Programmiersprachen

Alexandru Berlea

Insitut für Informatik
TU München

Wintersemester 2006/2007

Part I

Einleitung

Inhalt der Vorlesung

- **Deklarative Programmierparadigmen**
  - **Funktionale** Programmierung (SML, Haskell)
  - **Logische** Programmierung (Prolog)
  - **Constraint**-Programmierung (Prolog-Erweiterungen)

Inhalt der Vorlesung

- **Deklarative Programmierparadigmen**
  - **Funktionale** Programmierung (SML, Haskell)
  - **Logische** Programmierung (Prolog)
  - **Constraint**-Programmierung (Prolog-Erweiterungen)
- **Programmierkonzepte**, z.B.:
  - Generische Programmierung
  - Typ-Inferenz
  - Ausnahmen
  - Continuation Passing Style
  - Lazy-Evaluation

Organisation

- **Voraussetzung für Teilnahme**: Vordiplom
- **Voraussetzung für Schein**: Schriftliche Klausur
- **Übung**: Do 14:15-15:45 im Raum MI 02.07.014
  Erster Termin: 2. November

Motivation

- Verbesserte **Ausdrucksstärke**
- Verbesserung der Fähigkeit, **neue Sprachen** zu lernen
- **Effizienter/e Programme** schreiben
- Identifikation der für das jeweilige Problem **passende Sprache**
- Verbesserung der Fähigkeit, neue **Sprachen** zu **entwickeln**

Lernziele

Was wir lernen

- **Programmieren** in funktionalem, logischem und constraint-basiertem Stil
- **Vor- und Nachteile** der verschiedenen Paradigmen
- **Konzepte** in Sprachen zu erkennen und auszunutzen

Lernziele

Was wir lernen

- **Programmieren** in funktionalem, logischem und constraint-basiertem Stil
- **Vor- und Nachteile** der verschiedenen Paradigmen
- **Konzepte** in Sprachen zu erkennen und auszunutzen

Was wir nicht lernen:

- einzelne Programmiersprachen im Detail
- Implementierungstechniken für Programmiersprachen (☞ Vorlesungen Compilerbau, Abstrakte Maschinen)

Literatur

- J. Mitchel, Concepts in Programming Languages, Cambridge
- R. W. Sebesta, Programming Languages, Addison-Wesley
- R. Sethi, Programming Languages, Addison-Wesley
- L. Paulson, ML for the working programmer, Cambridge
- T. Frühwirth, Constraint-Programmierung, Springer
- L. Sterling, The Art of PROLOG, MIT Press

Programmierung: kurze Chronologie

- **Maschinen-Code** (Ende 1940)
  - Program = Folgen von Bits, die als Anweisungen interpretiert werden
- **Assembler-Sprachen** (Anfang 1950)
  - Symbolische Namen für Anweisungen
- **Fortran** (**For**mula **Tran**slation) (1954)
  - Mathematische Notation für Ausdrücke, z.B. `1 + 2*3`
  - Symbolische Namen für Variablen
  - Iteration
  - Unterprogramme
- **ALGOL** (**ALGO**rithmic **L**anguage) (Ende 1950)
  - Datentypen
  - Variablen-Scopes
  - rekursive Unterprogramme

Programmierung: kurze Chronologie (Forts.)

- **COBOL** (**CO**mmon **B**usiness **O**riented **L**anguage) (1960)
  - Hierarchische Datenstrukturen
  - Syntax ähnlich wie in Englisch

- **LISP** (**LIS**t **P**rocessing) (Ende 1950)
  - Verarbeitung symbolischer (nicht-numerischer) Daten als verkettete Listen
  - Funktional:
    Programm = Funktionsdefinitionen
    Berechnung = Funktionsanwendungen, keine Zuweisungen
  - Funktionen höherer Ordnung
  - Garbage Collection

- **SIMULA** (1967)
  - Korutinen
  - Klassen

Programmierung: kurze Chronologie (Forts.)

- **Prolog** (**Pro**gramming **Log**ic) (1972)
  - Logische P.:
    Programm = logische Formel
    Berechnung = Deduktion
- **ML** (**M**eta **L**anguage) (1980)
  - Typ-Inferenz
- **Ada** (nach Ada Lovelace) (1983)
  - Generische Programmeinheiten
  - Unterstützung für Nebenläufigkeit
- **Prolog-III** (1984)
  - Constraint-Programmierung

Programmierung: kurze Chronologie (Forts.)

