

# Sprachkonzepte

## Teil 2: Sprachen

Syntax, Semantik, Pragmatik

# Sprachen

---

**Sprache** = System von Zeichen, das der Gewinnung und Ausprägung von Gedanken, ihrem Austausch zwischen verschiedenen Menschen sowie der Fixierung von erworbenem Wissen dient (*Meyers Großes Standardlexikon, 1983*)

- in der Informatik geht es um die Fixierung von Gedanken zu Daten und Algorithmen in einer Form, die auf Rechnern nutzbar ist
- natürliche Sprachen gibt es etwa 6500  
([https://de.wikipedia.org/wiki/Sprachfamilien\\_der\\_Welt](https://de.wikipedia.org/wiki/Sprachfamilien_der_Welt))
- Programmiersprachen gibt es auch mindestens einige hundert, je nachdem wie eng man den Begriff der Programmiersprache fasst

# Programmiersprachen

---

Programmiersprachen sind formale Sprachen, die der Implementierung von Datenstrukturen und Algorithmen auf Rechner dienen

*Der Programmierbegriff ist hier bewusst etwas weiter als üblich gefasst.*

*Meist wird nur das Implementieren von Algorithmen als Programmieren aufgefasst.*

*Sprachen wie etwa SQL, XML, HTML, CSS sind dann keine Programmiersprachen.*

Aspekte einer Programmiersprache:

- Syntax

Welche Symbole gibt es?

Vokabular

Wie dürfen die Symbole kombiniert werden?

Grammatik

- Semantik

Welche Symbolfolgen haben Bedeutung?

statische Semantik

Und welche Bedeutung ist das?

dynamische Semantik

- Pragmatik

Für welche Zwecke ist die Sprache geeignet?

Domänen

Wie drückt man sich in der Sprache am besten aus?

Stil

1) *Syntax*

2) *Semantik*

3) *Pragmatik*

# Syntax: Vokabular

---

Übliches Vokabular von Programmiersprachen:

- Kommentare *ohne Einfluss auf die Bedeutung des Programms*
- Zwischenraum (*whitespace*)
- Terminalsymbole (*tokens*)

Schlüsselwörter (*keywords*) *if, else, while, ...*

Trennzeichen (*seperators*) *Klammern, Punkt, Komma, Semikolon, ...*

Operatoren *+, -, \*, /, ...*

Literale *Zahlen, Strings, ...*

Bezeichner (*identifiers*) *Variablennamen, ...*

**Vokabulare werden mit regulären Ausdrücken formal spezifiziert.**

*die regulären Ausdrücke beschreiben die Zusammenfassung von Einzelzeichen zu Elementen des Vokabulars*

# Vokabular: Reguläre Ausdrücke (1)

---

Prinzipieller Aufbau von regulären Ausdrücken:

- **Zeichenklassen** sind elementare reguläre Ausdrücke:

$a$	<i>das Zeichen 'a'</i>
$[abc-e]$	<i>eines der Zeichen 'a', 'b', 'c', 'd' oder 'e'</i>
$[^abc-e]$	<i>keines der Zeichen 'a', 'b', 'c', 'd' oder 'e'</i>
$.$	<i>beliebiges Zeichen (außer Zeilenwechsel)</i>

- **Quantifizierung** eines beliebigen regulären Ausdrucks  $r$ :

$r\{2\}$	<i>zweimal hintereinander</i>
$r\{1,3\}$	<i>ein bis drei Wiederholungen</i>
$r?$	<i>entspricht <math>r\{0,1\}</math></i>
$r^*$	<i>entspricht <math>r\{0,\}</math>, d.h. beliebig oft inklusive keinmal</i>
$r^+$	<i>entspricht <math>r\{1,\}</math>, d.h. mindestens einmal</i>

- **Verknüpfung** beliebiger regulärer Ausdrücke  $r$  und  $s$ :

$rs$	<i>Konkatenation, d.h. erst <math>r</math>, dann <math>s</math></i>
$r s$	<i>Alternative, d.h. <math>r</math> oder <math>s</math></i>
$(r)$	<i>Klammerung zum Überschreiben der Vorrangregeln (Quantifizierung vor Konkatenation vor Alternative)</i>

# Vokabular: Reguläre Ausdrücke (2)

---

Viele Implementierungen von regulären Ausdrücken enthalten Erweiterungen, z.B. `java.util.regex`:

- vordefinierte Zeichenklassen

`\d` steht für `[0-9]`

`\D` steht für `[^0-9]`

...

