Voici un principe d'esthétique . . . une règle, dis-je, pour les artistes:
Soyez réglé dans votre vie et ordinaire comme un bourgeois,
afin d'être violent et original dans vos œuvres.

[Here is a principle of aesthetics . . . a rule, I say, for artists:
Be regular in your life and ordinary as a bourgeois,
so that you may be violent and original in your work.]

— Gustave Flaubert, in a letter to Gertrude Tennant (December 25, 1876)

Some people, when confronted with a problem, think "I know, I'll use regular expressions."
Now they have two problems.

— Jamie Zawinski, alt.religion.emacs (August 12, 1997)

As far as I am aware this pronunciation is incorrect in all known languages.

— Kenneth Kleene, describing his father Stephen's pronunciation of his last name

2 Regular Languages

2.1 Languages

A ***formal language*** (or just a ***language***) is a set of strings over some finite alphabet Σ, or equivalently, an arbitrary subset of Σ^*. For example, each of the following sets is a language:

- The empty set $\varnothing$.[1]
- The set $\{\varepsilon\}$.
- The set $\{\mathtt{0}, \mathtt{1}\}^*$.
- The set $\{\mathtt{THE}, \mathtt{OXFORD}, \mathtt{ENGLISH}, \mathtt{DICTIONARY}\}$.
- The set of all subsequences of THE◇OXFORD◇ENGLISH◇DICTIONARY.
- The set of all words in *The Oxford English Dictionary*.
- The set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ with an odd number of 1s.
- The set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ that represent a prime number in base 13.
- The set of all sequences of turns that solve the Rubik's cube (starting in some fixed configuration)
- The set of all python programs that print "Hello World!"

As a notational convention, I will always use italic upper-case letters (usually L, but also A, B, C, and so on) to represent languages.

[1]The empty set symbol $\varnothing$ was introduced in 1939 by André Weil, as a member of the pseudonymous mathematical collective Nicholai Bourbaki. The symbol derives from the Danish and Norwegian letter Ø, which pronounced like the vowels in the German word "blöd" (stupid) or the French word "œuf" (egg), or a sound of disgust. The symbol has *nothing* to do with the Greek letter ϕ; calling the empty set "fie" or "fee" makes the baby Jesus cry.

Russell and Whitehead's *Principia Mathematica*, published in 1910, denoted the empty set with the Greek letter Λ, presumably as a mnemonic for the German word *leer*, meaning "empty".

Many early set theorists, including Fraenkel, Frege, Russell, and Zermelo, did not believe that the empty set (or "null-class") was an actual set, although they were willing to grudgingly admit that it was a technically useful concept. I encourage students whose are skeptical about $\varnothing$ to adopt the same attitude.

Formal languages are not "languages" in the same sense that English, Klingon, and Python are "languages". Strings in a formal language do not necessarily carry any "meaning", nor are they necessarily assembled into larger units ("sentences" or "paragraphs" or "packages") according to some "grammar".

It is ***very*** important to distinguish between three "empty" objects. Many beginning students have trouble keeping these straight.

- $\boldsymbol{\varepsilon}$ is the empty *string*, which is a sequence of length zero. ε is not a language.
- $\boldsymbol{\varnothing}$ is the empty *language*, which is a set containing zero strings. $\varnothing$ is not a string.
- $\boldsymbol{\{\varepsilon\}}$ is a language containing exactly one string, which has length zero. $\{\varepsilon\}$ is not empty, and it is not a string.

2.2 Building Languages

Languages can be combined and manipulated just like any other sets. Thus, if A and B are languages over Σ, then their union $A \cup B$, intersection $A \cap B$, difference $A \setminus B$, and symmetric difference $A \oplus B$ are also languages over Σ, as is the complement $\overline{A} := \Sigma^* \setminus A$. However, there are two more useful operators that are specific to sets of *strings*.