- Vieles mehr, z.B. **BASIC** (1964), **C**, **Pascal** (1971), **Smalltalk** (1980), **C++** (Anfang 1980), **Haskell** (1992), **Java** (1994), **C#** (2002)

- Mehr:
  Stammbaum der 50 bekanntester Programmiersprachen
  (☞ Éric Lévénez Site/O'Reilly-Poster in der Magistrale)
  Information über mehr als 2500 Sprachen
  (☞ Bill Kinnersles Site)

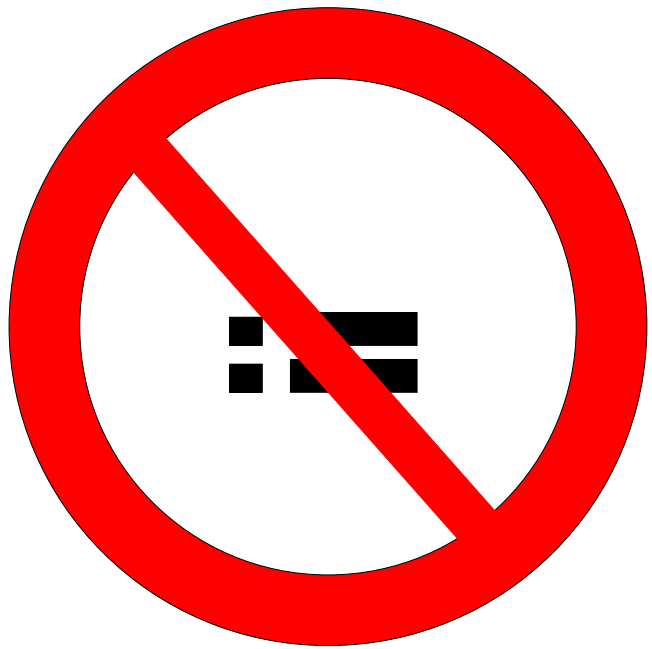
Programmierung: kurze Chronologie (Forts.)

Fazit:

- Immer höhere Abstraktion der unterliegenden Rechenmaschine
- Tendenz zur erhöhten **Deklarativität**
  - Ideal: Beschreibe Ergebnis statt Berechnung, die zum Ergebnis führt
  - **Funktionale**, **logische** u. **Constraint-basierte** Sprachen unterstützen einen deklarativen Stil.

Part II

Funktionale Programmierung



Überblick

1. **Einleitung**
   Funktionale vs. traditionelle Programmierung
   Grundlegende Merkmale der Funktionalen Programmierung
   Weitere, typische Merkmale der FP

Traditionelle Programmierung

- **Berechnungsbeschreibung** = Folge von Befehlen
  - imperativ (Lat. imperare = befehlen)
- **Berechnungsausführung** = Folge von Zustandsänderungen
  - Zustand $\equiv$ Inhalt der Speicherzellen
  - Zustandsänderung durch Zuweisung: *Speicherzelle* := *Wert*
  - zuweisungsorientiert
- eng verknüpft mit der zugrunde liegenden Rechen-Maschine (von Neumanns Modell):
  - CPU arbeitet Befehl nach Befehl sequentiell ab
  - Speicher dient zur Zustandsprotokollierung

Funktionale Programmierung (FP)

- **Berechnungsbeschreibung** = Fkt.-Definition/Deklaration (im mathematischen Sinne)
  - $f : E \mapsto A, f(x) = x + 1$
  - deklarativ
  - funktional
- **Berechnungsausführung** = Funktionsanwendung
  - $f(4)$
  - applikativ
- Abstrahiert von der zugrunde liegenden Rechen-Maschine

Ausdrucksstärke

- Formale Ausdrucksstärke:
  - Imperative Programmiersprachen $\mapsto$ **Turing-Maschine**
  - FP $\mapsto$ **Lambda-Kalkül**
  - Turing-Maschine = Lambda-Kalkül = berechenbare Fkt.
- Praktische Ausdrucksstärke:
  - werden wir näher später betrachten

Grundlegende Merkmale der FP

- Keine Seiteneffekte
- Referentielle Transparenz
- Funktionen sind Werte erster Klasse
- Hohe Abstraktion der zugrunde liegenden Rechenmaschine

Eine Funktion hat keine Seiteneffekte, wenn...

