For regular expressions $\alpha, \beta$, if $L(\alpha) = L(\beta)$, we write $\alpha \equiv \beta$ and say that $\alpha$ and $\beta$ are equivalent. The relation $\equiv$ on regular expressions is an equivalence relation; that is, it is

- reflexive: $\alpha \equiv \alpha$ for all $\alpha$;
- symmetric: if $\alpha \equiv \beta$, then $\beta \equiv \alpha$; and
- transitive: if $\alpha \equiv \beta$ and $\beta \equiv \gamma$, then $\alpha \equiv \gamma$.

If $\alpha \equiv \beta$, one can substitute $\alpha$ for $\beta$ (or vice versa) in any regular expression, and the resulting expression will be equivalent to the original.

Here are a few laws that can be used to simplify regular expressions.

$$
\begin{align*}
\alpha + (\beta + \gamma) & \equiv (\alpha + \beta) + \gamma \quad (9.1) \\
\alpha + \beta & \equiv \beta + \alpha \quad (9.2) \\
\alpha + \emptyset & \equiv \alpha \quad (9.3) \\
\alpha + \alpha & \equiv \alpha \quad (9.4) \\
\alpha(\beta\gamma) & \equiv (\alpha\beta)\gamma \quad (9.5) \\
\epsilon\alpha & \equiv \alpha\epsilon \equiv \alpha \quad (9.6) \\
\alpha(\beta + \gamma) & \equiv \alpha\beta + \alpha\gamma \quad (9.7) \\
(\alpha + \beta)\gamma & \equiv \alpha\gamma + \beta\gamma \quad (9.8) \\
\emptyset\alpha & \equiv \alpha\emptyset \equiv \emptyset \quad (9.9) \\
\epsilon + \alpha\epsilon^* & \equiv \alpha^* \quad (9.10) \\
\epsilon + \alpha^*\epsilon & \equiv \alpha^* \quad (9.11) \\
\beta + \alpha\gamma \leq \gamma \Rightarrow \alpha^*\beta \leq \gamma \quad (9.12) \\
\beta + \gamma\alpha \leq \gamma \Rightarrow \beta\alpha^* \leq \gamma \quad (9.13)
\end{align*}
$$

In (9.12) and (9.13), $\leq$ refers to the subset order:

$$
\begin{align*}
\alpha \leq \beta & \overset{\text{def}}{\iff} L(\alpha) \subseteq L(\beta) \\
& \iff L(\alpha + \beta) = L(\beta) \\
& \iff \alpha + \beta \equiv \beta.
\end{align*}
$$

Laws (9.12) and (9.13) are not equations but rules from which one can derive equations from other equations. Laws (9.1) through (9.13) can be justified by replacing each expression by its definition and reasoning set theoretically.

Here are some useful equations that follow from (9.1) through (9.13) that you can use to simplify expressions.

$$
\begin{align*}
(\alpha\beta)^* & \equiv \alpha(\beta\alpha)^* \quad (9.14) \\
(\alpha^*\beta)^* & \equiv (\alpha + \beta)^* \quad (9.15) \\
\alpha^*(\beta\alpha^*)^* & \equiv (\alpha + \beta)^* \quad (9.16) \\
(\epsilon + \alpha)^* & \equiv \alpha^* \quad (9.17) \\
\alpha\alpha^* & \equiv \alpha^*\alpha \quad (9.18)
\end{align*}
$$

An interesting fact that is beyond the scope of this course is that all true equations between regular expressions can be proved purely algebraically from the axioms and rules (9.1) through (9.13) plus the laws of equational logic [73].
3.3 Applications of Regular Expressions

A regular expression that gives a "picture" of the patterns we want to recognize is the medium of choice for applications that search for patterns in text. The regular expressions are then compiled, behind the scenes, into deterministic or nondeterministic automata, which are then simulated to produce a program that recognizes patterns in text. In this section, we shall consider two important classes of regular-expression-based applications: lexical analyzers and text search.

3.3.1 Regular Expressions in UNIX

Before seeing the applications, we shall introduce the UNIX notation for extended regular expressions. This notation gives us a number of additional capabilities. In fact, the UNIX extensions include certain features, especially the ability to name and refer to previous strings that have matched a pattern, that actually allow nonregular languages to be recognized. We shall not consider these features here; rather, we shall only introduce the notations that allow complex regular expressions to be written succinctly.

The first enhancement to the regular-expression notation concerns the fact that most real applications deal with the ASCII character set. Our examples have typically used a small alphabet, such as (0, 1). The existence of only two symbols allowed us to write succinct expressions like 0*1 for "any character." However, if there were 128 characters, say, the same expression would involve listing them all, and would be highly inconvenient to write. Thus, UNIX regular expressions allow us to write character classes to represent large sets of characters as succinctly as possible. The rules for character classes are:

- The symbol . (dot) stands for "any character."
- The sequence [a1,a2...a4] stands for the regular expression

\[ a_1 + a_2 + \cdots + a_4 \]

This notation saves about half the characters, since we don't have to write the + signs. For example, we could express the four characters used in C comparison operators by [<>==].

- Between the square brackets we can put a range of the form a-z to mean all the characters from a to z in the ASCII sequence. Since the digits have codes in order, so do the upper-case letters and the lower-case letters, we can express many of the classes of characters that we really care about with just a few keystrokes. For example, the digits can be expressed [0-9], the upper-case letters can be expressed [A-Z], and the set of all letters and digits can be expressed [A-Za-z0-9]. If we want to include a minus sign among a list of characters, we can place it first or last, so it is not confused with its use to form a character range. For example, the set of digits, plus the dot, plus, and minus signs that are used to form decimal numbers may be expressed [+-0-9]. Square brackets, characters that have special meanings in UNIX regular expressions, be represented as characters by preceding them with a backslash.

- There are special notations for several of the most common groups of characters. For instance:
  a) [[:digit:]] is the set of digits, the same as [0-9].3
  b) [[:alpha:]] stands for any alphabetic character, as does [a-zA-Z].
  c) [[:alnum:]] stands for the digits and letters (alphanumeric and characters), as does [a-zA-Z0-9].

In addition, there are several operators that are used in UNIX regular expressions that we have not encountered previously. None of these extend what languages can be expressed, but they sometimes make it easier to express what we want.

1. The operator | is used in place of + to denote union.
2. The operator ? means "zero or one of." Thus, R? in UNIX is the same as + in this book's regular-expression notation.
3. The operator + means "one or more of." Thus, R+ in UNIX is the same as R* in our notation.
4. The operator [n] means "n copies of." Thus, R[5] in UNIX is the same as RRRRR.

Note that UNIX regular expressions allow parentheses to group subexpressions just as for the regular expressions described in Section 3.1.2, and the operator precedence is used (with ?, +, and [n] treated like * as far as precedence is concerned). The star operator * is used in UNIX (without being a superset of course) with the same meaning as we have used.

3.3.2 Lexical Analysis

One of the oldest applications of regular expressions was in specifying the interface of a compiler called a "lexical analyzer." This component of a compiler program recognizes all tokens, those substrings of consecutive characters that belong together logically. Keywords and identifiers are examples of tokens, but there are many others.

The notation [[:digit:]] has the advantage that should some code other than A be used, including a code where the digits did not have consecutive codes, [[:digit:]] will represent 0-2 and 0-9 would represent whatever characters had codes in the range, inclusive.