Scheme

"God is recursive"

Scheme (Curry und andere Funktionen)

- Funktionen, die das Programmieren erleichtern:

reduce (reduziert eine Liste durch rekursives Anwenden einer Funktion)
filter (filtert eine Liste durch einem Praedikat)
process (Anwenden einer Funktion falls ein Praedikat stimmt)
curry (baut eine Lambda Funktion aus einer Funktion und Argumenten)
compose (schachteln von Funktionen)
disjoin/conjoin (Zusammenfuegen von Praedikaten)
take/drop (Listenfunktionen)

```
(define (user-get-matched mask lis)
  (filter (compose (curry match mask) user-hotmask)
            lis))
```

Scheme (Makros)

- Mit Makros koennen Spezialformen erzeugt werden
- Lisp-aehnliche Makros

```
(defmacro while (test . body)
  `(let loop ()
     (cond (,test ,@body (loop)))))
```

- R5RS Makros (hygienische Makros)

```
(define-syntax while
  (syntax-rules()
    ((while test body1 ...)
       (let loop ()
         (cond
           (test body1 ... (loop)))))))
```

Scheme (Verzoegerte Ausfuehrung)

- Mit delay kann man Auswertung verzoegern

```
(delay (+ 1 2)) -> promise
```

- Mit force kann man die Promise dann Auswerten

```
(let ((b (delay (+ 1 2)))) (force b)) -> 3
```

- Es koennen sogennante Stroeme benutzt werden
- Stroeme sind verzoegerte Listen

```
(defmacro cons-stream (x y) ‘(,x (delay ,y)))
(define (stream-car strom) (car strom))
(define (stream-cdr strom) (force (cdr strom)))
(define (math-N x) (cons-stream x (math-N (+ x 1))))
(strom-filter is-prime? (math-N))
```

Scheme (Continuations)

- Eine Continuation ist die "Gegenwart" der Aufuehrung
- Eine Continuation wartet auf das Ergebnis einer Auswertung
- call-with-current-continuation
  - macht die jetzige Continuation zu einer Prozedur,
  - uebergibt sie als Argument

```
(call/cc (lambda (exit)
   (for-each (lambda (x)
               (if (negative? x)
                   (exit x)))
             '(1 2 34 5 -3 2 3 4))))
```

- dynamic-wind sichert ein Codeblock vor Continuations

```
(dynamic-wind in body out)
```

- in wird vor, out nach body aufgerufen

Scheme (Zusammenfassung III)

- Kontrollstrukturen: if, cond
- Rekursion kann auch endrekursiv sein
- Mit Makros lassen sich Spezialformen definieren
- Mit delay/force laesst sich Ausfuehrung verzoegern
  - Stroeme sind eine elegante Moeglichkeit
- Continuations speichern die "Gegenwart"
  - Bedingte Spruenge sind moeglich

Scheme (Engines, Timer)

- Engines fuehren Berechnungen aus und verbrauchen Ticks
- Wir brauchen ein Timersystem (fuer Multitasking)

```
(start-timer ticks handler)
(stop-time)
(decrement-timer)
```

- Die Ticks sollen am Anfang einer Prozedur gezaehlt werden

```
(defmacro timed-lambda (formals . bla)
   `(lambda ,formals (decrement-timer) ,@bla))
```

- Implementierung von den Timerfunktionen -> Uebung :)
- Continuation ausserhalb der Engine (Rueckkehr)
- Continuation im Handler (wartet auf den neuen Tickwert)

Scheme (Engines)

```
(define make-engine
  (let ((do-complete #f) (do-expire #f))
    (define timer-handler
      (lambda ()
        (start-timer (call/cc do-expire) timer-handler)))
    (define new-engine
      (lambda (resume)
        (lambda (ticks complete expire)
          ((call/cc
             (lambda (escape)
               (set! do-complete
                 (lambda (ticks value)
                   (escape (lambda () (complete ticks value)))))
               (set! do-expire
                 (lambda (resume)
                   (escape (lambda ()
                             (expire (new-engine resume))))))
               (resume ticks)))))))
    (lambda (proc)
      (new-engine
        (lambda (ticks)
          (start-timer ticks timer-handler)
          (let ((value (proc)))
            (let ((ticks (stop-timer)))
              (do-complete ticks value)))))))))
```

Scheme (Engines)

- Nach der harten Arbeit: round-robin

```
(define round-robin
   (lambda (engines)
      (if (null? engines)
          '()
          ((car engines)
             10
             (lambda (ticks values)
                     (cons value
                           (round-robin (cdr engines))))
             (lambda (engine)
                     (round-robin
                        (append engine
                           (cdr engines)))))))))
```

Scheme (Metazirkulaerer Evaluator)

- Metazirkulaerer Evaluator: Scheme Evaluator in Scheme
- Ziel ist, verschiedene Merkmale von Scheme anzupassen
- Ist der Ausdruck:
  - Symbol, Wert in der Umgebung aufsuchen
  - Selbstauswertender Ausdruck, Wert zurueckgeben
  - Liste:
    - Quote, einfach cdr zurueckgeben
    - Lambda, neue Umgebung, rekursives Auswerten
    - If, Auswerten der If-form
    - set!, Umgebung modifizieren

