Chapter 2

A Systematic Introduction

2.1 Where We Are

The previous chapter introduced in a gentle informal way some basic concepts of finite-state linguistics:

- A set of strings, a LANGUAGE, can be represented as a simple finite-state network that consists of states and arcs that are labeled by atomic symbols. Each string (= *word*) of the language corresponds to a path in this network.
- A set of ordered pairs of strings, a RELATION, can be represented as a finite-state transducer. The arcs of the transducer are labeled by symbol pairs. Each path of the transducer represents a pair of strings in the relation. The first string of each ordered pair belongs to the UPPER language, the second string belongs to the LOWER language of the relation.
- New languages and relations can be constructed by set operations such as UNION, CONCATENATION, and COMPOSITION. These operations can also be defined for finite-state networks to create a network that encodes the resulting language or relation. Some network operations such as INTERSECTION and SUBTRACTION can be defined only for languages, not for relations.

In this chapter we will examine in greater detail the relationship between languages and relations, their descriptions, and the finite-state automata that encode them. We will try to be precise and clear without being pedantic. We wish to avoid introducing too many formal definitions and irrelevant details. This is a systematic introduction rather than a formal one. We will not present proofs, theorems, or detailed algorithms but will include some references to where they can be found.

2.2 Basic Concepts

Finite-state networks can represent only a subset of all possible languages and relations; that is, only some languages are finite-state languages. One of the funda-

mental results of formal language theory (Kleene, 1956) is the demonstration that finite-state languages are precisely the set of languages that can be described by a REGULAR EXPRESSION.

Figure 2.1 gives a simple example that shows (1) a minimal regular expression, (2) the language it describes, and (3) a network that encodes the language. It illustrates how these notions are related to one another.

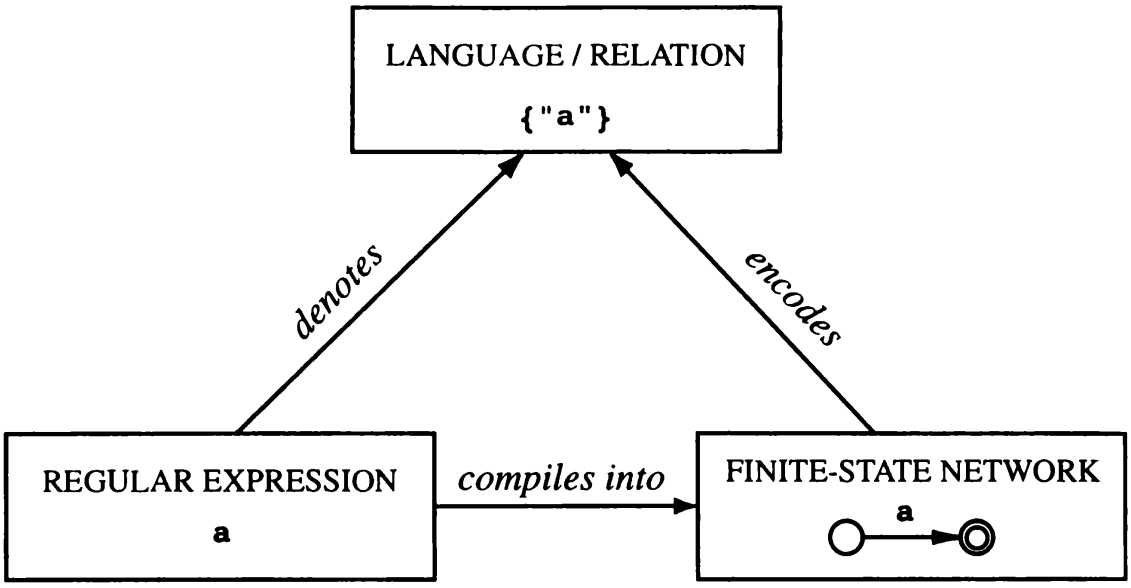


Figure 2.1: Regular Expression – Language – Network.

As Figure 2.1 indicates, a regular expression DENOTES a set of strings (i.e. a language) or a set of ordered string pairs (i.e. a relation). It can be COMPILED INTO a finite-state network that compactly ENCODES the corresponding language or relation that may well be infinite.

The language of regular expressions includes the common set operators of Boolean logic and operators such as concatenation that are specific to strings. It follows from Kleene's theorem that for each of the regular-expression operators for finite-state languages there is a corresponding operation that applies to finite-state networks and produces a network for the resulting language. Any regular expression can in principle be compiled into a single finite-state network.

We can build a finite-state network for a complex language by first constructing a regular expression that describes the language in terms of set operations and by then compiling that regular expression into a network. This is in general easier than constructing a complex network directly, and in fact it is the only practical way for all but the most trivial infinite languages and relations.

We use the term FINITE-STATE NETWORK to cover both simple automata that encode a regular language and transducers that encode a regular relation. A network consists of STATES and ARCS. It has a single designated START STATE and any number (zero or more) of FINAL STATES. In our network diagrams states are represented by circles. By convention in our diagrams, the leftmost circle is the start state; and final states are distinguished by a double circle. Each state is the

origin of zero or more labeled arcs leading to some destination state. The arc labels are either simple symbols or symbol pairs depending on whether the network encodes a language or a relation between two languages. Each string or pair of strings is encoded as a path of arcs that leads from the start state to some final state. In the network of Figure 2.1 there is just one path; it encodes the string "a". Unlike many introductory textbooks, we do not treat simple finite-state automata and transducers as different types of mathematical objects. Our presentation reflects rather closely the data structures in the actual **Xerox** implementation of finite-state networks. We hope it is as precise but more approachable than a rigorous formal introduction in terms of n-tuples of sets and functions (Roche and Schabes, 1997).

2.3 Simple Regular Expressions

We will start with a simple regular-expression language and expand it later in Section 2.4 with more constructions and operators. Even this initial description introduces many more types of regular expression operators than can be found in classical computer science literature (Hopcroft and Ullman, 1979). Because the expressions we use are meant to be typed, using common ASCII text editors on computers, our notation is slightly different from that used in most textbooks. No Greek symbols, just simple ASCII text.

We define the syntax and the semantics of the language in parallel. Many regular-expression operations can be applied both to regular languages and to regular relations. But some operators can be applied only to expressions whose denotation is semantically of the proper type, a language or a relation. In particular, complementation, subtraction and intersection can be applied to all regular languages but only to a subset of regular relations. This issue will be discussed in detail in Section 2.3.3.

However, we are intentionally and systematically not making a distinction between a language and the identity relation on that language. The identity relation is the relation that maps every string to itself. If an operation such as concatenation is applicable to both languages and relations and if one operand denotes a language and the other a relation, we will automatically interpret the former as the identity relation on the language. Thus the result will be a relation as usual.

2.3.1 Definitions

We first introduce some atomic expressions and then construct more complex formulas with brackets, parentheses, braces, and regular-expression operators. In this section we define a family of basic operators: *optionality*, *iteration*, *complementation*, *concatenation*, *containment*, *ignoring*, *union*, *intersection*, *subtraction*, *crossproduct*, *projection*, *reverse*, *inverse*, *composition* and *substitution*. Still more operators will be introduced in Section 2.4.

In the following definitions we use uppercase letters, A, B, etc., as variables

over regular expressions. Lower case letters, a, b, etc., stand for symbols. We shall have more to say about symbols in Section 2.3.6.

Atomic Expressions

- The EPSILON symbol **0** denotes the empty-string language or the corresponding identity relation.
- The ANY symbol **?** denotes the language of all single-symbol strings or the corresponding identity relation. The empty string is not included in ?.
- Any single symbol, **a**, denotes the language that consists of the corresponding string, here "a", or the identity relation on that language.
- Any pair of symbols a : b separated by a colon denotes the relation that consists of the corresponding ordered pair of strings, {<"a", "b">}. We refer to a as the UPPER symbol of the pair a:b and to b as the LOWER symbol.

  A pair of identical symbols, except for the pair ? : ?, is considered to be equivalent to the corresponding single symbol. For example, we do not distinguish between a : a and a in our regular expressions or in our networks.
- The pair ? : ? denotes the relation that maps any symbol to any symbol including itself. The identity relation, denoted by ?, is a subrelation of ? : ? that maps any symbol only to itself. Because ? does not cover the empty string, ? : ? is an example of an EQUAL-LENGTH RELATION: a relation between strings that are of the same length. In this case, length = 1.
- The boundary symbol .#. designates the beginning of a string in the left context and the end of a string in the right context of a restriction or a rule-like replace expression. See Sections 2.4.1 and 2.4.2 for discussion. .#. has no special meaning elsewhere.

Brackets

- [A] denotes the same language or relation as A.
- [] denotes the empty string language, i.e. the language that consists of the empty string. [] is thus equivalent to 0.
- [. .] has a special meaning in replace expressions. See Section 2.4.2 for discussion of dotted brackets.

In order to ensure that complex regular expressions have a unique syntax, we should in principle insist that *all* compound expressions constructed by means of an operator be enclosed in brackets. However, in keeping with common practice, we do not insist on complete bracketing. As we shall see

shortly, some syntactic ambiguities are semantically irrelevant. Bracketing can also be omitted when the intended interpretation can be derived from the rules of operator precedence presented in Section 2.3.5. However, we often add brackets just for clarity, to mark off the regular expression from the surrounding text.

