# Programmiersprachen

Alexandru Berlea

Institut 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)
  - Programm = 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)

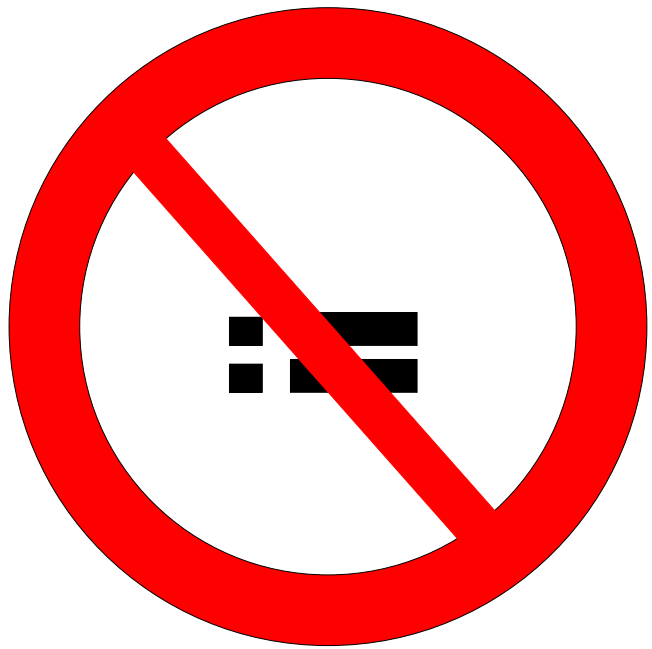
# Programmierung: kurze Chronologie (Forts.)

**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 Programmier

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...

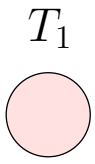
- ... der Aufruf der Funktion mit dem selben Parameter liefert immer das selbe Ergebnis
- d.h. Aufruf $\equiv$ Anwendung einer mathematischen Funktion
- 1. Vorteil:
  **Unabhängigkeit von der Auswertungsreihenfolge**
  (z.B. der Parameter)

# Ein Beispiel: Fibonaccibäume

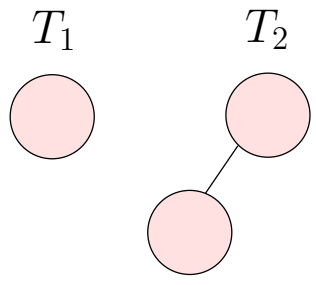
- **Definition**:
  - Der leere Baum ist ein Fibonacci-Baum der Höhe 0
  - Ein einzelner Knoten ist ein Fibonacci-Baum der Höhe 1
  - Sind $T_{h-1}$ und $T_{h-2}$ Fibonacci-Bäume der Höhen $h-1$ und $h-2$, so ist $T_h = k < T_{h-1}, T_{h-2} >$ ein Fibonacci-Baum der Höhe $h$.

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

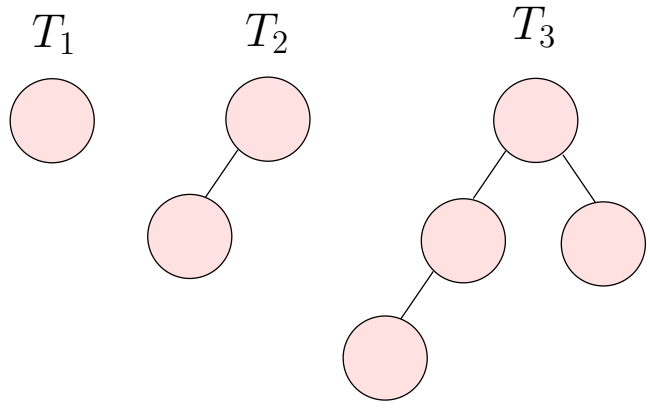
# Ein Beispiel: Fibonaccibäume



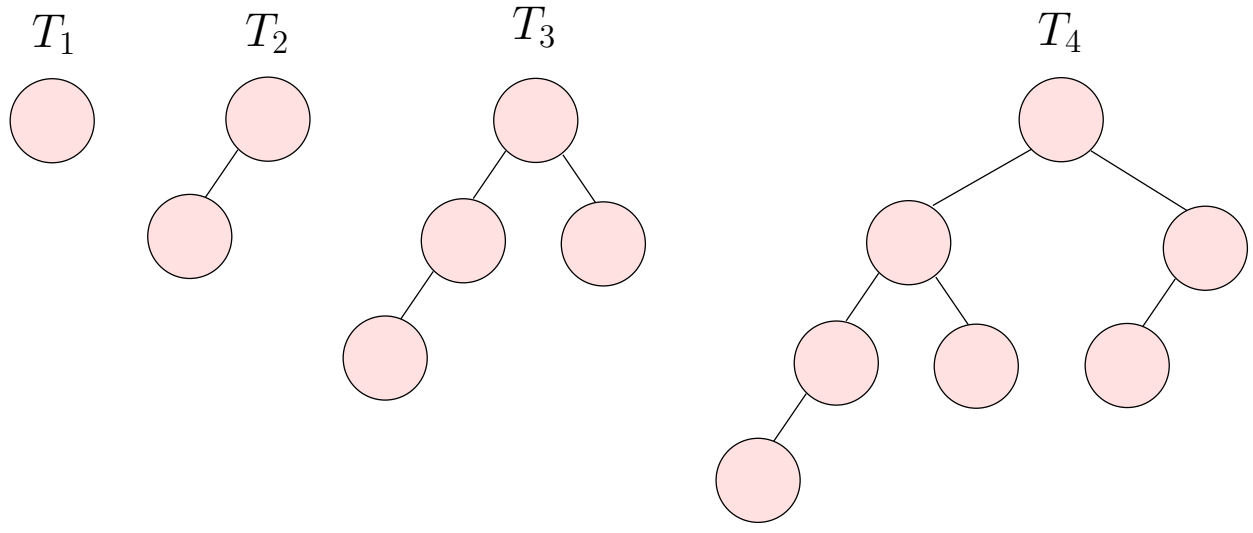
# Ein Beispiel: Fibonaccibäume



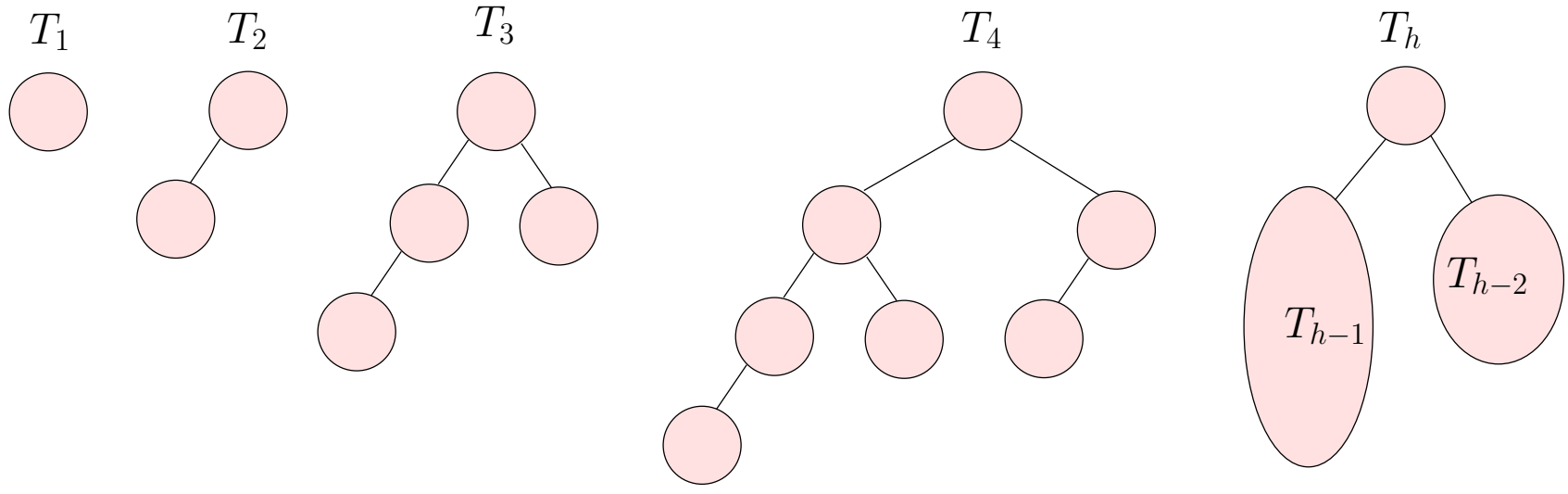
# Ein Beispiel: Fibonaccibäume



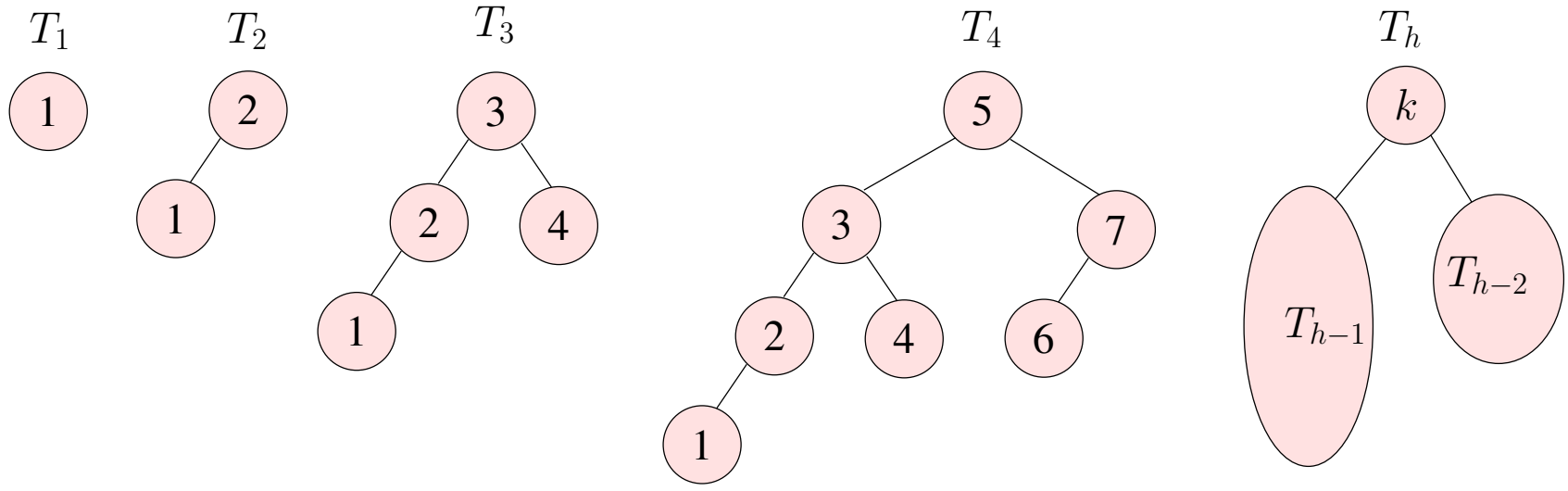
# Ein Beispiel: Fibonaccibäume



# 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 deterministisch

# 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 deterministisch
- `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 $\Longrightarrow$ referentielle 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 funktioniert hat, dann funktioniert sie immer richtig, unabhängig vom Auswertungskontext
  - bessere Lesbarkeit

# Grenzen der puren funktionalen Programmierung

- Manche Berechnungen sind **inhärent nicht referentiell 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 referentiell 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
```

# Überblick

## 3. Datentypen

Einleitung

Vordefinierte Typen (Typ-Konstanten)

Definition neuer Typen (Typ-Ausdrücke)

Pattern-Matching

Rekursive Typen

Polymorphe Typen

# Typen

- **Typen** = Mengen von Programm-Entitäten mit gleichartigem Verhalten

- Verwendung:
  - **Organisation**/Benennung von Konzepten, **Dokumentation**. Z.B.:
    - Basis-Typen: `int`,`float`
    - Komplexe Typen: `int channel` (CML), `int cont` (SML), Klassen (Java)

  - **Fehler erkennen und vermeiden** $\Longrightarrow$ **Type-safety**

  - **Optimierungen**: z.B. Komponente in einem Record/Objekt nachschlagen
    - Sequentielles durchsuchen, wenn der Typ unbekannt ist
    - Zugang via bekanntes Offset, sonst

# Type-Safety

- **Type-Safety** = Programm-Entitäten gemäß ihrer Eigenschaften manipulieren $\Longrightarrow$ keine unbeabsichtigte Semantik

- erreicht durch **Type-Checking** = Überprüfung der Funktionsargumente auf Konformheit mit dem Typ des Definitionsbereichs
  - Der Typ jeder Entität muss bekannt sein, insbesondere für Definitions- und Bildbereich für Funktionen/Operatoren
  - keine Konformheit $\Longrightarrow$ Typ-Fehler, z.B. `1 + true`

# Type-Safety

- Bsp.:
  - Type-safe: ML, Lisp, Java
  - Nicht type-safe: C, C++ $\Longrightarrow$ evt. unbeabsichtigte Semantik:
    - Implicit type-casts: ein Integer als Pointer benutzen
    - Pointer-Arithmetik: Wenn `p` Typ `T` hat, dann hat `p+1000` auch Typ `T`, obwohl die entsprechende Speicherzelle ein anderer Typ haben könnte.
    - Explizite Speicherfreigabe $\Longrightarrow$ evt. dangling pointers

# Type-Checking

Type-Safety wird erreicht via **Type-Checking**

- **Laufzeit-Typechecking** (dynamisch): Compiler generiert Code zur Überprüfung, dass Operanden den richtigen Typ haben

  $\Longrightarrow$ Verlangsamung der Ausführung

- **Compilezeit-Typechecking** (static): die (meisten) Typ-Überprüfungen können bei modernen Programmiersprachen bei Compile-Zeit stattfinden.
  - **Weniger Programmierfehler**
  - **Laufzeit Robustheit**
  - **Optimierter Code**

# Type-Checking

Type-Safety wird erreicht via **Type-Checking**

- **Laufzeit-Typechecking** (dynamisch): Compiler generiert Code zur Überprüfung, dass Operanden den richtigen Typ haben
  $\Longrightarrow$ Verlangsamung der Ausführung

- **Compilezeit-Typechecking** (static): die (meisten) Typ-Überprüfungen können bei modernen Programmiersprachen bei Compile-Zeit stattfinden.
  - **Weniger Programmierfehler**
  - **Laufzeit Robustheit**
  - **Optimierter Code**
  - aber notwendigerweise konservativ: ob ein Programm einen Typ-Fehler erzeugen könnte ist i.A. nicht entscheidbar:

```
if e then eWithTypeError else eWithTypeError
```

(stellt in der Praxis aber kein Problem:-)

# Type-Checking

**Laufzeit- und Compilezeit-Typechecking:**
Die meisten Programmiersprachen benutzen eine Kombination der beiden, z.B. Java:

- statisch: Unterscheiden zwischen Ganzzahlen und Funktionen
- dynamisch: Array-Bounds check

# Vordefinierte Typen in SML

| Typ | Bsp.-Werte | Bsp.-Operatoren |
|---|---|---|
| int | `0 3 ~7` | `+ - * div mod` : int × int ↦ int<br>`~` : int ↦ int |
| real | `3.0 7.0` | `+ - * /` : real × real ↦ real<br>`~` : real ↦ real |
| bool | `true false` | `not` : bool ↦ bool<br>`orelse andalso` : bool × bool ↦ bool |
| string | `"hallo"` | `^` : string × string ↦ string |
| char | `#"a" #"b"` | |
| unit | `()` | |

# Definition neuer Typen

- **Typ-Operatoren** konstruieren neue Typen aus bestehenden T.

$$op : MT^n \mapsto MT \text{ mit } n \geq 0$$

  mit $MT$ die Menge der Typen in der Sprache.

- Die vordefinierten Basis-Typen (`real`,`int`,`bool`,`unit`) sind **nullstellige Typ-Operatoren/Typ-Konstanten**.

- Durch Anwendung der Typ-Operatoren enstehen **Typ-Ausdrücken**.

# Produkt-Typen

Der Typ-Operator $* : MT^n \mapsto MT$ **mit** $n \geq 2$:

- steht zwischen Operanden (*infix operator*):
- ist definiert als:

$$\alpha_1 * \alpha_2 * \cdots * \alpha_n = \{(v_1, v_2, \ldots, v_n) \mid v_k \in \alpha_k \textbf{ für alle } k\}$$

Bsp.: $int * int = \{(x, y) \mid x, y \in int\}$

# Produkt-Werte

Werte vom Produkt-Typ kann man mit Tupel-Wertekonstruktoren konstruieren

```
- (1,2);
val it = (1,2) : int * int

- (1,2,true);
val it = (1,2,true) : int * int * bool

- (1,(2,true));
val it = (1,(2,true)) : int * (int * bool)

- ((1,2),true);
val it = ((1,2),true) : (int * int) * bool
```

Die Symbolenkombination (,) kann man als einen verteilter Konstruktor auffassen.

# Produkt-Werte

```
- (1,2,true) = (1,(2,true));
stdIn:42.1-42.26 Error: operator and operand don't agree [tycon mismatch]
 operator domain: (int * int * bool) * (int * int * bool)
 operand: (int * int * bool) * (int * (int * bool))
 in expression:
 (1,2,true) = (1,(2,true))

- (1,2,true) = (1,2,true);
val it = true : bool
```

# Record-Typen (Verbunde)

Ein **Record-Typ** besteht aus Tupeln mit benannten Komponenten:

```
- val p1 ={vorName="John",name="Smith",alter="23"};
val p1 = {alter="23",name="Smith",vorName="John"}
: {alter:string, name:string, vorName:string}

- val p2 = {vorName="Jan",name="Smith"};
val p2 = {name="Smith",vorName="Jan"}
: {name:string, vorName:string}
```

- Zwei Record-Typen sind gleich wenn sie gleich viele Komponente haben, jeweils mit dem selben Namen und dem selben Typ:
- Zwei Record-Werte ($\equiv$ **Records**) sind gleich, wenn sie vom selben Typ sind, und die jeweiligen Komponenten gleich sind
- Die Symbolenkombination $\{,=\}$ kann man als einen verteilter Operator bzw. Wertekonstruktor auffassen.

# Records

- Reihenfolge ist irrelevant

```
- {name="Schwarz",vorName="Peter",alter="25"} =
  {name="Schwarz",alter="25",vorName="Peter"};
val it = true : bool
```

- Tupel sind eine Spezialschreibweise für Records

```
- {1=true,2="Martin"};
val it = (true,"Martin") : bool * string
- {1=3,2=5};
val it = (3,5) : int * int
```

# Summentypen

- **Summentyp** = eine Sammlung von Werten anderer Typen
  z.B. sollte der Typ **Publication** u.a. folgendes enthalten:
  - `{title="Generics",conf="Fun in the afternoon", auth="J. Smith"}:{auth:string, conf:string, title:string}`
  - `{title="ML in 7 days",publisher="Hupfer",auth="J. Valjean"}:{auth:string, publisher:string, title:string}`
  - `{title="P=NP",journal="JC",auth="J. Walker",edit="V. Smirnoff"}:{auth:string, edit:string, journal:string, title:string}`

  I.A. können diese Typen beliebig unterschiedlich sein...

- Lösung: **Summentyp** $S \in MT$ wird aus einer Menge von Typen $MT_1 \subset MT$ mit Hilfe von symbolischen Funktionen (**Konstruktoren**) $c$ konstruiert:

$$S = \{c(v) \mid v \in \alpha, \alpha \in MT_1, c : \alpha \mapsto S\} \cup \{c \mid c : \bullet \mapsto S\}$$

# Summentypen

- Ein Summentyp wird definiert mit **datatype** durch Angabe seiner **Konstruktoren**:

  `datatype` = $Konstruktor_1$ | $Konstruktor_1$ | ... | $Konstruktor_n$

  z.B.:

```
datatype Publication =
  Article of {title:string,conf:string,auth:string} |
  Book of {title:string,publisher:string,auth:string} |
  Comm of {title:string,journal:string,auth:string,
           edit:string} |
  Bible | Koran
```

- Konstruktoren können sein:
  - **einstellig**: Article, Book, Comm
  - **nullstellig**: Bible, Koran

# Summentypen

- Noch ein Bsp.:

```
datatype Farbe = Rot | Blau | RGB of (int*int*int)
```

$$\texttt{Farbe} = \{\texttt{Rot}\} \cup \{\texttt{Blau}\} \cup \{\texttt{RGB (x,y,z)} \mid x, y, z \in \texttt{int}\}$$

- Der Compiler erkennt den Typ eines Wertes anhand des Konstruktors:

```
- Rot;
val it = Rot : Farbe
- RGB (80,200,130);
val it = RGB (80,200,130) : Farbe
```

# Aufzählungstypen (enumeration types)

- **Aufzählungstyp** = Ein Summentypen mit nur nullstelligen Konstruktoren
- für endliche Typen, deren Wert **aufgezäht** werden können:

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Zehn
```

# Aufzählungstypen

Der Compiler erkennt den Typ eines Wertes anhand des Konstruktors:

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Neun | Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Neun | Zehn
- Kreuz;
val Kreuz : Farbe
- val pik_bube = (Pik,Bube);
val pik_bube = (Pik,Bube) : Farbe * Wert
```

# Aufzählungstypen vs. ad-hoc Kodierungen

**Mögliche Kodierung:** Benutze Paare von Strings und Zahlen, z.B.:

| | | |
|---|---|---|
| `("Karo","10")` | $\equiv$ | Karo Zehn |
| `("Kreuz","12")` | $\equiv$ | Kreuz Bube |
| `("Pik","1")` | $\equiv$ | Pik As |

**Nachteile:**

- Beim Test auf eine Farbe muß immer ein String-Vergleich stattfinden $\longrightarrow$ ineffizient
- Darstellung des Buben als 12 ist nicht intuitiv $\longrightarrow$ unleserliches Programm
- Welche Karte repräsentiert das Paar `("Kaor","1")`? (Tippfehler werden vom Compiler nicht bemerkt)

# Vorteile Aufzählungstypen vs. ad-hoc Kodierungen

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Neun | Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Neun | Zehn
```

## Vorteile:

- Darstellung ist intuitiv.
- Tippfehler werden erkannt:

```
- (Kaor, As);
stdIn:29.2-29.6 Error: unbound variable or constructor: Kaor
```

- Interne Repräsentation ist effizient.

