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.

For comparison, a context-sensitive grammar can have production rules where both the left-hand and right-hand sides may be surrounded by a context of terminal and nonterminal symbols.

Parsing

Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols conforming to the rules of a grammar.

The term parsing comes from Latin pars meaning part.

A parse tree represents the syntactic structure of a string according to some grammar.

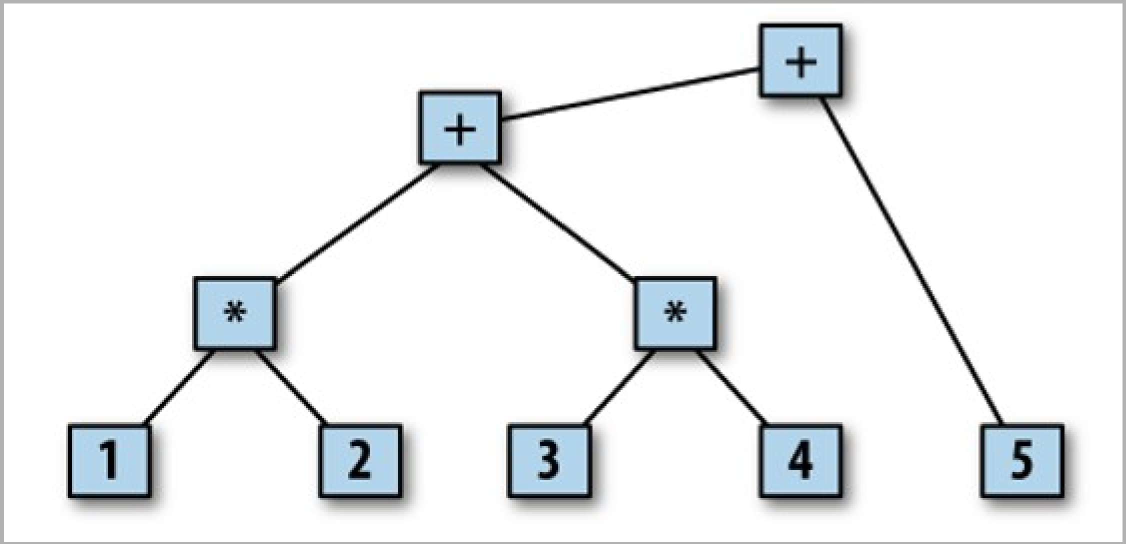
A grammar is a set of production rules for strings in a language.
The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax.

A grammar does not describe the meaning of the strings or what can be done with them only their form.

Grammars and Parsing

The parser's job is to figure out the relationship among the input tokens. A common way to display such relationships is a parse tree.

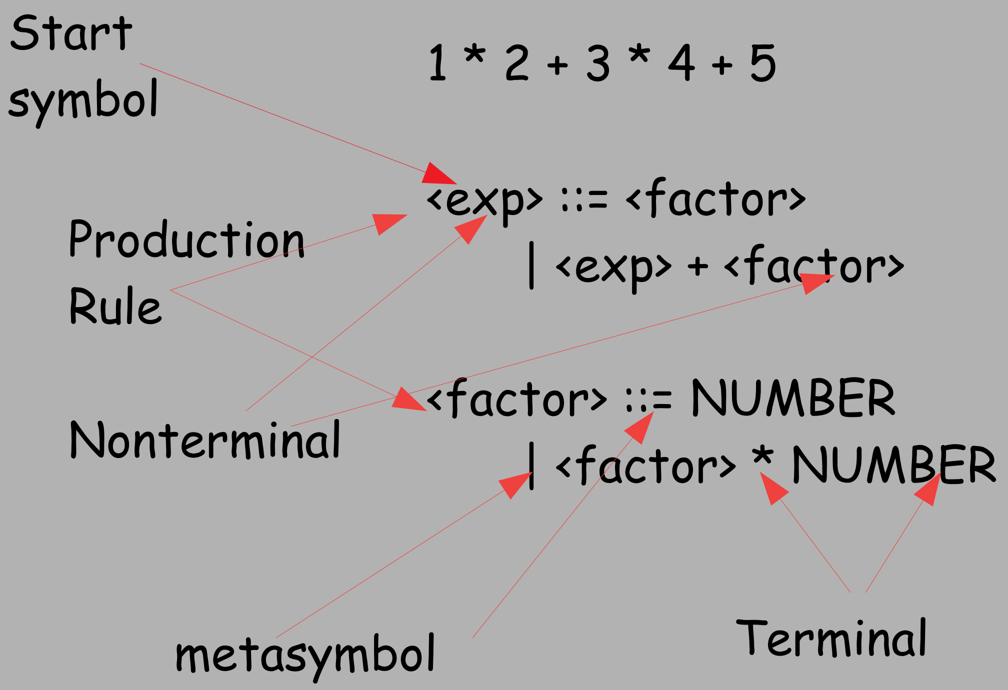
For example, under the usual rules of arithmetic, multiplication has higher precedence than addition, the arithmetic expression 1 * 2 + 3 * 4 + 5 would have the parse tree