The ***concatenation*** of two languages A and B, again denoted $\boldsymbol{A \bullet B}$ or just $\boldsymbol{AB}$, is the set of all strings obtained by concatenating an arbitrary string in A with an arbitrary string in B:

$$A \bullet B := \{xy \mid x \in A \text{ and } y \in B\}.$$

For example, if $A = \{\mathtt{HOCUS}, \mathtt{ABRACA}\}$ and $B = \{\mathtt{POCUS}, \mathtt{DABRA}\}$, then

$$A \bullet B = \{\mathtt{HOCUSPOCUS}, \mathtt{ABRACAPOCUS}, \mathtt{HOCUSDABRA}, \mathtt{ABRACADABRA}\}.$$

In particular, for every language A, we have

$$\varnothing \bullet A = A \bullet \varnothing = \varnothing \qquad \text{and} \qquad \{\varepsilon\} \bullet A = A \bullet \{\varepsilon\} = A.$$

The ***Kleene closure*** or ***Kleene star***[2] of a language L, denoted $\boldsymbol{L^*}$, is the set of all strings obtained by concatenating a sequence of zero or more strings from L. For example,

$$\{\mathtt{0}, \mathtt{11}\}^* = \{\varepsilon, \mathtt{0}, \mathtt{00}, \mathtt{11}, \mathtt{000}, \mathtt{011}, \mathtt{110}, \mathtt{0000}, \mathtt{0011}, \mathtt{0110}, \mathtt{1100}, \mathtt{1111}, \mathtt{00000}, \dots, \mathtt{011110011011}, \dots\}.$$

More formally, L^* is defined recursively as the set of all strings w such that either

- $w = \varepsilon$, or
- $w = xy$, for some strings $x \in L$ and $y \in L^*$.

This definition immediately implies that

$$\varnothing^* = \{\varepsilon\} \quad \text{and} \quad \{\varepsilon\}^* = \{\varepsilon\}.$$

For every language L that contains at least one non-empty string, the Kleene closure L^* is infinite and contains arbitrarily long (but *finite!*) strings. L^* can also be defined as the smallest superset

[2]named after logician Stephen Cole Kleene, who actually pronounced his last name "***clay***-knee", not "clean" or "cleanie" or "claynuh" or "dimaggio".

of L that contains the empty string ε and is closed under concatenation (hence "closure"). The set of all strings Σ^* is, just as the notation suggests, the Kleene closure of the alphabet Σ (where each symbol is viewed as a string of length 1).

A common variant of the Kleene closure operator is the ***Kleene plus***, defined as $L^+ := L \bullet L^*$. Thus, L^+ is the set of all strings obtained by concatenating a sequence of ***one*** or more strings from L. However, I recommend avoiding this notation, because it is easy to confuse with the more common use of $+$ to denote union.

The following identities, which we state here without (easy) proofs, are useful for designing, simplifying, and understanding languages.

Lemma 2.1. *The following identities hold for all languages A, B, and C:*

(a) $A \cup B = B \cup A$.

(b) $(A \cup B) \cup C = A \cup (B \cup C)$.

(c) $\varnothing \bullet A = A \bullet \varnothing = \varnothing$.

(d) $\{\varepsilon\} \bullet A = A \bullet \{\varepsilon\} = A$.

(e) $(A \bullet B) \bullet C = A \bullet (B \bullet C)$.

(f) $A \bullet (B \cup C) = (A \bullet B) \cup (A \bullet C)$.

(g) $(A \cup B) \bullet C = (A \bullet C) \cup (B \bullet C)$.

Lemma 2.2. *The following identities hold for every language L:*

(a) $L^* = \{\varepsilon\} \cup L^+ = L^* \bullet L^* = (L \cup \{\varepsilon\})^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup L \cup (L \bullet L^+)$.

(b) $L^+ = L \bullet L^* = L^* \bullet L = L^+ \bullet L^* = L^* \bullet L^+ = L \cup (L \bullet L^+) = L \cup (L^+ \bullet L^+)$.

(c) $L^+ = L^*$ *if and only if* $\varepsilon \in L$.

Lemma 2.3 (Arden's Rule). *For any languages A, B, and L such that $L = A \bullet L \cup B$, we have $A^* \bullet B \subseteq L$. Moreover, if A does not contain the empty string, then $L = A \bullet L \cup B$ if and only if $L = A^* \bullet B$.*

2.3 Regular Languages and Regular Expressions

Intuitively, a language is ***regular*** if it can be constructed from individual strings using any combination of union, concatenation, and unbounded repetition. More formally, a language L is regular if and only if it satisfies one of the following (recursive) conditions:

- L is empty;
- L contains exactly one string (which could be the empty string ε);
- L is the union of two regular languages;
- L is the concatenation of two regular languages; or
- L is the Kleene closure of a regular language.

Equivalently, a set of strings is regular if it is the set of all possible outputs of a program without subroutines (or `goto`). Specifically,

- The empty language corresponds to a program that produces no output.
- The singleton language $\{w\}$ corresponds to the one-line program `print(`w`)`.
- Sequencing: The language AB corresponds to the program containing all lines of program A, followed by all lines of program B.
- Branching: The language $A \cup B$ corresponds to the program `if` ——— A `else` B, where the line ——— represents some unknown condition.
- Iteration The language A^* corresponds to the program `while` ——— A, where again the line ——— represents some unknown condition.

