Grammars
Warning -- there is a bunch of terminology ahead. It isn't hard and you don't have to memorize it. The terminology is how this stuff is defined mathematically. Try to build up an intuitive sense of what it means.
Grammars are very old. There were grammars constructed for Sanskrit almost 3000 years ago.

The notation and terminology used for programming language grammars was developed by John Backus and Peter Naur for Fortran (so it is often called BNF, for "Backus-Naur Form") and has been for almost all programming languages since then.
Here are 6 definitions:

1. An *alphabet* $V$ is a finite collection of symbols, such as 
   \{0, 1\}, \{"a", "b", "c", ..., "z"\} or \{"if", "then", "else"\}
2. $V^*$ is the collection of all strings made up of 0 or more 
elements of $V$.
3. $\varepsilon$ is the *empty string* -- the string of length 0
4. $V^+ = V^* - \{\varepsilon\}$
5. A *language over $V$* is any subset of $V^*$
6. A *grammar* for a language is a method for specifying 
which strings are in a language. This consists of 
   a. An alphabet $V_T$ of *terminal symbols* 
   b. An alphabet $V_N$ of *nonterminal symbols* 
   c. One or more *start symbols* in $V_N$ 
   d. A set of *production rules* for expanding strings. We 
usually just call these the *grammar rules*. 
The language generated by the grammar is the set of strings in $V_T^*$ that can be generated from the start symbols by following the production rules:

- Begin with a start symbol
- At each step replace one of the nonterminal symbols by the right-hand side of one of its production rules
- Continue in this way until there are no remaining nonterminal symbols
In general, production rules have the form
\[ \alpha ::= \beta \] where \( \alpha \) and \( \beta \) are both strings in \((V_T \cup V_N)^*\).

A *derivation* is a sequence of steps that replaces the left side of a production rule with the right side of this rule. We usually continue derivations until we have derived a string of terminal symbols.
An example should help. Here is a simple grammar for a language of arithmetic expressions:

\( V_T = \{0, 1, 2, 3, \ldots, 9, +, \ast\} \)

\( V_N = \{E, T, N, D\} \)

E is the start symbol.

Rules:

\[
\begin{align*}
E &::= E + T \\
E &::= T \\
T &::= T \ast N \\
T &::= N \\
N &::= DN \\
N &::= D \\
D &::= 0 | 1 | 2 | \ldots | 9 \quad (\quad | \quad \text{means "or"})
\end{align*}
\]
Here is a derivation that shows that $3+42*5$ is a string in the language generated by this grammar:

\[
E ::= E+T \\
    ::= T+T \\
    ::= T+T*N \\
    ::= N+N*N \\
    ::= N+DN*N \\
    ::= D+DD*D \\
    ::= 3+42*5
\]
However, 3++4 is not in the language: The only rule containing + is \( E ::= E+T \), so both + symbols need to come from this rule:
\[
E ::= E+T
\]
\[
::= E+T+T
\]
There is no way for that middle \( T \) to become an empty string, so no string in the language can have two consecutive + symbols.
To save space, we often write all of the rules that have the same left side on one line, separating the right sides with |. The previous grammar would be written

\[
\begin{align*}
E & ::= E+T | T \\
T & ::= T*N | N \\
N & ::= DN | D \\
D & ::= 0 | 1 | ... | 9
\end{align*}
\]
Here is another grammar:

\[ V_T = \{a, b, c\} \]
\[ V_N = \{S, T, U\} \]

The start symbol is S

Rules:

\[ S ::= aSTU \]
\[ S ::= abU \]
\[ bT ::= bb \]
\[ bU ::= bc \]
\[ UT ::= TU \]
\[ cU ::= cc \]
Here is a quick derivation:

\[
S ::= \text{abU} \\
\text{ ::= abc}
\]
Here is another derivation:

\[
S ::= aSTU
::= aabUTU
::= aabTUU
::= aabbUU
::= aabbcU
::= aabbcc
\]

It isn't terribly difficult to show that this grammar generates the language \( \{ a^n b^n c^n : n \geq 1 \} \).
Types of Grammars:

**Regular**: All production rules are either of the form $A ::= a$ or $A ::= aB$, where $A$ and $B$ are nonterminal symbols and $a$ is a terminal symbol.

**Context Free**: All production rules have the form $A ::= \alpha$, where $A$ is a single nonterminal symbol and $\alpha$ might have both terminals and nonterminals.

**Context Sensitive**: All production rules have the form $\alpha ::= \beta$, where $|\alpha| \leq |\beta|$. 

**Arbitrary**
Parsing is the process of taking a string of terminal symbols and producing a derivation for it.

We usually display a derivation as a parse tree, where a rule such as $A ::= bcd$ is displayed as

```
A
\--- b
   \-- c
      \- d
```
Context sensitive languages are very difficult to parse. On the other hand, there are several well-known, efficient algorithms for parsing context-free languages.

Programming languages are not context-free: a statement such as

\[ x = y \]

is either valid or not depending on things (type declarations, earlier assignments to \( y \)) that may be nowhere near the code for this statement.
The standard practice ever since the creation of Fortran has been to give a context-free grammar that defines the syntax of a language, supplemented by prose that describes the semantics of the language. This corresponds to stages of a compiler or interpreter: we first parse a program to determine if it is syntactically correct and to build a parse tree for it, we then do semantic analysis on the parse tree to determine if it is type-correct and to either execute the program (in an interpreter) or generate machine code for it (in a compiler).
Here is a fairly complete grammar for the language of parenthesized arithmetic expressions. This stretches the definition if a grammar with the rule $G ::= \text{number}$ (which makes the alphabet of terminal symbols infinite), but in practice this isn't a problem.

$$
\begin{align*}
E &::= E+T \mid E-T \mid T \\
T &::= T*F \mid T/F \mid T\%F \mid F \\
F &::= F ** G \mid G \\
G &::= (E) \mid \text{number}
\end{align*}
$$