

Letzte Woche

Themen der letzten Vorlesung:

- Datentypen in PHP
- Syntax von PHP-Programmen
- Variablen, Funktionen
- Automatische Typkonversion
- Arrays

Offene Fragen...

Unterschiede zwischen `echo` und `print`:

- `echo` ist ein Sprachkonstrukt, `print` eine Funktion
- Bei `echo` darf man die Klammern weglassen:
`echo "Hallo Welt!";`
- `print` hat einen Rückgabewert (immer 1):
`$res = print("Hallo");`
- Für `print` gelten die Präzedenzregeln

Modulare Programmierung mit PHP

PHP hat kein Modulsystem. Globale Deklarationen sind überall sichtbar.

Einbinden von Dateien:

- `require(filename_expr);` und `include(filename_expr);`
- `filename_expr`: Ausdruck, der zu einem Dateinamen ausgewertet wird
- `require` wirft fatalen Fehler, wenn Datei nicht gefunden
- `include` gibt Warnung aus, wenn Datei nicht gefunden
- Nur einmal während der Ausführung einbinden: `require_once`, `include_once`

PHP und HTTP

HTTP-Anfrage und Umgebung sind über vordefinierte superglobale Variablen zugänglich.

Wichtige vordefinierte Variablen:

- \$_COOKIES** Array mit allen Cookies der Anfrage
- \$_GET** dekodierte Information aus **QUERY_STRING** als Feld
- \$_POST** dekodierter Rumpf der Anfrage als Feld
- \$_SESSION** Sessiondaten
- \$_REQUEST** Alles aus **\$_GET**, **\$_POST** und **\$_COOKIES**
- \$_SERVER** Infos über HTTP-Server

Variablen aus der Anfrage

Übergebene Variablen automatisch als globale Variablen:

- Vor 4.2 Standard, jetzt standardmäßig abgeschaltet
- Sicherheitsrisiko

Alternative `import_request_variables(types [, prefix])`

- Mögliche Werte für `types`: {"g","p","c"}
- Macht globale Variablen für Werte aus **GET**, **POST** und **COOKIE**
- Variablen heißen wie Eingabewidget/Cookie (evt. mit Präfix)

Beispiel: Query-Teil `?postleitzahl=21244`

```
import_request_variables("gp", "pre_");
```

⇒ `$pre_postleitzahl` hat den Wert `"21244"`

HTTP-Header setzen

HTTP-Header in der Antwort setzen:

```
header(string header_line [, bool replace [, int code]])
```

Beispiel: `header("Content-Type: text/html");`

- Wird ein `Location`-Header gesetzt, setzt PHP den Status-Code automatisch auf 302
- Liefert Fehler, wenn Ausgabe des Rumpfes schon begonnen hat
- `header()` sollte also immer als erstes erfolgen
- Alternative: Ausgabe puffern

Cookies

Cookie setzen:

```
bool  
setcookie(string name [, string value [, int expire  
          [, string path [, string domain [, bool secure]]]])
```

Der String `value` wird automatisch kodiert

Beispiel:

```
<?php  
if (isset($_COOKIE["login"]))  
    print("Welcome back, " . ($_COOKIE['login']));  
else{  
    setcookie("login", "anonymous user");  
    print("You are logged in as an anonymous user");  
} ?>
```

Sessionverwaltung

PHP hat eingebautes Sessionmanagement:

- `session_start()` startet eine neue Session
- Interpreter sendet Session-ID (Cookie oder Query-Teil)
- Sessiondaten werden in `$_SESSION` abgelegt
- `$_SESSION` in Datei speichern und später wieder lesen

Speichern der Daten mit `serialize()`:

- `serialize(val)` liefert Stringrepräsentation von `val`
- `unserialize(str)` liest Stringrepräsentation und gibt Wert zurück

PHP und HTML

PHP-Code kann mit HTML-Code vermischt werden:

```
<?php for ($i=0; $i<10; $i++){ ?>  
<BR>  
<?php } ?>
```

In alternativer Syntax:

```
<?php for ($i=0; $i<10; $i++): ?>  
<BR>  
<?php endfor; ?>
```

Kurzform für Ausgabe: `<?= <expr> [, <expr>] ?>`

```
<?php for ($i=0; $i<10; $i++) : ?>  
  <?= "<a href=\"/count.php?no=$i\">$i</a>\n" ?>  
<?php endfor; ?>
```

Dabei wird `<?= expr ?>` in `<?php echo expr; ?>` übersetzt

Eingabefelder und Arrays

PHP liest HTML-Eingabefelder als Array, wenn der Name mit [] endet:

- Endet mit "[]": Index wird automatisch erzeugt (von Null an)
- Endet mit "[x]": x wird als Index verwendet

Beispiel `html_array.php`:

```
<?php
$menu = array("Pizza", "Lasagne", "Spaghetti", "Panna Cotta");

if ($_GET['action']=="submitted"):
    echo "Ihre Bestellung:<BR>";

    $client = $_GET['personal'];
    echo "Name:", $client["name"],"<BR>";
    echo "Straße:", $client["street"],"<BR>";

    $dishes = $_GET['dishes'];
    for($i = 0; $i < count($dishes); $i++)
        echo $dishes[$i].", "<BR>";
```

html_array.php (Fortsetzung)

```
else : ?>
<FORM action=<?=$_SERVER['SCRIPT_NAME']?> method="get">

Name: <INPUT type="text" name="personal[name]"><br>
Straße: <INPUT type="text" name="personal[street]"><br>

<P> Bitte Gerichte auswählen: </P>
<SELECT multiple name="dishes[]">

<?php for($i=0;$i<count($menu);$i++): ?>

<OPTION value=<?=$i?>> <?=$menu[$i]?> </OPTION>

<?php endfor; ?>

</SELECT><BR>
<INPUT type="hidden" name="action" value="submitted">
<INPUT type="submit" name="submit" value="Abschicken!">
</FORM>

<?php endif; ?>
```

Webprogrammierung mit Continuations

Kontrolle und Web-Applikationen

Wenn HTTP verbindungsorientiert wäre:

```
send-page( compute-result( parse-request( send-page/suspend(page) ) ) ) )
```

Ist es aber nicht, daher:

```
bindings = parse_request(request);  
step = extract(bindings, "step");
```

```
if (step == 0) then  
    send_page(page);
```

```
else if (step == 1) {  
    arguments = extract(bindings, "arguments");  
    send_page(compute_page(results));  
}
```

⇒ Warten auf nächste Anfrage findet implizit nach Programmende statt

Kontrolle und Web-Applikationen

Typisch für Code von Web-Applikationen (bisher):

- Schlecht lesbar (Kontrolle)
- Schwerer zu programmieren
- Fehleranfällig

⇒ Ziel: Programmieren wie im ersten Ansatz gesehen, aber über HTTP kommunizieren.

Kontext

Betrachte Auswertung von:

```
send-page( compute-result( parse-request( send-page/suspend( page ) ) ) )
```

Nach der Ausführung von `send-page/suspend(page)` fehlt noch die Auswertung von

```
send-page( compute-result( parse-request( ... ) ) )
```

Die ist der *Kontext* von `send-page/suspend(page)`

Laufzeitrepräsentation für Kontext: *Continuation*

send-page/suspend

Webprogrammierung mit Continuations:

- Continuation von `send-page/suspend` abspeichern
- Seite enthält Referenz auf Continuation, Seite senden

Bei nächster Anfrage:

- Server überprüft, ob Anfrage Referenz auf Continuation enthält
- Wenn ja: Wende Continuation auf Anfrage an

Dieser Ablauf wird von `send-page/suspend` implementiert.

Voraussetzungen

Um `send-page` / `suspend` zu implementieren braucht man:

- Programmiersprache mit Zugriff auf Continuations. Hier: Scheme
- Webserver, der Continuations anwenden kann (z. B. in Scheme programmiert)

Zugriff auf Continuations in Scheme mit
`call-with-current-continuation` kurz `call/cc`

Scheme

Einige Eigenschaften von Scheme:

- Sehr einfache Syntax, Präfixnotation
- Wenige Sprachkonstrukte
- Lexikalische Bindung
- Proper tail recursion
- Funktionen als Werte erster Klasse
- Zugriff auf Continuations
- Dynamisch getypt
- Hygienisches Makrosystem
- Formale Semantik
- Sprachkonstrukte für imperative Programmierung

Syntax

Vollständig geklammerte Präfixnotation. Allgemein:

(operator operands*)

Klammern haben *immer* eine Bedeutung:

1 ≠ (1) und (+ 1 2) ≠ ((+ 1 2))

⇒ Immer erst auf den Operator schauen

Einzeilige Kommentare: **;** **Kommentartext**

Primitive Datentypen

Ganzzahlen

`1 -2 23`

unbeschränkt groß

Fließkommazahlen

`1.2 3.0 -4.56`

Rationale Zahlen

`1/2 4/3`

automatisches Kürzen,
Umwandlung in Ganzzahlen

Booleans

`#t #f`

Bei Test gilt: Alles außer `#f` ist wahr

Strings

`"abc" "123" "\"Hallo\""\n"`

Zeichen

`#\a #\A #\space #\newline`

Pragmatik

Es gibt eine Vielzahl von Compilern und Interpretern.

Benutzung:

```
[0 knauel@albert ~] scheme48
Welcome to Scheme 48 1.3
Copyright (c) 1993-2005 by Richard Kelsey and Jonathan Rees.
Please report bugs to scheme-48-bugs@s48.org.
Get more information at http://www.s48.org/.
Type ,? (comma question-mark) for help.
> 42
42
> (* 6 7)
42
> (- 42/3 17/9)
109/9
>
```

Definitionen und Variablen

Definition von Variablen:

```
(define var expr)
```

Bindet Wert von `expr` an `var`. `var` ist gesamten Programm (global) sichtbar.

Beispiele:

```
(define a 1)
```

```
(define hallo "Hallo")
```

```
(define ein-a #\a)
```

Variablen

Scheme ist dynamisch getypt. Variablen haben keinen Typ.

Fast keine Typkonversion. Aber:

- Alles außer **#f** ist wahr
- Arithmetische Operatoren nehmen meist beliebige Zahlen als Argumente

Regeln für Variablennamen:

- Kein Unterschied zwischen Groß- und Kleinschreibung
- Dürfen auch -, ?, !, +, : enthalten
- Operatornamen sind auch nur Variablennamen

Einfache Operatoren

Arithmetisch 1 + - * /

Liefern Fließkommazahl, wenn ein Argument eine Fließkommazahl ist

Arithmetisch 2 `quotient modulo remainder`

Vergleiche = < > <= >=

Für Strings `string-ref string-set!`
`string-length substring string=?`

Zeichen `integer->char char->integer`
`char=? char<? ...`

Bedingte Auswertung mit `if`

Bedingte Auswertung mit `if`:

```
(if test-expr conseq-expr alt-expr)
```

Wert des `if`-Ausdrucks ist Wert von `conseq-expr` wenn `test-expr` nicht `#f` ist, sonst Wert von `alt-expr`.

Beispiel:

```
(+ 1 (if (> x 0) x (- x)))
```

Bedingte Auswertung mit cond

Allgemeinere Form für bedingte Auswertung:

```
(cond (test-expr res-expr)
      ...
      (else res-expr))
```

Beispiel:

```
(cond ((< x 0) "negativ")
      ((= x 0) "null")
      (else "positiv"))
```

Funktionen

`lambda` erzeugt Funktionen:

```
(lambda (var*) body-expr+)
```

- Formale Parameter `var`, im Rumpf sichtbar (lokale Variablen)
- Rückgabewert ist Wert des letzten `body-expr`

Funktionen an Namen binden:

```
(define add-1  
  (lambda (n)  
    (+ n 1)))
```

Rekursion

Beispiel mit Rekursion:

```
(define fakultaet
  (lambda (n)
    (if (= n 0)
        1
        (* n (fakultaet (- n 1))))))
```

Kurzform für Definition von Funktionen:

```
(define (fakultaet n)
  (if (= n 0)
      1
      (* n (fakultaet (- n 1)))))
```

Paare

Paare sind die Bausteine für Listen:

- Bestehen aus zwei beliebigen Werten
- Erzeugt mit `cons`
- Erster Wert in einem Paar heißt `car`
- Zweiter Wert in einem Paar heißt `cdr`
- Externe Repräsentation: (`<Wert car>` . `<Wert cdr>`)

Beispiel:

```
> (cons 1 2)
'(1 . 2)
> (cons (+ 1 2) "bla")
'(3 . "bla")
> (car (cons 1 2))
1
```