

# Letzte Woche

Themen der letzten Vorlesung:

- Web-Applikationen mit PHP
- Sessionverwaltung
- PHP und HTML
- Eingabefelder und Arrays
- Webprogrammierung mit Continuations
- Kontext und Continuations
- Scheme: Syntax, Datentypen, Variablen, Operatoren
- Scheme: Funktionen, Rekursion, Paare

# Punkt-Klammer-Zap-Regel

Punkt-Klammer-Zap-Regel:

*Wenn in der externen Repräsentation ein Punkt von einer öffnenden Klammer gefolgt wäre, so werden sowohl Punkt als auch Klammerpaar weggelassen.*

Beispiele:

```
> (cons (cons 1 2) 3)
'( (1 . 2) . 3)
```

```
> (cons 1 (cons 2 3))
'(1 . (2 . 3)) ⇒ '(1 2 . 3)
```

# Listen

Listen sind die wichtigste Datenstruktur in Scheme. Induktiv definiert, rekursiv verarbeitet.

Eine Liste ist entweder

- die leere Liste '()
- ein Paar, dessen `cdr` eine Liste ist

Beispiel:

```
> '()  
'()  
> (null? '())  
#t  
> (cons 1 '())  
'(1)  
> (cons 1 (cons 2 '()))  
'(1 2)
```

## Funktionen auf Listen: map

`map` wendet Funktion auf jedes Element der Liste an:

```
> (map string-length (list "a" "b" "ab" ""))  
'(1 1 2 0)
```

```
> (map (lambda (x) (+ x 1)) (list 1 22 41))  
'(2 23 42)
```

Definition von `map`:

```
(define (map f lst)  
  (if (null? lst)  
      '()  
      (cons (f (car lst))  
            (map f (cdr lst))))))
```

# Funktionen auf Listen: filter

`filter` filtert eine Liste mittels eines Prädikates:

```
> (filter even? (list 1 23 42))  
'(42)  
> (filter pair? (list (cons 1 2) 2 (cons 4 5) '()))  
'((1 . 2) (4 . 5))
```

Definition:

```
(define (filter p lst)  
  (cond ((null? lst)  
        '())  
        ((p (car lst))  
         (cons (car lst) (filter p (cdr lst))))  
        (else  
         (filter p (cdr lst)))))
```

## Funktionen auf Listen: fold-right

`fold-right` ist die nützlichste Higher-Order-Funktion auf Listen.

`fold-right` ersetzt die Konstruktoren einer Liste.

```
(define (fold-right proc unit l)
  (if (null? l)
      unit
      (proc (car l)
            (fold-right proc unit (cdr l)))))
```

Beispiele:

```
(define (map f lst)
  (fold-right (lambda (h t)
                (cons (f h) t))
              '() lst))
> (fold-right + 0 (list 1 2 3 4 5))
15
```

# Symbole

Weiterer Datentyp: *Symbole*

- Objekt mit Namen
- Sinn: Gleich wenn Namen gleich
- Der Name dient als externe Repräsentation
- Literal: `'name'`

Operationen:

- Typtest mit `symbol?`
- Umwandlung in String: `symbol->string` `string->symbol`
- Test auf Gleichheit mit `eq?`

# Quote

`(quote <externe repräsentation>)`  $\Rightarrow$  Literal

Kurzform: `'<externe repräsentation>`

Quote ist besonders hilfreich, um Listen (oder Code) im Code aufzuschreiben.

```
> (quote a)
```

```
'a
```

```
> (quote 1)
```

```
1
```

```
> (quote "abc")
```

```
"abc"
```

```
> (quote (+ 1 2))
```

```
'(+ 1 2)
```

```
> (quote (1 . 2))
```

```
'(1 . 2)
```

```
> '(1 2 3)
```

```
'(1 2 3)
```

# Quasiquote

Problem mit `quote`:

Die gesamte Liste muß schon beim Schreiben des Programms bekannt sein.

Abhilfe schafft `quasiquote` mit Kurzform: ```

Arbeitet wie `quote` aber für ein `(unquote e)` oder `,e` in der externen Repräsentation wird `e` ausgewertet und eingesetzt.

```
> (quasiquote (1 2 3))
'(1 2 3)
> (quasiquote (1 2 (+ 2 1)))
'(1 2 (+ 2 1))
> (quasiquote (1 2 (unquote (+ 2 1))))
'(1 2 3)
> `(1 2 ,(+ 2 1))
'(1 2 3)
```

# Unquote-Splicing

Unquote-Splicing:

`(unquote-splicing e)` bzw. `,@e` wertet den Ausdruck `e` innerhalb eines `quasiquote` aus.