# Aufzählungstypen vs. Basis-Typen

Manche Basis-Typen können als spezielle vordefinierte Aufzählungstypen aufgefasst werden:

- `datatype bool = true | false`
- `datatype char = #"a" | #"b" | #"c" | ...`
- `datatype int = ... | ~2 | ~1 | 0 | 1 | 2 | ...`

# Pattern-Matching

Werte eines selben Typs können unterschiedlich behandelt werden, je nachdem mit welchem Konstruktor sie erzeugt wurden $\Longrightarrow$

**Fall-Unterscheidung (*pattern matching*)**

Die Fall-Unterscheidung erfolgt mit Hilfe des **case**-Ausdruckes:

```
case Ausdruck of
    Muster_1 => Ausdruck_1
  | Muster_2 => Ausdruck_2
       ...   =>
  | Muster_n => Ausdruck_n
```

# Pattern-Matching: Beispiel

```
datatype Farbe = Rot | Blau | RGB of (int*int*int)
```

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "scarlet"

val description = "scarlet" : string
```

- Im Unterschied zum gleichnamigen imperativen Konstrukt ist case einen Ausdruck (d.h. er hat einen Wert)
- In FP gibt es nicht den Unterschied zwischen Anweisungen und Ausdrücken $\Longrightarrow$ Alles ist ein Ausdruck
- Die rechten Seiten müssen alle den selben Typ haben; das ist der Typ des Gesamtausdruckes.

# Der If-Ausdruck

- Oft findet die Fallunterscheidung über einen Wert vom Typ `bool`:

```
fun max (x,y) = case x>= y of true => x
                            | false => y;
val max = fn : int * int -> int

max (3,2);
val it = 3 : int
```

- Dafür gibt es eine alternative, kürzere Syntax:

```
val fun max (x,y) = if x>= y then x
                    else y;
val max = fn : int * int -> int
```

# Pattern-Matching mit Variablen-Bindung

- Patterns können benutzt werden, um auf Bestandteile eines *konstruierten* Wertes zuzugreifen $\Longrightarrow$ **Dekomposition**
- Dafür müssen Patterns Variablen enthalten
- Beim Pattern-Matching über einen Wert werden die Variablen automatisch zu den entsprechenden Teilen des Wertes gebunden
- Die im Muster $Muster_i$ gebundenen Variablen sind im Ausdruck $Ausdruck_i$ sichtbar

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of Rot => "pure red"
                            | Blau => "pure blue"
                            | RGB(x,y,z) =>
                        if (x=y) andalso (y=z) then "gray"
                        else "something else";
val description = "something else" : string
```

# Pattern-Matching mit Variablen-Bindung

Wenn man eine Variablen-Bindung nicht braucht kann man _ (Unterstrich) benutzten

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(x,y,_) =>
                      if (x=y) then "kind of yellow"
                      else "something else";
val description = "something else" : string
```

# Pattern-Matching mit Variablen-Bindung

- ist ein wichtiges Feature, die die Dekomposition eines Wertes in seine Teile unterstützt
- Zusätzlich überprüft der Compiler automatisch, ob die Patterns redundant oder unvollständig sind...

# Unvollständige Patterns

```
- val description = case f of Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "kind of green" ;
stdIn:78.13-108.57 Warning: match nonexhaustive
 Rot => ...
 Blau=> ...
 RGB (80,200,130) => ...
val description = "kind of green" : string
```

Alle Fälle sollten behandelt werden, evt. ähnlich wie unten:

```
- val description = case f of Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "kind of green"
                    | _ => "don't know";
val description = "kind of green" : string
```

# Redundante Patterns

```
-   val description = case f of
                        Rot => "pure red"
                      | Blau => "pure blue"
                      | RGB(x,y,z) => "RGB colour"
                      | RGB(80,200,130) => "kind of green";
stdIn:125.8-139.57 Error: match redundant
 Rot => ...
 Blau => ...
 RGB (x,y,z) => ...
 --> RGB (80,200,130) => ...
```

# Redundante Patterns

Achtung: Die Reihenfolge ist wichtig

```
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "kind of green";
                    | RGB(x,y,z) => "RGB colour"
val description = "kind of green" : string
```

# Pattern-Matching: Einschränkungen

- Patterns dürfen nur Konstruktoren enthalten:

```
case "abc" of prefix^suffix => prefix;
stdIn:66.1-66.38 Error: non-constructor applied to argument in pattern:^
stdIn:66.32-66.38 Error: unbound variable or constructor: prefix
```

$\Longrightarrow$ Eindeutigkeit:

- Höchstens eine Variable mit einem gegeben Namen in einem Pattern (**Linearität**):

```
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(x,x,_) => "yellow"
stdIn: Error: duplicate variable in pattern(s): x
```

# Rekursive Typen

Um beliebig große Datenstrukturen repräsentieren zu können braucht man **rekursive Typen** (z.B. Listen/Bäume).

- Die Definition eines Typen ist rekursiv, wenn Typ-Konstruktoren Elemente des zu definierenden Typ erhalten.

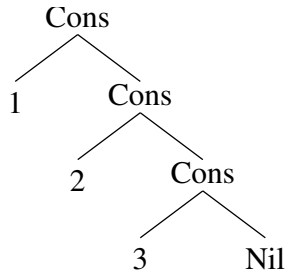
```
datatype IntList = Nil | Cons of (int*IntList);
```

- Damit Werte konstruierbar sind, muss mindestens ein **nullstelliger Konstruktor** angegeben werden.
  Bsp.: Liste mit Elementen 1, 2, 3

```
Cons(1,
=       Cons(2,
=            Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons (3,nil))) : IntList
```

# Aufbau einer Liste

```
Cons(1,Cons(2,Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons (3,nil))) : IntList
```



# Verarbeitung rekursiver Datentypen...

... erfolgt mit Hilfe von Pattern-Matching und rekursive Funktionen
Die Länge einer Liste:

```
fun length l = case l of
                    Nil => 0
                  | Cons(first,rest) => 1 + length rest;
val length = fn : IntList -> int

length (Cons(1,Cons(2,Cons(3,Nil))));
val it = 3 : int
```

# Polymorphe Typen

Listentypen unterscheiden sich nur in dem Typ ihrer Elemente:

```
datatype IntList = Nil | Cons of (int * IntList);
datatype RealList = NilR | ConsR of (real * RealList);
```

$\Longrightarrow$ manche Funktionen auf Listen sehen im Prinzip gleich aus:

```
fun lengthR l = case l of
                  NilR => 0
                | ConsR(first,rest) => 1 + length rest;
```

- Um Code-Duplizierung zu vermeiden, erlaube **polymorphe Funktionen**: statt nur Argumente eines bestimmten Typs, akzeptiere Argumente verschiedener Typen.

- Wenn der Typ erlaubter Argumente eine Instanz eines Typ-Ausdruckes mit Typ-Variablen sein muss$\Longrightarrow$ **parametrischer Polymorphismus**.

# Parametrischer Polymorphismus

- Ein **parametrisierter Typ** definieren:

```
datatype 'a List = Nil | Cons of ('a * 'a List);
datatype 'a List = Cons of 'a * 'a List | Nil
```

  - 'a ist ein Bezeichner für einen bestimmten beliebigen Typ (Typ-Variable), der in Typ-Ausdrücken in Konstruktoren benutzt werden darf.

- Der Compiler erkennt den Parameter-Typ automatisch:

```
- Cons(1,Cons(2,Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons 3)) : int List
- Cons(1.0,Cons(2.0,Cons(3.0,Nil)));
val it = Cons (1.0,Cons (2.0,Cons 3.0)) : real List
- Cons(true,Cons(true,Cons(false,Nil)));
val it = Cons (true,Cons (true,Cons false)) : bool List
```

# Polymorphe Funktionen

```
datatype 'a List = Nil | Cons of ('a * 'a List)

fun length l =
    case l of
        Nil => 0
      | Cons(first,rest) => 1 + length rest;
val length = fn : 'a List -> int
```

Der Compiler leitet den allgemeinst möglichen Typ ab:
Hier ist 'a als *ein beliebiger Typ* zu lesen $\Longrightarrow$ `length` ist **polymorph**:

```
- length (Cons(1,Cons(2,Cons(3,Nil))));
val it = 3 : int
- length (Cons(1.0,Cons(2.0,Cons(3.0,Nil))));
val it = 3 : int
- length (Cons(true,Cons(true,Cons(false,Nil))));
val it = 3 : int
```

# Der polymorphe Typ `list`

Ein polymorpher Typ `'a list` ist vordefiniert.
Konstruktoren:

- nullstellig: `nil` (entspricht unserem `Nil`)
- einstellig: `::` (entspricht unseren `Cons`)
  - **infixiert**: `kopf::rest`
  - **rechtsassoziativ**: `1::(2::(3::nil))` $\equiv$ `1::2::3::nil`
- Alternative Klammernotation:
  - `[]` $\equiv$ `nil`
  - `[1,2,3]` $\equiv$ `1::(2::(3::nil))` $\equiv$ `1::[2,3]`

# Der polymorphe Typ `list`

```
- fun length l = case l of
                   nil => 0
                 | first::rest => 1 + length rest;
val length = fn : 'a list -> int
- length [1,2,3];
val it = 3 : int
- length [true,true,false];
val it = 3 : int
```

- Alle Elemente einer Liste müssen vom selben Typ sein: [1,[1]] ist keine Liste!
- Liste von Listen von `ints`: `[[1,2,3],[4,5],[6],[7,8,9]]`

# Generische Programmierung

- Polymorphismus unterstützt einen **generischen Programmierstil:** möglichst allgemeine Programm-Spezifikationen, die von irrelevanten Merkmale der verarbeiteten Daten abstrahieren $\Longrightarrow$ bessere Wartbarkeit

- Verschiedenen Prägungen wie z.B.:
  - SML: **Polymorphe Typen**, **Funktoren**
  - Java, C#: **Generics**
  - C++: **Templates**
  - Haskell: **Typklassen**

# Polymorphismus ist polymorph;-)

Formen von Polymorphismus:

- **Parametrischer Polymorphismus:**
  mit Typ-Variablen parametrisierte Typen (wie oben)

- **Subtyp-Polymorphismus:**
  - $t_1$ ist ein **Subtyp** von $t_2$, wenn er spezifischer ist (z.B. $t_1$ ist eine Unterklassse von $t_2$)
  - Eine Funktion, die Argumente von einem Typ annimmt, nimmt auch Argumente von jedem Subtyp an.

- **Ad-hoc-Polymorphismus:** Funktionen mit (beliebig) verschiedenen Typen haben den selben Namen (**Überladen/overloading**)
  - z.B. arithmetische Funktionen
  - nur endlich viele Typen können angenommen werden

# Überblick

## 4. Typ-Inferenz

Typ-Inferenz vs. Typ-Checking
Inferenz-Regeln
Lösung
Unifikation
Polymorphe Funktionen
Typ-Annotationen
Schlussbemerkungen

# Typ-Inferenz

Zum Type-Checking muss man den Typ für jeden Ausdruck im Programm kennen. Ansätze:

- **Klassisches Type-Checking:**
  - **Typ-Deklarationen** für Variablen/Funktionen **erforderlich**.
  - Typherleitung aus den Typen der Teilausdrücke (bottom-up), z.B.

$$e_1 : int \wedge e_2 : int \Rightarrow e_1 + e_2 : int$$

# Typ-Inferenz

Zum Type-Checking muss man den Typ für jeden Ausdruck im Programm kennen. Ansätze:

- **Klassisches Type-Checking:**
  - **Typ-Deklarationen** für Variablen/Funktionen **erforderlich**.
  - Typherleitung aus den Typen der Teilausdrücke (bottom-up), z.B.

$$e_1 : int \wedge e_2 : int \Rightarrow e_1 + e_2 : int$$

- **Typ-Inferenz:**
  - **Typ-Deklarationen nicht nötig**
  - Typherleitung (insb. für Variablen/Funktionen) aus dem Kontext
    - der Typ eines Ausdrucks wird sowohl durch den Typ der Bestandteile (bottom-up) als auch durch den Typ der umgebenden Ausdrücke (top-down) eingeschränkt, z.B.

$$e_1 : int \wedge e_2 : int \Leftrightarrow e_1 + e_2 : int$$

# Terminologie

- Sowohl beim klassischen Type-Checking als auch bei der Typ-Inferenz werden Typen hergeleitet/**inferiert**:

- Unterschied:
  - Klassisches Type-Checking:
    - nur Typen der Zwischenergebnisse werden **inferiert**

```
1 + o.a().getInt();
```

      Die Typen von `o.a()` und `o.a().getInt()` werden inferiert.
    - Variablen- u. Fkt.-Typen müssen vorgegeben werden

```
int fact(int n) { if (n<= 0) return 1
                  else return n * fact(n-1);}
```

  - Typ-Inferenz: (möglichst) alle Typen werden **inferiert**

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

# SML Inferenz-Regeln

- Typen werden mit Hilfe von **Inferenz-Regeln** berechnet.

  Angabe der Beziehung zwischen dem Typ eines Ausdruckes und den Typen seiner Teilausdrücke.

- **Bsp.:** Wenn $x$ und $y$ vom Typ `int` sind, dann ist $x + y$ auch vom Typ `int` (und umgekehrt). Schreibweise:

$$\frac{x : \text{int} \qquad y : \text{int}}{x{+}y : \text{int}}$$

# Inferenz-Regeln

**Inferenz-Regel:** zeigen, welche Typen *gleich* sein (*unifizieren*) müssen.

**Case:**
$$\frac{e : t_1 \quad p_1 : t_1, \ \ldots, \ p_n : t_1 \qquad e_1 : t_2, \ \ldots, \ e_n : t_2}{\texttt{case } e \texttt{ of } p_1 \texttt{ => } e_1 \texttt{ | } \ldots \texttt{ | } p_n \texttt{ => } e_n \texttt{ : } t_2}$$

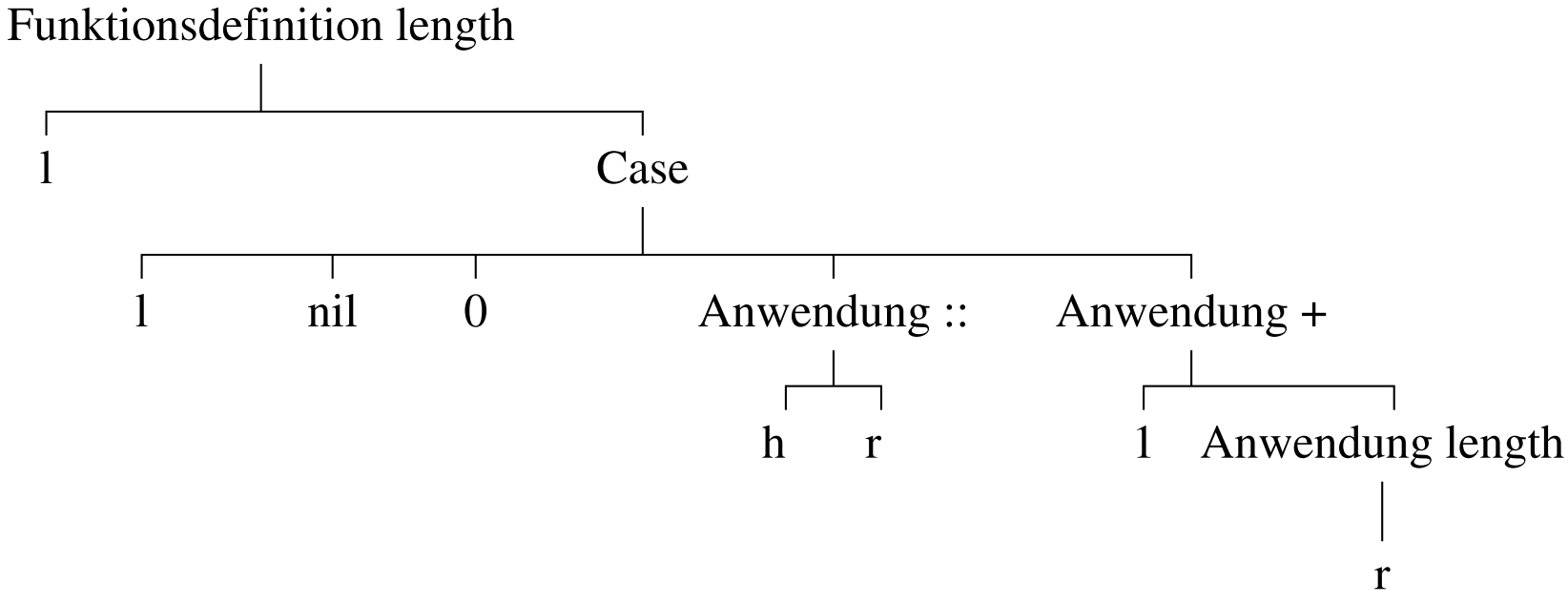
**If:**
$$\frac{p : \texttt{bool} \qquad A : t_1 \qquad B : t_1}{\texttt{if } p \texttt{ then } A \texttt{ else } B : t_1}$$

**Funktionsanwendung/Konstruktor**
$$\frac{f : t_1 \mapsto t_2 \qquad a : t_1}{f\ a : t_2}$$

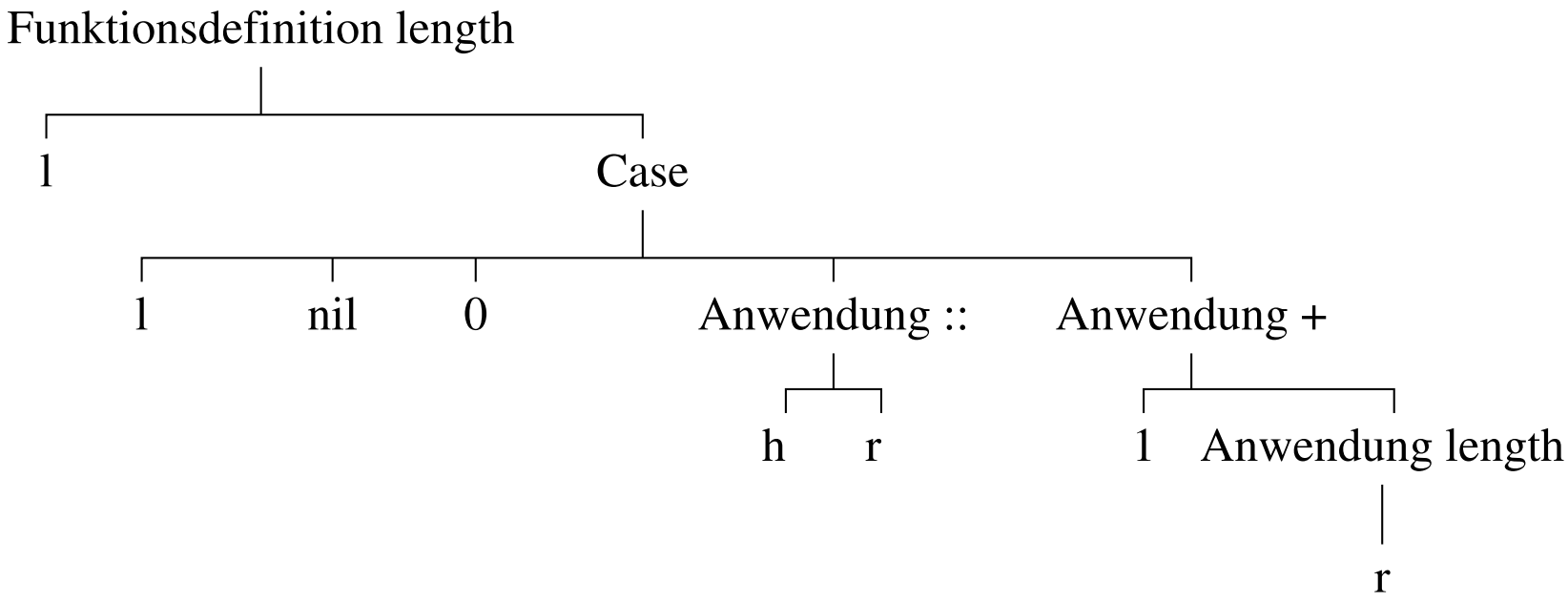
**Funktionsdefinition:**
$$\frac{x : t_1 \qquad e : t_2}{\texttt{fun } name\ x \texttt{ = } e : t_1 \mapsto t_2}$$

# Typ-Inferenz: Bsp.

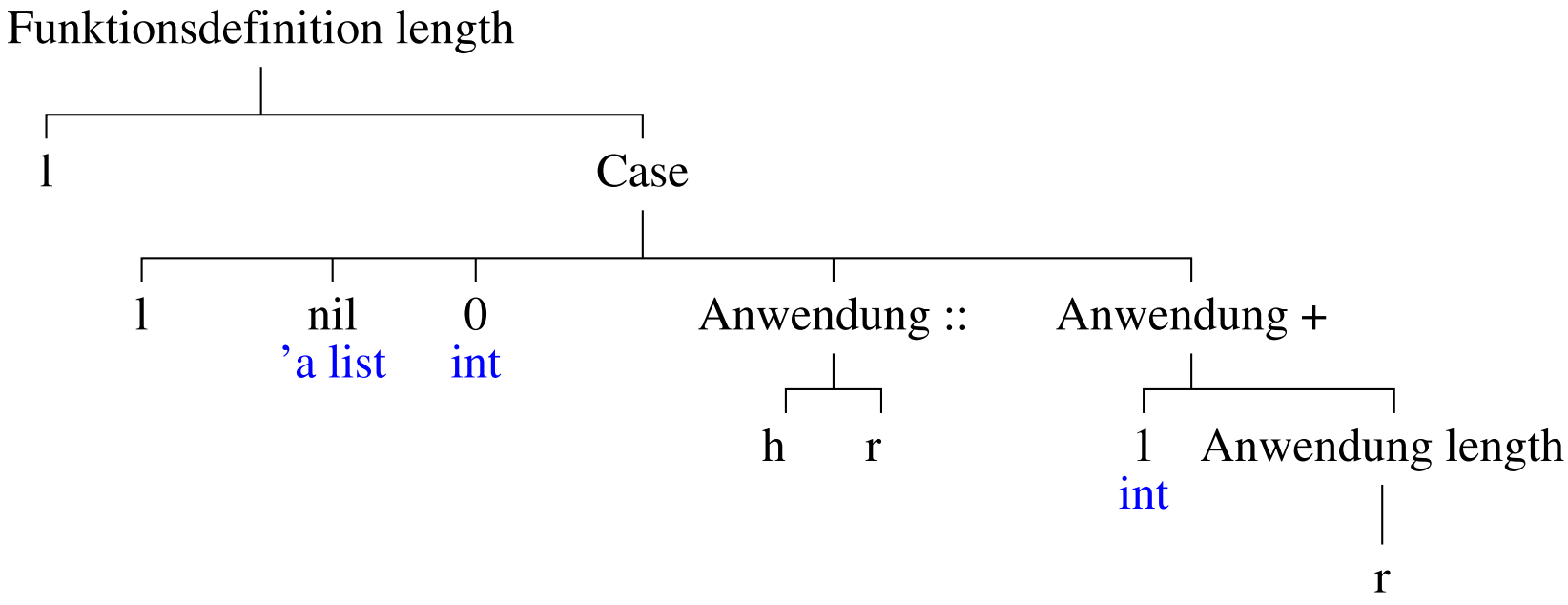
```
fun length l = case l of
                    nil => 0
                  | h::r => 1 + length r
```



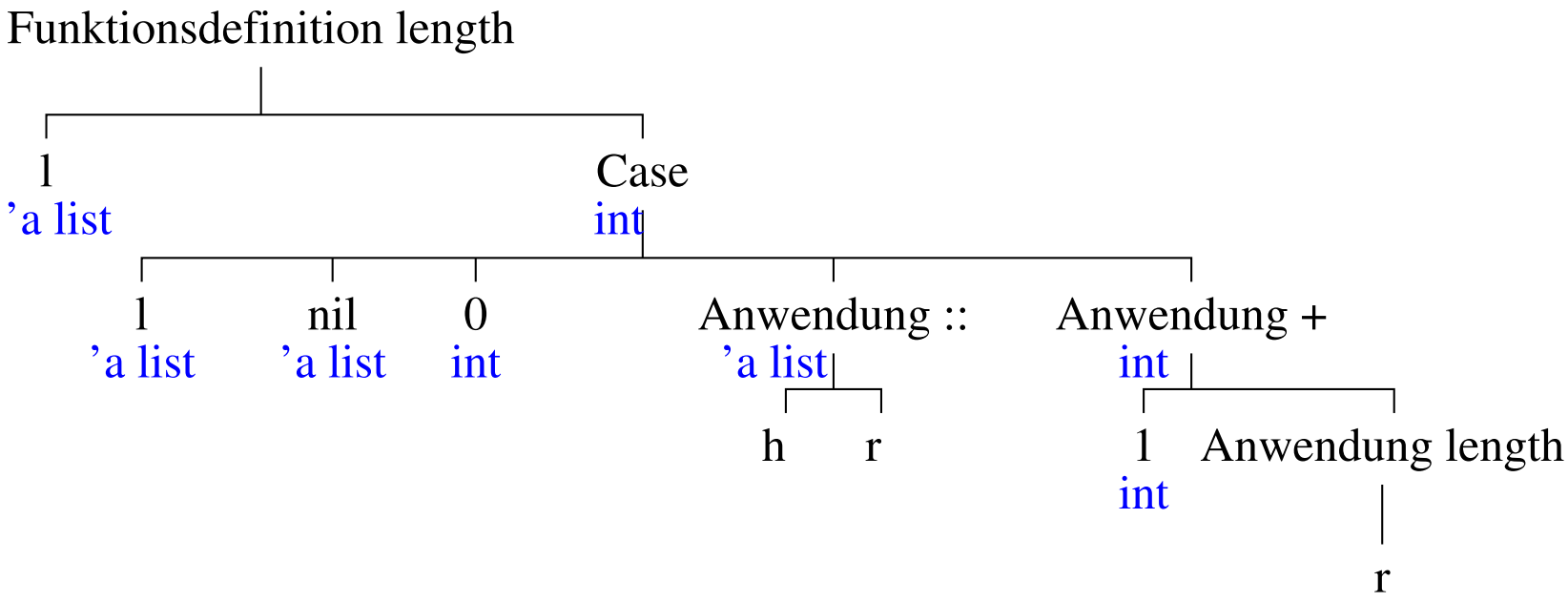
# Typ-Inferenz: Bsp.



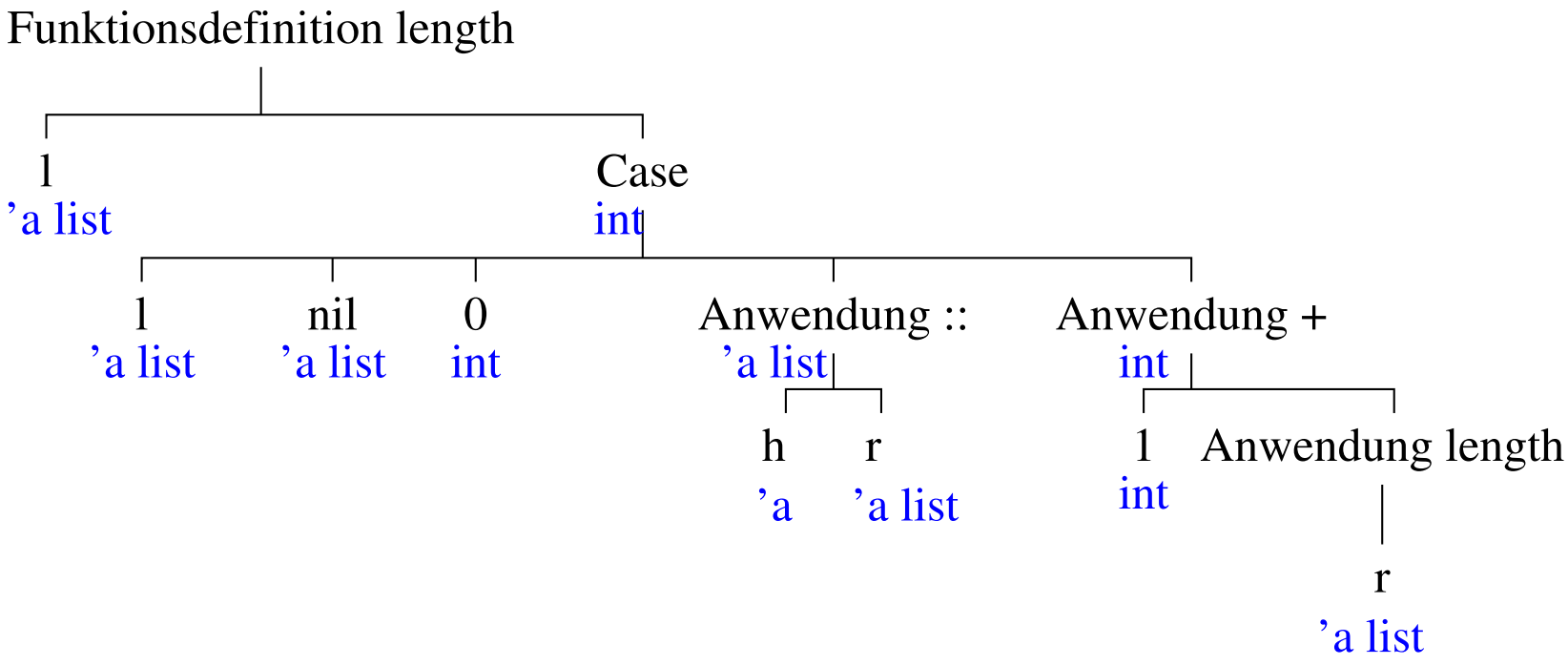
# Typ-Inferenz: 0-stellige Konstrukt./Konstanten



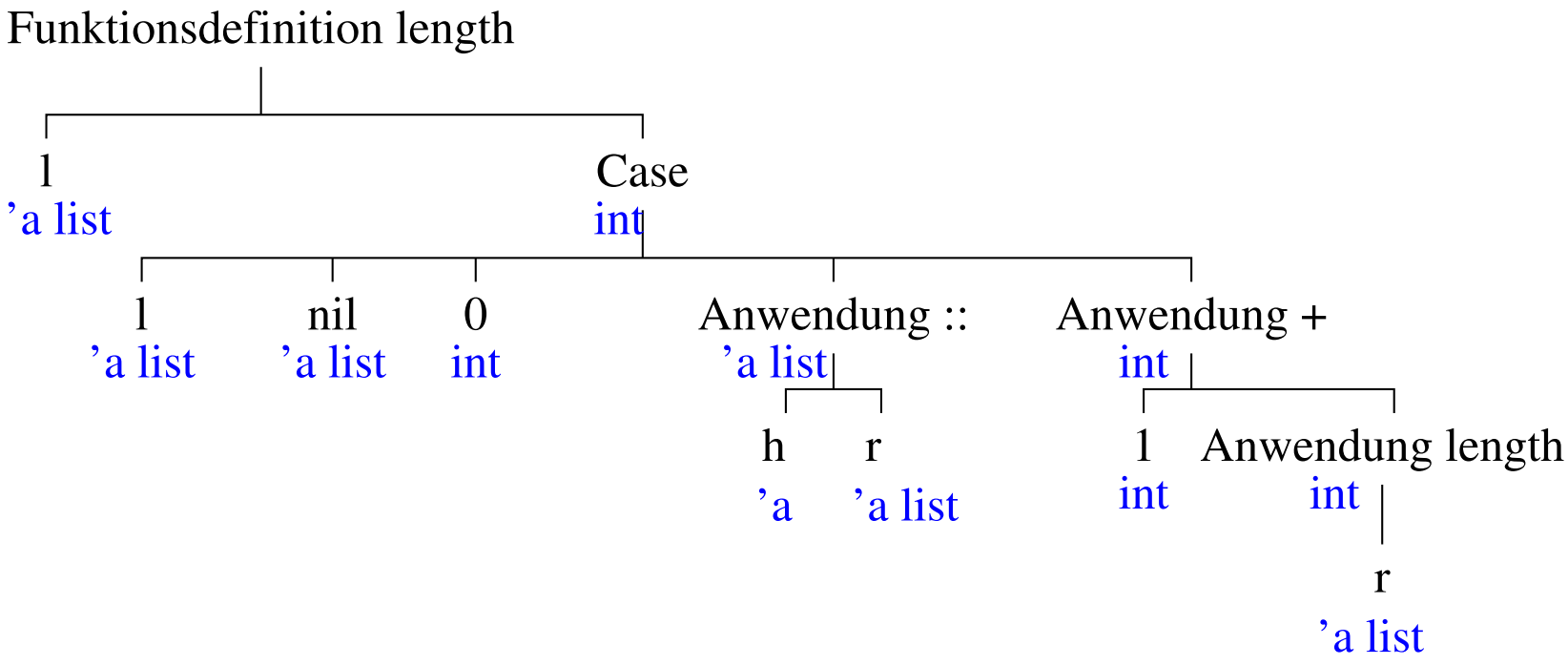
# Typ-Inferenz: Case



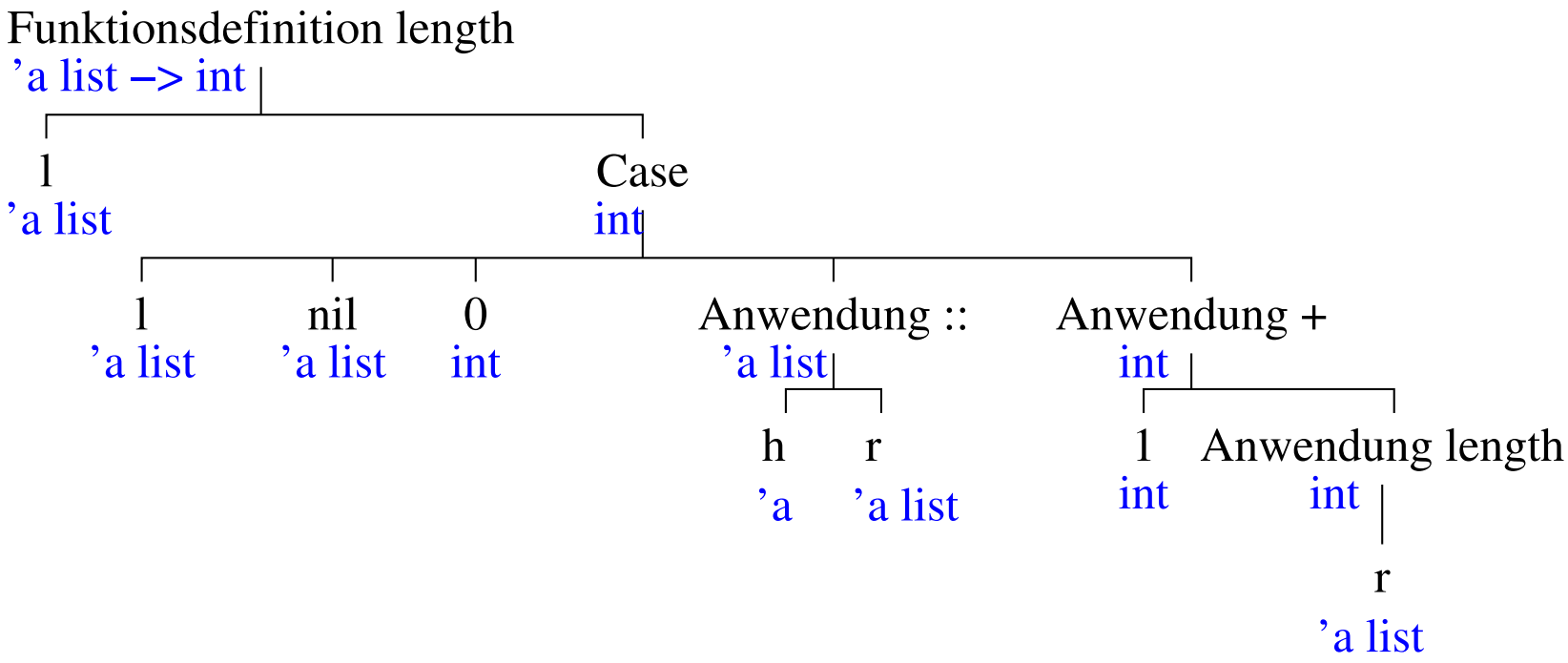
# Typ-Inferenz: Konstruktor ::



# Typ-Inferenz: Anwendung length



# Typ-Inferenz: Fkt.-Definition



# Typ-Inferenz: Idee

Allgemeine Lösung:

- Weise jedem Teilausdruck einen unbekannten Typ (eine Typ-Variable) zu.
- Wende die entsprechenden Inferenz-Regel an jedem Teilausdruck $\Longrightarrow$ System von Term-Gleichungen
- Löse das Term-Gleichungssystem
  - **lösbar**: Lösung gibt Typen für jeden Teilausdruck (inkl. Variablen und Funktionen) an
  - **unlösbar**: Typ-Fehler

# Unifikation

- Lösen von Systemen von Term-Gleichungen = **Unifikation**
  - Lösung = **Substitution** (Unifikator) der Variablen, die die Terme **strukturell gleich** machen.
  - Bsp.:
    - `('a -> ('a * 'b)) list = (int -> ('c * 'c) list)`
    - Lösung: {'a ↦ int, 'b ↦ int, 'c ↦ int}

- Die Lösung ist nicht immer eindeutig
  - Bsp.:
    - `('a -> 'a) = ('b -> 'c)`
    - Lösungen: {'a ↦ int, 'b ↦ int, 'c ↦ int}
      {'a ↦ bool, 'b ↦ bool, 'c ↦ bool}
      ...

# Unifikation

- Satz: Ein System von Term-Gleichungen hat entweder:
  - **keine** Lösung
  - **eine eindeutige** Lösung
  - **beliebig viele** Lösungen. In diesem Fall gibt es eine allgemeinste Lösung.

# Unifikation

- Satz: Ein System von Term-Gleichungen hat entweder:
  - **keine** Lösung
  - **eine eindeutige** Lösung
  - **beliebig viele** Lösungen. In diesem Fall gibt es eine allgemeinste Lösung.

- Die allgemeinste Lösung = **Most general unifier** (mgu)
  - $\sigma$=mgu falls für jeden anderen Unifikator $\theta$ gilt:

$$\theta = \theta' \circ \sigma$$

    für eine Substitution $\theta'$.
  - Bsp.:
    - `('a -> 'a)= ('b -> 'c)`
    - mgu = `{'b` $\mapsto$ `'a, 'c` $\mapsto$ `'a}`
    - `{'a` $\mapsto$ `int,'b` $\mapsto$ `int,'c` $\mapsto$ `int}` = `{'a` $\mapsto$ `int}` $\circ$ `mgu`

# Berechnung des mgu

- **Eingabe:** Zwei Terme $T_1$ und $T_2$
- **Ausgabe:** failure, wenn $T_1$ und $T_2$ nicht unifizierbar sind; ihr mgu, sonst.
- Idee:
  - schreibe Term-Gleichungen in Gleichungen zwischen entsprechenden Teil-Termen um;
  - nutze einen Keller als Workset, um noch nicht gelöste Gleichungen zu speichern;
  - nutzte eine Liste von Substitutionen $\theta$, die die Ausgabe aufsammelt.

# Algorithmus zur Berechnung des mgu

$\theta$ := $\emptyset$; push(stack, $T_1 = T_2$); failure = false;
while not empty(stack) and not failure do
  hole X = Y vom stack runter
  case
    **X ist eine Variable, die in Y nicht auftritt:**
      ersetze X durch Y im stack und in $\theta$
      füge X $\mapsto$ Y zu $\theta$ hinzu
    **Y ist eine Variable, die in X nicht auftritt:**
      ersetze Y durch X im stack und in $\theta$
      füge Y $\mapsto$ X zu $\theta$ hinzu
    **X und Y sind identische Konstanten oder Variablen:** continue
    **X ist $f(X_1, \ldots, X_n)$ und Y ist $f(Y_1, \ldots, Y_n)$ mit $f$=Operator:**
      push(stack, $X_1 = Y_1, X_2 = Y_2, \ldots, X_n = Y_n$)
    **sonst:**
      failure := true
if failure then output *failure* else output $\theta$

# *mgu*-Berechnung: Beispiel

- Zu unifizieren: $T_1$=`(('a * 'b) list) * ('c -> 'd)` und
  $T_2$=`('c list) * ('a * 'd -> 'b)`
- Stack $s \equiv [T_1 = T_2]$; Substitution $\theta \equiv \{\}$
- $s \equiv [$ `('a * 'b) list = 'c list` ; `'c -> 'd = 'a * 'd -> 'b`$]$;
  $\theta \equiv \{\}$
- $s \equiv [$ `'a * 'b = 'c` ; `'c -> 'd = 'a * 'd -> 'b`$]$;$\theta \equiv \{\}$
- $s \equiv [$ `'a * 'b -> 'd = 'a * 'd -> 'b` $]$; $\theta \equiv \{$'c$\mapsto$ 'a * 'b$\}$
- $s \equiv [$ `'a * 'b = 'a * 'd` ;`'d = 'b`$]$; $\theta \equiv \{$'c$\mapsto$ 'a * 'b$\}$
- $s \equiv [$ `'a = 'a` ;`'b = 'd`;`'d = 'b`$]$; $\theta \equiv \{$'c$\mapsto$ 'a * 'b$\}$
- $s \equiv [$ `'b = 'd` ;`'d = 'b`$]$; $\theta \equiv \{$'c$\mapsto$ 'a * 'b$\}$
- $s \equiv [$ `'d = 'd` $]$; $\theta \equiv \{$'c$\mapsto$ 'a * 'd, 'b$\mapsto$'d$\}$
- $s \equiv []$; $\theta \equiv \{$'c$\mapsto$ 'a * 'd, 'b$\mapsto$'d$\}$

# Der *occurs check* Test

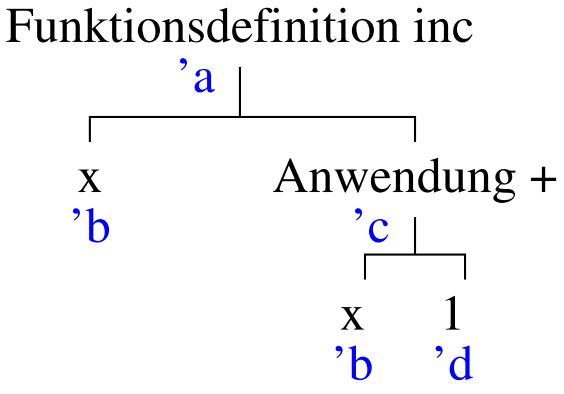
```
θ := ∅; push(stack, T1 = T2); failure = false;
while not empty(stack) and not failure do
   hole X = Y vom stack runter
   case
      ....
      X ist eine Variable, die in Y nicht auftritt:
         ersetze Y durch X im stack und in θ
         füge X = Y zu θ hinzu
      ....