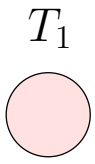
- ... der Aufruf der Funktion mit dem selben Parameter liefert immer das selbe Ergebnis
- d.h. Aufruf $\equiv$ Anwendung einer mathematischen Funktion
- 1. Vorteil:
  Unabhängigkeit von der Auswerttungsreihenfolge
  (z.B. der Parameter)

Ein Beispiel: Fibonaccibäume

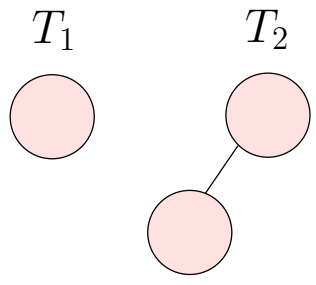
- **Definition**:
  - Der leere Baum ist ein Fibonacci-Baum der Höhe 0
  - Ein einzelner Knoten ist ein Fibonacci-Baum der Höhe 1
  - Sind T_{h-1} und T_{h-2} Fibonacci-Bäume der Höhen $h-1$ und $h-2$, so ist $T_h = k < T_{h-1}, T_{h-2} >$ ein Fibonacci-Baum der Höhe h.

- sind ein **Speziallfall von AVL-Bäume**
  - Binärbäume, bei denen an jedem inneren Knoten der Höhenunterschied zwischen dem rechten und linken Teilbaum maximal 1 ist