Scheme (Umgebung)

```
(define new-env
  (lambda (formals actuals env)
    (cond
      ((null? formals) env)
      ((symbol? formals) (cons (cons formals actuals) env))
      (else
       (cons (cons (car formals) (car actuals))
             (new-env (cdr formals) (cdr actuals) env))))))

(define lookup
  (lambda (var env)
    (cdr (assq var env))))

(define assign
  (lambda (var val env)
    (set-cdr! (assq var env) val)))
```

Scheme (Evaluator)

```
(define exec
    (lambda (exp env)
      (cond
        ((symbol? exp) (lookup exp env))
        ((pair? exp)
         (case (car exp)
           ((quote) (cadr exp))
           ((lambda)
            (lambda vals
              (let ((env (new-env (cadr exp) vals env)))
                (let loop ((exps (cddr exp)))
                   (if (null? (cdr exps))
                       (exec (car exps) env)
                       (begin
                          (exec (car exps) env)
                          (loop (cdr exps))))))))
           ((if)
            (if (exec (cadr exp) env)
                (exec (caddr exp) env)
                (exec (cadddr exp) env)))
           ((set!)
            (assign (cadr exp)
                    (exec (caddr exp) env)
                    env))
           (else
            (apply (exec (car exp) env)
                   (map (lambda (x) (exec x env))
                        (cdr exp))))))
        (else exp))))
```

Scheme (Lazy Evaluation)

- Auswertung in normaler Reihenfolge
  - Argumente nur dann ausgewertet, wenn gebraucht
- Verzoegerte Argumente werden in "Thunks" gepackt
  - Thunk : Ausdruck und Umgebung
- Thunks werden nur in 2 Faellen ausgewertet:
  - der Thunk ist selber die Funktion
  - die Funktion ist primitiv (also strikt)
- Die Implementierung sollte eine einfache Uebung sein :)

Scheme (Zusammenfassung IV)

- Multitasking ist einfach zu implementieren
- Mit einem eigenen Evaluator
  - kann man Scheme veraendern
  - kann man eine neue Sprache testen

Scheme (Guile)

- Guile ist ein Scheme Interpreter der FSF
- Webseite : http://www.gnu.org/software/guile
- Ziele:
  - Einbettbar in andere Programme
  - Translator fuer andere Sprachen
  - GOOPS OOP System
- Leider wird sehr viel getrollt und nicht viel gemacht
- Einigermassen langsam (langes Hochfahren)
- Frei, viel benutzt (zB gnome)
- Ziemlich viele Libraries

Scheme (Bigloo)

- Bigloo ist ein freier Scheme Compiler/Interpreter
- Webseite: http://kaolin.unice.fr/~serrano/bigloo
- Ziele:
  - Kleine, effiziente Binaries
  - Verknuepfung von Scheme mit C/C++ Code
  - Bigloo Object System
- Ziemlich grosse/praktische Libraries
- Verknuepfung mit C Code ist sehr einfach/praktisch
- Auch als Skriptinterpreter benutzbar, da schneller als Guile

Bigloo (Bigloo-lib)

- In der bigloo-lib (http://bigloo-lib.sourceforge.net):
  - GTK Wrapper
  - Regexp, Stringhandling
  - Unixanbindung
  - SQL Wrapper
  - CGI/HTTP Wrapper
  - XML Parser
  - LDAP Wrapper

Bigloo (GTK Programmierung)

- Wrapper fuer ziemlich alle GTK Funktionen
- Callbacks sind zB ziemlich einfach zu benutzen

```
(let ((hbox (gtk-hbox-new))
      (but (gtk-button-new "Hello World!")))
   (gtk-container-add *window* hbox)
   (gtk-container-add hbox but)
   (gtk-signal-connect
      but "clicked"
      (lambda args (print "Hello World!"))))
(gtk-widget-show-all *window*)
(gtk-main)
```

- Eignet sich sehr gut fuer Skripte

Guile (Squile)

- Squile von Forcer (http://forcix.cx)
- Datenbank Anbindung (Mysql und Postgresql)
- Durch filter, map, usw... kann man DBs sehr konfortabel ansprechen

```
(use-modules (ice-9 squile))
(let ((dbo (sql-open 'mysql "db" "localhost" "blub" "broesel")))
     (for-each
        format-db-entry
        (filter (map transform-db-entry
                (sql-query "SELECT * FROM TISCH")))))
```

Latte (Lang. for Transforming Text)

- Latte ist eine Schemeaehnliche Sprache
- Einfuegen von Markupelementen in Textdokumenten
  - HTML Generation
- Keine () sondern {}
- Symbole mit \ gekennzeichnet
- Woerter werden in "groups" (Listen) gepackt

```
{\gnoo-page {gnoo.org}
  {\div \align=center
      {\img \src=gnoo.jpg \alt={Gnoo !}}}
  Test}
```

BRL (Beautiful Report Language)

- Java Servlet zur Darstellung von DB-Inhalten (Report Language)
- Benutzt das Kawa System
- Nicht so wie Latte, benutzt "richtigen" Scheme
- Einfaches Benutzen von GET/POST "Variablen"
- DB Anbindung ueber JDBC

```
[(if (brl-blank? name)
     "<p>Please go back and fill in your name/color.</p>"
     (if (positive? (sql-execute-update st
                     "update favcolor set color="
                     (brl-sql-string color)
                     " where name="
                     (brl-sql-string name)))
```

Quellen

- http://www.schemers.org
- http://www.scheme.com
- http://mitpress.mit.edu/sicp
- http://www.gnu.org/software/guile
- http://kaolin.unice.fr/~serrano/bigloo
- http://forcix.cx