Be careful to distinguish square brackets from round parentheses. Parentheses indicate optionality.

Optionality

- `(A)` denotes the union of the language or relation `A` with the empty string language. I.e. `(A)` is equivalent to `[A | 0]`.

Restriction: None. `A` can be any regular expression.

Iteration

- `A+` denotes the concatenation of `A` with itself one or more times. The `+` operator is called *Kleene-plus* or *sigma-plus*.
- `A*` denotes the union of `A+` with the empty string language. In other words, `A*` denotes the concatenation of `A` with itself zero or more times. The `*` operator is known as *Kleene-star* or *sigma-star*. `?*` denotes the UNIVERSAL LANGUAGE, the set of all strings of any length including zero. `[?:?]*`, equivalent to the notation `?:?*`, denotes the UNIVERSAL EQUAL-LENGTH RELATION, the mapping from any string to any string of the same length.

Restriction: None. `A` can be any regular expression.

Complementation

- `~A` denotes the complement language of `A`, that is, the set of all strings that are not in the language `A`. The complementation operator `~` is also called *negation*. `~A` is equivalent to `[?* - A]`, i.e. to the Universal Language minus all the strings in `A`.
- `\A` denotes the term complement language, that is, the set of all single-symbol strings that are not in `A`. The `\` operator is also called *term negation*. `\A` is equivalent to `[? - A]`.

Restriction: In both types of complementation, `A` must denote a language. The complementation operation is not defined for relations. For more about this topic see Section 2.3.3.

Concatenation

- `[A B]`, equivalent to the notation `A B`, denotes the concatenation of the two languages or relations. The white space between regular expressions serves as the concatenation operator. Concatenation is an ASSOCIATIVE operation, that is, the notation `[A [B C]]` is equivalent to `[[A B] C]`. Because the order in which multiple concatenations are performed makes no difference for the result, we can ignore it and write simply `[A B C]` omitting the inner brackets. This is an example where a syntactic ambiguity is semantically irrelevant. The same applies to all associative operations.
- `{s}`, where `s` is some string of alphanumeric symbols "`abc...`" denotes the concatenation of the corresponding single-character symbols `[a b c ...]`. For example, `{word}` is equivalent to `[w o r d]`.
- `A^n` denotes the n-ary concatenation of `A` with itself. `n` must be an integer. For example, `a^3` is equivalent to `[a a a]`.
- `A^<n` denotes less than `n` concatenations of `A`, including the empty string. For example, `a^<3` denotes the language containing "" (the empty string), "a" and "aa".
- `A^>n` denotes more than `n` concatenations of `A`. For example, `a^>3` is equivalent to `[a a a a+]`.
- `A^{i,k}` denotes from `i` to `k` concatenations of `A`. For example, the language `a^{1,3}` contains the strings "a", "aa" and "aaa".

Restriction: None. `A` and `B` can be any regular expressions.

Containment

- `$A` denotes the language or relation obtained by concatenating the universal language both as a prefix and as a suffix to `A`. For example, `$[a b]` denotes the set of strings such as "cabbage" that contain at least one instance of "ab" somewhere. The notation `$A` is equivalent to `[?* A ?*]`.

Restriction: None. `A` can be any regular expression.

Ignoring

- `[A / B]` denotes the language or relation obtained from `A` by splicing in `B*` everywhere within the strings of `A`. For example, `[[a b] / x]` denotes the set of strings such as "xxxxaxxbx" that distort "ab" by arbitrary insertions of "x". Intuitively, `[A / B]` denotes the language or relation `A` ignoring arbitrary bursts of "noise" from `B`.

- `[A ./. B]` denotes the language or relation obtained from `A` by splicing in `B*` everywhere in the *inside* of the elements of `A` but not at the edges. For example, `[[a b] ./. x]` contains strings such as "axxxb" but not "xab" or "axbx".

Restriction: None. `A` and `B` can be any regular expressions.

Union

- `[A | B]` denotes the union of the two languages or relations. The union operator `|` is also called DISJUNCTION. The union operation is associative and also COMMUTATIVE; that is, `[A | B]` and `[B | A]` denote the same language or relation.

Restriction: None. `A` and `B` can be any regular expressions

Intersection

- `[A & B]` denotes the intersection of the two languages. The intersection operator `&` is also called CONJUNCTION. The operation is associative and commutative. Intersection can be expressed in terms of complementation and union. `[A & B]` and `~[~A | ~B]` are equivalent.

Restriction: `A` and `B` must denote languages or equal-length relations. Relations cannot, in general, be intersected. We explain the reason in Section 2.3.3.

Subtraction

- `[A - B]` denotes the language of all the strings in `A` that are not members of `B`. `[A - B]` is equivalent to `[A & ~B]`. Subtraction is neither associative nor commutative.

  The two varieties of complementation introduced above could be defined in terms of subtraction because `~A` is equivalent to `[?* - A]` and `\A` is equivalent to `[? - A]`.

Restriction: `A` and `B` should denote languages or equal-length relations. Relations cannot, in general, be subtracted.

Crossproduct

- `[A .x. B]` denotes the relation that pairs every string of language `A` with every string of language `B`. Here `A` is called the *upper* language and `B` the *lower* language of the relation.

  The expression `[?* .x. ?*]` denotes the *universal relation*, the mapping from any string to any string, including the empty string.

- `[ [A] : [B] ]` denotes the relation that pairs every string of language `A` with every string of language `B`. Here `A` is called the *upper* language and `B` the *lower* language of the relation. Because a pair such as `a:b` denotes the relation between the corresponding strings, `[a .x. b]` and `a:b` are obviously equivalent expressions.

  The `.x.` and the `:` are both general-purpose crossproduct operators, but they differ widely in precedence (see Section 2.3.5), with `:` having very high precedence and `.x.` very low precedence. In practice, it is most important to remember that `.x.` has lower precedence than concatenation, so that `[c a t .x. c h a t]` denotes the same relation as `[[c a t] .x. [c h a t]]`; in contrast, the `:` has higher precedence than concatenation, such that `[c a t : c h a t]` is equivalent to `[c a [t:c] h a t]`.

Restriction: `A` and `B` must denote languages, not relations. This restriction is inherent to how crossproduct is defined.

Projection

Recall that a relation relates two regular languages, called the upper language and the lower language.

- `A.u` denotes the upper language of the relation of `A`,
- `A.l` denotes the corresponding lower language of `A`.

Restriction: None. If `A` denotes a language, `A` is interpreted as the identity relation on `A`. In that case, `A.u` and `A.l` denote the same language as `A`.

Reverse

- `A.r` denotes the reverse of the language or relation `A`. For example, if `A` contains <"abc", "xy">, the reverse relation `A.r` contains <"cba", "yx">. Be careful to distinguish REVERSE and INVERSE. Reverse exchanges left and right; inverse exchanges the upper and lower sides of a relation.

Restriction: None. `A` can be any regular expression.

Inverse

- `A.i` denotes the inverse of the relation `A`. For example, if `A` contains <"abc", "xy">, the inverse relation `A.i` contains <"xy", "abc">. Be careful to distinguish *inverse* and *reverse*. Inverse exchanges the upper and lower sides of a relation; reverse exchanges left and right.

Restriction: None. If `A` denotes a language, in `A.i` it is interpreted as an identity relation. In this case `A.i` is indistinguishable from `A`.

Composition

- `[A .o. B]` denotes the composition of the relation `A` with the relation `B`. If `A` contains the string pair <*x*, *y*> and `B` contains <*y*, *z*>, the composite relation `[A .o. B]` contains the string pair <*x*, *z*>.
  Composition is associative but not commutative. We can write `[A .o. B .o. C]` because the result is the same relation regardless of whether we interpret it as `[A .o. [B .o. C]]` or `[[A .o. B] .o. C]`.

Restriction: None. If `A` or `B` denotes a language, in `[A .o. B]` it is interpreted as the corresponding identity relation. If both `A` and `B` denote languages `[A .o. B]` and `[A & B]` are indistinguishable; composition and intersection yield the same result in this case.

Substitution

- `` `[[A], s, L] `` denotes the language or relation derived from `A` by substituting every symbol *x* in the list `L` for every occurrence of the symbol `s`. For example, `[[a -> b], b, x y z]` denotes the same relation as `[a -> [x | y | z]`. If the list `L` is empty, all the strings and pairs of strings containing the symbol `s` are eliminated. For example, `` `[[a | b:c | c], b, ] `` is equivalent to `[a | c]`.

Restriction: All the members of `L` (if any) and the target of the substitution, the symbol `s`, must be simple symbols, not symbol-pairs.

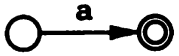
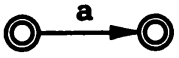
Expression	Language / Relation	Network
`~[?*]`	{ }	
`[]`	{“”}	
`a`	{“a”}	
`(a)`	{“”, “a”}	

Table 2.1: Minimal Languages

2.3.2 Examples

In order to understand the semantics of regular expressions it is useful to look at some simple examples in conjunction with the corresponding language or relation and the network that encodes it. We will start with the minimal examples in

Table 2.1 and proceed to cover some of the operators introduced in the previous section.