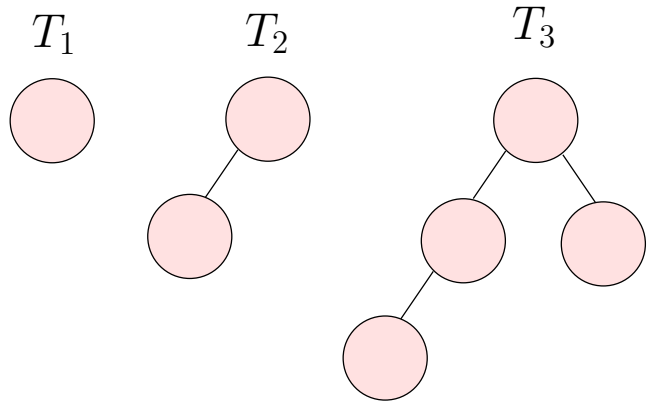
Ein Beispiel: Fibonaccibäume



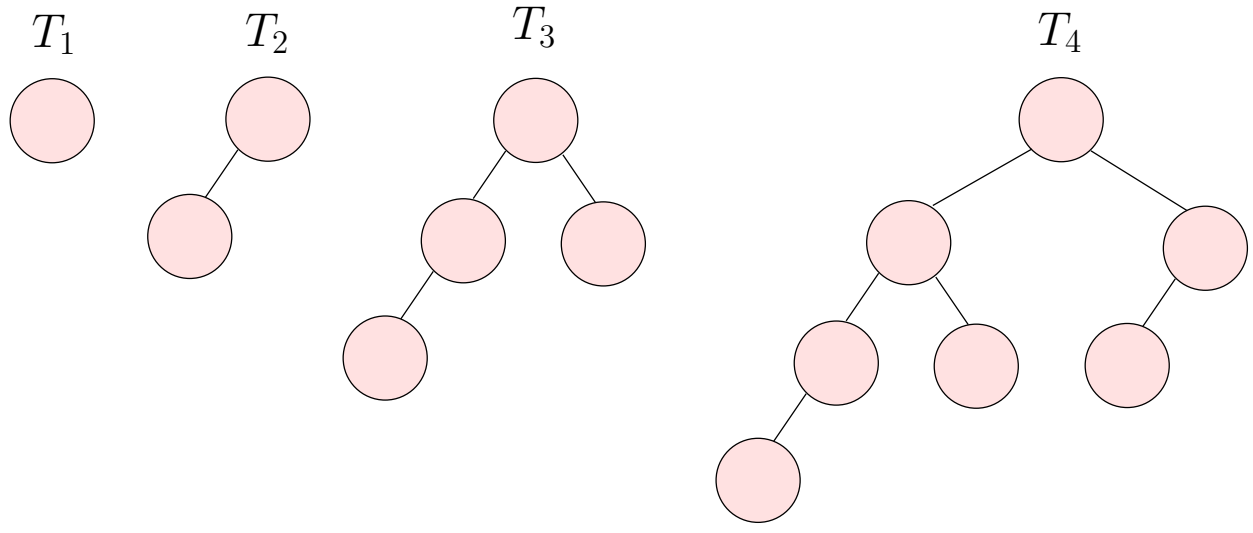
Ein Beispiel: Fibonaccibäume



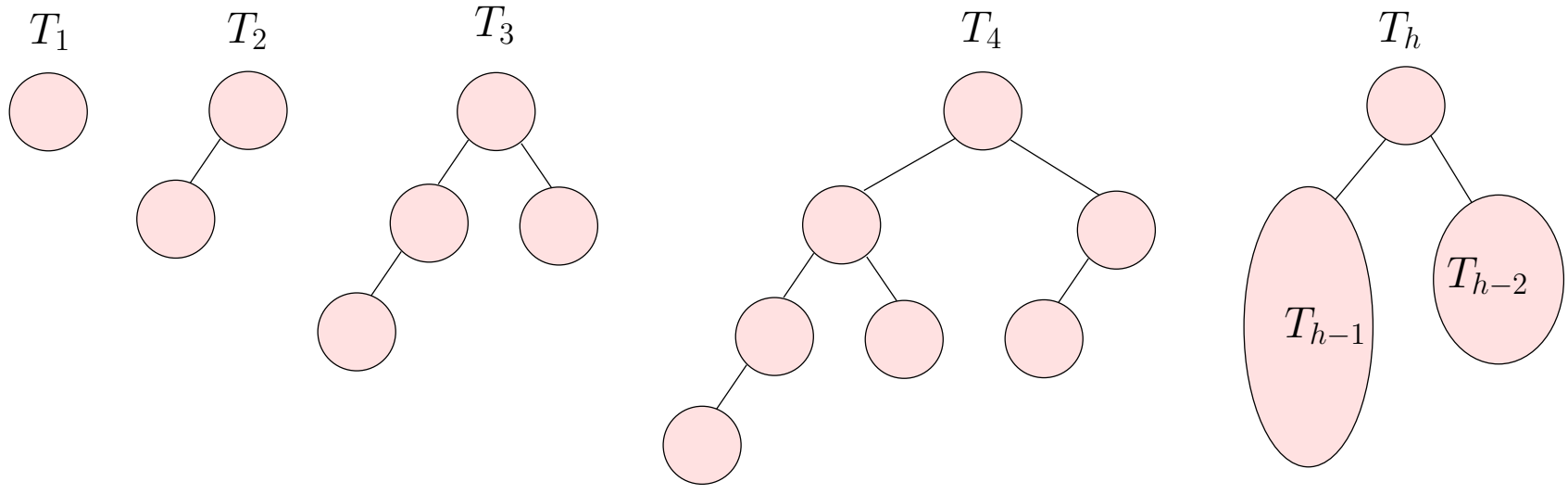
Ein Beispiel: Fibonaccibäume



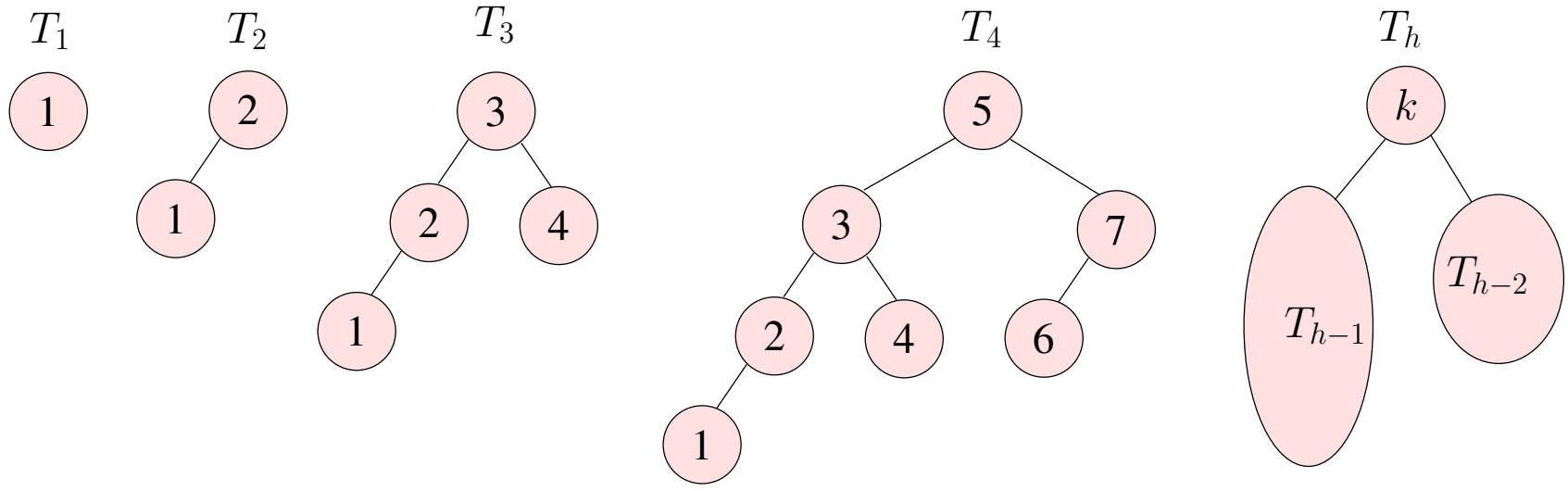
Ein Beispiel: Fibonaccibäume



Ein Beispiel: Fibonaccibäume



Natürlich annotierte Fibonaccibäume



..

sind ein **Spezialfall von binären Suchbäumen**:

- Für jeden Knoten v gilt, dass alle Knoten im linken Teilbaum kleinere Werte und alle Knoten im rechten Teilbaum größere Werte als v haben.

Natürlich annotierte Fibonaccibäume

Aufgabe: Aufbau eines natürlich annotierten Fibonacci-Baumes mit gegebener Höhe:

Lösung: **mit Seiteneffekten** (imperativ, in Java)

```
class FibTree{
    public int k;
    public FibTree l,r;

    FibTree(FibTree left, int key, FibTree right){
        k = key;
        l = left;
        r = right;
    }

    .......
```

Natürlich annotierte Fibonaccibäume

Lösung mit Seiteneffekten

```
.....

static int m = 1;

static FibTree makeFibTreeHelp(int h){
    if (h<=0) return null;
    else return new FibTree(makeFibTreeHelp(h-1),
                            m++,
                            makeFibTreeHelp(h-2));
}

static FibTree makeFibTreeImperativ(int h){
    m=1;
    return makeFibTreeHelp(h);
}
```

Nachteile der Lösung mit Seiteneffekten

```
static FibTree makeFibTreeHelp(int h){
  .........

        return new FibTree(makeFibTreeHelp(h-1),
                           m++,
                           makeFibTreeHelp(h-2));
}
```

- Ergebnis **abhängig von Reihenfolge der Auswertung**
  - Java spezifiziert die Auswertung von links nach rechts
  - C/C++ Compiler können eine beliebige Reihenfolge wählen