Regular languages are normally described using a compact notation called ***regular expressions***, which omit braces around one-string sets, use $+$ to represent union instead of $\cup$, and juxtapose subexpressions to represent concatenation instead of using an explicit operator $\bullet$. By convention, in the absence of parentheses, the * operator has highest precedence, followed by the (implicit) concatenation operator, followed by $+$. (These conventions mirror the precedence rules for exponents, implicit multiplication, and addition in high-school algebra.)

For example, the regular expression $\mathtt{10}^*$ is shorthand for the regular language $\{\mathtt{1}\} \bullet \{\mathtt{0}\}^* = \{\mathtt{1}, \mathtt{10}, \mathtt{100}, \mathtt{1000}, \mathtt{10000}, \ldots\}$, or equivalently, the code pattern

```
print(1)
while ———:
    print(0)
```

and *not* the language $\{\mathtt{10}\}^* = \{\varepsilon, \mathtt{10}, \mathtt{1010}, \mathtt{101010}, \mathtt{10101010}, \ldots\}$. As a larger example, the regular expression

$$\mathtt{0}^*\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$$

represents the language

$$(\{\mathtt{0}\}^* \bullet \{\mathtt{0}\}) \cup \Big(\{\mathtt{0}\}^* \bullet \{\mathtt{1}\} \bullet \big((\{\mathtt{1}\} \bullet \{\mathtt{0}\}^* \bullet \{\mathtt{1}\}) \cup (\{\mathtt{0}\} \bullet \{\mathtt{1}\}^* \bullet \{\mathtt{0}\})\big)^* \bullet \{\mathtt{1}\} \bullet \{\mathtt{0}\}^*\Big).$$

or the following equivalent code pattern:

```
if ———:
    print(0)
    while ———:
        print(0)
else:
    while ———:
        print(0)
    print(1)
    while ———:
        if ———:
            print(1)
            while ———:
                print(0)
            print(1)
        else:
            print(0)
            while ———:
                print(1)
            print(0)
        # end if-else
    # end while
    print(1)
    while ———:
        print(0)
# end if-else
```

Most of the time we do not distinguish between regular expressions and the languages they represent, for the same reason that we do not normally distinguish between the arithmetic expression "$2+2$" and the integer 4, or the Greek letter π and the area of the unit circle, or the not-Greek letter $\varnothing$ and the empty set. However, we sometimes need to refer to regular expressions themselves *as strings*. In those circumstances, we write $\boldsymbol{L(R)}$ to denote the language represented by the regular expression R. String w ***matches*** regular expression R if and only if $w \in L(R)$.

Here are several more examples of regular expressions and the languages they represent.

- $\mathtt{0}^*$ — the set of all strings of $\mathtt{0}$s, including the empty string.
- $\mathtt{00000}^*$ — the set of all strings of $\mathtt{0}$s whose length is at least 4.
- $(\mathtt{00000})^*$ — the set of all strings of $\mathtt{0}$s whose length is a multiple of 5.
- $(\mathtt{0}+\mathtt{1})^*$ — the set of all binary strings.
- $(\varepsilon+\mathtt{1})(\mathtt{01})^*(\varepsilon+\mathtt{0})$ — the set of all strings of alternating $\mathtt{0}$s and $\mathtt{1}$s, or equivalently, the set of all binary strings that do not contain the substrings $\mathtt{00}$ or $\mathtt{11}$.
- $(\mathtt{0}+\mathtt{1})^*\mathtt{0000}(\mathtt{0}+\mathtt{1})^*$ — the set of all binary strings that contain the substring $\mathtt{0000}$.
- $((\varepsilon+\mathtt{0}+\mathtt{00}+\mathtt{000})\mathtt{1})^*(\varepsilon+\mathtt{0}+\mathtt{00}+\mathtt{000})$ — the set of all binary strings that do not contain the substring $\mathtt{0000}$.
- $((\mathtt{0}+\mathtt{1})(\mathtt{0}+\mathtt{1}))^*$ — the set of all binary strings whose length is even.
- $\mathtt{1}^*(\mathtt{01}^*\mathtt{01}^*)^*$ — the set of all binary strings with an even number of $\mathtt{0}$s.
- $\mathtt{0}+\mathtt{1}(\mathtt{0}+\mathtt{1})^*\mathtt{00}$ — the set of all non-negative binary numerals divisible by 4 and with no redundant leading $\mathtt{0}$s.
- $(\mathtt{0}+\mathtt{1}(\mathtt{01}^*\mathtt{0})^*\mathtt{1})^*$ — the set of all non-negative binary numerals divisible by 3, possibly with redundant leading $\mathtt{0}$s.