Because we have not introduced a special symbol for the EMPTY LANGUAGE, i.e. the language that contains no strings, not even the empty string, we have to designate it by some complex expression such as ~ [?*] that denotes the complement of the universal language. It compiles into a network with a single non-final state with no arcs. The empty-string language, i.e. the language that contains only the empty string, denoted by 0 or by [], corresponds to a network with a single final state with no arcs. Note that [a] can also be interpreted as the identity relation { < "a", "a" > }. The network representation of the ANY symbol, ?, is a complex issue. We will discuss it later in Section 2.3.4.

Table 2.2 illustrates the difference between the two iteration operators Kleene-star and Kleene-plus. Because the networks in Table 2.2 are cyclic, they contain an infinite number of paths. Looking at the middle column it is evident that a* is equivalent to (a+) and to [a+ | 0]. This is not as easily deduced from the corresponding minimal networks.

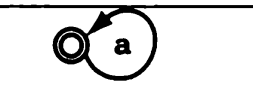
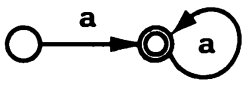
Expression	Language / Relation	Network
a*	{"", "a", "aa", ...}	
a+	{"a", "aa", ...}	

Table 2.2: Iteration

As the shape of the network for a+ in Table 2.2 suggests, a+ is equivalent to the concatenation [a a*]. Table 2.3 shows more examples of concatenation.

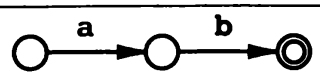
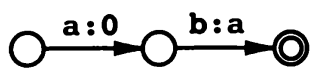
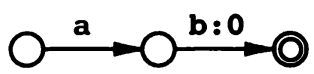
Expression	Language / Relation	Network
a 0 b	{"ab"}	
a:0 b:a	{<"ab", "a">}	
a b:0	{<"ab", "a">}	

Table 2.3: Concatenation

The concatenation of two languages yields a language, the concatenation of two relations yields a relation. As the last example in Table 2.3 shows, we system-

atically ignore the difference between a LANGUAGE and its IDENTITY RELATION. Thus `[a]` is interpreted as the identity relation { < "a", "a" > } when it is concatenated, unioned, or composed with a relation.

The last two regular expressions in Table 2.3 denote the same relation but they do not compile into the same network. This is is an important point. There is no general method for deriving a unique canonical representation for a relation that pairs strings of unequal length. Two networks may encode the same relation but there is no general algorithm for deciding whether they do. In this respect regular relations are computationally more difficult to handle than regular languages. Every regular language has a unique minimal encoding as a network, but some relations do not. We will come back to this point later.

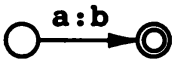
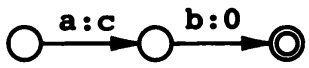
Expression	Language / Relation	Network
`a .x. b`	{<"a", "b">}	
`[a b] .x. c`	{<"ab", "c">}	

Table 2.4: Crossproduct

The crossproduct of two languages may also produce pairs of strings that are of different length, as in the second example in Table 2.4. The pair of strings <"ab", "c"> could be encoded in a network in other ways, for example, by a path in which the successive labels are `a:0` and `b:c` instead of `a:c` and `b:0` as in Table 2.4. However, because it is convenient to have a unique encoding for crossproduct relations, the **Xerox** compiler pairs the strings from left to right, symbol by symbol, and introduces one-sided epsilon pairs only at the right end of the path if needed. This is an arbitrary choice.

The final set of examples in Table 2.5 illustrates some simple cases of composition. In the last example, we interpret `b` as the identity relation because composition is inherently an operation on relations and not on languages.

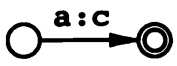
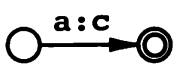
Expression	Language / Relation	Network
`a:b .o. b:c`	{<"a", "c">}	
`a:b .o. b .o. b:c`	{<"a", "c">}	

Table 2.5: Composition

Regular expressions were originally invented as a metalanguage to describe languages. The formalism was subsequently extended to describe relations. But as we have noted in many places, some operators can be applied only to a subset of regular expressions. Let us now review this issue in more detail.

2.3.3 Closure

A set operation such as union has a corresponding operation on finite-state networks only if the set of regular relations and languages is CLOSED under that operation. Closure means that if the sets to which the operation is applied are regular, the result is also regular, that is, encodable as a finite-state network.

Regular languages are closed with respect to all the common set operations including intersection, subtraction and complementation (= negation). This follows directly from Kleene's proof. Regular relations are closed under concatenation, iteration, union, and composition but not, in general, under intersection, complementation or subtraction (Kaplan and Kay, 1994; Roche and Schabes, 1997).

Figure 2.2: Networks for Two Regular Relations

Kaplan and Kay give a simple example of a case in which the intersection of two finite-state relations is not a finite-state relation. Consider two regular expressions and the corresponding networks in Figure 2.2. P is the relation that maps strings of any number of **a**s into strings of the same number of **b**s followed by zero or more **c**s. Q is the relation that maps strings of any number of **a**s into strings of the same number of **c**s preceded by zero or more **b**s.

Table 2.6 shows the corresponding string-to-string relations and their intersection. The left side of Table 2.6 is a partial enumeration of the P relation. The right side partially enumerates the Q relation. The middle section of the table contains the intersection of the two relations, that is, the pairs they have in common.

It is easy to see that the intersection contains only pairs that have on their lower side (i.e. as the second component) a string that contains some number of **b**s followed by exactly the same number of **c**s.

The lower-side language, $b^n c^n$, is not a finite-state language but rather a CONTEXT-FREE LANGUAGE, generated by a phrase-structure grammar that crucially depends on center-embedding: S $\rightarrow$ ε, S $\rightarrow$ b S c. Consequently it cannot be encoded by a finite-state network (Hopcroft and Ullman, 1979). The same holds

P Regular	P & Q Not Regular	Q Regular
<"", "">	<"", "">	<"", "">
<"", c>		<"", b>
<"", cc>		<"", bb>
...		...
<a, b>		<a, c>
<a, bc>	<a, bc>	<a, bc>
<a, bcc>		<a, bbc>
<a, bccc>		<a, bbbc>
...		...
<aa, bb>		<aa, cc>
<aa, bbc>		<aa, bcc>
<aa, bbcc>	<aa, bbcc>	<aa, bbcc>
<aa, bbccc>		<aa, bbbcc>
...		...

Table 2.6: Non-Regular Intersection of P and Q

of course for any relation involving this language. No operation on the networks in Figure 2.2 can yield a finite-state transducer for the intersection of P and Q; such a transducer does not exist.

The relations P and Q both have the property that a string in one language corresponds to infinitely many strings in the other language because of the iterated 0:b and 0:c pairs. It is this characteristic, the presence of "one-sided epsilon loops" in Figure 2.2 that makes it possible that their intersection be not regular.

From the fact that regular relations are not closed under intersection it follows immediately that they are not closed under complementation either. Intersection can be defined in terms of complementation and union. If regular relations were closed under complementation, the same would be true of intersection. It also follows that regular relations are not closed under the subtraction operation, definable by means of intersection and complementation.

The closure properties of regular languages and relations are summarized in Table 2.7 for the most common operations.

Although regular relations are not in general closed under intersection, a subset of regular relations have regular intersections. In particular, equal-length relations, relations between pairs of strings that are of the same length, are closed under intersection, subtraction and complementation. Such relations can be encoded by a transducer that does not contain any epsilon symbols.[1]

[1]This fact is important for us because it is the formal foundation of the two-

Operation	Regular Languages	Regular Relations
union	yes	yes
concatenation	yes	yes
iteration	yes	yes
reversal	yes	yes
intersection	yes	no
subtraction	yes	no
complementation	yes	no
composition	(not applicable)	yes
inversion	(not applicable)	yes

Table 2.7: Closure Properties

The **Xerox** calculus allows intersection to apply to all simple automata and to transducers that do not contain any one-sided epsilon pairs. The test is more restrictive than it should be in principle because the presence of one-sided epsilons in a transducer does not necessarily indicate that the relation it encodes is of the type that could yield a non-regular intersection.

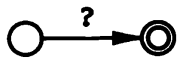
2.3.4 The ANY Symbol

So far we have given examples of regular expressions and the corresponding networks for most of the operators introduced in Section 2.3.1. One notable exception is complementation. We will come to that shortly but first we need to understand the semantics of the ANY symbol, `?`, introduced in the beginning of Section 2.3.1 as one of the atomic expressions.

Let us recall that the term complement of a language `A`, denoted by `\A`, is the union of the single-symbol strings that are not in `A`. For example, the language `[\a]` contains "`b`", "`c`", ..."`z`". In fact `[\a]` is an infinite language because the set of atomic symbols is in principle unbounded. Our symbol alphabet is not restricted to the 26 letters of the English alphabet or to the 256 ASCII characters.

For this reason we provide a special regular-expression symbol, `?`, to represent the infinite set of symbols in some yet unknown alphabet. It is called the ANY symbol. The regular expression `?` denotes the language of all single-symbol strings. Note that this set does not include the empty string. Because we do not make a distinction between a language and an identity relation, `?` can also be interpreted as the relation that maps any symbol into itself. The corresponding network is obviously the one in Figure 2.3. But note the annotation `Sigma: {?}`. We will explain it shortly.

level rule formalism called **twolc**, which is included on the CD-ROM and described on `http://www.fsmbook.com/`. Transducers compiled from two-level rules can be intersected because the 0 symbol is treated as an ordinary symbol in the rule formalism and not as an empty string.



