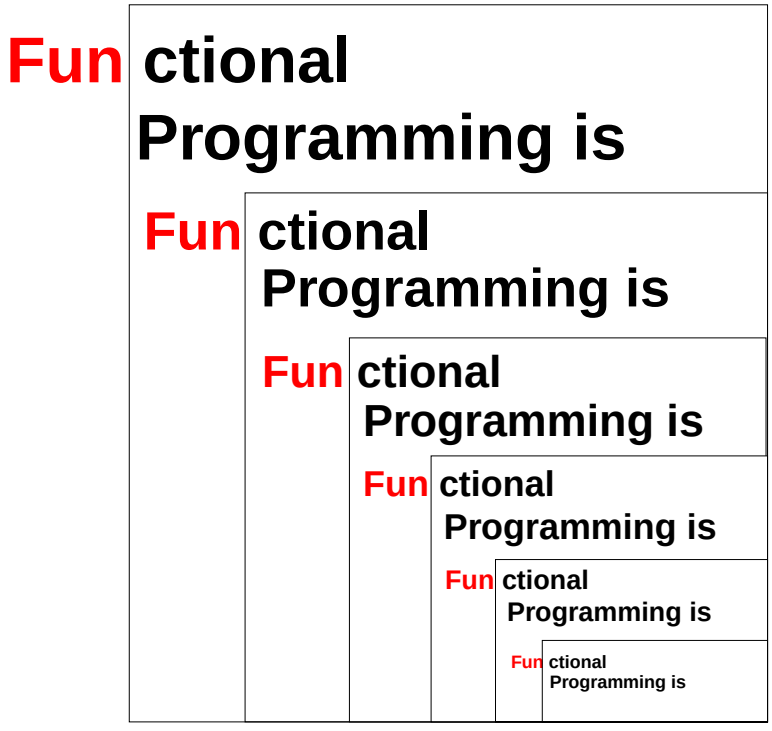
Funktionale Programmierung

Prof. Dr. Uwe Kastens

SS 2013

Functional Programming is Fun



Ziele, Material

Die Teilnehmer sollen

- die **Klarheit und Mächtigkeit** der funktionalen Programmierung erkennen,
- **Paradigmen** der funktionalen Programmierung erlernen,
- **Techniken** der funktionalen Programmierung an praktischen Beispielen erproben und einüben

Literatur:

- **Vorlesungsmaterial**:
  http://ag-kastens.upb.de/lehre/material/fp
- **Textbuch:**
  L. C. Paulson: ML for the Working Programmer, 2nd Edition, Cambridge University Press, 1996
  Schwerpunkt stärker auf Paradigmen und Techniken als auf der Sprache SML, enthält viele Beispiele bis hin zu nützlichern Modulen.
- Weiteres Buch zu SML:
  C. Myers, C. Clack, E.Poon: Programming with Standard ML, Prentice Hall, 1993
- siehe auch Internet-Links im Vorlesungsmaterial
  http://ag-kastens.upb.de/lehre/material/fp/wwwrefs.html

Inhalt

		Kapitel im Textbuch
1.	Einführung	
2.	LISP: FP Grundlagen	
3.	Grundlagen von SML	2
4.	Programmierparadigmen zu Listen	3
5.	Typkonstruktoren, Module	2, 4, 7
6.	Funktionen als Daten	5
7.	Datenströme	5
8.	Lazy Evaluation	
9.	Funktionale Sprachen: Haskell, Scala	
10.	Zusammenfassung	

Vorkenntnisse in FP aus Modellierung, GPS und PLaC

Modellierung:
- Modellierung mit Wertebereichen

GPS:
- Datentypen, parametrisierte Typen (Polytypen)
- **Funktionale Programmierung:**

SML-Eigenschaften:
- Notation
- Deklarationen, Muster
- Funktionen, Aufrufe
- Listen
- Datentypen, Polymorphie

Programmiertechniken:
- Rekursion zur Induktion
- Rekursion über Listen
- akkumulierender Parameter
- Berechnungsschemata (HOF)
- Currying (HOF)

PLaC:
- Spracheigenschaften und ihre Implementierung
- Typsysteme, Typanalyse für funktionale Sprachen

Gedankenexperiment

Wähle eine **imperative Sprache**, z. B. Pascal, C.

Erlaube Eingabe nur als Parameter und Ausgabe nur als Ergebnis des Programmaufrufes.

Eliminiere Variablen mit Zuweisungen.