The last one should ***not*** be obvious, even to experts. It is straightforward, but *really* tedious, to prove by induction that every string in $(\mathtt{0}+\mathtt{1}(\mathtt{01}^*\mathtt{0})^*\mathtt{1})^*$ is the binary representation of a non-negative multiple of 3. It is similarly straightforward, but *even more* tedious, to prove that the binary representation of *every* non-negative multiple of 3 matches this regular expression. In a later note, we will see a systematic method for deriving regular expressions for some languages that avoids (or more accurately, *automates*) this tedium.

Two regular expressions R and R' are ***equivalent*** if they describe the same language. For example, the regular expressions $(\mathtt{0}+\mathtt{1})^*$ and $(\mathtt{1}+\mathtt{0})^*$ are equivalent, because the union operator is commutative. More subtly, the regular expressions $(\mathtt{0}+\mathtt{1})^*$ and $(\mathtt{0}^*\mathtt{1}^*)^*$ and $(\mathtt{00}+\mathtt{01}+\mathtt{10}+\mathtt{11})^*(\mathtt{0}+\mathtt{1}+\varepsilon)$ are all equivalent; intuitively, these represent *different ways of thinking about* the language of all binary strings $\{\mathtt{0},\mathtt{1}\}^*$. In fact, almost every regular language can be represented by infinitely many distinct but equivalent regular expressions, even if we ignore ultimately trivial equivalences like $L=(L\varnothing)^*L\varepsilon+\varnothing$.

2.4 Designing Regular Expressions

★★★

Give some examples of designing regular expressions. Using Arden's rule? Mirroring DFA construction? What other *general* design strategies are there?

- Case analysis: Break the pattern into several cases, and build a regular expression for each case. For example, we can partition "at most three X's" into four subcases: no X's, exactly one X, exactly two X's, and exactly three X's.
- Bottom up: Find simpler subpatterns, build regular expressions for those subpatterns, and then treat those expressions as new symbols. Common subpatterns include explicit substrings, arbitrary substrings ($(0+1)^*$), runs (00^*), alternation ($(01)^*$)
- Top down: Break strings into chunks, build regular expression to describe how those chunks fit together, and then treats chunks as new symbols.

Whether formally proving a regular expression is correct, or just brainstorming about designing a regular expression, it's important to remember that any regular expression for a language L must be both exhaustive (= matches every string in L) and exclusive (= matches only strings in L). So test both positive and negative examples; look for boundary cases; break the language into simpler sublanguages (cases). In short, do all the things you normally do when writing code.

2.5 Things What Ain't Regular Expressions

Many computing environments and programming languages support patterns called ***regexen*** (singular ***regex***, pluralized like *ox*) that are considerably more general and powerful than regular expressions. Regexen include special symbols representing negation, character classes (for example, upper-case letters, or digits), contiguous ranges of characters, line and word boundaries, limited repetition (as opposed to the unlimited repetition allowed by *), back-references to earlier subexpressions, and even local variables. Despite its obvious etymology, a regex is ***not*** necessarily a regular expression, and it does ***not*** necessarily describe a regular language![3]

To avoid any confusion, I strongly recommend considering "regular expression" and "regex" as names of two completely different things (descended from a common ancestor), and *never* using one name to refer to the other thing.

Another type of pattern that is often confused with regular expression are ***globs***, which are patterns used in most Unix shells and some scripting languages to represent sets of file names. Globs include symbols for arbitrary single characters (`?`), single characters from a specified range (`[a-z]`), arbitrary substrings (`*`), and substrings from a specified finite set (`{foo,ba{r,z}}`). Globs are significantly *less* powerful than regular expressions.