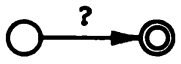
Sigma: {?}

Figure 2.3: The Language of All Single-Symbol Strings [?].

The correspondence between the regular expression ? and the network that encodes it in Figure 2.3 is deceptively straightforward. In fact the symbol ? in the regular expression does not have exactly the same interpretation that the arc label ? has in the network.

In the regular expression, ? stands for ANY symbol; the corresponding arc label ? represents any UNKNOWN symbol. In the case at hand the difference is subtle. Because the alphabet of known symbols in this network contains just ?, the arc labeled ? does represent any symbol whatsoever in this case. We call the known alphabet of a network its SIGMA. Thus the annotation Sigma: {?} in Figure 2.3 means that the sigma is empty except for ?.

The difference between ? as a regular expression symbol and ? as an arc label in networks is an unfortunate source of confusion for almost all users of the **Xerox** regular-expression calculus. Using two different symbols, say ? for ANY and ¿ for UNKNOWN would make it explicit. However, employing two different symbols suggests that ? and ¿ could both appear in regular expressions and as arc labels. This would be misleading. The **Xerox** calculus is designed and implemented in such a way that ANY is exclusively a regular expression concept and the concept of UNKNOWN is relevant only for finite-state networks. Because of this fact, we have chosen to "overload" the ? sign. We interpret ? as the ANY symbol in regular expressions and we also use ? to display the UNKNOWN symbol in network diagrams.



Sigma: {?, a}

Figure 2.4: The Language [\a]

An expression that in some way involves complementation typically compiles into a network that contains an instance of the unknown symbol. In such cases it is crucial to know the sigma alphabet because it cannot in general be deduced from the arcs and the labels of the network. For example, the language [\a], consisting of all single-symbol strings other than "a", is encoded by the network in Figure 2.4. In the network of Figure 2.4, ? does not include "a", in the network of Figure 2.3 it does. [\a] is equivalent to [? - a].

The sigma alphabet includes every non-epsilon symbol that appears in the net-

work either by itself or as a component of a symbol pair. As we see here, the sigma alphabet may also contain symbols that do not appear in the network but are "known" because they were present in another network from which the current one was derived by some operation that eliminated all the arcs where these symbols occurred.

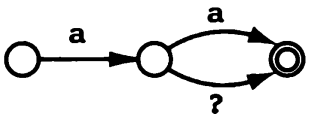
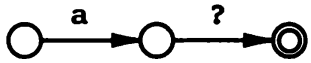
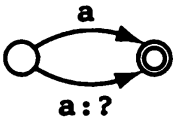
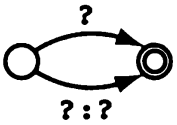
Regular Expression	Network	Sigma
`a ?`		{?, a}
`a \a`		{?, a}
`a:?`		{?, a}
`?:?`		{?}
`a \?`		{}

Table 2.8: Expressions with ?

Complex regular expressions that contain ? as a subexpression must be compiled carefully so that the sigma alphabet is correct and all the required arcs are present in the resulting network; this was a challenge for the algorithm writers. Table 2.8 illustrates the issue with a few examples. The first network in Table 2.8 is the result of concatenating the `[a]` network shown in Table 2.1 with the `[?]` network shown in Figure 2.3. The original networks both contain just one arc but the result of the concatenation has three arcs. The compiler adds a redundant "a" arc to the `[?]` network to make explicit the fact that the string "a" is included in the language. This expansion is necessary because the sigma of the resulting `[a ?]` network in Table 2.8 contains a. Consequently the networks for `[a ?]` and `[a \a]` correctly encode the fact that the language `[a ?]` includes the string "aa" but the language `[a \a]` does not.

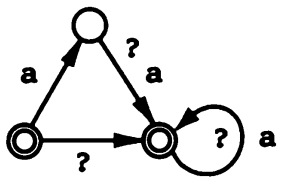
For the same reason, the network for `a:?` contains an explicit a-arc. Let us recall that the symbol pair `a:?` denotes the crossproduct `[a .x. ?]`. Because "a" is included in the language `[?]`, the crossproduct relation includes the pair <"a", "a"> along with all the pairings of "a" with other single-symbol strings. This identity pair is encoded in the network by the a-arc; the arc labeled `a:?` covers all the other pairs.

The network for the expression `?:?` illustrates another subtle fact about the

interpretation of the *unknown* symbol. The relation denoted by the regular expression `?:?` maps any symbol to any symbol including itself. That is, the relation `?:?` includes the identity relation denoted by the expression `?`. In the network for the relation `?:?`, the identity part of the relation is represented by the arc labeled `?`, the non-identity mapping by the arc labeled `?:?`.

The term complement `[\?]` denotes the empty language. Concatenation with an empty language always yields the empty language. Consequently the concatenation `[a \?]` is also empty, as the last example in Table 2.8 shows.

Figure 2.5 gives an example of a network compiled from an expression that contains ~, the other complementation operator. Here we make use of the graphic convention that lets us represent any number of arcs that share the same origin and destination by a single multiply labeled transition. In fact the network in Figure 2.5 has six arcs of which only four are explicitly shown.



Sigma: {?, a}

Figure 2.5: The Language `[~a]`

As the comparison between Figure 2.4 and Figure 2.5 shows, `[\a]` and `[~a]` denote very different languages. The set of single-symbol strings other than "a", `[\a]`, is included in the language `[~a]`. In addition the latter language contains the empty string (the start state is final), and any string whatever whose length is greater than one, for example, "aa". `[~a]` is equivalent to `[?* - a]`.

The expression `?*` denotes the UNIVERSAL LANGUAGE, the set of all strings of any length including the empty string. The corresponding network is shown in Figure 2.6. We may also interpret `?*` as the UNIVERSAL IDENTITY RELATION: every string paired with itself.

Sigma: {?}

Figure 2.6: The Universal Language/Identity Relation `?*`

If we added the symbols **a**, **b**, and **c** to the sigma of the network in Figure 2.6, the encoded language would be `[\[a | b | c]]*`.

The expression `[?:?]*` denotes the universal equal-length relation that pairs every string with all strings that have the same length. The corresponding network

is shown in Figure 2.7. For the sake of clarity we draw here the two arcs explicitly instead of abbreviating the picture into a single arc labeled with two symbols.

Figure 2.7: The Universal Equal-Length Relation `[?:?]*`

As in the network for the relation `?:?` in Table 2.8, the arc labeled `?` in Figure 2.7 represents the mapping of any unknown symbol into the same unknown symbol. The arc labeled `?:?` represents the mapping of any unknown symbol into a different unknown symbol. Both arcs are needed because the application of the network (in either direction) to a string such as "a" must yield two results: "a" itself and `?` representing all other possibilities.

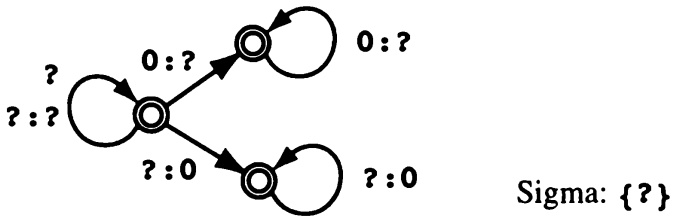


Figure 2.8: The Universal Relation `[?* .x. ?*]`

The expression `[?* .x. ?*]` in Figure 2.8 is another interesting special case. It denotes the promiscuous UNIVERSAL RELATION that maps any string to any other string including itself. The `?` loop in the start state pairs any string with itself; the `?:?` loop pairs any string with any other string of equal length. The `0:?` and `?:0` arcs allow a string to be paired with another string that does not have the same length. Because the relation `[?* .x. ?*]` pairs any string with any string, the expression `[A .o. [?* .x. ?*] .o. B]` is equivalent to the crossproduct of the "outer" languages of the `A` and `B` relations `[A.u .x. B.l]`.

2.3.5 Operator Precedence

Like conventional arithmetic expressions, complex regular expressions can often be simplified by taking advantage of a convention that ranks the operators in a precedence hierarchy. For example, `[[\a] [b]*]` may be expressed more concisely as `[\a b*]` because by convention the unary operators, term complement and Kleene star, "bind more tightly" to their component expressions than concatenation. Similarly, the inner brackets in `[a | [b c]]` are unnecessary because

concatenation takes precedence over union; `[a | b c]` denotes the same language. In turn, union has precedence over crossproduct, allowing us to simplify `[a .x. [b | c]]` to `[a .x. b | c]`. The ranking of all the operators discussed in the previous sections is summarized in Table 2.9, including the restriction and replacement operators that will be introduced in Section 2.4.

Type	Operators
Crossproduct	:
Prefix	~, \, $
Suffix	+, *, ^, .r, .u, .l, .i
Ignoring	/
Concatenation	(whitespace)
Boolean	|, &, -
Restriction and Replacement	=>, ->
Crossproduct and Composition	.x., .o.

Table 2.9: Precedence Ranking from High to Low

Unbracketed expressions with operators that have the same precedence are disambiguated from left to right. Consequently, `[a | b & b*]` is interpreted as `[[a | b] & b*]` and not as `[a | [b & b*]]`. For the sake of clarity, we advise using brackets in such cases even if the ambiguity would be resolved in the intended way by the left-to-right ordering.

