth, Philip \rangle, \langle Frances, John \rangle, \langle Diana, Charles \rangle\}. Its inverse, wife⁻¹, is the husband relation: \{\langle Philip, Elisabeth \rangle, \langle John, Frances \rangle, \langle Charles, Diana \rangle\}.

In this book we focus on relations that contain pairs of strings, for example,
in the alphabet. To save space, we may draw such networks with just one multiply labeled arc, as in Figure 1.30, but in reality each label has its own arc. Using this network, the analysis and generation methods defined in the previous section will convert any string from uppercase to lowercase or vice versa.

![Figure 1.30: The Lowercase/Uppercase Transducer. Each lowercase:UPPERCASE pair of symbols in fact has its own arc.](image)

The path that relates "xyzzy" to "XYZZY" cycles many times through the single state of the transducer. Figure 1.31 shows it in a linearized form. The lowercase/upercase relation may be thought of as the representation of a simple orthographic rule. In fact we view all kinds of string-changing rules in this way, that is, as infinite string-to-string relations. The networks that represent them are generally much more complex than the little transducer shown in Figure 1.30.

![Figure 1.31: A Linearized Path in the Lowercase/Uppercase Transducer](image)

Identity Relation

One technically important but otherwise uninteresting kind of relation is one in which the upper and the lower languages are the same and every string of the language is paired with itself. This is called the **identity relation**. For example, the network in Figure 1.12 represents the identity relation \{"canto", "canto"\}, \{"tigre", "tigre"\}, \{"mesa", "mesa"\} on the simple language \{"canto", "tigre", "mesa"\} of Figure 1.10.

### 1.5.3 Some Basic Set Operations

**Union**

The union of any two sets $S_1$ and $S_2$ is another set that contains all the elements in $S_1$ and all the elements in $S_2$. The union of the two sets introduced in Figure 1.23 is the set shown in Figure 1.32. The order of the sets in the union operation makes no difference: the union of $S_1$ and $S_2$ is the same set as the union of $S_2$ and $S_1$, just as in addition $2+3$ is the same as $3+2$. Union and addition are **commutative** operations.

![Figure 1.32: The Union of Sets \{A, B, C, D\} and \{E, F\}](image)

Now consider another pair of sets and their union, as shown in Figure 1.33. We have purposely scattered the members in the set ellipses to emphasize the fact that the set members are not ordered; the set \{Q, R, M\} is the same as \{Q, M, R\}, \{M, R, Q\}, etc. Note also that the two original sets have three members each, but the union has five rather than six members. In this case, the Q member is common to both the original sets, so it appears only once in the union.

![Figure 1.33: The Union of Sets \{Q, R, M\} and \{A, Q, P\}](image)

Some of the relations in Figure 1.29 are unions of other relations. For example the *spouse* relation \{<Elisabeth, Philip>, <Philip, Elisabeth>, <Frances, John>, <John, Frances>, <Diana, Charles>, <Charles, Diana>\} is obviously the union of the *wife* relation with its inverse, the *husband* relation. Similarly, *parent* is the
union of the *father* and *mother* relations.

Languages, being sets of strings, can also be unioned. So if we start with the language represented in Figure 1.34, and another language as represented in Figure 1.35, the union of these two languages is the language accepted by the network in Figure 1.36. There is no significant ordering of the arcs leading from a state in a finite-state network, just as there is no significant ordering of members in sets. The network could be represented in many other equivalent ways.

Figure 1.34: A Network for “fat” and “father”

Figure 1.35: A Network for “clear”, “clever”, “ear” and “ever”

Figure 1.36: A Network for the Union of {“fat”, “father”} with {“clear”, “clever”, “ear”, “ever”}

Unioning of languages (or the unioning of networks that accept languages) is valid for both simple one-level languages and the more complex relations, encoded as two-level transducers. In finite-state linguistic development, this often makes it possible for one person to work on adjectives, another on nouns, and a third on verbs, producing three different networks that can later simply be unioned together into a single network.

