Abschluss und Rückblick...

Zunächst...

- Abstrakte Datentypen
- Reflektives Programmieren

Anschließend...

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

Abstrakte Datentypen, kurz: ADTs

Ziel...

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

Implementierungstechnisch zentral...

- Haskells Modulkonzept, speziell "gezielter Export"

Ein Beispiel: (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
```


Ein Beispiel: (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!"
```


Ein Beispiel: (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

Ein Beispiel: (Warte-) Schlangen als ADT (2)

Programmiertechn. Vorteile aus der Benutzung von ADTs...

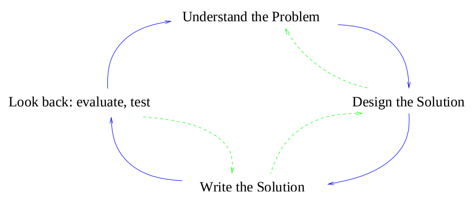
- *Geheimnisprinzip*: Nur die Schnittstelle ist bekannt, die Implementierung bleibt verborgen
  - Schutz der Datenstruktur vor unkontrolliertem oder nicht beabsichtigtem/zugelassenen Zugriff
  - Einfache Austauschbarkeit der zugrundeliegenden Implementierung
  - Arbeitsteilige Programmierung

Beispiel: `emptyQ == Qu []`

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

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]!

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 Motivation
- Warum funktionale Programmierung
- Warum Haskell

02 Erste Schritte in Haskell
- Hugs
- Elementare Datentypen: Wahrheitswerte, ganze Zahlen,...

03 Auswertungsstrategien für Funktionen: Lazy vs. eager

04 Hintergrund und Grundlagen: Der Lambda-Kalkül

Überblick über die Vorlesungsinhalte... (2)

05 Funktionen in Haskell: Notationelle Alternativen, Rekursionsmuster

06 Tupel und Listen, Typsynonyme

07 Listenkomprehension und Muster

08 Algebraische Datentypen
- Aufzählungstypen
- Produkttypen
- Summentypen
- Spezialfälle: Rekursive und polymorphe Typen

09 Polymorphie auf Typen und Funktionen, Typklassen
- Parametrische und ad-hoc Polymorphie

Überblick über die Vorlesungsinhalte... (3)

10 Funktionen höherer Ordnung

11 Ein-/Ausgabe, Monaden

12 Monaden, Programmieren im Großen
- Das Modulkonzept von Haskell

13 Abschluss und Rückblick
- Abstrakte Datentypen
- Reflektives Programmieren
- 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 Curryfizierung 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 — Lexical 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

Macros

- 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://www.cs.bell-labs.com/~wadler/realworld/

Organisatorisches... (1)

Abgabe- und Prüfungsgespräche...

- Wann, wo, Anmeldung...
  - *Wann*: Im gesamten Februar (Schwerpunkt: Mo, 02.02.04 - Fr, 06.02.04); einige (wenige) Termine in der ersten Märzwoche)
  - *Wo*: siehe Webseite (kurzfristige Festlegung/Änderung wg. Übersiedlung)
  - *Anmeldung*: Mo, 26.01.04 - Fr, 30.01.04 (durch Eintragen in Listen in den Laborräumen im 2. Stock, Argentinierstr. 8)
- Inhalt
  - Fragen zu Aufgaben und Vorlesung

⇝**...die Anmeldelisten hängen seit gestern aus!**

Ihr Urteil ist gefragt... (1)

(Online-) Evaluation von Pflichtlehrveranstaltungen bis zum 06.02.2004 (siehe TUWIS++)...

- Haben Sie profitiert von der LVA? In welcher Weise, wodurch, wodurch besonders? Haben sich Ihre Erwartungen und Ansprüche insgesamt oder i.w. erfüllt? Haben Sie eine andere und breitere und fundiertere Sicht auf Programmierung im allgemeinen und funktionale Programmierung im speziellen gewonnen?
- Was empfanden Sie dafür besonders hilfreich? Was war aus Ihrer Sicht besonders sinnvoll, was sollte beibehalten oder ausgebaut werden? (z.B. Lernziele, Zusammenfassungen, Faustregeln, Designentscheidungen und ihr Hintergrund,...)
- Wie hätte die LVA für Sie noch profitabler sein können? Was ist rückschauend aus Ihrer Sicht verbesserungswürdig, was sollte geändert oder angestrebt werden? (z.B. zeitnähere Rückmeldung in der newsgroup etwa bei Fragen zu Aufgabenstellungen, engerer Kontakt zu Tutoren, Skript, größere Programmierprojekte,...)

Ihr Urteil ist gefragt... (2)

Nutzen Sie die Möglichkeit...

- durch Ihre Rückmeldung auf die Vorlesung Einfluss zu nehmen!

Und seien Sie konstruktiv...

- unspezifische Rückmeldungen helfen wenig.

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.

Das FP-Team wünscht Ihnen eine...

erfolg- und erlebnisreiche vorlesungsfreie Zeit!