Languages
As computer scientists, we concern ourselves with the study of formal languages rather than natural languages. A formal language is defined by and conforms to pre-established rules -- the term "formal" comes from the word "form". We are concerned more with the form of an object rather than its intrinsic meaning.

Notation

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabet</td>
<td>finite (non-empty) set of symbols, denoted by ( \Sigma ).</td>
</tr>
<tr>
<td>letter</td>
<td>an element of an alphabet.</td>
</tr>
<tr>
<td>empty string</td>
<td>string with no letters, denoted by ( \varepsilon ) (often denoted by ( \lambda ) or ( \Lambda )).</td>
</tr>
<tr>
<td>language (over ( \Sigma ))</td>
<td>a nonempty set of finite strings of symbols in ( \Sigma ). A language may consist of just the empty string ( \varepsilon ).</td>
</tr>
<tr>
<td>word</td>
<td>an element of a language</td>
</tr>
<tr>
<td>length(( \omega ))</td>
<td>number of letters in the word ( \omega ). For example, length(( \varepsilon )) = 0, length(abac) = 4.</td>
</tr>
</tbody>
</table>

Examples of Alphabets and Languages
1. \( \Sigma = \{a,b\} \). One example of a language over \( \Sigma \) is \( L_1 \) = all strings of a's and b's which begin with a. Another example is the set \( L_2 = \{a^i \cdot b \mid i \geq 1\} \).
2. \( \Sigma = \{1,0\} \). \( \Sigma^* = \{\text{all bit strings, including the empty string}\} \). Let \( L = \{\text{all even binary numbers, where there are no leading 0's except for the number 0}\} = \{0, 10, 110, 1110, ... \} \)
3. \( \Sigma = \{a, \ldots, z\} \). There are 26 strings of length 1, \( 26^2 \) strings of length 2, \( 26^3 \) strings of length 3, etc. Define the language \( L \) to contain all finite, nonempty strings that can be formed from lowercase letters.
4. \( \Sigma = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\} \). One language over \( \Sigma \) is the set of all strings of digits in which the first digit in the string is not 0. This language looks like the set of positive integers.

Operations on Words and Languages

*Concatenation*. If \( x \) and \( y \) are two words (strings), then the concatenation \( xy \) of \( x \) and \( y \) is \( x \) and \( y \) written side-by-side. For example, if \( x=abc \) and \( y=da \), then \( xy=abcd \). We can also form the concatenation of two languages. For example, suppose \( L_1 = \{\text{all strings of a's} \} \) and \( L_2 = \{\text{all strings of b's}\} \). Then, \( L_1 \cdot L_2 = \{\text{all strings of a's and b's in which all the a's appear before all the b's}\} \)

Given that \( A \) and \( B \) are alphabets then the following statements are true:
1. \( (A^*)^* = A^* \)
2. \( A^*A^* = A^* \)
3. \( A^* = AA^* \)
4. \( (A \cup B)^* = (A^* \cup B^*) = (A^*B^*)^* = (B^*A^*)^* \)
Proof of \(A^*A^* = A^*\):
In order to prove \(A^*A^* = A^*\), we must show
1. \(w \in A^* \rightarrow w \in A^*A^*\) and
2. \(w \in A^*A^* \rightarrow w \in A^*\)

1) \(A^*A^* = A_0^* \cup A_1^* \cup \ldots \cup A_n^*\) but \(A_0^* = \varepsilon A^* = A^*\)

\[\therefore A^* \subseteq A^*A^*\] and \(\therefore w \in A^* \rightarrow w \in A^*A^*\)

2) Show that \(w^k \in A^*A^*\), which means a word of length \(k\) from the alphabet \(A\) (prove by induction).

**Basis:** \(k = 0\), \(w^0 = \varepsilon\)
\(\varepsilon \in A^*\) and \(\varepsilon \in A^*A^*\) true for \(k = 0\)

**Assume true for** \(k\): \(w^k \in A^*A^*\) \(\rightarrow\) \(w^k \in A^*\) and \(w^k \in A^*A^*\)

**Prove true for** \((k+1): w^{k+1} \in A^*A^*\)
\(w^{k+1} = ww^k\), \(w^k \in A^*\) and \(w \in A^*\) because \(A^* = [A_0 \cup A_1 \cup \ldots \cup A_n]\).
Therefore, \(w^{k+1} \in A^*\) by the laws of concatenation. We proved earlier \(A^* \subseteq A^*A^*\) so we know \(w^{k+1} \in A^*A^*\). \(\therefore w \in A^*A^* \rightarrow w \in A^*\)

**Kleene star**. The closure or Kleene star of the alphabet \(\Sigma\), denoted by \(\Sigma^*\), is the (infinite) set of all finite strings of letters of \(\Sigma\) including the empty string \(\varepsilon\). \(\Sigma^*\) can be defined recursively as follows:

(i) \(\varepsilon\) belongs to \(\Sigma^*\).
(ii) If \(\omega\) is a string in \(\Sigma^*\) and \(a\) is in \(\Sigma\), then \(a\omega\) belongs to \(\Sigma^*\).