if failure then output failure else output θ
```

- ...stellt sicher, dass die Unifikation terminiert; Z.B. gibt es keine endliche gemeinsame Instanz von `'a` und `'a list`;

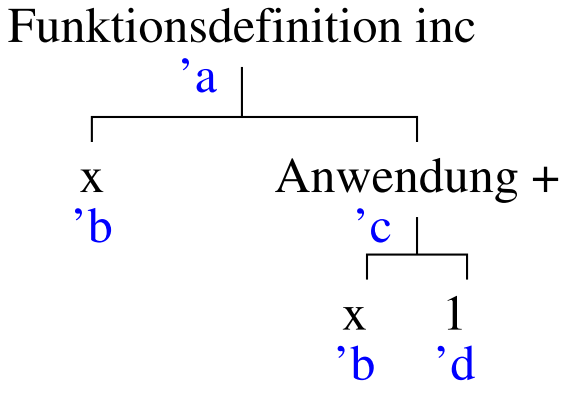
# Typ-Inferenz: Beispiel

```
fun inc x = x + 1
```



# Typ-Inferenz: Beispiel

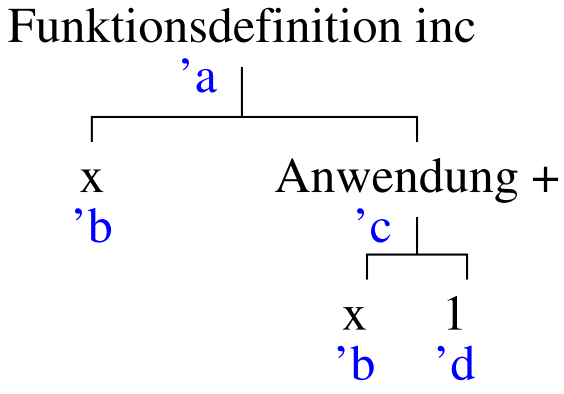
```
fun inc x = x + 1
```




$$\left\{ \begin{array}{l} \text{'a} = \text{'b} \rightarrow \text{'c} \quad \text{(Fkt.-Def. inc)} \end{array} \right.$$

# Typ-Inferenz: Beispiel

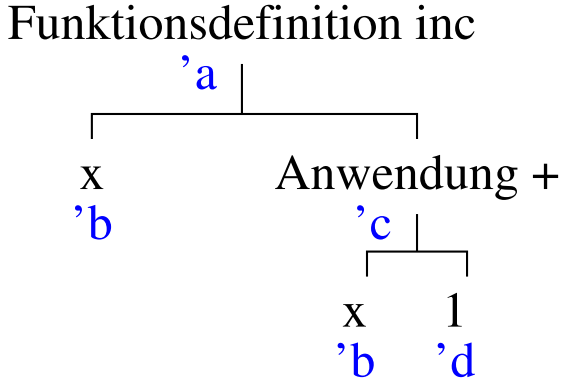
```
fun inc x = x + 1
```




$$\begin{cases} \text{'a} = \text{'b} \rightarrow \text{'c} & \text{(Fkt.-Def. inc)} \\ \text{'c} = \text{int} & \text{(Anwendung +)} \\ \text{'b} = \text{int} & \\ \text{'d} = \text{int} & \end{cases}$$

# Typ-Inferenz: Beispiel

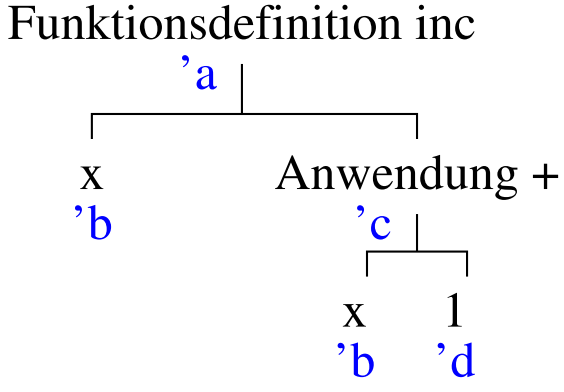
```
fun inc x = x + 1
```




$$
\begin{cases}
\text{'a} = \text{'b} \rightarrow \text{'c} & \text{(Fkt.-Def. inc)} \\
\text{'c} = \text{int} & \text{(Anwendung +)} \\
\text{'b} = \text{int} & \\
\text{'d} = \text{int} & \\
\text{'d} = \text{int} & \text{(Konstante 1)}
\end{cases}
$$

# Typ-Inferenz: Beispiel

```
fun inc x = x + 1
```




$$\begin{cases} \text{'a} = \text{'b} \rightarrow \text{'c} & \text{(Fkt.-Def. inc)} \\ \text{'c} = \text{int} & \text{(Anwendung +)} \\ \text{'b} = \text{int} & \\ \text{'d} = \text{int} & \\ \text{'d} = \text{int} & \text{(Konstante 1)} \end{cases}$$

$$\Longrightarrow \begin{cases} \text{'a} = \text{int} \rightarrow \text{int} \\ \text{'b} = \text{int} \\ \text{'c} = \text{int} \\ \text{'d} = \text{int} \end{cases}$$

# Typ-Inferenz: Rekursive Funktionen

- Der Typ-Inferenz-Algorithmus handelt korrekt **rekursive Funktionen**
- Bsp.:

```
fun sum n = if n<=0 then 0
            else n + sum (n-1)
```

- Der korrekte Typ erhält man durch Benutzung der selben Typ-Variablen in Fkt.-Def und rek. Anwendung ☞ Übung

```
val sum = fn : int -> int
```

# Typ-Inferenz: Polymorphe Funktionen

- **Polymorphe Funktionen** sind Fkt., deren inferierten Funktionstyp Typ-Variablen enthält.