    $\Longrightarrow$ das Ergebnis ist nicht deterministich

Nachteile der Lösung mit Seiteneffekten

```
static FibTree makeFibTreeHelp(int h){
  .........

        return new FibTree(makeFibTreeHelp(h-1),
                           m++,
                           makeFibTreeHelp(h-2));
}
```

- Ergebnis **abhängig von Reihenfolge der Auswertung**
  - Java spezifiziert die Auswertung von links nach rechts
  - C/C++ Compiler können eine beliebige Reihenfolge wählen
    $\Longrightarrow$ das Ergebnis ist nicht deterministich
- `makeFibTreeHelp` ist keine Funktion im mathematischen Sinne, weil sie Seiteneffekte hat
  - schwer zu verstehen
  - schwer zu beweisen, dass sie einen Fibonacci-Suchbaum konstruiert

Beweisideee für die imperative Lösung

```
static FibTree makeFibTreeHelp(int h){
   .........

          return new FibTree(makeFibTreeHelp(h-1),
                                m++,
                                makeFibTreeHelp(h-2));
}
```

- Die Funktion simuliert einen **in-order** Durchlauf
- plausibel, aber kein gültiger formaler Beweis

Lösung ohne Seiteneffekte (in Java)

Hilfsfunktion `maxKey` berechnet den maximalen Schlüssel in einem binären Suchbaum:

```
static int maxKey(FibTree t){
    if (t == null) return -1;
    if (t.r == null) return t.k;
    return maxKey(t.r);
}
```

Lösung ohne Seiteneffekte (in Java)

```
static FibTree t(int h, int k){
    if (h == 0) return null;
    if (h == 1) return new FibTree(null,k,null);
    return new FibTree(t(h-1,k),
                       maxKey(t(h-1,k))+1,
                       t(h-2,maxKey(t(h-1,k))+2));
}
```

Behauptung:

t(h,k) ist ein Suchbaum mit kleinstem Schlüssel k, $\forall h > 0$.

Beweis:

Induktion über *h* durch Einsetzen der Funktionsdefinition.