**Kleene Star Examples:**
1. \(\Sigma = \{a\}\). \(\Sigma^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\} = \{a^i \mid i \geq 0\} \) (where \(a^0 = \varepsilon\)).
2. \(\Sigma = \{a, b\}\). \(\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \ldots\}\).
3. \(\Sigma = \{1, 0\}\). \(\Sigma^* = \{\text{all bit strings, including the empty string}\}\).

In addition to \(\Sigma^*\), we have the notion of \(\Sigma^+\) and \(\Sigma^k\). \(\Sigma^+\) is the same as \(\Sigma^*\), except that it does not contain \(\varepsilon\). So we can write \(\Sigma^1\) as \(\Sigma^+\). Using the alphabet from the first example given above, \(\Sigma^1 = \{a^i \mid i \geq 1\}\). \(\Sigma^k\) is the subset of \(\Sigma^*\) of words of length \(k\). For example, \(\Sigma^0 = \varepsilon\), \(\Sigma^1 = \text{all words of length } 1\) (which is \(\Sigma\)), and \(\Sigma^2 = \text{all words of length } 2\).

**Union**. If \(L_1\) and \(L_2\) are two languages, then we can create a third language \(L_3 = L_1 \cup L_2 = \{\omega \mid \omega\ \text{is a word in } L_1 \text{ or } \omega\ \text{is a word in } L_2\}\). This is the union of the languages \(L_1\) and \(L_2\). For example, if \(L_1 = \{a\}^*\) and \(L_2 = \{b\}^*\), then \(L_1 \cup L_2 = \{\omega \mid \omega\ \text{is a finite string of all a's or all b's}\}\). Note how this differs from the example given in the explanation on concatenation.

**Regular Expressions**

Regular expressions allow us to describe particular types of languages in a convenient shorthand notation. A language is called regular if it can be represented by a regular expression. Suppose \(S = \{a\}\). Then define \(L = S^* = \{a^i \mid i \geq 0\}\). This is, of course, the Kleene star of \(S\). Rather than apply the Kleene star to a set, we will apply it directly to the letter \(a\) to mean a set of sequences of \(a\)'s of arbitrary finite length. That is, \(a^*\) represents \(S^*\). The expression \(a^*\) is called a regular expression. We can also use the \(*\) operation to form \(a^*\), so \(a^* = aa^*\).

We can form the union and concatenation of two regular languages. For example, we can form the regular expressions \(b^* \cup c\) and \(a^*b\). The former represents the set \(\{\omega \mid \omega\ \text{is a string of all b's or } \omega\ \text{is the singleton string } c\}\). The latter represents the set \(\{a^i b \mid i \geq 0\}\). A recursive definition of regular expressions is given below.
Regular expression definition. Let $\Sigma$ be an alphabet (finite set of symbols). Then

(i) $\emptyset, \varepsilon,$ and $a$ are regular expressions for each $a$ in $\Sigma$.
(ii) If $x$ and $y$ are regular expressions, then so are $xy, x \cup y,$ and $x^*.$

To see that $S = a(a \cup b)^*bb$ is regular for $a$ and $b$ in $\Sigma,$ we can apply the definition of regular expressions.

1. $a$ is regular
2. $b$ is regular
3. $a \cup b$ is regular
4. $(a \cup b)^*$ is regular
5. $a(a \cup b)^*bb$ is regular

To see if $S = \{a^i b^i \mid i \geq 1 \}$ is regular for $a$ and $b$ in $\Sigma,$ we can apply the definition of regular expressions.

1. $a$ is regular
2. $b$ is regular
3. $a^i$ is not regular
4. $b^i$ is not regular

∴ $\{a^i b^i \mid i \geq 1 \}$ is not regular.