2.6 Regular Expression Trees

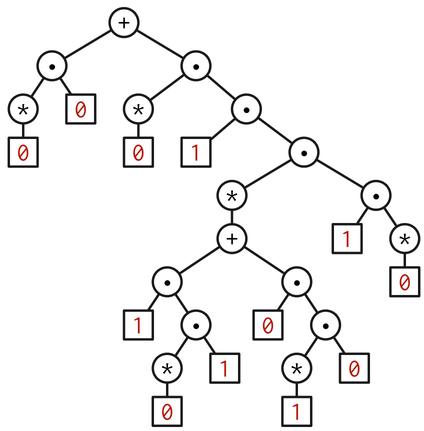
Regular expressions are convenient notational shorthand for a more explicit representation of regular languages called ***regular expression trees***. A regular expression tree is formally defined as one of the following:

- A leaf node labeled $\varnothing$.
- A leaf node labeled with a string in Σ^*.
- A node labeled $+$ with two children, each of which is the root of a regular expression tree.

[3]However, regexen are not all-powerful, either; see http://stackoverflow.com/a/1732454/775369.

- A node labeled • with two children, each of which is the root of a regular expression tree.
- A node labeled $*$ with one child, which is the root of a regular expression tree.

These cases mirror the definition of regular language exactly. A leaf labeled $\varnothing$ represents the empty language; a leaf labeled with a string represents the language containing only that string; a node labeled $+$ represents the union of the languages represented by its two children; a node labeled • represents the concatenation of the languages represented by its two children; and a node labeled $*$ represents the Kleene closure of the languages represented by its child.



A regular expression tree for $0^*0 + 0^*1(10^*1 + 01^*0)^*10^*$

It's convenient to define the ***size*** of a regular expression to be the number of nodes in its regular expression tree. The size of a regular expression could be either larger or smaller than its length as a raw string. On the one hand, concatenation nodes in the tree are not represented by symbols in the string; on the other hand, parentheses in the string are not represented by nodes in the tree. For example, the regular expression $0^*0 + 0^*1(10^*1 + 01^*0)^*10^*$ has size 29, but the corresponding raw string `0*0+0*1(10*1+01*0)*10*` has length 22.

A ***subexpression*** of a regular expression R is another regular expression S whose regular expression tree is a subtree of some regular expression tree for R. A ***proper subexpression*** of R is any subexpression except R itself. Every subexpression of R is also a substring of R, but not every substring is a subexpression. For example, the substring 10^*1 is a proper subexpression of $0^*0+0^*1(10^*1+01^*0)^*10^*$. However, the substrings 0^*0+0^*1 and 0^*1+01^* are *not* subexpressions of $\underbrace{0^*0 + 0^*1}(1\,\underbrace{0^*1 + 01^*}\,0)^*10^*$, even though they are well-formed regular expressions.

2.7 Proofs about Regular Expressions

The standard strategy for proving properties of regular expressions, just as for any other recursively defined structure, to argue inductively on the recursive *structure* of the expression, rather than considering the regular expression as a raw string. In fact, it suffices to argue inductively on the *size* of the regular expression, which is defined in terms of this recursive structure. If R' is a proper subexpression of R, then R' is smaller than R, but not vice versa.

Induction proofs about regular expressions follow a standard boilerplate that mirrors the recursive definition of regular languages. Suppose we want to prove that every regular expression is perfectly cromulent, whatever that means. The white boxes hide additional proof details that, among other things, depend on the precise definition of "perfectly cromulent". The boilerplate structure is *longer* than the boilerplate for string induction proofs, but don't be fooled into thinking it's *harder*. The five cases in the proof mirror the five cases in the recursive definition of regular language.

Proof: Let R be an arbitrary regular expression.
Assume that **every regular expression smaller than** R is perfectly cromulent.
There are five cases to consider.

- Suppose $R = \varnothing$.

  Therefore, R is perfectly cromulent.
- Suppose R is a single string.

  Therefore, R is perfectly cromulent.
- Suppose $R = S + T$ for some regular expressions S and T.
  The induction hypothesis implies that S and T are perfectly cromulent.

  Therefore, R is perfectly cromulent.
- Suppose $R = S \bullet T$ for some regular expressions S and T.
  The induction hypothesis implies that S and T are perfectly cromulent.

  Therefore, R is perfectly cromulent.
- Suppose $R = S^*$ for some regular expression. S.
  The induction hypothesis implies that S is perfectly cromulent.

  Therefore, R is perfectly cromulent.

In all cases, we conclude that w is perfectly cromulent. □

Here is an example of the structural induction boilerplate in action. Again, this proof is *longer* than a typical induction proof about strings or integers, but each individual case is still just a short exercise in definition-chasing.

Lemma 2.4. *Every regular expression that does not use the symbol $\varnothing$ represents a non-empty language.*

Proof: Let R be an arbitrary regular expression that does not use the symbol $\varnothing$. Assume that every regular expression that is smaller than R and does not use the symbol $\varnothing$ represents a non-empty language. There are five cases to consider, mirroring the definition of R.

