Heutiges Thema

- **Teil 1:** Programmieren im Großen, Module, abstrakte Datentypen, reflektive Programmierung
- **Teil 2:** Zusammenfassung und Ausblick

Teil 1: Programmieren im Großen

Das Modulkonzept von Haskell...

Modularisierung im allgemeinen (1)

Intuitiv:

- Zerlegung großer Programm(systeme) in kleinere Einheiten, genannt *Module*

Ziel:

- Sinnvolle, über- und durchschaubare Organisation des Gesamtsystems

Modularisierung im allgemeinen (2)

Vorteile:

- *Arbeitsphysiologisch* ...Unterstützung arbeitsteiliger Programmierung
- *Softwaretechnisch* ...Unterstützung der Wiederbenutzung von Programmen und Programmteilen
- *Implementierungstechnisch* ...Unterstützung von "separate compilation"

Insgesamt:

- Höhere Effizienz der Softwareerstellung bei gleichzeitiger Steigerung der Qualität (Verlässlichkeit) und reduzierten Kosten

Modularisierung im allgemeinen (3)

Anforderungen an das Modulkonzept zur Erreichung vorgenannter Ziele:

- Unterstützung des Geheimnisprinzips
  ...durch Trennung von
  - Schnittstelle (Import/Export)
    ...wie interagiert das Modul mit seiner Umgebung? Welche Funktionalität stellt es zur Verfügung (*Export*), *welche Funktionalität erwartet es* (*Import*)*?*
  - Implementierung (Daten/Funktionen)
    ...wie ist die Funktionalität des Moduls realisiert?

  in einem Modul.

Module in Haskell – Allgemeiner Aufbau

```
module MyModule where

-- Daten- und Typdefinitionen
data D1 ... = ...
...
data Dn ... = ...

type T1 = ...
...
type Tn = ...

-- Funktionsdefinitionen
f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ...
```


Das Modulkonzept von Haskell

...unterstützt:

- Export
  - Selektiv/Nicht selektiv
- Import
  - Selektiv/Nicht selektiv
  - Qualifiziert
  - Mit Umbenennung

...unterstützt nicht:

- automatischer Reexport!

Import: Nicht selektiv

```
module M1 where
...

module M2 where
import M1  -- Nicht selektiver Import:
           -- Alle im Modul M1 (sichtbaren) Bezeichner/
           -- Definitionen werden importiert und koennen
           -- in M2 benutzt werden.
```

Import: Selektiv

```
module M1 where
...                             -- wie auf voriger Folie

module M2 where
import M1 (D1, D2 (..), T1, f5)  -- Selektiver Import:
                                 -- Lediglich D1 (ohne Konstruktoren),
                                 -- D2 (einschliesslich Konstruktoren,
                                 -- T1 und f5 werden importiert und
                                 -- koennen in M2 benutzt werden.

module M3 where
import M1 hiding (D1, T2, f1)  -- Selektiver Import:
                                 -- Alle (sichtbaren) Bezeichner/
                                 -- Definitionen mit Ausnahme der
                                 -- explizit genannten werden importiert
                                 -- und koennen in M3 benutzt werden.
```


Export: Nicht selektiv

```
module M1 where     -- Nicht selektiver Export:
data D1 ... = ...   -- Alle im Modul M1 (sichtbaren)
...                 -- Bezeichner/Definitionen werden
data Dn ... = ...   -- exportiert und koennen von anderen
                    -- Modulen importiert werden.
type T1 = ...
...
type Tn = ...

f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ......
```


Export: Selektiv

```
module M1 (D1, D2 (..), T1, f2, f5) where
data D1 ... = ...
...                 -- Selektiver Export:
data Dn ... = ...   -- Nur die explizit genannten Bezeichner/
                    -- Definitionen werden exportiert und
type T1 = ...       -- koennen von anderen Modulen importiert
...                 -- werden.
type Tn = ...       -- Unterstuetzt Geheimnisprinzip!


f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ......
```