Remember that a regular expression is simply a way of describing a set. For example, the set $\{a\}^*\{b, c\}\{a\}^*$ can be written $a^*(b \cup c)aa^*.$ Or, we can write $a^*(b \cup c)a^*.$ Observe that regular expressions are not unique. That means that there may be several different expressions that represent the same set. Examples in addition to the one above are $a^*a^* = a^*$ and $a^*(a^*ba^*ba^*)^* = a^*(a^*ba^*)a^*.$

Examples of Regular Expressions
1. $a^*(b \cup \varepsilon)a^* =$ strings that contain at most one b and the rest a's.
2. $a(ab)^*b =$ strings that start with aa, end with bb, and have alternating substrings ba in between.
3. $(aa \cup bb \cup ab \cup ba)^* =$ even length strings of a's and b's.
4. $(a \cup b)(aa \cup bb \cup ab \cup ba)^* =$ odd length strings of a's and b's.
5. If $\Sigma = \{a, b, c, d\},$ then $\Sigma^*$ can be represented by $(a \cup b \cup c \cup d)^*.$
6. The regular expression $\varepsilon$ represents the language consisting of just the empty string and nothing else.

Which of the following strings are in the given language?

Language: $a^*(ba^*)^*(b \cup a)$

a? yes
b? yes
$\varepsilon$? no
aa? yes
ab? no
bb? no
aba? no
abab? no
abaabaab? yes
aaaaaaa? yes
BNF Grammars

Once we have the words for a formal language, can we put them together in a meaningful way? What sort of strings are acceptable? The rules under which one can construct acceptable strings constitute the "grammar" of a language. Syntactically correct strings in computer languages are called programs. Since the syntax rules for most programming languages are fairly complicated, we will restrict our attention to simple analogues.

We will consider grammars that are defined in the following manner. A grammar consists of a set of terminals (elements of an alphabet $\Sigma$), a set of variables (which are not terminals), a special variable called a start symbol, and a set of production rules that allow one to substitute variables and/or terminals for variables. Below is an example of a language and its associated grammar.

terminals:
$$\Sigma = \{\text{John, the, big, fried, fish, dog, caught, ate}\}$$

variables:
$$\langle\text{sentence}\rangle, \langle\text{noun}\rangle, \langle\text{noun phrase}\rangle, \langle\text{verb}\rangle, \langle\text{article}\rangle, \langle\text{adjective list}\rangle, \langle\text{adjective}\rangle$$

start symbol:
$$\langle\text{sentence}\rangle$$

production rules:
$$\langle\text{sentence}\rangle \rightarrow \langle\text{noun phrase}\rangle\langle\text{verb}\rangle\langle\text{noun phrase}\rangle$$
$$\langle\text{noun phrase}\rangle \rightarrow \langle\text{article}\rangle\langle\text{adjective list}\rangle\langle\text{noun}\rangle | \text{John}$$
$$\langle\text{article}\rangle \rightarrow \text{the}$$
$$\langle\text{noun}\rangle \rightarrow \text{fish} | \text{dog}$$
$$\langle\text{verb}\rangle \rightarrow \text{caught} | \text{ate}$$
$$\langle\text{adjective list}\rangle \rightarrow \varepsilon | \langle\text{adjective}\rangle | \langle\text{adjective}\rangle\langle\text{adjective list}\rangle$$
$$\langle\text{adjective}\rangle \rightarrow \text{big} | \text{fried}$$

Variables are enclosed in "<=>. The arrows "->" indicate that the variable is to be replaced by whatever is on the right-hand side. A "|" mark means that one has a choice of items to substitute for the variable. This format for denoting a grammar--arrows, vertical bars, terminals, etc.--is called the Backus Normal Formal or Backus-Naur Form.

A grammar generates a string if, starting with the start symbol, one can produce the string by successively replacing variables with corresponding symbols according to the production rules until only terminals remain. There are many syntactically correct sentences that can be formed from this grammar. Examples include:

*John caught the big fish*

*the big fried dog ate John*

These examples need to make sense only in form. The intrinsic idea that the last sentence is probably nonsensical is not important. To see how to form the last sentence, substitute according to the production rules.

$$\langle\text{sentence}\rangle \rightarrow \langle\text{noun phrase}\rangle\langle\text{verb}\rangle\langle\text{noun phrase}\rangle$$
$$\rightarrow \langle\text{article}\rangle\langle\text{adjective list}\rangle\langle\text{noun}\rangle\langle\text{verb}\rangle\langle\text{noun phrase}\rangle$$
$$\rightarrow \text{the}\langle\text{adjective list}\rangle\langle\text{noun}\rangle\langle\text{verb}\rangle\langle\text{noun phrase}\rangle$$
$$\rightarrow \text{the big}\langle\text{adjective list}\rangle\langle\text{noun}\rangle\langle\text{verb}\rangle\langle\text{noun phrase}\rangle$$
$$\rightarrow \text{the big}\langle\text{adjective}\rangle\langle\text{noun}\rangle\langle\text{verb}\rangle\langle\text{noun phrase}\rangle$$
Such a substitution process is called a derivation of the string. By the expression "language generated by a grammar," what we mean is the set of all strings of terminal symbols that can be derived from the start symbol by applying the rules of the grammar.