Effizientere seiteneffektfreie Lösung

Ursprüngliche Variante

```
static FibTree t(int h, int k){
    if (h == 0) return null;
    if (h == 1) return new FibTree(null,k,null);
    return new FibTree(t(h-1,k),
                       maxKey(t(h-1,k))+1,
                       t(h-2,maxKey(t(h-1,k))+2));}
```

Effizientere Variante:

```
static FibTree t(int h, int k){
    if (h == 0) return null;
    if (h == 1) return new FibTree(null,k,null);
   FibTree l = t(h-1,k);
    return new FibTree(l,
                       maxKey(l)+1,
                       t(h-2,maxKey(l)+2));}
```

Effizientere seiteneffektfreie Lösung

Ursprüngliche Variante

```
static FibTree t(int h, int k){
    if (h == 0) return null;
    if (h == 1) return new FibTree(null,k,null);
    return new FibTree(t(h-1,k),
                       maxKey(t(h-1,k))+1,
                       t(h-2,maxKey(t(h-1,k))+2));}
```

Effizientere Variante:

```
static FibTree t(int h, int k){
    if (h == 0) return null;
    if (h == 1) return new FibTree(null,k,null);
   FibTree l = t(h-1,k);
    return new FibTree(l,
                       maxKey(l)+1,
                       t(h-2,maxKey(l)+2));}
```

Es geht noch effizienter: statt maxKey zu benutzen, lass t gleichzeitig den maximalen Schlüssel im konstruierten Baum liefern. (☞ Übung)

Referentielle Transparenz

- **Referential transparency** =

  der Wert eines Programmausdruckes hängt nur von den Werten der Teilausdrücken und nicht vom Kontext der Auswertung

Referentielle Transparenz

- **Referential transparency** =

  der Wert eines Programmausdruckes hängt nur von den Werten der Teilausdrücken und nicht vom Kontext der Auswertung

  [$\Longrightarrow$ eine (implizite) Referenz zum (dynamischen) Kontext der Auswertung ist nicht nötig/sichtbar]

- Allgemeiner: der Wert eines Funktionsaufrufs hängt nur von den Werten der aktuellen Parameter, d.h.:

  keine Seiteneffekte $\Longrightarrow$ referenzielle Transparenz

Beispiel: Keine Referentielle Transparenz

```
class Opaque{
    public static int x = 1;
    static int f(){return x;}

    public static void main(String[] a){
        System.out.println("f() liefert "+f());
        x=2;
        System.out.println("f() liefert "+f());
    }
}
```

Ausgabe: erst 1 dann 2.

Beispiel: Referentielle Transparenz/Keine RT

- Funktionale Programmierung (SML):

  `e + e` $\equiv$ `let y = e in y + y end`

- Imperative Programmierung (Java/C++):

  `return (x++ + x++)` $\not\equiv$ `y = x++; return (y + y)`

Folgen der referentiellen Transparenz

- **Korrektheit:** formale Eigenschaften mit klassischen mathematischen Verfahren einfacher beweisen

Folgen der referentiellen Transparenz

- **Korrektheit:** formale Eigenschaften mit klassischen mathematischen Verfahren einfacher beweisen

- **Effizienz:** Optimierungen durch Compiler möglich, z.B.:
  - Auswertungsreihenfolge ist nicht wichtig
  - Parallele Auswertung der Teilausdrücke möglich
  - gleichwertige Ausdrücke nur einmal auswerten (*common sub-expression elimination*) [☞ Programm-Optimierung]

Folgen der referentiellen Transparenz

- **Korrektheit:** formale Eigenschaften mit klassischen mathematischen Verfahren einfacher beweisen
- **Effizienz:** Optimierungen durch Compiler möglich, z.B.:
  - Auswertungsreihenfolge ist nicht wichtig
  - Parallele Auswertung der Teilausdrücke möglich
  - gleichwertige Ausdrücke nur einmal auswerten (*common sub-expression elimination*) [☞ Programm-Optimierung]
- **Wartbarkeit:**
  - Wenn eine Funktion einmal richtig funkioniert hat, dann funktioniert sie immer richtig, unabhängig vom Auswertungskontext
  - bessere Lesbarkeit

Grenzen der puren funktionalen Programmierung

- Manche Berechnungen sind **inhärent nicht referenziell transparent**, z.B.:
  - **Ein- und Ausgabe**

```
x = read(); y = read();
```

  - **Zeit-Funktionen**

```
t1 = getCurrentTime(); t2 = getCurrentTime();
```

  - **Zufallsgeneratoren**

Grenzen der puren funktionalen Programmierung

- Manche Berechnungen sind **inhärent nicht referenziell transparent**, z.B.:
  - **Ein- und Ausgabe**

```
x = read(); y = read();
```

  - **Zeit-Funktionen**

```
t1 = getCurrentTime(); t2 = getCurrentTime();
```

  - **Zufallsgeneratoren**

- Deshalb werden wir auch **impures** FP betrachten, d.h. Erweiterungen der FP mit imperativen Konzepten.

