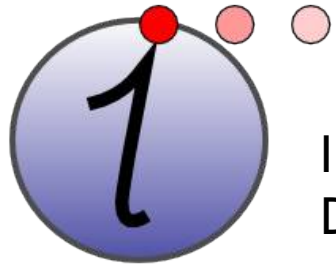




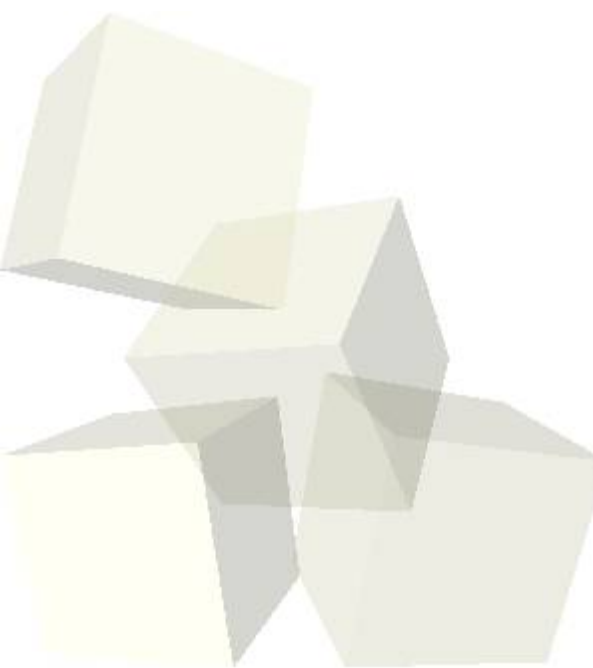
1.3.2006



Informatikbüro
Dipl.-Inform. Gerd Stolpmann

Symbolische Programmierung

Gerd Stolpmann





- Symbolische Datenverarbeitung: Was ist das?
- Symbolische Programmierung ist **das** Anwendungsgebiet der funktionalen Programmierung (FP):
FP-Programmiersprache Objective Caml
- Wir entwickeln ein symbolisches Programm
- Wissenswertes über FP



■ Merkmale:

- ◆ Komplexe innere Struktur (Hierarchien, Beziehungen)
- ◆ Lassen sich nur schlecht in Tabellen oder Arrays darstellen
- ◆ Verarbeitung ist regelgesteuert

■ (Historisches) Beispiel:

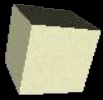
Numerische Integration vs. symbolische Integration

- Hohes Fehlerpotenzial bei der Programmierung: Ergebnis muss auch noch stimmen, wenn viele Regeln nacheinander angewendet werden



Beispiele Symbolische DV

Daten	Verarbeitung
Mathematische Formel	Ableitung bilden
Computerprogramm	Übersetzung in Maschinencode
Spezifikation einer diskreten Fourier-Transformation	Erzeugung eines optimalen Programms (FFTW: http://www.fftw.org)
Netz voneinander abhängiger Finanzverträge	Szenarien durchrechnen (LexiFi: http://www.lexifi.com)
XML-Dokument	Transformation in eine Web-Seite (viele Beispiele, etwa XSLT, akademisch: CDUCE - http://www.cduce.org)
Spezifikation eines Programms	Synthese des Programms



- Funktionale Programmiersprache auf ML-Basis (ML = MetaLanguage, von Robin Milner ca. 1973)
- Symbolische Daten können unmittelbarer dargestellt werden als in herkömmlichen Sprachen
- Interessante Eigenschaften:
 - ◆ Statisch typisiert (innovativ: Typinferenz)
 - ◆ Polymorphismen erlauben Code-Wiederverwendung
 - ◆ Keine Abstürze möglich (Verzicht auf unsichere Sprachmerkmale)
 - ◆ Vollautomatische Speicherverwaltung
 - ◆ Schnelle Ausführung (Übersetzung in Assembler)
 - ◆ ML ist einzige Sprache mit mathematisch definierter Bedeutung