2.3.6 More About Symbols

In all of our examples only ordinary single letters have been used as symbols. In this section we discuss briefly how other types of symbols may be represented.

Because **0** and **?** have a special meaning in regular expressions, we need to provide a way to use them as ordinary symbols. The same applies to the symbols used as regular expression operators: |, &, -, +, *, etc.

Special Symbols

The **Xerox** regular-expression compiler provides two ways to avoid the special interpretation of a symbol: prefixing the symbol with % or enclosing it in double quotes. For example, `%0` denotes the string "0", containing the literal zero symbol, rather than the epsilon (empty string); `%|` denotes the vertical bar itself, as opposed to the union operator |. The ordinary percent sign may be expressed as `%%`. Enclosure in double quotes has the same effect: `"0"`, `"?"`, and `"%"` are interpreted as ordinary letters.

On the other hand, certain other strings receive a special interpretation within a doubly quoted string. For example, `"\n"` is interpreted as the newline symbol,

`"\t"` as a tab, etc., following C conventions. The backslash is also used as a prefix in octal and hexadecimal numbers: `"\101"` and `"\x41"` are alternate ways to specify the symbol A. Inside double quotes, the prefix `\u` followed by four hexadecimal numbers encodes a 16-bit Unicode character; e.g. `"\u0633"` is the Arabic *siin* character.

Xerox networks can store and manipulate 16-bit Unicode characters, but until Unicode-capable word processors and operating systems become generally available, input and output of such characters will be awkward.

Multicharacter Symbols

All the examples given so far involve symbols that consist of a single character. The **Xerox** calculus also admits multicharacter symbols such as **+Noun** or **[Noun]**. In a regular expression these would have to specified as `"+Noun"` or `%+Noun`, or `" [Noun] "` or `% [Noun%]`, to escape the special interpretation of the plus sign and the brackets. The inclusion of special punctuation symbols as part of the spelling of tag-like multicharacter symbols is a long-standing **Xerox** convention, not a requirement of the **xfst** regular-expression language per se.

In linguistic applications, morphological and syntactic tags that convey information about part-of-speech, tense, mood, number, gender, etc. are typically represented by multicharacter symbols. It is often convenient to deal with HTML, SGML, and XML tags as atomic symbols rather than as a concatenation of single-character strings.

Multicharacter symbols are treated as atomic entities by the regular-expression compiler; that is, the multicharacter symbol **+Noun** is stored and manipulated just like **a**, **b** and **c**. For example, the sigma alphabet of the network compiled from `[{cat} "+Noun":0]` consists of the symbols **a**, **c**, **t** and **+Noun**.

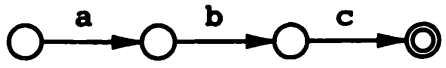
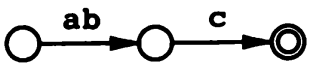
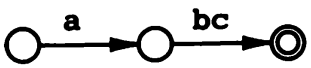
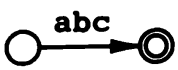
Expression	Network	Sigma
a b c		{a, b, c}
ab c		{ab, c}
a bc		{a, bc}
abc		{abc}

Table 2.10: Alternative Representations of {"abc"}

Because of multicharacter symbols the correspondence between a regular expression and the language it denotes is less direct than we have indicated. There

can be multiple regular expressions that denote what could be seen as the same set of strings but compile into different networks because the strings are tokenized differently. See Table 2.10.

From the point of view of the finite-state calculus, these networks are not equivalent. Their intersection is the network for the empty set. To avoid confusion it is best to represent ordinary strings only as a sequence of single character symbols. The three other alternatives in Table 2.10 should be avoided although they are accepted by the regular expression compiler. The purpose of the curly-brace notation for concatenation (Section 2.3.1) is to facilitate representing words in the preferred format. We can write `{abc}` instead of `[a b c]` to tell the compiler to break or "explode" the string into a sequence of single-character symbols.

Tokenization of Input Strings into Symbols

The APPLY algorithms for finding the path or paths for a string in a network must take into account the network's sigma alphabet and tokenize the input string into symbols accordingly. If the alphabet contains multicharacter symbols, there may not be a unique tokenization in all cases. For example, it might be the case that "+Noun" could be tokenized either as `<+,N,o,u,n>` or as a single symbol. The tokenizer resolves the ambiguity by processing the input string from left to right and by selecting at each point the longest matching symbol in the sigma of the network being applied. The multicharacter symbol always has precedence. In order to avoid tokenization problems, multicharacter symbols should not be confusable with the ordinary alphabet. To help avoid such confusion, a useful practical convention is that every multicharacter symbol should include some special character, or be surrounded by some special characters, that do not occur in ordinary alphabetic strings: `"+Noun"`, `" [Noun] "`, `"^TOKEN"`, `"<TABLE>"`, etc. For more on tokenization of input strings, see Sections 3.6.3 and 6.3.4.

2.4 Complex Regular Expressions

The set of regular-expression operators introduced in the preceding section is somewhat redundant because some of the operators could easily be defined in terms of others. For example, there is no particular need for the CONTAINS operator `$` since the expression `[?* A ?*]` denotes the same language as `$A`. Its purpose is to allow this commonly used construction to be expressed more concisely.

In this section we will discuss regular expressions for RESTRICTION and REPLACEMENT that are more complex than those covered so far. These rule-like expressions are specific to the **xfst** regular-expression compiler described in Chapter 3. Like the CONTAINS operator, these new constructs are definable in terms of more primitive expressions. Thus they do not extend the descriptive power of the regular-expression language but provide a higher level of abstraction that makes it easier for human users to define complex languages and relations.

2.4.1 Restriction

The restriction operator (Koskenniemi, 1983) is one of the two fundamental operators in the traditional two-level calculus.[2] It is also used in the **xfst** regular-expression calculus.

Restriction

- `[A => L _ R]` denotes the language of strings that have the property that any string from `A` that occurs as a substring is immediately preceded by some string from `L` and immediately followed by some string from `R`. We call `L` and `R` here the LEFT and the RIGHT context of `A`, respectively. For example, `[a => b _ c]` includes all strings that contain no occurrence of "a" and all strings like "back-to-back" that completely satisfy the condition, but no strings such as "cab" or "pack".
- `[A => L1 _ R1, L2 _ R2]` denotes the language in which every instance of `A` is surrounded either by a pair of strings from `L1` and `R1` or by a pair of strings from `L2` and `R2`. The list of allowed contexts, separated by commas, may be arbitrarily long.

Restriction: All components, `A`, `L`, `R`, etc. must denote regular languages, not relations.

In **xfst** restrictions, all components must denote regular languages. Expressions such `[a:b => c:d _ e:f]` are invalid in **xfst**.[3]

The restriction expressed by `[A => L _ R]` is that the substrings in A must appear in the context `L _ R`. The same constraint could also be coded in more primitive terms by means of complementation, concatenation, iteration and union. The expression `[~[[~[?* b] a ?*] | [?* a ~[c ?*]]]]` denotes the same language as `[a => b _ c]`, but the latter expression is obviously more convenient and perspicuous.

The outer edges of the left and the right context extend implicitly to infinity. That is, `L _ R` is compiled as `[?* L] _ [R ?*]`. A restriction with a completely empty context such as `[A => _ ]` denotes the universal language as it places no constraint on anything. Another consequence of the implicit extension is that an optional or a negative component at the outer edge of a context is vacuous. An expression such `[A => (b) _ ~c]` also denotes the universal language because `[?* (b)]` and `[~c ?*]` are both equivalent to `?*`. The reason is that

[2]The restriction operator was used in Koskenniemi's "two-level" rules and is included in the **twolc** language. The **twolc** compiler is included on the CD-ROM and described on `http://www.fsmbook.com/`.

[3]Two-level expressions are allowed in **twolc** restrictions because the two-level calculus actually treats symbol pairs as simple atomic symbols. In the **xfst** regular-expression language symbol pairs denote relations.

(b) and ~c both include the empty string. The concatenation between the universal language and any language that includes the empty string reduces to the universal language.

To refer to the beginning or the end of a string in a context specification, we need an explicit marker. The boundary symbol .#. indicates the beginning of a string in the left context and the end of a string in the right context. It has no special meaning anywhere else. The boundary symbol can be concatenated, unioned, etc. with other regular expressions like an ordinary symbol. For example, [a => b _ c | .#.] requires that **a** be followed by **c** or by the end of the string. As Figure 2.9 shows, the boundary marker is compiled away and does not appear in the resulting network. [4]

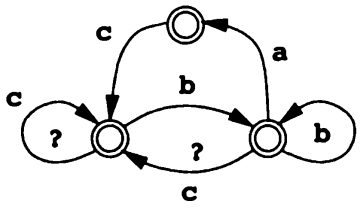


Figure 2.9: The Language a => b _ c | .#.

It is easy to see that the network in Figure 2.9 encodes a language in which every occurrence of **a** is immediately preceded by a **b**, and that the string either terminates at the **a** or continues with a **c**.