Kein automatischer Reexport (1)

```
module M1 where ...

module M2 where
import M1    -- Nicht selektiver Import:
             -- Alle im Modul M1 (sichtbaren) Bezeichner/
fM2...       -- Definitionen werden importiert und koennen
             -- in M2 benutzt werden.

module M3 where
import M2    -- Alle im Modul M2 (sichtbaren) Bezeichner/
             -- Definitionen werden importiert und koennen
             -- in M3  benutzt werden, nicht jedoch von
             -- M2 aus M1 importierte Namen. Mithin: Kein
             -- automatischer Reexport!
```

Kein automatischer Reexport (2)

Abhilfe: Expliziter Reexport!

```
module M4 (D1 (..), f1, f2) where   -- Selektiver
import M1                           -- Reexport

module M2 (M1,fM2) where   -- Nicht selektiver
import M1                  -- Reexport
```


Sonderfälle: Namenskollisionen, Lokale Namen

- Namenskollisionen
  - Abhilfe/Auflösen: Qualifizierter Import
    `import qualified M1`
- Umbenennen importierter Module
  - Lokale Namen importierter
    - Module
      `import MyM1 as M1`
    - Bezeichner
      `import M1 (f1,f2) renaming (f1 to g1, f2 to g2)`

Konventionen und gute Praxis

- Konventionen
  - Pro Datei ein Modul
  - Modul- und Dateiname stimmen überein (abgesehen von der Endung `.hs` bzw. `.lhs` im Dateinamen).
  - Alle Deklarationen beginnen in derselben Spalte wie `module`.
- Gute Praxis
  - Module unterstützen *eine* (*!*) klar abgegrenzte Aufgabenstellung (vollständig) und sind in diesem Sinne in sich abgeschlossen; ansonsten Teilen (Teilungskriterium)
  - Module sind "kurz" (ideal: 2 bis 3 Druckseiten; prinzipiell: "so lang wie nötig, so kurz wie möglich")

Wiederholung (vgl. Folie 27, Vorlesungsteil 1)

"Verstecken" von in `Prelude.hs` vordefinierten Funktionen...

Ergänze...

```
import Prelude hiding (reverse,tail,zip)
```

...am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden), um `reverse`, `tail` und `zip` zu verbergen.

Das Hauptmodul

Module `main`...

- ...muss in jedem Modulsystem als "top-level" Modul vorkommen und eine Definition namens `main` festlegen.

  $\leadsto$ ...ist der in einem übersetzten System bei Ausführung des übersetzten Codes zur Auswertung kommende Ausdruck.

Regeln "guter" Modularisierung (1)

(siehe dazu auch M. Chakravarty, G. Keller. *Einführung in die Programmierung mit Haskell*. Kapitel 10, Pearson Studium, 2004.)

Aus Modulsicht:

Module sollen...

- einen klar definierten, auch unabhängig von anderen Modulen verständlichen Zweck besitzen
- nur einer Abstraktion entsprechen
- einfach zu testen sein

Regeln "guter" Modularisierung (2)

Aus Gesamtprogrammsicht:

Aus Modulen aufgebaute Programme sollen so entworfen sein, dass...

- Auswirkungen von Designentscheidungen (z.B. Einfachheit vs. Effizienz einer Implementierung) auf wenige Module beschränkt sind
- Abhängigkeiten von Hardware oder anderen Programmen auf wenige Module beschränkt sind

Regeln "guter" Modularisierung (3)

Aus intra- und intermodularer Sicht:

Zwei zentrale Konzepte in diesem Zusammenhang sind...

- Intramodular: *Kohäsion*
- Intermodular: *Kopplung*

Regeln "guter" Modularisierung (4)

Aus intramodularer Sicht:

Kohäsion

- Anzustreben sind...
  - *Funktionale Kohäsion* (d.h. Funktionen ähnlicher Funktionalität sollten in einem Modul zusammengefasst sein, z.B. Ein-/Ausgabefunktionen, trigonometrische Funktionen, etc.)
  - *Datenkohäsion* (d.h. Funktionen, die auf den gleichen Datenstrukturen arbeiten, sollten in einem Modul zusammengefasst sein, z.B. Baummanipulationsfunktionen, Listenverarbeitungsfunktionen, etc.)
- Zu vermeiden sind...
  - *Logische Kohäsion* (d.h. unterschiedliche Implementierungen der gleichen Funktionalität sollten in verschiedenen Modulen untergebracht sein, z.B. verschiedene Benutzerschnittstellen eines Systems)
  - *Zufällige Kohäsion* (d.h. Funktionen sind ohne sachlichen Grund, zufällig eben, in einem Modul zusammengefasst)

