# Warum gibt es CSS-Präprozessoren? Vorteile, Nachteile?

* Erleichtern das Schreiben von Code
* Vereinfachte, bessere Syntax
* Aufgaben werden automatisiert
* Funktionen werden bereitgestellt
* Code ist immer Valides CSS
* Variablen sind möglich
* Es können Mixins verwendet werden
* Code kann verschachtelt werden

Definition CSS-Präprozessor:

Ein Präprozessor ist ein Computerprogramm, das Eingabedaten vorbereitet und

zur weiteren Bearbeitung an ein anderes Programm weitergibt. Der Präprozessor wird

häufig von Compilern oder Interpretern dazu verwendet einen Eingabetext zu konvertieren

und das Ergebnis im eigentlichen Programm weiter zu verarbeiten.

Um Variabilität zu erreichen,

# Was bedeutet Superset?

Ein Superset ist eine sogenannte Obermenge. Superset kommt aus der Mengenlehre und bedeutet, dass B ein Superset von A ist, sobald A in B enthalten ist. Das heißt, A ist ein Subset von B und alle Elemente von A sind ebenso Elemente von B.

Im Fall von SCSS und CSS3 ist SCSS ein Superset von CSS3. Ein valides CSS3 ist dementsprechend auch ein valides SCSS. Umgekehrt gilt das jedoch nicht.

Der praktische Vorteil daran ist, dass man in bestehenden CSS Projekten einfach anfangen kann in den gleichen Dateien SCSS Features zu verwenden. Mit SASS zum Beispiel geht das nicht. Wenn man in einer Datei SASS und CSS mischt und versucht, das durch den SASS Compiler zu schicken, wird das nicht funktionieren (nehm ich jetzt mal an, ausprobiert hab ichs nicht).

Ich würde daher SASS und Stylus nicht als supersets von CSS bezeichnen, weil eben die Menge der gültigen CSS stylesheets nicht in der Menge der gültigen SASS/Stylus stylesheets enthalten ist. Nicht jeder Text, der gültiges CSS ist, ist auch gültiges SASS/Stylus.

# Was ist Lexer? Parser? Kompilierung?

What parsers and lexers have in common:

1. They read *symbols* of some *alphabet* from their input.  
   Hint: The alphabet doesn't necessarily have to be of letters. But it has to be of symbols which are**atomic** for the language understood by parser/lexer.
   * Symbols for the lexer: ASCII characters.
   * Symbols for the parser: the particular tokens, which are terminal symbols of their grammar.
2. They analyse these *symbols* and try to match them with the *grammar* of the language they understood.  
   And here's where the real difference usually lies. See below for more.
   * Grammar understood by lexers: regular grammar (Chomsky's level 3).
   * Grammar understood by parsers: context-free grammar (Chomsky's level 2).
3. They attach *semantics* (meaning) to the language pieces they find.
   * Lexers attach meaning by classifying *lexemes* (strings of symbols from the input) as the particular *tokens*. E.g. All these lexemes: \*, ==, <=, ^ will be classified as "operator" token by the C/C++ lexer.
   * Parsers attach meaning by classifying strings of tokens from the input (sentences) as the particular *nonterminals* and building the *parse tree*. E.g. all these token strings: [number][operator][number], [id][operator][id], [id][operator][number][operator][number] will be classified as "expression" nonterminal by the C/C++ parser.
4. They can attach some additional meaning (data) to the recognized elements. E.g. when a lexer recognizes a character sequence constituting a proper number, it can convert it to its binary value and store with the "number" token. Similarly, when a parser recognize an expression, it can compute its value and store with the "expression" node of the syntax tree.
5. They all produce on their output a proper *sentences* of the language they recognize.
   * Lexers produce *tokens*, which are *sentences* of the *regular language* they recognize. Each token can have an inner syntax (though level 3, not level 2), but that doesn't matter for the output data and for the one which reads them.
   * Parsers produce *syntax trees*, which are representations of *sentences* of the *context-free language* they recognize. Usually it's only one big tree for the whole document/source file, because the whole document/source file is a proper *sentence* for them. But there aren't any reasons why parser couldn't produce a series of syntax trees on its output. E.g. it could be a parser which recognizes SGML tags sticked into plain-text. So it'll *tokenize* the SGML document into a series of tokens: [TXT][TAG][TAG][TXT][TAG][TXT]....

As you can see, parsers and tokenizers have much in common. One parser can be a tokenizer for other parser, which reads its input tokens as symbols from its own alphabet (tokens are simply symbols of some alphabet) in the same way as sentences from one language can be alphabetic symbols of some other, higher-level language. For example, if \* and - are the symbols of the alphabet M (as "Morse code symbols"), then you can build a parser which recognizes strings of these dots and lines as letters encoded in the Morse code. The sentences in the language "Morse Code" could be *tokens* for some other parser, for which these *tokens* are atomic symbols of its language (e.g. "English Words" language). And these "English Words" could be tokens (symbols of the alphabet) for some higher-level parser which understands "English Sentences" language. And **all these languages differ only in the complexity of the grammar**. Nothing more.

So what's all about these "Chomsky's grammar levels"? Well, Noam Chomsky classified grammars into four levels depending on their complexity:

* Level 3: Regular grammars  
  They use regular expressions, that is, they can consist only of the symbols of alphabet (a,b), their concatenations (ab,aba,bbb etd.), or alternatives (e.g. a|b).  
  They can be implemented as finite state automata (FSA), like NFA (Nondeterministic Finite Automaton) or better DFA (Deterministic Finite Automaton).  
  Regular grammars can't handle with *nested syntax*, e.g. properly nested/matched parentheses (()()(()())), nested HTML/BBcode tags, nested blocks etc. It's because state automata to deal with it should have to have infinitely many states to handle infinitely many nesting levels.
* Level 2: Context-free grammars  
  They can have nested, recursive, self-similar branches in their syntax trees, so they can handle with nested structures well.  
  They can be implemented as state automaton with stack. This stack is used to represent the nesting level of the syntax. In practice, they're usually implemented as a top-down, recursive-descent parser which uses machine's procedure call stack to track the nesting level, and use recursively called procedures/functions for every non-terminal symbol in their syntax.  
  But they can't handle with a *context-sensitive* syntax. E.g. when you have an expression x+3 and in one context this x could be a name of a variable, and in other context it could be a name of a function etc.
* Level 1: Context-sensitive grammars
* Level 0: Unrestricted grammars  
  Also called "phase-structure grammars".

# Workflow mit CSS-Präprozessoren erklären