1.5 SOME BACKGROUND IN SET THEORY

**Intersection**

For any two sets $S_1$ and $S_2$, the intersection is the set containing all the members that are common to both sets. The intersection of $\{Q, R, M\}$ and $\{Q, A, P\}$, as shown in Figure 1.37, is the set $\{Q\}$. The intersection of two sets that share no members, such as $\{A, B, Z\}$ and $\{M, N, C\}$, is the empty set. Like union, intersection is a commutative operation: the intersection of $S_1$ and $S_2$ is the same as the intersection of $S_2$ and $S_1$.

Figure 1.37: Intersection of Sets

Simple languages, and networks accepting simple languages, can also be intersected. Thus if one network accepts the simple language {“apple”, “banana”, “pear”}, and another network accepts the simple language {“kumquat”, “pear”, “grape”, “quince”, “banana”}, the intersection of these two simple networks accepts the language {“banana”, “pear”}. As we shall see in coming chapters, only simple one-level languages (modeled with one-level networks) can be intersected; relations (modeled with two-level transducers) can be intersected only in special cases.

**Subtraction**

The subtraction of sets is also easy to conceptualize and diagram. In the simplest case, shown in Figure 1.38, set $\{Q, M, R\}$ minus set $\{Q\}$ leaves set $\{M, R\}$.

Simple languages, and networks accepting simple languages, can also be subtracted. Thus if language $L_1$ is {“apple”, “banana”, “pear”}, and language $L_2$ is {“kumquat”, “pear”, “grape”, “quince”, “banana”}, the subtraction of $L_2$ from $L_1$ is {“apple”}. It is important to understand that subtraction, unlike intersection and union, is not a commutative operation: $L_2$ minus $L_1$ can be completely different from $L_1$ minus $L_2$, just as in numeric subtraction 3-2 is not the same as 2-3. As we shall see in coming chapters, only simple languages (modeled with one-level networks) can be subtracted; relations (modeled with two-level
transducers) can be subtracted only in special cases.

**Concatenation**

The operation of CONCATENATION is an easy one to grasp through linguistic examples; in the most intuitive case, consider the network that corresponds to the language that contains a single string "work", which happens to look like a verb in English (Figure 1.39). Now imagine another language of three words, "s", "ed" and "ing", which happen to look like verbal suffixes of English. The network for this language is shown in Figure 1.40. If we concatenate the verbal-suffix language on the end of the "work" language, we get the new language recognized by the network in Figure 1.41, which contains the three strings "works", "worked" and "working".

We can also see the "work" string itself as a concatenation of w, o, r and k; and the "ed" string is a concatenation of e and d, and similarly for "ing". In any finite-state language, the strings are simply concatenations of symbols available from the alphabet of the language.

As the example would suggest, concatenation is the main mechanism for building complex words in a finite-state grammar, and lex (Chapter 4) is the primary Xerox language to use for specifying the concatenations that construct entire natural-language words out of the parts known as MORPHEMES. The study and modeling of legal word formation is called MORPHOTACTICS or, in some traditions, MORPHOSYNTAX.

But there is another problem yet to be solved. The concatenation of verb baseforms like work with s, ed and ing is a productive morphotactic process in English, also applying to verbs like talk, kill and print. However, when we try to extend this simple concatenation of verbal endings "s", "ed" and "ing" to other English verbs like try, plot and wiggle, we immediately find problems.

\*trys \*tryed trying

\*plots \*ploted \*ploting

\*wiggles \*wiggleed \*wiggling

We use the prefixed asterisk (*) to mark those words that aren’t spelled correctly. The correct forms, paired vertically with the incorrect ones, are shown below. Spaces are used to line up the two strings letter-by-letter as closely as possible.

try s tried plot ed plot ing wiggleed wiggleing

tries tried plotted plotting wigg ed wiggl ing

There are many ways to characterize the differences or ALTERNATIONS between the two LEVELS of strings, but we note in rough terms that sometimes let-
ters need to be changed, such as a y becoming I in tried, or added, as in *ploting becoming plotting, or deleted, as in *wiggleing becoming wigglng. These alternations from one level to the other can be described by finite-state rules written in the form of Xerox replace rules.