Backus-Naur Form

Backus-Naur Form (BNF), created around 1960 to describe Algol 60
and named after two members of the Algol 60 committee

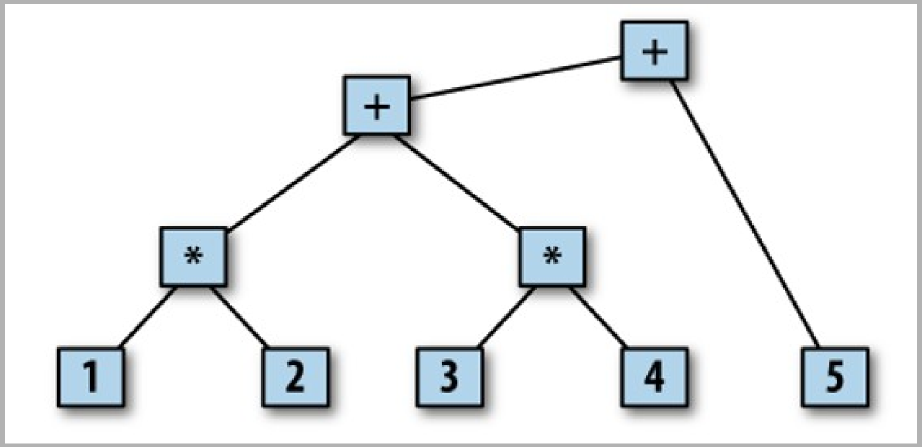
In order to write a parser, we need some way to describe the rules,
the grammar, the parser uses to turn a sequence of tokens into a parse tree.



1 * 2 + 3 * 4 + 5

```
<exp> ::= <factor>
        | <exp> + <factor>

<factor> ::= NUMBER
           | <factor> * NUMBER
```



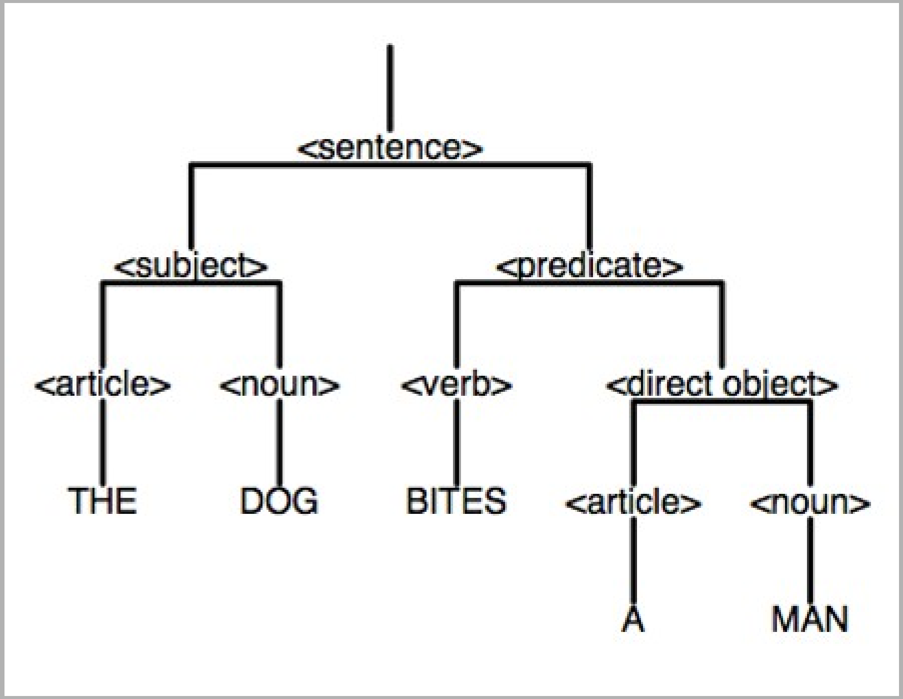
Backus-Naur Form (BNF) is a notation for expressing a CFG.

• Nonterminals are denoted by surrounding symbol with <>. e.g. <turtle>
• Alternation is denoted by | e.g. bad <cats> | good <dogs>.
<animal> ::= bad <cats> | good <dogs>
you could say the same thing without alternation:
<animal> ::= bad <cats>
<animal> ::= good <dogs>
• 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. <dogs> ::= corgi | 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 replace 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”.

