Programmiersprachen

Wintersemester 2005/2006

Alexandru Berlea

Lehrstuhl Prof. Seidl

Institut für Informatik
TU München

1 Allgemeines

Inhalt der Vorlesung: Programmierparadigmen

- funktional
- logisch
- imperativ
- objektorientiert

Organisation

- Voraussetzung für Teilnahme: Vordiplom
- Voraussetzung für Schein: Schriftliche Klausur
- Übung: Do 14:15-15:45 im Raum MI 02.07.014
  Erster Termin: 27 Okt.

2 Einleitung

Motivation

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

Lernziele

Was wir lernen:

- Vorteile und Nachteile der verschiedenen Paradigmen
- Programmieren in funktionalem (ML, Haskell), logischem (Prolog), imperativem (C) und objektorientiertem Stil (C++)
- Konzepte in neuen Sprachen zu erkennen und auszunutzen

Was wir nicht lernen:

- einzelne Programmiersprachen im Detail
- Implementierungstechniken für Programmiersprachen

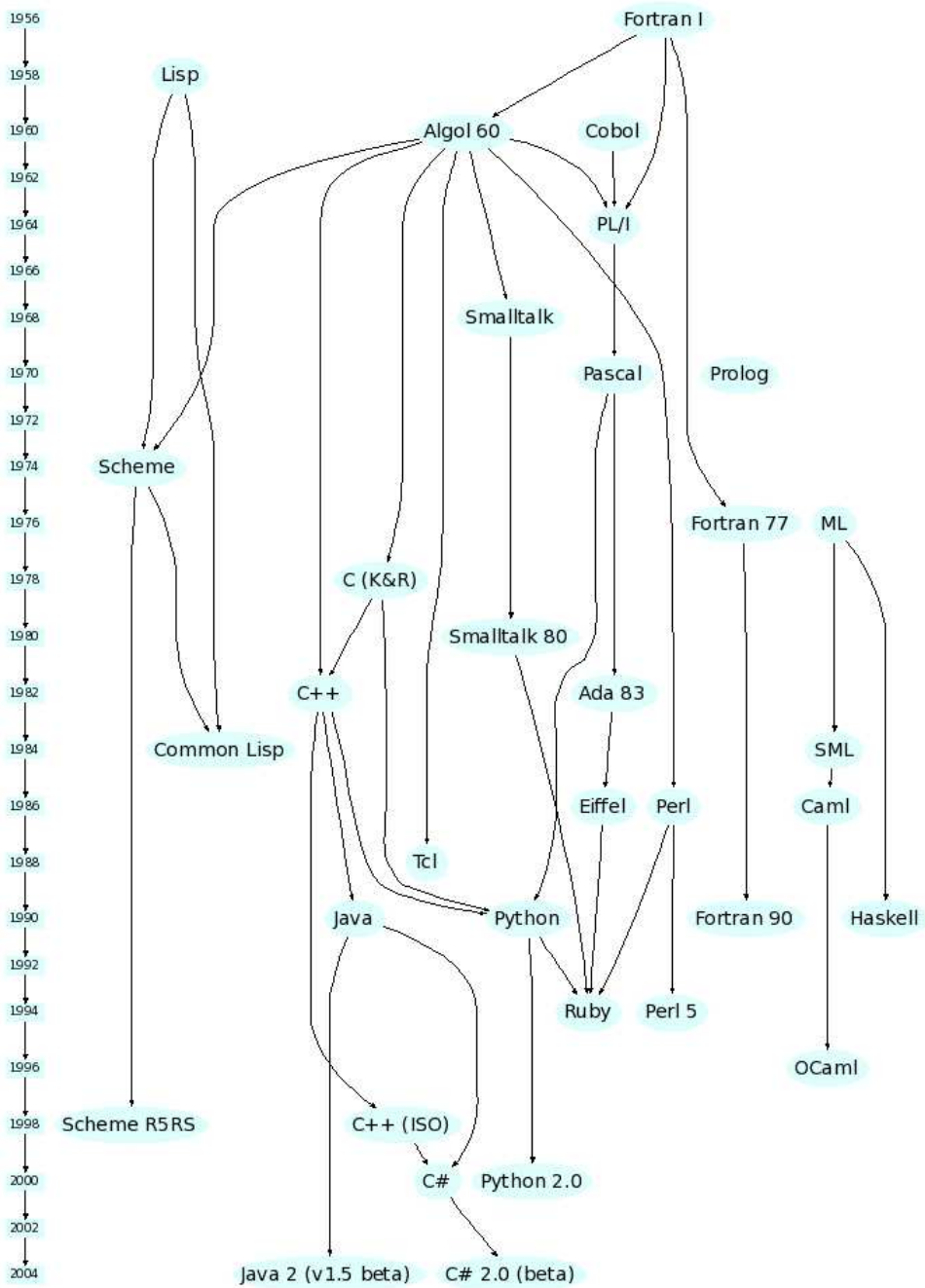
Literatur

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

2.1 Geschichte

Stammbaum

- 50 der meist bekannten Programmiersprachen:
  `http://www.levenez.com/lang/`
- Information über ca. 2500 Programmiersprachen:
  `http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm`


1956
1958
1960
1962
1964
1966
1968
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
Fortran I
Lisp
Algol 60
Cobol
PL/I
Smalltalk
Pascal
Prolog
Scheme
Fortran 77
ML
C (K&R)
Smalltalk 80
C++
Ada 83
Common Lisp
SML
Eiffel
Perl
Caml
Tcl
Java
Python
Fortran 90
Haskell
Ruby
Perl 5
OCaml
Scheme R5RS
C++ (ISO)
C#
Python 2.0
Java 2 (v1.5 beta)
C# 2.0 (beta)

2.1.1 Imperative Sprachen

Maschinen-Code, Ende 1940er, Anfang 1950er

- Program $\equiv$ Folgen von Bits, die als Anweisungen interpretiert werden
- schwer zu lesen und zu warten

Assembler-Sprachen, Anfang 1950er

- Symbolische Namen für Anweisungen
- Übersetzt nach Maschinen-Code durch Assembler-Programme

Plankalkül (1945), Konrad Zuse