Eliminiere schrittweise alle **Sprachkonstrukte**, die **nicht mehr sinnvoll anwendbar** sind.

eliminiertes Sprachkonstrukt	Begründung
...	...
...	...

Betrachte die **restliche Programmiersprache**.
Ist sie **sinnvoll anwendbar?**

Erweitere sie um nützliche Konstrukte (nicht Variablen mit Zuweisungen)

ergänztes Sprachkonstrukt	Begründung
...	...
...	...

2. LISP: FP Grundlagen

Älteste funktionale Programmiersprache (McCarthy, 1960).

Viele weit verbreitete **Weiterentwicklungen** (Common LISP, Scheme).

Ursprünglich wichtigste Sprache für Anwendungen im Bereich **Künstlicher Intelligenz**.

Hier nur **Pure LISP**:

Notation:
geklammerte Folge von Symbolen
gleiche Notation für Programm und Daten

Datenobjekte:
Listen: (1 2 3 4) (ggt 24 36)
atomare Werte: `T, F, NIL`, Zahlen, Bezeichner

Beispiel: Fakultätsfunktion mit akkumulierendem Parameter (**`defun`** nicht in Pure Lisp)

```
(defun AFac  (n a)
             (cond  ((eq n 0) a)
                    (T (AFac (- n 1) (* a n)))))
(defun Fac   (n) (AFac n 1))
```

Aufruf: `(Fac 4)`

Grundfunktionen

Aufruf: (`e0 e1 ... en`). Ausdruck `e0` liefert die aufzurufende Funktion, die übrigen liefern Werte der aktuellen Parameter.

Grundfunktionen:

`(cons e l)`	Listenkonstruktor
`(car l)`	Listenkopf
`(cdr l)`	Listenrest
`(null l)`	leere Liste?
`(eq a b)`	Vergleich atomarer Werte
`(equal l1 l2)`	tiefer Vergleich von Listen
`(atom e)`	atomar?
`(cond (p1 e1) ... (pn en))`	**Fallunterscheidung**; **nicht-strikte Auswertung**: `(pi ei)` von links nach rechts auswerten bis ein `pj T` liefert; Ergebnis ist dann `ej.`
`(quote e)`	`e` wird nicht ausgewertet, sondern ist selbst das Ergebnis
`(lambda (x1 ... xn) e)`	Funktionskonstruktor mit formalen Parametern `x1...xn` und Funktionsrumpf `e`

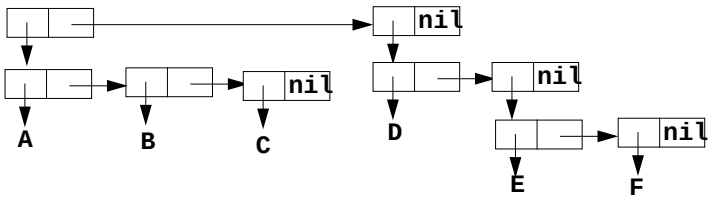
nicht in Pure Lisp:

`(setq x e)`	Wert von `e` wird an `x` gebunden (top level)
`(defun f (x1 ... xn) e)`	Funktion wird an `f` gebunden (top level)

Listen als Programm und Daten

Eine **Liste** ist eine evtl. leere Folge von Ausdrücken;
Jeder Ausdruck ist ein Atom oder eine Liste:

```
'()
'((A B C) (D (E F)))
```




```
(cdr (cdr '(1 2 3 4)))
```

Ein **Aufruf** ist eine Liste; beim Auswerten liefert ihr erstes Element eine **Funktion**;
die weiteren Elemente liefern die **aktuellen Parameterwerte** des Aufrufes.

Ein **Programm** ist eine Liste von geschachtelten Aufrufen.

Das Programm operiert auf Daten; sie sind Atome oder Listen.

Die Funktion **`quote` unterdrückt die Auswertung** eines Ausdruckes; der Ausdruck selbst ist das Ergebnis der Auswertung:
Die Auswertung von **`(quote (1 2 3 4))`** liefert **`(1 2 3 4)`**
Kurznotation für **`(quote (1 2 3 4))`** ist **`'(1 2 3 4)`**

Listen, die Daten in Programmen angeben, müssen durch **`quote`** gegen Auswerten geschützt werden: **`(length (quote 1 2 3))`** oder **`(length '(1 2 3))`**

Einige Funktionen über Listen

Länge einer Liste:

```
(defun Length (l) (cond ((null l) 0) (T (+ 1 (Length (cdr l))))))
```