Das Ergebnis muß eine Liste sein, die in die umgebende Liste eingebaut wird.

```
> `(1 ,@(map - (list 1 2)) 7)
'(1 -1 -2 7)
```

```
> `(1 ,(list 2 3) ,@(list 4 5))
'(1 (2 3) 4 5)
```

# Zuweisung

Zuweisung:

```
(set! var expr)
```

Setzt die Variable `var` auf den Wert des Ausdrucks `expr`. `var` muß gebunden sein.

```
> (define a 1)
> (set! a (+ a 10))
> a
11
```

Echte Scheme-Programmierer verwenden Zuweisungen nur sehr selten!

# Syntaktischer Zucker

Syntaktischer Zucker:

- Lokale Bindung mit `let`:

```
(let ((var-1 expr-1)      (let ((x 1)
    ... )                (y 2))
    expr+ )              (+ x y))
```

- Lokale Bindung, sequentiell mit `let*`:

```
(let* ((var-1 expr-1)    (let* ((x 1)
    ... )                (y (+ x 5)))
    expr+ )              (+ x y))
```

- Hintereinanderausführung mit `begin`:

```
(begin expr+ )      (begin 1 2 3) ⇒ 3
```

# Continuations einfangen

`call-with-current-continuation` bzw. `call/cc` verwandelt momentane Continuation in Prozedur

`(call-with-current-continuation proc)`

- `proc` ist eine Prozedur, die ein Argument nimmt
- Momentane Continuation wird in Escape-Prozedur `esc` gesichert
- Aufruf der übergebenen Prozedur mit `esc` als Argument:  
`(proc esc)`

Was passiert, wenn die Escape-Prozedur `esc` aufgerufen wird?

# Escape-Prozedur

Der Aufruf einer von `call/cc` erstellten Escape-Prozedur bewirkt:

- Momentane Continuation wird verworfen
- Die in `esc` gesicherte Continuation wird wiederhergestellt
- Dort, wo `call/cc` war klafft ein "Loch"
- `esc` wird mit einem Argument aufgerufen
- Das Argument füllt dieses Loch

# Beispiel: Ein Mini-Exception-System

```
(define (with-handler handler proc)
  (call-with-current-continuation
    (lambda (esc)
      (proc (lambda (reason)
              (call-with-current-continuation
                (lambda (restart)
                  (esc (handler reason restart))))))))))

(with-handler
  (lambda (reason restart)
    (if (eq? reason 'failure)
        (display "Something went wrong")
        (restart 'ignore)))
  (lambda (throw)
    (display "1")
    (throw 'annoyed)
    (display "2")
    (throw 'failure)
    (display "3"))))
```

# HTML in Scheme

HTML-Code wird in Scheme oft durch Listen repräsentiert:

`<tag> ... </tag>` → `'(tag ...)`

`<tag attr="val"> ... </tag>` → `'(tag (@ (attr "val")) ...)`

Vorteile:

- Mehr Struktur
- Schließendes Tag überflüssig

Beispiel:

```
(html
  (head (title "Eine Seite"))
  (body
    (form (@ (action "...") (method "POST"))
      (input (@ (type "submit") (value "submit"))))))
```

# Continuationbasierter Server

Wir verwenden den SUnet (Scheme Untergrund) Web-Server

- Lauffähig unter scsh (Scheme Shell)
- Läuft unter UNIX und Windows (Cygwin)
- Viel Dokumentation, viele Beispiele

Links und Informationen finden sich auf der Homepage

# SUrfllets

Continuationbasierte Web-Applikationen in SUnet heißen *SUrfllets* und leben in Modulen.

Sprache zur Definition von Modulen:

```
(define-interface interface-name
  (export name-1 name-2 ...))
(define-structure module-name interface
  (open module-1 module-2)
  (begin ...))
```

Der Code kann auch mit `(file "dateiname")` aus einer Datei geladen werden.

