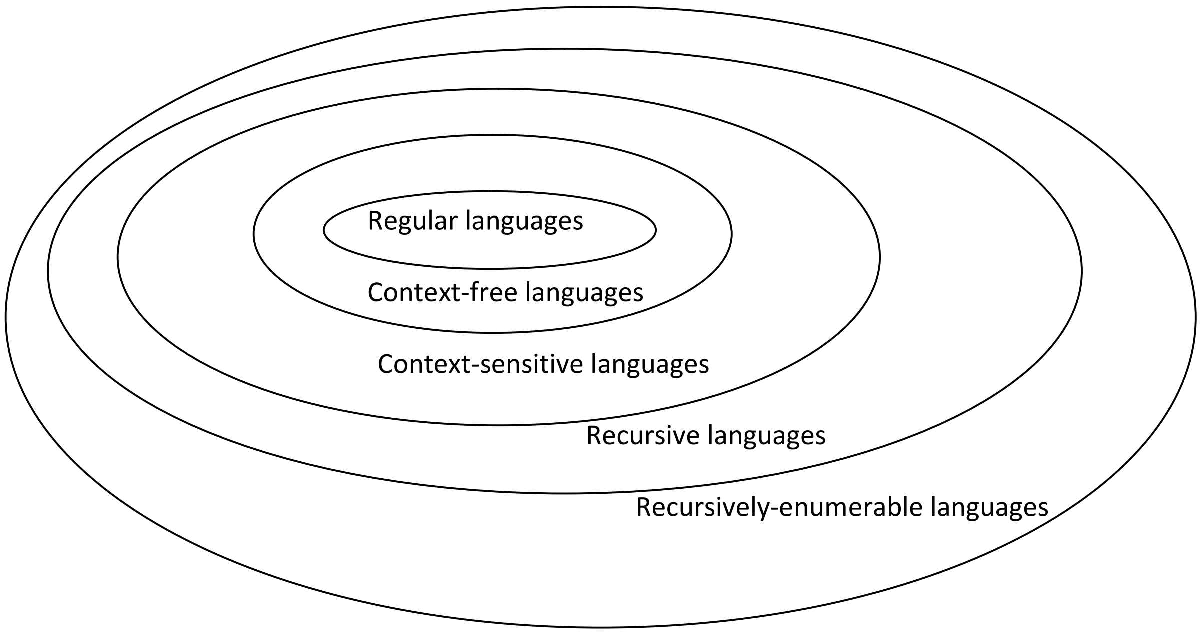
Languages (Introduction, Regular Expressions) Carol Zander

This topic is about modeling computation. Models are important for several reasons. They let us think about whether problems can even be solved using a computer (some can't). They also let us think abstractly about the method to solve a problem. Languages are modeled, very important from a programming language definition and compilation standpoint. All programming languages and their components are defined by different types of languages.

What is a language? **A language is a set of strings.** For example, one program you write is one long string. Or one sentence you write is another long string. Different types of languages give different powers of expression. Powers of expression refers to what can be represented using that form of expression, i.e., what you can specify using the given representation. If you think of representation of a language as sort of an algebra, some representations allow us to generate more complex strings than others. We can express more complex thoughts using English than we can in C++.

Noam Chomsky (the well-known political activist) is by profession a prominent linguist. He was interested in formal languages and whether or not they could represent properties of human language. While he was not interested in computers, he came up with a language hierarchy that turned out to be useful in computer science (for the compilation process). This is called the Chomsky hierarchy. It captures the power of expression of languages. A Venn diagram represents the power of these languages, the weakest at the center. Regular languages are subsets of every other category. For example, every language that is regular is context-free, but not conversely.



Chomsky defined four language categories:

• Regular	(Type 3)
• Context-free	(Type 2)
• Context-sensitive	(Type 1)
• Recursively-enumerable	(Type 0)

Regular expressions

The simplest type of language is the regular language. Regular languages are generated by regular expressions. These are commonly used for pattern matching. Regular expressions are usually defined recursively. All languages are defined over some alphabet, say A (often you will see sigma, Σ, used for the alphabet). For an alphabet A, the regular expressions over this alphabet (or set) consist of the following:

- λ (the empty string; sometimes epsilon, ε, is used)
- Any character in A
- If X and Y are regular expressions over A, then so are the following, listed from highest operator precedence to lowest:

highest precedence	(X)	Parentheses are used to group subexpressions
↓	X*	Kleene star operator: zero or more repetitions of X
	XY	Concatenation
lowest precedence	X|Y	Alternation, OR (sometimes + is used)

If you google regular expressions, you will find them defined within some language. Because of this, more syntax is given to make it easier to write regular expressions. We will only use the above operators, i.e., you are not allowed to use anything except what is given above.