<sentence> ::= <subject> <predicate>
<subject> ::= <article> <noun>
<predicate> ::= <verb> <direct-object>
<direct-object> ::= <article> <noun>
<article> ::= THE | A
<noun> ::= MAN | DOG
<verb> ::= BITES | 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

```
<sentence> ::= X | X <sentence>
```

This is also a grammar for that language:

```
<sentence> ::= X | <sentence> 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

```
<sentence> ::= X | Y | X <sentence>| Y <sentence>
```

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

```
<sentence> ::= <sub> | <sub> <sentence>
<sub> ::= X | 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:

<sentence> ::= X | Y | X <sentence>

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

X	ok
Y	ok
XX	ok
XXX	ok
XXXY	ok
XXYY	no

<sentence> ::= X | Y | <sentence> X

is a grammar for
a single X
a single Y
a single X or a single Y followed at the end by 1 or more X's, cant end in Y

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 Xs.

X	ok
Y	ok
XY	no
XXXXXYX	no
XYXXXXXX	ok

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

```
<sentence> ::= <listx> ";"
<listx> ::= X | X <listx>
```

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

```
<listx> ::= X | X "," <listx>
```

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?

```
<listx> ::= X ";" | X ";" <listx>
```

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:

```
<sentence> ::= <sub> | <sentence> <sub>
<sub> ::= X ","
```

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

```
<arglist> ::= "(" ")" | "(" <varlist> ")"
<varlist> ::= <var> | <varlist> "," <var>
<var>::= 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;

```
<tdecl> ::= <type> <varlist> ";"
<varlist> ::= <var> | <varlist> "," <var>
<var>::= X | Y | Z
<type>::= int | bool | string
```

Augment the Type Grammar with the Keyword Static
eg static int X;
or bool x;

```
<tdecl> ::= <type> <varlist> ";"
<varlist> ::= <var> | <varlist> "," <var>
<var>::= X | Y | Z
<type>::= static <basictype> | <basictype>
<basictype>::= int | bool | string
```

(In the C programming language, static is used with global variables and functions to set their scope to the containing file. In local variables, static is used to store the variable in the statically allocated memory instead of the automatically allocated memory)

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

```
<exp> ::= <exp> + <addpart> | <exp> - <addpart> | <addpart>
<addpart> ::= <addpart> * <mulpart> | <addpart> / <mulpart> | <mulpart>
<mulpart> ::= <group> ^ <mulpart> | <group>
<group> ::= <var> | ( <exp> )
<var> ::= X | Y | 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:

```
<exp> ::= <exp> + <addpart> | <exp> - <addpart> | <addpart>
<addpart> ::= <addpart> * <mulpart> | <addpart> / <mulpart> | <mulpart>
<mulpart> ::= <unary> ^ <mulpart> | <unary>
<unary> ::= - <unary> | <group>
<group> ::= <var> | ( <exp> )
<var> ::= X | Y | Z
```

we allow things like –X, –––X, and –(X). If we had done: <unary> ::= - <group> | <group> instead we would not be allowed to recursively apply a unary operator.

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.

```
<zev>::=<zev> + <pound> | <pound>                    //<zev> appears on left of +
```

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.

```
<pound>::=<pound> # <at> | <pound> % <at> | <at>    //<pound> appears on left of #
```

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

```
<at>::=<unary> @ <at> | <unary>                      //<at> appears on right of @
```

Write an unambiguous grammar for the Zev language.

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

```
<unary>::=* <unary> | <var>                    //* is applied to <unary> not <var>
```

Either square brackets [] or parenthesis () can be used for grouping.

```
<var>::=Z | E | V | ( <zev> ) | [ <zev> ]      //parens in same production with vars
```

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:

```
<exp>::= <exp> + <pound> | <exp> - <pound> | <mul>
<mul>::= <mul> * <var>| <mul> / <var>| <var>
<var>::= X | Y | X
```

```
<exp>::= <exp> + <pound> | <exp> - <pound>
<mul>::= <mul> * <var>| <mul> / <var>
<var>::= X | Y | X
```

The second grammar might look reasonable, but wait... given a <exp> how would you ever get rid of <exp> since every right hand side contains an <exp>?!

This means that you could never generate a sentence that was devoid of the nonterminal <exp>! The second grammar is invalid because all of the right hand side options contain the left hand side and so the production is inescapable.

Ambiguity

The processing of sentences in a language to get their meaning usually uses the parse trees.

The parse tree is a way to understand the structure of what was said in the sentence.

If for a given grammar G there exists a sentence in the language for which there is more than one parse tree then G is said to be ambiguous.

You only need one example sentence to make G ambiguous!

Note that the language is not ambiguous, the grammar is.

Also note that it is OK for there to be more than one derivation for a sentence using an unambiguous grammar.

Derivation doesn't make the grammar ambiguous, parse trees do.

<sentence> ::= X | <sentence> X | X <sentence>

This grammar is ambiguous because there exists a sentence that has more than one parse tree.

For example, XX has two parse trees: