

Regular expressions are great for finding or validating many types of simple patterns, for example phone numbers, email addresses, and URLs. However, they fall short when applied to patterns that can have a recursive structure, such as:

- HTML / XML open/close tags

- open/close braces `{/}` in programming languages

- open/close parentheses in arithmetical expressions

To parse these types of patterns, we need something more powerful.

We can move to the next level of formal grammars called context free grammars

Context Free Grammars

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.


A grammar is used to specify the syntax of a language.

It answers the question: What sentences are in the language and what are not?

A sentence is a finite sequence of symbols from the alphabet of the language.

The grammar we discuss here is for a context free languages.

The grammar is a context free grammar or CFG.



A grammar defines what are legal statements in a language.

A grammar has:

- Alphabet is a finite set of symbols that appear in the language
- Non-terminals symbols that can be replaced by collections of symbols found in a production (see below)
- Terminal symbols from the alphabet
- Productions replacement rules for non-terminals
- Start symbol the initial symbol from which all sentences in the language can be derived.

Note: it is usually the left hand side of the first production when a start symbol is not specifically given.

Backus-Naur Form (BNF) is a notation for expressing a CFG.

- Nonterminals are denoted by surrounding symbol with $\langle \rangle$. e.g. $\langle \text{turtle} \rangle$
- Alternation is denoted by $|$ e.g. $\text{bad } \langle \text{cats} \rangle | \text{good } \langle \text{dogs} \rangle$.

$\langle \text{animal} \rangle ::= \text{bad } \langle \text{cats} \rangle | \text{good } \langle \text{dogs} \rangle$

you could say the same thing without alternation:

$\langle \text{animal} \rangle ::= \text{bad } \langle \text{cats} \rangle$

$\langle \text{animal} \rangle ::= \text{good } \langle \text{dogs} \rangle$


- Replacement is denoted by $::=$. These are the productions.

The left hand side (lhs) of a production is the non-terminal symbol to the left of the $::=$.

The right hand side (rhs) of a production is the sequence of terminal and non-terminal symbols to the right of the $::=$.

e.g. $\langle \text{dogs} \rangle ::= \text{corgi} | \text{other}$

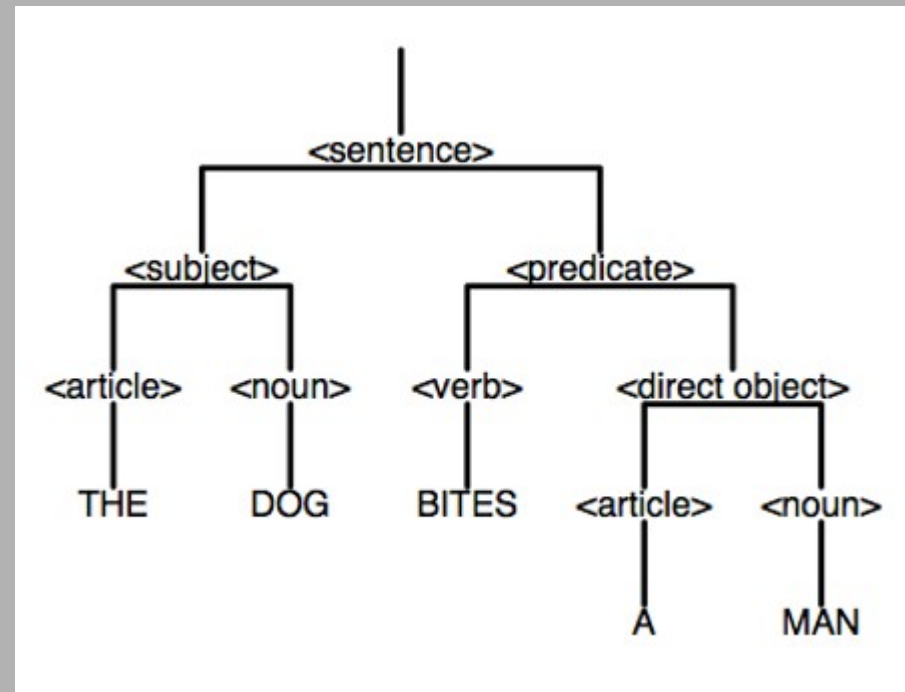
- Blanks are ignored or must be in some escaping scheme like quotes " "
- Terminals are unadorned symbols




If and only if a sentence is composed of an ordered list of elements from the alphabet and it can be derived from the start symbol by application of production rules then it is in the language defined by the grammar.

Specifically a context free grammar (CFG) is defined by a set of productions in which the left hand side of the production is a single non-terminal which may be replaced by the right hand side anywhere where the left hand side occurs, regardless of the context in which the left hand side symbol occurs. Hence "context free".

$\langle \text{sentence} \rangle ::= \langle \text{subject} \rangle \langle \text{predicate} \rangle$
 $\langle \text{subject} \rangle ::= \langle \text{article} \rangle \langle \text{noun} \rangle$
 $\langle \text{predicate} \rangle ::= \langle \text{verb} \rangle \langle \text{direct-object} \rangle$
 $\langle \text{direct-object} \rangle ::= \langle \text{article} \rangle \langle \text{noun} \rangle$
 $\langle \text{article} \rangle ::= \text{THE} \mid \text{A}$
 $\langle \text{noun} \rangle ::= \text{MAN} \mid \text{DOG}$
 $\langle \text{verb} \rangle ::= \text{BITES} \mid \text{PETS}$





Derivation is the ordered list of steps used in construction of a specific parse tree for a sentence from a grammar.

Left Most Derivation is a derivation in which the left most non-terminal is always replaced first.

Parse is to show how a sentence could be built from a grammar.

Metasymbols are symbols outside the language to prevent circularity in talking about the grammar.

Common Idioms and Hierarchical Development

Many of the linguistic structures in a computer language come from a small set of basic idioms.

Here are some basic forms for some common things you might want in your language.

Assume start symbol: <sentence>.

A grammar for a language that allows a list of X's

$$\langle \text{sentence} \rangle ::= X \mid X \langle \text{sentence} \rangle$$

This is also a grammar for that language:

$$\langle \text{sentence} \rangle ::= X \mid \langle \text{sentence} \rangle X$$

This is an example that there can be multiple correct grammars for the same language.

A Language that is a List of X's or Y's

$$\langle \text{sentence} \rangle ::= X \mid Y \mid X \langle \text{sentence} \rangle \mid Y \langle \text{sentence} \rangle$$

Here is a more hierarchical way to do the same thing:

$$\langle \text{sentence} \rangle ::= \langle \text{sub} \rangle \mid \langle \text{sub} \rangle \langle \text{sentence} \rangle$$
$$\langle \text{sub} \rangle ::= X \mid Y$$

Note that the first part handles this “listing” of the sub-parts which can either be an X or Y.

It is often clarifying to do grammars hierarchically.

Here are some grammars for two similar, but different languages:


$$\langle \text{sentence} \rangle ::= X \mid Y \mid X \langle \text{sentence} \rangle$$

is a grammar for a single X or a single Y or one or more X 's finished by a single X or Y . i.e there can't be multiple Y s at the end.

$$\langle \text{sentence} \rangle ::= X \mid Y \mid \langle \text{sentence} \rangle X$$

is a grammar for a single X or a single Y followed by one or more X 's.
i.e. only one Y not at the end.

If there was no Y we could not tell which way the string was built from the final string and so the languages would be identical i.e. a list of 1 or more X s.



A Language that is a List of X's Terminated with a Semicolon

$\langle \text{sentence} \rangle ::= \langle \text{listx} \rangle ";"$

$\langle \text{listx} \rangle ::= X \mid X \langle \text{listx} \rangle$

Note the hierarchical approach.

Hierarchical Complexification

Let's now combine the previous and make a list of sublists of Xs that end in semicolons. Note that the we describe a list of <list> and then describe how the sublists end in semicolons and then how to make a list.

Very top to bottom and hierarchical.

This will help you develop clean grammars and look for bugs such as unwanted recursion.

<sentence> ::= <list> | <sentence> <list>

<list> ::= <listx> ";"

<listx> ::= X | X <listx>

A Language Where There are not Terminal Semicolons but Separating Commas

$\langle \text{listx} \rangle ::= X \mid X ", " \langle \text{listx} \rangle$

A language where each X is followed by a terminating semicolon:
Sometimes you need to ask yourself is this a separating delimiter
or a terminating delimiter?


$\langle \text{listx} \rangle ::= X "; " \mid X "; " \langle \text{listx} \rangle$

Compare again with the separating case in which each X is separated
from the next

This is a terminating delimiter case. Hierarchically it looks like:

$\langle \text{sentence} \rangle ::= \langle \text{sub} \rangle \mid \langle \text{sentence} \rangle \langle \text{sub} \rangle$

$\langle \text{sub} \rangle ::= X ", "$



An arg list of X's, Y's and Z's: eg (X,Y)

$\langle \text{arglist} \rangle ::= "(" \text{ " } | "(" \langle \text{varlist} \rangle \text{ " } "$
 $\langle \text{varlist} \rangle ::= \langle \text{var} \rangle | \langle \text{varlist} \rangle \text{ " , " } \langle \text{var} \rangle$
 $\langle \text{var} \rangle ::= X | Y | Z$

A Simple Type Declaration: This is the grammar for a very simple C-like type declaration statement. It has a very hierarchical feel:
eg int X,Y;

$\langle \text{tdecl} \rangle ::= \langle \text{type} \rangle \langle \text{varlist} \rangle \text{ " ; " }$
 $\langle \text{varlist} \rangle ::= \langle \text{var} \rangle | \langle \text{varlist} \rangle \text{ " , " } \langle \text{var} \rangle$
 $\langle \text{var} \rangle ::= X | Y | Z$
 $\langle \text{type} \rangle ::= \text{int} | \text{bool} | \text{string}$

Augment the Type Grammar with the Keyword Static

eg static int X;

or bool x;

$\langle \text{tdecl} \rangle ::= \langle \text{type} \rangle \langle \text{varlist} \rangle ";"$

$\langle \text{varlist} \rangle ::= \langle \text{var} \rangle \mid \langle \text{varlist} \rangle "," \langle \text{var} \rangle$

$\langle \text{var} \rangle ::= X \mid Y \mid Z$

$\langle \text{type} \rangle ::= \text{static} \langle \text{basictype} \rangle \mid \langle \text{basictype} \rangle$

$\langle \text{basictype} \rangle ::= \text{int} \mid \text{bool} \mid \text{string}$

Tree Structure as Nested Parentheses

For instance consider this popular nested way to represent a tree:
(ant) or (cat, bat) or ((cat), (bat)) or ((cat, bat, dog), ant).

```
<tree> ::= "(" <list> ")" // deal with parentheses  
<list> ::= <thing> | <list> "," <thing> // do the list of things  
<thing> ::= <tree> | <name> // things are more trees or names  
<name> ::= ant | bat | cow | dog
```


Associativity and Precedence

Getting the parse tree to represent proper grouping when grouping delimiters like parentheses are missing requires that we understand associativity and precedence.

C++ has a precedence hierarchy that is over dozen levels deep.

Here is where hierarchical design again shines prominently.

A Grammar for an Arithmetic Expression

$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{addpart} \rangle \mid \langle \text{exp} \rangle - \langle \text{addpart} \rangle \mid \langle \text{addpart} \rangle$

$\langle \text{addpart} \rangle ::= \langle \text{addpart} \rangle * \langle \text{mulpart} \rangle \mid \langle \text{addpart} \rangle / \langle \text{mulpart} \rangle \mid \langle \text{mulpart} \rangle$

$\langle \text{mulpart} \rangle ::= \langle \text{group} \rangle ^ \langle \text{mulpart} \rangle \mid \langle \text{group} \rangle$

$\langle \text{group} \rangle ::= \langle \text{var} \rangle \mid (\langle \text{exp} \rangle)$

$\langle \text{var} \rangle ::= X \mid Y \mid Z$

recursion is on the right

This involves the five operators $+$, $-$, $*$, $/$, $^$ (where $^$ is exponentiation).

Operator Associativity determines the order of execution of homogeneous operators.

The first four are evaluated left to right.

That is their associativity is left to right or left associative.

Exponentiation in mathematics is done right to left, that is, it is right associative.

Operator precedence

Operator precedence is rule for determining the order of execution of heterogeneous operators in an expression.

precedence is handled by grouping operators of same precedence in the same production.

You can see that + and - have the same precedence as does * and /.

The operators with highest precedence occur farther down in the grammar, that is, an expression is a sum of products which is product of exponentiation.

Grouping with Parentheses

Finally, products of sums can be denoted by putting the sum in parentheses as in:

$$666*(42+496)*(x+y+z)$$

To get this affect the parenthesized expression is put in the same place that any variable could be put.