# Beispiel: Erstes SURflet

```
(define-structure surflet surflet-interface
  (open surflets scheme-with-scsh)
  (begin

    (define (main req)
      (send-html/finish
        '(html
          (body (h1 "Hello World")
                (p "The current time is"
                  ,(format-date "H:M:S p m/d/Y" (date))))))))))
))
```

# SUrlet API

`send-html/suspend` entspricht dem `send-page/suspend`:

- Continuation abspeichern
- Referenz auf Continuation als Argument übergeben
- Seite ausgeben und beenden

Weitere Funktionen

- `(get-bindings req)` extrahiert Bindings aus Request
- `(extract-single-binding key binding)` extrahiert einen Wert aus den Bindings
- `(send-html/finish sxml)` schickt letzte Nachricht an Client

# Formulare und Eingabewidgets

Eingabewidgets in Formularen sind Werte erster Klasse:

- `(make-text-field [default] [attributes])`  
erzeugt ein neues Texteingabefeld und gibt eine Scheme-Repräsentation zurück
- `(input-field-value input-field bindings [default])`  
extrahiert den Wert eines Eingabefeldes aus den Bindings
- `(make-submit-button [caption] [attributes])`  
Erzeugt einen Submit-Button
- `(surflet-form k-url [method] [attributes] [sxml])`  
erzeugt ein neues Formular

Für vollständige Referenz: siehe Handbuch. Benutzer können neue Eingabefelder programmieren.

## wishlist.scm (1)

```
(define-structure surflet surflet-interface
  (open surflets
    scheme-with-scssh)
  (begin

    (define (ask-for-wish)
      (let ((wish-input (make-text-field))
            (submit-button (make-submit-button)))
        (let ((req
              (send-html/suspend
                (lambda (k-url)
                  `(html
                    (body
                     (h1 "Enter your wish")
                     (surflet-form ,k-url
                                   (p "Your wish:"
                                     ,wish-input
                                     ,submit-button))))))))
          (input-field-value wish-input
                            (get-bindings req))))))
```

## wishlist.scm (2)

```
(define (print-wish wish)
  (send-html/finish
    `(html
      (body
        (h1 "Your (short) wishlist")
        (p ,wish))))))

(define (main req)
  (print-wish (ask-for-wish))))
```

# Session-Daten

Was ist eigentlich mit den Session-Daten passiert?

- Werte von Aufruf zu Aufruf transportieren: Rückgabewert von Funktionen (oder Session-API)
- Werte global für alle Sessions eines SURflets: globale Variable im SURflet-Modul
- Werte global für alle Sessions aller SURflets: globale Variable in einem Extramodul

# wishlist-pro.scm (1)

```
(define-structure surflet surflet-interface
  (open scheme-with-scsch
    surflets
    (subset srfi-1 (filter))
    inspect-exception)
  (begin

    (define *wish-id* 0)

    (define (fresh-wish-id)
      (set! *wish-id* (+ *wish-id* 1))
      *wish-id*)

    (define (make-wish text)
      (cons (fresh-wish-id) text))

    (define wish-id car)
    (define wish-text cdr)
```

## wishlist-pro.scm (2)

```
(define (output-wish-list wish-list delete-button k-url)
  `(ul
    ,@(map
      (lambda (wish)
        `(li ,(wish-text wish) " "
            (url ,(delete-button k-url (wish-id wish))
                "delete wish"))))
    wish-list)))
```

## wishlist-pro.scm (3)

```
(define (manage-wish-list wish-list)
  (let ((wish-input (make-text-field))
        (no-more-wishes (make-address))
        (submit-button (make-submit-button))
        (delete-button (make-annotated-address)))

    (let ((req
           (send-html/suspend
            (lambda (k-url)
              `(html
               (body
                (h1 "Wishlist Manager Pro")
                (surflet-form
                 ,k-url POST
                 ,(output-wish-list
                  wish-list delete-button k-url)
                 (p "Add wish:" ,wish-input
                   ,submit-button))
                 (p (url ,(no-more-wishes k-url)
                        "Perfectly happy"))))))))))))
```

## wishlist-pro.scm (4)

```
(case-returned-via (get-bindings req)
  ((no-more-wishes) wish-list)
  ((delete-button) => (lambda (id)
                        (manage-wish-list
                         (filter
                          (lambda (wish)
                            (not (= id (wish-id wish))))
                          wish-list))))
  (else
   (manage-wish-list
    (cons (make-wish
           (input-field-value
            wish-input (get-bindings req)))
          wish-list))))))
```

## wishlist-pro.scm (5)

```
(define (main req)
  (with-inspecting-handler
    8888
    (lambda (condition)
      (with-current-output-port (current-error-port)
        (display "starting remote handler for condition ")
        (display condition)
        (newline)
        (display "Please connect to port 8888")
        (newline)))
    (lambda ()
      (send-html/finish
        `(html
          (body
            (h2 "Your wishlist")
            (ul
              ,@(map (lambda (wish) `(li ,(wish-text wish)))
                (manage-wish-list '())))))))))
))
```