Rules that describe phonological and orthographical alternations can be viewed as infinite string-to-string relations just like the simple lowercase/uppercase rule in Figure 1.30. They naturally tend to be more complicated to express, but there is no fundamental difference. To discuss this idea in more depth we first have to introduce the last operation in this section.

Composition

The operation of composition is a little more complex than the ones we have seen so far. Composition is an operation on two relations, and the result is a new relation. If one relation contains the ordered pair \(<x, y>\) and the other relation contains the ordered pair \(<y, z>\), the relation resulting from composing \(<x, y>\) and \(<y, z>\), in that order, will contain the ordered pair \(<x, z>\). Composition brings together the "outside" components of the two pairs and eliminates the common one in the middle, as shown in Figure 1.42.

![Figure 1.42: The Composition of the Ordered Pairs \(<x, y>\) and \(<y, z>\). The common middle element y is eliminated in the result, which is the ordered pair \(<x, z>\).](image)

Looking back at the royal family tree in Figure 1.29, we see that some new relations can be defined in terms of others by way of composition. For example the father-in-law relation, \{<Philip, Diana>, <John, Charles>\}, is obviously the composition of father, \{<Philip, Charles>, <John, Diana>, <Charles, William>\}, and spouse, \{<Elisabeth, Philip>, <Philip, Elisabeth>, <Frances, John>, <John, Frances>, <Diana, Charles>, <Charles, Diana>\}. The grandfather relation, \{<Philip, William>, <John, William>\}, is the composition of father and parent.

To illustrate composition with a very simple string example, consider the relation \{"cat", "chat"\} and the relation \{"chat", "Katze"\}. The result of the composition is \{"cat", "Katze"\}, where "cat" and "Katze" happen to be the French and German words for "cat", respectively.

Without going into too many technical details, it is useful to have a general idea of how composition is carried out when such string-to-string relations are represented by finite-state networks. The transducers representing the two relations are shown in Figure 1.43. Recall that ε (epsilon) represents the empty string.

![Figure 1.43: The "cat", "chat" and "chat", "Katze" Transducers](image)

We can think of composition of transducers as a two-step procedure. First the paths of the two networks that have a matching string in the middle are merged, as shown in Figure 1.44. The method of finding the matching path is essentially the same as described in Sections 1.3.3 and 1.3.4 on Analysis and Generation except that here we are tracking a string in two networks simultaneously. We then eliminate the string "chat" in the middle, giving us a transducer that directly maps "cat" to "Katze", and vice versa, as shown in Figure 1.45. If we were to compose in the opposite order, that is "chat", "Katze" with "cat", "chat"", the result would be the empty relation because there is no matching string in the middle.

Let us move on to a less trivial example of network composition, a case in which one of the relations is infinite. This time we will compose the \{"cat", "chat"\} network with the lowercase/uppercase transducer in Figure 1.30. For
Projection

It is often useful to extract one side from a given relation. This simple operation on relations is called projection. From the lowercase/uppercase relation represented in Figure 1.46, we can derive a network for all lowercase strings or for all uppercase strings by just systematically relabeling the arcs in the desired way. The upper projection is obtained by replacing a:A by a, c:C by c, and so on. The lower projection is obtained by replacing each pair label with the lower (= second) member of the pair.

1.6 Composition and Rule Application

If we think of the lowercase/uppercase transducer as representing an orthographical rule, then the composition of the two networks in Figure 1.48 is in effect the application of this rule on the lower side of the "cat", "chat" pair. When a string-changing rule is represented by a finite-state transducer, composition and rule application become essentially the same notion. There are only a couple of minor points of difference.