- Typen: Bit, Fließkomma, Arrays, Records
- Iteration: `for`-Konstrukt
- Verzweigung: `if`-Konstrukt (ohne `else`)
- Zusicherungen
- nie implementiert; erst 1972 veröffentlicht

Fortran, 1954

- Mathematical FORmula TRANSlation
- IBM 704 (1954): Hardware-Unterstützung für Fließkommazahlen, Indexierung
- John Backus; erster Fortran-Compiler 1957
- Variablennamen (2 bis 6 Zeichen)
- direkte Beziehung zwischen Sprachkonstrukten und Hardware: z.B. – Verzweigung (drei-teilig), ganze Zahlen, Fließkommazahlen, Sprünge
- Iteration (`do ... while`), Unterprogramme
- ...

Fortran, 1954

- ...
- keine Datentyp-Deklarationen: Variablennamen beginnend mit I,J,K,L,M - ganze Zahlen; der Rest - Fließkommazahlen;
- Ein- und Ausgabe
- Optimismus: *eliminate coding errors and the debugging process*
- Speicherplatz für alle Variable wird vor der Laufzeit reserviert $\Longrightarrow$ Effizienz aber ...
- ... keine dynamische Speicherplatz Allokierung $\Longrightarrow$ keine Rekursion
- großen Einfluss auf nachfolgenden Programmiersprachen
- Fortran II, III, IV, 66, 77, 90 und 95

ALGOL, Ende 1950

- ALGOrithmic Language
- Ziel: eine *universale Programmiersprache*: universal = für verschiedene Architekturen (z.T. Angst vor IBM-Vorherrschaft)
- zum ersten Mal: Syntax-Beschreibung mit BNF-Notation (Backus-Naur Form)
- Block-Anweisungen $\Longrightarrow$ neue Variablen-Reichweiten (*scope*)
- Parameter-Übergabe als Werte (*call by value*) und als Namen (*call by name*)
- rekursive Unterprogramme
- ALGOL 58, 60, 68
- großen Einfluss: C, Pascal, Ada, C++, Java, C#

COBOL, 1960

- COmmon Business Oriented Language
- Sprache für Buchführung (*business-applications*)
- hierarchische Datenstrukturen
- Variablen- und Unterprogramm-Deklarationen getrennt
- keine Funktionen (Unterprogramme nur ohne Parameter)
- immer noch weit verbreitet – Ende 1990 “most widely used”