```
fun id x =  x
val id = fn : 'a -> 'a
```

- Problem: wie typt man `(id 1,id true)`?
  - `'a` ist frei in `'a ->'a`
  - `'a` muss bei jeder Anwendung mit dem Operanden-Typ unifizieren:

    $\{'a = \text{int}, 'a = \text{bool} \Longrightarrow$ Typ-Fehler??

- Lösung:
  - univ. Quantifizierung: $\forall$`'a.'a->'a` **(Typ-Schema)**
  - binde frische Typ-Variablen bei jeder (nicht-rek.) Anwendung

    $\{'a_1 = \text{int}, 'a_2 = \text{bool} \Longrightarrow$`(id 1,id true):int*bool`

# Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

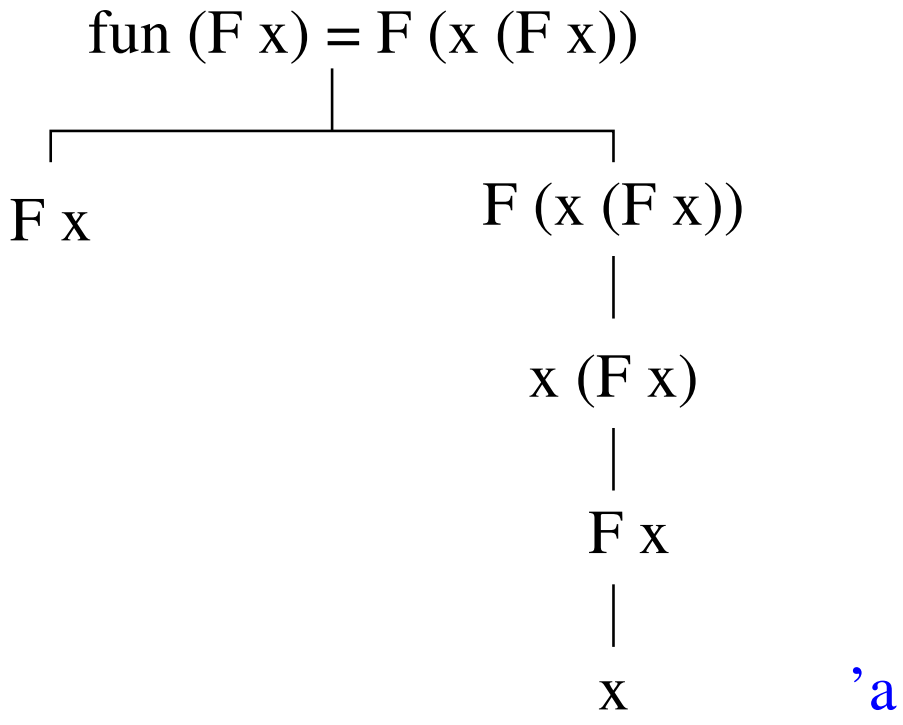
Welchen Typ hat `f`?

# Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?

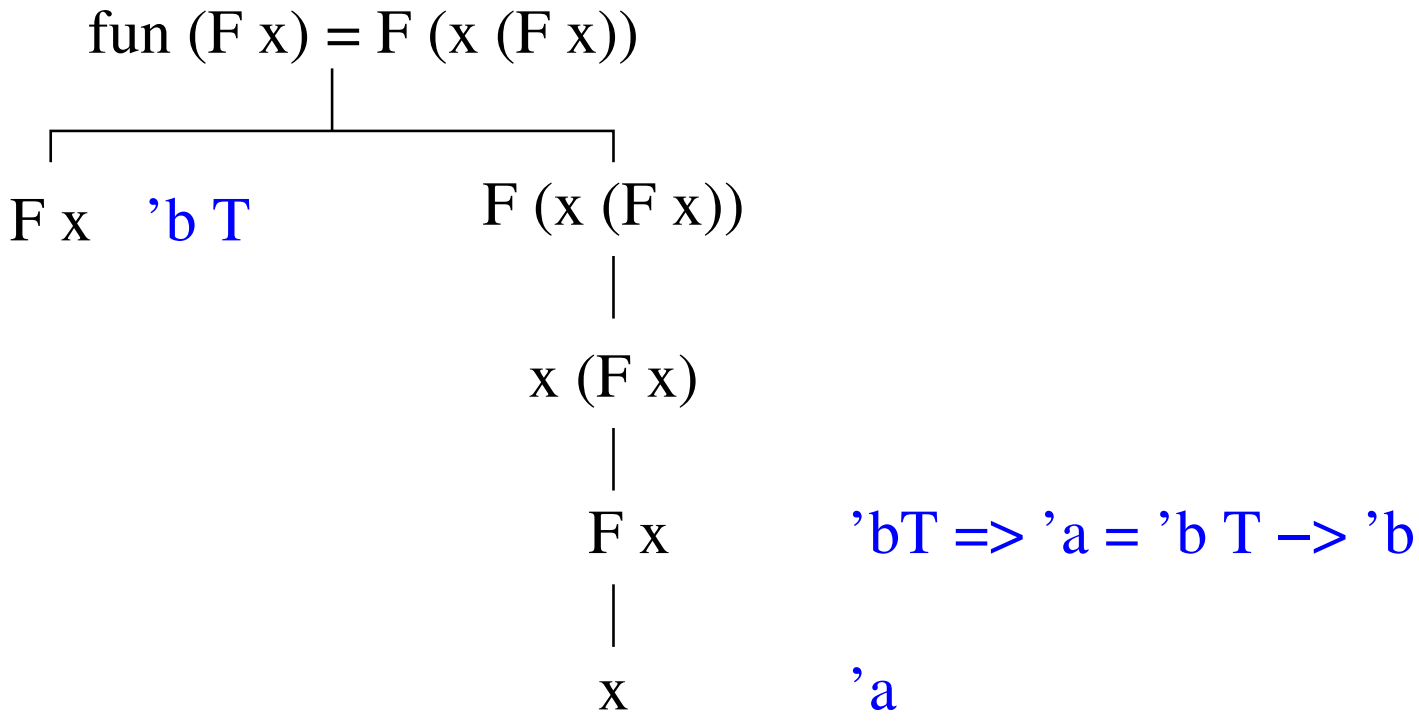


# Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?

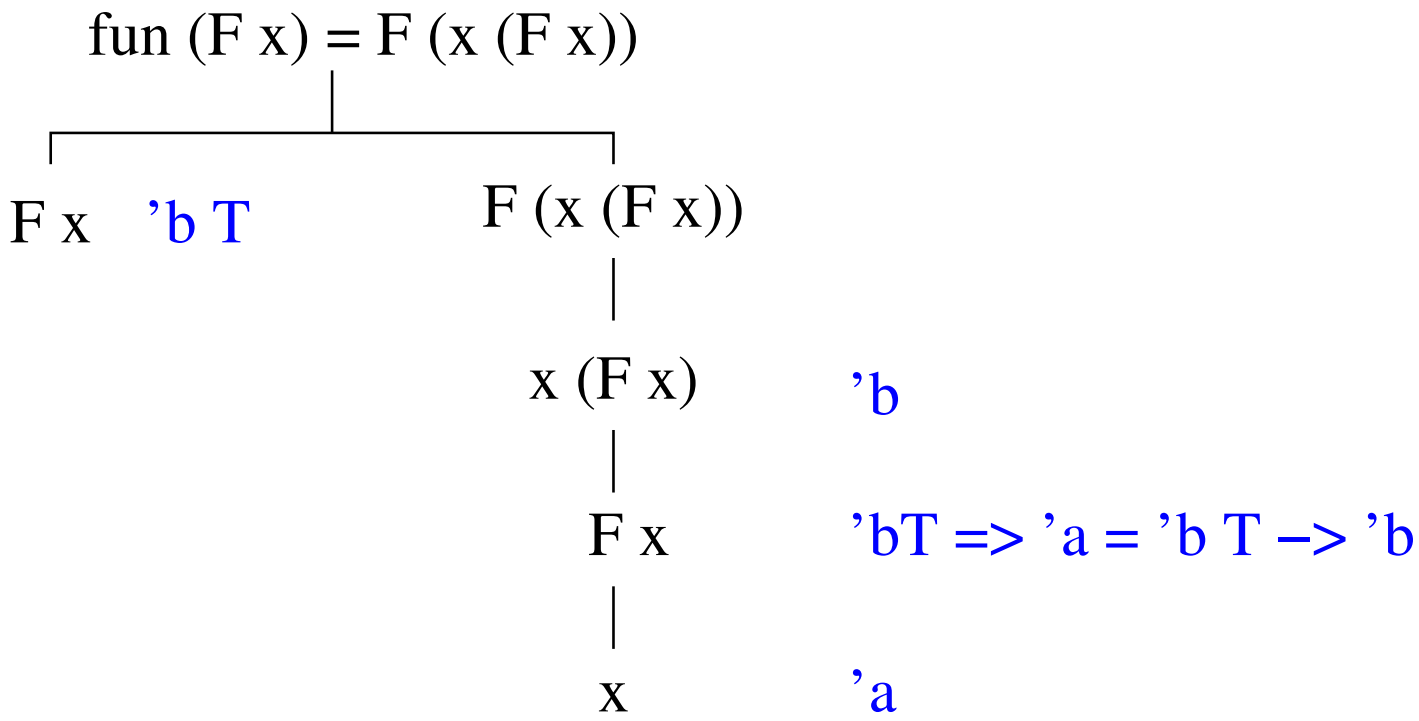


# Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?

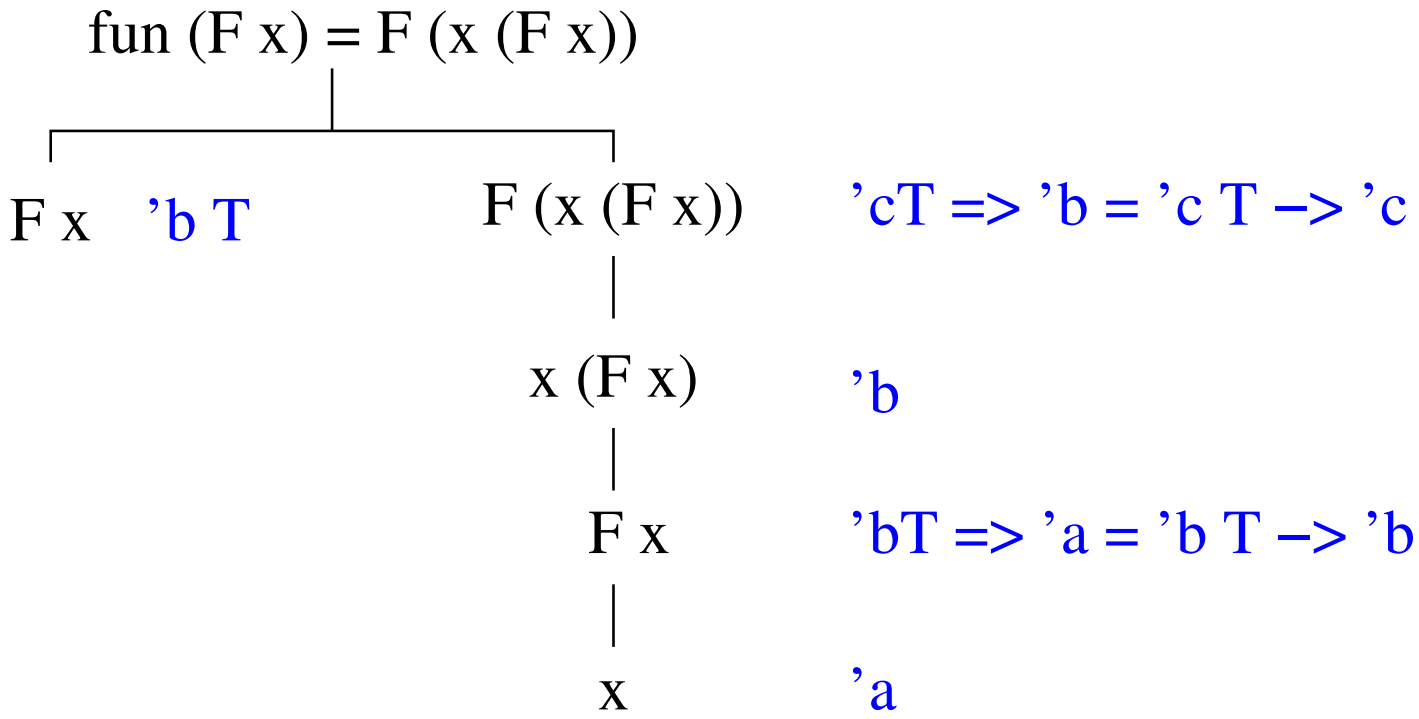


# Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?

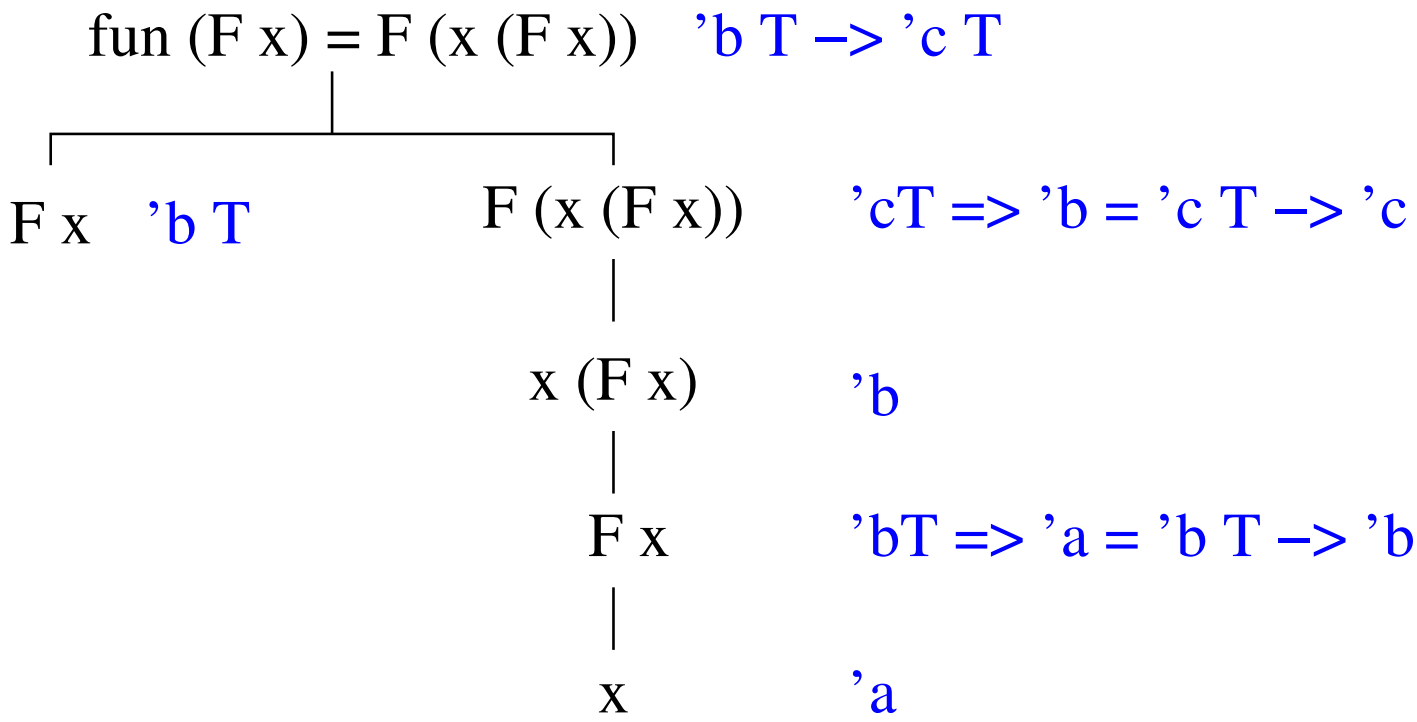


# Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?



# Typ-Inferenz: Fehlermeldungen

- Hat das Gleichungssystem keine Lösung $\Longrightarrow$ Typ-Fehler
- erwünscht: welche Stelle im Programm/Ausdruck den Fehler verursacht
- Idee: Anordnung und Lösen der Gleichungen durch einen DFS-Durchlauf leiten (**Syntax-gerichtet**)
  [☞ Compilerbau Vorlesung]

# Typ-Annotationen

- Wenn dem Programmierer der hergeleitete Typ eines Ausdrucks zu allgemein ist, kann er ihn mit Hilfe von **Typ-Annotationen** einschränken.

```
[];
val it = [] : 'a list
[] : int list;
val it = [] : int list

fun f x = [x];
val f = fn : 'a -> 'a list
fun f x = [x] :int list;
val f = fn : int -> int list
fun f (x :int) = [x];
val f = fn : int -> int list
```

- Die Typ-Annotationen werden als zusätzliche Constraints im Gleichungssystem zur Typ-Inferenz aufgenommen.

# Typ-Annotationen

- Der angegebene Typ muss eine Instanz des hergeleiteten Typs sein:

```
[1] : 'a list;
stdIn:17.1-17.17 Error: expression doesn't match constraint
  expression: int list
  constraint: 'a list
in expression: 1 :: nil: 'a list

fun f x = (x,x) : 'a * 'b;
stdIn:2.6-2.20 Error: expression doesn't match constraint
    expression: 'a * 'a
    constraint: 'a * 'b
in expression: (x,x): 'a * 'b
```

# Typ-Inferenz: Schlussbemerkungen

- Typen und Typ-Systemen erlauben automatisches Typ-Checking $\Longrightarrow$ Vermeiden von Laufzeitfehlern
- Typ-Inferenz entlastet den Programmierer und unterstützt Generizität
- Ähnliche Methoden können in Übersetzerbau und Software-Engineering eingesetzt werden, um Programm-Eigenschaften herauszufinden. (☞ Programmoptimierung).

# Überblick

## 5. Variablen

Variablendefinitionen

Variablengültigkeit

Variablensichtbarkeit

# Variablendefinitionen

Eine Variable $v$ ist ein Paar der Form $(name, wert)$ (geschrieben auch: $name \leftarrow wert$).

- ... bestehen aus *name* und einen Ausdruck für *wert*
- heißen auch **Variablen-Bindungen** (*variable bindings*)
- können explizit oder implizit sein

# Variablendefinitionen

- **Explizite Definition**:

```
val x = 1*2;
```

- **Implizite Definition**:
  - via Funktionsaufrufe

```
fun f x = 42
f (25*4)
```

Der Aufruf `f (25*4)` bindet `x` zu dem Wert von 25*4.

  - via Pattern-Matching mit Variablen-Bindungen

# Variablengültigkeit

- Die **Gültigkeit** einer Variable (*scope*) ist die Menge der Programmpunkte, an denen ihre Definition gilt.

- Top-level Variablen (definiert mit `val name = expr` in der Interpreter-Umgebung) sind gültig an allen nachfolgenden Programmpunkten:

```
val x = 1*2;
val y = x+1;
```

# Variablengültigkeit

- Parameter-Variablen (Funktionsargumente) sind gültig im Rumpf der Funktion

```
fun f x = x+1
```

- Pattern-Variablen sind gültig in der entsprechenden rechten Seite.

```
val description = case f of
                    Rot => "pure red"
                  | Blau => "pure blue"
                  | RGB(x,y,_) =>
                    if (x=y) then “kind of yellow”
                    else “something else”;
```

# Benutzerdefinierte Gültigkeitsbereiche

... können mit Hilfe des `let`-Ausdrucks eingeführt werden:

```
let
  val name = ausdruck
in
  ausdruck'
end
```

- Der Scope der Variable *name* ist *ausdruck'*.
- Der Wert des let-Ausdrucks ist der Wert von *ausdruck'*.

```
let
   val x = 1 + 1
in
  10 * x
end
val it = 20 : int
```

# Der let-Ausdruck

... kann auch den Scope einer Funktionsdefinition einschränken

```
let
  fun square x = x*x
in
  square 2
end;
val it = 4 : int
- square 3;
stdIn:75.1-75.7 Error: unbound variable or constructor: square
```

# Geschachtelte let-Ausdrücke

Oft möchte man geschachtelte Gültigkeitsbereiche:

```
let val x = 1
in let val y = x+1
   in x+y
   end
end;
val it = 3 : int
```

Äquivalent kann man schreiben:

```
let
  val x = 1
  val y = x+1
in x+y
end;
val it = 3 : int
```

# Der let-Ausdruck

Im Allgemeinen:

```
let
  val name_1 = ausdruck_1
  val name_2 = ausdruck_2
  ..........................
  val name_n = ausdruck_n
in
  ausdruck
end
```

- Der Scope der Variable $name_i$ besteht aus $ausdruck_{i+1}$, ..., $ausdruck_n$ und *ausdruck*.
- Der Wert des let-Ausdrucks ist der Wert von *ausdruck*.

# Der let-Ausdruck: Beispiel

```
let
  val increment = 2
  fun add x = x + increment
in
  add 4
end;
val it = 6 : int
```

# Statischer Gültigkeitsbereich

- Der Gültigkeitsbereich einer Variablendefinition in SML und in den meisten modernen Programmiersprachen ist durch die (**statische**) Struktur des Programmtextes definiert.
  $\Longrightarrow$ *static scoping*

- D.h., welche Variablen-Definitionen an einem Programmpunkt gültig sind, hängt nur von der (statischen) Struktur des Programmtextes ab.
  $\Longrightarrow$ *static/lexical scoping* $\equiv$ *static/lexical variable binding*

# Statischer Gültigkeitsbereich: Bsp.

Variablen-Definitionen gültig am Programmpunkt •

```
let
  val sep = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
            nil => ""
          |x::nil => Int.toString x
          |x::rest => • (Int.toString x)^sep^(doit rest)
    in "{"^(doit l)^"}" end
in set2String [1,2,3] end
val it = "{1;2;3}" : string
```

# Statischer Gültigkeitsbereich: Bsp.

Variablen-Definitionen gültig am Programmpunkt •

```
let
  val sep = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
            nil => ""
          |x::nil => Int.toString x
          |x::rest => (Int.toString x)^sep^(doit rest)
    in •"{"^(doit l)^"}" end
in set2String [1,2,3] end
val it = "{1;2;3}" : string
```

# Statischer Gültigkeitsbereich: Bsp.

Variablen-Definitionen gültig am Programmpunkt •

```
let
  val sep = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
            nil => ""
          |x::nil => Int.toString x
          |x::rest => (Int.toString x)^sep^(doit rest)
    in  "{"^(doit l)^"}" end
in • set2String [1,2,3] end
val it = "{1;2;3}" : string
```

# Dynamischer vs. statischer Scoping

- **Dynamic scoping:** welche Variablen-Definition an einem Programmpunkt gültig sind, hängt davon ab, wie dieser bei der Laufzeit erreicht wird.

```
int i = 1;

int f(){return i;}

int g{
  int i = 2;
  return f();
}
```

  - unter **statischem Scoping**: g() liefert **1**
  - unter dynamischen Scoping angenommen: g() liefert 2
    $\Longrightarrow$ Die referentielle Transparenz ist verletzt

# Dynamischer Gültigkeitsbereich: Bsp.

- Beispiel (Scheme):

```
(define mult (lambda (x y) (* x y)))
(define fact (lambda (n)
                (if (= n 0)
                    1
                    (mult (fact (- n 1) n)))))
(fact 3)
6

(define mult (lambda (x y) (y)))
(fact 3)
3
```

- Grund: Die Bindung der top-level Variablen in Scheme ist dynamisch.

# Variablensichtbarkeit

Eine Variablen-Definition ist an einem Programmpunkt $P$ zu einem bestimmten Zeitpunkt $t$ **sichtbar**, wenn:

1. die Variablen-Definition an $P$ zum Zeitpunkt $t$ gültig ist, und
2. alle anderen gültigen Variablen-Definitionen mit dem selben Namen zu einem früheren Zeitpunkt stattgefunden haben.

# Variablen-Sichtbarkeit: Beispiel

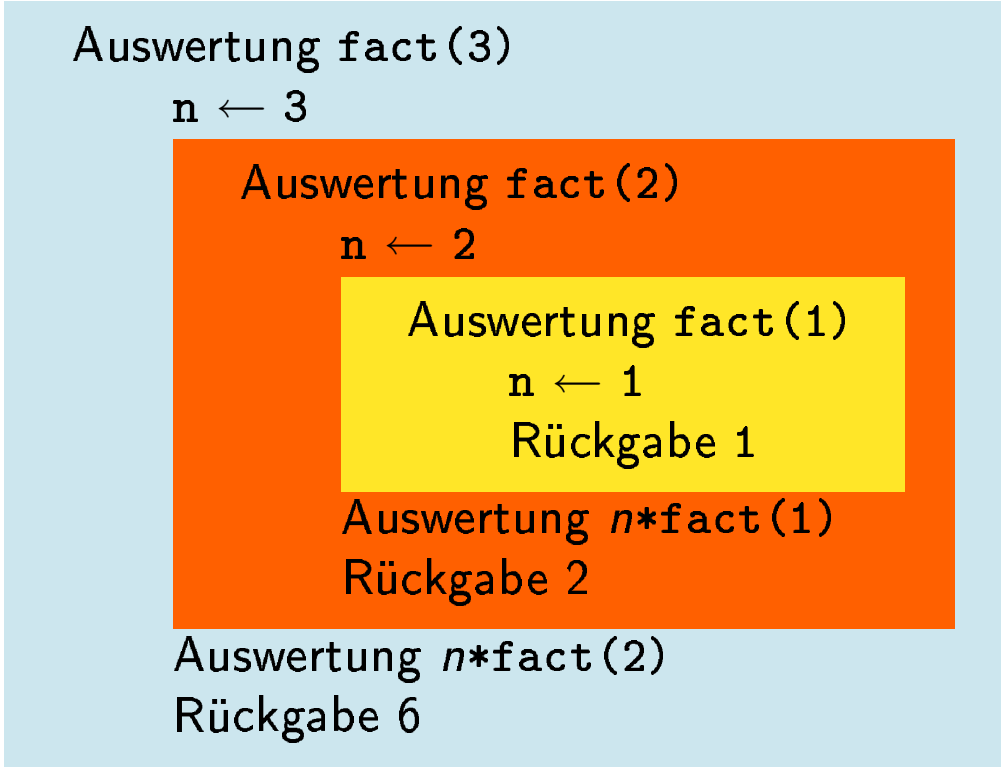
```
let
  val x = 1
in
  (let
     val x = 2
   in
     • x+1
   end) + • x
end
```

- • gültig: $x \leftarrow 1$, $x \leftarrow 2$
  sichtbar: $x \leftarrow 2$

- • gültig: $x \leftarrow 1$
  sichtbar: $x \leftarrow 1$

# Variablen-Sichtbarkeit: Beispiel

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




| Zeit | Var-Def (implizit) | gültig | sichtbar |
|---|---|---|---|
| $t_1$: | (fact 3) | $n \leftarrow 3$ | $n \leftarrow 3$ |
| $t_2$: | (fact 2) | $n \leftarrow 3$, $n \leftarrow 2$ | $n \leftarrow 2$ |
| $t_3$: | (fact 1) | $n \leftarrow 3$, $n \leftarrow 2$, $n \leftarrow 1$ | $n \leftarrow 1$ |

- Der **Kontext** eines Programmpunkts $P$ zu einem bestimmten Zeitpunkt $t$ = Menge sichtbarer Variablen-Definitionen am $P$ zum Zeitpunkt $t$.
  $\Longrightarrow$ ist ein dynamischer Konzept: Dem selben Programmpunkt können bei der Laufzeit verschiedene Kontexte zu verschiedenen Zeitpunkten entsprechen.

$$Kontext(P)_t = \{n \leftarrow 1\}$$
$$Kontext(P)_{t+\Delta t} = \{n \leftarrow 2\}$$

# Kontext: Beispiel

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

Kontext von •:

| Zeit | Kontext |
|---|---|
| $t_1$: (fact 3) | $n \leftarrow 3$ |
| $t_2$: (fact 2) | $n \leftarrow 2$ |
| $t_3$: (fact 2) | $n \leftarrow 1$ |

# Überblick

## 6. Funktionale Abschlüsse

Currying

Partielle Anwendung

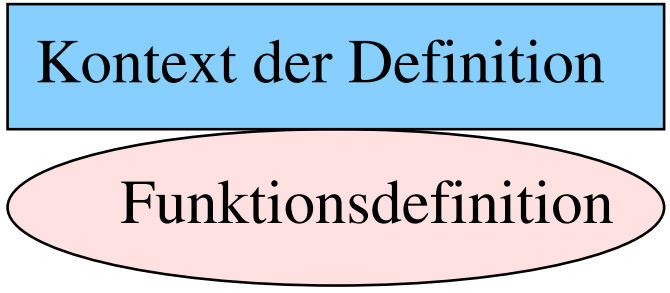
Funktionen höherer Ordnung

# Der Funktionstyp-Operator

- **Der Funktionstyp-Operator:** $- > : MT \times MT \mapsto MT$
  - $\boxed{\alpha \; - > \; \beta} = \{f : \alpha \mapsto \beta \mid \alpha, \beta \in MT\}$
  - rechtsassoziativ: $\alpha \mapsto \beta \mapsto \gamma \equiv \alpha \mapsto (\beta \mapsto \gamma)$

# Funktionale Abschlüsse

- Eine Funktion besteht aus der Funktionsdefinition und der Kontext des Programmpunktes an welchen die Funktion definiert ($\equiv$ konstruiert) wird.



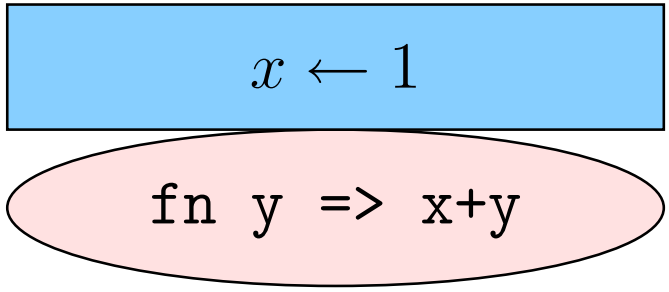

- Funktionen schließen ihren Kontext zum Zeitpunkt ihrer Definition ab.

$\Longrightarrow$ Eine Funktion wird auch **funktionaler Abschluss** (*closure*) genannt.

# Funktionaler Abschluss: Beispiel