Regeln "guter" Modularisierung (5)

Aus intermodularer Sicht:

Kopplung

- beschäftigt sich mit dem Import-/Exportverhalten von Modulen
  - Anzustreben ist...
    - *Datenkopplung* (d.h. Funktionen unterschiedlicher Module kommunizieren nur durch Datenaustausch (in funktionalen Sprachen per se gegeben))

Regeln "guter" Modularisierung (6)

Kennzeichen "guter" Modularisierung:

- *Starke Kohäsion*
  d.h. enger Zusammenhang der Definitionen eines Moduls
- *Lockere Kopplung*
  d.h. wenige Abhängigkeiten zwischen verschiedenen Modulen, insbesondere weder direkte noch indirekte zirkuläre Abhängigkeiten.

Abstrakte Datentypen, kurz: ADTs (1)

Ziel...

- Kapselung von Daten, Realisierung des Geheimnisprinzips auf Datenebene (engl. *information hiding*)

Implementierungstechnisch zentral...

- Haskells Modulkonzept, speziell "selektiver Export"

Abstrakte Datentypen, kurz: ADTs (2)

Hinweise auf drei klassische Literaturstellen:

- John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM, Vol. 20, No. 6, 396-404, 1977.
- John V. Guttag, J. J. Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica, Vol. 10, No. 1, 27-52, 1978.
- John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM, Vol. 21, No. 12, 1048-1064, 1978.

Abstrakte Datentypen, kurz: ADTs (3)

Die grundlegende Idee am Beispiel des Typs *Schlange*:

```
Schnittstellen/Signatur:
NEW:           -> Queue
ADD:      Queue x Item -> Queue
FRONT:    Queue -> Item
REMOVE:   Queue -> Queue
IS_EMPTY: Queue -> Boolean

Axiome/Gesetze:
(1) IS_EMPTY(NEW) = true
(2) IS_EMPTY(ADD(q,i)) = false
(3) FRONT(NEW) = error
(4) FRONT(ADD(q,i)) = if IS_EMPTY(q) then i else FRONT(q)
(5) REMOVE(NEW) = error
(6) REMOVE(ADD(q,i)) = if IS_EMPTY(q) then NEW
                                      else ADD(REMOVE(q),i)
```


Bsp.: (Warte-) Schlangen als ADT (1)

Warteschlange...

...eine FIFO (first-in/first-out) Datenstruktur

```
module Queue ( Queue,      --  Kein Konstruktorexport!!!
               emptyQ,     --  Queue a
               isEmptyQ,   --  Queue a -> Bool
               joinQ,      --  a -> Queue a -> Queue a
               leaveQ,     --  Queue a -> (a, Queue a)
             ) where

...  -- Fortsetzung siehe naechste Folie
```


Bsp.: (Warte-) Schlangen als ADT (2)

```
... -- Fortsetzung der vorherigen Folie

data Queue = Qu [a]

emptyQ = Qu []

isEmptyQ (Qu []) = True
isEmptyQ _       = False

joinQ x (Qu xs) = Qu (xs++[x])

leaveQ q@(Qu xs)
  | not (isEmptyQ q)  =  (head xs, Qu (tail xs))
  | otherwise         =  error "Niemand wartet!"
```

Bsp.: (Warte-) Schlangen als ADT (3)

Notationelle Spielart des Mustervergleichs...

```
leaveQ q@(Qu xs)   --  argument as "q or as (Qu xs)"
```

Mittels...

- `q` Zugriff auf das Argument als Ganzes
- `(Qu xs)` Zugriff auf/über die Struktur des Arguments

Bsp.: (Warte-) Schlangen als ADT (2)

Programmiertechn. Vorteile aus der Benutzung von ADTs...