BASIC, 1964

- Beginner's All-purpose Symbolic Instruction Code
- Einfach zu lernen, benutzen und implementieren – "user time is more important than computer time"
- Einziger Datentyp: Fließkommazahlen
- sehr eingeschränkt aber einfach zu lernen
- schlechte Strukturiertheit
- Nachfahren: QuickBASIC, Visual BASIC, VB.NET

PL/I, 1965

- Programming Language I
- Versuch Features von Fortran, ALGOL und COBOL zu kombinieren
- Für beides Businesss- und numerische Anwendungen
- Unterstützung für parallele Ausführung
- Ausnahmen
- rekursive Unterprogramme
- Pointer-Datentyp
- sehr komplex

SIMULA, 1967

- Erweiterung von ALGOL
- Zweck: Computer-Simulationen $\Longrightarrow$ Korutinen
- Klassen

Pascal, 1971

- `case`-Anweisung
- benutzerdefinierte Datentypen (wie in ALGOL 68)
- beliebt in der Lehre bis in den 1990er

C, 1971

- ursprünglich konzipiert für Systemprogrammierung
- Implementierungssprache für UNIX
- liberales Typsystem $\Longrightarrow$ flexibel aber unsicher
- ANSI C (1989), ISO C (1999)

Ada, 1983

- genannt nach Ada Lovelace, die als erste Programmiererin gilt
- ursprünglich für eingebettete und Echtzeit-Systeme
- Kapselung (*encapsulation*) via Programmeinheiten
- Generische Programmeinheiten (*generics*)
- Ausnahmen
- Unterstützung für Nebenläufigkeit
- sehr gross und complex
- Erweiterungen: Ada 95

Smalltalk, 1980

- Programmeinheiten = Objekte
- Bezieung zwischen Sprachkonzepten und Design-Konzepten (statt Hardware)
- “Everything is an object”
- Klasshierarchien aus SIMULA $\Rightarrow$ hat objektorientiertes Programmieren bekannt gemacht
- Programmierumgebung mit Hilfe einer graphischer Schnittstelle

C++

- C + Objekte
- unterstützt dynamisches Binden von Methoden
- multiple Vererbung
- Operatorenüberladung
- Templates für generische Programmierung
- Ausnahmen
- strengeres Typsystem als C

Java, 1994

- ursprünglich für eingebettete Prozesoren in elektronischen Geräten
- basiert auf C++; vereinfacht und sicherer gemacht
- keine Pointer, keine Pointerarithmetik, nur einfache Vererbung, keine Operatorenüberladung
- automatische Speicheranforderung und Speicherfreigabe (*garbage collection*)

C#, 2002

- basiert auf C++ und Java
- ist eine der .NET Sprachen; .NET Sprachen benutzen ein gemeinsames Typsystem
- Pointer, Operatorenüberladung

2.1.2 Logische Sprachen

Prolog, 1972

- Programming logic
- deklarativ statt imperativ: das Ergebnis ist beschrieben (statt der Berechnung, die zum Ergebnis führt)
- Programm $\equiv$ Menge von Formeln; das Ergebnis ist eine Variablen-Substitution die eine Formel erfüllt
- benutzt in Künsliche Intelligenz
- wird von uns näher betrachtet
- Constraint Logic Programming: Erweiterung der Formeln um Constraints aus spezifischen Anwendungsbereichen

2.1.3 Funktionale Sprachen

LISP, Ende 1950er

- LISt Processing
- Künstliche Intelligenz (Linguistik, Mathematik)
- Rechnen mit nicht-numerischen Daten $\Longrightarrow$ verkettete Listen
- Datenstrukturen: Atome (Bezeichner, Zahlen), Listen
- Funktional: Berechnung $\equiv$ nur Funktionsanwendungen, keine Zuweisungen
- Iteration durch rekursive Funktionen
- Nachfolger: Scheme (1975), COMMON LISP (1984)

Scheme, 1975

- statisch gebundene Variablen (*static scoping*)
- Funktionen = Werte erster Klasse (*first order values*)

COMMON LISP, 1984

- Versuch, LISP-Dialekte zu unifizieren
- sowohl statisch als auch dynamisch gebundene Variablen