```
val x = 1
• val f = fn y => x + y
val v1 = f 3;
val v1 = 4 : int

val x =2
val v2 = f 3;
val v2 = 4 : int
```



# Curry-Funktionen (*curried functions*)

**Currying** = Methode mit der man Funktionen von mehreren Argumenten konstruieren kann.

- genannt nach dem Erfinder, Haskell B. Curry

```
val sum = fn x => (fn y => x+y);
val sum = fn : int -> int -> int
```

`sum` erwartet ein Argument `x` und liefert eine Funktion zurück, die wiederum ein Argument `y` erwartet und `x+y` zurückliefert.

- Wegen der Rechtsassoziativität von => kann man auch schreiben:

```
val sum = fn x => fn y => x+y;
val sum = fn : int -> int -> int
```

# Curry-Funktionen

- **Funktionsanwendung** (Aufruf):

```
val sum = fn x => fn y => x+y;
(sum 2) 3;
val it = 5 : int
```

- Die Funktionsanwendung ist linksassoziativ:

$$\texttt{f x1 x2 x3} \equiv \texttt{(((f x1) x2) x3)}$$

Deshalb kann man auch schreiben:

```
sum 2 3;
val it = 5 : int
```

# Verkürzte Syntax

- Statt:

```
val sum = fn x => fn y => x+y;
val sum = fn : int -> int -> int
```

kann man mit verkürzter Syntax die Funktion `sum` so definieren:

```
fun sum x y = x+y;
val sum = fn : int -> int -> int
```

- I.a. ist:

```
fun f x1 x2 ... xn = expr
```

das selbe wie:

```
val f = fn x1 => fn x2 => ... fn xn => expr
```

# Partielle Anwendung

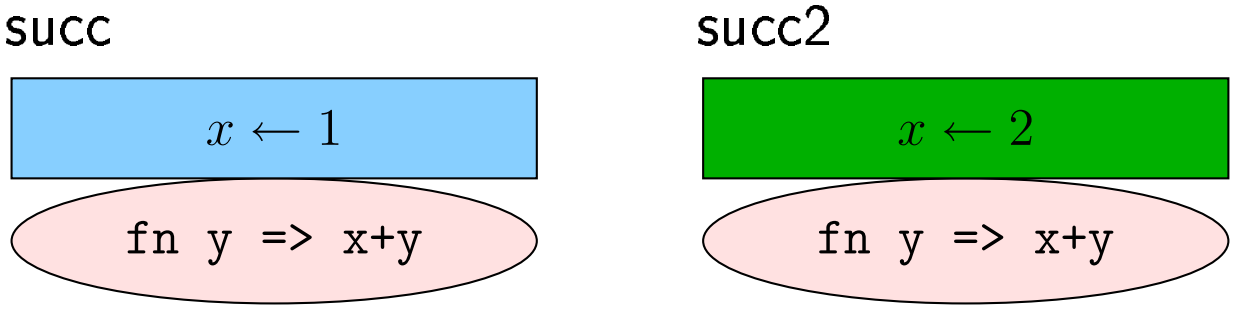
- Curry-Funktionen können **unterversorgt sein**, d.h. auf weniger Argumente als in der Deklaration angewendet werden.

- Liefern dann eine Funktion zurück, die den Rest der Argumente erwartet.
  ($\Longrightarrow$ Curry-Funktionen sind Funktionen höherer Ordnung)

```
fun f x y z t = x+y+z+t;
val f = fn : int -> int -> int -> int -> int
- val f1 = f 10;
val f1 = fn : int -> int -> int -> int
- val f2 = f1 20 30;
val f2 = fn : int -> int
- val v = f2 40;
val v = 100 : int
```

# Partielle Anwendung: Beispiel

```
- val sum = fn x => fn y => x + y;
val sum = fn : int -> int -> int
- • val succ = sum 1;
val succ = fn : int -> int
- succ 16;
val it = 17 : int
- • val succ2 = sum 2;
val succ2 = fn : int -> int
- succ2 16;
val it = 18 : int
```



# Funktionen höherer Ordnung

- ...bekommen Funktionen als Argumente (heißen auch Funktionale)...
- Bsp.: `map f l` wendet `f` auf jedes Element aus `l` an und liefert die Liste der Ergebnisse.

```
- fun map f l =
    case l of nil => nil
            | h::r => (f h)::map f r
val map = fn : ('a -> 'b) -> 'a list -> 'b list

- map (fn x => x+1) [1,2,3,4];
val it = [2,3,4,5] : int list
```

# Funktionen höherer Ordnung

- Bsp.: `filter p l` wendet `p` auf jedes Element `x` aus `l` an und liefert die Liste aller `x`, für welche `p x` den Wert `true` hat.

```
- fun filter p l =
    case l of   nil => nil
              | h::r => if p h then h::(filter p r)
                        else (filter p r);
val filter = fn : ('a -> bool) -> 'a list -> 'a list

- fun greaterThan c x = x>c;
val greaterThan = fn : int -> int -> bool
- val greaterThanFive = greaterThan 5;
val greaterThanFive = fn : int -> bool

- filter greaterThanFive [1,6,3,7,9,4,8];
val it = [6,7,9,8] : int list
```

# Funktionen höherer Ordnung:

- ...liefern Funktionen als Ergebnisse zurück:

```
fun curry f = fn x => fn y => f (x,y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

Int.max(2,7);
val it = 7 : int

- map (curry Int.max 3) [1,2,3,4,5,6];
val it = [3,3,3,4,5,6] : int list

fun uncurry f = fn (x,y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c

uncurry (fn x=> fn y => x+y);
val it = fn : int * int -> int
```

# Das Sieb des Eratosthenes

| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | | 5 | | 7 | | 9 | | 11 | | 13 | | 15 |
| 2 | 3 | | 5 | | 7 | | | | 11 | | 13 | | |
| 2 | 3 | | 5 | | 7 | | | | 11 | | 13 | | |
| 2 | 3 | | 5 | | 7 | | | | 11 | | 13 | | |

# Funktionen höherer Ordnung: Beispiele

Das Sieb des Eratosthenes:

```
fun list_n i n = if i>n then nil
                 else i::(list_n (i+1) n)
val firstTen = list_n 2 10;
val firstTen = [2,3,4,5,6,7,8,9,10] : int list
```

# Funktionen höherer Ordnung: Beispiele

Das Sieb des Eratosthenes:

```
fun list_n i n = if i>n then nil
                 else i::(list_n (i+1) n)
val firstTen = list_n 2 10;
val firstTen = [2,3,4,5,6,7,8,9,10] : int list


fun sieve n = filter (fn x => x mod n <> 0)
val sieve = fn : int -> int list -> int list
val l1 = sieve 2 firstTen
val it = [3,5,7,9] : int list
```

# Funktionen höherer Ordnung: Beispiele

Das Sieb des Eratosthenes:

```
fun list_n i n = if i>n then nil
                 else i::(list_n (i+1) n)
val firstTen = list_n 2 10;
val firstTen = [2,3,4,5,6,7,8,9,10] : int list


fun sieve n = filter (fn x => x mod n <> 0)
val sieve = fn : int -> int list -> int list
val l1 = sieve 2 firstTen
val it = [3,5,7,9] : int list


fun iter l primes = case l of nil => primes
                  | h::r => iter (sieve h r) (h::primes)
val iter = fn : int list -> int list -> int list
```

# Funktionen höherer Ordnung: Beispiele

Das Sieb des Eratosthenes:

```
fun list_n i n = if i>n then nil
                 else i::(list_n (i+1) n)
val firstTen = list_n 2 10;
val firstTen = [2,3,4,5,6,7,8,9,10] : int list


fun sieve n = filter (fn x => x mod n <> 0)
val sieve = fn : int -> int list -> int list
val l1 = sieve 2 firstTen
val it = [3,5,7,9] : int list


fun iter l primes = case l of nil => primes
                  | h::r => iter (sieve h r) (h::primes)
val iter = fn : int list -> int list -> int list

fun eratostenes n = iter (list_n 2 n) nil
eratostenes 10;
val it = [7,5,3,2] : int list
```

# Überblick

## 7. Verzögerte Auswertung

Auswertungsstrategien

Benutzer-kontrollierte Auswertung

Unendliche Datenstrukturen

# Auswertungsstrategien

- Wann werden Ausdrücke ausgewertet?
- Die meisten Sprachen legen sich auf einer Strategie fest:
  - **Strikte Auswertung** (*eager-evaluation*, strict evaluation, eifrige/vollständige Auswertung): Ein Ausdruck wird ausgewertet, sobald er an einer Variable gebunden wird.
    $\Longrightarrow$ SML, Java, C
  - **Verzögerte Auswertung** (*lazy-evaluation*, delayed evaluation): Ein Ausdruck wird ausgewertet, sobald er zur Auswertung eines umgebenden Ausdruckes gebraucht wird.
    $\Longrightarrow$ Miranda, Haskell

# Parameterübergabe bei “strikter” Auswertung

- Betrachten wir den folgenden SML-Code:

```
- fun f (x,y,z) = if x=0 then y else z;
val f = fn : int * 'a * 'a -> 'a
```

# Parameterübergabe bei "strikter" Auswertung

- Betrachten wir den folgenden SML-Code:

```
- fun f (x,y,z) = if x=0 then y else z;
val f = fn : int * 'a * 'a -> 'a
- fun h x = f(x,1,1 div x);
val h = fn : int -> int
```

# Parameterübergabe bei "strikter" Auswertung

- Betrachten wir den folgenden SML-Code:

```
- fun f (x,y,z) = if x=0 then y else z;
val f = fn : int * 'a * 'a -> 'a
- fun h x = f(x,1,1 div x);
val h = fn : int -> int
- h 0;
uncaught exception divide by zero raised at: <file stdIn>
```

- Grund: Bei der Auswertung des Aufrufs `f(0,1,1 div 0)` werden die **aktuellen Parameter** 0, `1`, `1 div 0` **ausgewertet**, wenn sie zu den **formalen Parameter** `x`, `y`, `z` gebunden werden.

- Diese Art der Übergabe der aktuellen Parameter (wie in SML,Java,C) heißt **Wertübergabe** (*call by value*)

# Parameterübergabe "verzögerte" Auswertung

- Nähmen wir verzögerte Auswertung an:

```
- fun f (x,y,z) = if x=0 then y else z;
val f = fn : int * 'a * 'a -> 'a
- fun h x = f(x,1,1 div x);
val h = fn : int -> int
- h 0;
1
```

- Grund: zur Auswertung des Aufrufs `f(0,1,1 div 0)` ist die Auswertung von `1 div 0` nicht nötig.

- Wenn ein Parameter ausgewertet wird, immer wenn sein Wert gebraucht wird $\Longrightarrow$ **call by name**. (Algol)

- Wenn das Ergebnis der ersten Auswertung eines Parameter gemerkt wird, und nachträglich nachgeschlagen, immer wann der Wert gebraucht wird $\Longrightarrow$ **call by need**. (Haskell)

# Benutzer-kontrollierte Auswertung

- Mit Hilfe funktionaler Abschlüsse kann man Ausdrücke **kontrolliert auswerten**.
  $\Longrightarrow$ In funktionalen Sprachen kann man eigene Auswertungsstrategien entwickeln

- Simulierung verzögerter Auswertung:

```
- fun f (x,y,z) = if x=0 then y() else z();
val f = fn : int * (unit -> 'a) * (unit -> 'a) -> 'a
- fun h x = f(x,fn () => 1,fn () => 1 div x);
val h = fn : int -> int
- h 0;
val it = 1 : int
```

# Unendliche Datenstrukturen

- Ein Vorteil der verzögerten Auswertung: Darstellung unendlicher Datenstrukturen.

- Erster Versuch: Datentyp zur Darstellung unendlicher Folgen (*streams*, **Ströme**):
  - Ein Stream ist ein Paar `Stream(firstTerm,restStream)`:

```
- datatype 'a stream = Stream of 'a * 'a stream
```

# Unendliche Datenstrukturen

- Ein Vorteil der verzögerten Auswertung: Darstellung unendlicher Datenstrukturen.

- Erster Versuch: Datentyp zur Darstellung unendlicher Folgen (*streams*, **Ströme**):
  - Ein Stream ist ein Paar `Stream(firstTerm,restStream)`:

```
- datatype 'a stream = Stream of 'a * 'a stream
```

```
fun generateNat n =
    Stream (n, generateNat (n+1));
val generateNat = fn : int -> int stream
```

# Unendliche Datenstrukturen

- Ein Vorteil der verzögerten Auswertung: Darstellung unendlicher Datenstrukturen.
- Erster Versuch: Datentyp zur Darstellung unendlicher Folgen (*streams*, **Ströme**):
  - Ein Stream ist ein Paar `Stream(firstTerm,restStream)`:

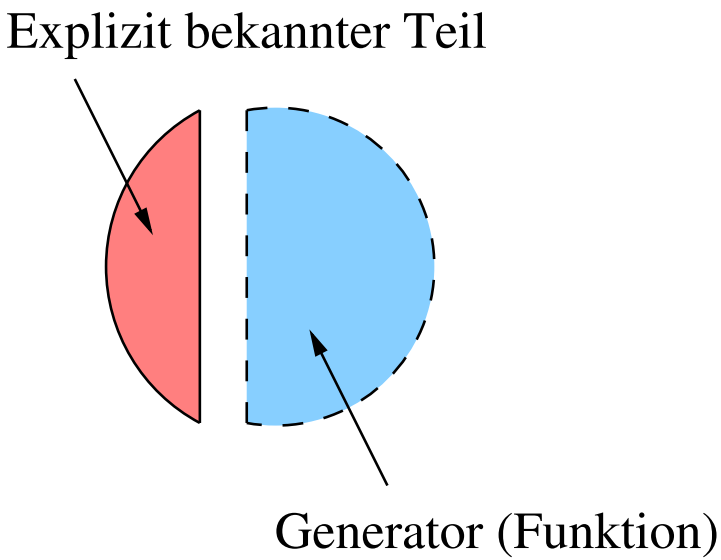
```
- datatype 'a stream = Stream of 'a * 'a stream
```

```
  fun generateNat n =
      Stream (n, generateNat (n+1));
val generateNat = fn : int -> int stream
```

  - Wegen der strikten Auswertung terminiert `generateNat 0` nie.

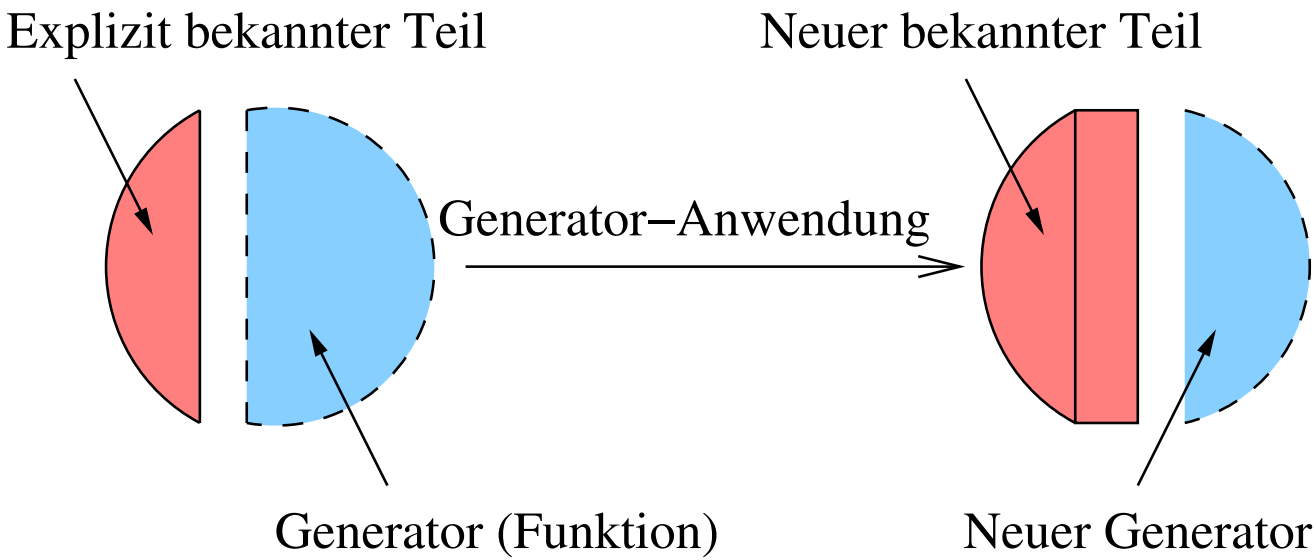
# Unendliche Datenstrukturen

- Idee:



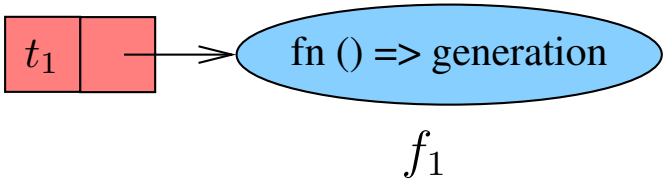
# Unendliche Datenstrukturen

- Idee:

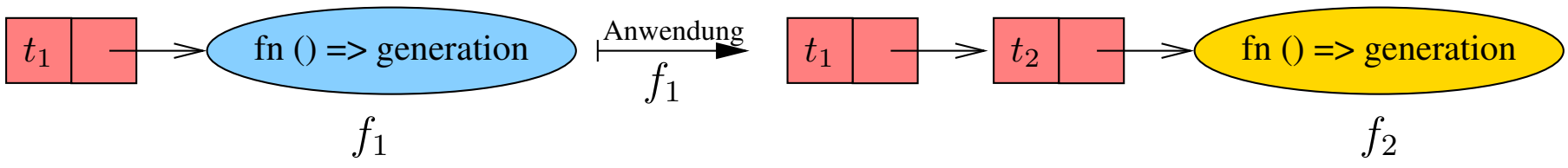


# Unendliche Datenstrukturen

# Unendliche Datenstrukturen

# Unendliche Datenstrukturen

# Unendliche Datenstrukturen

- Zweiter Versuch: mit funktionalen Abschlüssen:

```
datatype 'a stream =
    Stream of 'a * (unit -> 'a stream)
```

  - Dieser Datentyp kann keine endlich größen Daten repräsentieren, denn es fehlt einen nicht-rekursiver Konstruktor.
  - Das zweite Argument für `Stream` (Rest der Strömes) ist vom Typ `(unit -> 'a stream)`. Es ist also eine Funktion, die, wenn sie auf `()` angewandt wird, den Rest des Stroms ergibt.

# Unendliche Datenstrukturen

- Zweiter Versuch: mit funktionalen Abschlüssen:

```
datatype 'a stream =
    Stream of 'a * (unit -> 'a stream)
```

  - Dieser Datentyp kann keine endlich größen Daten repräsentieren, denn es fehlt einen nicht-rekursiver Konstruktor.
  - Das zweite Argument für `Stream` (Rest der Strömes) ist vom Typ `(unit -> 'a stream)`. Es ist also eine Funktion, die, wenn sie auf `()` angewandt wird, den Rest des Stroms ergibt.

```
fun generateNat n =
      Stream (n,fn () => generateNat (n+1));
val generateNat = fn : int -> int stream
val nats = generateNat 0;
val nats = Stream (0,fn) : int stream
```

  - `generateNat 0` terminiert!

# Verarbeitung unendlicher Datenstrukturen

```
- fun sum n (Stream (x,rest)) =
    if n=0 then 0
    else x + sum (n-1) (rest());
val sum = fn : int -> int stream -> int
```

- Der Rest des Stroms wird erzeugt, indem man `rest` auf `()` anwendet. Erst dadurch wird das nächste Element (und die Funktion, die den weiteren Rest des Stroms darstellt) erzeugt.

```
- sum 10 nats;
val it = 45 : int

- sum 1000 nats;
val it = 499500 : int
```

# Verarbeitung unendlicher Datenstrukturen

- Analog wie bei Listen kann man eine Reihe von nützlichen Funktionen für Ströme (≡ unendliche Listen) definieren:

```
- fun head (Stream (x,_)) = x
  fun tail (Stream (_,xs)) = xs()
```

# Verarbeitung unendlicher Datenstrukturen

- Analog wie bei Listen kann man eine Reihe von nützlichen Funktionen für Ströme (≡ unendliche Listen) definieren:

```
- fun head (Stream (x,_)) = x
  fun tail (Stream (_,xs)) = xs()

- head nats;
val it = 0 : int
- tail nats;
val it = Stream (1,fn) : int stream
- head(tail(tail(tail nats)));
val it = 3 : int
```

# Verarbeitung unendlicher Datenstrukturen

- Analog wie bei Listen kann man eine Reihe von nützlichen Funktionen für Ströme (≡ unendliche Listen) definieren:

```
- fun head (Stream (x,_)) = x
  fun tail (Stream (_,xs)) = xs()

- head nats;
val it = 0 : int
- tail nats;
val it = Stream (1,fn) : int stream
- head(tail(tail(tail nats)));
val it = 3 : int

- fun nth n s = if n=0 then head s
                else nth (n-1) (tail s)
```

# Verarbeitung unendlicher Datenstrukturen

- Extrahieren einer endlichen Teilliste:

```
- fun take n s =
    if n = 0 then nil
    else (head s)::(take (n-1) (tail s));
val take = fn : int -> 'a stream -> 'a list
- take 10 nats;
val it = [0,1,2,3,4,5,6,7,8,9] : int list
```

# Verarbeitung unendlicher Datenstrukturen

- Extrahieren einer endlichen Teilliste:

```
- fun take n s =
    if n = 0 then nil
    else (head s)::(take (n-1) (tail s));
val take = fn : int -> 'a stream -> 'a list
- take 10 nats;
val it = [0,1,2,3,4,5,6,7,8,9] : int list
```

- Funktionen höherer Ordnung (*Funktionale*):

```
fun map f s = Stream (f (head s),
                      fn () => map f (tail s))
val map = fn : ('a -> 'b) -> 'a stream -> 'b stream
```

# Verarbeitung unendlicher Datenstrukturen

- Extrahieren einer endlichen Teilliste:

```
- fun take n s =
    if n = 0 then nil
    else (head s)::(take (n-1) (tail s));
val take = fn : int -> 'a stream -> 'a list
- take 10 nats;
val it = [0,1,2,3,4,5,6,7,8,9] : int list
```

- Funktionen höherer Ordnung (*Funktionale*):

```
fun map f s = Stream (f (head s),
                      fn () => map f (tail s))
val map = fn : ('a -> 'b) -> 'a stream -> 'b stream

fun filter f s =
   if f (head s) then
      Stream(head s,fn () => filter f (tail s))
   else filter f (tail s)
val filter = fn : ('a -> bool) -> 'a stream -> 'a stream
```

# Verarbeitung unendlicher Datenstrukturen

- Jetzt können wir z.B. die unendliche Liste aller geraden Zahlen oder aller Quadratzahlen berechnen:

```
- take 10 (filter (fn x => x mod 2=0) nat);
val it = [0,2,4,6,8,10,12,14,16,18] : int list


- take 10 (map (fn x => x*x) nat);
val it = [0,1,4,9,16,25,36,49,64,81] : int list
```

# Unendliche Datenstrukturen

- So können wir auch die Liste aller Primzahlen berechnen (Sieb des Eratosthenes):

```
fun all_primes () =
 let
   fun sieve (Stream (n,ns)) =
     Stream
      (n,
       fn() => sieve
                (filter (fn x => x mod n<>0) (ns())))
      )
 in
   sieve (generateNat 2)
 end;
```

# Unendliche Datenstrukturen

- So können wir auch die Liste aller Primzahlen berechnen (Sieb des Eratosthenes):

```
fun all_primes () =
 let
   fun sieve (Stream (n,ns)) =
     Stream
      (n,
       fn() => sieve
                (filter (fn x => x mod n<>0) (ns())))
      )
 in
   sieve (generateNat 2)
 end;

- take 10 (all_primes());
val it = [2,3,5,7,11,13,17,19,23,29] : int list
```

# Unendliche Datenstrukturen

```
take 200 (all_primes());
val it =
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,
 103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,
 199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,
 313,317,331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,
 433,439,443,449,457,461,463,467,479,487,491,499,503,509,521,523,541,547,557,
 563,569,571,577,587,593,599,601,607,613,617,619,631,641,643,647,653,659,661,
 673,677,683,691,701,709,719,727,733,739,743,751,757,761,769,773,787,797,809,
 811,821,823,827,829,839,853,857,859,863,877,881,883,887,907,911,919,929,937,
 941,947,953,967,971,977,983,991,997,1009,1013,1019,1021,1031,1033,1039,1049,
 1051,1061,1063,1069,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,
 1163,1171,1181,1187,1193,1201,1213,1217,1223] : int list
```

# Überblick

## 8. Seiteneffekte

- Referenzen
- Sequenzen
- Vektoren
- Arrays
- Ein- und Ausgabe

# Referenzen

- Der postfixierte **Typ-Operator** `ref`

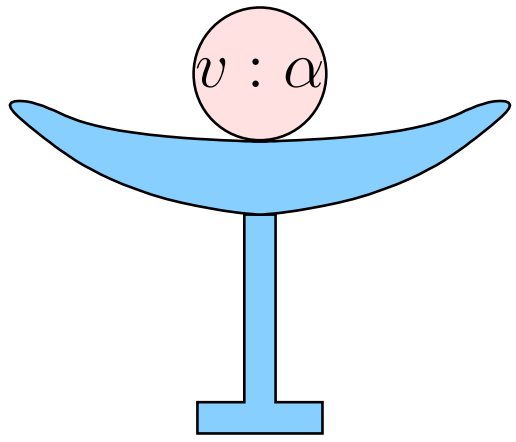
$$\texttt{ref} \; : \; \texttt{MT} \mapsto \texttt{MT}$$

  konstruiert ein Typ $\alpha$ `ref` aus einem Typ $\alpha$.

- Z.B.
  - `int ref`
  - `(int * bool) ref`
  - `int ref ref` $\equiv$ `((int ref) ref)` (links-assoziativ)

# Der `ref`-Konstruktor

- Der einzige **Konstruktor** des Datentyps `ref` heißt auch `ref`.
- Ein Wert vom Typ $\alpha$ `ref` ist ein Behälter ($\equiv$ eine Referenz) für Werte vom Typ $\alpha$:




$: \alpha$ `ref`

```
val p = ref 5;
val p = ref 5 : int ref
```

# Dereferenzierung

```
val p = ref 5;
val p = ref 5 : int ref
```

**Dereferenzierung** = Zugriff auf den Inhalt (Wert) einer Referenz:

- Da `ref` ein Konstruktor ist, kann man auf den Wert einer Referenz via **Pattern Matching** zugreifen:

```
val ref x = p;
val x = 5 : int
```

# Dereferenzierung

```
val p = ref 5;
val p = ref 5 : int ref
```

**Dereferenzierung** = Zugriff auf den Inhalt (Wert) einer Referenz:

- Da `ref` ein Konstruktor ist, kann man auf den Wert einer Referenz via **Pattern Matching** zugreifen:

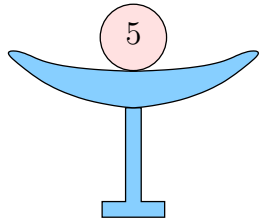
```
val ref x = p;
val x = 5 : int
```

- oder mit dem **Dereferenzierung-Operator "!"**:

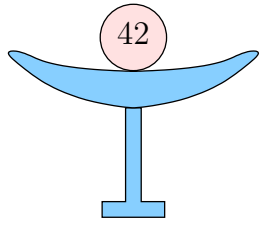
```
val x = !p;
val x = 5 : int
```

# Zuweisungen

- Der Behälter
  ist unveränderbar (wie alle funktionale Werte).




- Der Wert, auf den eine
  Referenz zeigt, kann mit := verändert werden:




```
p := 42;
val it = () : unit

p;
val it = ref 42 : int ref
```

# Seiteneffekte

- Das Setzen von p mittels := ist ein Seiteneffekt und hat keinen Wert, d.h. es ergibt ().