- *Geheimnisprinzip*: Nur die Schnittstelle ist bekannt, die Implementierung bleibt verborgen
  - Schutz der Datenstruktur vor unkontrolliertem oder nicht beabsichtigtem/zugelassenem Zugriff
  - Einfache Austauschbarkeit der zugrundeliegenden Implementierung
  - Arbeitsteilige Programmierung

Beispiel: `emptyQ == Qu []`

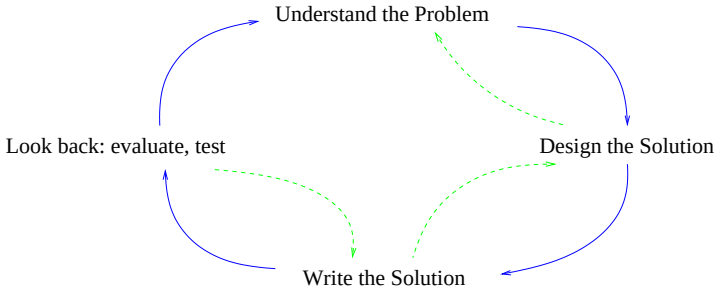
...führt in `Queue` importierenden Modulen zu einem Laufzeitfehler! (Die Implementierung und somit der Konstruktor `Qu` sind dort nicht sichtbar.)

Resümee algebraische vs. abstrakte Datentypen

- Algebraische Datentypen
  - ...werden durch die Angabe ihrer Elemente spezifiziert, aus denen sie bestehen.
- Abstrakte Datentypen
  - ...werden durch ihr Verhalten spezifiziert, d.h. durch die Menge der Operationen, die darauf arbeiten.

Reflektives Programmieren

Der Entwicklungszyklus nach S. Thompson, Kap. 11 [3] ...



In jeder dieser Phasen ist es hilfreich, (sich) Fragen zu stellen!

Für eine beispielhafte Auswahl siehe z.B. S. Thompson, Kap. 11 [3]!

Teil 2: Zusammenfassung und Ausblick

Abschluss und Rückblick...

- Blick über den Gartenzaun
  ...(ausgewählte) andere funktionale Programmiersprachen
- Rückblick auf die Vorlesung

Im Rückblick auf die Vorlesung...

- Welche Aspekte funktionaler Programmierung haben wir betrachtet?
  - ...paradigmentypische, sprachunabhängige Aspekte
- Welche nicht oder nur gestreift?
  - ...sprachabhängige, speziell Haskell-spezifische Aspekte

Frei nach (und im Sinne von) Dietrich Schwanitz...

"*Alles, was man wissen muss...*
...um selber weiter zu lernen."

Überblick über die Vorlesungsinhalte... (1)

00 Vorbesprechung

01 Einführung und Motivation
- Warum funktionale Programmierung
- Warum Haskell
- Einstieg in Haskell und Hugs

Grundlagen
- Elementare Datentypen: Wahrheitswerte, ganze Zahlen,...
- Tupel, Listen und Funktionen, insbesondere notationelle Varianten

Überblick über die Vorlesungsinhalte... (2)

02 Funktionen und Funktionssignaturen

03 Funktionen und Funktionsterme, Rekursionstypen, Komplexitätsklassen und Aufrufgraphen, Funktionen und Curryfizierung

04 Datentypdeklarationen
- Typsynonyme
- Algebraische Datentypen
  - Summentypen
  - Spezialfälle: Aufzählungs- und Produkttypen
  - Wichtige Varianten: Rekursive und polymorphe Typen, `newtype`-Deklarationen

Überblick über die Vorlesungsinhalte... (3)

05 Polymorphie auf Funktionen und Typen, Typklassen

- Parametrische und ad-hoc Polymorphie

06 Ergänzungen zu Polymorphie und Typklassen, Vererbung, Listen und Listenkomprehension, Muster

07 Funktionen höherer Ordnung, Ein- und Ausgabe, Fehlerbehandlung

08 Programmierung mit Monaden, Auswertungsstrategien für Funktionen: lazy vs. eager, Hintergrund und Grundlagen: der Lambda-Kalkül

Überblick über die Vorlesungsinhalte... (4)

09 Programmieren im Großen

- Das Modulkonzept von Haskell
- Abstrakte Datentypen
- Reflektives Programmieren

Abschluss und Rückblick

- Blick über den Gartenzaun und Resümee

Resümee (1)

Charakteristika fkt. und imp. Sprachen (P. Pepper [4])...