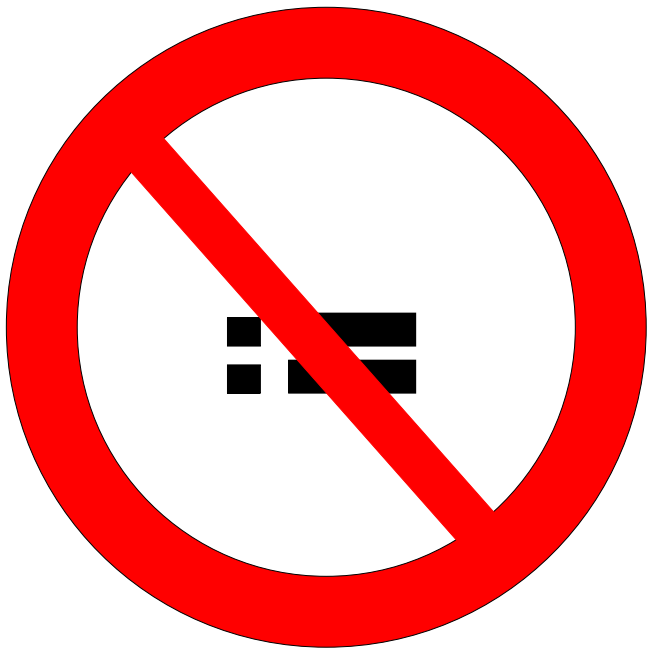
ML, 1980

- Meta Language
- statisch getypt (*statically typed*)
- Typen werden (meist) automatisch abgeleitet $\Longrightarrow$ Typ-Inferenz
- Unterstützung auch für imperatives Programmieren
- Dialekte: SML, MLTon, MoscowML, Caml, OCaml
- wird von uns näher betrachten.

Haskell, 1992

- **rein** funktional
- Benutzt verzögerte Auswertung $\equiv$ Bedarfsauswertung (*lazy evaluation*)

3 Funktionale Programmierung



Funtkionale Programmierung vs. traditionelle (prozedurale) Programmierung

- Traditionelle Programmierung
  - ▷ Berechnungsbeschreibung = Folge von Befehlen

    $\Longrightarrow$ imperativ

    Lat. imperare = befehlen
  - ▷ Berechnungsausführung = Folge von Zustandsänderungen

    Zustand $\equiv$ Inhalt der Speicherzellen

    Zustandsänderung durch Zuweisung:

    $$Speicherzelle := Wert$$

    $\Longrightarrow$ zuweisungsorientiert

- Traditionelle Programmierung
  - ▷ eng verknüpft mit der zugrunde liegenden Rechen-Maschine (von Neumanns Modell):
    - CPU arbeitet Befehl nach Befehl sequentiell ab
    - Speicher dient zur Zustandsprotokollierung

Funtkionale Programmierung vs. traditionalle (prozedurale) Programmierung

- Funktionale Programmierung (FP)
  - ▷ Berechnungsbeschreibung = (mathematische) Funktion

    $\Longrightarrow$ deklarativ

    $f : E \mapsto A, f(x) = x+1$ (Funktionsdefinition/Deklaration)
  - ▷ Berechnungsausführung = Funktionsanwendung

    $f(4)$
  - ▷ Abstrahiert von der zugrunde liegenden Rechen-Maschine

Funtkionale Programmierung vs. traditionalle (prozedurale) Programmierung

- Formale Ausdrucksstärke:
  - ▷ Imperative Programmiersprachen $\mapsto$ Turing-Maschine
  - ▷ FP $\mapsto$ Lambda-Kalkül
  - ▷ Turing-Maschine $\equiv$ Lambda-Kalkül $\equiv$ berechenbare Funktionen
- Praktische Ausdrucksstärke:
  - wollen wir näher betrachten

3.1 Merkmale der FP

3.1.1 Keine Seiteneffekte

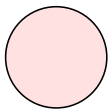
- Der Aufruf einer Funktion mit dem selben Parameter liefert immer das selbe Ergebnis
- Aufruf $\equiv$ Anwendung einer mathematischen Funktion
- 1. Vorteil: Unabhängigkeit von der Auswerttungsreihenfolge (z.B. der Parameter)

