

# **Einführung in Web-bezogene Sprachen**

**Dr. Michael Thies**

**basiert auf Material von Prof. Dr. Uwe Kastens**

**WS 2006 / 2007**

# Ziele

Die Vorlesung soll Studierende dazu befähigen

- S** • **Sprachen, die zur Entwicklung von Web-Präsenzen** eingesetzt werden, zu **verstehen**, anzuwenden und zu beurteilen,
- W** • **einfache Web-Präsenzen** mit den dafür heute gebräuchlichen Sprachen und Methoden zu **entwickeln**,
- E** • **Sprachen**, die in Zukunft für solche Aufgaben eingesetzt werden, dann **selbständig zu erlernen**,
- E** • grundlegende, allgemeine **Programmiertechniken** anzuwenden.

# Inhalt (S): Web-bezogene Sprachen

- S1. Einführung zu Web-bezogenen Sprachen**  
Übersicht, Klassifikation
- S2. HTML**  
Einführung, Zweck  
Notation, Struktur, Darstellung  
Tags, Attribute, Zeichen, Verweise  
Tabellen, Formulare Frames  
Einbettung von Fremdsprachen
- S3. PHP**  
Einführung, Zweck  
Notation, Struktur, Einbettung  
Variable, Ausdrücke, Typen, Ablaufstrukturen  
Ein-, Ausgabe: Dateien, interaktive E/A  
Funktionen, Aufrufe, Parameter  
Schleifen und Arrays, Strings und Muster
- S4. JavaScript**  
Übersicht: Notation, Struktur, Einbettung  
Variable, Ablaufstrukturen, Ereignisbehandlung
- S5. XML**  
Übersicht, Definition von Struktur, Transformation

# Inhalt (E): Eigenschaften von Sprachen

- E1. Einführung zu Eigenschaften von Sprachen**  
Übersicht, Klassifikation  
4 Ebenen von Spracheigenschaften
- E2. Symbole und Syntax**  
reguläre Ausdrücke  
kontextfreie Grammatiken
- E3. Statische und dynamische Semantik**  
Gültigkeitsbereiche, Lebensdauer, Rekursion,  
Typisierung, Aufrufe, Ablaufstrukturen

# Inhalt (**W**): Entwicklung von Web-Seiten

- W1. Einführung zum World Wide Web**  
Übersicht, Strukturen, Dienste, Werkzeuge
- W2. Statische HTML-Seiten entwickeln**
- W3. Dateien, Arrays, Funktionen benutzen**
- W4. HTML-Seiten mit PHP generieren**
- W5. Dynamische, interaktive Web-Seiten**
- W6. Projekt im Zusammenhang**
- W7. Datenspeicherung auf dem Client**

Vorlesungsnr. ca.

**Inhalt (EWS)**

1		Einführung zur Vorlesung
	<b>W1.</b>	Einführung zum World Wide Web
2	<b>S1.</b>	Einführung zu Web-bezogenen Sprachen
3	<b>E1.</b>	Einführung zu Eigenschaften von Sprachen
6	<b>S2.</b>	HTML
	<b>W2.</b>	Statische HTML-Seiten entwickeln (Übungen)
8	<b>E2.</b>	Symbole und Syntax
13	<b>S3.</b>	PHP
	<b>W3.</b>	Dateien benutzen
	<b>W4.</b>	HTML-Seiten mit PHP generieren
	<b>W5.</b>	Dynamische, interaktive Web-Seiten
19	<b>E3.</b>	Statische und dynamische Semantik
22	<b>S4.</b>	JavaScript
24	<b>W6.</b>	Projekt im Zusammenhang
26	<b>S5.</b>	XML
28	<b>W7.</b>	Datenspeicherung auf dem Client
29	---	Zusammenfassung

# Voraussetzungen für diese Vorlesung

Überblick über Begriffe und **Kalküle der Informatik**

z. B. aus der Vorlesung „Einführung in die Informatik für Medienwissenschaftler“:

- elementare Begriffe von Hardware und Software
- reguläre Ausdrücke, kontextfreie Grammatiken
- algorithmische Grundelemente

## Technische Voraussetzungen:

- Computer benutzen können,  
z. B. unter Windows; Linux wäre nützlich
- Texte in einfachem Editor (z. B. Notepad, Emacs) erstellen können
- Web-Browser bedienen können (z. B. Firefox, Mozilla, Internet Explorer, Opera)

# Verwendung des in EWS Gelernten

- alle EWS-Inhalte:  
Voraussetzung für die Nachfolgeveranstaltung
- Fähigkeiten Sprachen zu benutzen und zu erlernen:  
in fast allen Lehrveranstaltungen der Informatik und in einschlägigen Berufen
- Web-Seiten entwickeln:  
in einschlägigen Berufen, im Studium, im Alltag
- elementare Programmierkenntnisse:  
in vielen Lehrveranstaltungen der Informatik und in einschlägigen Berufen

# Literatur zur Vorlesung EWS

## Zur Vorlesung insgesamt:

1. elektronisches Skript: <http://ag-kastens.upb.de/lehre/material/ews2006>

## Zu Web-bezogenen Sprachen:

2. S. Münz, W. Nefzger: HTML & Web-Publishing Handbuch (Band 1), Franzis Verlag, 2002  
im WWW: <http://de.selfhtml.org>
3. Wolfgang Dehnhardt: Skriptsprachen für dynamische Webauftritte, Hanser Verlag, 2001
4. Rasmus Lerdorf: PHP kurz und gut, O'Reilly Verlag, 2000
5. David Flanagan: JavaScript kurz und gut, O'Reilly Verlag, 1998
6. Jennifer Niederst: HTML kurz und gut, O'Reilly Verlag, 2002
7. R. Eckstein, M. Casabianca: XML kurz und gut, O'Reilly Verlag, 2002

## Zu Sprachen allgemein:

8. elektronisches Skript: <http://ag-kastens.upb.de/lehre/material/gps>
9. D. A. Watt: Programmiersprachen - Konzepte und Paradigmen, Hanser, 1996 (vergr.)  
engl: Programming Language - Concepts and Paradigms, Prentice Hall, 1990

## Zur Entwicklung von Web-Seiten:

10. Mark Lubkowitz: Webseiten programmieren und Gestalten, Galileo Press GmbH, 2003
11. Peter Kentie: Web Graphics, Tools und Techniken für die Web-Gestaltung, Addison Wesley, 2000

# Das EWS-Skript im WWW

Vorlesung Einführung in Web-bezogene Sprachen WS 2005 - Mozilla

file:///comp/lectures/ews/www/index.html

UNIVERSITÄT PADERBORN  
Die Universität der Informationsgesellschaft

Fachgruppe Kastens > Lehre > Einführung in Web-bezogene Sprachen WS 2005

**Vorlesung Einführung in Web-bezogene Sprachen WS 2005**

<p><b>Vorlesungsfolien</b></p> <ul style="list-style-type: none"> <li>♦ Kapitelübersicht</li> <li>♦ Folienverzeichnis</li> <li>♦ Drucken</li> </ul>	<p><b>Übungsaufgaben</b></p> <ul style="list-style-type: none"> <li>♦ Aufgabenblätter</li> <li>♦ Drucken</li> </ul>
<p><b>Organisation</b></p> <ul style="list-style-type: none"> <li>♦ Allgemeines</li> <li>♦ Aktuelle Hinweise</li> </ul> <p>17.10.2005      Beginn der Veranstaltung</p>	<p><b>Wissenswertes</b></p> <ul style="list-style-type: none"> <li>♦ Ziele</li> <li>♦ Literatur</li> <li>♦ Glossar</li> <li>♦ Links</li> <li>♦ Grundlagen der Programmiersprachen</li> </ul>

Generiert mit Camelot | Probleme mit Camelot? | Geändert am: 19.10.2005

<http://ag-kastens.upb.de/lehre/material/ews2006>

# Erläuterte Folien im Skript



**UNIVERSITÄT PADERBORN**  
Die Universität der Informationsgesellschaft

Fachgruppe Kastens > Lehre > Einführung in Web-bezogene Sprachen WS 2005 > Folienverzeichnis > Folie 102

[Hauptseite](#)

[Kapitelübersicht](#)

[Folienverzeichnis](#)

[Vorherige Folie](#)

[Nächste Folie](#)

[Folienpaket drucken](#)

SUCHEN:

## Einführung in Web-bezogene Sprachen WS 2005 - Folie 102

### Ziele

Die Vorlesung soll Studierende dazu befähigen

- S** • **Sprachen, die zur Entwicklung von Web-Präsenzen** eingesetzt werden, zu **verstehen**, anzuwenden und zu beurteilen,
- W** • **einfache Web-Präsenzen** mit den dafür heute gebräuchlichen Sprachen und Methoden zu **entwickeln**,
- E** • **Sprachen**, die in Zukunft für solche Aufgaben eingesetzt werden, dann **selbständig zu erlernen**,
- E** • grundlegende, allgemeine **Programmiertechniken** anzuwenden.

**Ziele:**  
Ausbildungsziele  
kennenlernen

**In der Vorlesung:**  
Die Ziele werden erklärt  
und begründet.  
Die drei Themenstränge  
werden eingeführt.

**Verständnisfragen:**  
Sind dies auch Ihre Ziele?  
Haben Sie weitere Ziele?

© 2005 bei Prof. Dr. Uwe Kastens

# Organisation im WS 2006/2007

<b>Termine</b>	Vorlesung	Di 11:15 - 12:45	D1, Michael Thies
		Mi 09:15 - 10:45	D1, Michael Thies
		Beginn: Di 17. 10. 2006	
	Übungen	Mo 11:05 - 12:35 (E2.315) und Mi 11:05 - 12:35 (E2.310) <b>Beginn: Mi 18. 10. 2006</b>	
<b>Übungsbetreuer</b>	Dr. Dinh Khoi Le		
<b>Übungsanmeldung</b>	siehe Vorlesungsmaterial im Web		
<b>Hausaufgaben</b>	werden im Vorlesungsmaterial publiziert		
<b>Klausur</b>	nach Ende des Semesters; Termine werden bekanntgegeben		

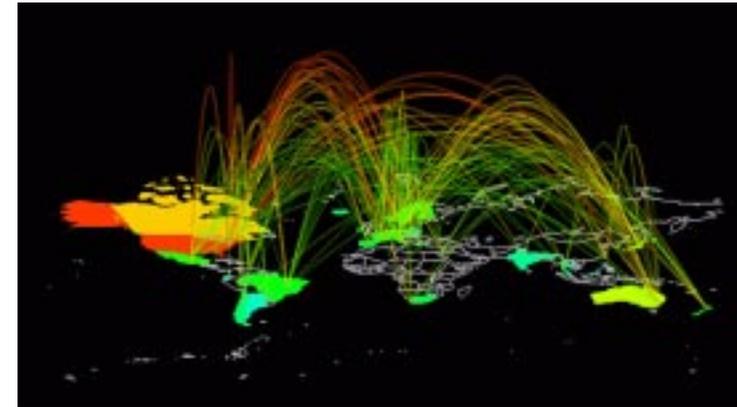
# W1. Einführung zum World Wide Web

## Internet (Interconnected Networks):

Zusammenschluss von vielen regionalen Netzen aus Millionen verbundener Rechner zu einem weltweiten Netz.

## World Wide Web (WWW, Web):

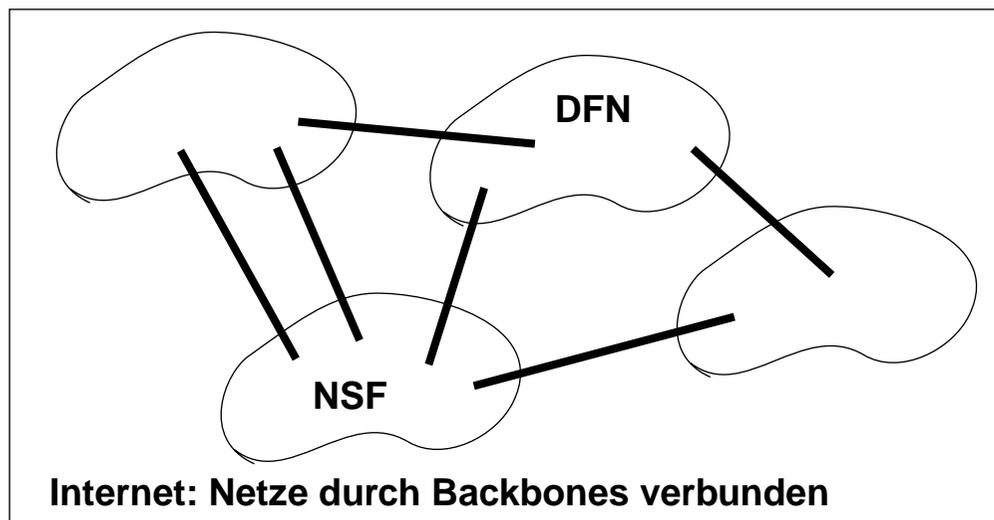
1991 prägte Tim Berners-Lee diesen Begriff für einen sog. Dienst im Internet: Einfacher Zugriff auf weltweit verknüpfte Informationen



Visualisierung des Datenverkehrs im Internet

[http://mappa.mundi.net/maps/maps\\_008](http://mappa.mundi.net/maps/maps_008)

Stephen G. Eick, Bell Laboratories-Lucent Technologies



## Backbone:

Hauptverbindungsleitung, die Netze miteinander verbindet.

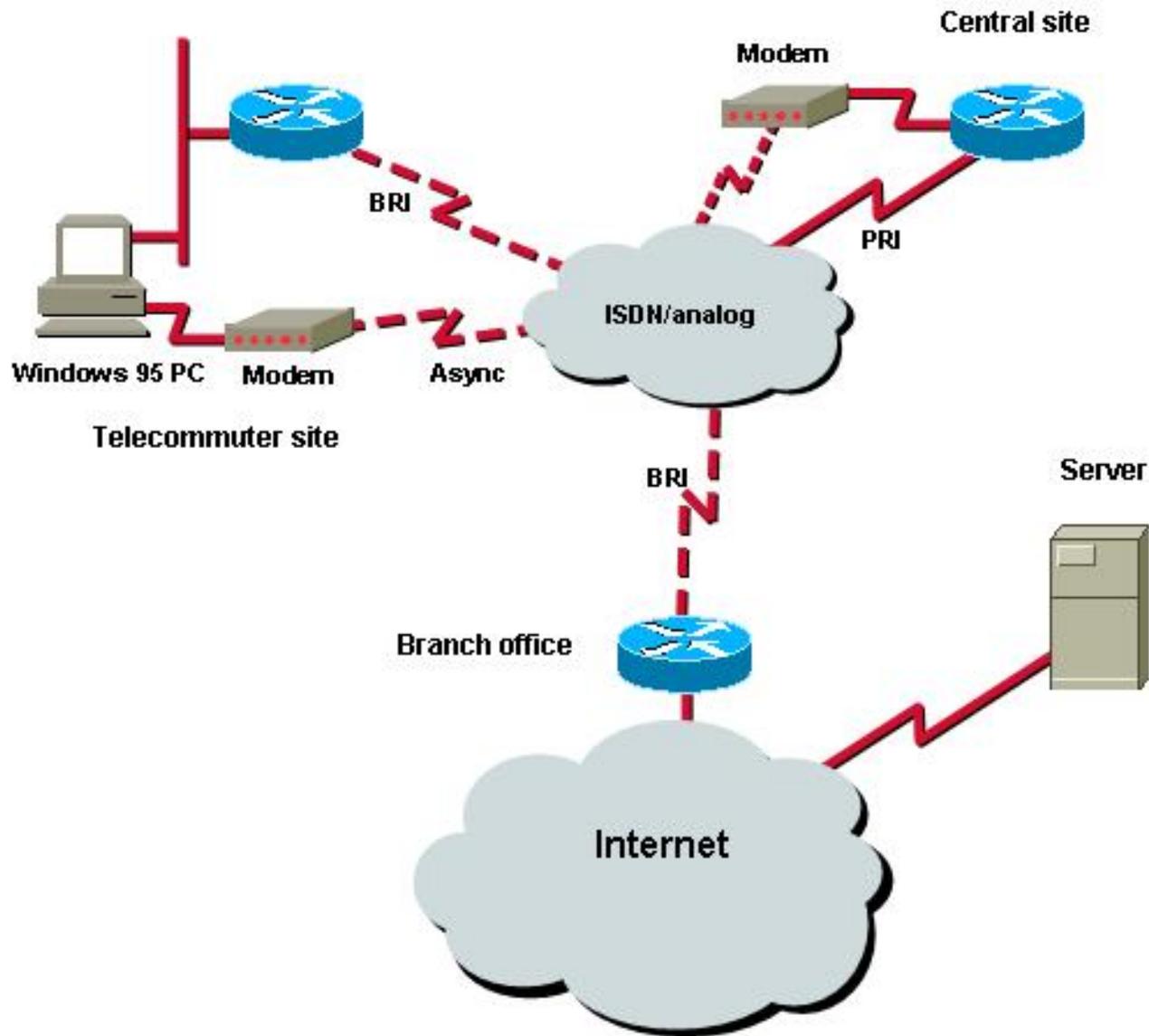
## Gateway:

Rechner, der ein Netz mit anderen Netzen verbindet

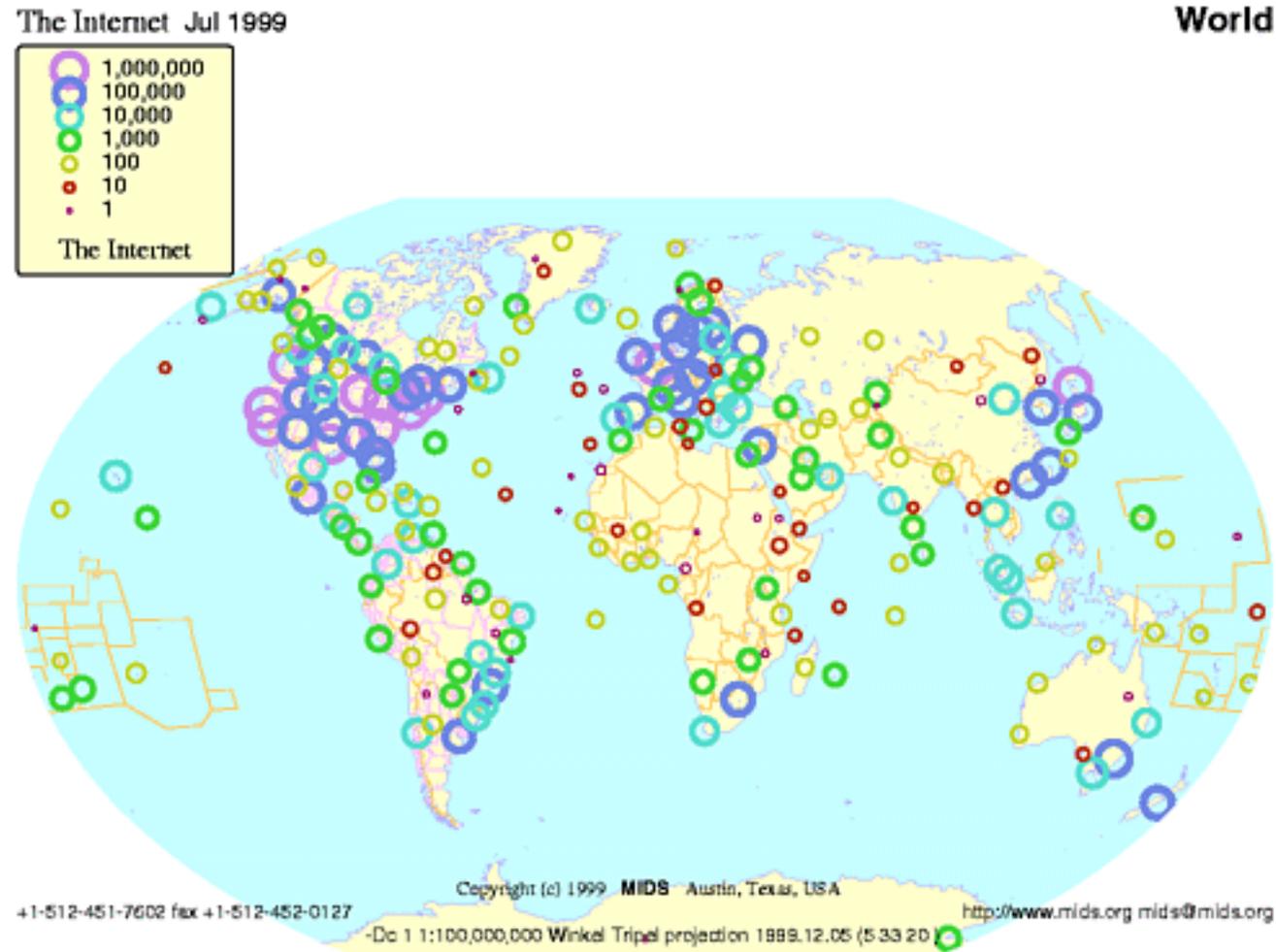
## Host:

Rechner, der ans Internet angeschlossen ist

# Vom PC ins Internet

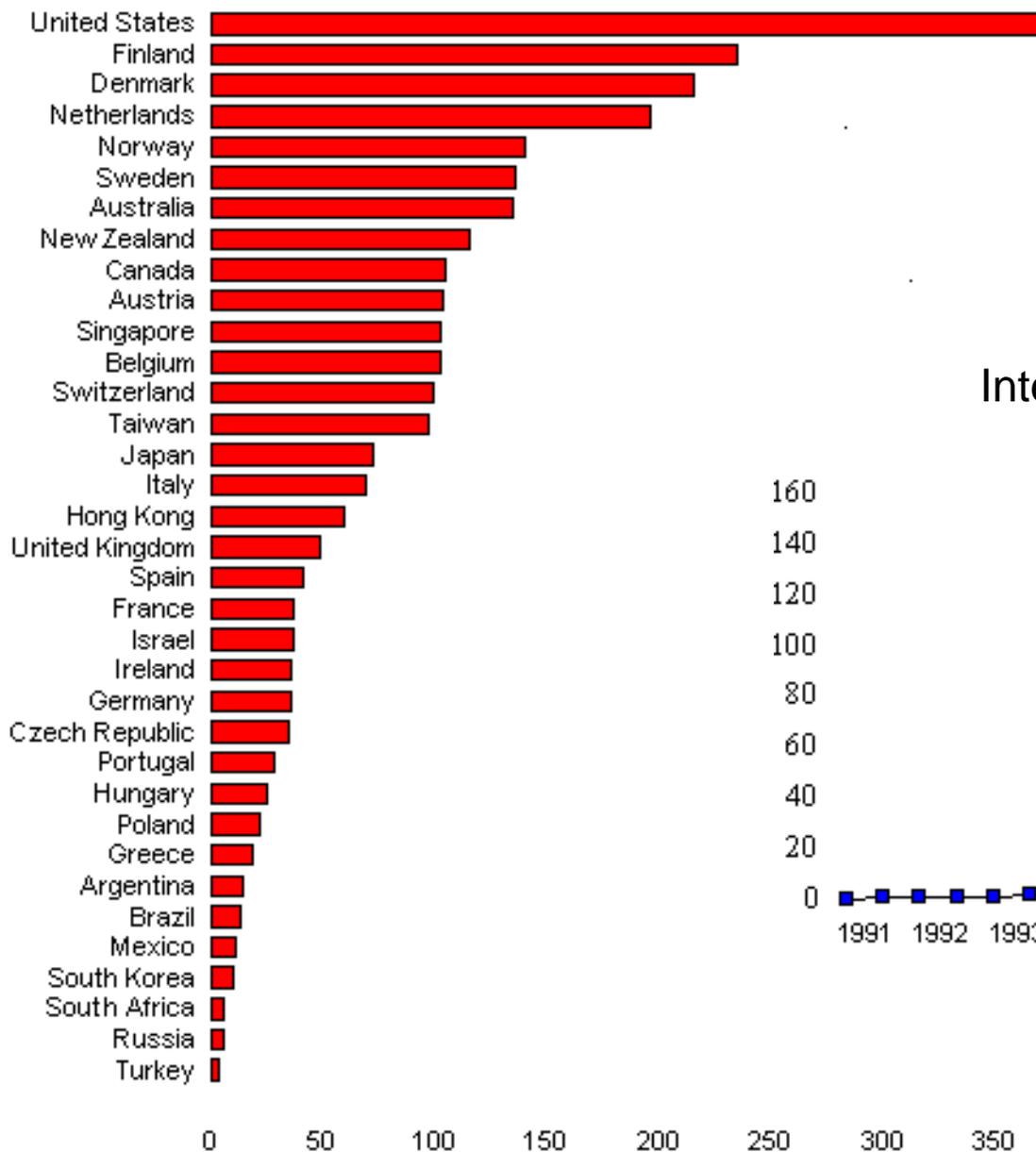


# Anzahl von Hosts im Internet



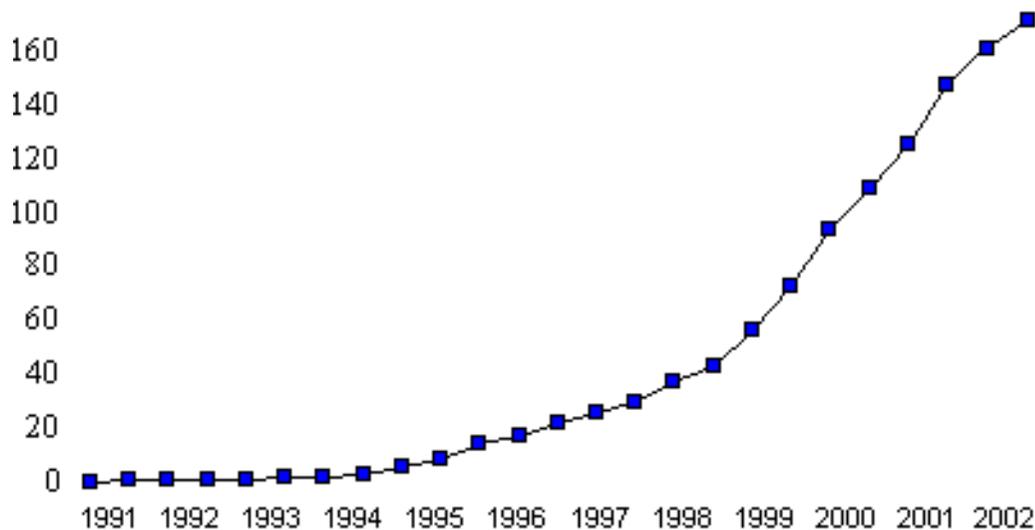
(c)Matrix Internet and Directory Services (MIDS)

# Anzahl der Internet Hosts



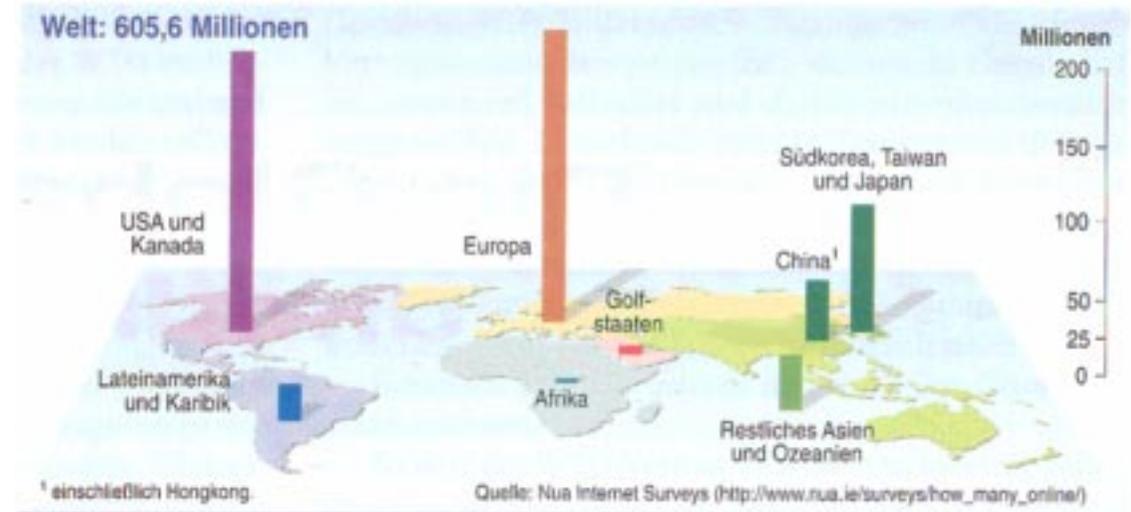
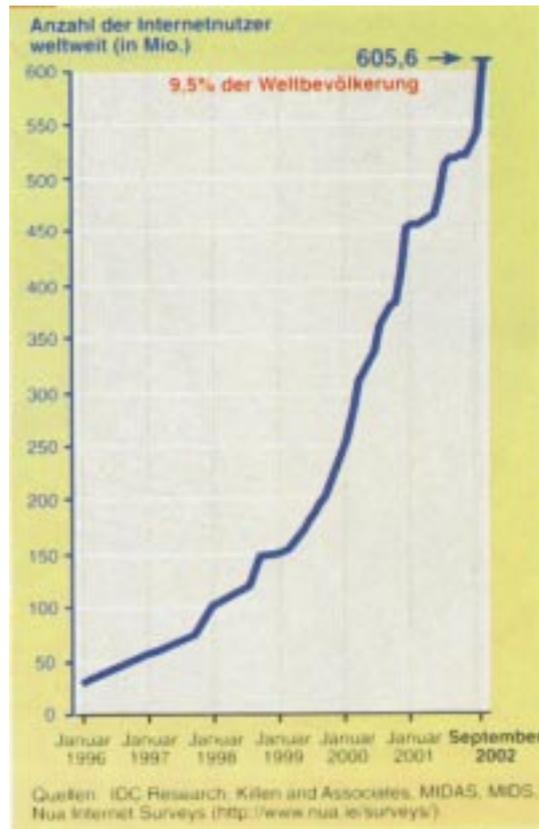
Anzahl der Internet Hosts pro1000 Einwohner in 2002

## Internet Hosts in Millionen



aus Data on Internet activity worldwide, <http://www.gandalf.it/data/data1.htm>

# Verteilung der Internet-Nutzer



aus LE MONDE diplomatique, Atlas der Globalisierung, taz Verlags- und Vertriebs GmbH

# Entwicklungsschritte zum Internet und WWW

- 1969** **ARPA-Net** wird vom US Department of Defense aufgebaut, anfangs 4 Universitäten: Utah, Los Angeles, Santa Barbara, Stanford
- 1972** Raymond S. Tomlinson: erstes **E-Mail**-System im Internet
- 1983** **TCP/IP** (Transmission Control Protocol / Internet Protocol): standardisiertes technisches Verfahren zum Versand von Daten: Datenmengen werden in **Pakete** zerlegt, jedes sucht sich unabhängig einen Weg zum Empfänger, dort werden sie wieder zusammengesetzt
- 1991** **Tim Berners-Lee** begründet am CERN ein Projekt zur weltweiten Verknüpfung von Medienelementen (Hypertext): **World Wide Web**. Schafft die Grundlagen für den Zugriff auf vernetzte Medien im Internet:  
**HTTP** (Hypertext Transfer Protocol): regelt Kommunikation im WWW  
**URL** (Uniform Resource Locator): Identifikation von Dateien im WWW  
**HTML** (Hypertext Markup Language): Sprache zur Beschreibung von Web-Seiten
- 1992** Marc Andreessen entwickelt Mosaic, den ersten **Internet Browser** mit grafischer Benutzungsoberfläche, später hat er Netscape mitgegründet
- 1994** World Wide Web Consortium (**W3C**) am MIT von Tim Berners-Lee gegründet; koordiniert die Weiterentwicklung technischer Standards zum WWW

# Basisdienste im Internet

## **Client/Server-Prinzip:**

Einige Rechner (Server) bieten eine Dienstleistung an,  
andere Rechner (Clients) nutzen den Dienst

Solche Dienste sind im Internet:

## **E-Mail (electronic mail):**

Versand von Nachrichten und Dateien über das Internet an E-Mail-Adressen;  
schnell, vielseitig, preiswert; verdrängt Briefe, Faxe, Telefonate;  
Form der Adressen: name@domain, z. B. mthies@uni-paderborn.de

## **Telnet:**

Anmelden und Arbeiten auf einem entfernten Rechner im Internet;  
unsicher, da unverschlüsselt; besser SSH (secure shell)

## **FTP (File Transport Protocol):**

Dateien von oder zu einem ans Internet angeschlossenen Rechner übertragen

## **WWW (World Wide Web, Web):**

Bereitstellen und Zugreifen von Hypertext-Dokumenten über das Internet

**... weitere Dienste ...**

# Software-Werkzeuge für das Internet

## Internet-Browser:

Integriertes Software-Werkzeug zur Benutzung des Internet:

- Anzeigen von Web-Seiten,
- Navigieren zu Web-Seiten,
- Ausführen von Programmen, die auf Web-Seiten stehen
- E-Mails schreiben, senden, empfangen, ablegen
- ...



Mosaic

historischer Browser

Netscape

früher Browser für alle Plattformen (seit 1994)

Mozilla, Firefox

offene Weiterentwicklungen von Netscape 6

Internet Explorer

Browser für Microsoft Windows

## Web-Server (HTTP-Daemon-Server):

Software, die auf einem an das Internet angeschlossenen Rechner läuft und Anfragen von anderen Rechnern (Clients) bedient, z. B. Apache, FoxServ, Microsoft IIS und PWS

# Adressierung im Internet: IP-Adressen

## Internet-Protocol-Adresse (IP-Adresse):

Jeder Host-Rechner am Internet wird durch eine IP-Adresse identifiziert.

Sie besteht aus 4 Zahlen, je zwischen 0 und 255, z. B.

131.234.22.29 oder 216.239.41.99

Je nach Größe des Netzes sind die ersten 1, 2, oder 3 Zahlen die

**Nummer des Netzes**, die übrigen die **Nummer des Rechners** in diesem Netz:

Klasse	Netznummer	Hostnummer	Netze	mit je ... Hosts
A	1- 126 . _ _ _ . _ _ _ . _ _ _		126	16,7 Mio
B	128 - 192 . _ _ _ . _ _ _ . _ _ _		16384	65535
C	193 - 223 . _ _ _ . _ _ _ . _ _ _		2,1 Mio	256

Die insges.  $2^{32} \sim 4,3$  Mrd Nummern für Netze und Hosts im **IPv4 Standard** reichen bald nicht mehr aus. Deshalb wird IPv4 ersetzt durch

**IPv6** mit 8 Zahlen, je zwischen 0 und  $2^{16}-1 = 65535$  (entspricht 2 Byte)

Notation (hexadezimal) , z. B. 7D7E:7F80:8182:8384:8586:8788:898A:8B8C

Damit können theoretisch  $2^{128} \sim 3,4 * 10^{38}$  Adressen vergeben werden.

IP-Adressen eignen sich schlecht für den menschlichen Gebrauch; stattdessen ...

# Adressierung im Internet: Domain-Namen

## Domain-Namen:

Hierarchisches Schema von lesbaren Namen;

**Domain Name Server (DNS)** ordnen sie den IP-Adressen zu.

Punkte trennen Unter-Domains von höheren Domains,  
die **Top-Level-Domain** steht am Schluss:

www.ub.uni-paderborn. de

-----  
Unter-Domains

-----  
Top-Level-Domain

Beispiele für Top-Level-Domains:

Staaten:    de ch uk at au

Typen:     com edu org net gov mil

## URI (Uniform Resource Identifier):

Weltweit eindeutige Adresse eines Dokumentes auf einem Host-Rechner, z. B.  
<http://ag-kastens.upb.de/lehre/material/ews2006/organisation.html>

allgemein:

Protokoll://Host-Adresse/Pfad auf dem Host/Dateiname.Typ

# S1 Einführung in Web-bezogene Sprachen

## Wofür benötigt man Web-bezogene Sprachen?

### Gestaltung von Web-Dokumenten:

Beschreibung der Struktur, der Hypertext-Verweise, der Präsentation durch Annotationen (mark-up)

Sprachen: HTML, XML

### Programmierung von Web-Diensten:

Erzeugung individueller Web-Seiten, Zugriff auf Dateien und Datenbanken, interaktive Elemente, wie Formulare, auf Web-Seiten

**Script-Sprachen:** spezielle Programmiersprachen für solche Zwecke  
PHP, JavaScript, Perl, VBScript, ASP, SQL

Mit passenden Hilfsmitteln werden auch allgemeine Programmiersprachen eingesetzt, wie Java, C++

### beide Zwecke hängen zusammen - Sprachen werden integriert:

Web-Seiten (in HTML) enthalten Programme (in PHP),  
Programme (in PHP) erzeugen Web-Seiten (in HTML)

In dieser Vorlesung wird in die Benutzung von **HTML** und **PHP** eingeführt.  
JavaScript und XML werden kurz gezeigt.

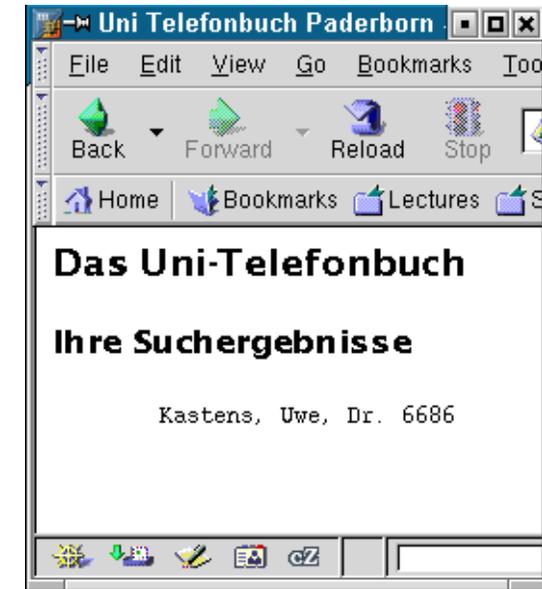
# Beispiel für eine dynamische, interaktive Web-Seite Telefonverzeichnis



**Anklicken der URL**  
fordert ein Formular an

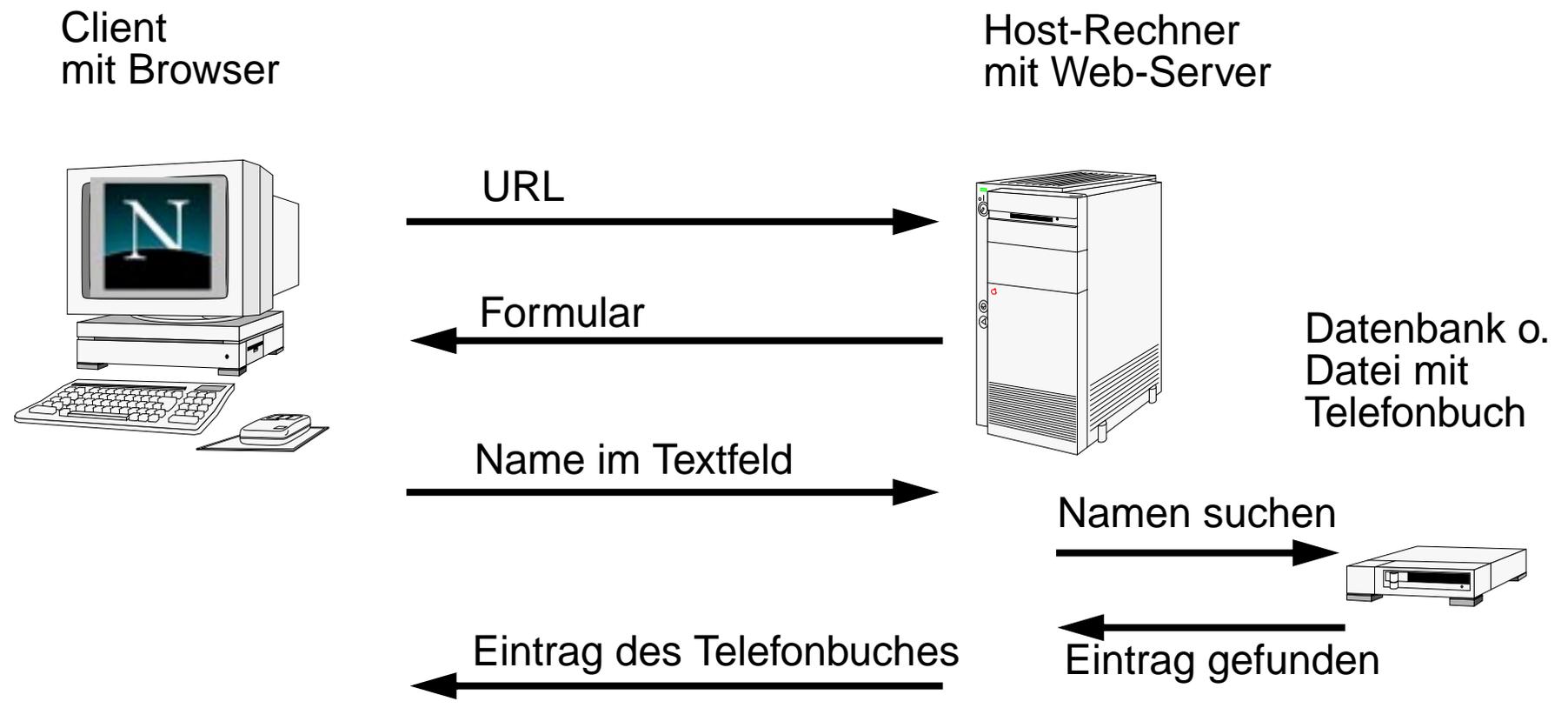


**Eintragen einer Anfrage**  
fordert eine Suche im  
Telefonbuch an

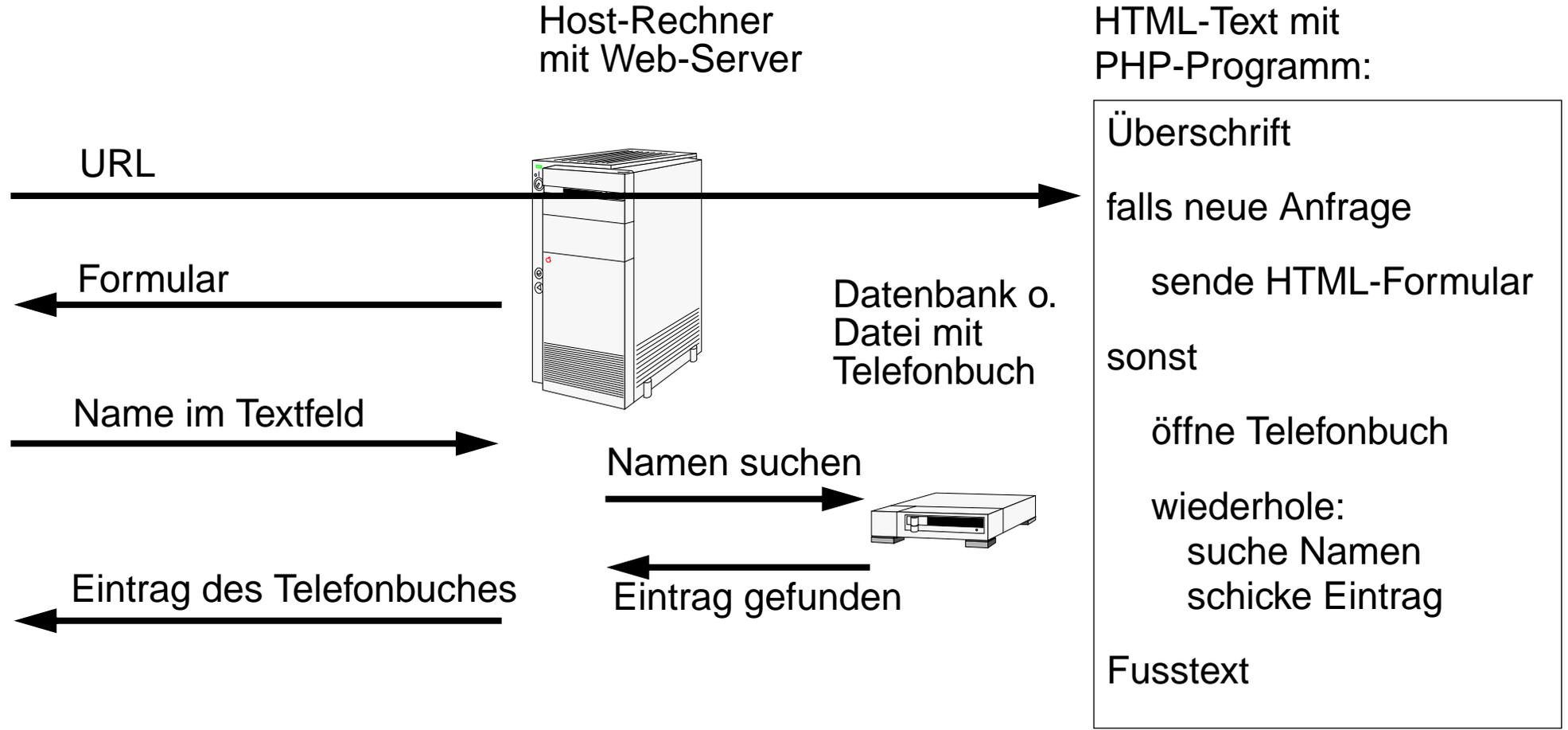


**Ergebnis der Suche**  
wird angezeigt

# Interaktion zwischen Client und Server



# Interaktion mit PHP-Programm im HTML-Text



# Struktur des PHP-Programms im HTML-Text

```

<html>
<head>
<title>Uni Telefonbuch Paderborn </title>
</head>
<body>
<h3>Das Uni-Telefonbuch</h3>
<?php // Typischer Aufbau einer PHP-Seite:
if (!isset($_REQUEST['wunsch'])) {
    echo <<<FORMULARANZEIGE
        <h4>Wen suchen Sie?</h4>
        <form action="http://ag-kastens/.../unitel.php"
            method="POST">
        <p><input type=text name=wunsch>
        <p><input type=submit value=Abschick>
        </form>
        FORMULARANZEIGE;
}
else {
    $wunsch = $_REQUEST['wunsch'];
    echo "<h4>Ihre Suchergebnisse</h4><p>\n<pre>\n";
    $fp = fopen("telefonbuch.txt" ,"r");
    while (!feof($fp)) {
        $line = fgets($fp, 64);
        if (preg_match("/$wunsch/i", $line)) {
            echo "\t$line";
        }
    }
    fclose($fp);
}
?>
</pre>
</body></html>

```

HTML-Text mit  
PHP-Programm:

Überschrift

falls neue Anfrage

sende HTML-Formular

sonst

öffne Telefonbuch

wiederhole:

suche Namen

schicke Eintrag

Fusstext

# Ein erster Eindruck von HTML

```
<html>  
<head>  
<title>Uni Telefonbuch Paderborn </title>  
</head>  
<body>  
<h3>Das Uni-Telefonbuch</h3>
```

Das Dokument und seine **Textelemente werden annotiert** mit sogenannten **Tags**, z. B.

```
<title> ... </title>  
<h3> ... </h3>
```

Die Tags haben Namen und **kennzeichnen Anfang und Ende** einer Struktur.

Strukturen können **geschachtelt** sein.

So wird die **Struktur** des Dokumentes und seine **Darstellung** beschrieben.

```
</pre>  
</body></html>
```

# Ein erster Eindruck von PHP

```
if (!isset($_REQUEST['wunsch'])) {  
    echo <<<FORMULARANZEIGE  
        <h4>Wen suchen Sie?</h4>  
        <form action="http://ag-kastens/.../unitel.php"  
            method="POST">  
        <p><input type=text name=wunsch>  
        <p><input type=submit value=Abschick>  
        </form>  
    FORMULARANZEIGE;  
}  
else {  
    $wunsch = $_REQUEST['wunsch'];  
    echo "<h4>Ihre Suchergebnisse</h4><p>\n<pre>\n";  
    $fp = fopen("telefonbuch.txt" ,"r");  
    while (!feof($fp)) {  
        $line = fgets($fp, 64);  
        if (preg_match("/$wunsch/i", $line)) {  
            echo "\t$line";  
        }  
    }  
    fclose($fp);  
}
```

PHP ist eine Programmiersprache mit

- **Ablaufstrukturen** wie bedingten Anweisungen und Schleifen:

```
if (...) { .. } else { .. }
```

```
while (...) { ... }
```

- **Variablen, Zuweisungen und Funktionsaufrufen:**

```
$line = fgets ($fp, 64);
```

Das Programm wird vom Web-Server ausgeführt. Die Ausgabe wird auf dem Browser des Client angezeigt.

# Integration von PHP-Programm und HTML-Text

```
<html>
<head>
<title>Uni Telefonbuch Paderborn </title>
</head>
<body>
<h3>Das Uni-Telefonbuch</h3>
<?php // Typischer Aufbau einer PHP-Seite:
if (!isset($_REQUEST['wunsch'])) {
    echo <<<FORMULARANZEIGE
        <h4>Wen suchen Sie?</h4>
        <form action="http://ag-kastens/../../unitel.php"
            method="POST">
            <p><input type=text name=wunsch>
            <p><input type=submit value=Abschick>
        </form>
        FORMULARANZEIGE;
}
else {
    $wunsch = $_REQUEST['wunsch'];
    echo "<h4>Ihre Suchergebnisse</h4><p>\n<pre>\n";
    $fp = fopen("telefonbuch.txt" ,"r");
    while (!feof($fp)) {
        $line = fgets($fp, 64);
        if (preg_match("/$wunsch/i", $line)) {
            echo "\t$line";
        }
    }
    fclose($fp);
}
?>
</pre>
</body></html>
```

PHP-Programm in HTML-Dokument eingebettet, geklammert durch

```
<?php
...
?>
```

HTML-Text wird vom PHP-Programm ausgegeben, durch echo-Anweisung:

```
echo "<h4>...</h4>";
```

# E1. Einführung zu Eigenschaften von Sprachen

**Sprachen in der Informatik** werden

- für bestimmte **Zwecke** geschaffen  
hier: **Auszeichnungssprachen** (HTML) und  
**Skriptsprachen** (PHP, Javascript)  
weitere Aufgabengebiete für Sprachen in dieser Einführung;
- je nach Zweck und **Niveau** mit einfachen oder komplexen,  
wenigen oder zahlreichen **Sprachkonstrukten** ausgestattet;
- durch **Regeln formal oder informell definiert**; sie legen fest:  
**Notation** der Symbole (Lexeme),  
**Struktur** der Sätze (Syntax),  
**Bedeutung** der Konstrukte (Semantik);
- durch Software-Werkzeuge **übersetzt** oder **interpretiert**

Alle diese Aspekte beeinflussen die **Eigenschaften der Sprachen**.

Die **Verbreitung der Sprachen** wird auch beeinflusst durch

- Erlernbarkeit und Handhabbarkeit,
- Verfügbarkeit von Werkzeugen,
- Marktmechanismen

## 4 Ebenen der Spracheigenschaften

Ein **Satz einer textuellen\* Sprache** ist eine **Folge von Zeichen** eines zu Grunde liegenden **Alphabetes**

Beispiel: ein PHP-Programm ist ein Satz der Sprache PHP;  
hier ein Ausschnitt daraus:

```
$line = fgets ($fp, 64);
```

Die **Struktur eines Satzes** wird in 2 Ebenen definiert:

1. **Notation von Grundsymbolen (Lexemen, token)**
2. **Syntaktische Struktur**

Die **Bedeutung eines Satzes** wird in 2 weiteren Ebenen an Hand der Struktur für jedes Sprachkonstrukt definiert:

### 3. **statische Semantik**

Eigenschaften, die vor der Ausführung bestimmbar sind.

### 4. **dynamische Semantik**

Eigenschaften, die erst während der Ausführung bestimmbar sind.

Auf jeder der 4 Ebenen gibt es auch Regeln, die korrekte Sätze erfüllen müssen.

\*) Es gibt auch **visuelle Sprachen**. Ihre Sätze werden aus graphischen Symbolen zusammengesetzt.

# Ebene 1: Notation von Grundsymbolen

Ein **Grundsymbol** wird aus einer **Folge von Zeichen des Alphabetes** gebildet. Die Regeln zur Notation von Grundsymbolen werden z. B. durch **reguläre Ausdrücke formal definiert** (siehe E2).

```
$line = fgets ($fp, 64);
```

## Typische Grundsymbole in Programmiersprachen

### 4 Symbolklassen:

Beispiele aus PHP

#### Bezeichner (identifizier)

`$line`      `fgets`

Namen für Variable, Funktionen, ...

#### Literale (literals)

`64`      `"telefonbuch.txt"`

Zahlwerte, Zeichenreihen

#### Wortsymbole (keywords)

`while`      `if`

kennzeichnen Sprachkonstrukte

#### Spezialzeichen

`<=`    `=`    `;`    `{`    `}`

Operatoren, Separatoren

**Zwischenräume, Tabulatoren, Zeilenwechsel** und **Kommentare** zwischen den Grundsymbolen dienen der Lesbarkeit und sind sonst bedeutungslos

#### Kommentar

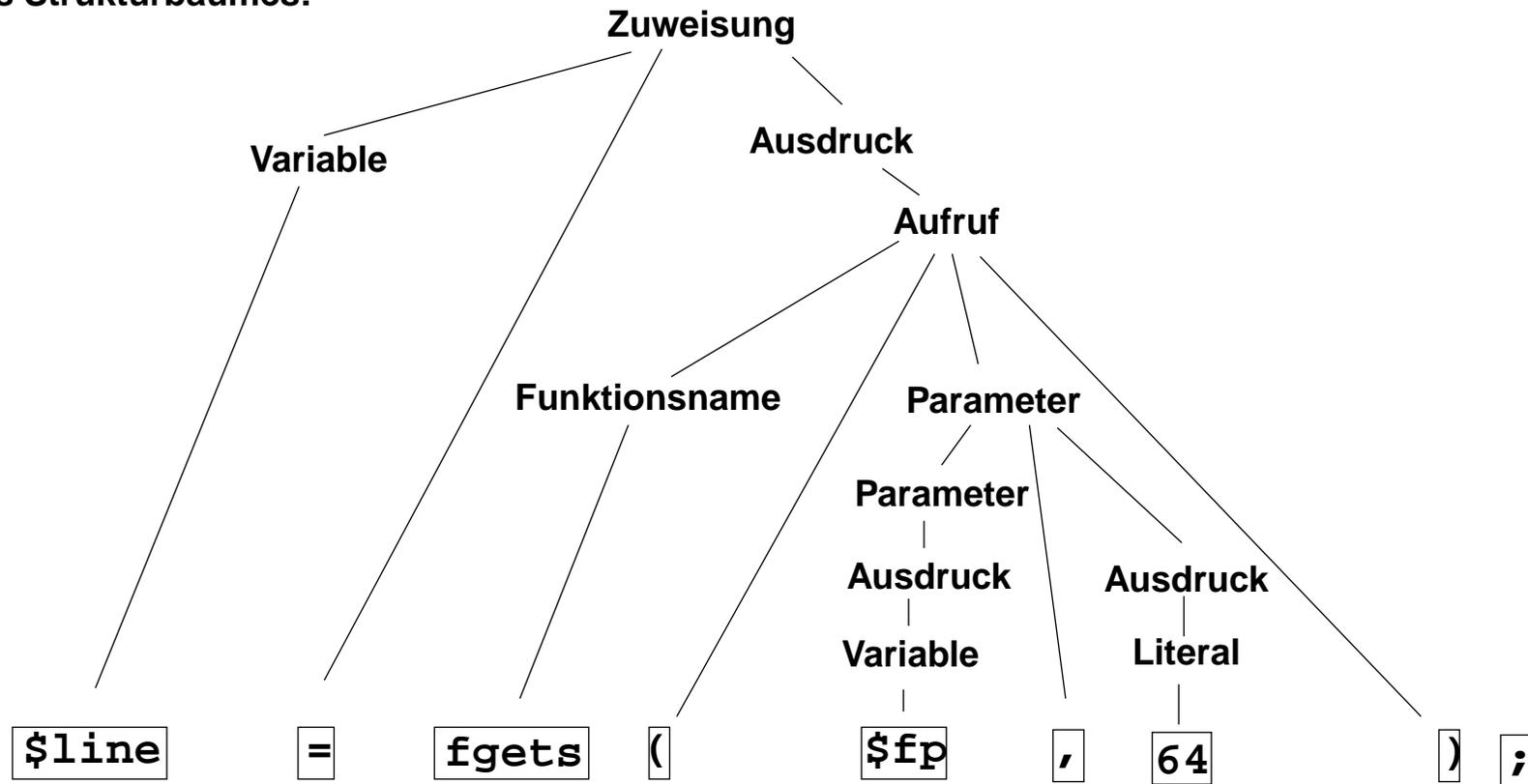
`/* Kommentar in C-Notation */`

# Ebene 2: Syntaktische Struktur

Ein Satz einer Sprache wird **in seine Sprachkonstrukte** gegliedert. Sie sind meist ineinander **geschachtelt**. Diese **syntaktische Struktur** wird durch einen **Strukturbaum** dargestellt. Die **Grundsymbole sind Blätter** in diesem Baum.

Die Syntax einer Sprache wird durch eine **kontextfreie Grammatik präzise definiert**. Die Grundsymbole sind die **Terminalsymbole** der Grammatik (siehe E2).

Teil des Strukturbaumes:



## Ebene 3: Statische Semantik

**Eigenschaften von Sprachkonstrukten**, die ihre Bedeutung (Semantik) beschreiben, soweit sie **an der Programmstruktur festgestellt** werden können (**statisch**), ohne das Programm auszuführen.

**Typische Eigenschaften der statischen Semantik** (für **übersetzte Sprachen**):

- **Bindung von Namen:**

Regeln, die einer **Anwendung** eines Namens seine **Definition** zuordnen.

z. B. „Zu dem Funktionsnamen in einem Aufruf muss es eine Funktionsdefinition mit gleichem Namen geben.“

- **Typregeln:**

Sprachkonstrukte wie **Ausdrücke** und **Variable** liefern bei ihrer Auswertung einen **Wert eines bestimmten Typs**. Er muss im Kontext zulässig sein und kann die Bedeutung von Operationen näher bestimmen.

z. B. „Die Operanden des + Operators müssen Zahlwerte sein.“

5 + "Text" ist in vielen Sprachen ein Typfehler

In der Sprache **PHP** gehören

die **Bindungsregeln** zur **statischen Semantik**,

die **Typregeln** aber zur **dynamischen Semantik**, da sie erst bei der Ausführung des Programms angewandt werden können.

## Ebene 4: Dynamische Semantik

**Eigenschaften von Sprachkonstrukten**, die ihre Wirkung beschreiben und **erst bei der Ausführung (dynamisch) bestimmt oder geprüft** werden können.

Typische Regeln der **dynamischen Semantik** beschreiben

- welche **Voraussetzungen für eine korrekte Ausführung** eines Sprachkonstruktes erfüllt sein müssen,  
z. B.  
„Ein numerischer Index einer Array-Indizierung, wie in `$var[$i]`, darf nicht kleiner als 0 sein.“
- welchen **Effekt die Ausführung** eines Sprachkonstruktes verursacht,  
z. B.  
„Eine Zuweisung der Form *Variable = Ausdruck* wird wie folgt ausgewertet:  
Die Speicherstelle der Variablen auf der linken Seite wird bestimmt.  
Der Ausdruck auf der rechten Seite wird ausgewertet.  
Das Ergebnis ersetzt dann den Wert an der Stelle der Variablen.“

In der Sprache PHP gehören auch die Typregeln zur dynamischen Semantik.

# Übersetzung von Sprachen

Ein **Übersetzer** transformiert jeden korrekten Satz (Programm) der **Quellsprache** in einen **gleichbedeutenden Satz der Zielsprache**.

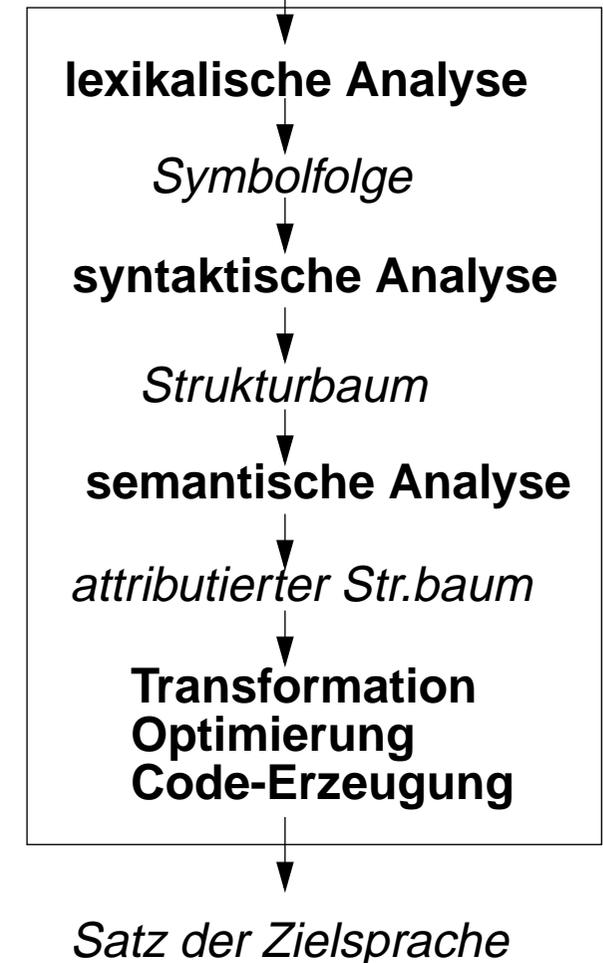
- Die meisten **Programmiersprachen zur Software-Entwicklung** werden übersetzt, z. B. C, C++, Java, Ada, Modula
- Zielsprache ist dabei meist eine **Maschinensprache** eines realen Prozessors oder einer abstrakten Maschine.
- Übersetzte Sprachen haben eine **stark ausgeprägte statische Semantik**.
- Der Übersetzer prüft die Regeln der statischen Semantik, wie Bindungs- und Typregeln; er findet viele Arten von **Fehlern vor der Ausführung**.

Es gibt auch **Übersetzer für andere Sprachen**:

Textformatierung:	LaTeX	-> PostScript
Spezifikationssprachen:	UML	-> Java

## Übersetzer und seine Phasen

*Satz der Quellsprache*



# Interpretation von Sprachen

Ein **Interpreter** liest einen Satz (Programm) einer Sprache und führt ihn aus.

Sprachen, die so **strikt interpretiert** werden:

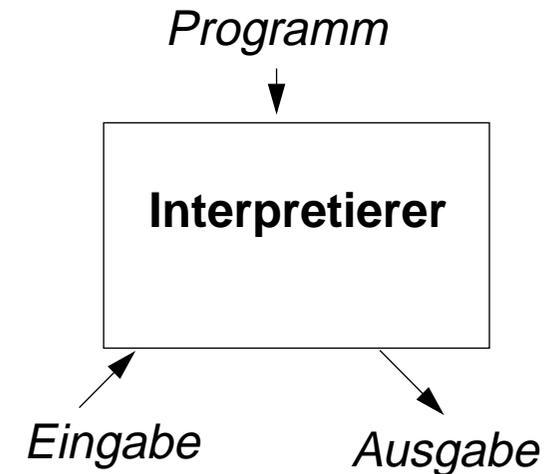
- haben **einfache Struktur** und **keine statische Semantik**,
- Bindungs- und Typregeln werden erst bei der Ausführung geprüft,
- nicht ausgeführte Programmteile bleiben ungeprüft,

z. B. Lisp, Prolog

Manche Interpreter erzeugen vor der Ausführung eine **interne Repräsentation des Satzes**;  
dann können auch Struktur und Regeln der statischen Semantik vor der Ausführung geprüft werden,  
z. B. **Skriptsprachen** PHP, JavaScript, Perl

Interpreter können **auf jedem Rechner verfügbar** gemacht werden und in andere Software (Browser) **integriert** werden.

Interpretation kann 10-100 mal **zeitaufwändiger** sein als die Ausführung von übersetztem Maschinencode.



# Zwecke von Sprachen: allgemeine Software-Entwicklung

## Anforderungen:

- Algorithmen klar und effizient formulieren
- komplexe Datenstrukturen klar und effizient formulieren
- prüfbare Regeln (statische Semantik) einhalten, dadurch Fehler reduzieren
- modulare Gliederung großer Programme mit expliziten Schnittstellen
- Schnittstellenprüfung beim Zusammensetzen von Bibliothekskomponenten

## Konsequenzen:

- umfangreiche, komplexe Sprachen: viele Konstrukte, viele Regeln
- relativ hoher Schreibaufwand, durch explizite, redundante Angaben

## Nutzen

hoch bei großen Software-Systemen  
recht umständlich für kleine, einfache Aufgaben

**Sprachstile:** imperativ, objektorientiert, (funktional)

**Sprachen:** Modula-2, Ada, C++, Eiffel, Java, (SML)

# Zwecke von Sprachen: Skriptsprachen

## **Scripting:**

Zusammensetzen von Kommandos zu einem wiederverwendbaren „Drehbuch“

## **Anforderungen:**

- kleine, einfache Aufgaben lösen ohne komplexe Algorithmen und Datenstrukturen
- existierende Funktionen nutzen und verknüpfen
- Verzicht auf Sicherheit durch prüfbare Regeln zugunsten kürzerer Programme
- Textverarbeitung und Ein- und Ausgabe sind wichtig
- gute Verfügbarkeit und Handhabbarkeit

## **Konsequenzen:**

- einfache Sprachen: wenige Konstrukte, wenige Regeln, kurze Programme
- dynamische Typprüfung
- interpretiert, d. h. ohne Übersetzung ausführbar

**Herkunft:** Kommandosprachen von Betriebssystemen, JCL, Unix Shell

**Sprachstile:** imperativ, objektorientiert

**Sprachen:** PHP, Perl, JavaScript, Python

# Zwecke von Sprachen: Auszeichnungssprachen

**Auszeichnungssprache:** Markup language

## **Anforderungen:**

- Struktur von Dokumenten beschreiben: Überschriften, Absätze, Listen, Tabellen
- hierarchische Gliederung, auch Hypertext-Verweise
- Darstellung der Strukturen beschreiben - aber abtrennbar
- von Menschen schreib- und lesbar
- keine Programmierung - aber integrierbar

## **Konsequenzen:**

- Strukturelemente werden mit lesbaren Markierungen gekennzeichnet (markup)
- geklammerte, geschachtelte Strukturen
- Menge und Bedeutung der Markierungen festlegbar (SGML, XML)
- Darstellung getrennt beschreibbar (HTML + CSS)

**Sprachen:** SGML, HTML, XML

# Sprachen für weitere Zwecke

## Zugriff auf Datenbanken

Selektionen aus Relationen, Verknüpfungen  
übersetzt in Datenbankzugriffe

SQL

## Spezifikation von Hardware-Bausteinen

VHDL

## Spezifikation von Software-Komponenten

UML

## allgemeine Spezifikation

SETL, Z

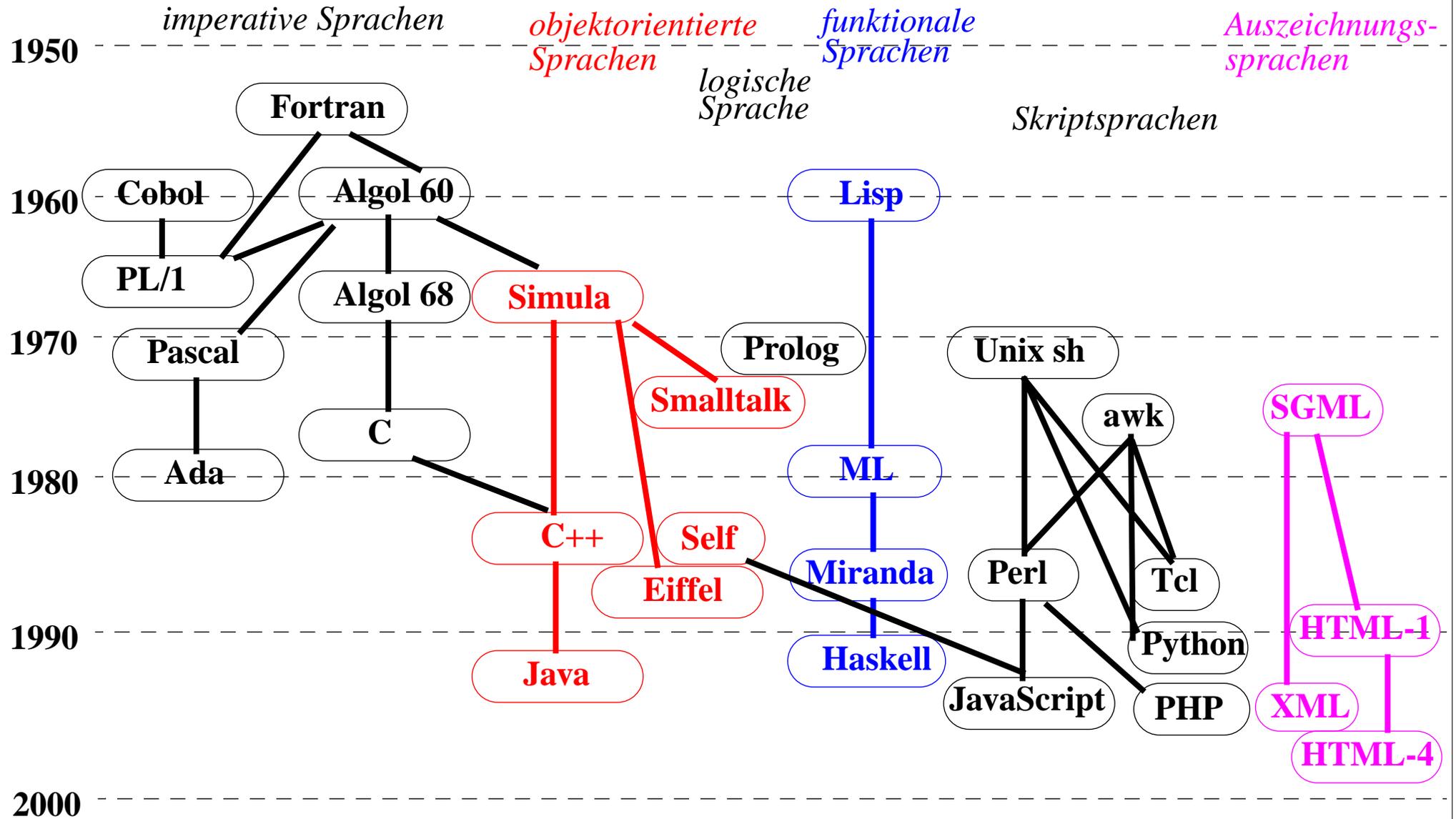
## Spezifikation von Grammatiken

EBNF

## Textsatz

TeX, LaTeX, PostScript

# Entstehungszeit und Verwandtschaft wichtiger Sprachen



nach [D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5] und [Computer Language History <http://www.levenez.com/lang>]

## S2 HTML

**HTML** steht für **Hypertext Markup Language**: eine **Auszeichnungssprache**, in der man **Dokumente** durch **Hyperlinks (Verweise)** miteinander **verknüpfen** kann.

Auszeichnungen (Markups) sind typografische Anweisungen für

- die Strukturierung von Texten in Abschnitte, Absätze, Listen, Tabellen, usw.
- die Gestaltung von Zeichen: Schrifttyp, Schriftgrad, Schriftstil, Schriftfarbe
- die Bildmontage: Platzierung und Größe
- das Layout des Dokumentes: Tabellen und Frames
- die Verknüpfung von Dokumenten: Anker, Links, sensitive Bildbereiche
- die Dialoggestaltung: Formulare, Schaltelemente

HTML-Markups oder **Tags** haben die Form `<table> ... </table>`  
Sie treten meist als **Klammern von Anfangs- und Ende-Tag** auf.

Tags können zusätzlich **Attribute** haben,

```
<table border="3" bgcolor="#2332FF">
```

HTML wurde ursprünglich aus der **Meta-Sprache SGML** abgeleitet.

HTML wird nun auch aus der **Meta-Sprache XML** abgeleitet: XHTML.

## S2.1 Auszeichnung von Strukturen und Texten

```
<!DOCTYPE HTML PUBLIC
  "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Hallo Welt</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```



Grundstruktur jeder HTML-Datei:

**<!DOCTYPE ...>**:  
Kommentar mit optionaler Angabe der HTML Definition

**<html>**:  
äußere Klammer

**<head>**:  
Angaben über das Dokument

**<body>**:  
Inhalt des Dokumentes

# Schreibweise von Tags und Attributen

Tags haben die Form

`<TagName>`

Anfangs-Tag

`</TagName>`

Ende-Tag

Zwischen `<` und `TagName` dürfen keine Zwischenräume stehen.

Tags, die **Strukturen** kennzeichnen, sollen **immer klammernd** geschrieben werden, auch wenn das Ende-Tag optional ist.

Tags können **Attribute** haben.

Sie ordnen dem gekennzeichneten Bereich bestimmte Eigenschaften zu.

Jedes Attribut hat die Form

`Name="Wert"`

z. B. `border="4"`

Das Anfangs-Tag kann zwischen dem Tag-Namen und dem `>` eine Folge von Attributen enthalten

`<table border="4" frame="box">`

Verwenden Sie **nur notwendige Attribute**. Viele Attribute sind veraltet oder werden von verschiedenen Browsern unterschiedlich behandelt.

Detaillierte Angaben zur Darstellung werden besser mit anderen Mitteln (z. B. Style Sheets, siehe Folie 2.32) gemacht als mit Attributen.

# Überschriften und Absätze

```
<html><head>
  <title>&Uuml;berschriften</title>
</head>
<body>
  <h1>1. Kapitel</h1>
  <p> In diesem Kapitel beschreiben wir
    die Tags von HTML.
    <br>Jeder Abschnitt ist einer Klasse
    von Tags gewidmet.
  </p>
  <h2>1.1 Struktur-Tags</h2>
  <p>Struktur-Tags gliedern ein
    Dokument hierarchisch in
    Abschnitte und Unterabschnitte.
  </p>
  <h3>Absatz-Tag</h3>
  <h4>Attribute des Absatz-Tag
  </h4>
```

**<h1>**, **<h2>**, ..., **<h6>**:  
Überschriften fallenden  
Ranges

**<p>**:  
Jeder Absatz wird explizit  
ausgezeichnet.

**<br>**:  
erzwingt Zeilenwechsel.



# Aufzählungen

```

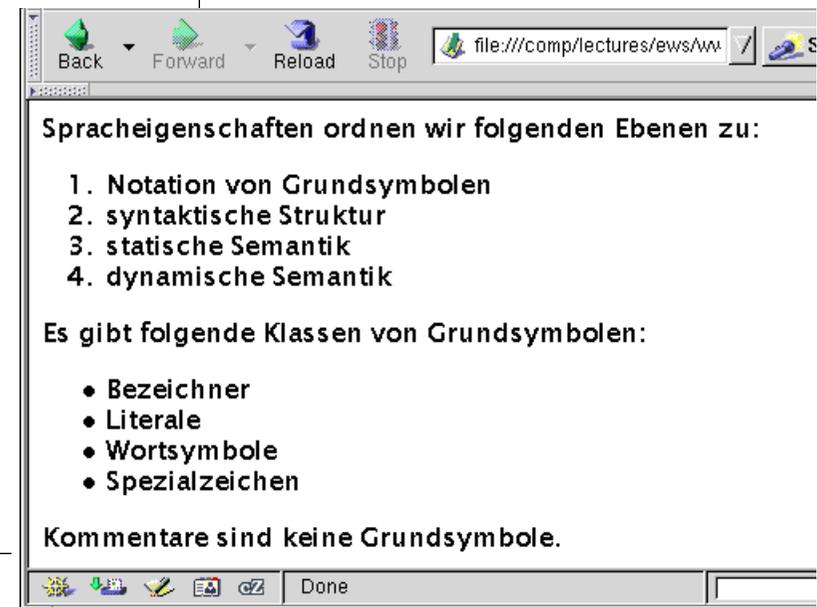
<html><head>
  <title>Aufzählungen</title>
</head><body>
  <p> Spracheigenschaften ordnen wir
    folgenden Ebenen zu:
    <ol>
      <li>Notation von Grundsymbolen</li>
      <li>syntaktische Struktur</li>
      <li>statische Semantik</li>
      <li>dynamische Semantik</li>
    </ol>
    Grundsymbolklassen sind:
    <ul>
      <li>Bezeichner</li>
      <li>Literale</li>
      <li>Wortsymbole</li>
      <li>Spezialzeichen</li>
    </ul>
    Kommentare sind keine
    Grundsymbole.
  </p>
</body></html>

```

**<ol>**:  
nummerierte Aufzählung  
von Textelementen

**<ul>**:  
Aufzählung von  
Textelementen mit  
Markierung („Bullet“)

**<li>**:  
Kennzeichnung der  
Textelemente



# Definitionslisten

```
<html><head>
  <title>Definitionslisten</title>
</head><body>
  <p> Wir definieren folgende Begriffe:
    <dl>
      <dt>Client/Server-Prinzip</dt>
      <dd>Einige Rechner (Server) bieten
        eine Dienstleistung an, andere
        Rechner (Clients) nutzen den
        Dienst.</dd>
      <dt>Gateway</dt>
      <dd>Rechner, der ein Netz mit
        anderen Netzen verbindet.</dd>
      <dt>W3C</dt>
      <dd>World Wide Web Consortium</dd>
    </dl>
  </p>
</body></html>
```

**<dl>**:

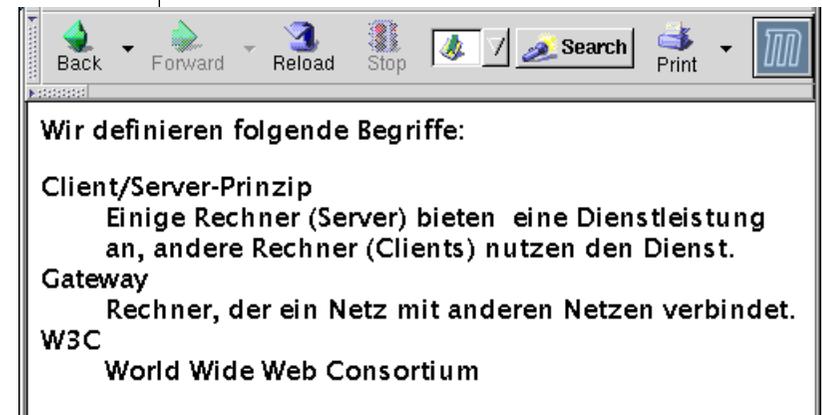
eine Liste mit Paaren  
von Begriffen und  
Erklärungen dazu

**<dt>**:

markiert den Begriff

**<dd>**:

markiert die Erklärung

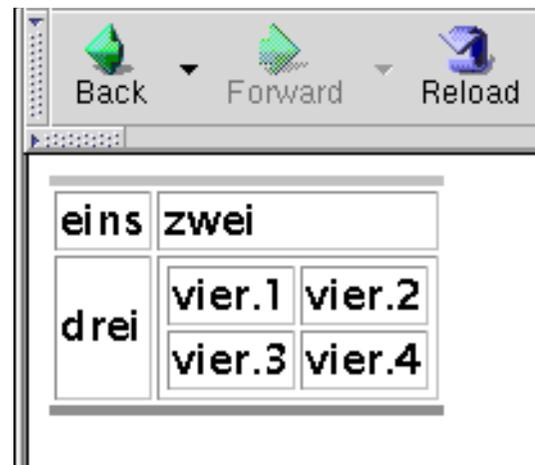


# Tabellen

```

<html><head>
  <title>Tabellen</title>
</head><body>
  <table border="4" frame="hsides">
    <tr><td>eins</td> <td>zwei</td>
    </tr>
    <tr><td>drei</td>
      <td><table border="2" frame="void">
        <tr><td>vier.1</td>
          <td>vier.2</td>
        </tr>
        <tr><td>vier.3</td>
          <td>vier.4</td>
        </tr>
      </table>
    </td>
  </tr>
</table>
</body></html>

```



**<table>**:  
eine Tabelle

Attribut **border**:  
Breites des Randes um  
die Zellen

Attribut **frame**:  
äußerer Rahmen;  
Werte:  
void, box,  
above, below,  
hsides, vsides,  
lhs, rhs

**<tr>**:  
Tabellenzeile

**<td>**:  
Tabellenzelle,  
kann selbst eine  
Tabelle enthalten

# Anker

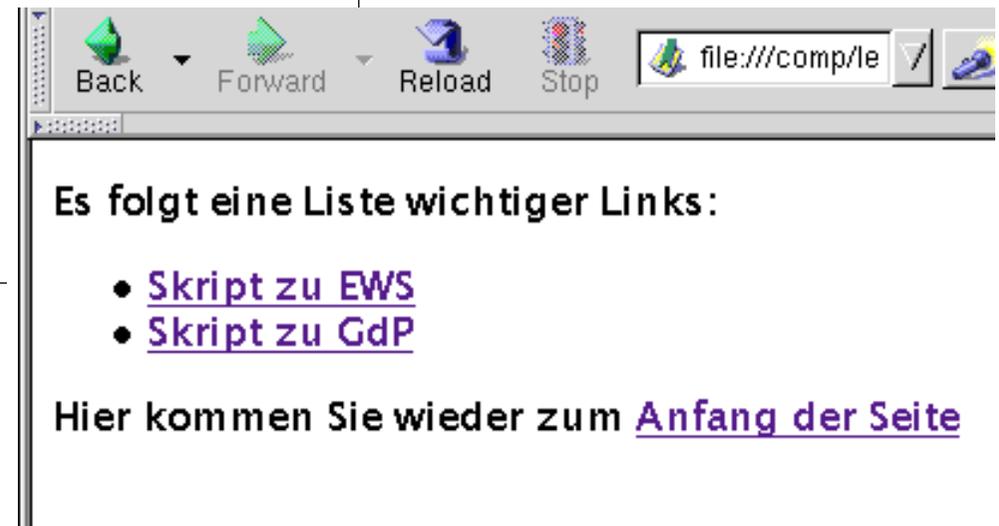
```

<html><head>
  <title>Anker</title>
</head><body>
  <a name="SeitenAnfang">
  <p> Es folgt eine Liste wichtiger Links:
  <ul>
    <li> <a href="http://ag-kastens.upb.de/lehre/material/ews">
      Skript zu EWS</a>
    </li>
    <li> <a href="http://ag-kastens.upb.de/lehre/material/gdp">
      Skript zu GdP</a>
    </li>
  </ul>
  Hier kommen Sie wieder zum
  <a href="#SeitenAnfang">
  Anfang der Seite</a>
  </p>
</body></html>

```

**<a name="XYZ">:**  
platziert den Anker XYZ;  
wird adressiert mit #XYZ

**<a href="Adr">Text  
</a>:**  
setzt einen Link zu Adr  
mit dem angegebenen  
**Text**



## Repräsentation spezieller Zeichen

Zeichen, die in der Notation von **HTML eine spezielle Bedeutung** haben, müssen **umschrieben** werden, wenn sie in normalem Text nicht als HTML-Zeichen gemeint sind

<	&lt;	<p>Das Tag <b>&amp;lt;p&amp;gt;</b> kennzeichnet einen Absatz</p>
>	&gt;	
&	&amp;	für jedes Zeichen gibt es auch einen numerischen Code: <b>&amp;#38;</b>
"	&quot;	
	&nbsp;	non-breaking space, Leerzeichen, das nicht durch Zeilenumbruch ersetzt werden darf

**Allgemeine Form:** **&Zeichename;** oder **&#Zahl;** wobei die Zahl das Zeichen codiert

HTML-Beschreibungen enthalten Listen mit solchen Zeichendefinitionen (*entities*), z. B.

ä	&auml;	
Ä	&Auml;	
ß	&szlig;	<p>Ma&szlig;kr&uuml;ge</p>
ñ	&ntilde;	<p>El ni&ntilde;o</p>
©	&copy;	<p>&copy;2006 bei Dr. M. Thies</p>

## Zweck-bezogene Hervorhebungen von Text

Textstücke können gekennzeichnet werden, damit sie durch **besonderen Schriftsatz hervorgehoben** werden. Dafür sollte man den **Zweck der Hervorhebung** kennzeichnen und **nicht die Darstellung** festlegen, z. B.

```
<p>Vor Verlassen des Raumes das <em>Licht ausschalten</em>,
aber <strong>nicht den Notausschalter</strong> benutzen!
```

Hier ist der Zweck, **Betonung** und **starke Betonung**, angegeben. Die Darstellung im Schriftsatz kann man unabhängig davon festlegen.

Das ist in folgendem Beispiel nicht mehr möglich und daher **nicht empfohlen**:

```
<p>Vor Verlassen des Raumes das <b>Licht ausschalten</b>,
aber <font color="red">nicht den Notausschalter</font> benutzen!
```

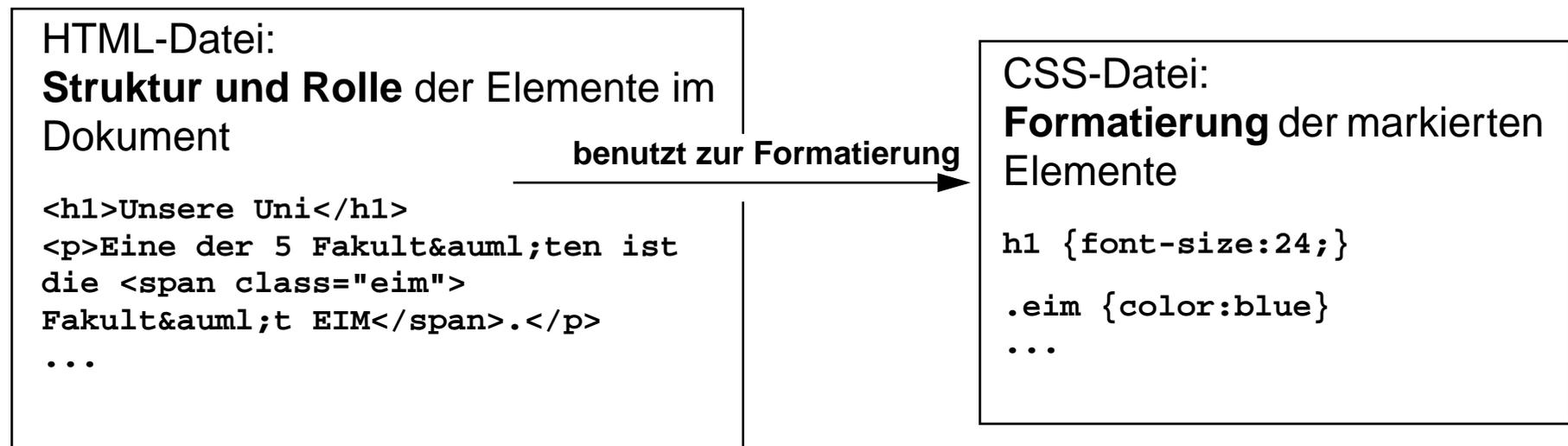
Weitere Beispiele für **Zweck-bezogene Hervorhebungen**:

<code>&lt;cite&gt;</code>	Zitat	Zweck wird durch das class-Attribut individuell bestimmt
<code>&lt;samp&gt;</code>	Beispiel	
<code>&lt;dfn&gt;</code>	Definition	
<code>&lt;code&gt;</code>	Quell-Code	<code>&lt;span class="meinAnteil"&gt;</code>
<code>&lt;kbd&gt;</code>	Tastatureingabe	für elementaren Fließtext
<code>&lt;var&gt;</code>	Variablenname	<code>&lt;div class="meinAnteil"&gt;</code>
		für Textblöcke mit Unterstrukturen

## S2.2 Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) ist eine Sprache zur **Definition von Formateigenschaften für HTML-Auszeichnungen**.

CSS wird verwendet, um **Formatierungsangaben** von den Auszeichnungen der Struktur und der Zweck-bezogenen Hervorhebungen zu **trennen**:



Ergebnisse:

- **konsistente Formatierung im ganzen Dokument**
- konsistente Formatierung **in vielen Dokumenten** (Corporate Identity)
- **einfache Änderung** der Formatierung in der CSS-Datei - HTML-Dateien bleiben unverändert

# Notationen von Formatangaben in CSS

Im <head>-Abschnitt der HTML-Datei wird ein **Link** auf die CSS-Datei eingetragen mit dem **relativen Dateinamen** oder der vollen URL:

```
<link rel="stylesheet" type="text/css" href="formate.css">
```

CSS-Angaben können auch in die HTML-Datei eingebettet werden - sind dann aber **nicht wiederverwendbar**: `<style type="text/css"> CSS-Angaben </style>`

Formatangaben in CSS für bestimmte Arten von HTML-Tags:

```
h1, h2 {text-align:center;color:red;}
```

```
em      {font-weight:bold;}
```

```
strong  {font-weight:bold;color:red;}
```

allgemeine Form: **HTML-Tags{ Folge von Formatangaben }**

Formatangabe: **Name:Wert;** keine Leerzeichen; nur mehrteilige Werte in "

Beispiele:

```
font-family:Times;  
font-style:italic;  
text-decoration:underline;  
list-style-type:disc;  
list-style-type:lower-alpha;
```

# Freie Klassifikation von HTML-Tags

Bestimmte **Zwecke der Hervorhebung** kann man durch **frei erfundene Namen klassifizieren**, z. B.

Die Idee, `<span class="meinAnteil">der Entwurf</span>` und die ..

```
<div class="meinAnteil">
```

```
<h3>Entwurf</h3>
```

```
<p>Das Software-System besteht aus ...</p>
```

```
...
```

```
</div>
```

In einem CSS Style Sheet kann man solchen Klassen Formatierangaben zuordnen:

```
.meinAnteil {font-weight:bold;color:blue;}
```

oder in einer anderen CSS-Datei bescheidener auf die Hervorhebung verzichten:

```
.meinAnteil {}
```

Auch den Struktur-Tags mit festgelegter Bedeutung können über das Klassenattribut Formateigenschaften zugeordnet werden:

```
<p class="meinAnteil">Der Entwurf des Systems ist ...</p>
```

Allerdings wäre eine Klammerung mit `<span>` konsequenter.

# Beispiel: 2 verschiedene Formatierungen

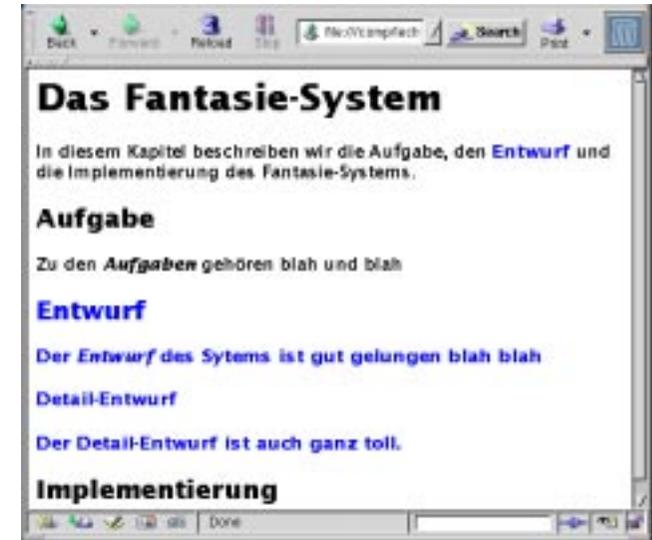
```
<html><head>
  <title>CSS Beispiel</title>
  <link rel="stylesheet" type="text/css"
    href="cssbsp.css">
</head><body>
  <h1>Das Fantasie-System</h1>
  <p> In diesem Kapitel beschreiben wir
  die Aufgabe, den
  <span class="ukas">Entwurf</span>
  und die Implementierung des
  Fantasie-Systems.</p>

  <h2>Aufgabe</h2>
  <p>Zu den <dfn>Aufgaben</dfn>
  geh&ouml;ren blah und blah</p>

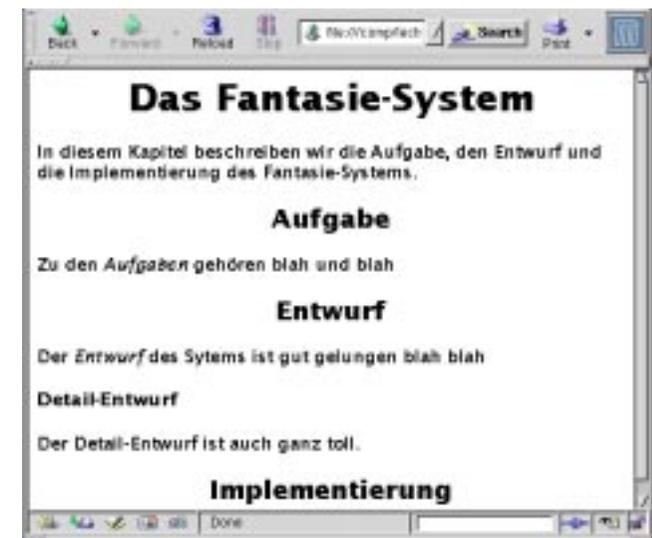
  <div class="ukas">
  <h2>Entwurf</h2>
  <p>Der <dfn>Entwurf</dfn> des Systems ist
  gut gelungen blah blah</p>

  <h4>Detail-Entwurf</h4>
  <p>Der Detail-Entwurf ist auch ganz
  toll.</p>
  </div>

  <h2>Implementierung</h2>
</body></html>
```



```
h1, h2, h3 {text-align:left;}
dfn      {font-style:italic;font-weight:bold;}
.ukas    {font-weight:bold;color:blue;}
```



```
h1, h2, h3 {text-align:center;}
dfn {font-style:italic;}
.ukas {}
```

## S2.3 Formulare mit Eingabeelementen

Graphische Elemente für **interaktive Eingaben**:

z. B. Textfelder, Auswahllisten, Schaltelemente, usw.

- strukturiert und gestaltet mit allgemeinen Dokument-Tags
- zu Formularen zusammengefasst

**Eingabedaten** werden

- zum **Web-Server gesandt** und dort verarbeitet,  
siehe Beispiel Telefonbuch;  
siehe Abschnitt *W4 Interaktive, dynamische Web-Seiten*
- **im Browser** geprüft und/oder verarbeitet  
siehe Abschnitt *S4 JavaScript*

The screenshot shows a web browser interface with a navigation bar at the top containing 'Back' and 'Forward' buttons, and 'Home' and 'Bookmarks' icons. Below the navigation bar is a form with several sections:

- Zuname:** A text input field containing the text 'Kastens'.
- Gästebuch:** A text area containing the text 'Das sieht noch recht mager aus!'.
- Versand:** A checkbox labeled 'eilig' which is checked.
- Zahlung:** Two radio buttons, 'Bar' (selected) and 'Karte'.
- Wochentag:** A dropdown menu with 'Freitag', 'Samstag' (highlighted), and 'Sonntag' options.
- At the bottom of the form are two buttons: 'löschen' and 'abschicken'.

# Ein einfaches Formular

```
<form action="http://www.upb.de" method="get">
  <p>Ihr Zuname:
    <input type="text" name="Zuname" size="10">
  </p>
  <input type="submit" value="abschicken">
</form>
```

**<form>**

fasst Eingabelemente (`<input>`) und weitere Elemente zu einem Formular **zusammen**. Alle Eingabedaten eines Formulars werden gemeinsam verarbeitet.

**action**-Attribut

**Ziel**, an das die Eingabe geschickt wird;  
Web-Adresse;  
Web-Seite enthält ggf. ein Programm, das die Eingabe verarbeitet

**method**-Attribut

charakterisiert die **Art der Übertragung** zum Ziel;  
hier **get**: als Parameter der URL



# Schaltflächen (Knöpfe)

```
<input type="button" value="zurück"
      onclick="javascript:history.back()">
<input type="reset" value="löschen">
<input type="submit" value="abschicken">
```

**<input type="button">**  
charakterisiert eine **allgemeine Schaltfläche**

**value**-Attribut

**Beschriftung** der Schaltfläche

**onclick**-Attribut

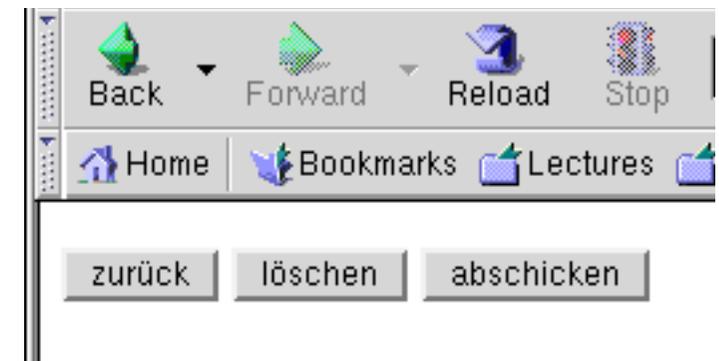
**Aktion**, die beim Betätigen ausgeführt wird;  
hier Aufruf einer Funktion in JavaScript

**<input type="reset">**

spezielle Schaltfläche: alle **Eingabeelemente** in den  
**Initialzustand zurücksetzen**

**<input type="submit">**

spezielle Schaltfläche: **Eingabe abschicken**;  
eine davon in jedem Formular!



# Einzeiliges Textfeld

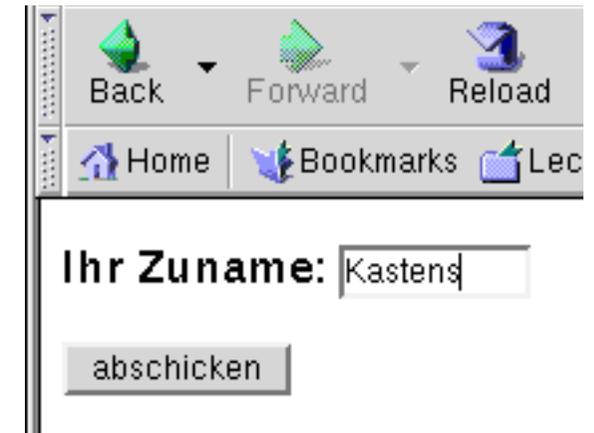
```
<input type="text" name="Zuname" size="10" maxlength="60">
```

`<input type="text">`  
charakterisiert **einzeiliges Textfeld**

**name**-Attribut  
**identifiziert das Eingabeelement**  
im verarbeitenden Programm

**size**-Attribut  
Größe des angezeigten Textfeldes in Zeichen

**maxlength**-Attribut  
Anzahl der Zeichen, die **maximal eingegeben**  
werden können  
Scrolling, wenn `maxlength > size`



# Mehrzeiliges Textfeld

```
<p>Eintrag ins Gästebuch:<br>  
  <textarea name="Eintrag" rows="5" cols="20">Hier schreiben!  
  </textarea>  
</p>
```

## <textarea>

charakterisiert **mehrzeiliges Textfeld**;  
zwischen den Tags kann  
**Initialisierung** stehen

## name-Attribut

**identifiziert das Eingabeelement**  
im verarbeitenden Programm

## rows-Attribut

Anzahl der angezeigten **Zeilen**

## cols-Attribut

Anzahl der angezeigten **Spalten**



# Auswahlliste

```
<p>Wochentag:<br>
  <select name="tag" size="3">
    <option value="fr">Freitag
    <option value="sa">Samstag
    <option value="so">Sonntag
    <option value="mo">Montag
  </select>
</p>
```

## <select>

Klammert eine Liste von Einträgen,  
aus der einer ausgewählt werden kann

## name-Attribut

**identifiziert das Eingabeelement**  
im verarbeitenden Programm

## size-Attribut

Anzahl der angezeigten **Einträge**

## value-Attribut

für das gewählte Element wird das  
Paar (**name, value**) übertragen



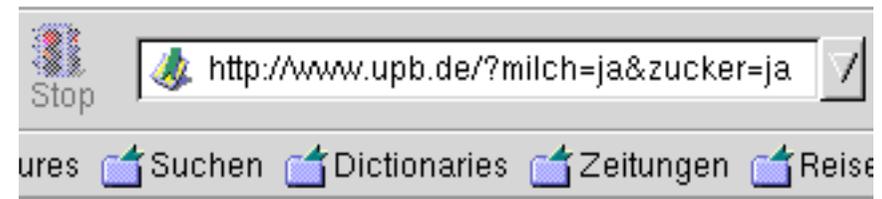
# Checkbox

```
<p>Nehmen Sie zum Kaffee<br>  
  <input type="checkbox" name="milch" value="ja">Milch<br>  
  <input type="checkbox" name="zucker" value="ja">Zucker<br>  
</p>
```

`<input type="checkbox">`  
ein Element, das man **auswählen** kann

**name**-Attribut  
identifiziert das Eingabeelement  
im verarbeitenden Programm

**value**-Attribut  
wenn das Element gewählt ist, wird das  
Paar (**name, value**) übertragen



# Auswahlknöpfe

```
<p>Zahlungsart :<br>  
  <input type="radio" name="pay" value="cash">Bar<br>  
  <input type="radio" name="pay" value="card">Kreditkarte<br>  
  <input type="radio" name="pay" value="order">&Uuml;berweisung  
</p>
```

**<input type="radio">**

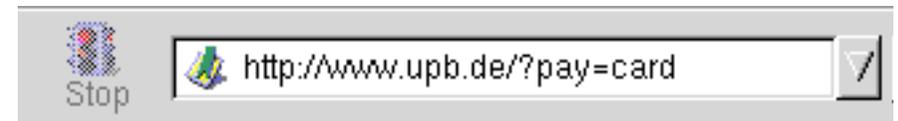
eine Gruppe von Knöpfen, von denen man genau einen **auswählen** kann (radio buttons)

**name**-Attribut

**identifiziert das Eingabeelement** im verarbeitenden Programm; alle Elemente der Gruppe haben den gleichen Namen

**value**-Attribut

wenn der Knopf gewählt ist, wird das Paar (**name, value**) übertragen



# Datei zum Web-Server senden

```
<form action="http://www.upb.de" method="get"
  enctype="multipart/form-data">
  <p>Dateiname:
    <input type="file" name="Datei">
  </p>
```

**<input type="file">**

**Datei versenden;**

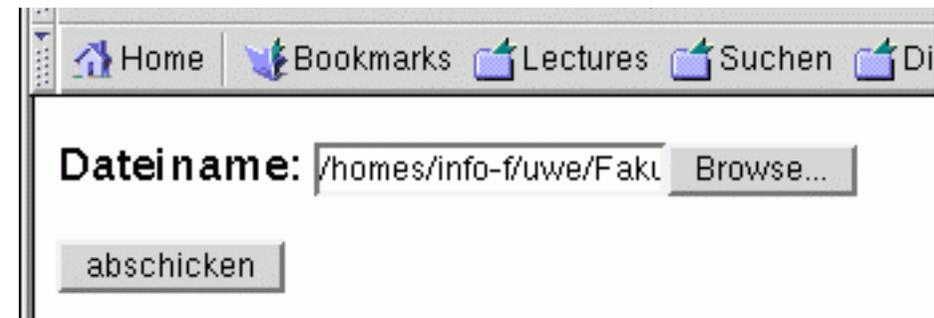
Name direkt angeben oder im  
Dateiselektor auswählen

**name**-Attribut

**identifiziert das Eingabeelement**

**enctype="multipart/form-data"**

weiteres Attribut im **form**-Tag nötig



## E2. Symbole und Syntax Überblick

Grundbegriffe zur **formalen Definition von Sprachen**:

- **Alphabet**: Menge von Zeichen
- **Wort**: Folge von Zeichen aus Alphabet - gebildet nach bestimmten Regeln (Ebene 1)  
**Satz**: Folge von Symbolen aus Vokabular - gebildet nach bestimmten Regeln (Ebene 2)  
(Wort, Satz), (Zeichen, Symbol) und (Alphabet, Vokabular) paarweise synonym
- **Sprache**: Menge von Worten bzw. Sätzen

**Kalküle** zur Bildung von Worten und Sätzen:

- **reguläre Ausdrücke**  
zur Definition der **Notation von Grundsymbolen**  
(Ebene 1 der Spracheigenschaften)  
  
zur Definition von **Textmustern**  
angewandt zum Suchen und Ersetzen von Zeichenfolgen  
programmiert in PHP, Perl, Unix-sh
- **kontextfreie Grammatik**  
zur Definition der **Menge der syntaktisch korrekten Sätze** einer Sprache  
(Ebene 2 der Spracheigenschaften)

**Struktur** der syntaktisch korrekten Sätze einer Sprache

# Alphabete und Zeichenfolgen

Ein **Alphabet** ist eine nicht-leere **Menge von Zeichen** zur Bildung von Zeichenfolgen.

Wir betrachten hier nur Alphabete mit endlich vielen Zeichen.

Alphabete werden in Formeln häufig mit  $\Sigma$  bezeichnet;  
sonst gibt man ihnen individuelle Namen oder benutzt sie unbenannt.

Beispiele:

$\Sigma =$	$\{T, F\}$
Dualziffern =	$\{0, 1\}$
Dezimalziffern =	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Kleinbuchstaben =	$\{a, b, \dots, z\}$
Nukleotide =	$\{A, C, G, U\}$
ASCII =	standardisierter Zeichensatz mit 128 Zeichen

Ein **Wort über einem Alphabet A** ist eine Folge von Zeichen aus A.

formal: eine Folge  $a_1 a_2 \dots a_n$ , mit  $a_i \in A$ , für  $i = 1, \dots, n$ .

$n$  ist die **Länge der Folge** bzw. die **Länge des Wortes**.

Beispiele: Wort der Länge 7 über dem Alphabet Dualziffern: 1001101

Die **leere Folge** bzw. das **leere Wort** wird mit  $\varepsilon$  (epsilon) bezeichnet.

# Reguläre Ausdrücke

Ein **regulärer Ausdruck**  $R$

definiert eine **Mengen von Worten** über einem Alphabet  $A$ , die **Sprache**  $L(R)$ .

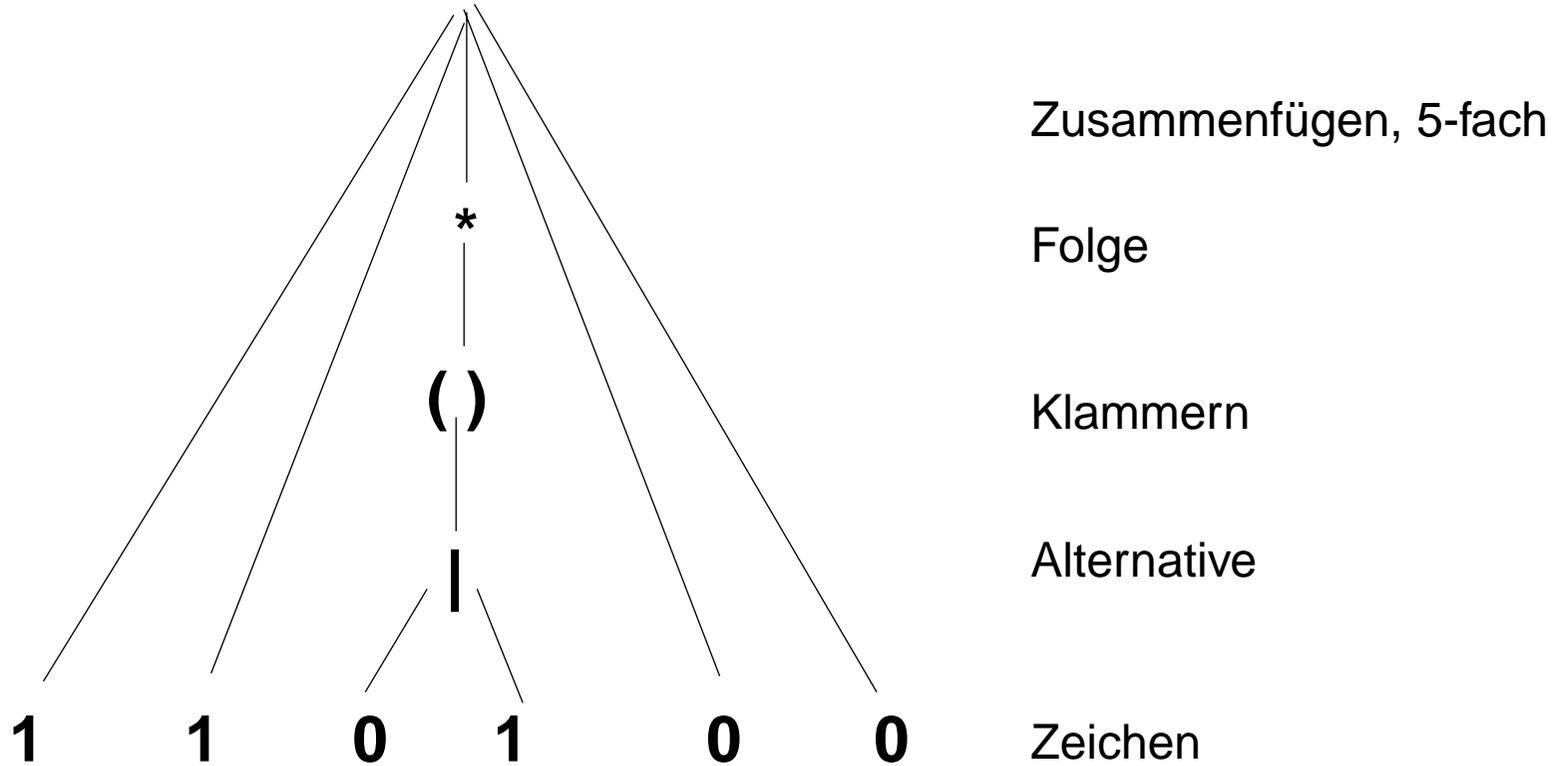
Ein regulärer Ausdruck kann aus folgenden 8 Formen rekursiv zusammengesetzt sein.

$F$  und  $G$  seien reguläre Ausdrücke.

regulärer Ausdruck $R$	Sprache $L(R)$	Erklärung
$a$	$\{ a \}$	Zeichen $a$ als Wort
$FG$	$\{ fg \mid f \in L(F), g \in L(G) \}$	Zusammenfügen von 2 Worten
$F \mid G$	$\{ f \mid f \in L(F) \} \cup \{ g \mid g \in L(G) \}$	Alternativen
$\varepsilon$	$\{ \varepsilon \}$	das leere Wort
$(F)$	$L(F)$	Klammerung
$F^+$	$\{ f_1 f_2 \dots f_n \mid f_i \in L(F), \text{ für } n \geq 1, i=1, \dots, n \}$	nicht-leere Folgen von Worten aus $L(F)$
$F^*$	$\{ \varepsilon \} \cup L(F^+)$	Folgen von Worten aus $L(F)$
$F^n$	$\{ f_1 f_2 \dots f_n \mid f_i \in L(F), \text{ für } i = 1, \dots, n \}$	Folgen von genau $n$ Worten aus $L(F)$

# Struktur eines regulären Ausdruckes

**1 1 ( 0 | 1 ) \* 0 0**



verbal:

Jedes Wort aus der Sprache dieses regulären Ausdruckes besteht aus zwei 1, gefolgt von beliebig vielen 0 oder 1, gefolgt von zwei 0.

## Beispiele für reguläre Ausdrücke

Name	regulärer Ausdruck $A$	Worte aus seiner Sprache $L(A)$
<i>Abc</i> =	$(a \mid b) (c \mid d \mid \varepsilon)$	ac bc ad bd a b
<i>Anrede</i> =	Sehr geehrte( $r \mid \varepsilon$ ) (Frau $\mid$ Herr)	Sehr geehrte Frau
<i>Dig</i> =	$0 \mid 1 \mid \dots \mid 9$	7
<i>sLet</i> =	$a \mid b \mid \dots \mid z$	x
<i>cLet</i> =	$A \mid B \dots \mid Z$	B
<i>Let</i> =	$sLet \mid cLet$	m N
<i>Bezeichner</i> =	$Let ( Let \mid Dig )^*$	Maximum min3 a
<i>GeldBetrag</i> =	$Dig^+, Dig^2$	23,95 0,50
<i>KFZ</i> =	$(cLet \mid cLet^2 \mid cLet^3) \cdot (cLet \mid cLet^2) \cdot (Dig \mid Dig^2 \mid Dig^3 \mid Dig^4)$	PB-AX-123
<i>Dual</i> =	$1^3 (1 \mid 0)^* 0^3$	1111000 111000 1111101010000

Wenn ***Namen*** von regulären Ausdrücken **in regulären Ausdrücken** verwendet werden, müssen sie von den Zeichen unterschieden werden können; hier *Namen* kursiv.

## Beispiele für Definitionen von Grundsymbolen

*Pascal\_Identifier* =  $Let (Let | Dig)^*$

*C\_Identifier* =  $(Let | \_ ) (Let | \_ | Dig)^*$

*ADA\_Identifier* =  $Let ( ( \_ | \varepsilon ) (Let | Dig) )^*$

*PHP\_Var\_Identifier* =  $\$ (Let | \_ ) (Let | \_ | Dig)^*$

*Pascal\_Real* =  $((Dig^+ \cdot Dig^+) ((e | E) (+ | - | \varepsilon) Dig^+) | \varepsilon) | (Dig^+ (e | E) (+ | - | \varepsilon) Dig^+)$

*HexDig* =  $Dig | a | b | c | d | e | f | A | B | C | D | E | F$

*HTML\_CharRef* =  $\& (Let^+ | \# Dig^+ | \#x HexDig^+) ;$

# Reguläre Ausdrücke als Textmuster

In Sprachen, die zur **Textverarbeitung** eingesetzt werden, benutzt man **reguläre Ausdrücke**, um **Textmuster zu definieren**.

## Beispiele:

suche alle Dateinamen der Form `ews(0|1|2|3|4|5|6|7|8|9)3.html`

in der Schreibweise von Unix-sh `ls ews[0-9][0-9][0-9].html`

Aufruf einer PHP-Funktion `preg_match ("/[dD]aß/", $absatz)`  
sucht ein Textmuster in einer Zeichenreihe      Muster      Zeichenreihe

```
$d = "[0-9]";
```

```
preg_match ("/ews$d$d$d\.html/", $files)
```

Reguläre Ausdrücke als Textmuster sind umfassend in der Skriptsprache Perl definiert und so in PHP übernommen.

Auf den vorigen Folien haben wir die **grundlegenden Begriffe** für reguläre Ausdrücke mit der **dafür üblichen Notation** eingeführt. In **PHP, Perl**, Unix-sh werden die gleichen Begriffe aber in anderer **Notation** verwendet.

# Notation von regulären Ausdrücken in PHP

$a$	das Zeichen $a$
$FG$	Zusammenfügen von 2 Worten
$F G$	Alternativen
$(F)$	Klammerung
$F?$	Option; wie $F \epsilon$
$F^+$	nicht-leere Folge von Worten aus $L(F)$
$F^*$	beliebig lange Folge von Worten aus $L(F)$
$F\{m,n\}$	Folge mit mindestens $m$ und höchstens $n$ von Worten aus $L(F)$
$F\{m\}$	Folge mit genau $m$ Worten aus $L(F)$
$[abc]$	alternativ ein Zeichen aus der Klammer
$[^abc]$	alternativ ein anderes Zeichen als die in der Klammer
$[a-zA-Z]$	alternativ ein Zeichen aus Zeichenbereichen
$.$	beliebiges Zeichen
$^$	Anfang der Zeichenfolge (nichts darf vorangehen)
$\$$	Ende der Zeichenfolge (nichts darf darauf folgen)

## Beispiele für reguläre Ausdrücke in PHP-Notation

Reguläre Ausdrücke kommen in PHP-Programmen immer als Zeichenreihenliterale (Strings) vor. Diese werden in " eingeschlossen, z. B. "1|0".

Statt Namen für reguläre Ausdrücke zu definieren, weisen wir die Zeichenreihe einer Variablen zu, z. B. `$binDig = "(1|0)";`

**Variable = regulärer Ausdruck als PHP-String**

`$Abc = "(a|b)(c|d)?"`;

`$Anrede = "Sehr geehrte(r)? (Frau|Herr)";`

`$Dig = "[0-9]";`

`$sLet = "[a-z]";`

`$cLet = "[A-Z]";`

`$Let = "[a-zA-Z]";`

`$Bezeichner = "[a-zA-Z][a-zA-Z0-9]*";`

`$GeldBetrag = "$Dig+,$Dig{2}";`

`$KFZ = "$cLet{1,3}-$cLet{1,2}-$Dig{1,4}";`

`$Dual = "1{3}[10]*0{3}";`

# Beispiele für Definitionen von Grundsymbolen in PHP-Notation

```
$Pascal_Identifier = "[a-zA-Z][a-zA-Z0-9]*";
```

```
$C_Identifier = "[a-zA-Z_][a-zA-Z_0-9]*";
```

```
$ADA_Identifier = "[a-zA-Z](_[a-zA-Z0-9])*";
```

```
$PHP_Var_Identifier = "\$[a-zA-Z_][a-zA-Z_0-9]*";
```

```
$Pascal_Exponent = "((e|E)(\+|-)?[0-9]+)";
```

```
$Pascal_Real = "(([0-9]+\.[0-9]+)$Exponent?) | ([0-9]+$Exponent)";
```

```
$HexDig = "[0-9a-fA-F]";
```

```
$HTML_CharRef = "&(([a-zA-Z]+) | ([#][0-9]+) | (#x$HexDig+));";
```

## E2.2 Kontextfreie Grammatiken

**Kontextfreie Grammatik (KFG):** formaler Kalkül zur Definition einer

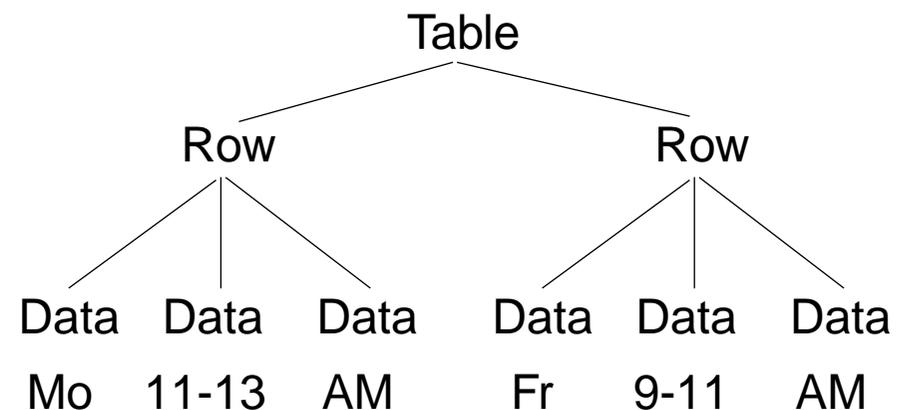
- **Sprache** als Menge von Sätzen; jeder **Satz** ist eine **Folge von Symbolen**
- **Menge von Bäumen**; jeder Baum repräsentiert die **Struktur eines Satzes** der Sprache

**Anwendungen:**

- Programme einer **Programmiersprache** und deren **Struktur**, z. B. Java, Pascal, C
- **Sprachen als Schnittstellen** zwischen Software-Werkzeugen, **Datenaustauschformate**, z. B. HTML, XML
- Bäume zur Repräsentation **strukturierter Daten**, z. B. in HTML

### Beispiel: Tabellen in HTML:

```
<table><tr><td>Mo</td>  
    <td>11-13</td>  
    <td>AM</td>  
</tr>  
<tr> <td>Fr</td>  
    <td>9-11</td>  
    <td>AM</td>  
</tr>  
</table>
```



# Definition: Kontextfreie Grammatik

Eine kontextfreie Grammatik  $G = (T, N, P, S)$  besteht aus:

- T **Menge der Terminalsymbole** (kurz: Terminale)
- N **Menge der Nichtterminalsymbole** (kurz: Nichtterminale)  
(T und N sind disjunkt)
- S **Startsymbol**; S ist ein Nichtterminal:  $S \in N$
- P **Menge der Produktionen**  
jede Produktion hat die Form  $A ::= x$   
A ist ein Nichtterminal, d. h.  $A \in N$  und  
x ist eine (evtl. leere) Folge von Terminalen und Nichtterminalen,  
d. h.  $x \in (T \cup N)^* = V^*$

$V = T \cup N$  heißt auch **Vokabular**, seine Elemente heißen **Symbole**

Man sagt

„In der Produktion  $A ::= x$  steht A auf der **linken Seite** und x auf der **rechten Seite**.“

Man gibt Produktionen häufig **Namen**: **p1**:  $A ::= x$

In Symbolfolgen aus  $V^*$  werden die Elemente nur durch Zwischenraum getrennt:  $A ::= B C D$

# Beispiel zur Definition einer KFG

**Terminale**  $T = \{ (, ) \}$   
**Nichtterminale**  $N = \{ \text{Klammern}, \text{Liste} \}$   
**Startsymbol**  $S = \text{Klammern}$   
**Produktionen**  $P =$   
 {  
 p1:  $\text{Klammern} ::= '( \text{Liste} )'$   
 p2:  $\text{Liste} ::= \text{Klammern Liste}$   
 p3:  $\text{Liste} ::=$   
 }

**Vokabular**  $V = T \cup N =$   
 $\{ (, ), \text{Klammern}, \text{Liste} \}$

Unbenannte Terminale werden in ' eingeschlossen, um Verwechslungen mit KFG-Zeichen zu vermeiden: '( '

**Namen**  $\cap N \cap V^*$

Diese Grammatik definiert eine Sprache, deren Sätze Folgen von geschachtelten Klammerpaaren sind, z. B.

$(( )) \quad (( ( )) ( ( ) ))$

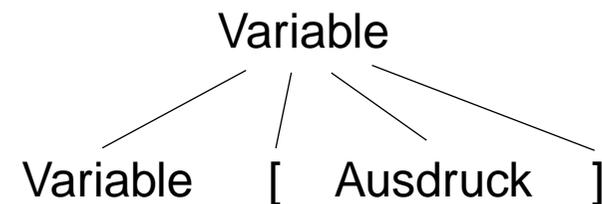
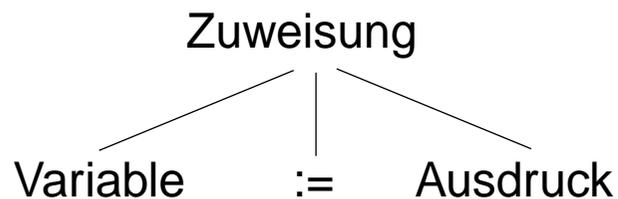
# Bedeutung der Produktionen

Eine Produktion  $A ::= x$  ist eine **Strukturregel**: A besteht aus x

## Beispiele:

	DeutscherSatz	::=	Subjekt Prädikat Objekt
<i>Ein</i>	DeutscherSatz	<i>besteht aus (der Folge)</i>	Subjekt Prädikat Objekt
	Klammern	::=	'(' Liste ')'
	Zuweisung	::=	Variable ':=' Ausdruck
	Variable	::=	Variable '[' Ausdruck ']'

**Produktion** graphisch dargestellt als gewurzelter **Baum**  
mit geordneten Kanten und mit Symbolen als Knotenmarken  
(zum Begriff „Baum“ siehe nächste Folie):

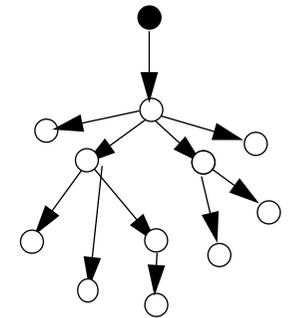
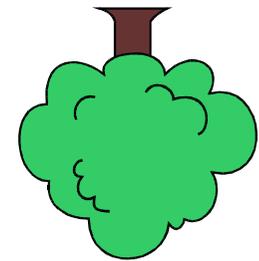
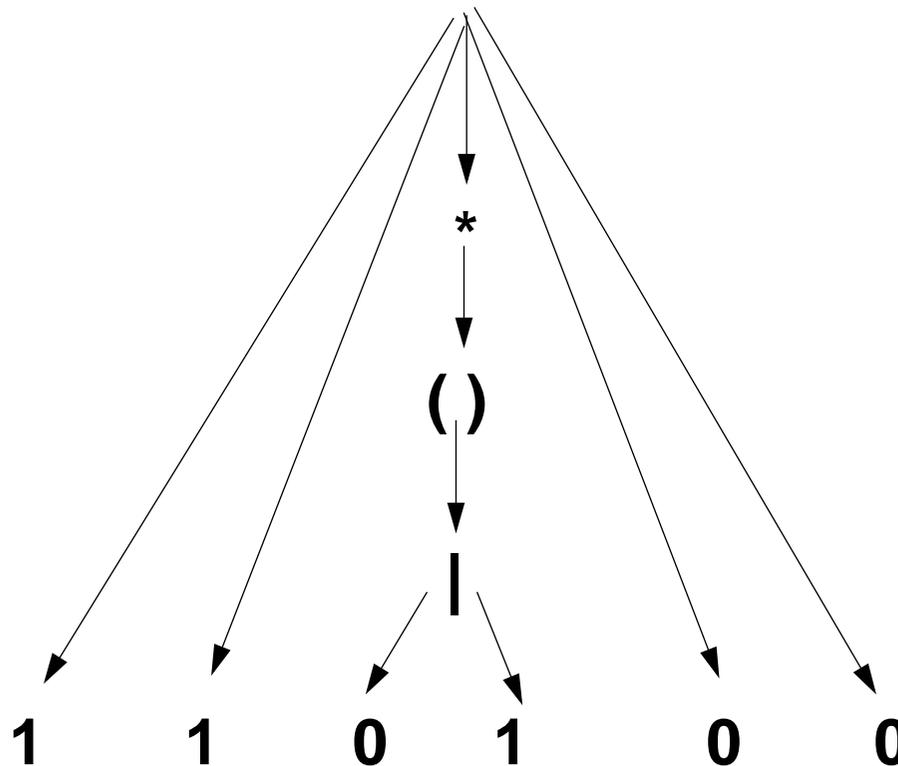


# Bäume als Abstraktion

**Bäume** bezeichnen in der Informatik **abstrakte Datenstrukturen** mit speziellen Eigenschaften.

Sie werden zur abstrakten Beschreibung bestimmter Zusammenhänge eingesetzt. Ein besonders wichtiger ist die „**besteht-aus**“-Beziehung zwischen einem Objekt und seinen Teilen.

Z. B. ein regulärer Ausdruck besteht aus Teilausdrücken:



# Begriffe zu Bäumen

**Definition:** Ein (gewurzelter, gerichteter) **Baum** besteht aus

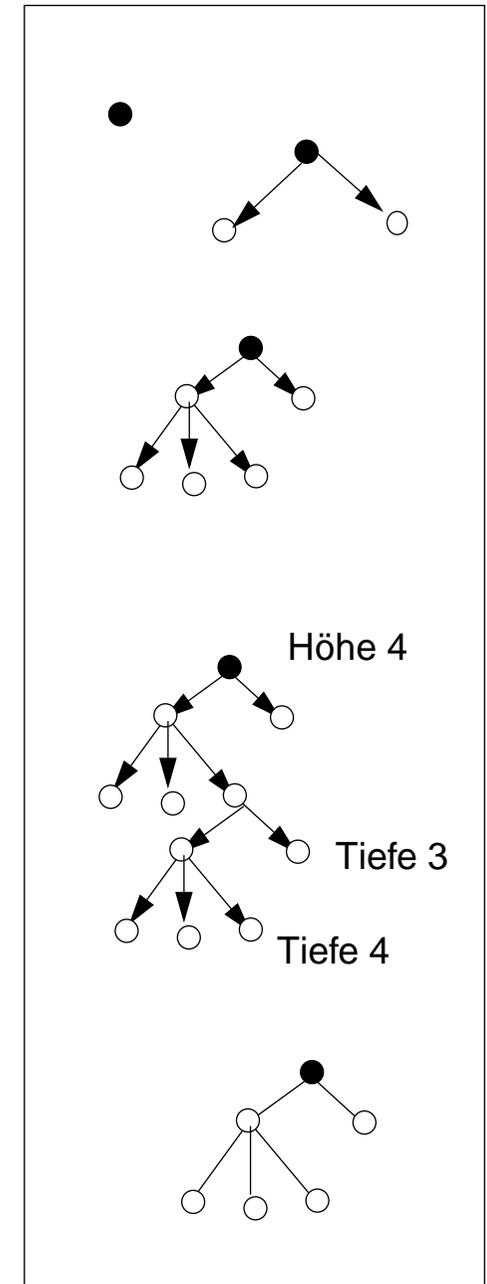
- einem **Knoten**  $k$  und
- einer (evtl. leeren) **Folge von Bäumen** und **Kanten** von  $k$  zu jedem Element der Folge.
- In **einen** Knoten mündet keine Kante, in alle anderen genau eine.

Begriffe und Eigenschaften:

- **Wurzel:** Knoten in den keine Kante mündet.
- **Blätter:** Knoten, von denen keine Kante ausgeht
- **innere Knoten:** es mündet eine Kante und es gehen welche aus
- Es gibt genau eine Wurzel.
- Wenn ein Baum  $n$  Knoten hat, dann hat er  $n-1$  Kanten.
- **Tiefe eines Blattes:** Anzahl der Kanten auf dem Weg von der Wurzel zu dem Blatt.
- **Höhe des Baumes:** größte Tiefe aller seiner Blätter.

Knoten und/oder Kanten können **beschriftet** werden.

Man kann die Pfeilspitzen weglassen, wenn die Wurzel bekannt ist.



# Ableitungen

Produktionen sind **Ersetzungsregeln**: Ein Nichtterminal  $A$  in einer Symbolfolge  $u A v$  kann durch die rechte Seite  $x$  einer Produktion  $A ::= x$  ersetzt werden.

Das ist ein **Ableitungsschritt**; er wird notiert als  $u A v \Rightarrow u x v$

z. B. Klammern **Klammern** Liste  $\Rightarrow$  Klammern ( Liste ) Liste  
mit Produktion p1: **Klammern** ::= ( Liste )

Beliebig viele Ableitungsschritte nacheinander angewandt heißen **Ableitung**; notiert als  $u \Rightarrow^* v$

Eine kontextfreie Grammatik **definiert eine Sprache**; das ist eine **Menge von Sätzen**.

**Jeder Satz ist eine Folge von Terminalsymbolen, die aus dem Startsymbol ableitbar ist:**

$$L(G) = \{ w \mid w \in T^* \text{ und } S \Rightarrow^* w \}$$

Grammatik auf EWS-3.13 definiert geschachtelte Folgen paariger Klammern als Sprachmenge:

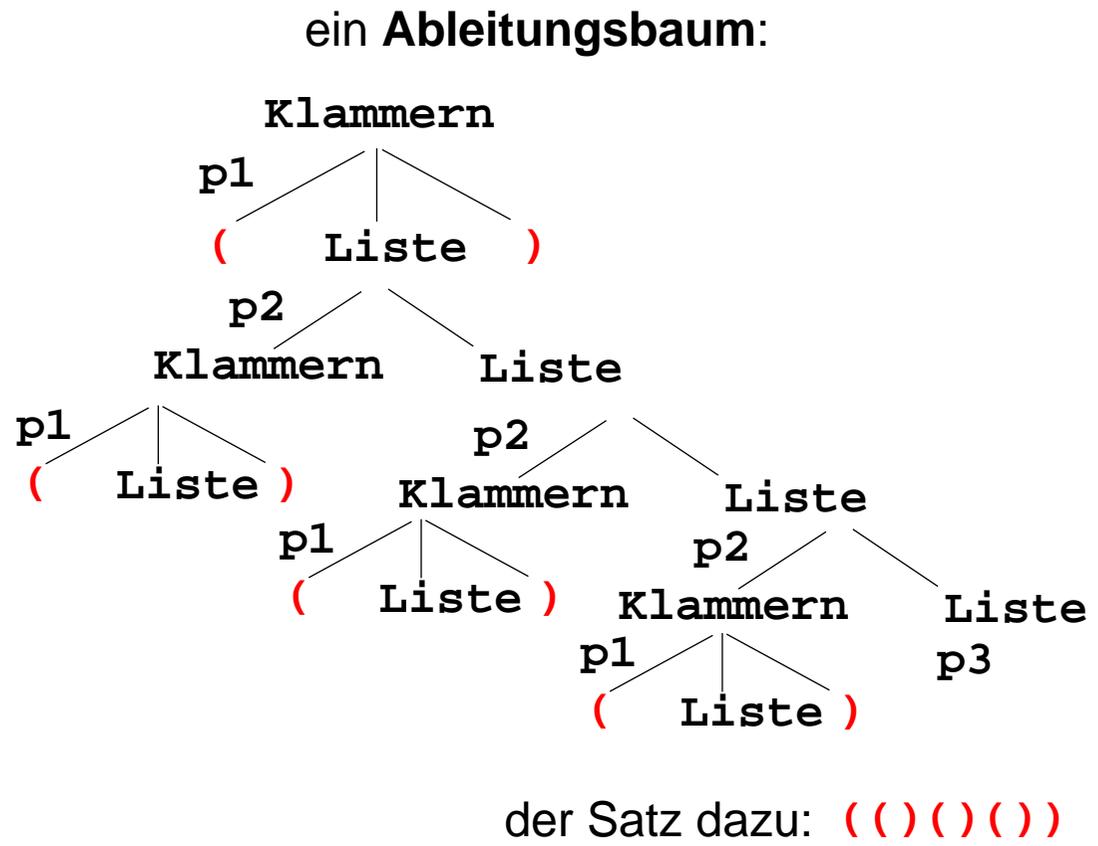
$$\{ (), ( ( ) ), ( ( ) ( ) ( ) ), ( ( ( ) ( ) ) ( ( ) ) ), \dots \} \subseteq L(G)$$

Ableitung des Satzes $((()()))$ :	<b>S</b>	= <b>Klammern</b>	p1
		$\Rightarrow$ ( <b>Liste</b> )	p2
		$\Rightarrow$ ( Klammern <b>Liste</b> )	p2
		$\Rightarrow$ ( Klammern <b>Klammern</b> Liste )	p1
		$\Rightarrow$ ( <b>Klammern</b> ( Liste ) Liste )	p1
		$\Rightarrow$ ( ( <b>Liste</b> ) ( Liste ) Liste )	p3
		$\Rightarrow$ ( ( ) ( <b>Liste</b> ) Liste )	p3
		$\Rightarrow$ ( ( ) ( ) <b>Liste</b> )	p3
		$\Rightarrow$ ( ( ) ( ) )	



# Satz zum Ableitungsbaum

Man erhält den **Satz** aus einem **Ableitungsbaum**, wenn man seine **Terminale** in der Reihenfolge eines **links-abwärts Durchlaufes** aufschreibt.



### Ein links-abwärts Durchlauf

- **beginnt** mit einem Besuch der **Wurzel**,
- bei dem Besuch eines Knotens werden nacheinander für jeden seiner **Unterbäume von links nach rechts** jeweils ein **links-abwärts Durchlauf** durchgeführt.

# Grammatik für arithmetische Ausdrücke

## Grammatik für arithmetische Ausdrücke

### Produktionen:

p1: Expr ::= Expr AddOpr Fact  
p2: Expr ::= Fact  
p3: Fact ::= Fact MulOpr Opd  
p4: Fact ::= Opd  
p5: Opd ::= '(' Expr ')'  
p6: Opd ::= Ident  
p7: AddOpr ::= '+'  
p8: AddOpr ::= '-'  
p9: MulOpr ::= '\*'  
p10: MulOpr ::= '/'

### Beispiele:

b + c

a \* (b + c)

a \* b + c

a + b \* c

$T = \{ (, ), +, -, *, /, \text{Ident} \}$

$N = \{ \text{Expr}, \text{Fact}, \text{Opd}, \text{AddOpr}, \text{MulOpr} \}$

$S = \text{Expr}$

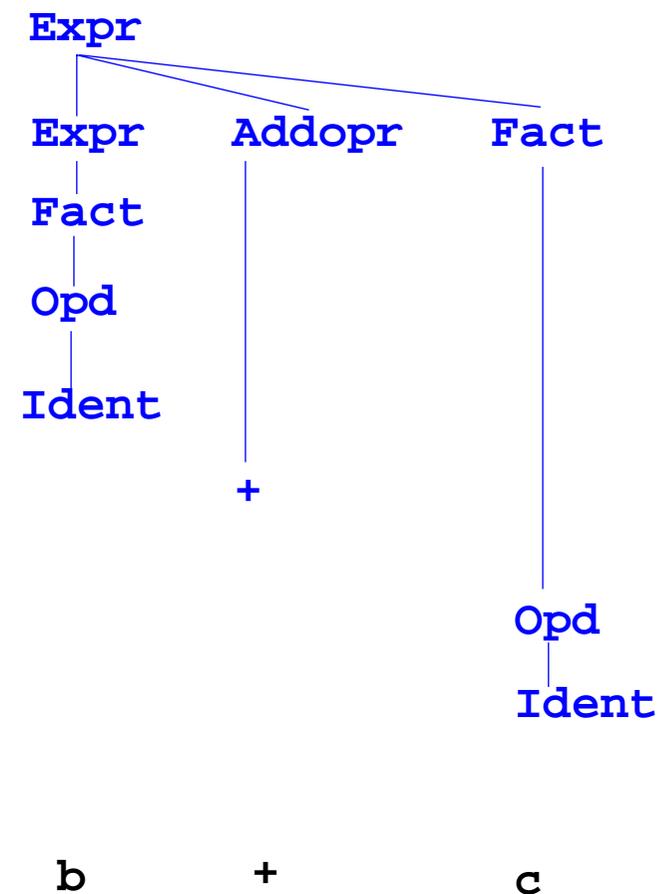
# Beispiel für eine Ableitung zur Ausdrucksgrammatik

Satz der Ausdrucksgrammatik  $b + c$

**Ableitung:**

	<b>Expr</b>			
p1	=> <b>Expr</b>	<b>Addopr</b>	<b>Fact</b>	
p2	=> <b>Fact</b>	<b>Addopr</b>	<b>Fact</b>	
p4	=> <b>Opd</b>	<b>Addopr</b>	<b>Fact</b>	
p6	=> <b>Ident</b>	<b>Addopr</b>	<b>Fact</b>	
p7	=> <b>Ident</b>	<b>+</b>	<b>Fact</b>	
p4	=> <b>Ident</b>	<b>+</b>	<b>Opd</b>	
p6	=> <b>Ident</b>	<b>+</b>	<b>Ident</b>	
	<b>b</b>	<b>+</b>	<b>c</b>	

**Ableitungsbaum:**



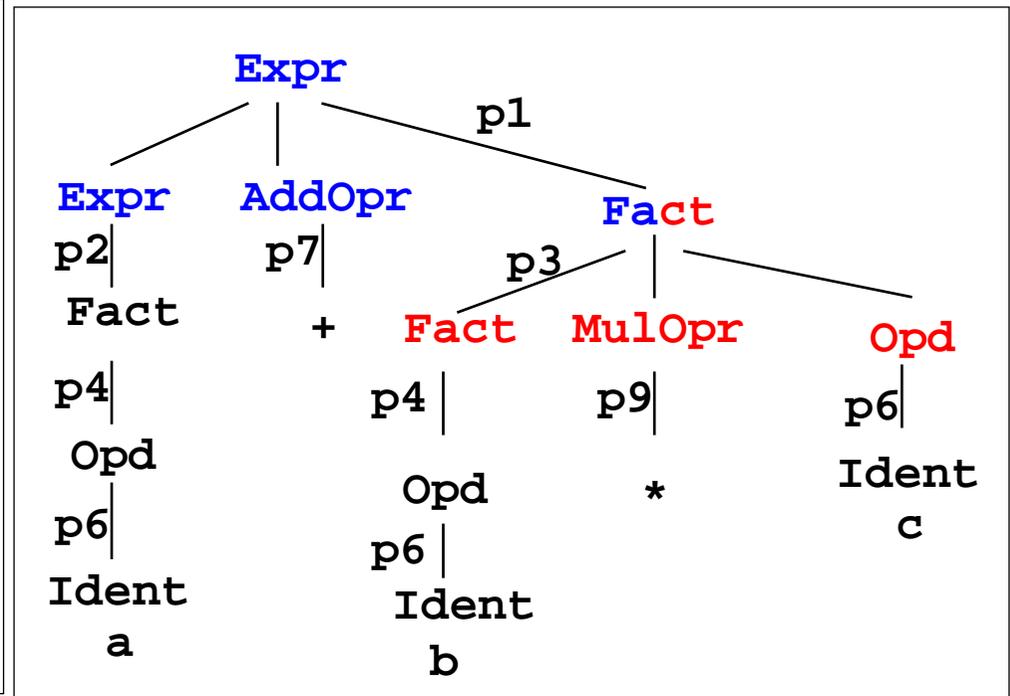
# Präzedenz und Assoziativität in Ausdrucksgrammatiken

Die Struktur eines Satzes wird durch seinen Ableitungsbaum bestimmt.

Ausdrucksgrammatiken legen dadurch die **Präzedenz** und **Assoziativität** von Operatoren fest.

Im Beispiel hat **AddOpr** geringere Präzedenz als **MulOpr**, weil er **höher in der Hierarchie der Kettenproduktionen**  $\text{Expr} ::= \text{Fact}$ ,  $\text{Fact} ::= \text{Opd}$  steht.

Name	Produktion
p1:	$\text{Expr} ::= \text{Expr AddOpr Fact}$
p2:	$\text{Expr} ::= \text{Fact}$
p3:	$\text{Fact} ::= \text{Fact MulOpr Opd}$
p4:	$\text{Fact} ::= \text{Opd}$
p5:	$\text{Opd} ::= '(' \text{Expr} ')'$
p6:	$\text{Opd} ::= \text{Ident}$
p7:	$\text{AddOpr} ::= '+'$
p8:	$\text{AddOpr} ::= '-'$
p9:	$\text{MulOpr} ::= '*'$
p10:	$\text{MulOpr} ::= '/'$



Im Beispiel sind **AddOpr** und **MulOpr** **links-assoziativ**, weil ihre **Produktionen links-rekursiv** sind, d. h.

$a + b - c$  entspricht  $(a + b) - c$ .

# Schemata für Ausdrucksgrammatiken

**Ausdrucksgrammatiken** konstruiert man **schematisch**,  
sodass **strukturelle Eigenschaften** der Ausdrücke definiert werden:

**eine Präzedenzstufe**, binärer  
Operator, linksassoziativ:

$$A ::= A \text{ Opr } B$$

$$A ::= B$$

eine Präzedenzstufe, binärer  
Operator, **rechtsassoziativ**:

$$A ::= B \text{ Opr } A$$

$$A ::= B$$

eine Präzedenzstufe,  
**unärer Operator**, präfix:

$$A ::= \text{Opr } A$$

$$A ::= B$$

eine Präzedenzstufe,  
unärer Operator, **postfix**:

$$A ::= A \text{ Opr}$$

$$A ::= B$$

**Elementare Operanden**: nur aus  
dem Nichtterminal der höchsten  
Präzedenzstufe (sei hier H)  
abgeleitet:

$$H ::= \text{Ident}$$

**Geklammerte Ausdrücke**: nur aus  
dem Nichtterminal der höchsten  
Präzedenzstufe (sei hier H) abgeleitet;  
enthalten das Nichtterminal der  
niedrigsten Präzedenzstufe (sei hier A)

$$H ::= '( A )'$$

# Nützliche Grammatik-Konstrukte

**beliebig lange Folgen** von `stmt` z. B. in:

```
Block ::= '{' stmts '}'
```

```
stmts ::= stmts stmt
```

```
stmts ::=
```

**nicht leere Folgen** von `stmt`:

```
stmts ::= stmts stmt
```

```
stmts ::= stmt
```

nicht-leere Folgen von `stmt`  
**getrennt durch ;:**

```
stmts ::= stmts ';' stmt
```

```
stmts ::= stmt
```

**Optionale Parameter** z. B. in:

```
Call ::= Name '(' ParamOpt ')'
```

```
ParamOpt ::= Parameter
```

```
ParamOpt ::=
```

Abkürzung für **beliebig lange Folgen**:

```
Block ::= '{' stmt* '}'
```

Die 2 Produktionen für `stmts` entfallen.

Entsprechend für **nicht-leere Folgen**:

```
Block ::= '{' stmt+ '}'
```

**Alternative Produktionen** für dasselbe  
Nichtterminal mit | zusammenfassen, z. B.

```
stmts ::= stmts ';' stmt |  
        stmt
```

**Optionales** mit [ ] klammern, z. B.

```
Call ::= Name '(' [ Parameter ] ')'
```

Die 2 Produktionen für `ParamOpt` entfallen.

# Ausschnitte aus einer HTML-Grammatik

Die **Syntax von HTML** ist im Kalkül der „Document Type Definition (**DTD**)“ formal definiert. Man kann **Ausschnitte zu einigen Aspekten** von HTML in **KFGn** übertragen. Es folgen Beispiele dafür.

## Grundstruktur (ohne Attribute innerhalb von Tags):

```
HTMLDoc ::= '<html>' '<head>' HeadContent '</head>'
          '<body>' Block '</body>'
          '</html>'
```

```
Block ::= Paragraph | Table | List | Heading | ...
```

```
Paragraph ::= '<p>' Inline ['</p>']
```

```
Inline ::= ... Fließtext mit Auszeichnungen ohne Blockstrukturen ...
```

```
Flow ::= Block | Inline
```

```
Tabellen: Table ::= '<table>' Row* '</table>'
```

```
Row ::= '<tr>' Cell* '</tr>'
```

```
Cell ::= '<td>' Flow '</td>'
```

# HTML-Grammatik: Listen und Attribute

## Listen:

```
List          ::= '<ol>' ListElement+ '</ol>' |  
              '<ul>' ListElement+ '</ul>'  
  
ListElement   ::= '<li>' Flow '</li>'
```

## Attribute in Anfangs-Tags.

In dieser Grammatik werden Tags weiter zerlegt:

```
AnfangsTag    ::= '<' TagName Attribute* '>'  
  
Attribute     ::= AttributeName '=' AttributeValue  
  
AttributeName ::= Identifier  
  
AttributeValue ::= StringLiteral | Identifier | Number | ...
```

# S3. PHP

## S3.1 Einleitung

### Entstehung und Einsatzziele:

- ursprünglich 1994 von Rasmus Lerdorf entwickelt; PHP: **P**ersonal **H**ome **P**age **T**ools
- **Skriptsprache, vieles übernommen von Perl**, einiges vereinfacht
- PHP-Programme werden **eingebettet** in Texte anderer Sprachen - meist HTML
- wichtigster Einsatz: Programme auf Web-Servern (siehe Beispiel Telefonbuch)
- **Interpretation mit interner Analyse** der Programmfragmente vor der Ausführung

# Charakteristika der Sprache PHP

- Notation an C und Perl orientiert
- **einfache Programmstrukturen**
- einfache Regeln zur **statischen Bindung von Bezeichnern**
- wenige, einfache Typen, **dynamisch typisiert**
- wichtigste **Operationen auf Zeichenreihen, Zahlwerten und Arrays**
- Definition und Aufruf **parametrisierter Funktionen**
- sehr große Menge **vordefinierter nützlicher Funktionen**
- **Klassen und Objekte** zur übersichtlichen Formulierung von komplexeren Strukturen

# Ausführung von PHP-Programmen

Ein PHP-Interpreter verarbeitet jeweils eine Datei, in die ein **PHP-Programm eingebettet** ist. Es kann aus **mehreren Stücken** bestehen. Sie werden jeweils vom umgebenden Text **abgegrenzt** durch:

```
<?php ... PHP-Programmstück ... ?>
```

auch mehrzeilige Programmstücke, auch mehrere Stücke:

## eingebettete Programmstücke:

Warnung auf Englisch:

```
<?php  
    echo "Beware of the dog!"  
?>
```

Warnung auf Deutsch:

```
<?php  
    echo "Bissiger Hund!"  
?>  
Aufpassen!
```

## Ergebnis der Ausführung:

Warnung auf Englisch:

**Beware of the dog!**

Warnung auf Deutsch:

**Bissiger Hund!**

Aufpassen!

Die umgebenden Texte werden übernommen; an den Stellen der **Programmstücke** wird deren **Ausgabe** eingesetzt.

## PHP-Interpreter

- kann **direkt mit der Programmdatei aufgerufen** werden,
- ist **auf Web-Server** installiert, verarbeitet eine Seite beim Anfordern der URL

# Ein zweites Beispiel als Eindruck von PHP

## PHP-Programm:

```
<?php
// ein Dreieck aus *-Zeichen
$line = 1;
while ($line < 16) {
    $col = 1;
    while ($col <= $line) {
        echo "*";
        $col = $col + 1;
    }
    echo "\n";
    $line = $line + 1;
}
?>
```

## Ausgabe dazu:

```
*
**
***
****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

## S3.2 Variable und Zuweisungen

Eine **Variable speichert Werte** während der **Ausführung** eines Programmes.

Eine **Variable** wird präzise beschrieben durch 3 Dinge

- **Name** im Programm
- **Speicherstelle** bei der Ausführung des Programmes
- **Wert** an der Speicherstelle zu einem bestimmten Zeitpunkt der Programmausführung

`$line`

42

### Notation von Variablennamen im Programm:

\$ als Kennzeichen einer Variablen - zur Unterscheidung von Funktionen und Typen - gefolgt von einem Namen der Form `[a-zA-Z_][a-zA-Z_0-9]*`

- Eine Variable kann **Werte beliebiger Typen** aufnehmen.
- Der Wert einer Variablen wird durch Ausführen von Zuweisungen verändert (s. 4.6)  
`$line = 1;`
- In Ausdrücken werden Werte von Variablen für Berechnungen benutzt.  
`$line < 16`
- Variable brauchen in PHP nicht deklariert zu werden

# Zuweisungen

Eine Zuweisung hat die Form:

**Variable = Ausdruck;**

z. B. **\$line = 1;**

Eine **Zuweisung wird ausgeführt** durch folgende Schritte

- die **Speicherstelle** der Variablen wird bestimmt,
- der Ausdruck wird ausgewertet und liefert einen **Wert**,
- der **Wert wird** an der Speicherstelle der Variablen **gespeichert** (ersetzt den Wert, der bisher dort stand).

Auf der **linken Seite einer Zuweisung** bezeichnet die Variable die **Speicherstelle**, an die zugewiesen wird.

**\$line = 1;**

In einem **Ausdruck** steht die Variable für den **Wert**, den ihre Speicherstelle gerade enthält. **\$line < 16**

Ausführung einer Folge von Zuweisungen

Werte im Speicher an der Stelle von **a** **b**

**\$a = 1;**

**1** **null**

**\$b = 7;**

**1** **7**

**\$a = \$b + 2;**

**9** **7**

**\$b = \$b + 1;**

**9** **8**

**\$a = \$a \* 2;**

**18** **8**

## S3.3 Ausdrücke

Die **Auswertung eines Ausdruckes** liefert einen **Wert eines bestimmten Typs**.

`$anzahl + 5`                      `$i * 3 <= 100`

### Elementare Ausdrücke:

- **Literal:** Notation gibt den Wert an, z. B.                      `42`    `"Hello"`
- **Variable:** hat einen Wert, z. B.                                      `$anzahl`
- **Aufruf einer Funktion:** liefert einen Wert als Ergebnis, z. B.    `abs($konto)`

### Ausdrücke mit **Operatoren**

- **2-stellig** mit 2 Operanden (Ausdrücke), z. B.                      `$anzahl + 5`    `$zeile . "*"`
- **1-stellig** mit 1 Operanden (Ausdruck), z. B.                      `! $gefunden`

Der Operator verknüpft die Werte der Operanden zum Wert des Ausdruckes.

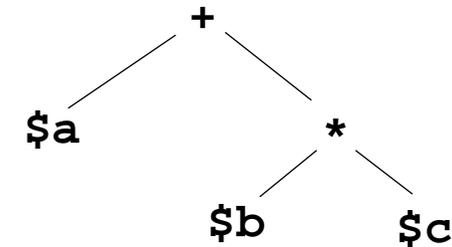
# Präzedenz und Assoziativität von Operatoren

Ein Operator mit höherer Präzedenz bindet seine Operanden stärker als ein Operator mit niedrigerer Präzedenz.

z. B. \* hat höhere Präzedenz als +; deshalb hat der Ausdruck  $\$a + \$b * \$c$  die Struktur:

Wenn man trotzdem erst  $\$a + \$b$  verknüpfen wollte, müsste man dies durch Klammerung ausdrücken:

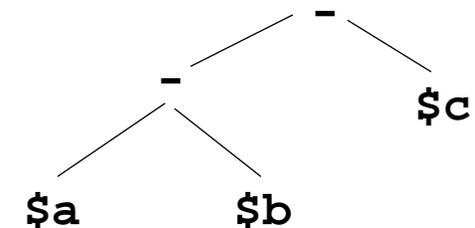
$$(\$a + \$b) * \$c$$



Ein Operator ist linksassoziativ, wenn beim Aufeinandertreffen von Operatoren gleicher Präzedenz der linke seine Operanden stärker bindet als der rechte.

(Entsprechendes gilt für rechtsassoziativ.)

Beispiel: - ist linksassoziativ; deshalb hat  $\$a - \$b - \$c$  die Struktur:



# Präzedenz und Assoziativität aller Operatoren in PHP

Präzedenz	Assoziativität	Operatoren
1	left	,
2	left	or
3	left	xor
4	left	and
5	right	print
6	right	= += -= *= /= .= &=  = ^= <<= >>=
7	left	?:
8	left	
9	left	&&
10	left	
11	left	^
12	left	&
13	non-ass.	== != === !==
14	non-ass.	< <= > >=
15	left	<< >>
16	left	+ - .
17	left	* / %
18	right	! ~ ++ -- @ (int) (float) (string) ...
19	right	[
20	non-ass.	new

# Arithmetische Datentypen und Operationen

<b>Datentypen:</b>	<code>int</code>	ganze Zahlen	
	<code>float</code>	Gleitpunktzahlen, d. h. reelle Zahlen mit begrenzter Genauigkeit	
<b>int-Literale:</b>	<code>-127</code>	<b>dezimal;</b>	Schreibweise: $[+-]?([1-9][0-9]^* 0)$
	<code>064</code>	<b>oktal</b>	$[+-]?(0[0-7]^+)$
	<code>0x1A</code>	<b>hexadezimal</b>	$[+-]?(0[xX][0-9a-fA-F]^+)$
<b>float-Literale:</b>	<code>1.234</code>		
	<code>1.2e3</code>	Wert ist $1.2 * 10^3 = 1200$	
	<code>7E-3</code>	Wert ist $7 * 10^{-3} = 0.007$	

**Operatoren:**    `+`   `-`   `*`   `/`   `%`

`%` bedeutet **modulo**, d. h.  $\$a \% \$b$  ergibt den Rest der Division von  $\$a$  durch  $\$b$ .  
z. B.  $37 \% 10$  ergibt 7.

Verknüpfung **zweier int-Werte** liefert einen `int`-Wert, z. B.  $7 * 5$  liefert 35;  
Ausnahmen:        `/` liefert immer einen `float`-Wert,  
wenn der Wert nicht mehr als `int`-Wert darstellbar ist

Ist mindestens **ein Operand ein float-Wert**, so ist das Ergebnis ein `float`-Wert,  
z. B.  $2.5 * 2$  ergibt (etwa) 5.0

# Datentyp `string` für Zeichenreihen

Ein Wert vom Typ `string` ist eine beliebig lange Folge von Zeichen des ASCII-Zeichensatzes, auch **Zeichenreihe** genannt.

## 3 Notationen für **Literale**:

- mit `'` geklammert:  
Alle Zeichen stehen für sich selbst; nur für `'` muss `\'` geschrieben werden, z. B.  
`'Hello World!'`      `'Mary\'s car'`

- mit `"` geklammert:  
Es können darin Umschreibungen und Variable vorkommen, z. B.  
`"Summe $jahr = \t${betrag}EUR"`

<code>\$jahr</code>	Variable, ihr Wert wird eingesetzt
<code>\${betrag}</code>	ebenso; zur Abgrenzung des Namens vom umgebenden Text
<code>\n</code>	Zeilenwechsel
<code>\"</code>	<code>"</code> und viele mehr ...

- „Heredoc“ für nicht-leere Folgen von Zeilen als Zeichenreihe; Umschreibungen wie bei `"`  

```
print <<<EOS
  Frohe
  Weihnachten
EOS;
```

**EOS** ist ein **frei wählbarer Bezeichner**; `<<<EOS` steht am Zeilenende, das schließende `EOS` am Zeilenanfang, danach höchstens ein `;`

# Operationen mit Zeichenreihen

**Konkatenationsoperator** `.` verknüpft zwei Zeichenreihen zu einer neuen, z. B.

```
"Hello " . "world!"      $Sterne = $Sterne . "*" ;
```

Wenn die Variable `$str` einen `string`-Wert hat,  
**indiziert** `$str{$i}` das **(i+1)-te Zeichen** darin, z. B.

```
$Sterne{3} = "!";
```

## Ausgabe von Zeichenreihen:

**echo** Folge von Ausdrücken (durch Kommata getrennt),  
deren Auswertung Zeichenreihen liefert;

**print** Ausdruck, dessen Auswertung eine Zeichenreihe liefert;

```
echo $Sonne, $Mond, $Sterne;
```

```
print "Summe $jahr =\t${betrag}EUR";
```

# Datentyp `bool` für logische Operationen

Der **Datentyp** `bool` (oder auch `boolean`) hat die beiden **Literale** `true` und `false` (geschrieben mit Groß- oder Kleinbuchstaben).

Logische Werte werden insbesondere für Bedingungen in Schleifen und bedingten Anweisungen benötigt, z. B.

```
if ($alter < 14 or $alter > 65) print "Rabatt";
```

## Logische Operatoren:

`$a and $b` `true`, falls beide `$a` und `$b` `true` sind

`$a && $b` ebenso

`$a or $b` `true`, falls `$a` oder `$b` oder beide `true` sind

`$a || $b` ebenso

`$a xor $b` `true`, falls genau einer von `$a` und `$b` `true` ist

`!$a` `true`, falls `$a` `false` ist

## Vergleichsoperatoren liefern Werte vom Typ `bool`:

<code>==</code>	gleich	<code>&lt;</code>	kleiner
<code>!=</code>	ungleich	<code>&gt;</code>	größer
<code>&lt;&gt;</code>	ungleich	<code>&lt;=</code>	kleiner oder gleich
		<code>&gt;=</code>	größer oder gleich

# Konversion: Umwandlung von Werten

## Konversion:

Ein **Wert eines Typs** wird in einen „entsprechenden“ **Wert eines anderen Typs** umgewandelt.

- **explizite Konversion (engl. type cast):**

Der **Zieltyp**, in den der Wert eines Ausdruckes umgewandelt werden soll, wird **explizit angegeben**, z. B.

```
(string)(5+1) liefert die Zeichenreihe "6"
```

- **implizite Konversion (engl. coercion):**

Wenn der Typ eines Wertes nicht zu der darauf angewandten Operation passt, wird **versucht, ihn anzupassen**, z. B.

```
$sum = 42; print "Summe = " . $sum;
```

Die ganze Zahl 42 wird in die Zeichenreihe "42" konvertiert.  
(Der Wert der Variablen `$sum` bleibt unverändert.)

In PHP kommt man mit impliziter Konversion weitgehend aus.

# Verfügbare Konversionen

`int -> float` ganze Zahl in Gleitpunktzahl mit Exponent,  
ggf. Genauigkeitsverlust:  
`2000000001*2000000001` liefert `4.000000004E+18`

`float -> int` abrunden (Richtung 0): `(int)-3.6` liefert `-3`

`int -> string` Wert als Zeichenreihe: `echo 42, 3.2;`  
`float -> string`

`string -> int` aus dem Anfang der Zeichenreihe wird versucht,  
`string -> float` einen Zahlwert zu lesen: `1 + "10.5"` liefert `11.5`  
`1 + "Meier8"` liefert `1`

`bool -> int` `false -> 0`, und `true -> 1`  
`int -> bool` `0 -> false`, alle anderen Werte `-> true`

weitere durch Zusammensetzen

## S3.4 Ablaufstrukturen

Ausführbare Programmteile werden **aus Anweisungen zusammengesetzt**. Durch **Bedingungen** und **Verzweigungen** können je nach dem Ergebnis von Berechnungen **unterschiedliche Abläufe** durch die Programmstruktur ausgeführt werden.

Zu folgenden **algorithmischen Grundelementen** gibt es in jeder imperativen Programmiersprache jeweils einige Anweisungsformen:

Beispiele aus PHP

- **Zuweisung** `$st = $st . "*" ;`
- **Anweisungsfolge** `{ $st = $st . "*" ; $i = $i+1 ; }`
- **Alternativen** `if ( $i > 100 ) echo "zu groß" ; else echo "ok" ;`
- **Schleife** `while ( $i < 20 ) { $st = $st . "*" ; $i = $i+1 ; }`
- **Funktionsaufruf** `fclose ( $out ) ;`

Meist gibt es

- mehrere Formen für Schleifen und Alternativen,
- unterschiedliche Notationen der Anweisungen und
- Anweisungsformen für weitere Ablaufstrukturen.

# Grammatik für elementare Anweisungsformen in PHP

Statement

```
 ::= Variable '=' Expression ';'
 | '{' Statement* '}'
 | 'if' '(' Expression ')' Statement [ 'else' Statement ]
 | 'while' '(' Expression ')' Statement
 | FunctExpr '(' [ Parameters ] ')' ';'

```

FunctExpr ::= FunctName | ...

Parameters ::= Parameters ',' Expression | Expression

# Bedingte Anweisung

einseitig: `if ( Bedingung ) Then-Teil`

zweiseitig: `if ( Bedingung ) Then-Teil else Else-Teil`

Die **Bedingung** wird ausgewertet. Wenn sie `true` liefert, wird der **Then-Teil** ausgeführt; liefert sie `false`, wird in der zweiseitigen Form der **Else-Teil**, in der einseitigen **nichts** ausgeführt.

## Beispiele:

```
if ($a < 0) { $a = -$a; }
```

```
if ($a > $b) {  
    echo "a ist größer als b";  
} else {  
    echo "a ist nicht größer als b";  
}
```

## Stilregel:

Die Alternativen der bedingten Anweisung immer als **geklammerte Folge** schreiben - auch wenn es nur einzelne Anweisungen sind!

Verzweigung über mehrere Bedingungen in verschiedene Fälle, `if`-Kaskade:

```
if      ($jahr % 4 != 0)    { $tage = 365; }  
else if ($jahr % 100 != 0) { $tage = 366; }  
else if ($jahr % 400 != 0) { $tage = 365; }  
else      { $tage = 366; }
```

# Iterative Berechnungen mit `while`-Schleifen

`while` ( Bedingung ) Schleifenrumpf

Die **Bedingung** wird **ausgewertet**; wenn sie `true` liefert, wird der **Schleifenrumpf ausgeführt** und dann die Bedingung erneut geprüft. Erst wenn sie `false` liefert, wird die **Iteration beendet**.

## Beispiel:

```
$s = 0;
while ($s < $n) {
    echo "*";
    $s = $s + 1;
}
echo "\n";
```

## Überlegungen zum Entwurf der Schleife:

`$s` gibt an, wieviele Sterne schon ausgegeben wurden.

Wenn  $0 \leq n$  gilt, dann ist immer  $s \leq n$ .

Nach der Schleife gilt  $s \leq n$  und  $s \geq n$   
also  $s == n$

Also wurden  $n$  Sterne ausgegeben.

Jede Ausführung des Schleifenrumpfes **ändert Variable in der Bedingung**.

Die Bedingung muss irgendwann `false` liefern - sonst **terminiert die Schleife** nicht.

Hinter der Schleife gilt die **Negation der Bedingung**; hier  $!(s < n)$  also  $(s \geq n)$

# Anwendungsmuster: Iterationen zählen

Eine **Variable** zählt die Ausführungen des Schleifenrumpfes mit.

Varianten:

- aufwärts oder abwärts zählen,
- versetzt zählen, mit Startwert  $\neq 0$ ,
- mit Schrittweite  $\neq 1$  zählen
- auch als `for`-Schleife formulierbar

**Beispiel:** Sterne ausgeben auf der vorigen Folie.

**Beispiel:** Celsius in Fahrenheit umrechnen:

```
echo "\tC \tF\n";
```

```
$c = 10;
```

```
while ($c <= 20) {
```

```
    echo "\t" . $c . "\t" .
```

```
        round ($c*9.0/5 + 32) . "\n";
```

```
    $c = $c + 1;
```

```
}
```

C	F
10	50
11	52
12	54
13	55
14	57
15	59
16	61
17	63
18	64
19	66
20	68

# Anwendungsmuster: Iterieren bis Zielbedingung gilt

Das Ziel ist es, dass nach **Abschluss der Iteration** eine bestimmte **Zielbedingung** gilt.

Wir zerlegen die Zielbedingung logisch in zwei Teile: **Inv and Halt**, sodass **Inv** vor, während und nach der Schleife gilt; die Negation von **Halt** wird zur Schleifenbedingung.

Dann entwerfen wir den Schleifenrumpf.

**Beispiel:** Dualdarstellung einer Zahl berechnen.

Methode: Iterativ durch 2 dividieren und die Reste aufschreiben.

**Zielbedingung:** `$zahl == 0` zerlegt in `$zahl >= 0 and $zahl <= 0`  
`$zahl >= 0` gilt unverändert; `!($zahl <= 0)` wird Schleifenbedingung

```
$zahl = 42;
$dual = "";
echo "$zahl dual ist ";
while ($zahl > 0) {
    $dual = (int)($zahl%2) . $dual;
    $zahl = (int)($zahl/2);
}
echo "$dual\n";
```

# Anwendungsmuster: Suchschleife

Die **Elemente einer Folge** werden durchsucht, bis das **Gesuchte gefunden** ist oder **alle Elemente vergeblich untersucht** wurden.

Die Schleife hat **2 verschiedenartige Ergebnisse**:

gefunden oder Folge ist durchsucht

Sie müssen nach der Schleife unterschiedlich behandelt werden.

Die **Schleifenbedingung** lautet:

nicht gefunden und Folge ist noch nicht durchsucht

## Beispiel:

Position des ersten  
Auftreten eines bestimmten  
Zeichens in einer  
Zeichenreihe suchen.

```
$str = "Ein # und noch ein #";  
$lg = strlen($str); $zeichen = "#";  
$i = 0; $gefunden = false;  
  
while (!$gefunden and $i<$lg) {  
    if ($str{$i} == $zeichen)  
        { $gefunden = true; }  
    else { $i = $i + 1;}  
}  
  
if ($gefunden)  
    { echo "$zeichen an Position $i\n";}  
else { echo "$zeichen kommt nicht vor\n";}
```

# Funktionsaufrufe

Eine **Funktion definiert eine Berechnung**, die mit bestimmten (formalen) **Parametern** ausgeführt werden kann. Die Berechnung kann ein **Ergebnis** liefern.

Der **Aufruf einer Funktion** führt die definierte Berechnung aus mit den (aktuellen) **Parameterwerten**, die beim Aufruf angegeben sind.

## Beispiel:

Die Funktion `strlen` berechnet die Länge einer Zeichenreihe, die als Parameter angegeben wird. Der Aufruf `strlen("abc")` liefert als Ergebnis 3

Funktionen, die ein Ergebnis liefern, können in Ausdrücken aufgerufen werden:

```
$lg = strlen ("abc");      $p = strpos ($str, $zeichen);
```

Aufrufe haben die **Form**: `FuncExpr ( [ Parameters ] )`

Ein **Aufruf** wird in folgenden **Schritten** ausgewertet:

1. **FuncExpr auswerten** (ist meist nur ein Name) liefert eine Funktion.
2. **Parameter auswerten** und an die Funktion **übergeben**.
3. **Berechnung der Funktion** mit den übergebenen Parameterwerten ausführen; **Ergebnis** an die Aufrufstelle **zurückgeben**.

# Bibliothek von Funktionen zu PHP

Zu PHP gibt es eine sehr große **Bibliothek mit zahlreichen nützlichen Funktionen** zu vielen Themen. Sie können in jedem PHP-Programm aufgerufen werden.

**Beschreibungen** der Funktionen findet man z. B. im PHP-Manual (siehe URL).

Einige der **Themen** sind:

## Arrays

Konfiguration und Protokolle

Datenbanken

Datum/Uhrzeit

Dateiverzeichnisse

## Dateien

Grafik

HTTP

IMAP

LDAP

## Mathematik

MCAL

Mcrypt

Mhash

PDF

POSIX

## Strings

Variablenmanipulation

XML

# string-Funktionen aus der Bibliothek

Beschreibung einer Bibliotheksfunktion:

<b>Name und Zweck</b> der Funktion:	<code>strtr</code> -- Tauscht bestimmte Zeichen aus
<b>Typen der Parameter</b> und des <b>Ergebnisses:</b>	<code>string strtr ( string str, string from, string to)</code>
Beschreibung der <b>Wirkung:</b>	Diese Funktion erzeugt aus <code>str</code> einen neuen String als Ergebnis, indem alle Vorkommen von Zeichen aus <code>from</code> in die entsprechenden Zeichen aus <code>to</code> umgesetzt werden. Sind <code>from</code> und <code>to</code> von unterschiedlicher Länge werden die überzähligen Zeichen ignoriert.
<b>Beispiel</b> für einen Aufruf:	<code>\$addr = strtr(\$addr, "äöü", "aou");</code>

Weitere `string`-Funktionen:

`str_replace` -- Ersetzt alle Vorkommen eines Strings in einem anderen String

`strcmp` -- lexikographischer Vergleich zweier Strings

`strpos` -- Sucht erstes Vorkommen des Suchstrings und liefert die Position

`strtolower` -- Setzt einen String in Kleinbuchstaben um

## S3.5 Ein- und Ausgabe mit Dateien

1. Eine **Datei speichert Daten** im Dateisystem des Rechners.
2. Dateien werden bei der **Ausführung von Programmen** geschrieben und gelesen.
3. Die Dateien sind **persistent**, d. h. sie **bleiben erhalten** - auch nach Ausführung des Programms, das sie geschrieben hat.
4. Der **Inhalt** einer Datei kann als eine (sehr lange) **Zeichenreihe** (`string`) aufgefasst werden. Zeichen für Zeilenwechsel **gliedern sie in Zeilen**.  
(Wir betrachten hier nur solche Dateien; es gibt auch andere: sog. Binär-Dateien, ihr Inhalt ist speziell strukturiert und codiert).
5. **Im Dateisystem** wird eine Datei durch den **Pfad** und den **Dateinamen** identifiziert, z. B. `/home/rasmus/file.txt`
6. **Im Programm** wird eine Datei durch einen **Dateizeiger** identifiziert. Er wird beim **Öffnen der Datei** erzeugt.
7. Der **Dateizeiger** gibt auch die **Position innerhalb der Datei** an, wo gerade gelesen oder geschrieben wird.

# Funktionen zum Öffnen und Schließen von Dateien

`fopen` -- Öffnet eine Datei oder URL

```
resource fopen (string filename, string mode...)
```

`mode` spezifiziert den gewünschten Zugriffstyp:

"r" zum Lesen vom Anfang der Datei

"w" zum Schreiben vom Anfang der Datei

"a" zum Weiterschreiben am Ende der Datei

```
$handle = fopen ("/home/rasmus/file.txt", "r");
```

`feof` -- Prüft, ob der Dateizeiger am Ende der Datei steht

```
bool feof (resource handle)
```

Gibt **TRUE** zurück, falls der Dateizeiger am Ende der Datei steht oder ein Fehler aufgetreten ist, andernfalls **FALSE**.

`fclose` -- Schließt einen offenen Dateizeiger

```
bool fclose (resource handle)
```

Die Datei, auf die `handle` zeigt, wird geschlossen.  
Gibt bei Erfolg **TRUE** zurück, im Fehlerfall **FALSE**.

## Schema: eine Ausgabedatei schreiben

`fputs` -- Schreibt Daten an die Position des Dateizeigers

```
int fputs (resource handle, string str [, int length])
```

`fputs` schreibt den Inhalt einer Zeichenkette `string` in die Datei, auf welche der Dateizeiger `handle` zeigt. Wenn der `length` Parameter gegeben ist, wird das Schreiben nach `length` Bytes beendet, oder wenn das Dateiende (EOF) erreicht ist, je nachdem, was eher eintritt.

`fputs` gibt bei Erfolg die Anzahl der geschriebenen Bytes zurück, andernfalls **FALSE**.

```
// ein dreieck aus *-Zeichen schreiben
$out = fopen ("Sterne.txt", "w");
if (!$out) { echo "Sterne.txt nicht geöffnet"; exit; }
$line = 0;
while ($line < 15) {
    $col = 0; $str = "";
    while ($col < $line) {
        $str = $str . "*"; $col = $col + 1;
    }
    $str = $str . "\n"; $line = $line + 1;
    fputs ($out, $str);
}
fclose ($out);
```

## Schema: Eingabedatei lesen

`fgets` -- Liest eine Zeile von der Position des Dateizeigers

```
string fgets (resource handle [, int length])
```

Gibt eine Zeile bis zu (length -1) Bytes Länge zurück, welche aus der Datei von der aktuellen Position des Dateizeigers `handle` aus gelesen wird.

Beispiel: Eine Datei Zeile für Zeile einlesen

```
$handle = fopen ("/tmp/inputfile.txt", "r");  
if (!$handle)  
    { echo "/tmp/inputfile.txt nicht geöffnet"; exit; }  
  
while (!feof($handle)) {  
    $buffer = fgets($handle, 4096);  
    echo $buffer;  
}  
  
fclose ($handle);
```

## Schema: eine Datei ändern

`copy` -- Kopiert eine Datei

```
bool copy (string source, string dest)
```

**Beispiel:** Aus einer Datei mit je einem Vereinsnamen pro Zeile sollen alle Zeilen gelöscht werden, die „FC“ im Vereinsnamen enthalten, d. h. „1.FC Köln“ verschwindet und „SV Werder Bremen“ bleibt drin.

**Vorgehen:** Datei kopieren; die Kopie lesen; das Original überschreiben.

```
copy ("vereine", "vereine.bak");
$in = fopen ("vereine.bak", "r");
if (!$in) { echo "vereine.bak nicht geöffnet"; exit; }
$out = fopen ("vereine", "w");
if (!$out) { echo "vereine nicht geöffnet"; exit; }

while (!feof ($in)) {
    $buffer = fgets ($in);
    if (strpos ($buffer, "FC") === false) {
        fputs ($out, $buffer);
    }
}
fclose ($in);
fclose ($out);
```

Dateien kopieren  
und öffnen

Datei durchlesen  
Teil-string suchen  
Zeile ausgeben

Dateien schließen

## S3.6 Arrays

Ein **Array** ist eine **Abbildung von Indizes (oder Schlüsseln) auf Werte**.

Jedes **Element eines Arrays** ist ein **Paar aus Index** und zugeordnetem **Wert**.

**Beispiele:** Index      Wert  
                  Monatsnr. Monatsname

```
array ( 1 => "Januar",  
        2 => "Februar",  
        ...  
       12 => "Dezember" )
```

Bundesligatabelle

```
array ( 1 => "FC Bayern München",  
        2 => "Hamburger SV",  
        3 => "SV Werder Bremen",  
        ...  
       18 => "1.FC Kaiserslautern" )
```

**Schlüssel können ganze Zahlen (int) oder Zeichenreihen (string) sein,**  
**Werte können beliebigen Typ haben.**

# Zuweisung von Array-Referenzen

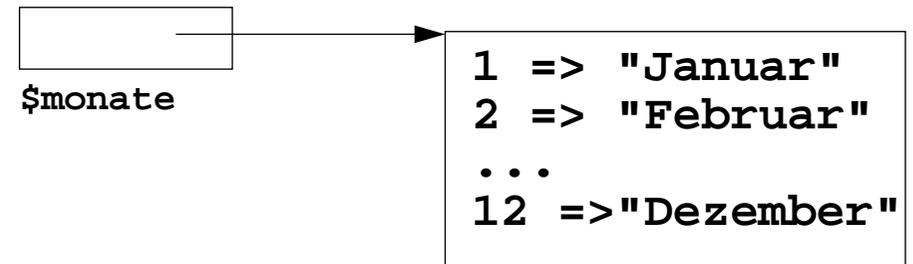
Jeder **Array-Wert** wird bei der Ausführung des Programms zusammenhängend im Speicher untergebracht.

Die **Referenz auf die Stelle des Array-Wertes** im Speicher kann man Variablen zuweisen.

```
$monate =
```

```
array ( 1 => "Januar",  
        2 => "Februar",  
        ...  
       12 => "Dezember" );
```

Variable    Referenz    Array-Wert



Man kann auch die **Elemente einzeln** an Indexpositionen der Variable zuweisen, z. B.

```
$monate[1] = "Januar";  
$monate[2] = "Februar";  
...  
$monate[12] = "Dezember";
```

## Array-Variable indizieren

Eine **Variable**, die eine Array-Referenz enthält, kann man **indizieren**, um den Wert eines bestimmten **Elementes zu lesen** oder zu (über)**schreiben**.

```
echo $monate[5];           gibt "Mai" aus
```

```
$monate[1] = "Jan";       überschreibt das 1. Element
```

**Beispiel:** In einer Klausur wurden maximal 10 Punkte vergeben. Die von den Teilnehmern erreichten Punktzahlen werden je eine pro Zeile von einer Datei gelesen. Mit einem Array von 11 Zählern wird die **Häufigkeit jeder Punktzahl ermittelt**:

```
$i = 0;
while ($i<=10) {
    $punkte[$i]=0; $i=$i+1;
}
$in = fopen("Klausur", "r");
if (!$in) { echo "Klausur nicht geöffnet"; exit; }

while (!feof($in)) {
    $buffer = fgets($in);
    if (strlen($buffer)>0) {
        $punkte[(int)$buffer] += 1;
    }
}
fclose ($in);
```

```
echo "Punkte\tAnzahl\n";
$i = 0;
while ($i <= 10) {
    echo $i, "\t",
        $punkte[$i], "\n";
    $i = $i + 1;
}
```

## Zeichenreihen als Array-Schlüssel

Auch **Zeichenreihen** können **als Indizes** verwendet werden, man spricht dann von **Schlüsseln**.

**Beispiele:** Monatsname => Monatsnummer oder Vereinsname => Punktestand

Neue **Notation für Schleifen zum Durchlaufen aller Elemente eines Arrays**

```
foreach ($arr as $key => $value){
    ... Benutzung der Variablen $key und $value
}
```

```
$liga = array (
    "VfB Stuttgart" => 22,
    "1.FC Köln" => 12,
    ...
    "SV Werder Bremen" => 35);

arsort ($liga);
echo "Tabellenstand\n\n";

foreach ($liga as $verein => $punkte) {
    echo $verein, "\t", $punkte, "\n";
}
```

### Tabellenstand

FC Bayern München	41
Hamburger SV	37
SV Werder Bremen	35
FC Schalke 04	31
Hertha BSC Berlin	25
Borussia M'gladbach	22
VfB Stuttgart	22
Borussia Dortmund	21
Hannover 96	20
VfL Wolfsburg	18
Bayer 04 Leverkusen	18
Eintracht Frankfurt	18
DSC Arminia Bielefeld	17
1.FSV Mainz 05	15
1.FC Nürnberg	13
1.FC Köln	12
MSV Duisburg	11
1.FC Kaiserslautern	9

## Weitere Schleifenformen

### foreach-Schleifen über Arrays in 2 Formen

- für jedes Element soll auf Index/Schlüssel **und** Wert zugegriffen werden:

```
$monate = array (1 => "Jan", 2 => "Feb", ...12 => "Dez");  
foreach ($monate as $nr => $name)  
{ echo $nr, "\t", $name, "\n";}
```

- für jedes Element soll **nur auf den Wert** zugegriffen werden:

```
foreach ($monate as $name) {echo $name, "\n";}
```

### Allgemeine for-Schleife

```
for (Initialisierung; Bedingung; Fortschaltung) { Rumpf }
```

hat die **gleiche Bedeutung** wie die while-Schleife

```
Initialisierung; while (Bedingung) {Rumpf; Fortschaltung}
```

wird meist als **Zählschleife** benutzt

```
for ($i = 1; $i <= 12; $i++) {echo $i, "\t", $monate[$i], "\n";}
```

**$\$i++$**  erhöht den Wert von  $\$i$  um 1. Wird  **$\$i++$**  als Ausdruck verwendet, so hat er den Wert von  $\$i$  vor dem Erhöhen, z. B. in  $\$monate[\$i++]$ .

## S3.7 Funktionsdefinitionen

**Funktion:** Rechenvorschrift mit einem **Namen** und ggf. **formalen Parametern**, die an mehreren Stellen im Programm mit unterschiedlichen **aktuellen Parametern aufgerufen** werden kann.

Beispiel für die Definition einer Funktion:

```
function prTabellenZelle ($v) { print "<td><b>$v</b></td>"; }
```

Aufrufe der Funktion:

```
prTabellenZelle ("Hallo!");           prTabellenZelle ($x * $y);
```

Zweck von Funktionen:

- **Wiederholung** gleicher Berechnungen an mehreren Stellen des Programmes **vermeiden**
- **abstrahieren:** das **Was** soll berechnet werden? vom **Wie** soll das geschehen?  
also die **Aufgabe** von der **Lösung im Detail**  
im Programmtext: Name und akt. Parameter im **Aufruf** Rumpf in der **Definition**

# Syntax von Funktionsdefinitionen

```
FunctDef ::= 'function' FunctName '(' [FormParams] ')'  
          '{' Statement* '}'
```

```
FormParams ::= FormParams ',' FormParam | FormParam
```

```
FormParam ::= VariableName                                call-by-value  
            | VariableName = Literal                    call-by-value, optional  
            | '&' VariableName                          call-by-reference
```

```
Statement ::= 'return' [Expression] ';' 
```

Es darf **keine 2** Funktionsdefinitionen geben,  
die den **gleichen Funktionsnamen definieren**.

Das heißt insbesondere, dass der Name verschieden  
von allen Funktionsnamen der Bibliothek sein muss.

## Funktionen mit und ohne Ergebnis

**Funktionen ohne Ergebnis** werden als **Anweisungen** aufgerufen. Der Aufruf bewirkt einen Seiten-Effekt (Veränderung von Werten von Variablen, Ausgabe, Eingabe), z. B.

```
function prTabellenZelle ($v) { print "<td><b>$v</b></td>"; }
```

Aufrufe der Funktion als Anweisungen:

```
prTabellenZelle ("Hallo!");          prTabellenZelle ($x * $y);
```

**Funktionen** können durch Ausführen einer **return-Anweisung** ein **Ergebnis** liefern. Dann können ihre Aufrufe als **Ausdruck** verwendet werden, z. B.

```
function TabellenZelle ($v) { return "<td><b>$v</b></td>"; }
```

Aufrufe der Funktion als Ausdrücke:

```
print TabellenZelle ("Hallo!");      $z = TabellenZelle ($x * $y);
```

Die Ausführung einer **return-Anweisung** liefert den **Wert des Ausdruckes als Ergebnis** und **beendet die Ausführung des Funktionsrumpfes**. Eine **return-Anweisung** ohne Ausdruck beendet die Ausführung des Funktionsrumpfes ohne Ergebnis.

**Funktionen mit Ergebnis und ohne Seiten-Effekt** sind meist **allgemeiner einsetzbar**.

# Parameterübergabe call-by-value

## Parameterübergabe:

Bezug zwischen **aktuellem Parameter im Aufruf** und **formalem Parameter in der Funktionsdefinition**.

**Call-by-value:** wichtigste Art der Parameterübergabe (auch in C, C++, Java, Pascal, Ada, ...)

Der **formale Parameter ist eine Variable**, die nur während der Ausführung des Funktionsrumpfes existiert.

Der **aktuelle Parameter ist ein Ausdruck**. Er wird beim Aufruf ausgewertet. Mit seinem Wert wird der formale Parameter initialisiert.

**Zuweisungen an den formalen Parameter** im Funktionsrumpf **wirken sich nicht** auf den **aktuellen Parameter** aus.

Beispiel:

```
function fak ($n) {  
    $sum = 1;  
    while ($n > 1) {  
        $sum = $sum * $n;  
        $n = $n - 1;  
    }  
    return $sum;  
}
```

```
$k = 5;  
print fak ($k);  
print " ist Fakultät von $k\n";
```

# Globale und lokale Variable

Wir unterscheiden global und lokale Variable:

## Globale Variable:

- wird durch Benutzung des Namens **außerhalb von Funktionsdefinitionen eingeführt**;
- ihr **Name gilt im ganzen Programm**; aber **in Funktionsrümpfen nur**, wenn er dort als **global** deklariert wird;
- sie **existiert** während der **gesamten Ausführung** des Programms.

## Lokale Variable:

- wird durch Benutzung des Namens **in einer Funktionsdefinition eingeführt**;
- ihr **Name gilt im Rumpf dieser Funktion**; er **kann mit Namen anderer Variable** in anderen Funktionen oder globaler Variable **übereinstimmen**;
- sie **existiert jeweils während eines Aufrufes** dieser Funktion

```
function namesOut ($v) {  
    global $out, $lineCnt;  
  
    $lg = strlen ($v);  
    fputs($out, $v);  
    $lineCnt++;  
}  
  
$out = fopen ("names", "w");  
$lineCnt = 0;  
$sum = 0;  
  
while ( ... ) {  
    namesOut (...);  
}  
  
print $lineCnt;  
fclose ($out);  
  
global: $out, $lineCnt, $sum  
  
lokal in namesOut: $v, $lg
```

## Eine HTML-Seite erzeugen

```
<?php
function headOut ($title) {
    global $out;
    fputs ($out, <<<KOPF
<html><head>
    <title>$title</title>
</head><body>
    <h3>$title</h3>\n
KOPF
    );
}

function footOut () {
    global $out;
    fputs ($out,
        "</body></html>\n");
}
```

```
$out = fopen ("such.html" ,"w");
$tel = fopen ("tele.txt" ,"r");
headOut ("Suche im Telefonbuch");
$name = "Thies";
fputs ($out,
    "<h4>Ergebnisse f&uuml;r $name".
    "</h4><p>\n<pre>\n");

while (!feof($tel)) {
    $line = fgets($tel, 64);
    if (preg_match("/$name/i",$line)) {
        fputs ($out, "\t$line");
    }
}
fputs ($out, "</pre></p>\n");
footOut();fclose($tel);fclose($out);
?>
```

**HTML-Datei:**

```
<html><head>
    <title>Suche im Telefonbuch</title>
</head><body>
<h3>Suche im Telefonbuch</h3>
<h4>Ergebnisse f&uuml;r Thies</h4><p>
<pre>
    Thies, Michael, Dr. 6682
</pre></p>
</body></html>
```

## S3.8 PHP-Eingabe mit HTML-Formularen

**HTML-Formulare** enthalten grafische Elemente für **interaktive Eingaben** (siehe S2.3)

Ein Formular-Element liefert ein **Paar von Zeichenreihen** `name` und `value`.  
`name` ist der Name des Formular-Elementes, `value` der eingegebene Wert.

Ein einzeiliges Textfeld mit der gezeigten Eingabe

```
<input type="text" name="Zuname" size="10">
```

Zuname:

liefert das Paar "Zuname" und "Kastens".

Ein PHP-Programm nimmt die Eingabe aller Elemente eines Formulars als **Schlüssel-Wert-Paare im vordefinierten, globalen Array** `$_REQUEST` entgegen.

Im obigen Beispiel hat dann `$_REQUEST["Zuname"]` den Wert "Kastens".

Mit einer **foreach-Schleife** über das Array `$_REQUEST` kann man die **Werte aller Formular-Elemente** ermitteln:

```
foreach ($_REQUEST as $name => $value) {  
    echo "$name => $value\n";  
}
```

# Programmstruktur für Formular-Eingabe

Das PHP-Programm wird **zweimal ausgeführt** und läuft dabei in verschiedene Zweige:

Am Inhalt des Arrays wird die Entscheidung getroffen

Beim ersten Mal wird das **Formular angeboten**.

Beim zweiten Mal wird die **Eingabe verarbeitet**.

```
<html><head>
  <title>PHP Formular-Eingabe</title>
</head>
<body>
<?php

if (!isset($_REQUEST['submit'])) {

  // HTML-Formular ausgeben
  <form action="http://..." method="POST">
  // Formular-Elemente ...
  </form>

} else {

  // Formular-Eingabe aus $_REQUEST
  // entnehmen, verarbeiten und
  // Ergebnisse ausgeben

}
?>
</body></html>
```

# Ein komplettes Beispiel

```

<html><head><title>PHP Formular-Eingabe</title></head><body>
<?php
if (!isset($_REQUEST['submit'])) {
echo <<<FORMULARANZEIGE
<form action="http://..." method="POST">
  <p>Zuname:<br>
  <input type="text" name="Zuname" size="10"></p>
  <p>Gästebuch:<br>
  <textarea name="eintrag" rows="3" cols="10"></textarea></p>
  <p>Versand:<br>
  <input type="checkbox" name="speed" value="fast">eilig</p>
  <p>Zahlung:<br>
  <input type="radio" name="pay" value="cash">Bar<br>
  <input type="radio" name="pay" value="card">Karte<br></p>
  <p>Wochentag:<br>
  <select name="tag" size="3">
    <option value="freitag">Freitag
    ...
  </select></p>
  <input type="reset" value="löschen"><br>
  <input type="submit" value="abschicken" name="submit"><br>
</form>
FORMULARANZEIGE;
} else { echo "<h4>Ihre Eingabe:</h4><p>\n<pre>";
  foreach ($_REQUEST as $name => $value) {
    echo "$name => $value\n";
  }
  echo "</pre>";
}
?>
</body></html>

```

Home Boc

**Zuname:**  
Kastens

**Gästebuch:**  
Das sieht  
noch recht  
mager aus!

**Versand:**  
 eilig

**Zahlung:**  
 Bar  
 Karte

**Wochentag:**  
Freitag  
Samstag  
Sonntag

löschen  
abschicken

**Ihre Eingabe:**

```

Zuname => Kastens
eintrag => Das sieht
noch recht
mager aus!
speed => fast
pay => cash
tag => samstag
submit => abschicken

```

# E3. Statische und dynamische Semantik am Beispiel PHP

## E3.1 Statische Bindung von Namen

**Namen im Programmtext** werden an **Objekte der Programmausführung** gebunden:

Variablenname	an Variable mit Speicherstelle
Funktionsname	an definierte Funktion

**Gültigkeitsbereich** einer Bindung:

der Bereich des Programms, in dem ein Name  $n$  an das Programmobjekt  $o$  gebunden ist.

Außerhalb des Gültigkeitsbereiches ist  $n$  nicht an  $o$  sondern an ein anderes Programmobjekt oder gar nicht gebunden.

Sprachen haben unterschiedliche **Gültigkeitsregeln** (engl.: **scope rules**).

Eine **Bindung** entsteht

- **explizit durch eine Definition**, z. B. in PHP: Funktionen und formale Parameter; in C, C++, Java, u.v.a.m.: Variablen, Funktionen, Typen, Parameter, Marken etc.
- **implizit durch Benutzung des Namens**: z. B. in PHP: Variable

**Explizite Definitionen** ordnen dem Programmobjekt **statische Eigenschaften** zu: z. B. in PHP:

Funktion: Anzahl und Übergabeart der Parameter, Funktionsrumpf

Parameter: Übergabeart

in statisch typisierten Sprachen:

Variable: Typ ihrer Werte

```
{ float x; int i; x = 3.1; i = 0; }
```

# Gültigkeitsbereiche in PHP-Programmen

## Gültigkeitsregeln in PHP: Art des Namens

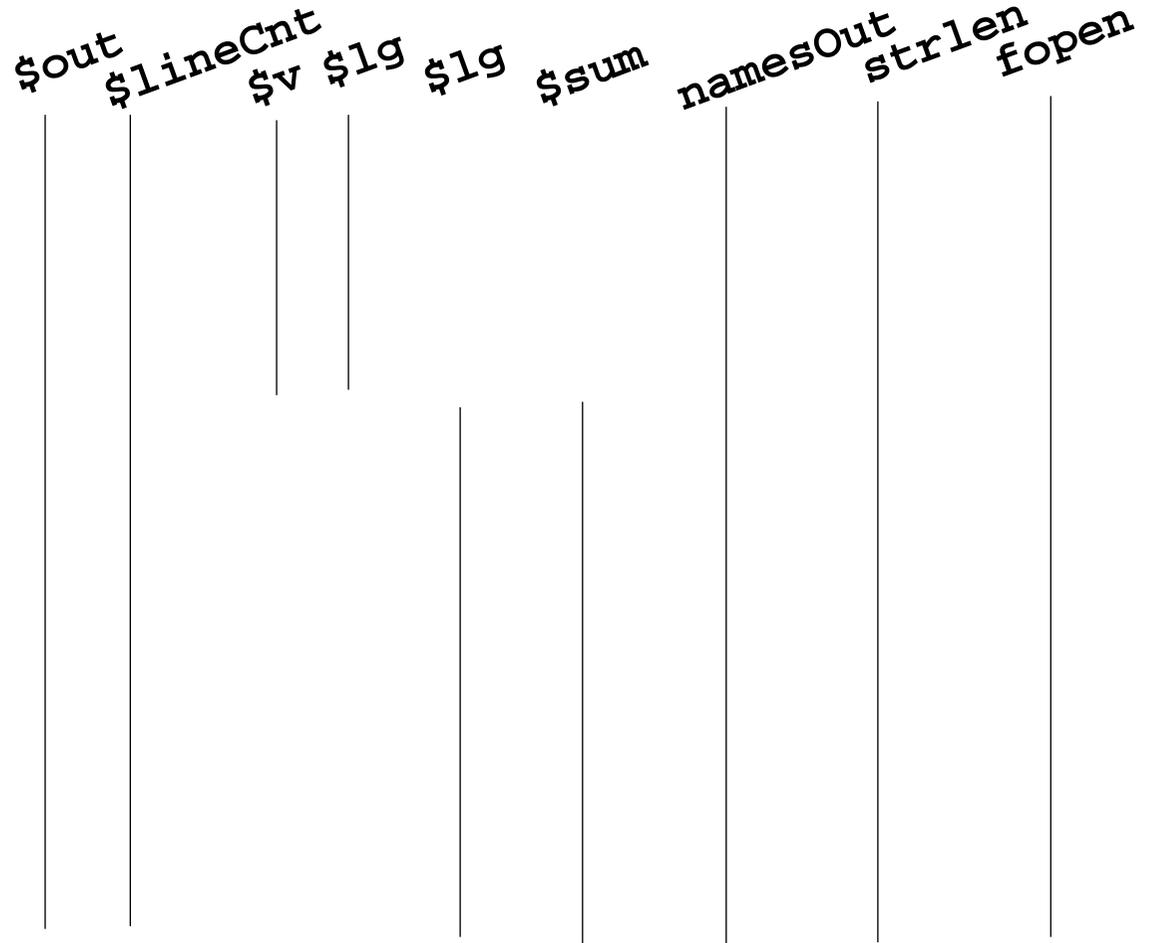
- Funktion
- Parameter
- lokale Variable
- globale Variable `$x`

## Gültigkeitsbereich

- ganzes Programm
- sein Funktionsrumpf
- ihr Funktionsrumpf
- ganzes Programm außer Funktionsrümpfe ohne `global $x`

Beispiel für Gültigkeitsbereiche:

```
function namesOut ($v) {
    global $out, $lineCnt;
    $lg = strlen ($v);
    fputs($out, $v);
    $lineCnt++;
}
$out = fopen ("names", "w");
$lineCnt = 0;
$sum = 0;
$lg = 5;
while ( ... ) {
    namesOut (...);
}
print $lineCnt;
fclose ($out);
```



# E3.2 Lebensdauer von Variablen

**Lebensdauer:** Begriff der **dynamischen Semantik**

Zeit, während der eine **Variable im Speicher existiert**;

sie wird ausgedrückt in Bezug auf die **Ausführung von Programmabschnitten**

## Art von PHP-Variablen

globale Variable

lokale Variable und Parameter einer Funktion

## Lebensdauer

gesamte Ausführung des Programms

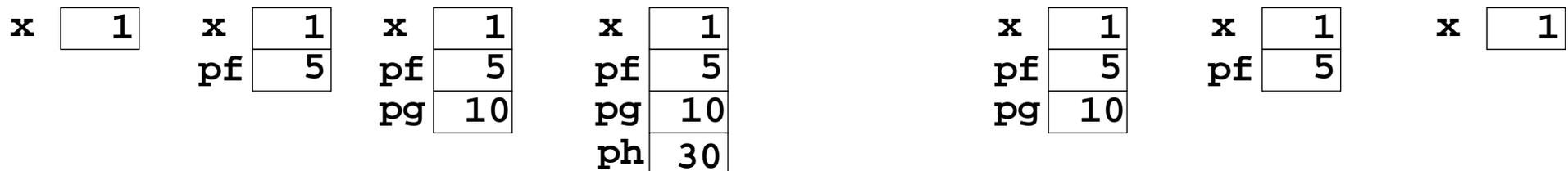
Ausführung eines Aufrufes der Funktion

## Beispielprogramm:

```
function ff ($pf) {gg(2*$pf); print $pf . "\n";}
function gg ($pg) {hh(3*$pg); print $pg . "\n";}
function hh ($ph) {print $ph . "\n";}
$x = 1; ff (5); print $x . "\n";
```

## ausgeführte Aufrufe:

```
ff(5);      gg(10);      hh(30); print $ph;      print $pg;      print $pf; print $x;
```



## Variablen im Speicher und ihre Lebensdauer

# Rekursive Funktionsaufrufe

**Rekursiv:** auf sich selbst bezogen

## Rekursive Funktion $F$ :

Der Rumpf von  $F$  enthält einen Aufruf von  $F$  oder von einer anderen Funktion, die  $F$  direkt oder indirekt aufruft.

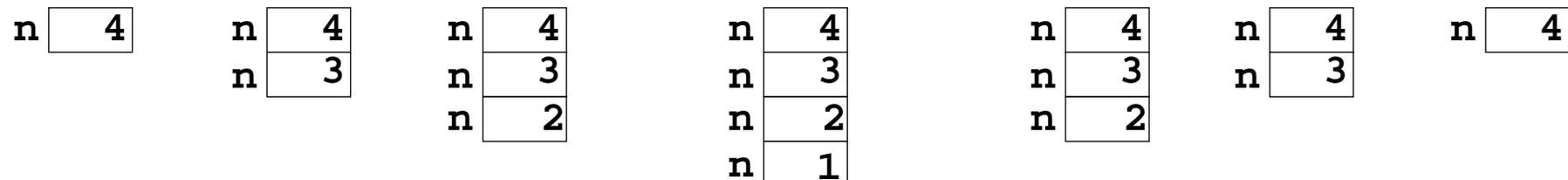
## Beispiel:

```
function fak ($n) {
  if ($n <= 1) return 1;
  else return $n * fak ($n - 1);
}

print fak (4). "\n";
```

## ausgeführte Aufrufe:

fak(4)      fak(3)      fak(2)      fak(1) return 1;    return 2;    return 6;    return 24;



**Variablen im Speicher und ihre Lebensdauer**

## E3.3 Dynamische Semantik von Aufrufen

Der **Aufruf einer Funktion** führt die definierte Berechnung aus mit den (aktuellen) **Parameterwerten**, die beim Aufruf angegeben sind. (siehe EWS-4.23)

Aufrufe haben die **Form**: `FuncExpr ( [ Parameters ] )`

Ein **Aufruf** wird in folgenden **Schritten** ausgeführt:

0. **FuncExpr auswerten** (ist meist nur ein Name), liefert eine Funktion.
1. **Aktuelle Parameter** an der Aufrufstelle **auswerten**.
2. **Speicher** für formale Parameter und lokale Variablen **anlegen**.
3. Die **Werte** der **aktuellen** Parameter an die **formalen** Parameter **zuweisen** (bei Übergabeart *call-by-value*).
4. **Anweisungen aus dem Funktionsrumpf** im Speicher des Aufrufes ausführen.
5. **Speicher** (aus Schritt 2) **freigeben**; **Ergebnis** an die Aufrufstelle **zurückgeben**.

Verschieden **Arten der Parameterübergabe**: (siehe EWS-4.38)

**Call-by-value**: Der formale Parameter ist eine Variable, die mit dem Wert des aktuellen Parameters initialisiert wird.  
in PHP, C, C++, Java, Ada, Pascal, u.v.a.m.

Call-by-reference: in PHP, C++, Pascal

Call-by-value-and-result: in Ada

## S4 JavaScript

**Skriptsprache**, 1995 bei Netscape als *LiveScript* entwickelt dann in *JavaScript* (unpassend) umbenannt. Standard ECMA 262 (1996) fasst JavaScript (Netscape) und JScript (Microsoft) zusammen

- abgeleitet von Perl; Notation wie C, C++, Java, PHP; sonst kein Bezug zu Java
- **interpretiert, dynamisch typisiert**
- spezielle **objektorientierte** Eigenschaften
- eingebettet in HTML
- Interpretierer **in Web-Browser integriert** (Netscape, Internet Explorer)
- Zugriff auf Elemente des dargestellten Dokumentes (DOM)

### **Anwendungszwecke:**

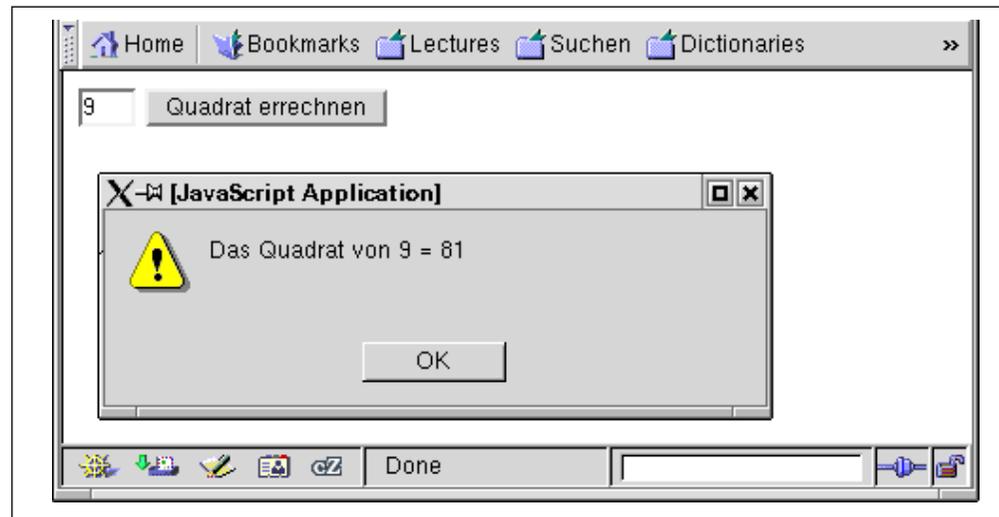
- Programme, die im Web-Browser des **Client** ausgeführt werden
- **Bedienoberflächen** in dynamischen Web-Seiten
- **Reaktionen auf Ereignisse** bei der Interaktion mit Web-Seiten
- Formular-Elemente **dynamisch erzeugen**, Eingabe **prüfen**
- **Animationseffekte**

# Ein erstes Beispiel in JavaScript

```
<html><head>
  <title>Quadrat</title>
<script type="text/javascript">
  function Quadrat() {
    var Zahl = document.Formular.Eingabe.value;
    var Ergebnis = Zahl * Zahl;
    alert ("Das Quadrat von " + Zahl + " = " + Ergebnis);
  }
</script>
</head><body>
  <form name="Formular" action="">
    <input type="text" name="Eingabe" size="3">
    <input type="button" value="Quadrat errechnen"
      onClick="Quadrat()">
  </form>
</body></html>
```

Funktionsdefinition  
eingebettet mit `script`-  
Tags

Aufruf der  
Funktion bei  
`onClick`-Ereignis



Anzeige im  
Browser

## S3.1 Einbettung in HTML script-Tags

**Programmfragmente in JavaScript** können auf unterschiedliche Arten in HTML-Dateien eingebettet werden. Wir betrachten hier 3 von 4 Arten:

1. Mit **script**-Tags geklammerte Programmfragmente:

```
<script type="text/javascript">
function makeTextElem (name) {
    return "<input type=\"text\" +
        \" name=\"\" + name +
        \"\" size=10>";
}
</script>
```

```
<script type="text/javascript">
document.writeln
    (makeTextElem ("Zuname"));
</script>
```

Solche Programmfragmente werden **ausgeführt und die Ausgabe**, die sie erzeugen wird an ihrer Stelle in die HTML-Datei **eingesetzt** und angezeigt (wie in PHP).

**Funktionsdefinitionen** erzeugen keine Ausgabe;  
man bettet sie sinnvoll in dem **head**-Teil ein.  
(siehe vorige Folie)

## Einbettung: Werte spezieller HTML-Attribute

2. Einige **HTML-Attribute** benennen **Ereignisse**, die der Bediener auslösen kann, z. B. einen Knopf betätigen.  
Als **Wert des Attributes** wird eine **Anweisungsfolge** angegeben. Sie wird ausgeführt, wenn das Ereignis eintritt.

```
<input type="button" value="Quadrat errechnen"  
onClick="Quadrat()">
```

3. In einem **Anker-Element** kann man statt einer URL auch eine **Anweisungsfolge** mit vorangestelltem **javascript:** angeben. Beim Klicken darauf wird sie ausgeführt.

```
<a href="javascript: alert ('habt Geduld');">Musterlösung</a>
```

Meist werden an solchen Stellen anderweitig definierte **Funktionen aufgerufen**.



## S3.2 Spracheigenschaften Notation

Die Notation stimmt in Vielem mit der von PHP, vom Kern von C, C++ und Java überein.

einige wichtige **Unterschiede**:

**Bezeichner, Wortsymbole:**

**Groß- und Kleinschreibung ist unterscheidend (case-sensitive):**

`headOut` verschieden von `headout`

`true` und `false` aber nicht `True` oder `False`

**Bezeichner:**

**einheitliche Schreibweise** für alle Arten von Bezeichnern:

(Buchstabe | \$ | \_) (Buchstabe | \$ | \_ | Ziffer)\*

**Anweisungen:**

**abschließendes ;** kann am Zeilenende entfallen

**mit ;**

```
lineCount = 1;
sum = 0;
while (lineCount < 100) {
    ...
}
```

**ohne ;**

```
lineCount = 1
sum = 0
while (lineCount < 100) {
    ...
}
```

# Variable und Zuweisungen

Variable und Zuweisungen haben die **gleiche Bedeutung wie in PHP** (siehe EWS-4.5).

- **Variablennamen** brauchen nicht durch ein **\$**-Zeichen gekennzeichnet zu werden.
- Eine Variable kann (wie in PHP) **Werte beliebiger Typen** annehmen.
- Es wird (wie in PHP) unterschieden zwischen  
**globalen Variable x**: gilt im ganzen Programm, außer in Funktionen mit einem lokalen **x**  
**lokale Variable**: gilt in der Funktion, in der sie **definiert** ist
- Es gibt **Definitionen für Variable**.  
**Lokale Variable muss** man, **globale kann** man definieren.  
Wird eine Variable in einer Funktion benutzt aber nicht definiert, so ist sie global.

In einer **Variablendefinition** können eine oder mehrere Variable definiert werden, sie können auch durch **Zuweisung eines Anfangswertes initialisiert** werden, er muss durch ein **Literal oder eine andere Variable** angegeben werden.

```
var line;  
var sum, result;  
  
var col, count = 3, row;  
var  minimum = count,  
    maximum = 999;
```

```
function compute (n) {  
    var sum = n;  
    sum *= col;  
    return sum;  
}
```

**n lokal**  
**sum lokal**  
**col global**

# Datentypen

## number:

**numerische Werte**, ganze Zahlen und Gleitpunktzahlen zusammengefasst,  
spezieller Wert für undefinierte Werte **NaN** (not a number)

**Literale** wie in PHP

**arithmetische Operatoren** wie in PHP

## string:

**Zeichenreihen**

**Literale:** Klammerung mit " oder ' ist gleichbedeutend

'Er sagt "Hallo!'"      "Sag's auch!"

Umschreibungen nicht-druckbarer Zeichen wie in PHP \n, \t, usw.

In Zeichenreihen werden **Variablenwerte nicht eingesetzt** (anders als in PHP)

**Operatoren:** Kontatenation +:      "Er heißt " + name

**string-Funktionen** in Objekt-Notation (wird nicht hier erklärt)

## boolean:

**Wahrheitswerte**

**Literale:** false, true

**Operatoren:** Konjunktion &&, Disjunktion ||, Negation ! (wie in PHP)

**Undefined:** einziger Wert `undefined`, Ergebnis bei Zugriff auf nicht-zugewiesene Variable

**Objekte inklusive Arrays**

**Funktionen als Werte**

# Konversion

**Alle Konversionen implizit** (coercion), so wie der Kontext es erfordert:

in **boolean**:

von **number**: 0, NaN -> **false**, sonst -> **true**

von **string**: "" -> **false**, sonst -> **true**

von **undefined**: **false**

in **number**:

von **boolean**: **false** -> 0, **true** -> 1

von **string**: ganze Zeichenreihe (ohne Leerzeichen am Anfang und Ende)  
wie ein numerisches Literal (ggf. mit Vorzeichen) lesen und  
konvertieren;

leere Zeichenreihe (oder nur Leerzeichen) -> 0;

sonst **NaN**

von **undefined**: **NaN**

in **string**:

von **boolean**: **false** -> "**false**", **true** -> "**true**"

von **number**: Zahlwert als Zeichenreihe, wie Literal (ggf. mit Vorzeichen)

von **undefined**: "**undefined**"

# Operatoren

Präzedenz (steigend)	Stellig- keit	Assozia- tivität	Operatoren	Erklärung
1	2	rechts	= += -= usw.	Zuweisungsoperatoren
2	3	links	? :	bedingter Ausdruck
3	2	links		log. Disjunktion (oder)
4	2	links	&&	log. Konjunktion (und)
5	2	links		Bitoperator
6	2	links	^	Bitoperator
7	2	links	&	Bitoperator
8	2	links	== != === !==	Gleichheit, Identität
9	2	links	< <= > >=	Ordnungsvergleich
10	2	links	<< >> >>>	shift-Operatoren
11	2	links	+ -	Konkatenation, Add., Subtr.
12	2	links	* / %	Arithmetik
13	1		! - ~	Negation (log., arithm., bitw.)
	1		++ --	Inkrement, Dekrement
	1		typeof void	Typ?; nach <code>undefined</code> konv.
14	1		() [] .	Aufruf, Index, Objektzugriff

# Ablaufstrukturen

**Anweisungsfolge:** wie in PHP

```
{ st = st + "*"; i = i + 1; }  
{ var k = 42; document.writeln (5*k); }
```

Durch `var` eingeführte Bindung gilt nicht nur in der Anweisungsfolge, sondern in umgebender Funktion bzw. im umgebenden Programm.

**Bedingte Anweisung:** wie in PHP

```
if (a < 0) {a = b;}      if (a < b) {min = a;} else {min = b;}
```

Bei einzelnen Anweisungen sind die `{}`-Klammern nicht nötig aber sinnvoll.

**while-Schleife:** wie in PHP

```
s = 0; while (s < n) {document.write ("*"); s++;}
```

**for-Schleife:** wie in PHP

```
for (s = 0; s < n; s++) {document.write ("*");}
```

**Funktionsaufrufe:** wie in PHP, aber nur call-by-value als Parameterübergabe

```
headOut ("Test")      document.write ("*")
```

**return-Anweisung:** wie in PHP

```
return n*42;          return;
```

# Funktionen

## Funktionsdefinitionen: wie in PHP

```
function Ueberschrift (grad, text) {  
    var marke = "h"+grad;  
    document.writeln ("<"+marke+">" + text + "</"+marke+">");  
}
```

`grad` und `text` sind formale Parameter, `grad`, `text` und `marke` sind lokale Variable.

```
function fak (n) {  
    if (n<=1) return 1; else return n * fak (n-1);  
}
```

Funktionen können im `head`- oder im `body`-Teil der HTML-Datei definiert werden. Aufrufe können in jedem JavaScript-Fragment stehen.

## Funktionen als Werte:

Funktionen kann man als Werte notieren. **Literal** für eine Funktion ohne Namen:

```
function (a, b) { return a + b; }
```

Solche **Funktionsliterate** kann man in Ausdrücken verwenden, z. B. einer **Variablen zuweisen**

```
var add = function (a, b) { return a + b; };
```

und den Wert der Variablen (die Funktion) aufrufen: `x = add (42*k, 3);`

# Arrays

Ein **Array ist eine Abbildung** von Indizes (oder Schlüsseln) auf Werte (wie in PHP):  
Jedes **Element eines Arrays** ist ein **Paar aus Index** und zugeordnetem **Wert**;  
erste Komponente der Paare: **numerischer Index oder ein string als Schlüssel**.  
In JavaScript sind **Arrays spezielle Objekte** (wird nicht hier erklärt).

**Arrays** kann man auf verschiedene Weise **erzeugen**:

Liste von Werten: `monatsName = new Array("", "Jan", ..., "Dez");`  
indiziert von 0 an, erster Wert ist hier irrelevant

leeres Array erweitern: `monatsName = new Array();`  
`monatsName[1] = "Jan"; monatsName[2] = "Feb"; ..`

auch `monatsNr = new Array();`  
`monatsNr["Jan"] = 1; monatsNr["Feb"] = 2; ...`

**Zugriff auf die Werte** von Array-Elementen durch

Indizierung wie in PHP: `monatsName[4]` oder `monatsNr["Apr"]`  
Objektselektion: `monatsNr.Apr`

**Schleife zum Aufzählen aller Elemente** eines Arrays (ähnlich wie in PHP):

`for (key in arr) { ... }` `arr` muss ein Array sein;  
mit `key` wird im Rumpf auf den Schlüssel eines Elementes zugegriffen

```
for (mname in monatsNr)
{ document.writeln (mname + "=>" + monatsNr[mname]); }
```

## S3.3 Objekte

Ein Grundkonzept von JavaScript ist der **Objektbegriff**. Er wird hier nur soweit eingeführt, dass die notwendigen Notationen verstanden werden.

Das aktuelle **Fenster** und das **Dokument** sind auch als Objekte verfügbar.

Ein Objekt wird **im Speicher** erzeugt und durch seine **Speicherstelle** eindeutig **identifiziert**.

```
student = {name:"E. Mustermann", matrNr:9999999};
```

erzeugt ein Objekt und weist seine Speicherstelle der Variablen zu.

Ein Objekt **besteht aus Komponenten**.

Sie haben jeweils einen **Namen** und einen **Wert** - wie Variable.

Das obige Objekt hat Komponenten mit Namen `name`, `matrNr`, usw.

Objektcomponenten werden durch **Objekt-Ausdruck.Name** zugegriffen.

```
student.name liefert "E. Mustermann"
```

**Arrays** und **Zeichenreihen** sind auch Objekte.

Array-Objekte haben auch eine Komponente `length`:

```
monatsName.length liefert 13 (den größten numerischen Schlüssel + 1, also 12+1)
```

Einige der **Komponenten können auch Funktionen** sein; sie heißen dann **Methoden**.

Ihre Aufrufe können die übrigen **Komponenten des Objektes lesen oder verändern**:

```
monatsName.reverse()    monatsName.sort()    document.writeln()
```

# Funktionen auf Zeichenreihen-Objekten

**Zeichenreihen sind Objekte** in JavaScript. **Aufrufe** von Funktionen (Methoden) auf Zeichenreihen werden in **Objekt-Notation** geschrieben, z. B.

```
var Aussage = "Der Mensch ist des Menschen Feind";  
var Suche = Aussage.indexOf("Mensch");
```

Die Funktion **indexOf** wird für die Zeichenreihe der Variablen **Aussage** mit dem Parameter **"Mensch"** aufgerufen. In PHP hätten wir in **Funktions-Notation** stattdessen geschrieben:

```
$Suche = strpos ($Aussage, "Mensch");
```

Diese Aufrufe **ändern die Zeichenreihe nicht**, auf die sie angewandt werden:

```
Aussage.toLowerCase();
```

**liefert eine neue Zeichenreihe**: alle Großbuchstaben durch Kleinbuchstaben ersetzt.

Weitere **string-Funktionen**:

<b>charAt(i)</b>	Zeichen an Position i liefern
<b>replace(r, s)</b>	Auftreten des regulären Ausdruck r suchen und ersetzen durch s
<b>search(r)</b>	suchen mit regulärem Ausdruck r
<b>substr(p,l)</b>	Teilzeichenreihe ab Position p der Länge l liefern

Einige **string-Funktionen** erleichtern die **Auszeichnung in HTML**:

```
Inh = "Inhalt"; document.write(Inh.anchor("IH"));
```

gibt einen Anker in HTML aus: `<a name="IH">Inhalt</a>`

# Zugriff auf Elemente des Dokumentes

Aus dem JavaScript-Programm kann man auf **Elemente des Dokumentes zugreifen**, das der Browser anzeigt. Das ist meist die HTML-Datei, in die das Programm eingebettet ist. Damit kann man z. B. den **Inhalt von Formular-Elementen** prüfen:

```
<script type="text/javascript">
  function Kontrolle() {
    var Zahl = document.QuadratForm.Eingabe.value;
    alert ("Eingabe war " + Zahl);
  }
</script>
<form name="QuadratForm" action="">
  <input type="text" name="Eingabe" size="3">
  <input type="button" value="Quadrat errechnen"
    onClick="Kontrolle()">
</form>
```

Hier wird die Zugriffsstruktur angewandt:

**document.FormularName.EingabeElementName.AttributName**

Alternativ kann man die Formulare im Dokument und ihre Elemente jeweils indizieren:

**document.forms[i].elements[j].AttributName**

also für obiges Beispiel:

```
var zahl = document.forms[0].elements[0].value;
```

# Document Object Model (DOM)

Das **Document Object Model (DOM)** regelt, welche Informationen ein Browser zur Client-seitigen Programmierung, z. B. in JavaScript, verfügbar macht:

1. Eigenschaften des gerade **angezeigten Dokumentes**, wie
 

<code>document.title</code>	Titel-string
<code>document.forms[i]</code>	Formulare, durchnummeriert von 0 an
<code>document.images[i]</code>	Bilder, durchnummeriert von 0 an
2. Methoden für das gerade **angezeigte Dokument**, wie
 

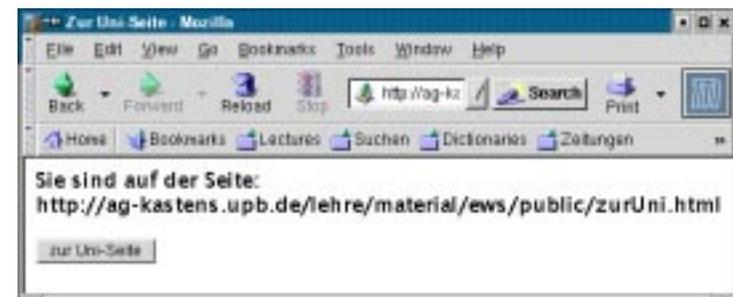
<code>document.write(string, ...)</code>	Ausgabe von Zeichenreihen
<code>document.writeln(string, ...)</code>	ebenso mit abschließendem Zeilenwechsel
3. Eigenschaften der vom Browser **angezeigten URL**, wie
 

<code>location.href</code>	die gesamte URL
----------------------------	-----------------
4. Methoden für die vom Browser **angezeigte URL**, wie
 

<code>location.reload()</code>	erneut laden
<code>location.replace(url)</code>	ein anderes Dokument anzeigen

## Beispiel: Seitenwechsel

```
<p>Sie sind auf der Seite:<br>
<script type="text/javascript">
  document.write(location.href);
</script>
<form name="UniURL" action="">
<input type="button" value="zur Uni-Seite"
  onClick='location.replace("http://www.upb.de")'>
</form>
```



## S3.4 Ereignisbehandlung

**Ereignis** in der Informatik (engl. event): Wahrnehmung einer Zustandsänderung.

**Ereignis-getriebene Programmierung:** Ereignissen werden Operationen zugeordnet (Ereignisbehandler); sie werden **bei Eintreten des Ereignisses ausgelöst**; typisch für Regelung und Steuerung realer Prozesse, Programmierung von Bedienoberflächen, z. B.

### Ereignis

Temperatur überschreitet 90 C  
Temperatur unterschreitet 80 C

Knopf wird gedrückt  
Mauszeiger ist über dem Bild

### Ereignisbehandlung

Kühlung anschalten  
Kühlung ausschalten

Formular abschicken  
Bildüberschrift blinken lassen

Behandlung des Ereignisses **Click** als Attribut von Formular-Elementen:

```
<form name="testForm">
  <input type="button" value="ping"
    onclick='alert("ping!");'>
  <input type="button" name="oKnopf" value="pong">
</form>
<script type="text/javascript">
  document.testForm.oKnopf.onclick=
    function(){alert("pong!");};
</script>
```

Operation als Attributwert  
im HTML-Tag zugeordnet

Funktion als Attributwert  
zugewiesen

## Wichtige Ereignisse

<b>Ereignisbehandler</b>	<b>HTML-Elemente</b>	<b>Bedeutung</b>
onclick	Knopf, Checkbox, Anker	Element wird angeklickt
onchange	Textfeld, Textbereich, Auswahl	Wert wird geändert
onkeydown onkeyup onkeypress	Dokument, Bild, Anker, Textfeld	Taste gedrückt, losgelassen
onmousedown onmouseup	Dokument, Knopf, Anker	Maustaste gedrückt, losgelassen
onmouseout	Bereiche, Anker	Mauszeiger verlässt einen Bereich
onmouseover	Anker	Mauszeiger über Anker
onreset, onsubmit	Formular	Reset, Submit für ein Formular
onselect	Textfeld, Textbereich	Element wird ausgewählt
onfocus onblur	Fenster, alle Formular-Elemente	Eingabefokus wird dem Element gegeben, entzogen

## Beispiel: Bildtausch

Für ein `img`-Element werden die Ereignisse `onmouseover` und `onmouseout` benutzt, um das Bild auszutauschen:

```
<html><head>
  <title>Bildtausch</title>
  <script type="text/javascript">
    function enter () { document.ews.src="ewsEin.jpg"; }
    function leave () { document.ews.src="ewsAus.jpg"; }
  </script>
</head><body>
  
</body></html>
```

Bild wechselt zwischen



und



# Unterschiede: Netscape Navigator und Internet Explorer

Leider sind Eigenschaften des Ereignismodells im Netscape Navigator und Internet Explorer unterschiedlich realisiert. Man muss auf sie in **separaten Programmzweigen** zugreifen.

```
<html><head>
  <title>Navigator vs. Explorer</title>
  <script type="text/javascript">
    function coord(e) {
      var isNavigator =
        navigator.appName.indexOf("Netscape") != -1;
      var x = isNavigator ? e.pageX : event.clientX;
      var y = isNavigator ? e.pageY : event.clientY;
      alert("Coordinate = (" + x + ", " + y + ")");
    }
  </script>
</head>
<body>
  <a name="hier">
  <a href="#hier" onclick="coord(event);">
    
  </a>
</body>
</html>
```

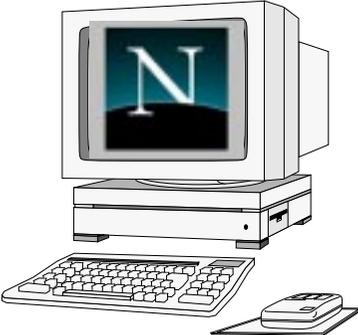
Unterscheidung der Browser

gerade eingetretenes Ereignis in der Variablen **event**:  
**Netscape: lokal** im Ereignisbehandler  
**Explorer: global**

Koordinaten der Stelle, wo ein Ereignis e eingetreten ist:  
**Netscape: e.pageX, e.pageY**  
**Explorer: e.clientX, e.clientY**

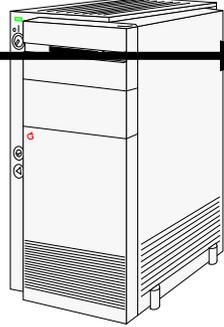
# Interaktion zwischen Client und Server

Client mit Browser



Benutzer-Interaktion  
Formularprüfung

Host-Rechner mit Web-Server



HTML-Datei mit  
Prüffunktion in JavaScript  
PHP-Programm für  
Formular erzeugen  
Eingabe verarbeiten

URL

Formular mit Prüffunktion

geprüfte Formular-Eingaben

Ergebnisse

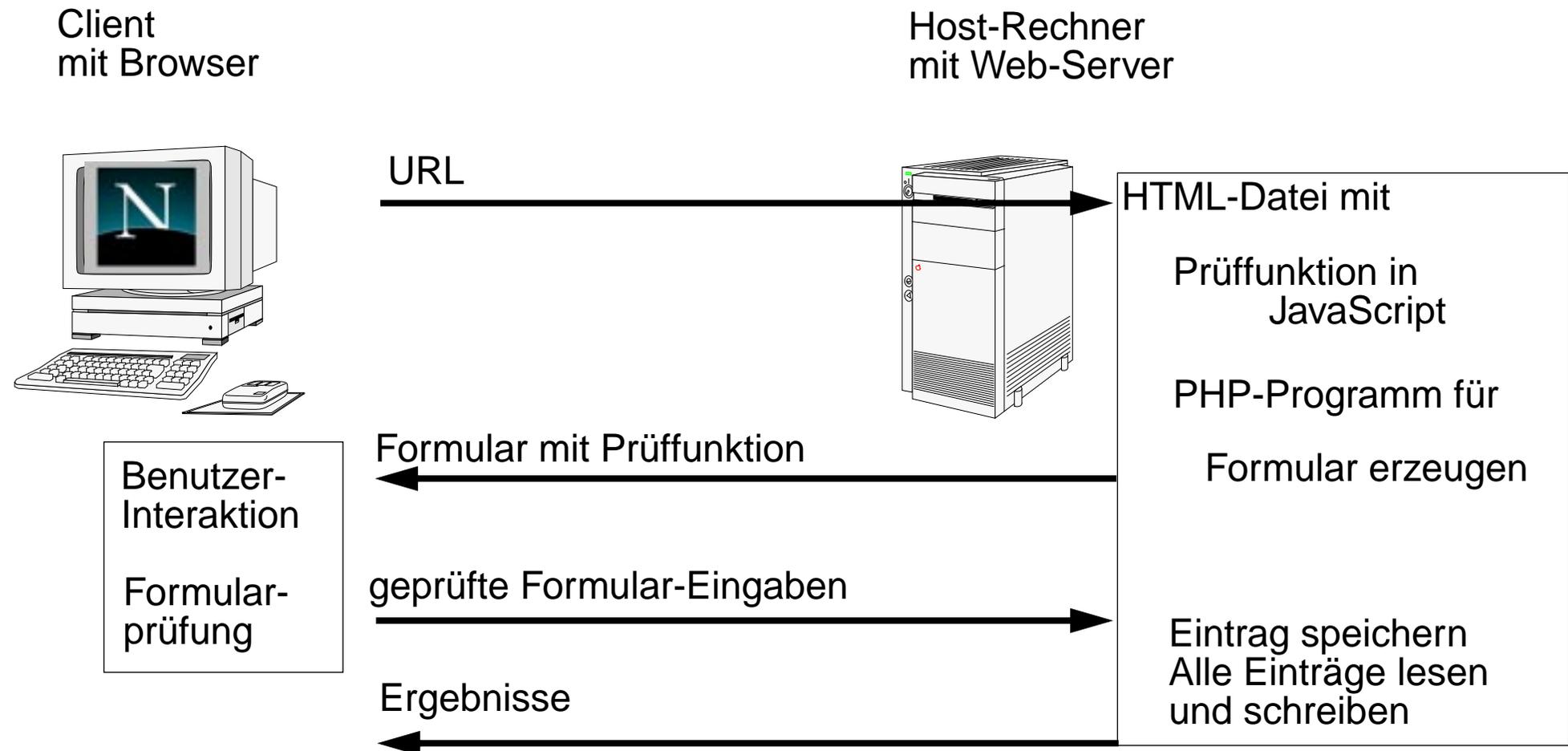
# Beispiel mit Formularprüfung

```
<html><head><title>Geprüfte Formular-Eingabe</title>
<script type="text/javascript">
  function check () {
    if (document.SpendenForm.Zuname.value.length < 2)
      { alert ("Zuname zu kurz!"); return false; }
    var betrag = document.SpendenForm.Spende.value;
    if (betrag <= 0) { alert ("Betrag angeben!"); return false; }
    if (betrag > 1000) { return confirm ("Höhe der Spende: " + betrag); }
    return true;
  }
</script>
</head><body>
<?php
  if (!isset($_REQUEST['submit'])) {
echo <<<FORMULARANZEIGE
  <form name="SpendenForm"
    action="http://ag-kastens.upb.de/..." method="POST">
    <p>Zuname:<br><input type="text" name="Zuname" size="10"></p>
    <p>Höhe der Spende:<br>
      <input type="text" name="Spende" size="10"></p>
    <input type="reset" value="löschen"><br>
    <input type="submit" value="abschicken" name="submit"
      onclick="return check();"><br>
  </form>
FORMULARANZEIGE;
  } else {
    echo "<h4>Vielen Dank für Ihre Spende:</h4><p>\n<pre>";
    foreach ($_REQUEST as $name => $value) { echo "$name => $value\n";}
    echo "</pre>";
  }
?></body></html>
```

# W7 Projekt im Zusammenhang Gästebuch

**Aufgabe:** Web-Seite mit einem kleinen Gästebuch; fordert Web-Surfer auf, etwas einzutragen; speichert die Einträge und zeigt alle an, die bisher gemacht wurden.

## Plan für Interaktionen:



# Planung der beiden Phasen

## Phase 1:

- 1a. **HTML-Formular** zum Eintragen
- 1b. **Prüfen**, ob etwas eingetragen wurde;  
sonst nur Einträge ansehen;  
in JavaScript

## Phase 2:

statt des Formulars wird angezeigt:

- 2a. Falls ein Eintrag gemacht wurde,  
**Eintrag an Datei anfügen**;  
Dank als HTML-Zeile
- 2b. HTML-Überschrift für Einträge;  
**alle Einträge** aus der Datei lesen  
und **anzeigen**

Phasen 1 und 2 als  
**Zweige im PHP-Programm**

Willkommen zu meinem Gästebuch!  
Hier kannst Du etwas eintragen:

Löschen Abschicken

Willkommen zu meinem Gästebuch!  
Danke für Deinen Eintrag!

Das wurde bisher eingetragen:

....

....

# Realisierung der Phase 1 (Formular)

## 1. HTML-Datei mit Formular entwickeln:

```
<html><head>
  <title>Mein Gästebuch</title>
</head><body>
  <h1>Willkommen zu meinem Gästebuch!</h1>
  <h2>Hier kannst du etwas eintragen:</h2>
  <form name="gastForm" action="gaestebuch.html" method="POST">
    <textarea name="eintrag" cols="48" rows="8"></textarea><br>
    <input type="reset" name="reset" value="Löschen">
    <input type="submit" name="submit" value="Abschicken">
  </form>
</body></html>
```

## 2. Layout prüfen, verbessern durch Style Sheet in CSS



```
<style type="text/css">
  h1,h2,h3
    {text-align:left}
  h1 {font-size:24}
  h2 {font-size:18}
  h3 {font-size:12}
</style>
```

# Realisierung der Phase 1 (Formular-Prüfung)

## 1. Formular-Prüfung einfügen:

```
<html><head>
  <title>Mein Gästebuch</title>
  <script type="text/javascript">
    function check () {
      if (document.gastForm.eintrag.value.length < 1)
        { return confirm ("Willst Du nichts eintragen?"); }
      return true;
    }
  </script>
</head><body>
  <h1>Willkommen zu meinem Gästebuch!</h1>
  <h2>Hier kannst du etwas eintragen:</h2>
  <form name="gastForm" action="gaestebuch.html" method="POST">
    <textarea name="eintrag" cols="48" rows="8"></textarea><br>
    <input type="reset" name="reset" value="Löschen">
    <input type="submit" name="submit" value="Abschicken"
      onclick="return check();">
  </form>
</body></html>
```



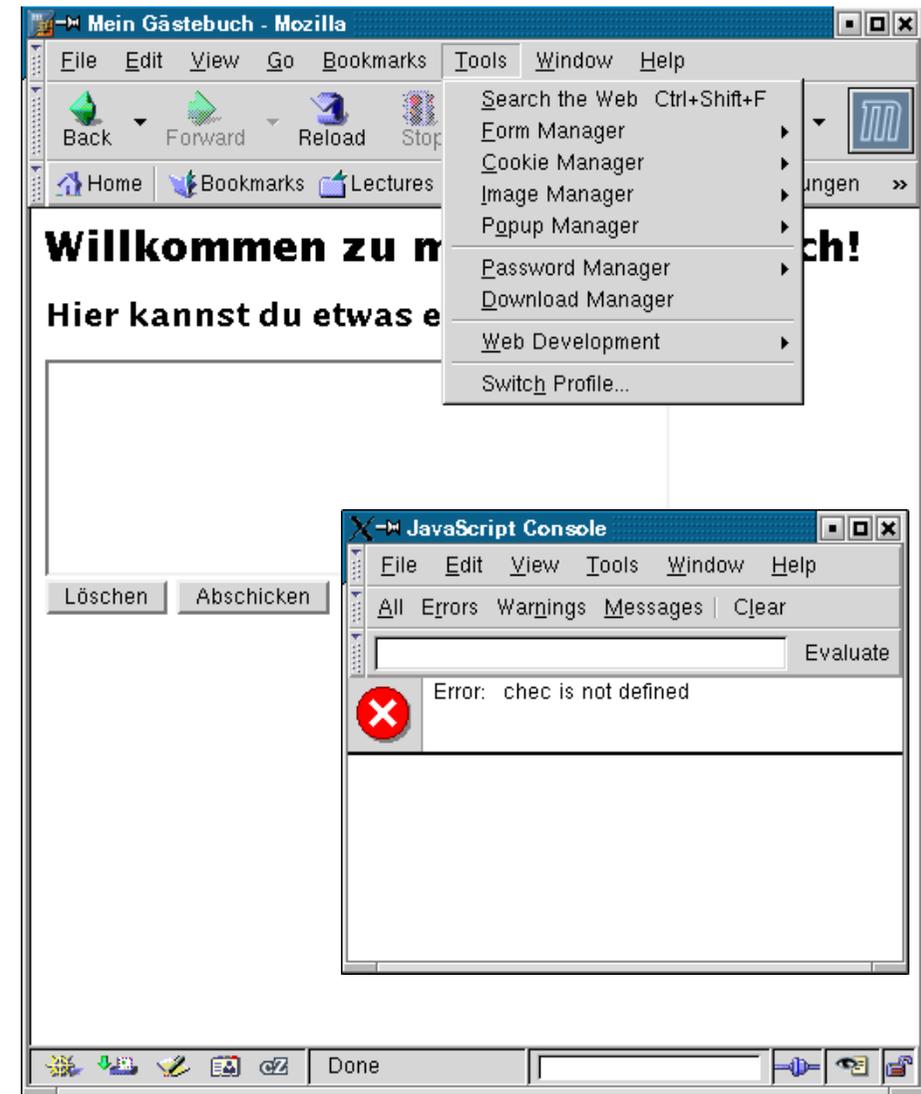
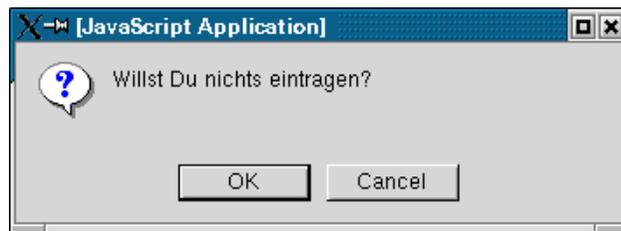
CSS

# Testen der Phase 1

## Alle Fälle der Bedienung des Formulars prüfen:

1. normal: eintragen und abschicken,
2. löschen,
3. nichts eintragen, abschicken, bestätigen,
4. nichts eintragen, abschicken, nicht bestätigen.

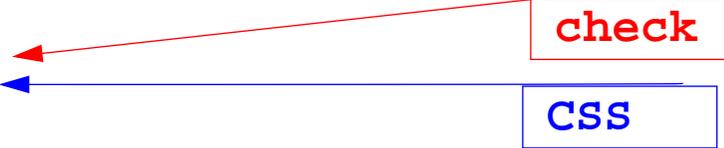
ggf. Fehler suchen, korrigieren und nochmal testen



# Herstellen der Standardstruktur

PHP-Programm mit 2 Zweigen:

```
<html><head>
  <title>Mein Gästebuch</title>
</head><body>
  <h1>Willkommen zu meinem Gästebuch!</h1>
<?php
  if (!isset($_REQUEST["submit"])) {
echo <<< EINTRAG
  <h2>Hier kannst du etwas eintragen:</h2>
  <form name="gastForm" action="gaestebuch.php" method="POST">
    <textarea name="eintrag" cols="48" rows="8"></textarea><br>
    <input type="reset" name="reset" value="Löschen">
    <input type="submit" name="submit" value="Abschicken"
      onclick="return check();">
  </form>
EINTRAG;
  } else {
    // Phase 2
  }
?>
</body></html>
```



## Realisieren der Phase 2 (Gästebuch)

Eintrag wird nur gespeichert, wenn etwas eingetragen wurde.

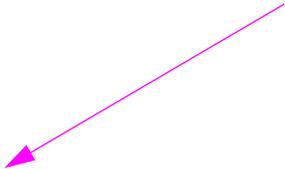
Datei öffnen, lesen und ausgeben.

(Schema 4.29)

```
if (strlen($_REQUEST["eintrag"]) >= 1) {
    $fp = fopen("gaestebuch.txt","a");
    fputs($fp, "<p>" . $_REQUEST["eintrag"] .
              "</p>\n");
    fclose($fp);
    echo "<h2>Danke für deinen Eintrag</h2>";
}

echo "<h3>Das wurde bisher eingetragen:</h3>";
$gb = fopen("gaestebuch.txt","r");
while (!feof($gb)) {
    $eintr = fgets($gb, 4096);
    echo $eintr;
}
fclose($gb);
```

Phase 2



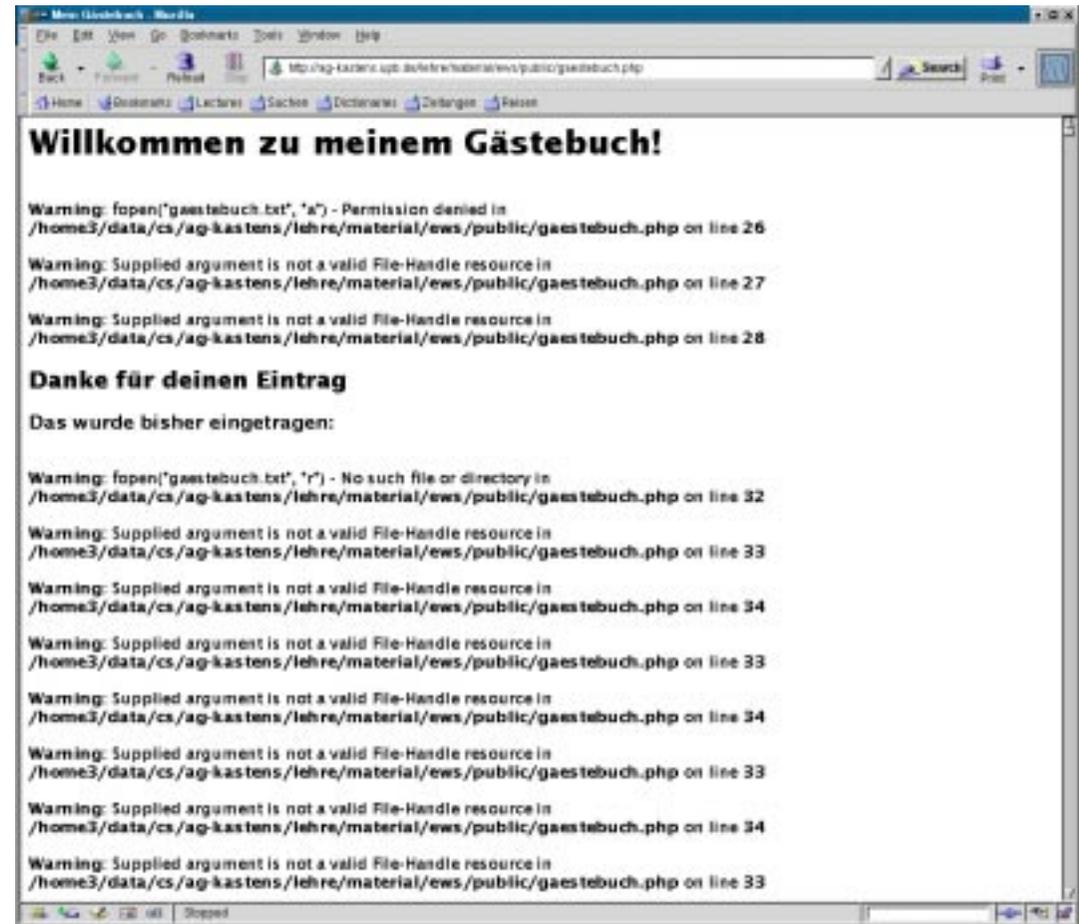
## Testen der Phase 2

### Alle Fälle prüfen:

1. normaler Eintrag in leere Datei
2. normaler Eintrag in nicht-leere Datei
3. kein Eintrag und Datei ist leer
4. kein Eintrag und Datei ist nicht leer
5. Datei existiert nicht

ggf. **Fehler finden**, korrigieren und nochmal testen

Wenn alles funktioniert, eine **andere Person** erproben lassen.



```
if (!file_exists("gaestebuch.txt")) {  
    echo "<h3>Das Gästebuch ist leider leer!</h3>";  
    echo "</body></html>";  
    exit;  
}
```

in der  
Mitte von  
Phase 2  
einfügen

# Alle Programmteile zusammen

```

<html><head>
  <title>Mein Gästebuch</title>
</head><body>
  <h1>Willkommen zu meinem Gästebuch!</h1>
  <?php
    if (!isset($_REQUEST["submit"])) {
  echo << EINTRAG
    <h2>Hier kannst du etwas eintragen:</h2>
    <form name="gastForm" action="gaestebuch.php"
      method="POST">
      <textarea name="eintrag" cols="48" rows="8">
      </textarea><br>
      <input type="reset" name="reset" value="Löschen">
      <input type="submit" name="submit" value="Abschicken"
        onclick="return check();">
    </form>
    EINTRAG;
    } else {
      if (strlen($_REQUEST["eintrag"]) >= 1) {
        $fp = fopen("gaestebuch.txt","a");
        fputs($fp, "<p>" . $_REQUEST["eintrag"] . "</p>\n");
        fclose($fp);
        echo
          "<h2>Danke für deinen Eintrag</h2>";
      }
      echo "<h3>Das wurde bisher eingetragen:</h3>";
      $gb = fopen("gaestebuch.txt","r");
      while (!feof($gb)) {
        $seintr = fgets($gb, 4096);
        echo $seintr;
      }
      fclose($gb);
    }
  ?>
</body></html>
  <script type="text/javascript">
    function check () {
      if (document.gastForm.eintrag.value.length < 1)
      { return
        confirm ("Willst Du nichts eintragen?");
      }
      return true;
    }
  </script>
  <style type="text/css">
    h1,h2,h3 {text-align:left}
    h1 {font-size:24}
    h2 {font-size:18}
    h3 {font-size:12}
  </style>
  if (!file_exists("gaestebuch.txt")) {
    echo
      "<h3>Das Gästebuch ist leider " .
      "leer!</h3></body></html>";
    exit;
  }

```

# S5 XML Übersicht

**XML** (Extensible Markup Language, dt.: Erweiterbare Auszeichnungssprache)

- seit 1996 vom W3C definiert, in Anlehnung an SGML
- Zweck: Beschreibungen **allgemeiner Strukturen** (nicht nur Web-Dokumente)
- **Meta-Sprache** (“erweiterbar”):  
Die Notation ist festgelegt (Tags und Attribute, wie in HTML),  
Für beliebige Zwecke kann **jeweils eine spezielle syntaktische Struktur** definiert werden (DTD)  
Außerdem gibt es Regeln (XML-Namensräume), um XML-Sprachen in andere **XML-Sprachen zu importieren**
- **XHTML** ist so als XML-Sprache definiert
- Weitere aus XML **abgeleitete Sprachen**: SVG, MathML, XSL-FO, SMIL, WML
- **individuelle XML-Sprachen** werden benutzt, um strukturierte Daten zu speichern, die von **Software-Werkzeugen geschrieben und gelesen** werden
- XML-Darstellung von strukturierten Daten kann mit verschiedenen Techniken **in HTML transformiert** werden, um sie **formatiert anzuzeigen**:  
XML+CSS, XML+XSL, SAX-Parser, DOM-Parser

Dieses Kapitel orientiert sich eng an **SELFHTML** (Stefan Münz), <http://de.selfhtml.org/xml/intro.htm>

# Notation und erste Beispiele

Ein Satz in einer XML-Sprache ist ein Text, der durch Tags strukturiert wird.

Tags werden **immer** in **Paaren von Anfangs- und End-Tag** verwendet:

```
<ort>Paderborn</ort>
```

Anfangs-Tags können Attribut-Wert-Paare enthalten:

```
<telefon typ="dienst">05251606686</telefon>
```

Die **Namen von Tags und Attributen** können für die XML-Sprache **frei gewählt** werden.

Mit Tags gekennzeichnete Texte können geschachtelt werden.

```
<adressBuch>
<adresse>
  <name>
    <nachname>Kastens</nachname>
    <vorname>Uwe</vorname>
  </name>
  <anschrift>
    <strasse>Wiesenweg 37</strasse>
    <ort>Paderborn</ort>
    <plz>33106</plz>
  </anschrift>
</adresse>
</adressBuch>
```

$(a+b)^2$  in MathML:

```
<msup>
  <mfenced>
    <mrow>
      <mi>a</mi>
      <mo>+</mo>
      <mi>b</mi>
    </mrow>
  </mfenced>
  <mn>2</mn>
</msup>
```

# Ein vollständiges Beispiel

Kennzeichnung des Dokumentes als XML-Datei

Datei mit der Definition der Syntaktischen Struktur dieser XML-Sprache (DTD)

Datei mit Angaben zur Transformation in HTML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE produktnews SYSTEM "produktnews.dtd">
<?xml-stylesheet type="text/xsl" href="produktnews.xsl" ?>
<produktnews>
  Die neuesten Produktnachrichten:
  <beschreibung>
    Die Firma <hersteller>Fridolin Soft</hersteller> hat eine neue
    Version des beliebten Ballerspiels
    <produkt>HitYourStick</produkt> herausgebracht. Nach Angaben des
    Herstellers soll die neue Version, die nun auch auf dem
    Betriebssystem <produkt>Ganzfix</produkt> läuft, um die
    <preis>80 Dollar</preis> kosten.
  </beschreibung>
  <beschreibung>
    Von <hersteller>Ripfiles Inc.</hersteller> gibt es ein Patch zu der Sammel-CD
    <produkt>Best of other people's ideas</produkt>. Einige der tollen
    Webseiten-Templates der CD enthielten bekanntlich noch versehentlich nicht
    gelöschte Angaben der Original-Autoren. Das Patch ist für schlappe
    <preis>200 Euro</preis> zu haben.
  </beschreibung>
</produktnews>
```

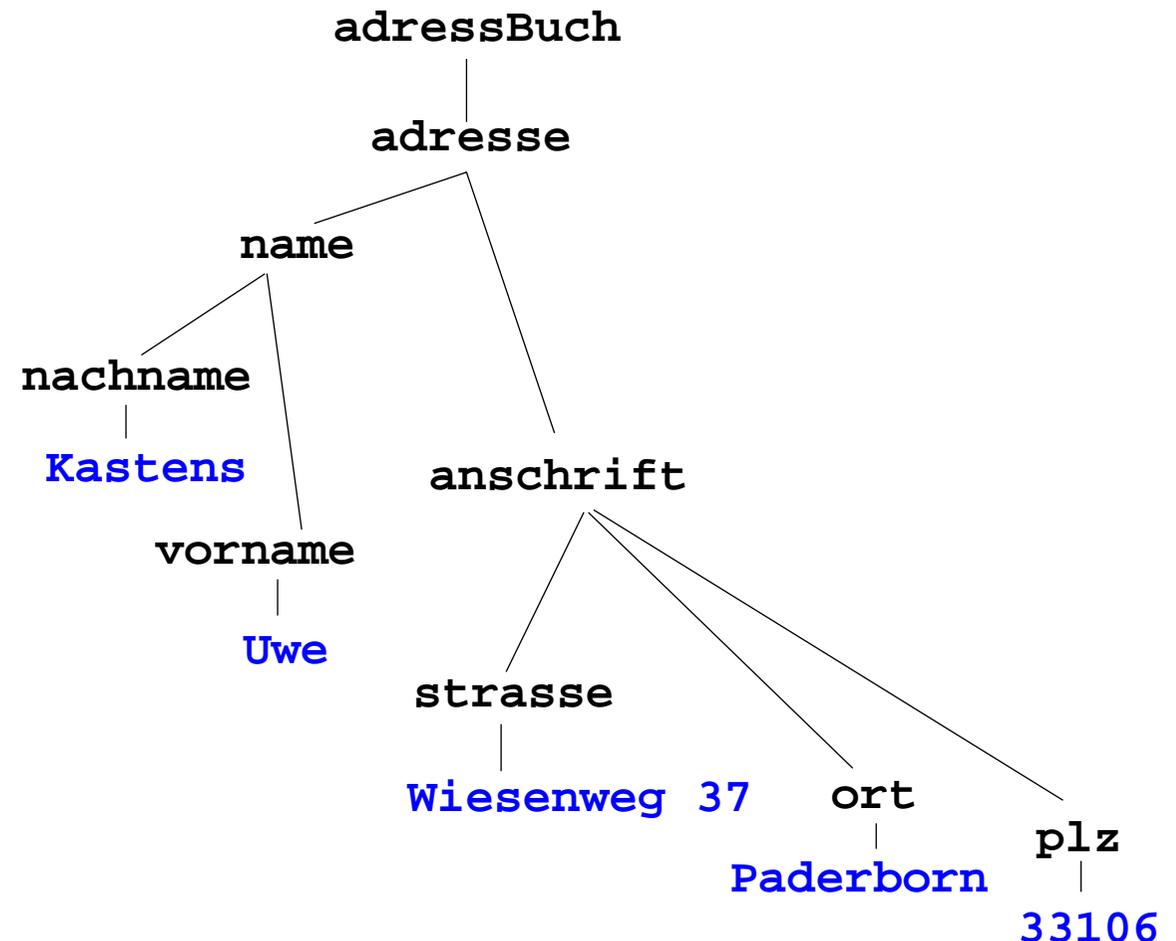
# Baumdarstellung von XML-Texten

Jeder XML-Text ist durch Tag-Paare **vollständig geklammert** (wenn er *wohldefiniert* ist).

Deshalb kann er eindeutig **als Baum dargestellt** werden. (Attribute betrachten wir noch nicht)

Wir markieren die inneren Knoten mit den Tag-Namen; die **Blätter** sind die elementaren Texte:

```
<adressBuch>
<adresse>
  <name>
    <nachname>Kastens
    </nachname>
    <vorname>Uwe
    </vorname>
  </name>
  <anschrift>
    <strasse>Wiesenweg 37
    </strasse>
    <ort>Paderborn</ort>
    <plz>33106</plz>
  </anschrift>
</adresse>
</adressBuch>
```



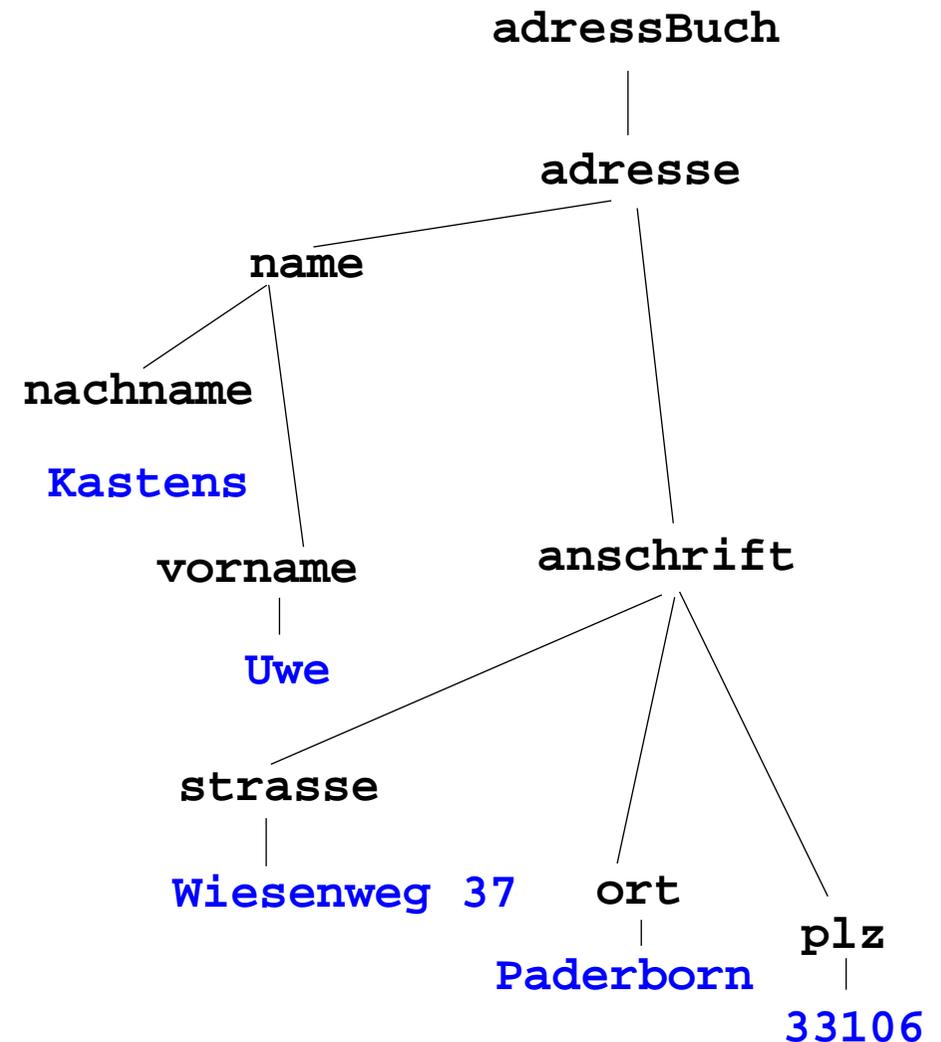
XML-Werkzeuge können die Baumstruktur eines XML-Textes ohne weiteres ermitteln und ggf. anzeigen.

# Grammatik definiert die Struktur der XML-Bäume

Mit **kontextfreien Grammatiken (KFG)** kann man **Bäume** definieren.

Folgende KFG definiert korrekt strukturierte Bäume für das Beispiel Adressbuch:

```
adressBuch ::= adresse*  
adresse    ::= name anschrift  
name       ::= nachname vorname  
Anschrift  ::= strasse ort plz  
nachname   ::= PCDATA  
vorname    ::= PCDATA  
strasse    ::= PCDATA  
ort        ::= PCDATA  
plz        ::= PCDATA
```



# Document Type Definition (DTD) statt KFG

Die Struktur von XML-Bäumen und -Texten wird in der **DTD-Notation** definiert. Ihre Konzepte entsprechen denen von KFGn:

## KFG

```

adressBuch ::= adresse*
adresse    ::= name anschrift
name      ::= nachname vorname
Anschrift ::= strasse ort plz
nachname  ::= PCDATA
vorname   ::= PCDATA
strasse   ::= PCDATA
ort       ::= PCDATA
plz      ::= PCDATA
  
```

## DTD

```

<!ELEMENT adressBuch(adresse)*      >
<!ELEMENT adresse    (name, anschrift) >
<!ELEMENT name      (nachname, vorname)>
<!ELEMENT anschrift (strasse, ort, plz)>
<!ELEMENT nachname  (#PCDATA)          >
<!ELEMENT vorname   (#PCDATA)          >
<!ELEMENT strasse   (#PCDATA)          >
<!ELEMENT ort       (#PCDATA)          >
<!ELEMENT plz      (#PCDATA)          >
  
```

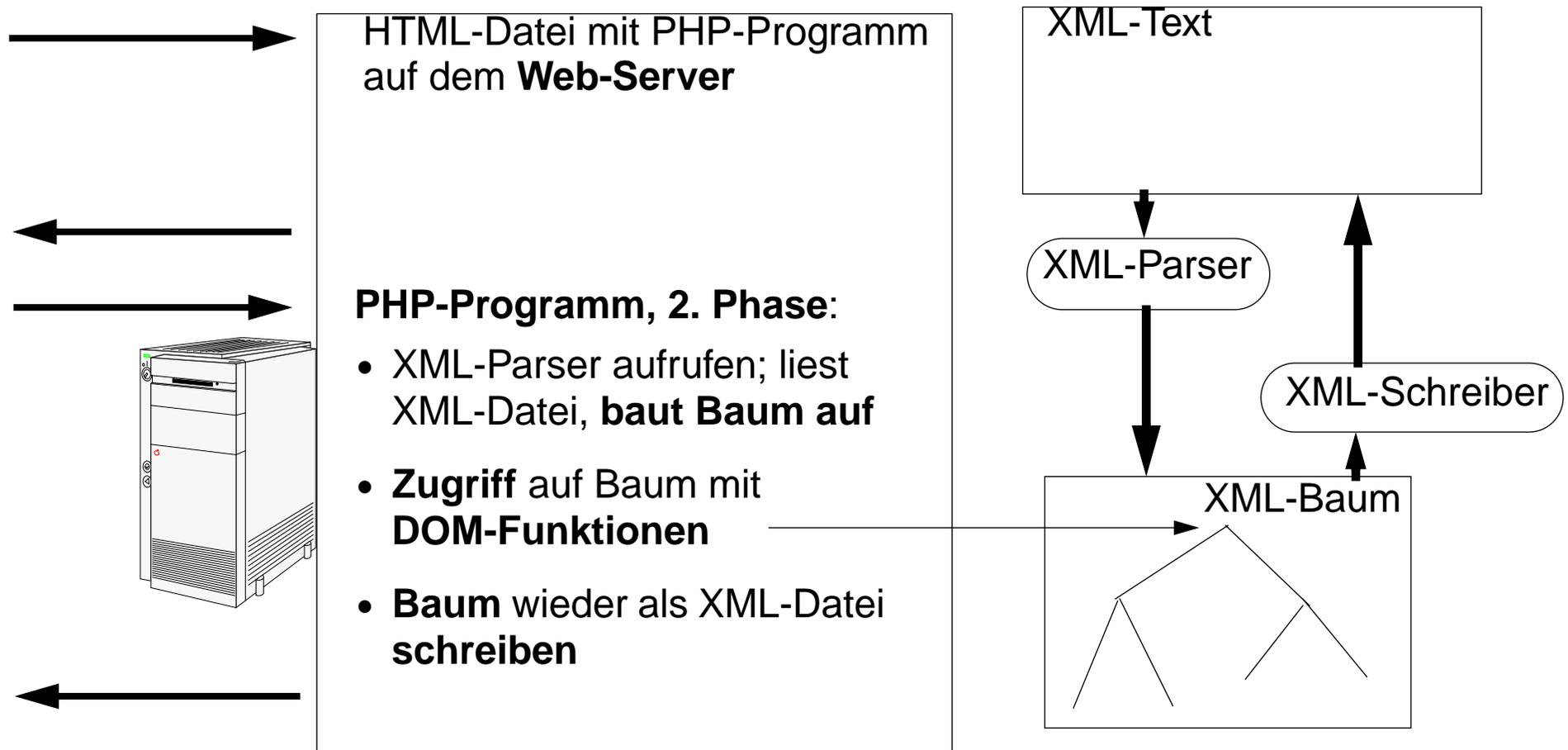
## weitere Formen von DTD-Produktionen:

<b>X (Y)+</b>	nicht-leere Folge
<b>X (A   B)</b>	Alternative
<b>X (A)?</b>	Option
<b>X EMPTY</b>	leeres Element

# XML-Datei als Speicher für strukturierte Daten

Ein **Server-Programm** benutzt eine XML-Datei als Speicher für strukturierte Daten; **liest, ändert und schreibt** sie zurück.

**Anwendungsbeispiel:** Eine Web-Seite sammelt Adressen von potentiellen Kunden. Die 2. Phase des PHP-Programms **trägt Adressen ein** und/oder **zeigt vorhandene an**.



# Präsentation von XML-Daten am Beispiel

**XML-Strukturen** enthalten selbst **keine Information**, die zum **Layout** der Elemente bei einer **Präsentation im Browser** verwendet werden könnte.

Es gibt mehrere Techniken, um **XML-Strukturen zu präsentieren**:

- Der **Browser zeigt den XML-Text mit Tags an** und hebt die Schachtelung hervor - ohne weitere Information möglich
- Den Tags werden mit **CSS Stylesheets Layout-Informationen** zugeordnet.
- Die **XML-Struktur wird in HTML transformiert**, spezifiziert durch **XSL**
- Die **Transformation nach HTML wird in PHP programmiert**, zum Strukturieren des XML-Textes wird ein Parser benutzt.

# Beispiel: Produktinformationen

An diesem Beispiel werden 3 der 4 **Präsentationstechniken** gezeigt. (aus SELFHTML)

## Anwendung:

**Notizen über Produktinformationen** werden in einer Firma als **strukturierte Texte in XML** formuliert, gespeichert, präsentiert und von Werkzeugen verarbeitet (z. B. Mail, News).

## DTD:

```
<!ELEMENT produktnews (#PCDATA | beschreibung)*>
<!ELEMENT beschreibung (#PCDATA | hersteller | produkt | preis)*>
<!ELEMENT hersteller (#PCDATA)>
<!ELEMENT produkt (#PCDATA)>
<!ELEMENT preis (#PCDATA)>
```

## Text-Beispiel:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE produktnews SYSTEM "produktnews.dtd">
<produktnews>Die neuesten Produktnachrichten:
```

```
<beschreibung>
```

```
Die Firma <hersteller>Fridolin Soft</hersteller> hat eine neue Version des beliebten Ballerspiels
<produkt>HitYourStick</produkt> herausgebracht. Nach Angaben des Herstellers soll die neue
Version, die nun auch auf dem Betriebssystem <produkt>Ganzfix</produkt> läuft, um die
<preis>80 Dollar </preis> kosten.
```

```
</beschreibung>
```

```
<beschreibung>
```

```
Von <hersteller>Ripfiles Inc.</hersteller> gibt es ein Patch zu der Sammel-CD <produkt>Best of
other people's ideas</produkt>. Einige der tollen Webseiten-Templates der CD enthielten
bekanntlich noch versehentlich nicht gelöschte Angaben der Original-Autoren. Das Patch ist für
schlappe <preis>200 Euro</preis> zu haben.
```

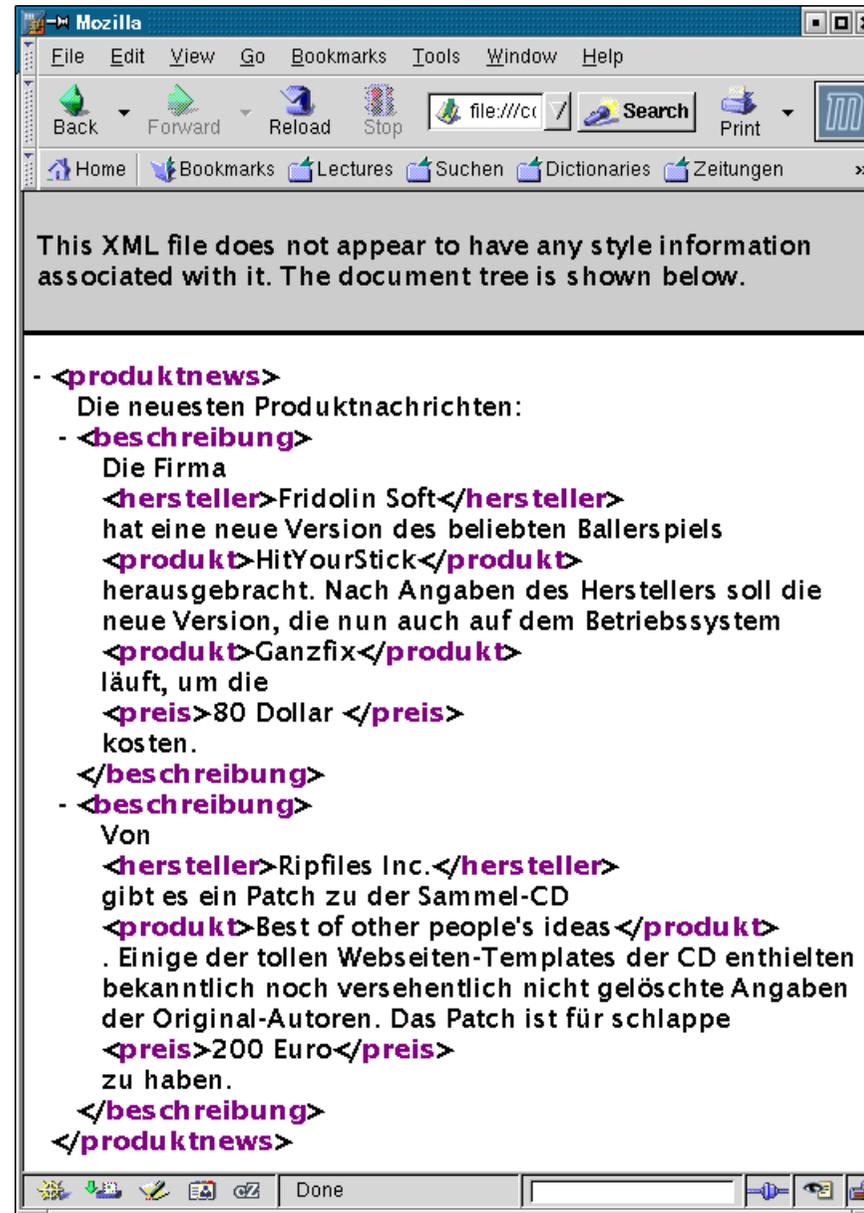
```
</beschreibung>
```

```
</produktnews>
```

# XML-Text direkt präsentiert

## Der Browser

- hebt die Tags hervor,
- zeigt Folgen an,
- zeigt Schachtelung an,
- wendet Strategie für Zeilenumbrüche an.



# XML-Text formatiert mit CSS-Stylesheet

## produktnews

```
{ position:absolute;
  top:10pt;
  left:40pt;
  font-family:sans-serif;
  font-size:18pt;
}
```

## beschreibung

```
{ position:relative;
  display:block;
  width:300px;
  font-size:14pt;
  margin-top:20pt;
  margin-bottom:20pt;
}
```

## hersteller

```
{ font-weight:bold;
  color:blue;
}
```

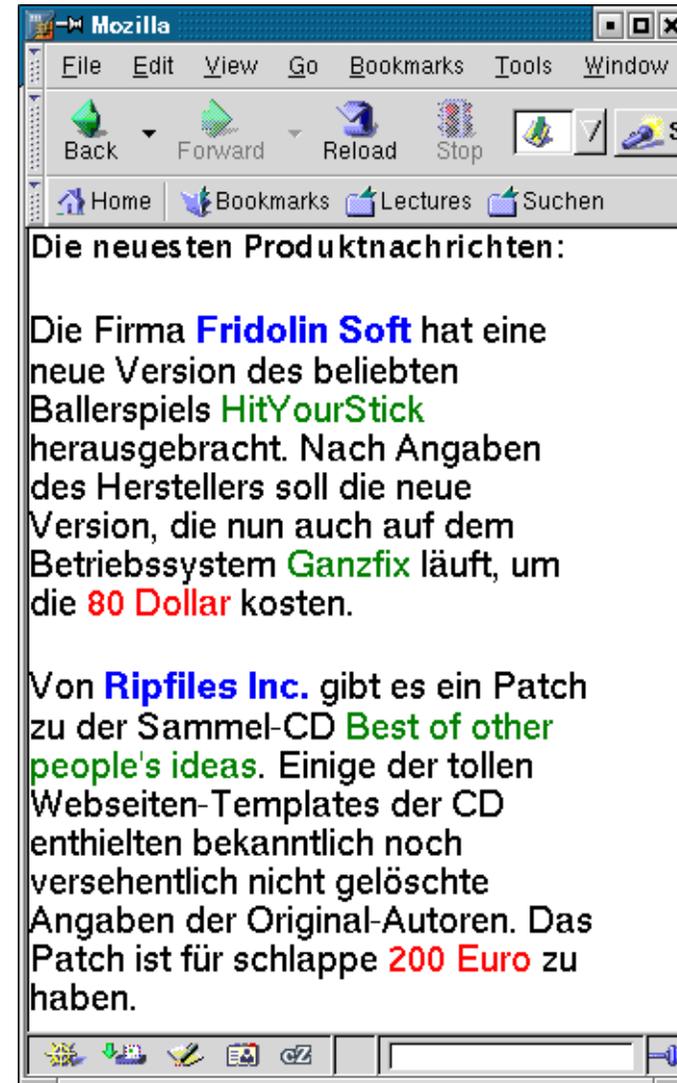
## produkt

```
{ color:green; }
```

## preis

```
{ color:red; }
```

CSS-Stylesheet ordnet den verwendeten **XML-Tags** Formatierangaben zu. Der Browser wendet sie an:



# XML-Text mit XSL in HTML transformiert

```

<xsl:template match="/">
  <html><head>
    <style type="text/css">
      .titel {font-family:sans-serif;
              font-size:18pt; color:green;}
      .absatz {font-family:sans-serif;
              font-size:10pt; color:black;}
      .rot {font-weight:bold; color:red;}
      .blau {font-weight:bold; color:blue;}
    </style></head>
    <body><div class="titel">
      <xsl:apply-templates />
    </div></body></html>
</xsl:template>

<xsl:template match="beschreibung">
  <div class="absatz"><p>
    <xsl:apply-templates />
  </p></div>
</xsl:template>

<xsl:template match="hersteller">
  <span class="rot">
    <xsl:value-of select="." />
  </span>
</xsl:template>
...
<xsl:template match="preis">
  <b><xsl:value-of select="." /></b>
</xsl:template>

</xsl:stylesheet>

```

Ein **XSL-Template** beschreibt die Transformation eines **XML-Elementes** oder der **Baumwurzel** in einen HTML-Text.

Der Inhalt des Elementes wird

- **übersetzt und eingesetzt** oder
- **unverändert eingesetzt**.