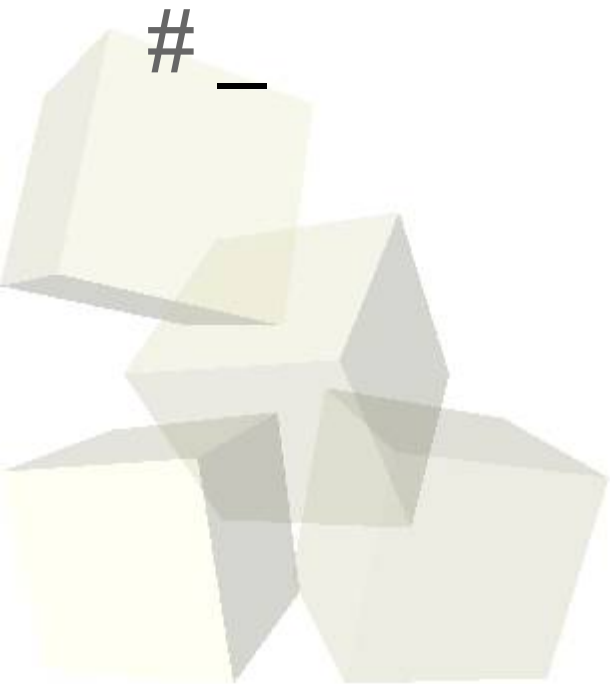


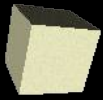
- Die Toploop ist eine interaktive Sitzung, in der Definitionen und Programme sofort ausgeführt werden:

```
$ ocaml
```

```
Objective Caml version 3.08.4
```

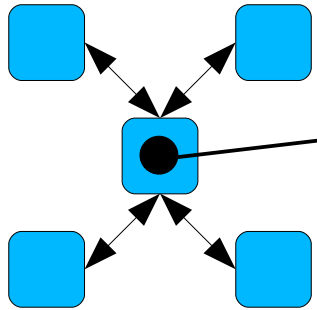
```
# _
```





Was heißt „funktional“?

Imperative Programmierung:



Zuweisung:

Variable := Wert

Variable und
Beziehungsgeflecht

Kritik: Effekt der Zuweisung auf das Beziehungsgeflecht, in der eine Variable steht, ist vom Programmierer schwer zu übersehen!

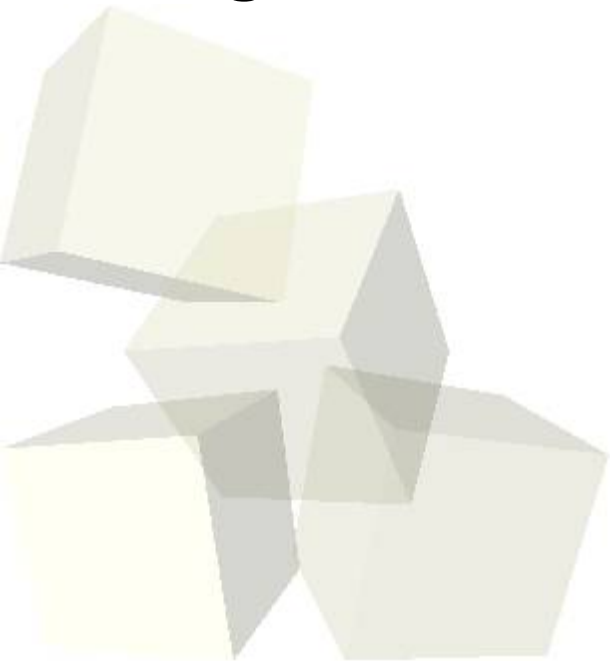
Funktionale Programmierung:

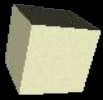
- **Verzicht auf Zuweisung!**
- „Veränderung“ einer Datenstruktur wird nur durch die Erzeugung neuer Versionen der gesamten Struktur ausgedrückt
- ... starker Effekt auf Programmierstil
- ... hohe Anforderungen an Compiler und Runtime