Examples of Grammars

1. \( \Sigma = \{a, b\} \quad V = \{S\} \)
   \[ S \rightarrow aSb | \epsilon \]
   The language generated is \( \{a^i b^i \mid i \geq 0\} \). (This language cannot be represented as a regular expression.)

2. \( \Sigma = \{a, b, c\} \quad V = \{S, A, B\} \)
   \[ S \rightarrow bB | aA \]
   \[ A \rightarrow aS | a \]
   \[ B \rightarrow bS | b \]
   The language generated is \((aa \cup bb)^+\).

3. \( \Sigma = \{a, b, c\} \quad V = \{S, M, F\} \)
   \[ S \rightarrow bS | aM \]
   \[ M \rightarrow bS | aF \]
   \[ F \rightarrow bF | aF | \epsilon \]
   The language generated is the set of all words containing the substring aa, that is the language of the regular expression \((a \cup b)^* aa (a \cup b)^*\).

4. \( \Sigma = \{a, ..., z, A, ..., Z, 0, 1, ..., 9, _\} \quad V = \{\text{identifier, alphanum, letter, tail, digit}\} \)
   \[ \text{<identifier>} \rightarrow <\text{<letter> <tail>} | <\text{<alphanum> <tail}> | _ <\text{<alphanum> <tail}> | \epsilon \]
   \[ <\text{<alphanum>}} \rightarrow <\text{<letter> | <digit>}
   \[ <\text{<letter>}} \rightarrow a | b | ... | z | A | B | ... | Z
   \[ <\text{<digit>}} \rightarrow 0 | 1 | ... | 9
   The language generated is all UCSD Pascal identifiers and reserved words.

The Pascal language can be defined by a BNF grammar. Some examples of variables and terminals are:

variables: identifier, block, header, statement, ...

terminals: program, var, record, begin, end, with, if, then, else, repeat, a, b,...,z,...

One most often sees the production rules in syntax diagrams, usually at the rear of every Pascal test. For example:

\[ <\text{<with statement}> \rightarrow \text{with} <\text{<record list>} do <\text{<statement}>}
\]
\[ <\text{<statement}> \rightarrow <\text{<simple statement}> | <\text{<compound statement}>}
\]
\[ <\text{<compound statement}> \rightarrow \text{begin} <\text{<statement}> end}
\]
\[ <\text{<simple statement}> \rightarrow <\text{<assignment statement}> | <\text{<conditional statement}> | \]
\[ <\text{<case statement}> | <\text{<loop statement}> | <\text{<with statement}> | <\text{<null statement}>}

The language of the BNF grammar of Pascal is the set of syntactically correct Pascal programs. To determine whether or not a string is such a program, it must be derivable from the start symbol using the rules of grammar. An algorithm that determines whether or not a string is a program (whether or not a string is in a language) is called a parser.
Finite State Machines

Computer scientists are interested in determining just what computers can do. In order to figure this out, they have constructed theoretical machines with certain constraints and tried to see what kinds of problems those machines could solve. There are many different theoretical models, with many different types of constraints. As you might expect, different models can solve different kinds of problems. Some models are more powerful than others in that they can solve the others' problems in addition to ones that the others cannot solve.

We will look at one type of theoretical machine. But before we define that type of machine, we first look at three examples of problems that type of machine is capable of solving.