- Funktional...
  - Programm ist *Ein-/Ausgaberelation*
  - Programme sind *"zeit"-los*
  - Programmformulierung auf *abstraktem*, *mathematisch geprägten Niveau*
- Imperativ...
  - Programm ist *Arbeitsanweisung* für eine Maschine
  - Programme sind *zustands-* und *"zeit"-behaftet*
  - Programmformulierung konkret *mit Blick auf eine Maschine*

Resümee (2)

Die Fülle an Möglichkeiten (in funktionalen Programmiersprachen) erwächst aus einer kleinen Zahl von elementaren Konstruktionsprinzipien.

P. Pepper [4]

Im Falle von...

- Funktionen
  - (Fkt.-) Applikation, Fallunterscheidung und Rekursion
- Datenstrukturen
  - Produkt- und Summenbildung, Rekursion

Tragen bei zu Mächtigkeit und Eleganz und damit auch zu...

Functional programming is fun!

Rückschauend... war es das?

Blick über den Gartenzaun

...auf ausgewählte andere funktionale Programmiersprachen:

- ML: Ein "eager" Wettbewerber
- Lisp: Der Oldtimer
- APL: Ein Exot

...und einige ihrer Charakteristika.

ML: Ein "eager" Wettbewerber von Haskell

- ML ist eine strikte funktionale Sprache
- Lexical scoping, Curryfizieren (wie Haskell)
- stark typisiert mit Typinferenz, keine Typklassen
- umfangreiches Typkonzept für Module und ADT
- zahlreiche Erweiterungen (beispielsweise in OCAML) auch für imperative und objektorientierte Programmierung
- sehr gute theoretische Fundierung

Beispiel: Module/ADTs in ML

```
structure S = struct
    type 't Stack              = 't list;
    val  create                = Stack nil;
    fun  push x (Stack xs)     = Stack (x::xs);
    fun  pop (Stack nil)       = Stack nil;
    |    pop (Stack (x::xs))   = Stack xs;
    fun  top (Stack nil)       = nil;
    |    top (Stack (x:xs))    = x;
end;

signature st = sig type q; val push: 't -> q -> q; end;

structure S1:st = S;
```


Lisp: Der Oldtimer funktionaler Programmiersprachen

- Lisp ist eine noch immer häufig verwendete strikte funktionale Sprache mit imperativen Zusätzen
- umfangreiche Bibliotheken, leicht erweiterbar
- einfache, interpretierte Sprache, dynamisch typisiert
- Listen sind gleichzeitig Daten und Funktionsanwendungen
- nur lesbar, wenn Programme gut strukturiert
- in vielen Bereichen (insbesondere KI, Expertensysteme) erfolgreich eingesetzt
- sehr gut zur Metaprogrammierung geeignet

Ausdrücke in Lisp

Beispiele für Symbole:

A	(Atom)
austria	(Atom)
68000	(Zahl)

Beispiele für Listen:

```
(plus a b)
((meat chicken) water)
(unc trw synapse ridge hp)
```

nil bzw. () entsprechen leerer Liste

Eine Zahl repräsentiert ihren Wert direkt —
ein Atom ist der Name eines assoziierten Wertes

(setq x (a b c)) bindet x global an (a b c)

(let ((x a) (y b)) e) bindet x lokal in e an a und y an b

Funktionen in Lisp

Erstes Element einer Liste wird normalerweise als Funktion interpretiert, angewandt auf restliche Listenelemente.

(quote a) bzw. 'a liefert Argument a selbst als Ergebnis.