Ein symbolisches Programm

- Aufgabe:
 - ◆ Stelle Formeln mit $+ - \times \div x^y$ dar
 - ◆ Funktionsplotter
 - ◆ Symbolisches Ableiten von Formeln
- Entwicklung in O'Caml mit Hilfe von **Algebraischen Datentypen**





■ Einfaches:

- ♦ Ganzzahlen: 1, 2, 3, ... (Typ int)
- ♦ Fließkommazahlen: 3.1415 (Typ float)
- ♦ Strings: "Eine Zeichenkette" (Typ string)

■ Konstanten:

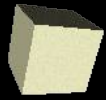
- ♦ `let pi = 3.1415` (global)
- ♦ `let pi = 3.1415 in <code>` (nur in diesem Scope)

■ Funktionen:

- ♦ `let f x = 3 * (x-1)` (Definition)
- ♦ `f 42` (Aufruf – ohne Klammern!)
- ♦ `f (f 42)` (so wird geklammert)
- ♦ `let f x y = x + y` (zwei Argumente)

■ Rekursive Funktionen:

- ♦ `let rec fak n =
 if n = 1 then 1 else n*(fak(n-1))`



Algebraische Typen – O'Caml 2

- Aufzählung:

```
type frucht = Apfel | Birne | Banane
```

- Mits Tags versehene Werte:

```
type währung = Euro of float | Dollar of float
```

z.B.

Euro 5.0 (Float mit Tag Euro)

Dollar 7.82 (Float mit Tag Dollar)

nach_euro (Dollar 7.82) (Funktionsaufruf)

- Pattern Matching:

```
let addiere_währung x y =
```

```
  match (x,y) with
```

```
  | (Euro x_e, Euro y_e)    -> Euro (x_e +. y_e)
```

```
  | (Dollar x_d, Dollar y_d) -> Dollar (x_d +. y_d)
```

```
  | _ -> failwith „Gemischte Währungsaddition nicht unterstützt“
```



■ Definition:

(rekursiv!)

type formel =

- | Add of formel * formel
- | Sub of formel * formel
- | Mul of formel * formel
- | Div of formel * formel
- | Pot of formel * formel
- | Zahl of float
- | X

(* = Tupel)

■ Beispiel:

$2x+1$

Add(Mul(Zahl 2.0, X), Zahl 1.0)



Parser und Pretty Printer

- Laden eines Printers:

```
# #use „formelprinter.ml“;;
```

```
...
```

```
# let f = Add(Mul(Zahl 2.0, X), Zahl 1.0);;
```

```
f : formel = <:formel< 2x+1 >>
```

- Warum diese Schreibweise? Weil man dafür auch einen Parser definieren kann:

```
# let f = <:formel< 2x+1 >>;;
```

```
f : formel = <:formel< 2x+1 >>
```



■ Berechne den Wert der Formel für ein X:

let rec berechnen formel x =

match formel with

Add (formel1, formel2)	-> berechnen formel1 x +. berechnen formel2 x
Sub (formel1, formel2)	-> berechnen formel1 x -. berechnen formel2 x
Mul (formel1, formel2)	-> berechnen formel1 x *. berechnen formel2 x
Div (formel1, formel2)	-> berechnen formel1 x /. berechnen formel2 x
Pot (formel1, formel2)	-> berechnen formel1 x **. berechnen formel2 x
Zahl z	-> z
X	-> x

- Funktion „berechnen“ hat zwei Argumente: formel und x.
- Die Operatoren +. -. *. /. ** sind Addition, Subtraktion, Multiplikation, Division und Potenzierung für floats.
- Beispiel:
berechnen <:formel< 2x+1>> 3.0 ergibt 7.0



- Von mir vordefiniert...
- Aufruf:
plot (berechne <:formel< $2x+1$ >>) (-5.0) 5.0
- Die Besonderheit bemerkt?
- „berechne“ wird nur mit einem Argument aufgerufen: Partieller Aufruf!
- „plot“ ruft das bereits partiell aufgerufene „berechne“ auf („Callback“), jeweils für die x-Werte, für die etwas gezeichnet werden soll



- Ableiten ist eine regelgestützte Formel-Transformation

<u>Original</u>	<u>Ableitung</u>	
c	0	($c = \text{Konstante}$)
$f + g$	$f' + g'$	
$c f$	$c f'$	
$f g$	$f' g + f g'$	
$f : g$	$(f' g - f g') : g^2$	
(usw.)		