Fibonacci-Bäume

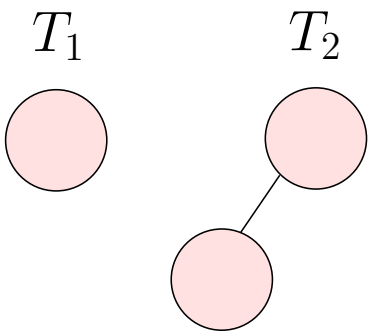
- **Definition**:
  - ▷ Der leere Baum ist ein Fibonacci-Baum der Höhe 0
  - ▷ Ein einzelner Knoten ist ein Fibonacci-Baum der Höhe 1
  - ▷ Sind T_{h-1} und T_{h-2} Fibonacci-Bäume der Höhen $h-1$ und $h-2$, so ist $T_h = k < T_{h-1}, T_{h-2} >$ ein Fibonacci-Baum der Höhe h.
- sind ein Speziallfall von AVL-Bäume
  - ▷ Binärbäume, bei denen an jedem inneren Knoten der Höhenunterschied zwischen dem rechten und linken Teilbaum maximal 1 ist

Fibonacci-Bäume

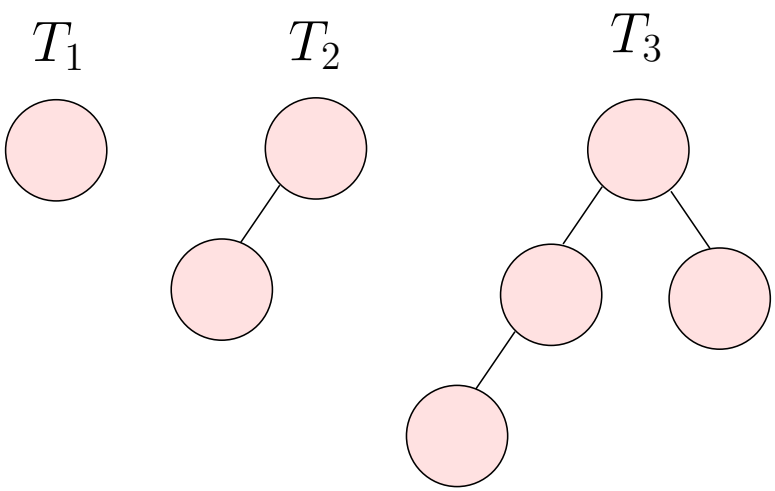
T_1



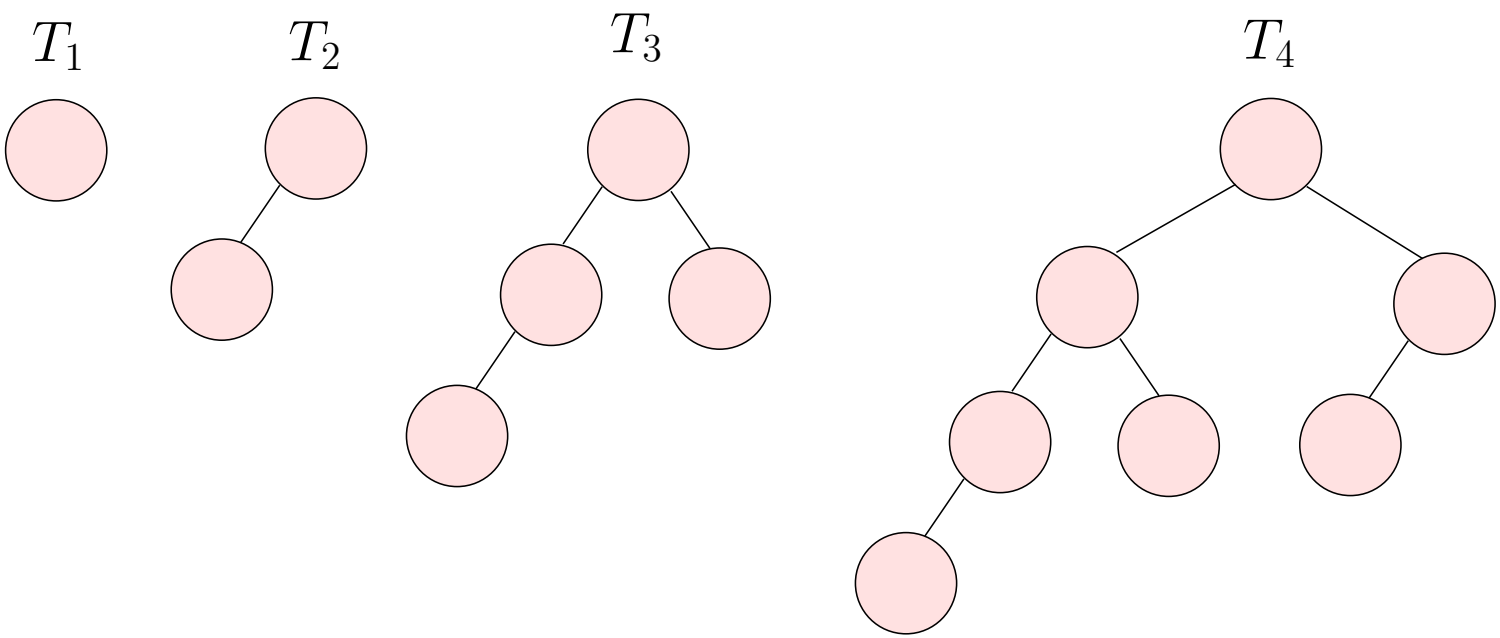
Fibonacci-Bäume



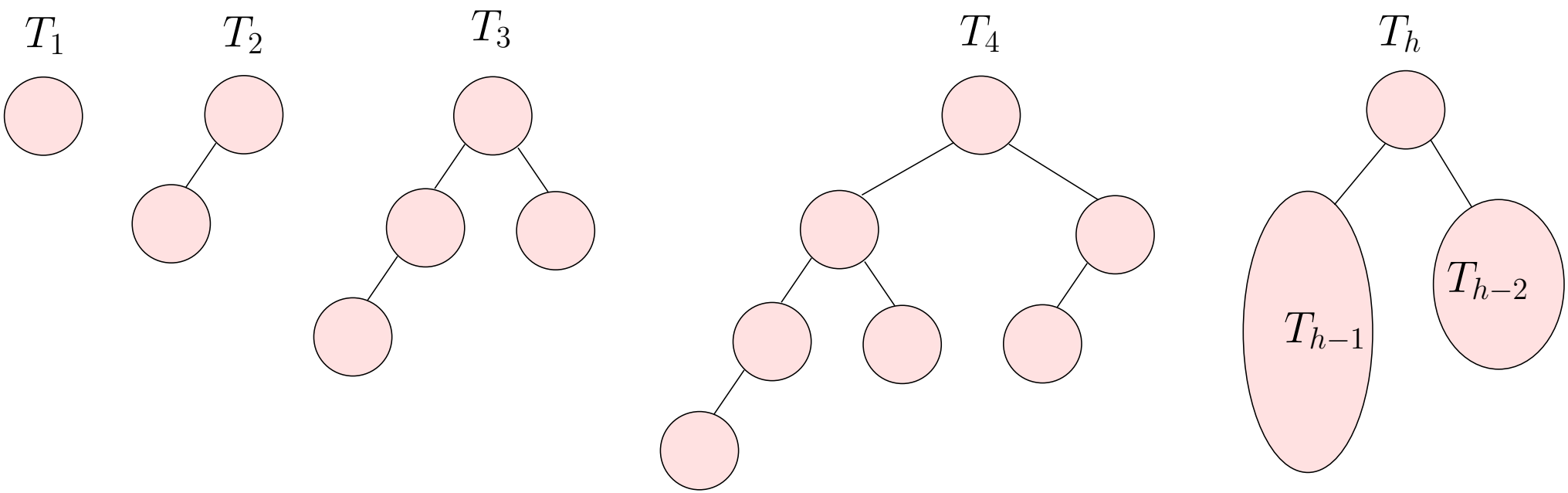
Fibonacci-Bäume



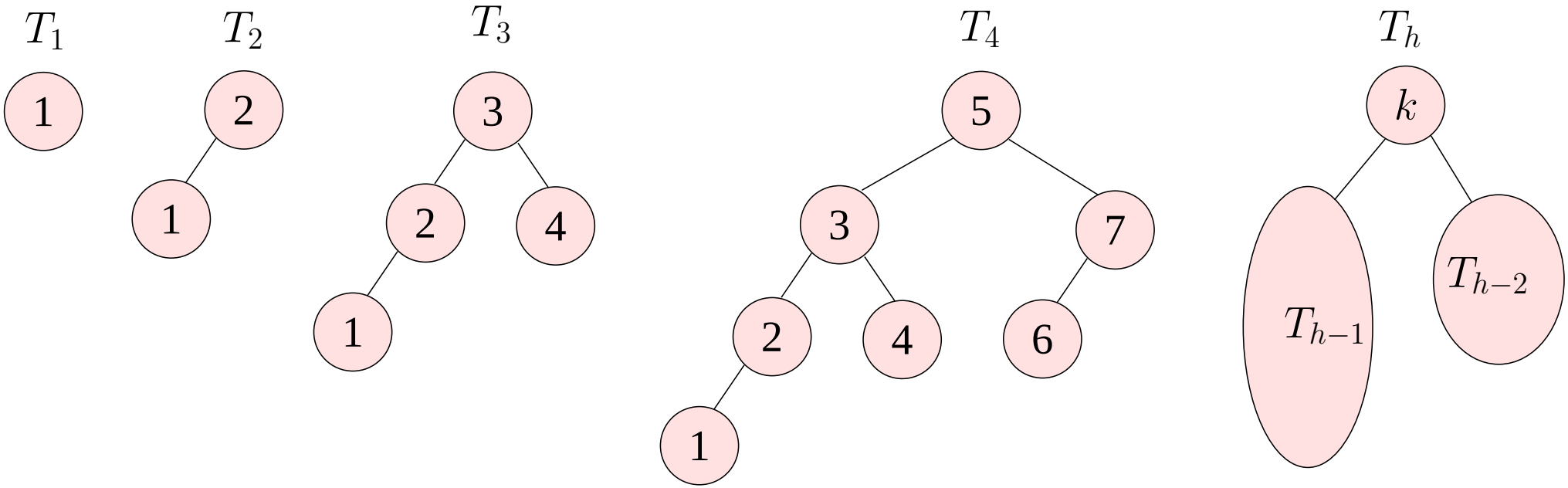
Fibonacci-Bäume



Fibonacci-Bäume



Natürlich annotierte Fibonacci-Bäume



... 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 Fibonacci-Bäume

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

Imperative Lösung (in Java)

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

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

    .......