- unterschiedlich aggressive Einstellung der Quantifizierungen

*greedy*      `.*foo` findet in `fooxxxfoo` einzig den Token `fooxxxfoo`

*.\* verbraucht zunächst die gesamte Eingabe, in einem anschließenden Backtracking wird dann vom Ende her geprüft, ob unter den verbrauchten Zeichen `foo` war*

*possessive*    `.*+foo` findet in `fooxxxfoo` keinen Token

*wie greedy, nur ohne Backtracking*

*reluctant*     `.*?foo` findet in `fooxxxfoo` zwei Tokens `foo` und `xxxfoo`

*.\*? verbraucht zunächst kein Zeichen der Eingabe, der Verbrauch wird dann gegebenenfalls zeichenweise erhöht, um ein `foo` zu matchen*

# Reguläre Ausdrücke: Beispiel (1)

---

Lexikalische Analyse arithmetischer Ausdrücke mit dem Compilerbau-Werkzeug ANTLR4 von Terence Parr:

```
// ExprLexer.g4
```

```
lexer grammar ExprLexer;
```

```
Number: Digits ( '.' Digits ) ? ;
```

*regulärer Ausdruck für den Token Number  
(Tokennamen beginnen mit Großbuchstaben)*

```
fragment Digits: ([ 0-9 ])+ ;
```

*Digits ist kein Token,  
sondern benennt nur einen Hilfsausdruck*

```
PLUS: '+' ;
```

```
MINUS: '-' ;
```

```
MUL: '*' ;
```

```
DIV: '/' ;
```

```
LPAREN: '(' ;
```

```
RPAREN: ')' ;
```

*Zwischenraum (whitespace) soll nicht in die  
Ergebnis-Tokenfolge aufgenommen werden*

```
WS: [ \t\r\n ]+ -> channel(HIDDEN) ;
```



## Reguläre Ausdrücke: Beispiel (2)

---

Aus der Datei `ExprLexer.g4` generiert ANTLR4 eine Klasse `ExprLexer.java`. Die Klasse enthält einen Automaten, der die Tokens gemäß den regulären Ausdrücken erkennt.

*Die erstellte Tokenfolge kann anschließend mit einer in EBNF geschriebenen Grammatik auf eine syntaktische korrekte Reihenfolge geprüft werden.*

- Beispiel:

Aus einer eingelesenen Zeichenfolge

`1.2 + 34.5`

wird in der lexikalischen Analyse eine Tokenfolge

`Number("1.2") PLUS("+") Number("34.5")`

bzw. mit dem verborgenen Whitespace

`Number("1.2") WS(" ") PLUS("+") WS(" ") Number("34.5")`

# Syntax: Grammatik

---

Grammatikregeln (Produktionen) legen die erlaubten Tokenfolgen fest.

- bei den üblicherweise verwendeten kontextfreien Grammatiken haben **Produktionen** die Form

*Nichtterminalsymbol = Folge von Terminal- und Nichtterminalsymbolen*

- eines der Nichtterminalsymbole wird als **Startsymbol** ausgezeichnet  
*aus dem Startsymbol lassen sich durch wiederholte Anwendung von Produktionen alle syntaktisch gültigen Tokenfolgen (= Folgen von Terminalen) ableiten*

**Grammatiken werden mit EBNF (*extended Backus-Naur form*) formal spezifiziert.**

# Grammatik: EBNF

---

Aufbau von EBNF (Erweiterte Backus-Naur-Form):

- **Terminale** sind nicht weiter ersetzbar elementare Symbole (= Tokens)  
"abc"    *Zeichenfolge abc*
- **Nichtterminale** sind per Produktionsregel ersetzbare elementare Symbole  
name    *frei wählbarer Bezeichner*
- **Symbolfolgen** aus Symbolfolgen s und t mit Terminalen und Nichtterminalen
  - st    *s gefolgt von t*
  - s | t    *entweder s oder t*
  - {s}    *beliebig viele s inklusive einmal*
  - [s]    *kein oder ein s*
  - (s)    *Klammerung zum Überschreiben der Vorrangregeln*

} Erweiterung gegenüber BNF
- **Produktionsregeln:**  
Nichtterminal = Symbolfolge ;