- Ziel: Möglichst wiedererkennbare Umsetzung in Programm, um Korrektheit zu gewährleisten.



■ Umsetzung:

```
let rec ableiten formel =  
  match formel with  
  (* Regel:  $c' = 0$  ( $c = \text{konst.}$ ) *)  
  | Zahl z -> Zahl 0.0  
  
  (* Regel:  $x' = 1$  *)  
  | X -> Zahl 1.0  
  
  (* Regel:  $(f \pm g)' = f' \pm g'$  *)  
  | Add (formel1, formel2) -> Add (ableiten formel1, ableiten formel2)  
  | Sub (formel1, formel2) -> Sub (ableiten formel1, ableiten formel2)  
  
  (* Regel:  $(c f)' = c f'$  ( $c = \text{konst.}$ ) *)  
  | Mul (Zahl z, formel2) -> Mul (Zahl z, ableiten formel2)  
  | Mul (formel1, Zahl z) -> Mul (ableiten formel1, Zahl z)  
  
  (* Regel:  $(f g)' = f' g + f g'$  *)  
  | Mul (formel1, formel2) ->  
    Add (Mul(ableiten formel1, formel2), Mul(formel1, ableiten formel2))  
  
  ..... (usw.)
```

■ Aufruf:

```
# ableiten <:formel< 2x+1 >>;;
```

```
- : formel = <:formel< 2 1 + 0 >>
```

```
# ableiten <:formel< (x+1)2 + 2x >>;;
```

```
- : formel = <:formel< 2 (x + 1)1 (1 + 0) + 2 1 >>
```

```
# ableiten <:formel< (x3 - 5)2 >>;;
```

```
- : formel = <:formel< 2 (x3 - 5)1 (3 x2 - 0) >>
```

(Hinweis: „Malpunkt“ wird weggelassen, d.h.
 $2 \ 1 = 2 \bullet 1$)



Ist das alles praxisrelevant?

- Funktionale Programmierung ist ungewohnt, aber:
 - ◆ Sehr kompakter Programmierstil
 - ◆ Integrität und Korrektheit haben Vorrang
 - ◆ Programme sind änderungsfreundlich
 - ◆ Lücke zwischen „Design“ und Implementierung wird durch FP-Einsatz kleiner
 - ◆ Kombinierbar mit imperativen Programmen
 - ◆ Exzellenter Compiler (nahezu fehlerfrei);
Exzellente Laufzeitumgebung (teilvalidiert)



Wer nutzt FP-Sprachen?

- Akademischer Bereich:
 - ◆ „Künstliche Intelligenz“
 - ◆ zunehmend im Compilerbau, Programmgeneratoren usw.
 - ◆ Scientific Computing, z.B. in der Bioinformatik
- Open Source: z.B.
 - ◆ Unison: Portabler Dateisynchronisierer
 - ◆ MLDonkey: Multi-Protokoll Filesharing-Client
 - ◆ Darcs: Versionskontrollsystem
- Industrie: z.B.
 - ◆ Intel: Chipverifikation
 - ◆ Microsoft: Treiberverifikation
 - ◆ Ericsson: Netzwerkkomponenten
 - ◆ Baretta: Steuerung von Schneidemaschinen



- Objective Caml: <http://caml.inria.fr>
- Tutorial O'Caml: <http://www.ocaml-tutorial.org>
- FP-Proseminar:
<http://www.plm.eecs.uni-kassel.de/plm/fileadmin/pm/courses/prog-seminar/funk-ml.pdf>
- FP und XML:
<http://www.xml.com/pub/a/2001/02/14/functional.html?page=1>