- If $R = \varnothing$, we have a contradiction; we can ignore this case.
- If R is a single string w, then $L(R)$ contains the string w. (In fact, $L(R) = \{w\}$.)
- Suppose $R = S + T$ for some regular expressions S and T.
  S does not use the symbol $\varnothing$, because otherwise R would.

Thus, the inductive hypothesis implies that $L(S)$ is non-empty.
Choose an arbitrary string $x \in L(S)$.
Then $L(R) = L(S + T) = L(S) \cup L(T)$ also contains the string x.

- Suppose $R = S \bullet T$ for some regular expressions S and T.
Neither S nor T uses the symbol $\varnothing$, because otherwise R would.
Thus, the inductive hypothesis implies that both $L(S)$ and $L(T)$ are non-empty.
Choose arbitrary strings $x \in L(S)$ and $y \in L(T)$.
Then $L(R) = L(S \bullet T) = L(S) \bullet L(T)$ contains the string xy.
- Suppose $R = S^*$ for some regular expression S.
Then $L(R)$ contains the empty string ε.

In every case, we conclude that the language $L(R)$ is non-empty. $\square$

Similarly, most algorithms that accept regular expressions as input actually require regular expression *trees*, rather than regular expressions as raw *strings*. Fortunately, it is possible to ***parse*** any regular expression of length n into an equivalent regular expression tree in $O(n)$ time. (The details of the parsing algorithm are beyond the scope of this chapter.) Thus, when we see an algorithmic problem that starts "Given a regular expression...", we can assume without loss of generality that we are actually given a regular expression *tree*.

2.8 Proofs about Regular Languages

The same boilerplate also applies to inductive arguments about properties of regular *languages*. Languages themselves are just unadorned sets; they don't have any recursive structure that we build an inductive proof around. In particular, proper subsets of an *infinite* language L are not necessarily "smaller" than L! Rather than trying to argue directly about an arbitrary regular language L, we choose an arbitrary regular expression that represents L, and then build our inductive argument around *the recursive structure of that regular expression*.

Lemma 2.5. *Every non-empty regular language is represented by a regular expression that does not use the symbol $\varnothing$.*

Proof: Let R be an arbitrary regular expression; we need to prove that either $L(R) = \varnothing$ or $L(R) = L(R')$ for some $\varnothing$-free regular expression R'. For every regular expression S that is smaller than R, assume that either $L(S) = \varnothing$ or $L(S) = L(S')$ for some $\varnothing$-free regular expression S'. There are five cases to consider, mirroring the definition of R.

- If $R = \varnothing$, then $L(R) = \varnothing$.
- If R is a single string w, then R is already $\varnothing$-free.
- Suppose $R = S + T$ for some regular expressions S and T. There are four subcases to consider:
  - If $L(S) = L(T) = \varnothing$, then $L(R) = L(S) \cup L(T) = \varnothing$.
  - Suppose $L(S) \neq \varnothing$ and $L(T) = \varnothing$. The inductive hypothesis implies that there is a $\varnothing$-free regular expression S' such that $L(S') = L(S) = L(S) \cup L(T) = L(R)$.

  - Suppose $L(S) = \varnothing$ and $L(T) \neq \varnothing$. The inductive hypothesis implies that there is a $\varnothing$-free regular expression T' such that $L(T') = L(T) = L(S) \cup L(T) = L(R)$.
  - Finally, suppose $L(S) \neq \varnothing$ and $L(T) \neq \varnothing$. The inductive hypothesis implies that there are $\varnothing$-free regular expressions S' and T' such that $L(S') = L(S)$ and $L(T') = L(T)$. The regular expression $S' + T'$ is $\varnothing$-free and $L(S' + T') = L(S') \cup L(T') = L(S) \cup L(T) = L(S + T) = L(R)$.

- Suppose $R = S \bullet T$ for some regular expressions S and T. There are two subcases to consider.
  - If either $L(S) = \varnothing$ or $L(T) = \varnothing$ then $L(R) = L(S) \bullet L(T) = \varnothing$.
  - Otherwise, the inductive hypothesis implies that there are $\varnothing$-free regular expressions S' and T' such that $L(S') = L(S)$ and $L(T') = L(T)$. The regular expression $S' \bullet T'$ is $\varnothing$-free and equivalent to R.