# EBNF: Beispiel (1)

---

Syntaxanalyse arithmetischer Ausdrücke mit ANTLR4:

```
// ExprParser.g4
parser grammar ExprParser;
options { tokenVocab=ExprLexer; }

expr : multExpr
    | expr (PLUS | MINUS) multExpr
    ;

multExpr : primary
    | multExpr (MUL | DIV) primary
    ;

primary : LPAREN expr RPAREN
    | value
    ;

value : (PLUS | MINUS)? Number
    ;
```

## EBNF: Beispiel (2)

---

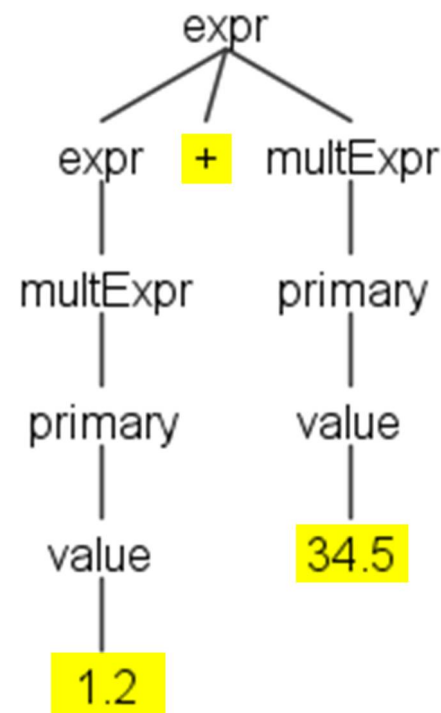
Aus `ExprParser.g4` generiert ANTLR4 eine Klasse `ExprParser.java`. Die Klasse enthält Methoden, die eine eingelesene Tokenfolge auf syntaktisch korrekte Reihenfolge prüfen.

*ANTLR4 erstellt aus der Tokenfolge einen Ableitungsbaum (parse tree).*

- Beispiel:

Ableitungsbaum zum Ausdruck

**1.2 + 34.5**



# Abstrakte Syntax (1)

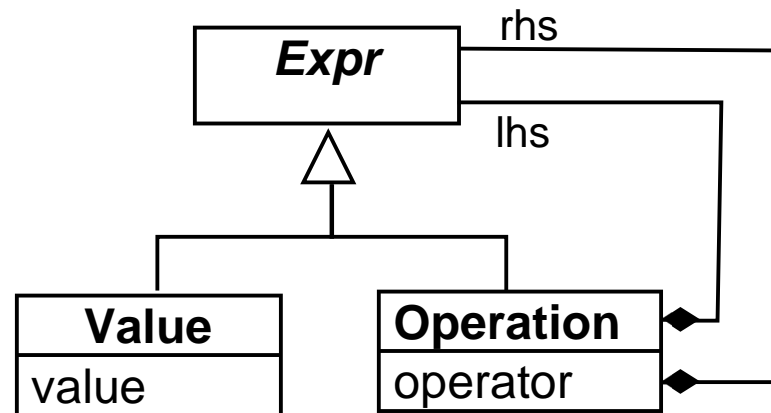
---

Die abstrakte Syntax einer Sprache beschreibt, *was* man mit der Sprache ausdrücken kann, und abstrahiert davon, *wie* man es ausdrückt.

- Beschreibung mit einem **Modell**, das die Elemente der Sprache mit ihren Beziehungen zeigt

- Beispiel:

Modell für arithmetische Ausdrücke mit Elementen **Value** und **Operation**



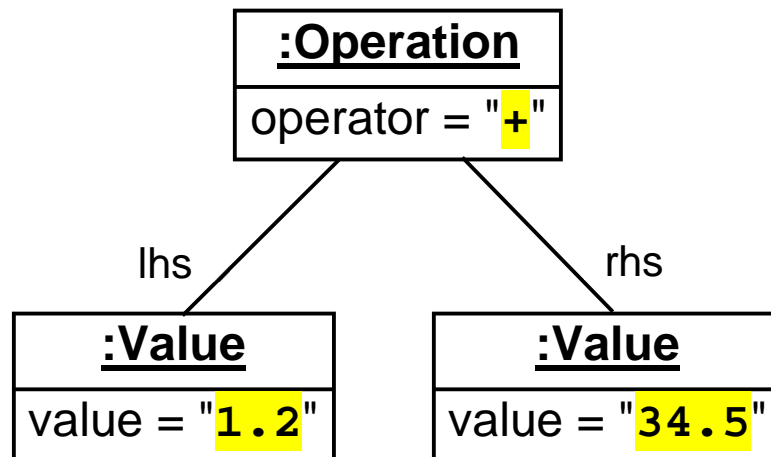
## Abstrakte Syntax (2)

---

Ein **AST** (*Abstract Syntax Tree*) zeigt eine Formulierung als Instanz des Modells der Sprache.

- Beispiel:

Abstrakter Syntaxbaum (**AST**) des Ausdrucks **1.2 + 34.5**



*der AST wird aus dem Ableitungsbaum (parse tree) der konkreten Syntax abgeleitet*

1) *Syntax*

2) ***Semantik***

3) *Pragmatik*



# Semantik (1)

---

Nicht jede syntaktisch korrekte Symbolfolge ist auch semantisch korrekt

- Beispiel für semantischen Unsinn in syntaktisch korrektem Deutsch:

Dunkel war's, der Mond schien helle,  
schneebedeckt die grüne Flur,  
als ein Wagen blitzesschnelle,  
langsam um die Ecke fuhr.

Drinne saßen stehend Leute,  
schweigend ins Gespräch vertieft,  
als ein toteschoss'ner Hase  
auf der Sandbank Schlittschuh lief.

...

Quelle: [https://de.wikipedia.org/wiki/Dunkel\\_war's,\\_der\\_Mond\\_schien\\_helle](https://de.wikipedia.org/wiki/Dunkel_war's,_der_Mond_schien_helle)

## Semantik (2)

---

- Beispiel für semantischen Unsinn in syntaktisch korrektem Java:

```
public abstract final class Beispiel {  
    public private static void main(String[] args) {  
        System.out.print(12345678901234567890);  
        System.out.print(args[x]);  
    }  
}
```

*Das obige Programm enthält vier semantische Fehler,  
genau genommen sogar fünf.*

# Statische Semantik (1)

---

Die statische Semantik einer Programmiersprache regelt die Wohlgeformtheit von Formulierungen in Form von Konsistenzregeln für den AST.

- einfache Konsistenzregeln lassen sich unter Umständen mit einer strikten konkreten Syntax erzwingen  
*allerdings kann eine strikte Grammatik sehr umfangreich werden und eventuell unflexibel hinsichtlich späterer Spracherweiterungen sein*
- Konsistenzregeln für komplexe Beziehungen zwischen Sprachelementen können in der Regel erst nach der Syntaxanalyse auf dem AST geprüft werden  
*klassische Beispiele:*
  - > Regeln zur Typsicherheit von Ausdrücken
  - > Wertebereiche von Zahlliteralen
  - > Regeln zur Eindeutigkeit von Namen

## Statische Semantik (2)

---

- Beispiele für semantische Fehler in Java:

```
public abstract final class Beispiel {  
    public private static void main(String[] args) {  
        System.out.print(12345678901234567890);  
        System.out.print(args[x]);  
    }  
}
```

*eine Klasse kann nicht zugleich abstract und final sein \**

*eine Methode kann nicht zugleich public und private sein \**

*eine Variable muss vor ihrer Benutzung definiert werden\*\**

*ein ganzzahliges Literal ohne L muss im Zahlbereich von int liegen*

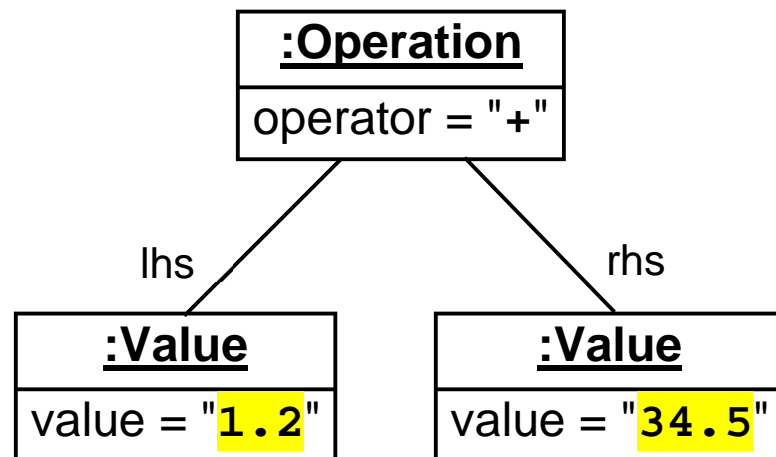
- \* die Kombinationen **abstract final** und **public private** könnte man alternativ auch mit einer strikten konkreten Syntax verhindern
- \*\* der Wert von **x** muss außerdem im gültigen Indexbereich von **args** liegen, was aber nicht mehr eine Frage der statischen Semantik ist, sondern der dynamischen Semantik