The first difference is that composition is technically defined as an operation on two relations, whereas we generally think of linguistic rules, such as the change of y to i in some context, as being applied to individual underlying strings like "tries". But it is easy to bridge the gap between the traditional intuition and the formalism. The application of the lowercase/uppercase rule to an individual string, such as "chat", can be seen as the composition of the corresponding identity relation {"chat", "chat"} with the lowercase/uppercase relation, which gives us the result in Figure 1.49.
yields an input/output relation as in Figure 1.49. But it is of course a trivial matter to obtain the output language from the input/output relation by projection, i.e. by extracting the lower-side language {"CHR"}.

Once we think of rule application as composition, we immediately see that a rule can be applied to several words, or even a whole language of words, at the same time if the words are represented as a finite-state network. Two rule transducers can also be composed with one another to yield a single transducer that gives the same result as the successive application of the original rules, just as our \(<"cat", "Katze"\>) transducer translates directly from English "cat" to German "Katze", eliminating the intermediate French translation "chat".

### 1.7 Lexicon and Rules

This is not yet the time to go into the details of real linguistic rules. The transducers that represent them are often vastly more complex than the one-state network in Figure 1.46. For the time being, let us ignore the internal details and think of a transducer that, for example, maps the incorrect string "trys" to the correct string "tries" as shown in the black box in Figure 1.50.

![Figure 1.50: Rules Map Abstract Upper-Side or LEXICAL Strings into SURFACE Strings](image)

Typically, the lexicon is a finite-state transducer, defined with a lexc program, that generates morphotactically well-formed but still rather abstract strings. Such strings are sometimes called "underlying", morphophonemic or, in the finite-state tradition, LEXICAL strings. Rules then map the abstract lexical strings into properly spelled SURFACE strings. This scheme supposes at least two separately defined finite-state networks as in Figure 1.51.

From the perspective of a linguist, this idea appears too simple. The mapping between lexical strings and surface strings can be very complex. It is certainly not possible to describe dozens of morphological alternations by a single rule. Orthographical and phonological rules are typically about the realization of just one symbol, for example, y/e alternation in English, or about a particular class of symbols such as word-final devoicing in German. The linguistic description of the phonological and morphological alternations of natural languages typically requires dozens of rules. And there are other possible complications: some rules may have exceptions, and some rules may have priority over other rules.

For centuries, going back to the famous Sanskrit grammarian Panini who lived around 500 BC, linguists have described phonological alternations and historical sound changes in terms of UNDERLYING strings and REWRITE RULES that are applied in a given order with the output of one rule "feeding" the next (see Figure 1.52). The output of a rewrite rule is an INTERMEDIATE string, with the output of the final rule being a SURFACE string. Most modern phonologists model the underlying, intermediate and surface strings as sequences of phonemes, which in turn are bundles of features; the alternation rules can match and modify individual features within phonemes. In computational linguistics, we usually deal with orthographical symbols rather than phonetic feature bundles, but the principle is the same.

It was C. Douglas Johnson (Johnson, 1972) who apparently first realized that the phonological rewrite rules used by linguists could theoretically be modeled as finite-state transducers (FSTs). Furthermore, individual rule transducers can be arranged in a cascade in which the output of one transducer serves as the input for the next one. Johnson also observed that for any cascade of transducers that performs a particular mapping, there exists in theory a single transducer that performs the same mapping in a single step. This single transducer can be computed by successively composing the transducers with one another until just one remains. In other words, even if the mapping between lexical strings and surface strings is very complex and cannot be described by just one rule, we can in principle always produce a single rule transducer that does the equivalent mapping, as Figure 1.53 illustrates.