Funktionen als Werte erster Klasse

- Funktionen sind *first-class objects* wenn sie:
  - als Parameter übergeben werden können;
  - als Rückggabewerte von Funktionen zurückgeliefert werden können.

  [d.h. sie sind ein Wert wie jeder andere Wert auch.]

- **Funktion höherer Ordnung**: eine Funktion, die Funktionen als Argumente bekommt oder eine Funktion als Ergebnis liefert.

Beispiel: Funktionskomposition

- Versuch in C:

```
int comp(int (*f)(int),int (*g)(int),int x){
  return (*f)((*g)(x));
}

int inc(int x){return x+1;}

main(){
  printf("res=%i\n",comp(inc,inc,3));
}
```

Ausgabe: `res=5`

Beispiel: Funktionskomposition

- **Versuch in C:**

```
int comp(int (*f)(int),int (*g)(int),int x){
  return (*f)((*g)(x));
}

int inc(int x){return x+1;}

main(){
  printf("res=%i\n",comp(inc,inc,3));
}
```

Ausgabe: `res=5`

- Probleme:
  - $comp(f, g, x)$ liefert keinen Funktionswert: erwünscht wäre $f \circ g$ statt $f(g(x))$
  - nicht generisch: *comp* kann nicht beliebige Funktionen komponieren.

Beispiel: Funktionskomposition

- **Lösung in SML:**
  - Definition: `fun comp (f,g) = fn x => f(g(x))`
  - Anwendung:

```
fun hoch2 x = x*x;
val hoch4 = comp (hoch2,hoch2);
hoch4 3;
```

  - Ausgabe: `81`

Funktionskomposition in Java

```
interface IntFunctionI {
    int run(int i);
}

class Inc implements IntFunctionI {
    public int run(int i){ return i+1;}
}

class IntFunctionComp implements IntFunctionI {
    IntFunctionI f1;
    IntFunctionI f2;

    IntFunctionComp(IntFunctionI f, IntFunctionI g){
        f1=f; f2=g;
    }
    public int run(int i){ return f1.run(f2.run(i));}
}

class TestMain{
    public static void main(String[] a){
        IntFunctionI inc = new Inc();
        IntFunctionI f = new IntFunctionComp(inc,inc);
        System.out.println(f.run(3));
    }
}
```

Funktionskomposition in Java

- aufwändig, (noch) nicht generisch
- aber, **funktionale Konzepte:**
  - lassen sich auch in imperativen Sprachen (mehr oder weniger aufwändig) implementieren;
  - **helfen, die passende Abstraktion für ein Problem zu finden**.

Hohe Abstraktion des physischen Modells

- **keine explizite Verwaltung von Speicherzellen**
  - Variablen sind keine Speicherzellen sondern Namen für einen Wert

  $\Longrightarrow$ keine explizite Anforderung/Freigabe des benötigten Speichers
  $\Longrightarrow$ Garbage-Collection

- **keine explizite Kontrolle des Kontrollflußes**
  - keine Schleifen $\Longrightarrow$ Iteration durch Rekursion ersetzt

```
fun fact n = if n<=0 then 1
             else n*fact(n-1)
```

Weitere, typische Merkmale der FP-Sprachen

- **Typ-Inferenz:** Typen müssen (meist) nicht explizit spezifiert werden; sie werden automatisch vom Compiler inferiert
  - Deklaration: `fun comp (f,g) = fn x => f(g(x))`
  - Compiler antwortet mit dem inferierten Typ:

    *val comp = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b*

- **Pattern-Matching**
  - zur **Fall-Unterscheidung** für Funktionen durch Muster-Angabe
  - zur **Dekomposition** eines Wertes

Fazit

- Typische Merkmale der FP:
  - Funktionen sind Werte erster Klasse
  - Gute Unterstützung durch das Typ-System
  - (Möglichst) keine Seiteneffekte
  - Hoher Abstraktionsgrad
  - Automatische Speicherfreigabe (*garbage collection*)
  - Pattern-Matching

Überblick

2. Eine funktionale Sprache: SML, Einleitung

Funktionale Programmierung in SML

- **ML**, 1973 Robin Milner (**M**eta-**L**anguage: die Spezifikationssprache eines Theorem-Beweis-Programms, 1973 - Robin Milner)
- **SML**, 1983 Robin Milner: Versuch, die verschiedenen Dialekte von ML zu standardisieren (Standard: SML'97)
- SML-Compiler: SML/NJ, MoscowML, Poly/ML, MLton, SML.NET
- **SML/NJ** = Standard-Implementierung
- Verwandte Sprachen: Caml, OCaml

Struktur eines Programms

- **Programmspezifikation:** Menge von Wert-Definitionen.
- **Programmausführung:** Auswertung eines Wertes.