## Statische Semantik (3)

---

- mögliche statische Semantik für den AST des arithmetischen Ausdrucks:

die Zahlenwerte der Operanden müssen im Zahlbereich des 32-Bit-Gleitkommaformats von IEEE 754 liegen, d.h. zwischen ca.  $-3,4 \cdot 10^{38}$  und  $+3,4 \cdot 10^{38}$



# Dynamische Semantik (1)

Die **dynamische Semantik** einer Programmiersprache ergibt sich aus der Weiterverarbeitung des AST.

- dynamische Semantik in Form eines Interpreters, der den AST ausführt

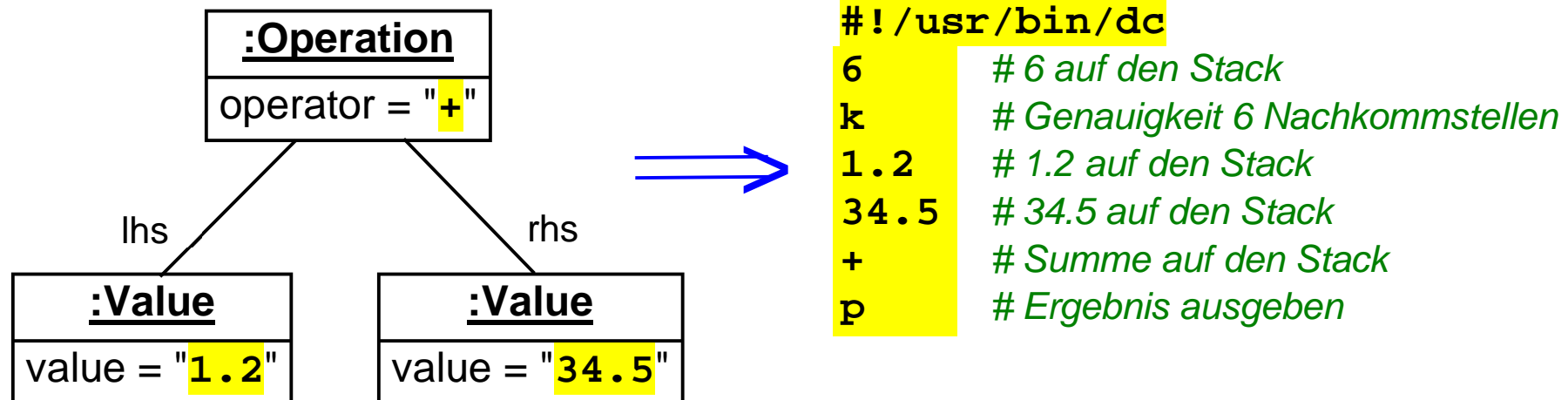
*Beispiel arithmetischer Ausdruck:*

*den AST in Tiefensuche ablaufen und dabei den Wert des Ausdrucks berechnen*

- dynamische Semantik in Form eines Compilers, der den AST auf eine Zielsprache abbildet

*Beispiel arithmetischer Ausdruck:*

*den AST auf Befehle für den Linux Desktop-Calculator `dc` abbilden*



# Dynamische Semantik (2)

---

Beispiele für Aspekte der dynamischen Semantik von Programmiersprachen:

- Skriptsprachen: Typsicherheit von Ausdrücken  
*der Interpreter enthält Code, der die Typsicherheit zur Laufzeit gewährleistet*  
*Typsicherheit als Teil der statischen Semantik braucht Variablendeklarationen, die es in Skriptsprachen üblicherweise nicht gibt*
- Java: Indexprüfung bei Array-Zugriffen  
*der Compiler erzeugt bei der Abbildung auf den Bytecode Befehle für die Indexprüfung*  
*Array-Längen und Index-Werte lassen sich im Allgemeinen nicht statisch bestimmen*
- Java: Typecast von Referenzen  
*die Typsicherheit ist in Java weitgehend Teil der statischen Semantik, aber Downcasts und Crosscasts von Referenzen lassen sich nicht statisch prüfen*
- Java: Methodenaufrufe mit dynamischer Bindung  
*die Klasse des this-Objekts und damit die gemeinte Methodenimplementierung sind im Allgemeinen erst zur Laufzeit bekannt*