- Suppose $R = S^*$ for some regular expression S. There are two subcases to consider.
  - If $L(S) = \varnothing$, then $L(R) = \varnothing^* = \{\varepsilon\}$, so R is represented by the $\varnothing$-free regular expression ε.
  - Otherwise, The inductive hypothesis implies that there is a $\varnothing$-free regular expression S' such that $L(S') = L(S)$. The regular expression $(S')^*$ is $\varnothing$-free and equivalent to R.

In all cases, either $L(R) = \varnothing$ or R is equivalent to some $\varnothing$-free regular expression R'. $\square$

2.9 Not Every Language is Regular

You may be tempted to conjecture that *all* languages are regular, but in fact, the following cardinality argument *almost all* languages are *not* regular. To make the argument concrete, let's consider languages over the single-symbol alphabet $\{\diamond\}$.

- Every regular expression over the one-symbol alphabet $\{\diamond\}$ is itself a string over the seven-symbol alphabet $\{\diamond, +, (,), *, \varepsilon, \varnothing\}$. By interpreting these symbols as the digits 1 through 7, we can interpret any string over this larger alphabet as the base-8 representation of some unique integer. Thus, the set of all regular expressions over $\{\diamond\}$ is *at most* as large as the set of integers, and is therefore countably infinite. It follows that the set of all regular *languages* over $\{\diamond\}$ is also countably infinite.

- On the other hand, for any real number $0 \leq \alpha < 1$, we can define a corresponding language

$$L_\alpha = \left\{ \diamond^n \;\middle|\; \alpha 2^n \bmod 1 \geq 1/2 \right\}.$$

  In other words, L_α contains the string $\diamond^n$ if and only if the $(n+1)$th bit in the binary representation of α is equal to 1. For any distinct real numbers $\alpha \neq \beta$, the binary representations of α and β must differ in some bit, so $L_\alpha \neq L_\beta$. We conclude that the set of ***all*** languages over $\{\diamond\}$ is *at least* as large as the set of real numbers between 0 and 1, and is therefore uncountably infinite.

We will see several explicit examples of non-regular languages in later lectures. In particular, the set of all regular expressions over the alphabet $\{0, 1\}$ is itself a non-regular language over the alphabet $\{0, 1, +, (,), *, \varepsilon, \varnothing\}$!

Exercises

1. (a) Prove that $\varnothing \bullet L = L \bullet \varnothing = \varnothing$, for every language L.
   (b) Prove that $\{\varepsilon\} \bullet L = L \bullet \{\varepsilon\} = L$, for every language L.
   (c) Prove that $(A \bullet B) \bullet C = A \bullet (B \bullet C)$, for all languages A, B, and C.
   (d) Prove that $|A \bullet B| \le |A| \cdot |B|$, for all languages A and B. (The second $\cdot$ is multiplication!)
      i. Describe two languages A and B such that $|A \bullet B| < |A| \cdot |B|$.
      ii. Describe two languages A and B such that $|A \bullet B| = |A| \cdot |B|$.
   (e) Prove that L^* is finite if and only if $L = \varnothing$ or $L = \{\varepsilon\}$.
   (f) Prove that if $A \bullet B = B \bullet C$, then $A^* \bullet B = B \bullet C^* = A^* \bullet B \bullet C^*$, for all languages A, B, and C.
   (g) Prove that $(A \cup B)^* = (A^* \bullet B^*)^*$, for all languages A and B.

2. Recall that the reversal w^R of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

   The reversal L^R of any language L is the set of reversals of all strings in L:

$$L^R := \left\{ w^R \;\middle|\; w \in L \right\}.$$

   (a) Prove that $(A \bullet B)^R = B^R \bullet A^R$ for all languages A and B.
   (b) Prove that $(L^R)^R = L$ for every language L.
   (c) Prove that $(L^*)^R = (L^R)^*$ for every language L.

3. Prove that each of the following regular expressions is equivalent to $(\mathtt{0} + \mathtt{1})^*$.

   (a) $\varepsilon + \mathtt{0}(\mathtt{0} + \mathtt{1})^* + \mathtt{1}(\mathtt{1} + \mathtt{0})^*$
   (b) $\mathtt{0}^* + \mathtt{0}^*\mathtt{1}(\mathtt{0} + \mathtt{1})^*$
   (c) $((\varepsilon + \mathtt{0})(\varepsilon + \mathtt{1}))^*$
   (d) $\mathtt{0}^*(\mathtt{1}\mathtt{0}^*)^*$
   (e) $(\mathtt{1}^*\mathtt{0})^*(\mathtt{0}^*\mathtt{1})^*$

4. For each of the following languages in $\{\mathtt{0}, \mathtt{1}\}^*$, describe an equivalent regular expression. There are infinitely many correct answers for each language. (This problem will become significantly simpler after we've seen finite-state machines.)