Beispiele für primitive Funktionen:

(car '(a b c))	⇒ a	(atom 'a)	⇒ t
(car 'a)	⇒ error	(atom '(a))	⇒ nil
(cdr '(a b c))	⇒ (b c)	(eq 'a 'a)	⇒ t
(cdr '(a))	⇒ nil	(eq 'a 'b)	⇒ nil
(cons 'a '(b c))	⇒ (a b c)	(cond ((eq 'x 'y) 'b)	
(cons '(a) '(b))	⇒ ((a) b)	(t 'c))	⇒ c

Definition von Funktionen in Lisp

(lambda (x y) (plus x y)) ist Funktion mit zwei Parametern

((lambda (x y) (plus x y)) 2 3) wendet die Funktion an: ⇒ 5

(define (add (lambda (x y) (plus x y)))) definiert einen globalen Namen „add“ für die Funktion

(defun add (x y) (plus x y)) ist abgekürzte Schreibweise dafür

Beispiel:

```
(defun reverse (l) (rev nil l))
(defun rev (out in)
     (cond ((null in) out)
           (t (rev (cons (car in) out) (cdr in)))))
```


Closures

- kein Curryfizieren in Lisp, Closures als Ersatz
- Closures: lokale Bindungen behalten Wert auch nach Verlassen der Funktion

  Beispiel:
  ```
  (let ((x 5))
     (setf (symbol-function 'test)
        #'(lambda () x)))
  ```
- praktisch: Funktion gibt Closure zurück

  Beispiel:
  ```
  (defun create-function (x)
     (function (lambda (y) (add x y)))
  ```
- Closures sind flexibel, aber Curryfizieren ist viel einfacher

Dynamic Scoping — Static Scoping

- lexikalisch: Bindung ortsabhängig (Source-Code)
- dynamisch: Bindung vom Zeitpunkt abhängig
- normales Lisp: lexikalisches Binden

  Beispiel:
  ```
  (setq a 100)
  (defun test () a)
  (let ((a 4)) (test))    ⇒ 100
  ```
- dynamisches Binden durch (defvar a) möglich

  obiges Beispiel liefert damit 4

Makros

- Code expandiert, nicht als Funktion aufgerufen (wie C)
- Definition: erzeugt Code, der danach evaluiert wird

  Beispiel:
  ```
  (defmacro get-name (x n)
      (list 'cadr (list 'assoc x n)))
  ```
- Expansion und Ausführung:

  (get-name 'a b) ⇔ (cadr (assoc 'a b))
- nur Expansion:

  (macroexpand '(get-name 'a b)) ⇒ '(cadr (assoc 'a b))

Lisp vs. Haskell: Ein Vergleich

Kriterium	Lisp	Haskell
Basis	einfacher Interpreter	formale Grundlage
Zielsetzung	viele Bereiche	referentiell transparent
Verwendung	noch häufig	zunehmend
Sprachumfang	riesig (kleiner Kern)	moderat, wachsend
Syntax	einfach, verwirrend	modern, Eigenheiten
Interaktivität	hervorragend	nur eingeschränkt
Typisierung	dynamisch, einfach	statisch, modern
Effizienz	relativ gut	relativ gut
Zukunft	noch lange genutzt	einflussreich

APL: Ein Exot

- APL ist eine ältere applikative (funktionale) Sprache mit imperativen Zusätzen
- zahlreiche Funktionen (höherer Ordnung) sind vordefiniert, Sprache aber nicht einfach erweiterbar
- dynamisch typisiert
- verwendet speziellen Zeichensatz
- Programme sehr kurz und kompakt, aber kaum lesbar
- vor allem für Berechnungen mit Feldern gut geeignet

Beispiel: Programmentwicklung in APL

Berechnung der Primzahlen von 1 bis N

Schritt 1. $(\iota N) \circ.| (\iota N)$

Schritt 2. $0 = (\iota N) \circ.| (\iota N)$

Schritt 3. $+/[2]\ 0 = (\iota N) \circ.| (\iota N)$

Schritt 4. $2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N))$

Schritt 5. $(2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N))) / \iota N$

Erfolgreiche Einsatzfelder funktionaler Programmierung

- Compiler in compilierter Sprache geschrieben
- Theorembeweiser HOL und Isabelle in ML
- Modelchecker (z.B. Edinburgh Concurrency Workbench)
- Mobility Server von Ericson in Erlang
- Konsistenzprüfung mit Pdiff (Lucent 5ESS) in ML
- CPL/Kleisli (komplexe Datenbankabfragen) in ML
- Natural Expert (Datenbankabfragen Haskell-ähnlich)
- Ensemble zur Spezifikation effizienter Protokolle (ML)
- Expertensysteme (insbesondere Lisp-basiert)
- ...
- http://homepages.inf.ed.ac.uk/wadler/realworld/

Zum Schluss: Eine hoffnungsvolle Prognose...