With the help of the boundary symbol we can specify contexts that include the empty string. For example, [a => .#. ~[b ?*] _] expresses the constraint that **a** cannot be preceded by a string that starts with **b**. The set of permissible left-context strings, ~[b ?*], includes the empty string. In order to restrict **a** in the intended way, the left context must start with the boundary symbol. Without it the restriction is vacuous: [a => ~[b ?*] _] denotes the universal language ?* where **a** may occur freely anywhere.

Restrictions with alternative contexts such as [a => b _ c, d _ e] could also be expressed as an iterated union of elementary restrictions: [[a => b _ c] | [a => d _ e]]* denotes the same language. The Kleene star is needed here because the language must include not only strings like "bac" and "dae" but also strings like "bacdae" in which different instances of **a** are sanctioned by different contexts.

[4]The sigma alphabet of the network in Figure 2.9 contains only symbols that are present in the network: a, b, c and the unknown symbol. If there are no "hidden" symbols in the sigma alphabet, we generally do not list the sigma explicitly in our diagrams.

2.4.2 Replacement

This section introduces a large set of rule-like regular expressions for denoting complex string-to-string relations. Replacement expressions describe strings of one language in terms of how they differ from the strings of another language. Because the differences may depend on contexts and other parameters, the syntax of replacement expressions involves many operators and special constructs.

The family of replace operators is specific to the **Xerox** regular-expression calculus. In the original version, developed by Ronald M. Kaplan and Martin Kay in the early 1980s (Kaplan and Kay, 1981; Kaplan and Kay, 1994), the goal was to model the application of phonological rewrite rules by finite-state transducers. Although the idea had been known before (Johnson, 1972), Kaplan and Kay were the first to present a compilation algorithm for rewrite rules. The notation introduced in this section is based on Kaplan and Kay's work with extensions introduced by researchers at XRCE (Karttunen, 1995; Karttunen and Kempe, 1995; Karttunen, 1996; Kempe and Karttunen, 1996).

We introduce the main types of replacement expressions first in their unconditional version and then go on to discuss how the operation can be constrained by context and by other conditions.

Unconditional Replacement

A very simple replacement of one string by another can be described as a crossproduct relation. For example, `[a b c .x. d e]` maps the string "abc" to "de" and vice versa. The upper and the lower languages of this relation consist of a single string. In contrast, the replacement expressions introduced in this section generally denote infinite relations.

Simple Replacement

- `[A -> B]` denotes the relation in which each string of the universal language (the upper language) is paired with all strings that are identical to it in every respect except that every instance of A that occurs as a substring in the original upper-language string is represented by a string from `B`.
- `[A <- B]` denotes the inverse of `B -> A`.
- `[A (->) B]` denotes an optional replacement, that is, the union of `[A -> B]` with the identity relation on `A`.
- `[A (<-) B]` is the optional version of `A <- B`. [5]

Restriction: `A` and `B` must denote regular languages, not relations.

[5] Because all the replace operators have an inverse and an optional variant we will not list them explicitly in the following sections. Optionality is indicated by parentheses around the operator, inversion by the leftward direction of the arrow.

For example, the relation `[a b c -> d e]` contains the crossproduct `[a b c .x. d e]` and infinitely many other pairs because the upper language of the relation is the universal language. The transducer that encodes the relation is unambiguous when applied downward. It maps "abcde" uniquely to "dede". It is not unambiguous in the other direction. When applied upwards it maps "dede" to "abcabc", "abcde", "deabc" and "dede" because "de" in the lower language may be an "original" string or the result of a replacement.

If the `B` component of `[A -> B]` denotes the empty string language, then the `A` substrings are in effect deleted in the lower language. If `B` contains more than one string, the relation is one-to-many. It may fail to be one-to-one even if `B` contains just one string. For example, `[a | a a -> b]` pairs the upper-side string "aa" both with "bb" and "b". We come back to this point below.

The replacement of a non-empty language by the empty language is in effect a prohibition of that non-empty language. For example, `[a b c -> \?]` denotes the same identity relation as `~$ [a b c]`, excluding all strings that contain "abc". Except for this special case, the upper language of all `->` replacements is the universal language. This holds also for expressions such as `[\? -> a b c]`. If there is nothing to be replaced, replacement reduces to the universal identity relation.

If the set of strings to be replaced contains or consists of the empty string, as in `[0 -> %+]`, the resulting transducer will contain an epsilon loop as shown in Figure 2.10.

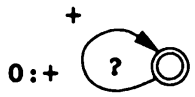


Figure 2.10: The Relation `0 -> %+`

When applied to an input string in the downward direction, this transducer inserts arbitrarily long bursts of + signs in every position in the input word. (The notation `%+` in a regular expression denotes the literal plus sign rather than the Kleene-plus operator.) This is the correct behavior because the empty string has no length and any number of concatenations of the empty string with itself yield the empty string. Any upper-side input string will have an infinite number of lower-side outputs.

It is often desirable to define a relation that involves making just one insertion at each position in the input string. As this cannot be defined as a replacement for the empty string, we have to introduce special notation to make it possible. Dotted brackets indicate single insertion.

Single Insertion

- `[[. A .] -> B]`, where the input expression `A` is surrounded by the special "dotted brackets" `[.` and `.]`, is equivalent to `[A -> B]`

if the language denoted by A does not contain the empty string. If the language of A does include the empty string, then `[ [. A .] -> B]` denotes the relation that pairs every string of the universal language with all strings that are identical to the original except that every instance of A that occurs as a substring in the original is represented by a string from B, and all the other substrings are represented by a copy in which a string from B appears before and after every original symbol.

Restriction: A and B must denote regular languages, not relations.

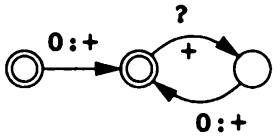


Figure 2.11: The Relation `[. 0 .] -> %+`

Figure 2.11 shows the network compiled from `[[. 0 .] -> %+]`. This transducer maps "cab" to "+c+a+b+", inserting just a single plus sign in each original string position. The epsilon symbol is redundant in `[. 0 .]`. Just as we allow an empty pair of square brackets `[]` as an alternate notation for the empty string, an empty pair of dotted brackets `[..]` is also accepted.

The dotted-bracket notation allows us to assign the desired interpretation to expressions such as `[[. (a) .] -> %+]`, where the language denoted by the first expression consists of "a" and the empty string. The corresponding transducer replaces **a** by + and inserts a single + in all positions in the rest of the string. For example, it maps "cab" to "+c+++b+".

The marking expression introduced below provides a general way of defining relations that change the input only by insertions of new material leaving the original parts unchanged.

Marking

- `[A -> B ... C]` denotes a relation in which each string of the upper-side universal language is paired with all strings that are identical to the original except that every instance of A that occurs as a substring is represented by a copy that has a string from B as a prefix and a string from C as a suffix.

  The sequence of three periods is a defined operator and is required in such marking rules; it has no special meaning in other constructions. B or C can be omitted.

Restriction: A, B, and C must denote regular languages, not relations.

Expressions of the form `A -> B ... C` are typically used to mark or bracket instances of the language `A` in some special way. For example, the transducer for `[a|e|i|o|u -> %[ ... %]]` maps "`abide`" to "`[a]b[i]d[e]`", enclosing each vowel in literal square brackets.

Parallel replacement

- `[A -> B, C -> D]` denotes the simultaneous replacement of `A` by `B` and `C` by `D`. A parallel replacement may include any number of components, separated by commas.

Restriction: `A`, `B`, and `C`, `D`, etc. must denote regular languages.

Parallel replacements allow two symbols to be exchanged for each other. For example, `[a -> b, b -> a]` maps "`baab`" to "`abba`", which is awkward to do with sequentially ordered rules. Similarly, `[%. -> %, , %, -> %.]` exchanges the comma and the period, mapping, for example, from "`1,000.0`" to "`1.000,0`". If the `A` and `C` components contain strings that overlap, the relation is not one-to-one. For example, `[a b -> x, b c -> y]` maps "`abc`" to both "`xc`" and "`ay`" because the two replacements are made independently of one another in all possible combinations. We return to this issue below.

Conditional Replacement

All the different types of replace relations introduced in the previous section can be constrained to apply only if the string to be replaced or its replacement appears in a certain context.

Replacement contexts are specified in the same way as the contexts for restriction: `L _ R`, where `L` is the left context, `R` is the right context, and `_` marks the site of the replacement. The boundary symbol `.#.` can be used to mark the beginning and the end of a string. Any number of contexts may be given, with a comma as the separator.

The notion of context is self-evident in the case of restriction expressions because they denote simple languages, but this is not the case for replacements that involve relations. The notation introduced below provides special symbols, `||`, `//`, `\\`, and `\/`, to distinguish among four different ways of interpreting the context specification.

Conditional Replacement

The four expressions below all denote a relation that is like `[A -> B]` except that the replacement of an original upper-side substring by a string from `B` is made only if the indicated additional constraint is fulfilled. Otherwise no change is made. Note that these restrictions are stated as they apply to right-arrow rules, where the upper side is naturally treated as the input side.

- `[A -> B || L _ R]`
  Every replaced substring in the upper language is immediately preceded by an upper-side string from L and immediately followed by an upper-side string from R.
- `[A -> B // L _ R]`
  Every replaced substring in the upper language is immediately followed by an upper-side string from R and the lower-side replacement string is immediately preceded by a string from L.
- `[A -> B \\ L _ R]`
  Every replaced substring in the upper language is immediately preceded by an upper-side string from L and the lower-side replacement string is immediately followed by a string from R.
- `[A -> B \/ L _ R]`
  Every lower-side replacement string is immediately preceded by a lower-side string from L and immediately followed by a lower-side string from R.