Aufruf **`(Length '(A B C))`** liefert **`3`**

`Map`-Funktion (**`map`** ist vordefiniert):

```
(defun Map (f l)
            (cond ((null l) nil)
                  (T (cons (funcall f (car l))
                           (Map f (cdr l))))))
```

Aufruf **`(Map (lambda (n) (cons n (cons n nil))) '(1 2 3))`**

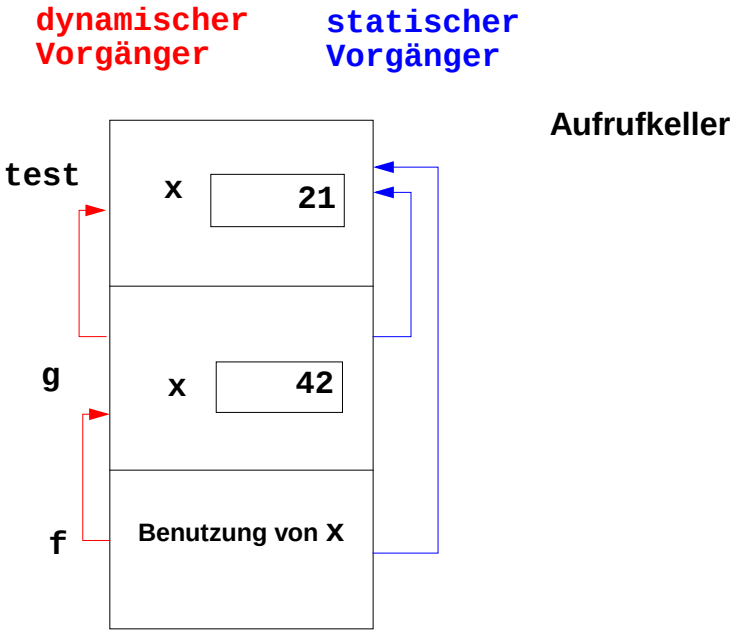
liefert **`'((1 1) (2 2) (3 3))`**

```
(funcall f p)
```

wertet **`f`** zu einer Funktion aus und ruft diese mit dem Parameter **`p`** auf

Statische oder dynamische Bindung von Namen

```
test
   def x = 21
   f
      x
   g
      def x = 42
      (f ())
   (g ())
```



statische Bindung von x: nächste Definition von x auf der **statischen Vorgängerkette**, d. h. in der **Definitionsumgebung**

dynamische Bindung von x: nächste Definition von x auf der **dynamischen Vorgängerkette**, d. h. in der **Aufrufumgebung**

Statische oder dynamische Bindung in Common Lisp?

Bindungen in geschachtelten Lambda-Ausdrücken:

```
(print
((lambda (x)              ; Umgebung test1 bindet x
   ((lambda (f)           ; Umgebung test2 bindet f
      ((lambda (x)        ; Umgebung g bindet x
         (funcall f 1)    ; Aufruf von f benutzt ein x
       )
       42                 ; gebunden an g.x
      )
    )
    (lambda (n) x)        ; gebunden an test2.f, benutzt x
   )
 )
 21                       ; gebunden an test1.x
)
)
```

Ergebnis bei statischer oder bei dynamischer Bindung?

Bindungen und Closures

Definition (freie Variable): Wenn ein **Name x innerhalb einer Funktion f nicht** durch eine Parameterdefinition oder eine lokale Definition **gebunden** ist, dann bezeichnet x eine **freie Variable bezüglich der Funktion f**.

Beispiele:

```
fun f (a, b) = a*x+b
fn (a, b) => a*x+b

(defun f (a b) (+ (* a x) b))
(lambda (a b) (+ (* a x) b))
```

Beim **Aufruf** einer Funktion f werden ihre freien Variablen je nach statischer oder dynamischer Bindung in der Definitions- oder Aufrufumgebung gebunden.

Funktionen können **als Daten** verwendet werden: Funktionen als Parameter, Ergebnis, Komponente von zusammengesetzten Werten, Wert von Variablen (imperativ).
Für den Aufruf benötigen sie eine Closure:

Die **Closure** einer Funktion f ist eine **Menge von Bindungen**, in der beim Aufruf von f die **freien Variablen von f gebunden** werden.

Dynamische Bindung: Closure liegt im Aufrufkeller.
Statische Bindung: Closure ist in der Kette der **statischen Vorgänger** enthalten; diese müssen ggf. auf der Halde statt im Laufzeitkeller gespeichert werden, da Schachteln (und deren Variablen) noch benötigt werden, wenn ihr Aufruf beendet ist