In finite-state systems we can in fact go one step further. Finite-state rules ap-
The Classic Phonological Paradigm Changing Underlying Strings into Surface Strings Via a Cascade of Rewrite Rules

```
Underlying String

Rewrite Rule 1

Intermediate String

Rewrite Rule 2

Intermediate String

Rewrite Rule 3

Intermediate String

Rewrite Rule n

Surface String
```

Figure 1.52: The Classic Phonological Paradigm Changing Underlying Strings into Surface Strings Via a Cascade of Rewrite Rules

...to entirely lexicons, which are also encoded as finite-state transducers, so the separate lexicon network and rule network shown in Figure 1.51 can be composed together into a single all-inclusive network, a LEXICAL TRANSDUCER, as shown in Figure 1.54. Such a lexical transducer incorporates all the lexicon and rule information in a single network data structure, mapping directly between a language of underlying or “lexical” strings and a language of surface strings.

Another way to arrive at a single lexical transducer is to use the two-level rule formalism invented by Kimmo Koskenniemi (Koskenniemi, 1983). Like classical rewrite rules, two-level rules can also be modeled as finite-state transducers (Karttunen et al., 1987), and a system of two-level rules can be composed with the lexicon to produce a single lexical transducer (Karttunen et al., 1992).²

²The Xerox twole compiler for two-level rules is included in the software licensed with this book and is documented on the webpage http://www.fsmbook.com/. At Xerox, most developers have abandoned twole rules in favor of the xfst replace rules described herein.

```
Underlying String

Rule FST 1

Intermediate String

Rule FST 2

Intermediate String

Rule FST 3

Intermediate String

Rule FST n

Surface String

Single Rule FST
```

Figure 1.53: A Cascade of Alternation Rules, Compiled into Finite-State Transducers, Can be Combined into a Single Equivalent FST via Composition. This mathematical possibility, shown by Johnson, can be performed in practice using the Xerox finite-state software.
1.8 The Big Picture

The Xerox lexc compiler and the xsf interface are the tools that linguists use to create finite-state networks, including the lexical transducers that do morphological analysis and generation. You will learn how to use lexc to build finite-state lexicons, specifying baseforms and affixes and describing the morphotactic possibilities for each word. In fact lexc is just one of several ways to specify finite-state transducers, but it is especially designed to facilitate the work of the lexicographer. You will also learn to use xsf replace rules to formalize the alternation rules traditionally used in phonology and morphology. And you will learn to use the full arsenal of the Finite-State Calculus, available through the xsf interface, to combine, customize, test and constrain your systems to fit your needs.

Because all the lexicons and rules defined by the linguist are compiled into finite-state networks, they can (respecting some formal mathematical restrictions) be combined together using any operations that are valid for finite-state networks, including union, concatenation, subtraction, intersection, and composition. Lexicon networks are almost always composed together with rule networks that map the somewhat abstract lexical strings into correctly spelled surface strings. For some natural languages, it is possible and convenient to divide up the work, doing nouns, verbs, and adjectives separately; the resulting sublanguage networks can then simply be unioned together into a single lexical transducer when they are finished.

And why doesn't everyone do computational linguistics this way? Because although the mathematical properties of finite-state networks have been well un-

1.9 Finite-State Networks are Good Things

Without getting into formal details of how finite-state languages differ from other kinds of formal languages, such as context-free and context-sensitive languages, we do want to emphasize here that computing with finite-state machines is attractive. First, the mathematical properties of finite-state networks are well understood, allowing us to manipulate and combine finite-state networks in ways that would be impossible using traditional algorithmic programs; there is a mathematical beauty to finite-state computing that translates into unparalleled flexibility. Second, finite-state networks are computationally efficient for tasks like natural-language morphological analysis, resulting in phenomenal processing speeds. Third, in most cases, finite-state networks can store a great deal of information in relatively little memory, and finite-state networks can be further compressed using commercial Xerox technology.