1) *Syntax*

2) *Semantik*

3) *Pragmatik*



# Pragmatik: Domänen (1)

---

Eine Domäne ist in der Softwartetechnik ein abgrenzbares Problemfeld oder ein bestimmter Einsatzbereich für eine Software (*siehe Wikipedia "Problemdomäne"*)

Einteilung von Programmiersprachen nach ihrem Domänenbezug:

- GPL (*General-purpose Language*)

GPLs haben eine universelle Syntax und Semantik, die sie für viele verschiedene Domänen einsetzbar macht

der Domänenbezug entsteht durch domänenspezifische Bibliotheken und Frameworks, die in der Sprache implementiert sind

*alle höheren Programmiersprachen werden hier üblicherweise eingeordnet,  
z.B. Java, C, C++, Python, Scala, ...*

- DSL (*Domain-specific Language*)

DSLs sind in Syntax und Semantik auf eine spezifische Domäne abgestimmt

*z.B. SQL für die Domäne Datenbanken*

*z.B. HTML und CSS für die Domäne Webseiten*

## Pragmatik: Domänen (2)

---

Der Übergang zwischen GPLs und DSLs ist fließend:

- auch eine GPL kann für bestimmte Domänen besser und für andere schlechter oder gar nicht geeignet sein

*C und C++ eignen sich z.B. besonders gut für die Domäne Systemprogrammierung, unter anderem, weil sie den Typ Adresse unterstützen*

*in manchen Domänen sind funktionale Spracheigenschaften besonders nützlich, in anderen objektorientierte, usw.*

*für die Auswahl einer bestimmten GPL sind oft die verfügbaren Bibliotheken und Frameworks wichtiger, als die bei vielen GPLs ähnlichen Spracheigenschaften*

- manche APIs von GPL-Bibliotheken lassen sich fast wie eine DSL verwenden  
die Namen der Bibliotheksfunktionen bilden das Vokabular und die möglichen Aufrufreihenfolgen und Aufrufverschachtelungen der Funktionen die Syntax

*man nennt solche APIs Fluent Interfaces oder interne DSLs*

- DSLs können in eine GPL eingebettet sein

*in Form von String-Literalen und speziellen Kommentaren*

# Pragmatik: Stil (1)

---

Bei **Stil** geht es um diejenigen Merkmale eines Textes, die nicht die Bedeutung betreffen, sondern nur die Art und Weise, wie diese Bedeutung sprachlich formuliert ist. (*sinngemäß aus Wikipedia "Stil", Abschnitt "Sprache"*)

*beim Programmieren lässt sich auch mit ein und derselben Sprache die gleiche Bedeutung meist auf unterschiedliche Art und Weise erzielen, also mit unterschiedlichem Stil*

Stilaspekte bei Programmiersprachen:

- Programmorganisation  
*Aufteilung in Dateien und Ordner, Gliederung, ...*
- Layout  
*Einrückung, Whitespace, Zeilenlänge, ...*
- Namenskonventionen  
*Zeichenvorrat, Groß- / Kleinschreibung, Aufbau, Länge, ...*
- Idiome (*Redewendungen*)  
*übliche Formulierungen für wiederkehrende Situationen, bevorzugte und geächtete Sprachelemente, ...*

## Pragmatik: Stil (2)

---

Für GPLs gibt es oft viele, mitunter auch konkurrierende **Stilempfehlungen**.

*Anspruch der Empfehlungen ist immer, dass sie die Codequalität verbessern, etwa hinsichtlich Lesbarkeit, Wartbarkeit und Fehlerrisiken. Ob sie diesen Anspruch einlösen, ist teilweise umstritten*

Beispiele für Stilempfehlungen zur Sprache C:

- MISRA C (*Motor Industry Software Reliability Association*)
- SEI CERT C Coding Standard
- Linux kernel coding style
- GNU coding standards