   (a) Strings that end with the suffix $\mathtt{0}^9 = \mathtt{000000000}$.
   (b) All strings except $\mathtt{010}$.
   (c) Strings that contain the substring $\mathtt{010}$.
   (d) Strings that contain the subsequence $\mathtt{010}$.

(e) Strings that do not contain the substring `010`.

(f) Strings that do not contain the subsequence `010`.

(g) Strings that contain an even number of occurrences of the substring `010`.

(h) Strings in which each run of `0`s has even length. (A *run* is a maximal substring in which all symbols are equal. For example, the string `001110000001` consists of four runs.)

*(i) Strings that contain an even number of occurrences of the substring `000`.

(j) Strings in which every occurrence of the substring `00` appears before every occurrence of the substring `11`.

(k) Strings w such that *in every prefix of* w, the number of `0`s and the number of `1`s differ by at most 1.

*(l) Strings w such that *in every prefix of* w, the number of `0`s and the number of `1`s differ by at most 2.

*(m) Strings in which the number of `0`s and the number of `1`s differ by a multiple of 3.

*(n) Strings that contain an even number of `1`s and an odd number of `0`s.

★(o) Strings that represent a number divisible by 5 in binary.

5. For any string w, define *stutter*(w) as follows:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet stutter(x) & \text{if } w = ax \text{ for some symbol } a \text{ and string } x \end{cases}$$

Let L be an arbitrary regular language.

(a) Prove that the language $stutter(L) = \{stutter(w) \mid w \in L\}$ is also regular.

*(b) Prove that the language $stutter^{-1}(L) = \{w \mid stutter(w) \in L\}$ is also regular.

(This problem will be easier after the next chapter.)

6. Recall that the ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

$$w^R = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x \bullet a & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

The reversal L^R of a language L is defined as the set of reversals of all strings in L:

$$L^R := \left\{ w^R \;\middle|\; w \in L \right\}$$

(a) Prove that $(L^*)^R = (L^R)^*$ for every language L.

(b) Prove that the reversal of any regular language is also a regular language. (You may assume part (a) even if you haven't proved it yet.)

You may assume the following facts without proof:

- $L^* \bullet L^* = L^*$ for every language L.

- $(w^R)^R = w$ for every string w.
- $(x \bullet y)^R = y^R \bullet x^R$ for all strings x and y.

[Hint: Yes, all three proofs use induction, but induction on what? And yes, all ***three*** *proofs.]*

7. This problem considers two special classes of regular expressions.

   - A regular expression R is ***plus-free*** if and only if it never uses the $+$ operator.
   - A regular expression R is ***top-plus*** if and only if either
     - R is plus-free, or
     - $R = S + T$, where S and T are top-plus.

   For example, $\mathtt{1}((\mathtt{0}^*\mathtt{10})^*\mathtt{1})^*\mathtt{0}$ is plus-free and (therefore) top-plus; $\mathtt{01}^*\mathtt{0} + \mathtt{10}^*\mathtt{1} + \varepsilon$ is top-plus but not plus-free, and $\mathtt{0}(\mathtt{0} + \mathtt{1})^*(\mathtt{1} + \varepsilon)$ is neither top-plus nor plus-free.

   Recall that two regular expressions R and S are ***equivalent*** if they describe exactly the same language: $L(R) = L(S)$.

   (a) Prove that for any top-plus regular expressions R and S, there is a top-plus regular expression that is equivalent to RS.

   (b) Prove that for any top-plus regular expression R, there is a ***plus-free*** regular expression S such that R^* and S^* are equivalent.

   (c) Prove that for any regular expression, there is an equivalent top-plus regular expression.

   You may assume the following facts without proof, for all regular expressions A, B, and C:

   - $A(B + C)$ is equivalent to $AB + AC$.
   - $(A+B)C$ is equivalent to $AC + BC$.
   - $(A+B)^*$ is equivalent to $(A^*B^*)^*$.

8. (a) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R) = \varnothing$.

   (b) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R) = \{\varepsilon\}$. *[Hint: Use part (a).]*

   (c) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)$ is finite. *[Hint: Use parts (a) and (b).]*

   In each problem, assume you are given R as a regular expression *tree*, not just a raw string.