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.(*)

Edsger W. Dijkstra (11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

(*) Zitat aus: Introducing a course on calculi. Ankündigung einer Lehrveranstaltung an der University of Texas at Austin, 1995.

Lohnenswerte Literaturhinweise...

Bereits in der Vorbesprechung angegeben:

- Wadler, P. An angry half-dozen. ACM SIGPLAN Notices 33(2), 25-30, 1998
- Wadler, P. Why no one uses functional languages. ACM SIGPLAN Notices 33(8), 23-27, 1998

Aber man sollte stets beide Seiten der Medaille kennen...

- Hughes, J. Why Functional Programming Matters. Computer Journal 32(2), 98-107, 1989
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual Symposium on Principles of Programming Languages (POPL'92), eingeladener Vortrag, 1992.
- Simon Peyton-Jones. *Wearing the Hair Shirt: A Retrospective on Haskell*, eingeladener Vortrag, 30th Annual Symposium on Principles of Programming Languages (POPL'03), 2003.

Organisatorisches zur schriftlichen LVA-Prüfung 1(3)

Hinweise zur schriftlichen LVA-Prüfung...

- Worüber...
  - Vorlesungsstoff, Übungsstoff, und folgender wissenschaftlicher (Übersichts-) Artikel:
    Paul Hudak. *Conception, Evolution, and Application of Functional Programming Languages*. ACM Computing Surveys, Vol. 21, No. 3, September 1989, 359 - 411.
- Wann, wo und wie lange...
  - Der Haupttermin ist am...
    - Do, den 24.01.2008, von 16:00 Uhr s.t. bis 18:00 Uhr, im Radinger-Hörsaal am Getreidemarkt 2; die Dauer beträgt 90 Minuten.
- Hilfsmittel...
  - Keine.

Organisatorisches zur schriftlichen LVA-Prüfung 2(3)

Hinweise zur schriftlichen LVA-Prüfung...

- Mitzubringen sind...
  - Papier, Stifte, **Studierendenausweis**.
- Anmeldung: Ist erforderlich...
  - *Wann*: Zwischen dem 10.01.2008 und dem 20.01.2008
  - *Wie*: Elektronisch über TUWIS++
- Voraussetzung...
  - Mindestens 50% der Punkte aus der Laborübung, d.h. mindestens 450 Punkte.

Organisatorisches zur schriftlichen LVA-Prüfung 3(3)

Neben dem Haupttermin wird es drei Nebentermine für die schriftliche LVA-Prüfung geben, und zwar...

- zu Anfang
- in der Mitte
- am Ende

der Vorlesungszeit im SS 2008.

Danach in jedem Fall Zeugnisausstellung.

Die genauen Termine werden rechtzeitig auf der Webseite der LVA angekündigt!

Auch zur Teilnahme an der schriftlichen LVA-Prüfung an einem der Nebentermine ist eine Anmeldung über TUWIS++ erforderlich.

Besprechungen weiterer Aufgabenblätter...

Im neuen Jahr, voraussichtlich an zwei Donnerstagsterminen!

Beachten Sie bitte die entsprechenden Hinweise auf der Webseite der Lehrveranstaltung und in der newsgroup zur Vorlesung!

Einladung zum Kolloquiumsvortrag

Die Complang-Gruppe lädt ein zu folgendem Vortrag...

Automatic Verification of Combined Specifications

Prof. Dr. Ernst-Rüdiger Olderog

Carl v. Ossietzky Universität Oldenburg, Deutschland

ZEIT: Freitag, 13. Dezember 2007, 14:00 Uhr c.t.

ORT: TU Wien, Elektrotechnik, EI 5 Hochenegg-Hörsaal, Gußhausstr. 25-29 (Altbau), 2. Stock

MEHR INFO: `http://www.complang.tuwien.ac.at/talks/Olderog2007-12-13`

Alle Interessenten sind herzlich willkommen!

Das FP-Team wünscht Ihnen...

Ein frohes Weihnachtsfest

und

einen guten Rutsch ins neue Jahr!