Example Application Problems
1. A touch key lock opens when the combination 5-1-3-0 is entered on its 10-digit key pad. We construct a state transition diagram (or state diagram or transition diagram) to analyze the operation of the lock. The states are circles, numbered 1-5. The transitions are the arcs. They tell what kind of input will take you from one state to another. Start at state 1. After a 5 is entered (it's possible that it is not entered until several non-5's are entered), proceed to state 2. If a 1 is entered immediately after the 5, proceed to state 3. Otherwise, if a 5 is entered, remain at state 2. (This indicates the first digit of the correct combination was just reentered.) If you are at state 2 and the next digit is neither a 5 nor a 1, return to state 1 in order to start all over again. The only way to reach state 5, where the lock opens, is to enter a sequence of digits that ends with 5130. When state 5 is reached, there is no need to enter more digits. (Additional digits would indicate the beginning of a new problem.)
2. A newspaper vending machine sells newspapers for 25 cents. The only coins that the machine accepts are nickels, dimes, and quarters. (If you give the machine more money than needed to add to 25 cents, then it takes your money and does not return change.) The following state transition diagram is a model for the actual physical machine. The states are labeled according to the amount of money that one needs to pay in order to get a paper. So the starting state is state 25. The arcs are labeled according to the coins that are inserted.
3. The machine below is capable of determining if a string is a nonnegative real constant in Pascal. (Assume that a real number cannot begin with a + or - sign, and that the string that is input to the machine contains only digits, +, -, ., and/or the letter E. Any other character indicates the end of the string.) To operate this machine, enter the string at state 1. If the first character is a digit, proceed to state 2 with the rest of the string. If at the end of processing the entire string you end up at states 5 or 7, then the string is a real number. Otherwise, if you get stuck at a state (because there is no arrow out of that state corresponding to the next character to process) or if you run out of characters, then the string is not a nonnegative, noninteger real Pascal constant.
The state transition diagrams are examples of finite state machines. What we need to know in order to model such a machine is not how it is constructed but how it works. What is required of a finite state machine are the following:

- an input alphabet $\Sigma$
- a finite set of states
- a special start state
- at least one final state
- a set of transitions that tell how to get from one state to the next

If we restrict our set of transitions so that for each state and each element of $\Sigma$, there is exactly one state to which one can proceed next, then we have what is called a deterministic finite automaton. It's deterministic because when you are at one state, given the next symbol to process, there is exactly one state (as opposed to several) to which you can proceed next. It's finite because there are finitely many states. It's an automaton, because it is a machine. Such machines are language acceptors. When you process a particular string on a particular automaton, either you end up in a final accepting state or you do not. The language accepted by the finite state machine is the set of all strings which take the machine to a final state.

How do our examples fit into the definition of finite state machines?

1. $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
   $V = \{1,2,3,4,5\}$
   Start state = \{1\}
   Accepting states = \{5\}
   The transitions are indicated by the labeled arrows.

2. $\Sigma = \{25 \text{ cents}, 10 \text{ cents}, 5 \text{ cents}\}$
   $V = \{25, 20, 15, 10, 5, 0\}$
   Start state = \{25\}
   Accepting states = \{0\}

3. $\Sigma = \{0,1,2,3,4,5,6,7,8,9,\text{E, . ,+,-}\}$
   $V = \{1,2,3,4,5,6,7,8\}$
   Start state = \{1\}
   Accepting states = \{5, 7\}
   Note that state 8 is a dead state. It is impossible to leave state 8 after entering. In that case, the string that is being processed is not acceptable.

Finite state machines are sometimes referred to as language acceptors. For instance, in the vending machine example, the strings that are accepted are any in which the sum of the coins is at least 25 cents. So we could define the language accepted as $L = \{\omega \mid \text{the "letters" in } \omega \text{ sum to at least 25 cents}\}$. We can construct such machines to accept regular expressions. The finite state machines given below are all acceptors of regular languages.

Examples of finite state machines that accept regular expressions:

1. $a^*(b \cup \varepsilon)a^* = \text{strings that contain at most one } b \text{ and the rest } a's.$
2. \((ab)^*b = \) strings that start with aa, end with bb, and have alternating substrings \(ba\) in between.

3. \((aa \cup bb \cup ab \cup ba)^* = \) even length strings of a’s and b’s.

4. \((a \cup b)(aa \cup bb \cup ab \cup ba)^* = \) odd length strings of a’s and b’s.
Some simple expressions

1. $a \cup b \cup c$

2. $a^*$

3. $a^*$

4. $ab$

5. $(ab)^*$

6. $(ab)^*$
7. $a^*b^*$

This is a non-deterministic finite state machine. In order to make it deterministic, one needs to remove the epsilon. One can remove the epsilon by using a terminal which leaves the state into which the epsilon arrives and connect the epsilon to the state where this terminal goes. In this case, use $b$ since it leaves state 2 and goes to state 2.

8. $a^*b^*c^*$

Give a regular expression to define the language accepted by the following finite state machine:
(01*) ∪ (1(0 ∪ 1)*)

There are other examples of machine models that are important. Among the most well known of these machine models are machines with stacks (pushdown automata) and Turing machines, which are general enough to model any computer.