From a development point of view, finite-state programming is declarative; the linguist encodes facts about the language being modeled, not ad hoc algorithms. The algorithms that are required for the compilation of networks, and for applying them to do analysis and generation, are already part of the toolkit and are completely language-independent. Multiple language modules therefore all share the same runtime "engine", and adding new language modules to an existing system involves just plugging new finite-state networks into the existing framework. Finite-state morphology is an excellent example of the principle of separating language-independent code from language-dependent data in natural-language processing systems.

1.10 Exercises

It takes some mind-tuning to become skilled in thinking about and computing with finite-state networks; it is a very different paradigm from the writing of procedures and algorithms. Throughout the book, we will present exercises to consolidate what has been learned, and these are the first. Be aware that superficially different solutions may be equivalent.

1.10.1 The Better Cola Machine

Starting from the Cola Machine example in Section 1.2.3, consider a slightly more sophisticated cola machine that returns change. As in the original Cola Machine,
the valid inputs are nickels (N) worth 5 cents, dimes (D) worth 10 cents and quarters (Q) worth 25 cents; and it still costs 25 cents for a drink. As before, we won’t try to model the selection or the delivery of drinks, but the new and better machine should now return appropriate change when you enter too much money. In particular:

- If the machine has reached the final state and the user continues to add coins, the extra coins will simply be returned to the user.
- If the user enters a partial sum, e.g. 2 dimes (equal to 20 cents), and then (s)he enters too much money (i.e. either another dime or a quarter), then the machine will accept the input, move into the final state and return appropriate change.

Your task is to draw this better cola machine using states and labeled arcs.

Hints:

1. Start with the machine as shown in Figure 1.8. All the states and arcs in this machine are still valid for this exercise.

2. Arbitrarily, we will choose to think of entering coins as a kind of generation, so we will match our inputs against the upper-side symbols of the network.

3. We will need to model the new machine as a two-level transducer, with input symbols on top of the arcs and output symbols on the bottom of the arcs. Use epsilon symbols where necessary so that there is always one symbol (or epsilon) above, and one symbol (or epsilon) below, each arc. Symbols indicating our change (a kind of output) should appear on the bottom of arcs.

4. From the final state, there should be three arcs that loop back to the final state:
   - One labeled with N on top and N on the bottom
   - One labeled with D on top and D on the bottom
   - One labeled with Q on top and Q on the bottom

   This models what happens when too much money is entered: the extra coins are simply returned.

5. You will need to create some new intermediate states in the network diagram.

   Trace the behavior of your machine through the following scenarios:
   - If you enter exactly 25 cents, you should reach the final state as in the original cola machine, with no change returned.

   - If you enter a quarter and then a nickel, you should reach the final state and get back a nickel.
   - If you enter too little money, you should not reach the final state (and you will therefore go thirsty).
   - If you enter three dimes ("DDD") you should reach the final state and get back a nickel (N).
   - If you enter three nickels and then a quarter, the machine should reach the final state and return 15 cents in some appropriate combination of coins (the choice of coins to be returned is up to you).
   - Imagine and handle all other possible scenarios to make sure that your machine is robust and honest.

   One solution to this exercise is shown on page 466.

1.10.2 The Softdrink Machine

The next exercise is to draw an even more sophisticated Softdrink Machine. This Softdrink Machine gives correct change, allows you to choose and receive your favorite drink, and even allows you to abort your purchase (and get your money back) by pressing the Coin Return button.

1. Start with a copy of The Better Cola Machine. All the states and arcs remain valid for this exercise.

2. In addition to the old inputs, N, D and Q, the Softdrink machine also has the following legal inputs in its alphabet:
   - C, which is what you input when you press the Cola button
   - O, which is what you input when you press the Orange button
   - L, which is what you input when you press the Lemon-Lime button
   - R, which is what you input when you press the Coin Return button

   We arbitrarily choose to view the machine as a generator, so all the input symbols, including input coins and C, O, L and R, should appear on the top of arcs. Output symbols, representing drinks and returned coins, appear on the lower side of arcs, roughly matching our experience of where outputs appear on real soft-drink machines. I.e. we enter coins in a slot toward the top, and the drinks and any change fall out toward the bottom.