```
p := 42;
val it = () : unit

op :=;
val it = fn : 'a ref * 'a -> unit
```

# Gleichheit von Referenzen

- Zwei Referenzen sind nur dann gleich, wenn sie derselbe Behälter (Zeiger) sind. Es genügt nicht, dass sie den gleichen Wert enthalten:

```
val p = ref 17;
val p = ref 17 : int ref

val q = ref (!p);
val q = ref 17 : int ref

p=q;
val it = false : bool
```

# Gleichheit von Referenzen

```
val x=1;
val x = 1 : int

val (p,q) = (ref x,ref x);
val p = ref 1 : int ref
val q = ref 1 : int ref

p=q;
val it = false : bool
```

# Gleichheit von Referenzen

```
val (p,q) = (ref (ref 1),ref (ref 2));
val p = ref (ref 1) : int ref ref
val q = ref (ref 2) : int ref ref

p := !q;
val it = () : unit

p=q;
val it = false : bool

!p = !q;
val it = true : bool
```

# Inferenz-Regeln für Referenz-Typen

**Ref:**
$$\frac{e : t}{\texttt{ref}\ e : \texttt{ref}\ t}$$

**Deref:**
$$\frac{e : \texttt{ref}\ t}{\texttt{!}e : t}$$

**Assign:**
$$\frac{e_1 : \texttt{ref}\ t \qquad e_2 : t}{e_1\ \texttt{:=}\ e_2 : \texttt{unit}}$$

Diese Regeln vertragen sich aber nicht mit Polymorphie!

# Polymorphische Referenzen

- Obigen Inferenz-Regeln akzeptieren:

```
let
  val id = fn x => x
  val fp = ref id
  val (x,y) = (!fp 1,!fp true)
in
  (x,y)
end
```

... mit dem (*verallgemeinerten*) polymorphischen Typ `∀'a.('a->'a) ref` für `fp` und frischen neuen Variablen `'a1 = int` bzw. `'a2 =bool` bei jeder Anwendung.

# Polymorphische Referenzen

- Die obigen Inferenz-Regeln akzeptieren auch:

```
let
  val id = fn x => x
  val fp = ref id
  val (x,y) = (!fp 1,!fp true)
  val () = fp := !not
  val z = !fp 2
in
  (x,y,z)
end
```

$\Longrightarrow$ würde Typ-Fehler bei der Laufzeit erzeugen!

- Obiger Code sollte äquivalent sein zu:

```
(!(ref (fn x => x)) 1, !(ref (fn x => x)) true,
 !(ref not) 2)
```

welcher korrekterweise nicht getypt werden kann.

- Ist aber nicht äquivalent. Problem:
  - ursprünglich: eine Referenz wird alloziert
  - modifizierte Version: drei Referenzen

# Polymorphische Referenzen

- Lösung = **Value Restriction**: im Scope einer Deklaration `val id = e` wird der Typ von `id` generalisiert (jedes Auftreten eine anderen Instanz), nur wenn `e` ein **syntaktischer Wert** `v` ist:

```
v ::= Konstante | Bezeichner | Konstruktor v | fn x => Ausdruck
```

- $\Longrightarrow$ Folgendes wird abgewiesen:

```
let
  val id = fn x => x
  val fp = ref id
  val (x,y) = (!fp 1,!fp true)
  val () = fp := !not
  val z = !fp 2
in (x,y,z) end
```

- Aus dem selben Grund werden top-level Deklarationen `val id = e` verboten, wenn `e` kein syntaktischer Wert ist.

# Sequenzen

- Beim Arbeiten mit Seiteneffekten ist die Ausführungsreihenfolge wichtig.

- Typische Benutzungen:

| vor Berechnung | nach Berechnung |
|---|---|
| `let val _ = <effect>`<br>`    val x = <value>`<br>`in x`<br>`end` | `let val x = <value>`<br>`    val _ = <effect>`<br>`in x`<br>`end` |

- Abkürzung

| vor Berechnung | nach Berechnung |
|---|---|
| `(<effect>; <value>)` | `<value> before <effect>` |

# Beispiel: Objekte mit Referenzen und Abschlüssen

```
val konto =
  let
    val betrag = ref 100
  in
    {einzahlen = fn b => betrag := !betrag + b,
     auszahlen = fn b => betrag := !betrag -b,
     auszug = fn () => !betrag}
  end;
val konto = auszahlen=fn,auszug=fn,einzahlen=fn
  : {auszahlen:int -> unit, auszug:unit -> int, einzahlen:int -> unit}

#einzahlen konto 500;
val it = () : unit

#auszug konto ();
val it = 600 : int
```

# Beispiel: Objekte mit Referenzen und Abschlüssen

```
val konto =
  let
    val betrag = ref 100
  in
    {einzahlen = fn b => betrag := !betrag + b,
     auszahlen = fn b => betrag := !betrag -b,
     auszug = fn () => !betrag}
  end;
```

- Die Referenz `betrag` ist im `konto` Objekt in den funktionalen Abschlüssen eingekapselt.
- Der Betrag kann nur mit den Methoden `einzahlen` und `auszahlen` manipuliert werden.

# Vektoren

- Der postfixierte **Typ-Operator** `vector` : MT $\mapsto$ MT
- Ein Vektor ist eine Liste fester Länge, auf deren Elemente in konstanter Zeit zugegriffen werden kann:

```
val vec = #[1,3,5,7];
val vec = #[1,3,5,7] : int vector

Vector.sub(vec,3);
val it = 7 : int
```

- Wird außerhalb der Vektorgrenzen zugegriffen, wird die Exception `Subscript` geworfen:

```
Vector.sub(vec,4);
uncaught exception subscript out of bounds raised at: stdIn:1426.1-1426.11
```

# Vektoren

- Ein Vektor kann aus einer Liste oder oder als Wertetabelle für eine Funktion erzeugt werden:

```
Vector.fromList [1,2,3];
val it = #[1,2,3] : int vector

Vector.tabulate (6, fn x => x*x);
val it = #[0,1,4,9,16,25] : int vector
```

- Ähnliche Funktionale wie bei Listen sind vordefiniert (`map`, `foldl`, `foldr`, u.v.m.):

```
Vector.foldri (fn (i,x,xs) => (x+i)::xs) []
              (#[0,1,2,3],1,NONE);
val it = [2,4,6] : int list
```

# Arrays

- Der postfixierte **Typ-Operator** `array` : $\texttt{MT} \mapsto \texttt{MT}$

- Vektoren kann man, wenn sie einmal erzeugt sind, nicht mehr verändern. Dafür muß man **Arrays** verwenden.

```
val arr = Array.fromList [11,12,13];
val arr = [|11,12,13|] : int array
```

- Arrays kann man im Gegensatz zu Vektoren nicht direkt hinschreiben:

```
[|11,12,13|];
stdIn:1433.2-1433.5 Error: syntax error: deleting BAR INT COMMA
```

- Ähnlich wie bei Vektoren kann auf Elemente eines Arrays mit Hilfe von `Array.sub` zugreifen:

```
Array.sub(arr,2);
val it = 13 : int
```

# Arrays

- Zur Modifizierung der Array-Einträge benutzt man die Funktion `Array.update`:

```
(Array.update(arr,1,4); arr);
val it = [|11,4,13|] : int array

Array.update(arr,5,4);
uncaught exception subscript out of bounds raised at: stdIn:1.1-1364.2
```

- Wenn man ein Array nicht mehr verändern will, kann man es in einen Vektor transformieren:

```
Array.extract(arr,0,SOME 2);
val it = #[11,4] : int vector

Array.extract(arr,0,NONE);
val it = #[11,4,13] : int vector
```

# Ein- und Ausgabe

- Die einfachste Funktion zur Bildschirmausgabe ist

  `print: string -> unit`

```
print "Palim-palim world!\n";
Palim-palim world!
val it = () : unit
```

- Will man einen Wert ausgeben, so muß man diesen zunächst in den `string`-Typ konvertieren. Für die Basis-Typen bietet SML vordefinierte Funktionen an, z.B.

  `Int.toString: int -> string`

```
print (Int.toString (7*6)^"\n");
42
val it = () : unit
```

# Ein- und Ausgabe

- Will man strukturierte Daten ausgeben, muß man selbst entsprechende Funktionen definieren.

- Erste Möglichkeit: **toString-Funktion**

```
datatype 'a Tree = Leaf of 'a
                 | Node of 'a Tree * 'a * 'a Tree

fun Tree2String a2String t =
 case t of
    Leaf a => "Leaf "^a2String a
  | Node (l,a,r) =>
          "Node("^(Tree2String a2String l)^ ","^
                  (a2String a)^",",
                  (Tree2String a2String r)^")"
val Tree2String = fn : ('a -> string) -> 'a Tree -> string

fun printTree a2String =
  print o (Tree2String a2String)
val printTree = fn : ('a -> string) -> 'a Tree -> unit
```

# Ein- und Ausgabe

- Zweite Möglichkeit: direkte Ausgabe auf dem Bildschirm

```
fun printTree printA t =
 case t of Leaf a => (print "Leaf "; printA a)
         | Node(l,a,r) =>
                    (print "Node(";
                     printTree printA l;
                     print ",";
                     printA a;
                     print ",";
                     printTree printA r;
                     print ")")
val printTree = fn : ('a -> unit) -> 'a Tree -> unit
```

# Ein- und Ausgabe aus Dateien

- Zur (Text-)Ein- und Ausgabe aus Dateien stellt das Modul `TextIO` eine Kollektion von Typen und Funktionen zur Verfügung:

```
type elem = char
type vector = string
type instream
type outstream

val stdIn : instream
val stdOut : outstream
val stdErr : outstream
```

# Einlesen aus Text-Dateien

- Der Typ `instream` repräsentiert Dateien, aus denen man nur lesen kann. Als Sonderfall kann man auch einen String zum Lesen öffnen.

```
val openIn : string -> instream
val openString : string -> instream
val closeIn : instream -> unit
val input1 : instream -> elem option
val inputN : instream * int -> vector
val endOfStream : instream -> bool
```

# Schreiben in Text-Dateien

- Ein `outstream` dagegen dient zum Schreiben:

```
val openOut : string -> outstream
val openAppend : string -> outstream
val closeOut : outstream -> unit
val output : outstream * vector -> unit
val output1 : outstream * elem -> unit
```

# Überblick

## 9. Kontrollfluß-Manipulieren

Ausnahmen

Continuations

# Ausnahmen

- **Ausnahmen** (*exceptions*) sind Werte eines **vordefinierten Typs** `exn` $\in MT$.
- Konstruktoren für die Werte des Typen `exn` können vom Benutzer definiert werden:

  `exception` *AusnahmeKonstruktor* `[of` *Typ*`]`

- Beispiel:

```
exception LeereListe
exception BannedWords of string list
```

- Dadurch wurde der exn-Datentyp um zwei Ausnahmen-Konstruktoren erweitert. Das geht mit keinem anderen Datentyp!

# Vordefinierte Ausnahmekonstruktoren

| | |
|---|---|
| **Div** | bei Division durch Null |
| **Empty** | bei Zugriff auf eine leere Liste (`hd []`) |
| **Match** | bei unvollständigem Match in einem case-Ausdruck oder im Funktionskopf |
| **Fail** `of string` | ohne bestimmte Bedeutung, zur Benutzung durch den Programmierer |

```
- 1 div 0;
uncaught exception divide by zero raised at: <stdin>

- tl (tl [1]);
uncaught exception Empty raised at: boot/list.sml:37.38-37.43
```

# Der Typ `exn`

- Der Typ `exn` ist das selbe unabhängig vom Typ des eingebetteten Wertes.

```
LeereListe;
val it = LeereListe(-) : exn

- BannedWords ["viagra","rolex","medication"];
val it = BannedWords(-) : exn
```

$\Longrightarrow$ `exn` ist kein polymorpher Typ.

# Ausnahmenverarbeitung

- **Ausnahmen Werfen:**
  - Eine Ausnahme `ex` kann bei der Laufzeit während einer Ausdrucksauswertung *geworfen* werden.
  - `ex` reist in die *Vergangenheit* zu den noch nicht zu Ende ausgewerteten Ausdrucksauswertungen.

- **Ausnahmen Behandeln:**
  - Eine Ausnahme `ex`, die in die Vergangenheit reist, kann abgefangen (gehandlet) werden.

# Ausnahmen Werfen

**raise : exn -> 'a**

- kann an einer beliebigen Stelle in einem Ausdruck $E$ vorkommen
- liefert nichts zurück
- simuliert nur einen Rückgabewert für den Wert von $E$
  $\Longrightarrow$ der virtuelle Rückgabewert hat den Typ von $E$
  $\Longrightarrow$ der Rückgabetyp von `raise` muss polymorph sein

```
1 + (raise Div);
uncaught exception divide by zero raised at: stdIn:363.12-363.15

1::(raise Div);
uncaught exception divide by zero raised at: stdIn:263.1-263.4
```

# Ausnahmen Behandeln

$$
\begin{array}{rcl}
E \texttt{ handle } P_1 & \texttt{=>} & E_1 \\
\mid P_2 & \texttt{=>} & E_2 \\
\ldots & \texttt{=>} & \\
\mid P_n & \texttt{=>} & E_n
\end{array}
$$

- $P_1$, $P_2$, ..., $P_n$ **sind Muster** ähnlich wie bei einem case Ausdruck.
- Terminiert das Auswerten von $E$ normal, wird dessen Wert geliefert.
- Wirft das Auswerten von $E$ eine Ausnahme `ex`, liefert der Ausdruck den Wert von $E_i$, wenn $P_i$ das erste passende Muster ist.
  $\Longrightarrow$ $E_i$ müssen den selben Typ wie $E$ haben
- Sonst wird `ex` weitergeworfen und evt. bei der Auswertung eines umgebenden Ausdrucks (insbesondere Funktionsaufrufs) behandelt
- Das Laufzeitsystem behandelt nicht gefangene Ausnahmen.

# Ausnahmen Verwenden

Ausnahmen können verwendet werden:

- zur **Fehlerbehandlung**
- als *lange Sprünge* (*longjumps*)
- als **Berechnungsmechanism**

# Ausnahmen zur Fehlerbehandlung

```
fun head l = case l of nil => raise Empty | h::_ => h
fun tail l = case l of nil => raise Empty | _::r => r

fun member x l = if x=head l then true
                 else member x (tail l)
                 handle Empty => false


member 2 [1,2,3];
val it = true : bool

member 4 [1,2,3];
val it = false : bool
```

# Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```

# Ausnahmen als Longjumps

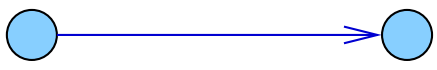
```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```

member $x$ $[a, b, c]$

# Ausnahmen als Longjumps

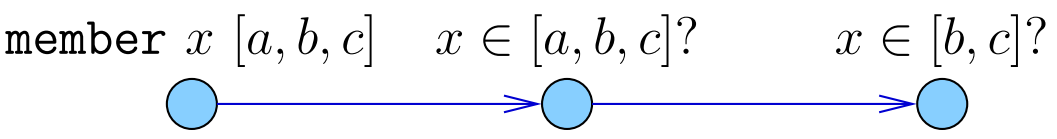
```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```

member $x$ $[a, b, c]$ $x \in [a, b, c]$?



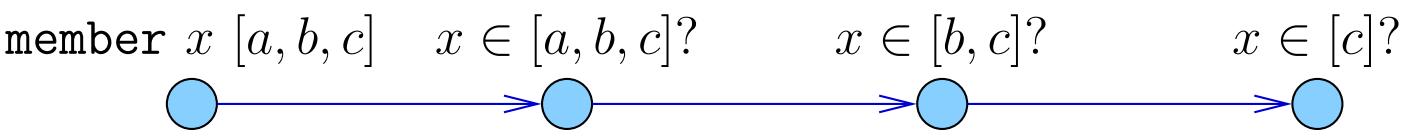
# Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



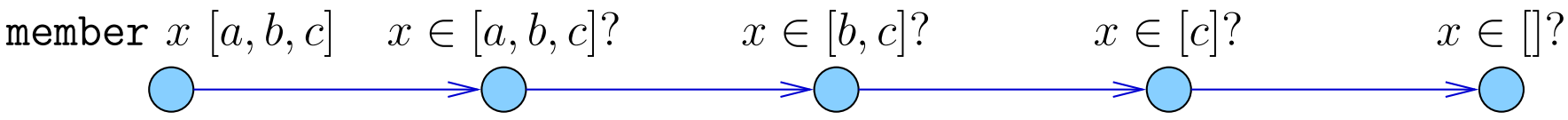
# Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



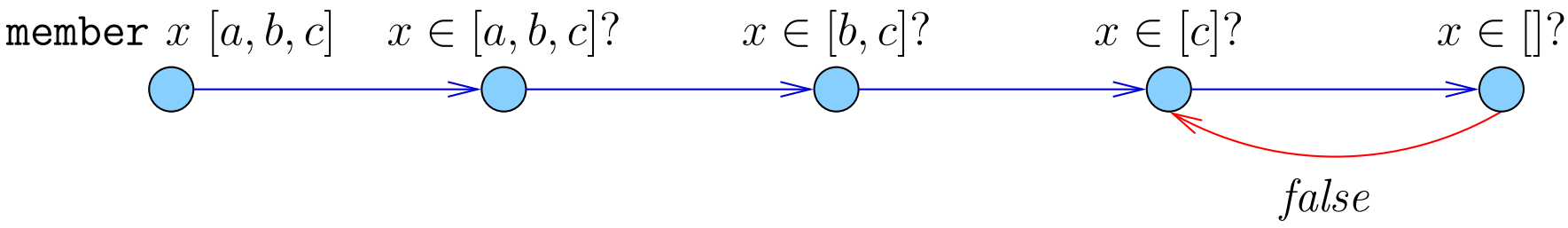
# Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



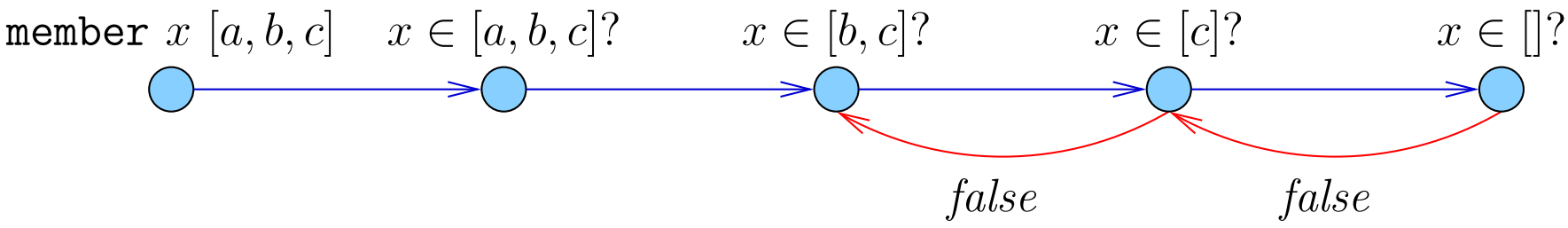
# Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



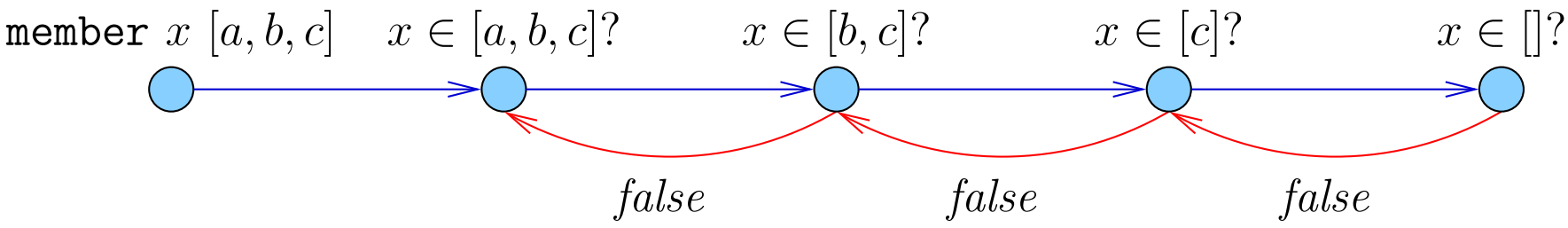
# Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



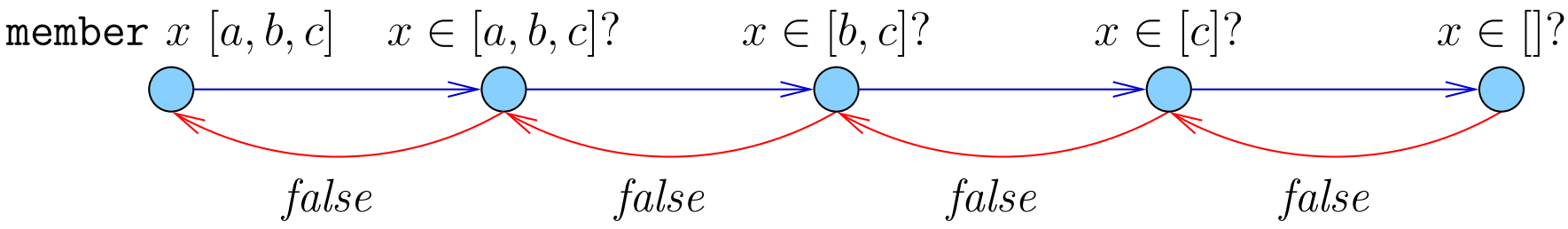
# Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



# Ausnahmen als Longjumps

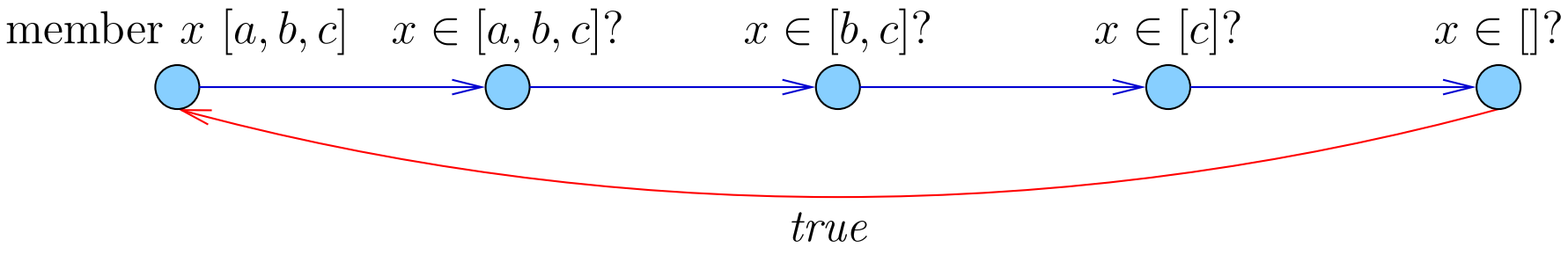
```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