```

Imperative Lösung (in Java)

```
.........

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);              }
```

Probleme der iterativen Lösung

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

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

- Das Ergebnis der Funktion hängt von der Reihenfolge der Auswertung ab
  - ▷ Java spezifiziert die Auswertung von links nach rechts
  - ▷ C/C++ Compiler können eine beliebige Reihenfolge wählen

    $\Longrightarrow$ das Ergebnis ist nicht deterministich

Probleme der iterativen Lösung

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

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

- `makeFibTreeHelp` ist keine Funktion im mathematischen Sinne, weil sie Seiteneffekte hat
  - ▷ Schwer zu verstehen
  - ▷ Schwer zu beweisen, dass sie einen Fibonacci-Suchbaum konstruiert

Beweisidee (für Java)

```
    ..........

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

- Die Funktion simuliert einen *in-order* (links, Wurzel, rechts) Durchlauf
- plausibel aber kein gültiger formaler Beweis

Funktionale Lösung (immer noch in Java)

Hilfsfunktion `maxKey` berechnet den maximalen Schlüssel in einem Baum:

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

Funktionale Lösung (immer noch 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.*

Korrektheitsbeweis

3.1 ***Lemma.*** *Sei minKey eine mathematische Funktion, die den kleinsten Schlüssel in einem Baum berechnet. Dann gilt: minKey(t(h,k))=k für alle* $h > 0$.

Beweis: Induktion über h.

Für $h = 1$, hat $t(1, k)$ nur einen Knoten mit Label $k \Rightarrow$ *minKey(t(1,k))=k.*

Angenommen *minKey(t(i,k))=k* für alle i mit $1 \leq i \leq h$ und alle k.
Wir zeigen, dass *minKey(t(h+1,k))=k.*

$$
\begin{aligned}
t(h+1,k) = FibTree(\ & t(h,k),\\
& maxKey(t(h,k))+1,\\
& t(h-1, maxKey(t(h,k))+2));
\end{aligned}
$$

$\Longrightarrow$

$$
\begin{aligned}
minKey(t(h+1,k)) &= MIN(minKey(t(h,k)),\\
&\qquad maxKey(t(h,k))+1,\\
&\qquad minKey(t(h-1, maxKey(t(h,k))+2)))\\
&= MIN(minKey(t(h,k)),\\
&\qquad minKey(t(h-1, maxKey(t(h,k))+2)))\\
&= MIN(minKey(t(h,k)), maxKey(t(h,k))+2)\\
&= minKey(t(h,k))) = k
\end{aligned}
$$