Let's start with a simple alphabet A = {a, b, c}. The regular expression (a|b|c)* would generate all possible strings of a, b, and c. The regular expression c (a|b|c)*c would generate all possible strings of a, b, and c that start and end in a 'c' .

Let's generate a regular expression for **unsigned binary numbers**. For this case, our alphabet A = {0, 1}.
Our expression is: (0|1) (0|1)*

The first term forces our string to start with a digit (note that the empty string is not in the language). The second term allows us to have as many digits, in any order, as we want.

Now generate a regular expression for **signed binary numbers** (the sign is optional for positive numbers.)
For this case, our alphabet is A = {+, -, 0, 1}. Our expression is: (+|-|λ) (0 |1) (0|1)*

We need the empty string (λ) in the first set of options to allow for positive numbers with an implied sign.
This could be written without λ as follows: (+|-) (0 1) (0|1)* | (0|1) (0|1)*

All the languages we've seen so far are infinite languages (an infinite number of strings can be generated). All the keywords in a computing language are tokens in that language. Suppose the alphabet are all the characters that can be used to write a program. Then a keyword such as "while" is generated by the regular expression:

while

This is a finite language with exactly one string in it.

Let's do some more interesting regular expressions for computer languages. Let's write a regular expression for identifiers with a simplified alphabet of {a, b, c, 1, 2, 3, _}. First, write a regular expression for C-like identifiers. These follow the rules:

1. Must start with a letter
2. Then there is either a letter, number, or underscore

(a|b|c) (a|b|c|1|2|3|_)*

There are housekeeping things to check when you generate a regular expression:

- Can I generate all of the strings in the language?
- Can I generate anything that isn't in the language?
- Boundary conditions warrant special consideration.

Often I ask: what is the shortest string in the language that my expression generates? Is the empty string in the language? Example boundary conditions to consider are: what can it start with? What can it end with?

Let's play a bit more with regular expressions. You start to get a feel for the expressiveness of regular expressions. You can't keep track of many things, for example, you can't count, but you can keep tract of state. For example, whether a string is even or odd is state information.

Write a regular expression for the alphabet {a,b} for all strings that have an even number of a's. The logic is that there can be a 'b' anywhere, no restrictions, but every 'a' must have a buddy 'a' to keep it even. In between the paired a's though, you must allow for other characters:

    (b | ab*a)*

Write a regular expression for strings that start with an 'a', end in 'b', and have an even number of a's in total.

    ab*a(b | ab*a)*b

How about an odd number of a's? Start with even, (b | ab*a)* and add an 'a' to make the a's odd:

    (b | ab*a)*a

The only problem with this regular expression is that strings can't end in 'b'. So fix it up to allow ending b's:

    (b | ab*a)*ab*

Now write the regular expression for strings that have an even number of a's and an even number of b's.
The a's and b's can be in any order, e.g., aabb, abaaab, ababbb, babaaaaababbab, aaaaaaaa, abbbbbbbba .

    (aa | bb | (ab|ba)(aa|bb)*(ab|ba))*

The regular expression has three terms to allow for pairs of a's, pairs of b's, or mixed up a's and b's. To build a string, allow "aa" or "bb" anywhere. To make sure the a's and b's that are mixed up have a buddy 'a' or 'b' force each "ab" or "ba" to have a matching "ab" or "ba" so the count always remains even.

To be sure all possible strings are generated, allow "aa"s and "bb"s to come between mixed up 'a' and 'b' terms. While this regular expression is hard to come up with, once you see it, it has nice symmetry. I was given a similar problem – alphabet {a,b,c}, come up with a regular expression for even a's, even b's, and even c's.
It is not small and elegant like the one above, but is excessively long and not at all interesting.

Now write a regular expression for all strings that are of the form a^nb^n, in set notation { a^nb^n | integer $n \geq 0$},
for example: λ ab aabb aaabbb aaaabbbb aaaaabbbbb

It doesn't take you long to decide this is difficult. In fact, this is a trick question as this language is not regular. There is no regular expression for a^nb^n. The language a^nb^m, where the number a's and b's don't have to necessarily be the same is generated by the regular expression a*b*, but there is no way to count characters.