# Ausnahmen als Longjumps

```
exception Result of bool

fun member x list =
  let fun search l =
        case l of nil => raise (Result false)
                | h::r => if h=x then raise (Result true)
                          else search r
  in search list handle (Result r) => r
  end
```



# Ausnahmen als Berechnungsmechanismus

## Inhomogene Listen

```
datatype 'a List = Nil | Atom of 'a
                 | List of 'a List  * 'a List

val l = List(List(Atom 1,Atom 2),Atom 3)
```

# Ausnahmen als Berechnungsmechanismus

## Inhomogene Listen

```
datatype 'a List = Nil | Atom of 'a
                 | List of 'a List  * 'a List

val l = List(List(Atom 1,Atom 2),Atom 3)
```

```
exception OnEmpty
exception OnAtom
fun first l = case l of Nil => raise OnEmpty
                      | Atom _ => raise OnAtom
                      | List (first,_) => first
first l;
val it = List (Atom 1,Atom 2) : int List

first (first (first l));
uncaught exception OnAtom raised at: stdIn:412.64-412.70
```

# Ausnahmen als Berechnungsmechanismus

```
fun rest l = case l of Nil => raise OnEmpty
                     | Atom _ => raise OnAtom
                     | List (_,rest) => rest

fun atoms l =
 ((atoms (first l) handle OnEmpty => 0 | OnAtom => 1) +
  (atoms (rest l) handle OnEmpty => 0 | OnAtom => 1));

atoms (Atom 1);
val it = 2 : int

fun countAtoms l = (atoms l) div 2
countAtoms (Atom 1);
val it = 1 : int

countAtoms (List(List(Atom 1,Atom 2),Atom 3));
val it = 3 : int
```

# Continuations

# Rekursionsarten

Je nach Art der rekursiven Aufrufe unterscheiden wir folgende Arten von Rekursion:

- **End-Rekursion** (lineare Rekursion):
  - Bei der Funktionsauswertung gibt es nur einen rekursiven Aufruf.
  - Dieser ist gleichzeitig der Rückgabewert.

```
fun fac1 (n,acc) = if n=1 then acc
                   else fac1(n-1,n*acc)

fun loop x = if x<2 then x
             else if x mod 2 = 0 then loop(x div 2)
                  else loop(3*x+1)
```

# Rekursionsarten

- **Repetitive Rekursion:** nur einen rekursiven Aufruf, der aber nicht der Rückgabewert ist.

```
fun fac n = if n=1 then 1 else n*(fac(n-1))
```

- **Baumartige (kaskadenartige) Rekursion:** mehrere, nicht verschachtelte rekursive Aufrufe.

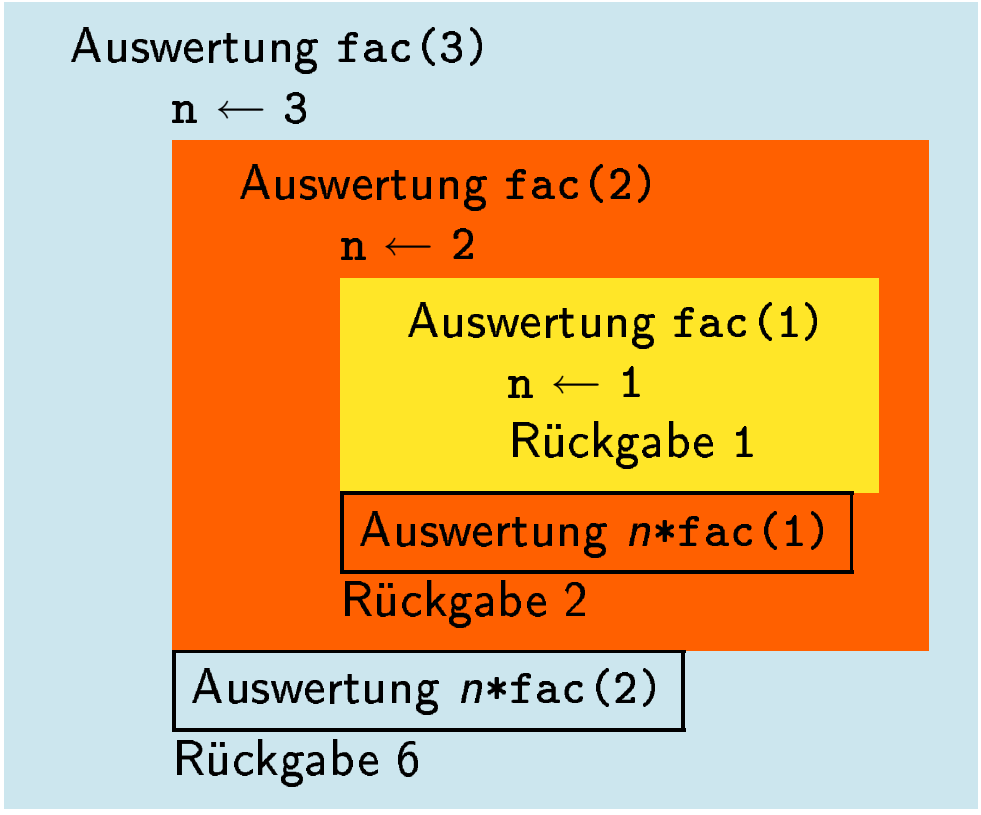
```
fun fib n = if n=0 then 1
            else if n=2 then 1
                 else fib(n-1)+fib(n-2)
```

- **Wilde Rekursion:** verschachtelte rekursive Aufrufe

```
fun f n = if n<2 then n else f(f(n div 2))
```

# Speicherverhalten der repetitiv-rekursiven Fkt.

```
fun fac n = if n=1 then 1 else n*(fac(n-1))
```

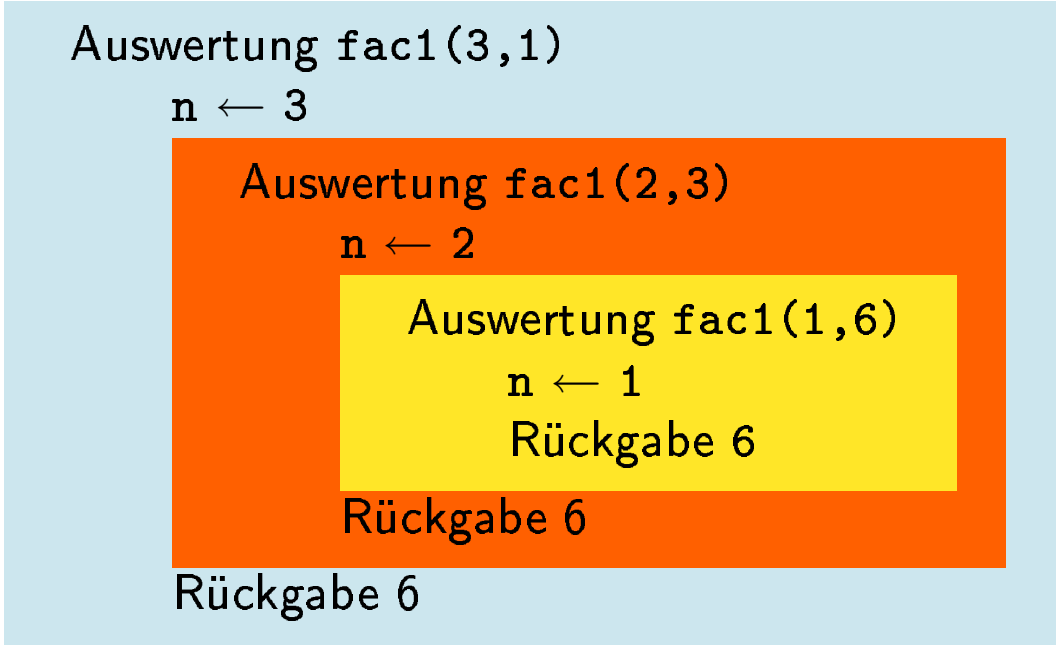



Bei jedem neuen Funktionsaufruf muss der aktuelle Aufruf seine lokale Variablen speichern, um diese nach dem Aufruf verwenden zu können.

$\Longrightarrow$ Speicherverbrauch wächst mit Anzahl geschachtelter Aufrufe.

# Speicherverhalten der tail-rekursiven Funktionen

```
fun fac1 (n,acc) = if n=1 then acc else fac1(n-1,n*acc)
```




Rekursiver Aufruf = Rückgabewert
$\Longrightarrow$ Keine Berechnung nach dem Aufruf
$\Longrightarrow$ Lokale Variablen werden nicht mehr gebraucht
$\Longrightarrow$ müssen nicht gespeichert werden

$\Longrightarrow$ Tail-Rekursion hat das beste Speicherverhalten. Es wird zur wilden Rekursion hin immer schlechter.

# Eliminierung der repetitiven Rekursion

- Typische Erscheinung **repetitiver Rekursion**:

I.A.: `fun f x = if x = <`$x_0$`> then <`$v_0$`> else <e(x,f(g(x)))>`

Bsp.: `fun fac x = if x = 1 then 1 else x*(fac(x-1))`

$\Longrightarrow$ $x_0$=1, $v_0$=1, g(x)=x-1 und e(x,y)=x*y

# Eliminierung der repetitiven Rekursion

- Typische Erscheinung **repetitiver Rekursion**:

I.A.: `fun f x = if x = <`$x_0$`> then <`$v_0$`> else <e(x,f(g(x)))>`

Bsp.: `fun fac x = if x = 1 then 1 else x*(fac(x-1))`

$\Longrightarrow$ $x_0$=1, $v_0$=1, g(x)=x-1 und e(x,y)=x*y

- **End-rekursive Variante**: mit Zusatz-Param. (Akkumulator)

Bsp.: `fun fac1 (x,a) = if x = 1 then a else fac1(x-1,x*a)`

$\Longrightarrow$ Aufruf mit fac1 (x,1).

I.A.: `fun f1 (x,a) = if x = <`$x_0$`> then a else f1(g(x),e(x,a))`

$\Longrightarrow$ Aufruf mit f1 (x,$v_0$).

# Eliminierung der repetitiven Rekursion

```
fun f x       = if x = <x0> then <v0> else <e(x,f(g(x)))>
fun f1 (x,a) = if x = <x0> then  a  else f1(g(x),e(x,a))
```

$\Longrightarrow$ Aufruf mit f1 (x, $v_0$).

$$
\begin{aligned}
f1(x, v_0) &= f1(g(x), e(x, v_0)) = f1(g(g(x)), e(g(x), e(x, v_0))) = \ldots \\
&= e(g^n(x), e(g^{n-1}(x), \ldots e(g(x), e(x, v_0)) \ldots)) \\
&= g^n(x) \,\square_e\, (g^{n-1}(x) \,\square_e \ldots (g(x) \,\square_e\, (x \,\square_e\, v_0)) \ldots)
\end{aligned}
$$

# Eliminierung der repetitiven Rekursion

```
fun f x       = if x = <x0> then <v0> else <e(x,f(g(x)))>
fun f1 (x,a) = if x = <x0> then  a  else f1(g(x),e(x,a))
```

$\Longrightarrow$ Aufruf mit f1 (x, $v_0$).

$$
\begin{aligned}
f1(x, v_0) &= f1(g(x), e(x, v_0)) = f1(g(g(x)), e(g(x), e(x, v_0))) = \ldots \\
&= e(g^n(x), e(g^{n-1}(x), \ldots e(g(x), e(x, v_0)) \ldots)) \\
&= g^n(x) \,\square_e\, (g^{n-1}(x) \,\square_e \ldots (g(x) \,\square_e\, (x \,\square_e\, v_0)) \ldots)
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= e(x, e(g(x), e(g(g(x), \ldots e(g^n(x), v_0) \ldots))) \\
&= x \,\square_e\, (g(x) \,\square_e\, (g^2(x) \,\square_e \ldots (g^n(x) \,\square_e\, v_0) \ldots))
\end{aligned}
$$

# Eliminierung der repetitiven Rekursion

```
fun f x       = if x = <x0> then <v0> else <e(x,f(g(x)))>
fun f1 (x,a) = if x = <x0> then  a  else f1(g(x),e(x,a))
```

$\Longrightarrow$ Aufruf mit f1 (x, $v_0$).

$$
\begin{aligned}
f1(x, v_0) &= f1(g(x), e(x, v_0)) = f1(g(g(x)), e(g(x), e(x, v_0))) = \ldots \\
&= e(g^n(x), e(g^{n-1}(x), \ldots e(g(x), e(x, v_0)) \ldots)) \\
&= g^n(x) \,\square_e\, (g^{n-1}(x) \,\square_e \ldots (g(x) \,\square_e\, (x \,\square_e\, v_0)) \ldots)
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= e(x, e(g(x), e(g(g(x), \ldots e(g^n(x), v_0) \ldots))) \\
&= x \,\square_e\, (g(x) \,\square_e\, (g^2(x) \,\square_e \ldots (g^n(x) \,\square_e\, v_0) \ldots))
\end{aligned}
$$

$\Longrightarrow$ $f(x) = f1(x, v_0)$, wenn $\square_e$ = assoziativ, kommutativ, z.B.:

```
n*((n-1)*(...*(2*1))) = 1*(2*(...*((n-1)*n)))
```

# Rekursionsarten

- Vorsicht bei Listenfunktionen:

```
fun f  x      = if x=0 then nil else x::f(x-1)
fun f1 (x,a) = if x=0 then a   else f1(x-1,x::a)
```

- Der Listenkonstruktor :: ist weder kommutativ noch assoziativ.
  $\Longrightarrow$`f x` und `f1 (x,nil)` berechnen nicht den gleichen Wert:

```
- f 5;
val it = [5,4,3,2,1] : int list


- f1(5,nil);
val it = [1,2,3,4,5] : int list
```

# Rekursionsarten

- Selbst baumartige Rekursion kann manchmal linearisiert werden:

```
fun fib n = case n of 0 => 0
                    | 1 => 1
                    | n => fib (n-1) + fib (n-2)

fun fib1 n =
    let fun iter (m,f1,f2) =
            if m = n then f1 else iter(m+1,f2,f1+f2)
    in iter(0,0,1)
    end
```

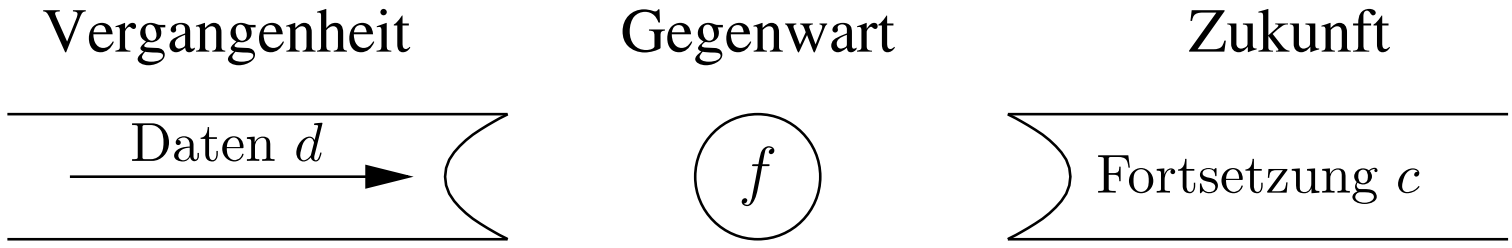
- Das läßt sich aber nicht so einfach verallgemeinern.

# Continuation-passing style

## Continuation-passing style (CPS)

(≈ *Fortsetzungen-Durchreichen-Programmierstil*):

Abstraktion einer Berechnung:

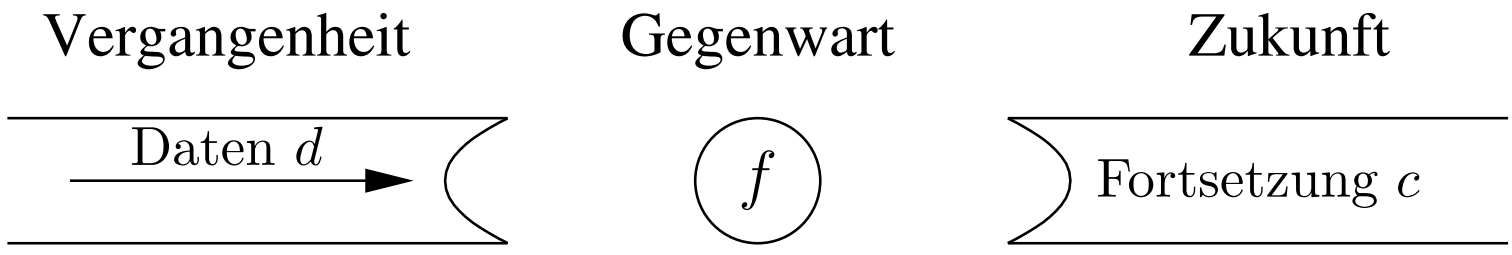


# Continuation-passing style

## Continuation-passing style (CPS)

(≈ *Fortsetzungen-Durchreichen-Programmierstil*):

Abstraktion einer Berechnung:




*Vergangenheit* = Daten $d$, die der aktuellen Bearbeitungsfunktion übergeben werden

*Gegenwart* = Eine *aktuelle* Bearbeitungsfunktion $f$

*Zukunft* = Eine *Fortsetzungsfunktion* $c$, die den Rest der Berechnung beschreibt (**continuation**)

Die Berechnung liefert $c(f(d))$.

# Continuation-passing Style

- Eine Funktion $f$ im CPS:
  - statt einen Wert $v$ zurückzuliefern...
  - .. ruft eine Continuation $k$ mit $v$ auf ($k$ = explizite Angabe, wie die Berechnung fortgesetzt werden soll);

  $\Longrightarrow$ $f$ bekommt die Continuation(s) als zusätzliche Parameter.

# Continuation-passing Style

- Eine Funktion $f$ im CPS:
  - statt einen Wert $v$ zurückzuliefern...
  - .. ruft eine Continuation $k$ mit $v$ auf ($k$ = explizite Angabe, wie die Berechnung fortgesetzt werden soll);

  $\Longrightarrow$ $f$ bekommt die Continuation(s) als zusätzliche Parameter.

- Berechnung von $a + b * c$ **ohne Continuations**:

```
fun add (x,y) = x + y
fun mult (x,y) = x * y
fun compute (a,b,c) = add(a,mult(b,c))
```

# Continuation-passing Style

- Eine Funktion $f$ im CPS:
  - statt einen Wert $v$ zurückzuliefern...
  - .. ruft eine Continuation $k$ mit $v$ auf ($k$ = explizite Angabe, wie die Berechnung fortgesetzt werden soll);

  $\Longrightarrow$ $f$ bekommt die Continuation(s) als zusätzliche Parameter.

- Berechnung von $a + b * c$ **ohne Continuations**:

```
fun add (x,y) = x + y
fun mult (x,y) = x * y
fun compute (a,b,c) = add(a,mult(b,c))
```

- Berechnung von $a + b * c$ **im CPS**:

```
fun add1 ((x,y),k) = k (add (x,y))
fun mult1((x,y),k) = k (mult (x,y))
fun compute ((a,b,c),k) =
                    mult1((b,c),fn x => add1((a,x),k))
```

# Continuation Passing Style: Beispiel

- Berechnung von $a * b + c * d$ mit CPS:

```
fun add1 ((x,y),k) = k (add (x,y))
val add1 = fn : (int * int) * (int -> 'a) -> 'a
fun mult1((x,y),k) = k (mult (x,y))
val mult1 = fn : (int * int) * (int -> 'a) -> 'a

fun compute ((a,b,c,d),k) =
   mult1((a,b),
         fn x => mult1((c,d),
                       fn y => add1((x,y),k)))
val compute = fn : (int * int * int * int) * (int -> 'a) -> 'a
```

# Continuation Passing Style: Beispiel

- Berechnung von $a * b + c * d$ mit CPS:

```
fun add1 ((x,y),k) = k (add (x,y))
val add1 = fn : (int * int) * (int -> 'a) -> 'a
fun mult1((x,y),k) = k (mult (x,y))
val mult1 = fn : (int * int) * (int -> 'a) -> 'a

fun compute ((a,b,c,d),k) =
   mult1((a,b),
         fn x => mult1((c,d),
                       fn y => add1((x,y),k)))
val compute = fn : (int * int * int * int) * (int -> 'a) -> 'a

compute ((1,2,3,4),fn x => x);
val it = 14 : int
```

# Fakultät mit CPS

- Ohne CPS:

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

- Im CPS:

```
fun fac1 (n,k) = if n <= 1 then k 1
                 else fac1 (n-1,fn x => k (n*x))
val fac1 = fn : int * (int -> 'a) -> 'a

fac1 (3,fn x => x);
val it = 6 : int
```

# Vorteile der CPS

- **Optimierung des Speicher-Verhaltens**: Jeder Aufruf, insbesondere end-rekursive Aufrufe, liefern stets den Wert der aufrufenden Funktion:
  - Transformation der rekursiven Funktionen in end-rekursive Funktionen
  - Compiler-Optimierungen: Der SML-Compiler benutzt CPS als Zwischendarstellung für Compilierung und Optimierungen
    $\Longrightarrow$ unnötige Rücksprünge werden eliminiert
- **Erhöhung der Ausdrucksstärke**: Explizitheit des Kontrollflusses
  - benutzer-definierte Kontrollstrukturen durch explizite Modellierung des Kontrollflusses: z.B. Threads, Korutinen, Ausnahmen

# Implizite Continuations

- Soweit haben wir Continuations explizit konstruiert und durchgereicht.
- Jede Berechnung eines Ausdruckes in SML hat eine implizite Continuation:

  **die aktuelle Continuation** (*current continuation* = **cc**)

  Dies ist die Funktion, die die Zukunft der Auswertung dieses Ausdruckes repräsentiert, d.h. eine Abstraktion dessen, was das System mit dem Wert des Ausdruckes machen wird.

# Implizite Continuations

Jede Ausdrucksauswertung eines Programms hat eine **cc**:

- Bsp.: Ausdruck `1 + 2 * 3`

- Implizite Continuations für jeden Teilausdruck:

| Ausdruck | Implizite Continuation |
|---|---|
| `1 + 2 * 3` | `k` (abhängig vom Kontext der Auswertung) |
| `1` | `fn x => k (x + 2 * 3)` |
| `2` | `fn x => k (1 + x * 3)` |
| `3` | `fn x => k (1 + 2 * x)` |
| `2 * 3` | `fn x => k (1 + x)` |

# Implizite Continuations

Sei: `fun len l = case l of [] => 0 | h::r => 1 + len r`

| Ausdruck | Implizite Continuation |
|---|---|
| `len [1,2]` | k (abhängig vom Kontext der Auswertung) |
| `case [1,2] ...` | k |
| `0` | k |
| `1 + len [2]` | k |
| `1` | `fn x => k (x + len [2])` |
| `len [2]` | `fn x => k (1 + x)` |
| `[2]` | `fn x => k (1 + len x)` |

# Implizite Continuations verwenden

Aktuelle **Continuations als first class value** (SML/NJ, Scheme, Python):

- Man kann auf implizite Continuations zugreifen und sie als "first class value" manipulieren.
- Insbesondere kann man **eine Continuation $c$ aktivieren** = Abbruch des normalen Programmablaufs; Fortsetzung mit $c$.