Die Interpreter-Umgebung

Die SML/NJ Interpreter-Umgebung wird mit `sml` aufgerufen...

```
~/>sml
Standard ML of New Jersey, Version 110.0.7
-
```

Definitionen von Variablen, Funktionen u.s.w können direkt eingegeben werden.
Alternativ kann man sie aus einer Datei einlesen:

```
- use "test.sml";
[opening test.sml]
val it = () : unit
```

Die Interpreter Umgebung - Ausdrücksauswertung

```
- 1+2;
val it = 3 : int
- 1+
= 2;
val it = 3 : int
```

- Bei `-` wartet der Interpreter auf Eingabe.
- Bei unvollständiger Eingabe bittet `=` um weitere Eingabe.
- Das `;` bewirkt Auswertung der bisherigen Eingabe.
- Das Ergebnis wird berechnet und mit seinem Typ ausgegeben.

Vorteil: Das Testen von einzelnen Funktionen kann stattfinden, ohne jedesmal neu (alles) zu übersetzen ($\mapsto$ inkrementelle Übersetzung)

Variablendefinitionen

- In FP ist eine Variable ein **Bezeichner** für einen Wert (nicht für eine Speicherzelle wie bei der imperativen Programmierung)
- Die Variable behält dann für immer diesen Wert.
- Eine Variable wird mit `val` deklariert und belegt:

```
- val x = 1;
val x = 1 : int
- val y ="Eine Zeichenkette";
val y = "Eine Zeichenkette" : string
- val z = x+1;
val z = 2 : int
```

Variablendefinitionen

Eine erneute Definition für `x` weist nicht `x` einen neuen Wert zu, sondern erzeugt eine neue Variable mit Namen `x`. Dadurch ist die alte Definition nicht mehr sichtbar.

```
- val x = 1;
val x = 1 : int
- val y ="Eine Zeichenkette";
val y = "Eine Zeichenkette" : string
- val z = x+1;
val z = 2 : int
- val x = 20;
val x = 20 : int
- val t = x+1;
val t = 21 : int
```

Funktionsdefinitionen

Ein Funktionswert wird mit Hilfe des Schlüsselsworts `fn` definiert:

- Bsp.: `fn x => x` ist die Identitätsfunktion

- Variablen können Funktionswerte bezeichnen:

```
val identity = fn x => x;
val identity = fn : 'a -> 'a
val increment = fn x => x+1;
val increment = fn : int -> int
```

Funktionen anwenden

Wenn `f` ein Funktionswert und `x` ein Wert aus dem Definitionsbereich von `f` ist, dann ist `f x` (keine Klammern nötig) die Anwendung von `f` auf `x`, also der Wert der Funktion `f` an Stelle `x`.

```
val identity = fn x => x;
val identity = fn : 'a -> 'a
val increment = fn x => x+1;
val increment = fn : int -> int
identity 2;
val it = 2 : int
increment 3;
val it = 4 : int
increment (increment 3);
val it = 5 : int
```

Rekursive Funktionen definieren

- Ein Wert wie `fn x => x` ist eine **namenlose (anonyme)** Funktion $\Longrightarrow$ keine Definition rekursiver Funktionen möglich

- Um rekursive Funktionen zu definieren braucht man nicht-namenlose Funktionen, die mit Hilfe des Schlüsselworts `fun` eingeführt werden:

```
fun fact n = if n<=0 then 1 else n*(fact (n-1))
val fact = fn : int -> int
fact 3;
val it = 6 : int
```

Syntaktischer Zucker

Nicht-rekursive Funktionswerte können sowohl mit `fn` als auch mit `fun` definiert werden. D.h., statt:

```
val identity = fn x => x;
val identity = fn : 'a -> 'a
val increment = fn x => x+1;
val increment = fn : int -> int
```

kann man schreiben:

```
fun identity x = x;
val identity = fn : 'a -> 'a
fun increment x = x+1;
val increment = fn : int -> int
```