3. In addition to the old outputs, N, D and Q (representing change from overpayment), the new Softdrink Machine also has the following possible outputs:
• **COLA**, a multicharacter symbol representing a can of cola
• **ORANGE**, a multicharacter symbol representing a can of orange soda
• **LEMONLIME**, a multicharacter symbol representing a can of lemon-lime soda

The output symbols **COLA**, **ORANGE** and **LEMONLIME** should appear only on the bottom of appropriate arcs.

4. From the final state, entering **C** should cause **COLA** to be returned, and the machine should move back into the Start state (and similarly for **O** and **L**). In other words, there should be an arc leading from the final state to the start state, with **C** on top, and **COLA** on the bottom.

5. From a non-final state, entering **C**, **O** or **L** should have no effect. The machine should accept the input, loop back to the same state, and output nothing.

6. From any state, entering **R** should cause any money already entered to be returned, and the machine should transition back to the start state. When giving change and when performing a coin return, don’t worry about returning the same coins; just return some combination of coins adding up to the amount already entered.

Anticipate and handle all possible input scenarios so that your machine is honest and robust. Test it, by tracing the path through the states and arcs, for at least the following cases:

• You enter “DNNNC”. The machine should return “COLA”.
• You enter “DDNQNO”. The machine should return “QNORANGE”.
• You enter “DDDL”. The machine should return “NLEMONLIME”.
• You enter “DQCC”. The machine should return 20 cents (in some appropriate combination of coins) and a “COLA”.
• You enter “NNNNOO”. The machine should return “ORANGE”.
• You enter “NDR”. The machine should return 15 cents, in some appropriate combination of coins.

1.10.3 Relations and Relatives

Father-in-Law

In Section 1.5.3 on Composition we suggested that the father-in-law relation is the composition of the father and spouse relations. Perhaps this is wrong. When

Charles and Diana were married, Diana’s mother Frances and Diana’s father John, the 8th Earl of Spencer, had divorced. Frances had become Mrs. Raine McCrorquodale. In the meantime she had also been married to Mr. Peter Kydd. While Mr. Kydd is clearly not Prince Charles’ father-in-law, perhaps Mr. McCrorquodale is, by virtue of being the husband of Diana’s mother. In that case Charles has two fathers-in-law.

Using composition and possibly other set operations, give a new definition of father-in-law that gives us the relation \{
<Philip, Diana>, <John, Charles>,
<Raine, Charles>\}, expanding the family tree as shown in Figure 1.29.

Cousin

The inverse of a relation R, denoted \(R^{-1}\), consists of the same pairs as R but in the opposite order. Thus husband is the same relation as wife\(^{-1}\); child is the inverse of parent, and vice versa.

Give a definition of cousin using inverse relations, composition, and possibly other set operations to ensure that no one is his own cousin or the cousin of his brother or sister.

Hint: You may find it convenient to start by defining sibling as the composition of child and parent minus the identity relation. The subtraction of the identity relation is required to satisfy the intuition that no one is his own sibling.

1.10.4 Lowercase/Uppercase Composition

The composition of the network representing \{<“Katze”, “Katze”>\} with the lowercase/uppercase transducer in Figure 1.46 yields the empty relation as the result. Why? Because the string “Katze” contains both uppercase and lowercase letters. Thus it is neither in the upper nor the lower language of the network.

What modification in the network in Figure 1.46 needs to be made if we want the outcome in Figure 1.55 when the “Katze” network comes first in the composition?

![Figure 1.55: New upper-case rule applied to “Katze”](image)

What result do we get if we put the new lowercase/uppercase transducer on top and compose it with a network representing \{<“KATZE”, “KATZE”>\}? Why?