Restriction: A, B, L, and R must denote regular languages, not relations.

In other words, in | | replacements both the left and right contexts are matched in the upper-language string. In // replacements, the left context is matched on the lower-side and the right context on the upper side of the replacement site. In \\ replacements, conversely, the left context is matched on the upper side and the right context on the lower side. In \/ replacements, both the left and the right context are matched on the lower side. The slant of the context separators is intended to convey where each context is matched.[6]

In practice, when writing right-arrow phonological/orthographical alternation rules, it is usually desirable to constrain a replacement by the original upper-side context. That is, most rules written in practice are | | rules, and many users of the **Xerox** calculus have no need for the other variants. However, there are linguistic phenomena such as vowel harmony, tone spreading, and umlaut that are most naturally described in terms of a left-to-right or right-to-left process in which the result of a replacement itself serves as the context for another replacement.[7]

Figure 2.12 illustrates the difference between the | | and // versions of the same expression, the deletion of an initial **a**. When applied downward to an input

[6]The slant of the context operators suggests where contexts are matched in *right-arrow* rules. For left-arrow rules, where the input side is the lower side, the slant of the context operators is interpreted in the inverse way. See Section 3.5.5.

[7]The **Xerox** finite-state software also includes the **twolc** compiler for rules written in "two-level" format. Two-level rules give the rule writer a similar option to control the distribution of all symbols and symbol pairs in terms of both the upper and the lower context or in any combination of the two. The **twolc** compiler is included on the CD-ROM and is described on `http://www.fsmbook.com/`.

Figure 2.12: `[a -> 0 || .#. _]` vs. `[a -> 0 // .#. _]`

string such as "aab", the transducer on the left in Figure 2.12 deletes only the initial **a**, yielding "ab" whereas the `//` version yields "b" because the deletion of the initial **a** creates the lower-side context for the deletion of the next one. Similarly for an input word like "baa", `[a -> 0 || _ .#.]` deletes only the final **a**, but `[a -> 0 \\ _ .#.]` deletes all the **a**s at the end of the word. See Section 3.5.5.

Any number of parallel replacements may be conditioned jointly by one or more contexts, as in the rule `[a -> b, b -> a || .#. _ , _ .#.]` that exchanges **a** and **b** at the beginning and at the end of a word. Because the comma is used as a separator between alternate contexts, we need to introduce a second separator, a double comma `,,` to make it possible to express the parallel combination of conditional replacements.

In practical morphological applications, `//` replacement can be useful for languages with vowel harmony whereas `\\` replacement can be useful for languages with umlaut. Most applications need only `||` replacement.

Parallel Conditional Replacement

We give just one example to illustrate the syntax.

- `[A -> B || L1 _ R1 ,, C -> D || L2 _ R2]` replaces `A` by `B` in the context of `L1` and `R1` and simultaneously `C` by `D` in the context of `L2` and `R2`.

Restriction: All components must denote regular languages.

As the reader must wonder what use there could possibly be for such complex expressions, we give an example in Figure 2.13. This expression denotes the mapping from the first hundred Arabic numerals to their Roman counterparts. For example, it maps "0" to the empty string, "4" to "IV", "44" to "XLIV", and so on.

The three contexts are necessary in Figure 2.13 because zero is mapped to an epsilon everywhere, whereas the Roman value of the nine other digits depends on their position. The corresponding transducer is unambiguous in the Arabic-to-Roman direction, but it gives multiple results in the other direction because every Roman numeral in the lower language can be interpreted as an original one or as a result of the replacement. An unambiguous bidirectional converter can be derived by composing the relation in Figure 2.13 with `~$[I|V|X|L|C]` on the

```
%0 -> 0                                          || _ (?) .#. ,,

 1 -> I,      2 -> I I,      3 -> I I I,
 4 -> I V,    5 -> V,        6 -> V I,
 7 -> V I I,  8 -> V I I I,  9 -> I X || _ .#. ,,

 1 -> X,      2 -> X X,      3 -> X X X,
 4 -> X L,    5 -> L,        6 -> L X,
 7 -> L X X,  8 -> L X X X,  9 -> X C || _ ? .#.
```

Figure 2.13: A Relation from Arabic to Roman Numerals

upper side and with ~$ [%0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9] on the lower side. These constraints eliminate all Roman numerals from the upper language and all Arabic numerals from the lower language, leaving the relation otherwise intact.

Directed Replacement

As we already mentioned in connection with a couple of examples, replacement relations introduced in the previous sections are not necessarily one-to-one even if the replacement language contains just one string. The transducer compiled from [a | a a -> b] maps the upper language "aa" to both "bb" and "b". The transducer for [a b -> x, b c -> y] gives two results, "xc" and "ay", for the upper-language input string "abc".

This nondeterminism arises in two ways. First of all, possible replacements may overlap. We get a different result in the "abc" case depending on which of the two overlapping substrings is replaced. Secondly, there may be more than one possible replacement starting at the same point, as in the beginning of "aa", where either "a" or "aa" could be replaced.

The family of directed replace operators introduced in the following section eliminates this type of nondeterminism by adding directionality and length constraints. Directionality means that the replacement sites are selected starting from the left or from the right, not allowing any overlaps. Whenever there are multiple candidate strings starting at a given location, the longest or the shortest one is selected.

Directed Replacement

The four expressions below denote a relation that is like [A -> B] except that the substrings to be replaced are selected under the specified regimen.

The new replace operators can be used in all types of replace expressions introduced so far in place of ->.

- `[A @-> B]`
  Replacement strings are selected from left to right. If more than one candidate string begins at a given location, only the longest one is replaced.
- `[A ->@ B]`
  Replacement strings are selected from right to left. If more than one candidate string begins at a given location, only the longest one is replaced.
- `[A @> B]`
  Replacement strings are selected from left to right. If more than one candidate string begins at a given location, only the shortest one is replaced.
- `[A >@ B]`
  Replacement strings are selected from right to left. If more than one candidate string begins at a given location, only the shortest one is replaced.

Restriction: `A` and `B` must denote regular languages.

The additional constraints attached to the directed replace operators guarantee that any upper-language input string is uniquely factorized into a sequence of two types of substrings: the ones that are replaced and the ones that are passed on unchanged.

Figure 2.14 illustrates the difference between `[a | a a -> b]` and the left-to-right, longest match version `[a | a a @-> b]`. It is easy to see that the transducer for the latter maps the upper language string "aa" unambiguously to "b".

Figure 2.14: The Relation `[a | a a -> b]` vs. `[a | a a @-> b]`

The left-to-right, longest-match replace operator `@->` is commonly used for text normalization, tokenization, and for "chunking" regions of text that match a given pattern. For example `[["\t"|"\n"|" "]+ @-> "\n"]` yields a transducer that reduces a maximally long sequence of tabs, newlines, and spaces into a single newline character.

To give a simple example of chunking, let us assume that a noun phrase consists of an optional determiner, `(d)`, any number of adjectives, `a*`, and one or

```
d a n n   v   a a n
-------       -----
[ d a n n ] v [ a a n ]
```

Figure 2.15: Application of `[(d) a*n+ @-> %[...%]]` to "dannvaan"

more nouns, `n+`. The expression `[(d) a* n+ @-> %[ ... %]]` compiles into a transducer that inserts brackets around maximal instances of the noun-phrase pattern. For example, it maps "dannvaan" into "`[dann]v[aan]`", as shown in Figure 2.15. Although the input string "dannvaan" contains many other instances of the noun-phrase pattern, "dan", "an", "nn", etc., the left-to-right and longest-match constraints pick out just the two maximal ones.

```
C* V+ C* @-> ... "-" || _ C V
```

Figure 2.16: A Simple Syllabification Rule

Directional replacement and marking expressions may be further constrained by specifying one or more contexts. For example, if `C` and `V` stand for consonants and vowels, respectively, a simple syllabification rule may be expressed as in Figure 2.16. The marking expression in Figure 2.16 compiles into an unambiguous transducer that inserts a hyphen after each longest available instance of the `C* V+ C*` pattern that is followed by a consonant and a vowel. The relation it encodes consists of pairs of strings such as the example in Figure 2.17. The choice between `||` and `//` makes no difference in this case, but the two other context markers, `\\` and `\/`, would not yield the intended result here.

```
s t r u k   t u   r a   l i s   m i
s t r u k - t u - r a - l i s - m i
```

Figure 2.17: Application of `[C* V+ C* @-> ... "-" || _ C V]`

2.5 Properties of Finite-State Networks

In this section we consider briefly the formal properties of finite-state automata. All the networks presented in this chapter have the three important properties defined in Table 2.11.

EPSILONFREE	There are no arcs labeled with the epsilon symbol.
DETERMINISTIC	No state has more than one outgoing arc with the same label.
MINIMAL	There is no other network with exactly the same paths that has fewer states.