3.2 ***Lemma.*** *Sei t(h,k)=FibTree(l,k',r). Dann gilt:*

1. $maxKey(l) < k'$ *für* $h = 2$
2. $maxKey(l) < k' < minKey(r)$ *für* $h > 2$

Beweis:

1. $t(2,k) = FibTree(FibTree(null, k, null), k+1, null) \Longrightarrow maxKey(l) = k$ und $k' = k+1$.
2. $t(h,k) =$
   $FibTree(t(h-1,k), maxKey(t(h-1,k))+1, t(h-2, maxKey(t(h-1,k))+2))$
   $\Longrightarrow$

$$\begin{aligned} l &= t(h-1,k) \\ k' &= maxKey(t(h-1,k))+1 \\ r &= t(h-2, maxKey(t(h-1,k))+2) \end{aligned}$$

Bleibt zu zeigen, dass:
$maxKey(t(h-1,k)) < maxKey(t(h-1,k))+1 <$
$minKey(t(h-2, maxKey(t(h-1,k))+2)) \Longleftrightarrow$
$maxKey(t(h-1,k)) < maxKey(t(h-1,k))+1 < maxKey(t(h-1,k))+2$ $\square$

3.3 ***Korollar.*** *t(h,k) ist ein Suchbaum.*

Beweis: Alle Teilbüme mit Höhe $i < h$ aus $t(h, k)$ sind entweder `null`, haben einen Knoten oder erfüllen Lemma 3.2.

- Bleibt noch zu zeigen, dass die Funktion $t(h,k)$ immer terminiert:

```
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));
}
```

  Folgt direkt per Induktion über h.

- Per Induktion kann man ferner zeigen, dass $t(i,k)$ alle Schlüssel zwischen *minKey(t(i,k))* und *maxKey(t(i,k))* enthält.

  $\Longrightarrow$ unsere Funktion konstruiert einen natürlich annotierten Fibbonaci-Baum

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. Idee:

- *maxKey* ist unnötig
- Funktion t soll ein Paar der Form (Baum,maximaler Schlüssel im Baum) zurückliefern

  Hilfklasse `Pair`

```
class Pair{
    public FibTree tree;
    public int max;

    public Pair(FibTree t,int m){tree=t;max=m;}
}
```

Dritte Variante

```
static Pair t1(int h, int k){
  if (h == 0) return new Pair(null,k);
  if (h == 1) return new Pair(new FibTree(null,k,null),k);
  Pair leftRes = t1(h-1,k);
  Pair rightRes = t1(h-2,leftRes.max+2);
  return new Pair(
    new FibTree(leftRes.tree,leftRes.max+1,rightRes.tree),
    rightRes.tree==null ? leftRes.max+1 : rightRes.max);
}
```

3.1.2 Referenzielle Transparenz

- Abwesenheit von Seiteneffekten sorgt für referenzielle Transparenz (*referential transparency*)
- Referenzielle Transparenz: der Wert eines Programmausdruckes hängt nur von den Werten der Teilausdrücken und nicht von dem Kontext der Auswertung; 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 $\Longrightarrow$ referenzielle Transparenz $\equiv$ keine Seiteneffekte

Referentielle Transparenz/Keine RT

- Funktionale Programmierung (SML):

  `E + E` $\equiv$ `let x = E in x + x`

- Imperative Programmierung (Java/C++):

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

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.

Folgen der referentiellen Transparenz

- Korrektheit: einfacher formale Eigenschaften mit klassischen mathematischen Verfahren zu 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*)
- Wartbarkeit:
  - ▷ Wenn eine Funktion einmal richtig funkioniert hat, dann funktioniert sie immer richtig, unabhängig vom Auswertungskontext
  - ▷ bessere Lesbarkeit

3.1.3 Funktionen sind Werte erster Klasse

- Funktionen $\equiv$*first-class objects*

  $\Longrightarrow$ Können als Parameter übergeben werden

  $\Longrightarrow$ Können als Rückggabewerte von Funktionen zurückgeliefert werden

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

  $\longrightarrow$ Eine Funktion, die Funktionen als Argumente bekommen oder eine Funktion als Ergebnis liefert heißt Funktion höherer Ordnung.

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));
}
```

Probleme:

- $comp(f, g, x)$ liefert $f(g(x))$; erwünscht wäre $f \circ g$ (als Funktionswert)
- $comp$ ist nicht generisch

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`