In order to return to the top of the grammar the parentheses act as a counter for the number of times you return to the top of the grammar.

Unary operators generally bind very tightly so they go close to the variables and grouping operators. Here we have added unary minus:

$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{addpart} \rangle \mid \langle \text{exp} \rangle - \langle \text{addpart} \rangle \mid \langle \text{addpart} \rangle$
 $\langle \text{addpart} \rangle ::= \langle \text{addpart} \rangle * \langle \text{mulpart} \rangle \mid \langle \text{addpart} \rangle / \langle \text{mulpart} \rangle \mid \langle \text{mulpart} \rangle$
 $\langle \text{mulpart} \rangle ::= \langle \text{unary} \rangle ^ \langle \text{mulpart} \rangle \mid \langle \text{unary} \rangle$
 $\langle \text{unary} \rangle ::= - \langle \text{unary} \rangle \mid \langle \text{group} \rangle$
 $\langle \text{group} \rangle ::= \langle \text{var} \rangle \mid (\langle \text{exp} \rangle)$
 $\langle \text{var} \rangle ::= X \mid Y \mid Z$

we allow things like $-X$, $---X$, and $-(X)$. If we had done: $\langle \text{unary} \rangle ::= - \langle \text{group} \rangle \mid \langle \text{group} \rangle$ instead we would not be allowed to recursively apply a unary operator.

Write an unambiguous grammar for the Zev language.

In the Zev language a variable is one of the letters Z, E, or V.

The lowest precedence binary operator is + and it is evaluated left to right.

Then there are two binary operators £ and % which have equal precedence but higher precedence than +.

They are also evaluated left to right and so are left associative.

This is also the binary @ operator which is of higher precedence and is right associative.

The * operator is a unary operator and applied on the left.

Either brackets or parens can be used for grouping.

For example: $[Z@[E£E]]$, $[[Z]]$, $*[[Z]\%[E]\%[V]]$, $V£**V£*V$, and E are in the language.

Inescapable Productions

Consider these two simple grammars:

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{pound} \rangle \mid \langle \text{exp} \rangle - \langle \text{pound} \rangle \mid \langle \text{mul} \rangle$$
$$\langle \text{mul} \rangle ::= \langle \text{mul} \rangle * \langle \text{var} \rangle \mid \langle \text{mul} \rangle / \langle \text{var} \rangle \mid \langle \text{var} \rangle$$
$$\langle \text{var} \rangle ::= X \mid Y \mid X$$
$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{pound} \rangle \mid \langle \text{exp} \rangle - \langle \text{pound} \rangle$$
$$\langle \text{mul} \rangle ::= \langle \text{mul} \rangle * \langle \text{var} \rangle \mid \langle \text{mul} \rangle / \langle \text{var} \rangle$$
$$\langle \text{var} \rangle ::= X \mid Y \mid X$$

The second grammar might look reasonable, but wait... given a $\langle \text{exp} \rangle$ how would you ever get rid of $\langle \text{exp} \rangle$ since every right hand side contains an $\langle \text{exp} \rangle$?

This means that you could never generate a sentence that was devoid of the nonterminal $\langle \text{exp} \rangle$! The second grammar is invalid because all of the right hand side options contain the left hand side and so the production is inescapable.

$\langle \text{zev} \rangle ::= \langle \text{zev} \rangle + \langle \text{pound} \rangle \mid \langle \text{pound} \rangle$

// $\langle \text{zev} \rangle$ appears on left of +

$\langle \text{pound} \rangle ::= \langle \text{pound} \rangle \pounds \langle \text{at} \rangle \mid \langle \text{pound} \rangle \% \langle \text{at} \rangle \mid \langle \text{at} \rangle$

// $\langle \text{pound} \rangle$ appears on left of \pounds

$\langle \text{at} \rangle ::= \langle \text{unary} \rangle @ \langle \text{at} \rangle \mid \langle \text{unary} \rangle$

// $\langle \text{at} \rangle$ appears on right of @

$\langle \text{unary} \rangle ::= * \langle \text{unary} \rangle \mid \langle \text{var} \rangle$

//* is applied to $\langle \text{unary} \rangle$ not $\langle \text{var} \rangle$

$\langle \text{var} \rangle ::= Z \mid E \mid V \mid (\langle \text{zev} \rangle) \mid [\langle \text{zev} \rangle]$

//parens in same production with vars