Table 2.11: Properties of Networks

The **Xerox** regular-expression compiler always makes sure that the result is epsilonfree, deterministic and minimal. If the network encodes a regular language, it is guaranteed that it is the best encoding for the language in the sense that any other network for the same language has the same number of states and arcs and differs only with respect to the order of the arcs, which generally is irrelevant.

The situation is more complex in the case of regular relations. Even if a transducer is epsilonfree, deterministic, and minimal in the sense of Table 2.11 there may still be another network with fewer states and arcs for that same encoded relation. If the network has arcs labeled with a symbol pair that contains an epsilon on one side, these one-sided epsilons could be distributed differently, or perhaps even eliminated, and this might reduce the size of the network.

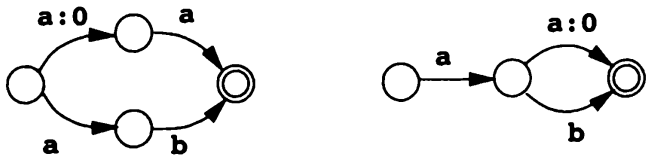


Figure 2.18: `[a:0 a|a b]` vs. `[a [a:0 | b]]`

For example, the two networks shown in Figure 2.18 encode the same relation, {<"aa", "a">, <"ab", "ab">}. They are both deterministic and minimal, but one is smaller than the other due to a more optimal placement of the one-sided epsilon transition. In the general case there is no way to determine whether a given transducer is the best encoding for an arbitrary relation.

For transducers, the intuitive notion of determinism makes sense only with respect to a given direction of application. But there still are two ways to think about determinism, as defined in Table 2.12.

Although both transducers in Figure 2.18 are in fact unambiguous in both directions, the one on the left is not SEQUENTIAL in either direction. When it is applied downward, to the string "aa", there are two paths that have to be pursued initially even though only one will succeed. The same is true in the other direction

UNAMBIGUOUS	For any input there is at most one output.
SEQUENTIAL	No state has more than one outgoing arc with the same symbol on the input side.

Table 2.12: Properties of Transducers

as well. In other words, there is *local* ambiguity at the start state because **a** may have to be deleted or retained. In this case, the ambiguity is resolved by the next input symbol one step later.

If the relation itself is unambiguous in the relevant direction and if all the ambiguities in the transducer resolve themselves within some fixed number of steps, the transducer is called SEQUENTIABLE. That is, we can construct an equivalent sequential transducer in the same direction (Roche and Schabes, 1997; Mohri, 1997; Kempe, 2000a; Kempe, 2000b; Gaál, 2001; Gaál, 2002). Figure 2.19 shows the downward sequentialized version of the leftmost transducer in Figure 2.18.

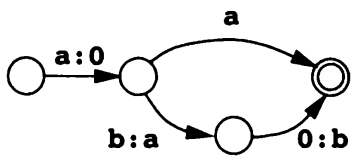


Figure 2.19: Downward Sequentialized Version of `[a:0 a | a b]`

The sequentialization algorithm combines the locally ambiguous paths into a single path that does not produce any output until the ambiguity has been resolved. In the case at hand, the ambiguous path contains just one arc. When a **b** is seen, the delayed **a** is produced as output, and then the **b** itself in a one-sided epsilon transition. Otherwise, an **a** must follow, and in this case there is no delayed output. In effect the local ambiguity is resolved with one symbol lookahead.

The network in Figure 2.19 is sequential but only in the downward direction. Upward sequentialization produces the second network shown in Figure 2.18, which clearly is the best encoding for this little relation.

Even if a transducer is unambiguous, it may well be unsequentiable if the resolution of a local ambiguity requires an unbounded amount of lookahead. For example, the simple transducer for `[a+ @-> 0 || b _ c]` in Figure 2.20 cannot be sequentialized in either direction.

This transducer reduces any sequence of **a**s that is preceded by a **b** to an epsilon or copies it to the output unchanged, depending on whether the sequence of **a**s is followed by a **c**. A sequential transducer would have to delay the decision until it reaches the end of an arbitrarily long sequence of **a**s. It is obviously impossible for

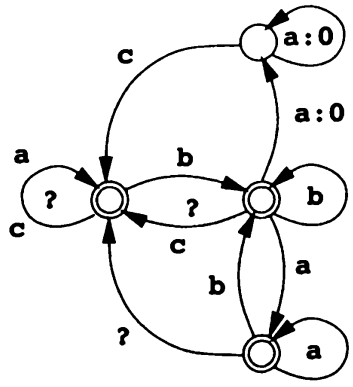


Figure 2.20: Unsequentiable Transducer `[a+ @-> 0 || b _ c]`

any finite-state device to accumulate an unbounded amount of delayed output.

However, in such cases it is always possible to split the unambiguous but non-sequentiable transducer into a BIMACHINE. A bimachine for an unambiguous relation consists of two sequential transducers that are applied in a sequence. The first half of the bimachine processes the input from left-to-right; the second half of the bimachine processes the output of the first half from right-to-left. Although the application of a bimachine requires two passes, a bimachine is in general more efficient to apply than the original transducer because the two components of the bimachine are both sequential. There is no local ambiguity in either the left-to-right or the right-to-left half of the bimachine if the original transducer is unambiguous in the given direction of application. Figure 2.21 shows a bimachine derived from the transducer in Figure 2.20.

Left-to-right Right-to-left

Figure 2.21: Bimachine for `[a+ @-> 0 || b _ c]`

The left-to-right half of the bimachine in Figure 2.21 is only concerned about the left context of the replacement. A string of **a**s that is preceded by **b** is mapped to a string of **a1**s, an auxiliary symbol to indicate that the left context has been matched. The right-to-left half of the bimachine maps each instance of the auxiliary symbol either to **a** or to an epsilon, depending on whether it is preceded by **c** when the intermediate output is processed from right-to-left. Figure 2.22 illustrates the application of the bimachine to three input strings, "`aaa`", "`baaa`" and "`baaac`".

```
 a a a        b a  a  a        b a  a  a  c
 - - ->       - -  -  ->       - -  -  -  ->
 a a a        b a1 a1 a1       b a1 a1 a1 c
<- - -       <- -- -- --      <- -- -- -- -
 a a a        b a  a  a        b             c
```

Figure 2.22: Application of a Bimachine

The bimachine in Figure 2.21 encodes exactly the same relation as the transducer in Figure 2.20. The composition of the left-to-right half `L` of the bimachine with the reverse of the right-to-left half `R` yields the original single transducer `T`. That is, `T = [L .o. R.r]` when the auxiliary symbols introduced in the bimachine factorization are removed from the sigma alphabet.

2.6 Exercises

Construction of finite-state networks by hand, as we did in the exercises of Chapter 1, is a tedious affair and very error-prone except for the most trivial cases. With the help of the regular-expression calculus we can give a precise specification of the language or the relation we wish to construct and get the regular-expression compiler to do the rest of the work for us.

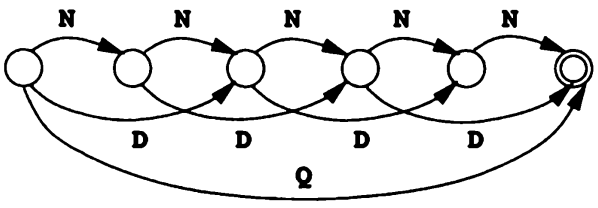


Figure 2.23: The Simple Cola Machine

As a warm-up exercise, let us first construct a regular expression that denotes the language of the Simple Cola Machine in Chapter 1, redrawn in Figure 2.23. The strings of the language are made up of three symbols, **N** (nickel = 5 cents), **D** (dime = 10 cents), and **Q** (quarter = 25 cents). The language consists of strings whose value is exactly 25 cents or, equivalently, five nickels.

The simplest method to construct the network in Figure 2.23 is to take advantage of the fact that the value of each symbol in the language can be expressed in terms of nickels. A dime is two nickels, a quarter is five nickels. We can encode these facts as a relation from higher-valued coins to the equivalent sequence of nickels: `[D -> N^2, Q -> N^5]`. The upper language of this relation is the

universal language in which the three symbols **N**, **D**, and **Q** occur in any combination any number of times along with any other symbols.

But for our Cola Machine, we are only interested in sequences whose value adds up to five nickels and there should be no other symbols. To derive that sublanguage, we simply compose the lower side of the relation with `[N^5]` and extract the upper side of the result. The expression in Figure 2.24 compiles exactly into the network displayed in Figure 2.23.

```
[[D -> N^2, Q -> N^5] .o. N^5].u
```

Figure 2.24: A Regular Expression Definition of the Simple Cola Machine

2.6.1 The Better Cola Machine

Modify the regular expression in Figure 2.24 so that it compiles into a transducer that maps any sequence of coins whose value is exactly 25 cents into the multicharacter symbol **[COLA]**.

Having succeeded in that task, modify the expression once more to describe a vending machine that accepts any sequence of coins and outputs "`[COLA]`" every time the value is equal to a multiple of 25 cents.

Finally, make the machine honest. Any extra money should be returned to the customer. For example, the sequence "QDDNNN" should map into two colas and a dime: "`[COLA] [COLA] D`". If the customer does not insert enough money for a cola, the money should be returned; that is, the transducer should map a string such as "DN" into itself or into some equivalent sequence of coins such as "NNN".

A solution to this problem is given in Appendix B, page 466. It is of course not the only one. There are infinitely many regular expressions that denote any given regular relation.