# Implizite Continuations verwenden

Das Modul `SMLofNJ.Cont` stellt einen Typ und Operationen für Continuations zur Verfügung:

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
```

- Werte vom Typ `'a cont` repräsentieren Fortsetzungen von Berechnungen, die Werte vom Typ `'a` zurückliefern.
- Die aktuelle Continuation wird mit Hilfe von callcc (*call with current continuation*) explizit verfügbar.
- Eine Continuation kann mit Hilfe von throw aktiviert werden.

# Implizite Continuations verwenden

- `callcc (fn k => e)`
  - macht die Fortsetzung der Berechnung, die `e` benutzt (die aktuelle Continuation), innerhalb des Ausdrucks `e` selbst verfügbar.

- `throw k a`
  - aktiviert die Continuation `k` mit dem Wert `a`

# Implizite Continuations verwenden

```
1 + callcc (fn k => e)
```

Falls die Auswertung von e:

- k **nicht aktiviert** $\Longrightarrow$
  - callcc (fn k => e) liefert den Wert von e;
  - 1 + der Wert von e wird zurückgeliefert.

# Implizite Continuations verwenden

```
1 + callcc (fn k => e)
```

Falls die Auswertung von e:

- k **nicht aktiviert** $\Longrightarrow$
  - callcc (fn k => e) liefert den Wert von e;
  - 1 + der Wert von e wird zurückgeliefert.
- k **aktiviert** (via throw k e') $\Longrightarrow$
  - Berechnung wird fortgesetzt als hätte callcc (fn k => e) den Wert von e' zurückgeliefert;
  - 1 + den Wert von e' zurückgeliefert.

# Continuations

- Bsp.:

```
1 + callcc (fn k => 2);
val it = 3 : int
```

```
1 + callcc (fn k => throw k 3);
val it = 4 : int
```

# Continuations und Effizienz: Bsp.

```
fun plist l = case l of nil => 1
                      | 0::_ => 0
                      | h::r => h * (plist r)
```

plist [1,2,3,4,5,0,6,7,8] braucht:

- 5 Multiplikationen: 1*(2*(3*(4*(5*0))));
- 5 rekursive Aufrufe; 5 Rücksprünge.

# Continuations und Effizienz: Bsp.

```
fun plist l = case l of nil => 1
                      | 0::_ => 0
                      | h::r => h * (plist r)
```

plist [1,2,3,4,5,0,6,7,8] braucht:

- 5 Multiplikationen: 1*(2*(3*(4*(5*0))));
- 5 rekursive Aufrufe; 5 Rücksprünge.

- Besser: mit Continuations

```
fun plist1 l = callcc (fn k =>
  let fun p l = case l of nil => 1
                        | 0::_ => throw k 0
                        | h::r => h * (p r)
  in
    p l
  end)
```

plist1 [1,2,3,4,5,0,6,7,8] braucht:

- keine Multiplikation;
- 5 rekursive Aufrufe; kein Rücksprung: **liefert direkt** 0.

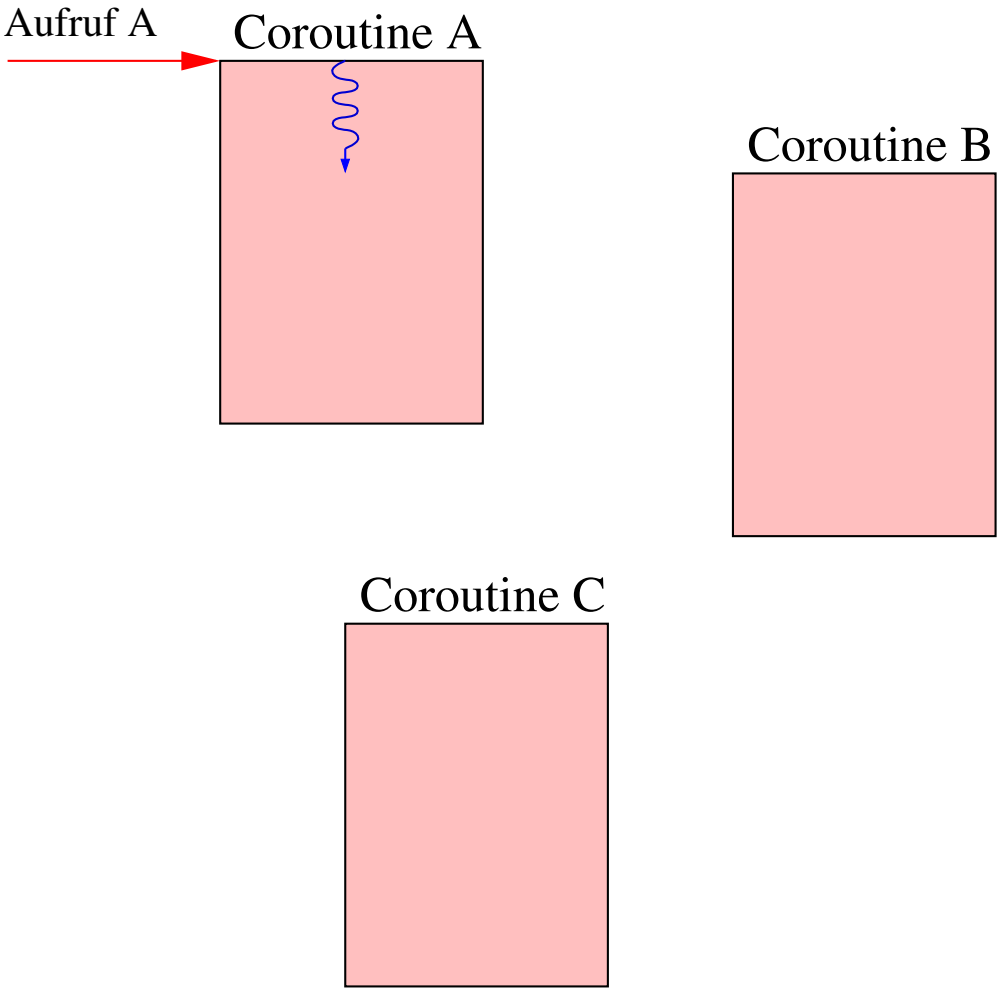
# Continuations-Anwendung: Coroutinen

- **Coroutine** = Funktion, die, nach dem sie einen Wert zurückliefern, in dem zuletzt verlassenen Zustand fortgesetzt werden kann.
  - Erster Startpunkt einer Coroutine = Eintrittspunkt der Coroutine
  - Nach einer Rückgabe setzt die Berechnung bei einem neuen Aufruf nach dem Rückgabepunkt fort.

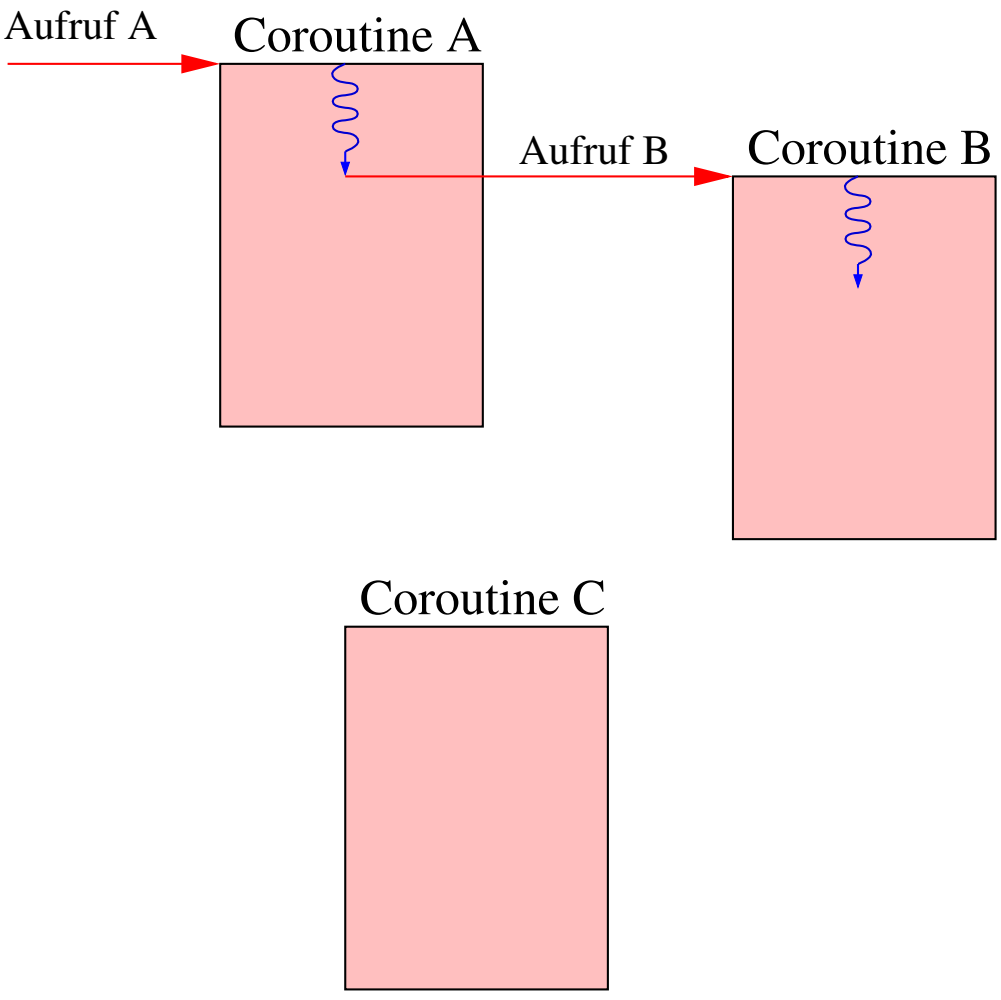
$\Longrightarrow$ Coroutinen = Verallgemeinerung von Funktionen:

  - mehrere Eingangspunkte;
  - mehrere Rückgaben aus einer einzigen Funktionsinstanz.

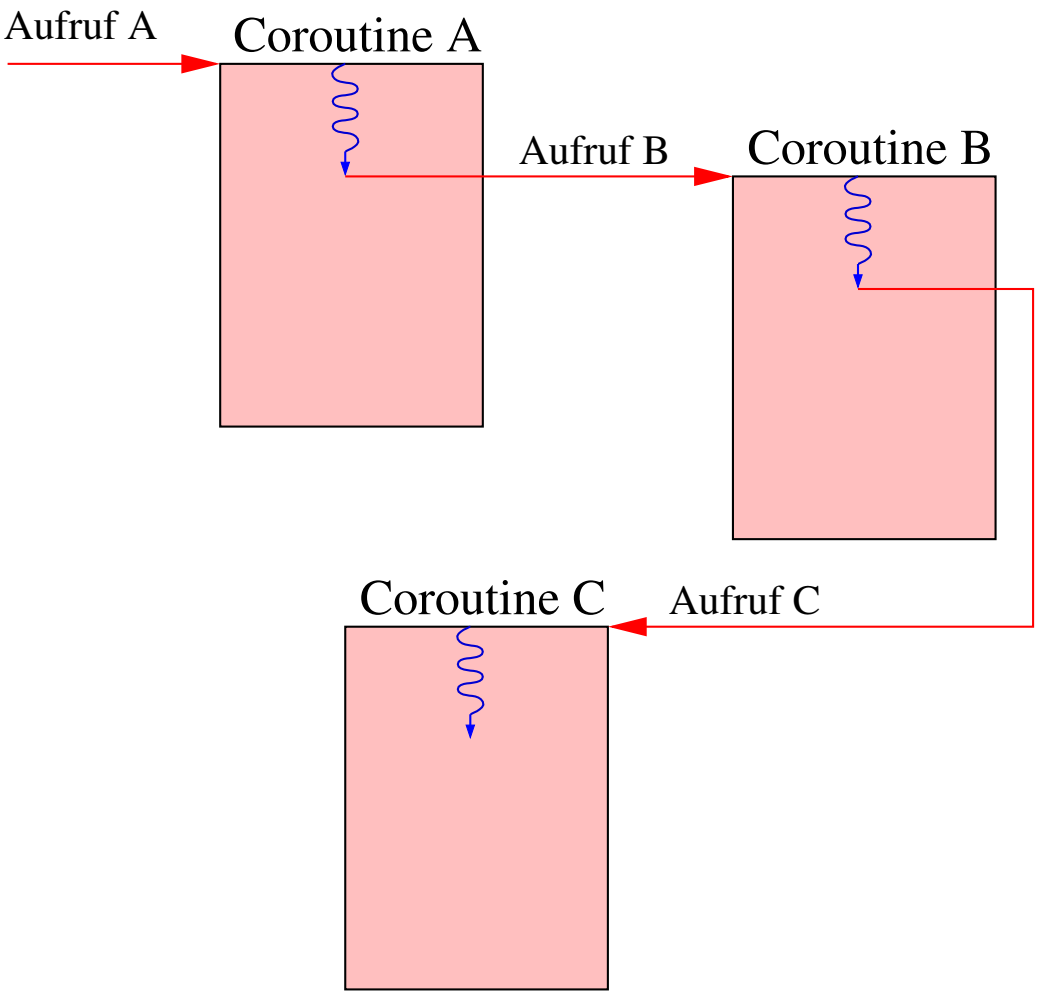
# Coroutinen



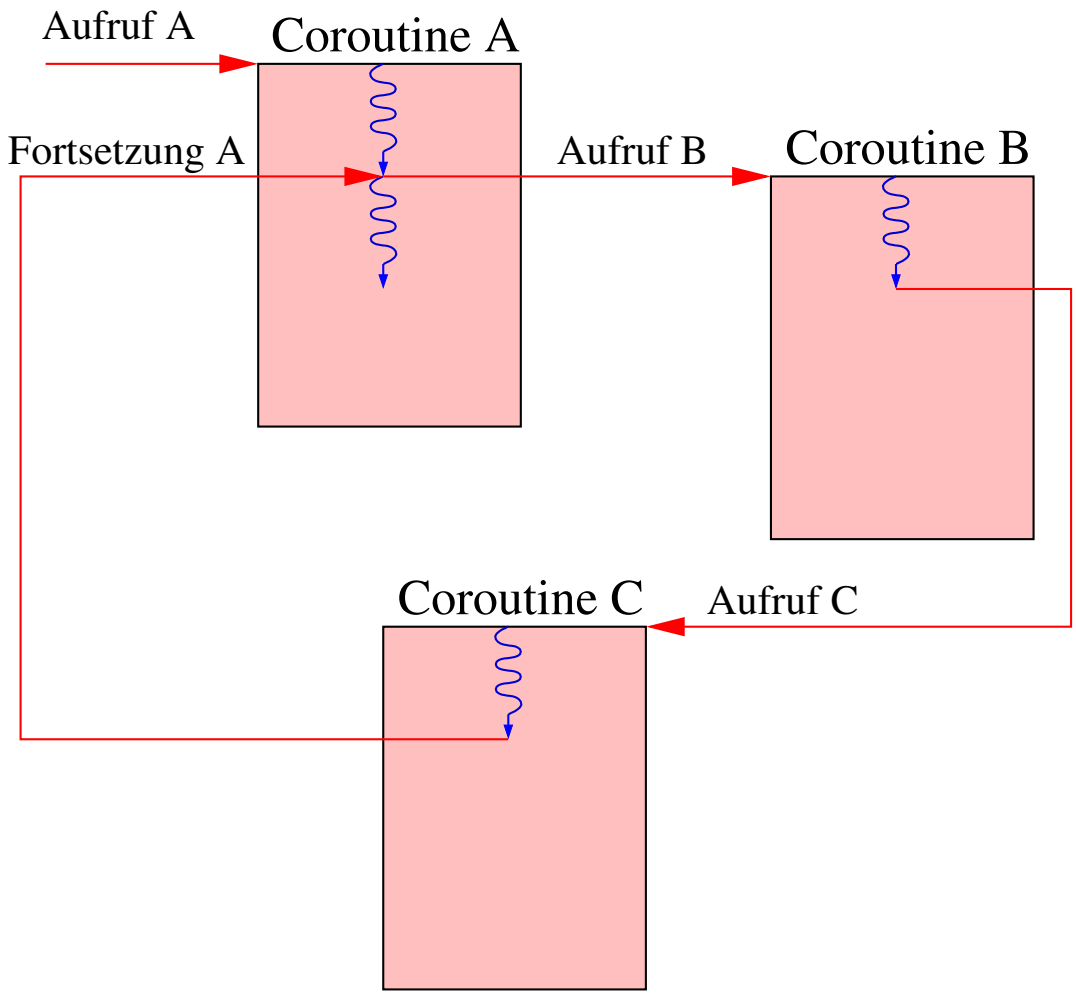
# Coroutinen



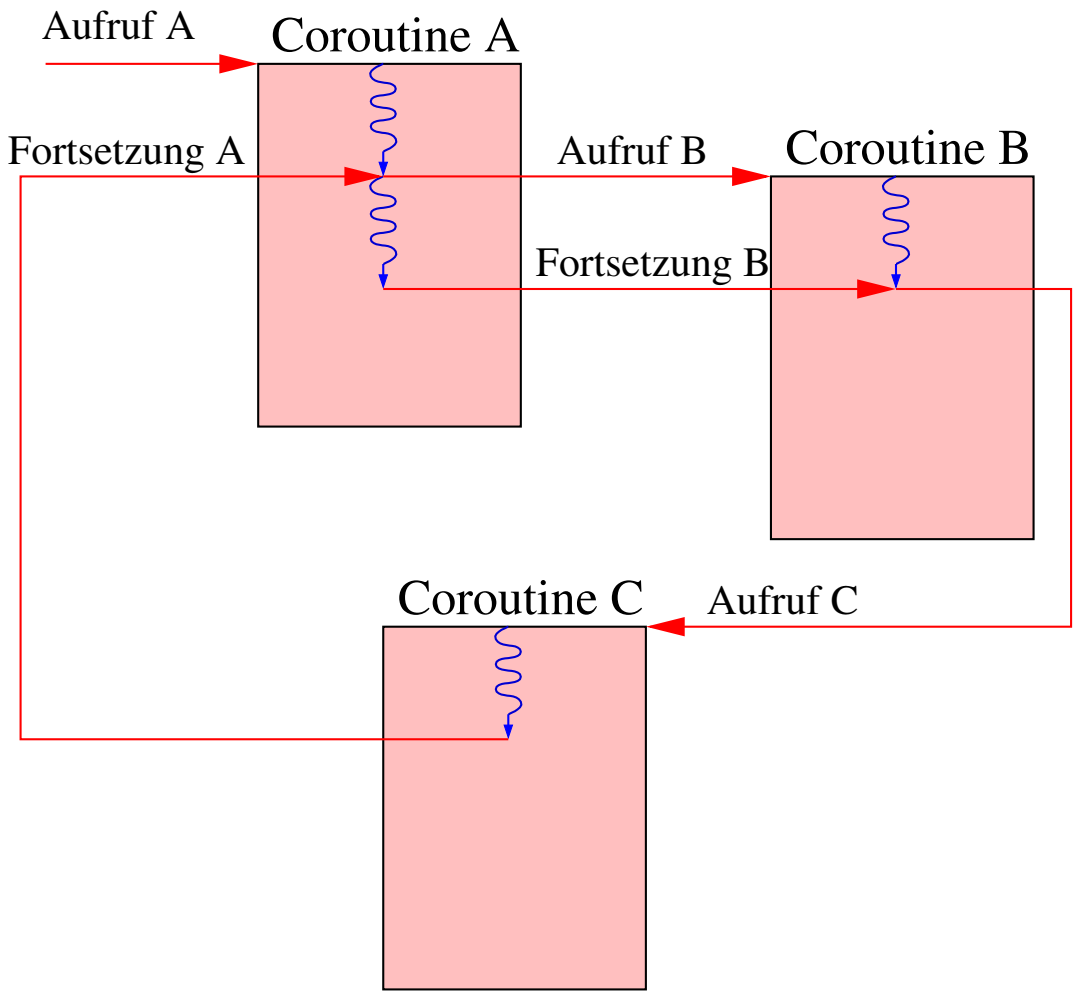
# Coroutinen


Aufruf A
Coroutine A
Aufruf B
Coroutine B
Coroutine C
Aufruf C

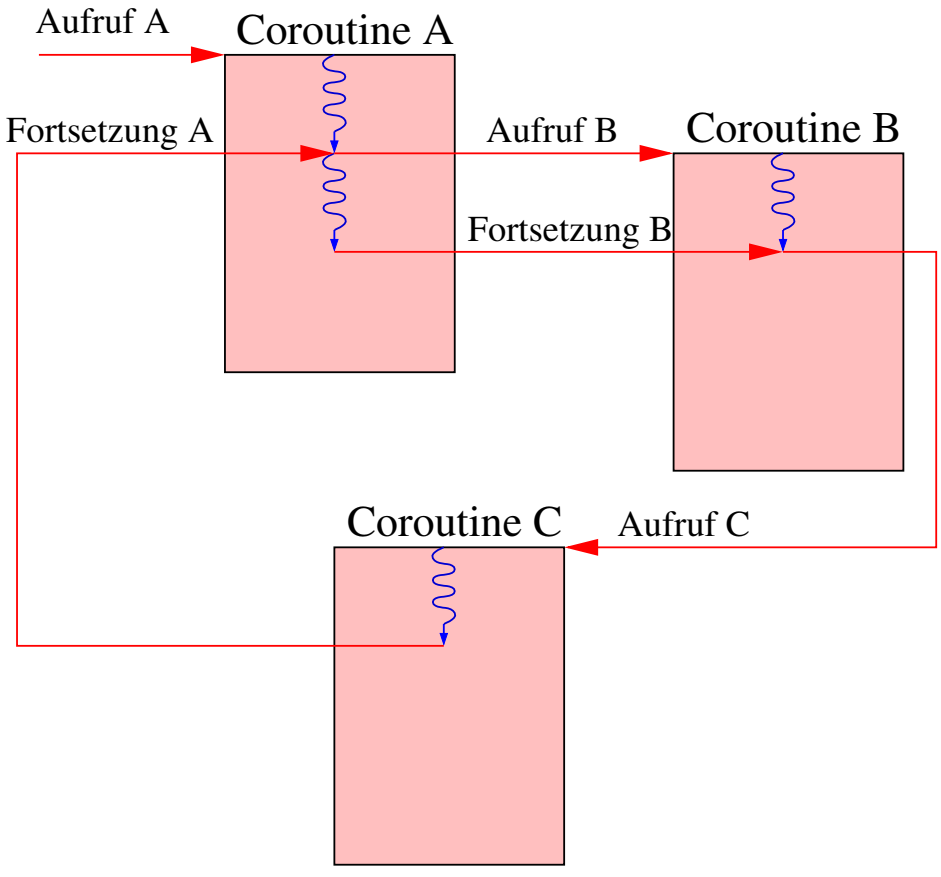
# Coroutinen


Aufruf A
Coroutine A
Fortsetzung A
Aufruf B
Coroutine B
Coroutine C
Aufruf C

# Coroutinen


Aufruf A
Coroutine A
Fortsetzung A
Aufruf B
Coroutine B
Fortsetzung B
Coroutine C
Aufruf C

# Coroutinen




Statt Subordination der Aufgerufenen gegenüber der Aufrufenden

$\Longrightarrow$ **Koordination** $\Longrightarrow$ **Coroutinen**

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Eine Erzeuger-Verbraucher-Architektur:

- Erzeuger stellt eine Ressource zur Verfügung
- Verbraucher benutzt die Ressource
- Kontrolle liegt alternativ beim Erzeuger bzw. Verbraucher

# Erzeuger-Verbraucher-Architektur mit Coroutinen

```
val buf = ref 0

fun produce (n,consumer:state) =
 (buf := n; produce (n+1,resume consumer))

fun consume (producer:state) =
 (print (Int.toString (!buf));consume (resume producer))
```

- `resume`-Aufrufe implementieren die Übergabe der Kontrolle zwischen dem Erzeuger und dem Verbraucher.
- Bei der Übergabe der Kontrolle muss der aktuelle Zustand des Aufrufenden mit übergeben werden.
- Der Zustand einer Coroutine ist eine Continuation.

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Implementierung der Kontrollübergabe, erster Versuch:

```
val buf = ref 0

fun resume k = callcc (fn k1 => throw k k1)

fun produce (n,consumer) =
 (buf := n; produce (n+1,resume consumer))

fun consume (producer) =
 (print (Int.toString (!buf));consume (resume producer))
```

- Die Typ-Inferenz für `produce` liefert (☞ Übung):

$$\{'a = 'a \texttt{ cont cont}$$

- Lösung hinsichtlich Typ-Überprüfbarkeit: definiere `state` rekursiv

```
datatype state = S of state cont
```

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Implementierung der Kontrollübergabe, zweiter Versuch:

```
val buf = ref 0

datatype state = S of state cont

fun resume (S k) = callcc (fn k1 => throw k (S k1))

fun produce (n,consumer:state) =
 (buf := n; produce (n+1,resume consumer))

fun consume (producer:state) =
 (print (Int.toString (!buf));consume (resume producer))
```

$\Longrightarrow$ produce and consume sind typbar (☞ Übung):

```
val produce = fn : int * state -> 'a
val consume = fn : state -> 'a
```

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Äquivalent dazu (durch Inlining von `resume`):

```
val buf = ref 0

datatype state = S of state cont

fun produce (n,S cons) =
  (buf := n;
   produce (n+1,callcc (fn k => throw cons (S k))))

fun consume (S prod) =
  (print (Int.toString (!buf));
   consume(callcc (fn k => throw prod (S k))))
```

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Anfang:

```
val buf = ref 0

datatype state = S of state cont

fun produce (n,S cons) =
 (buf := n;
  produce (n+1,callcc (fn k => throw cons (S k))))

fun consume (S prod) =
 (print (Int.toString (!buf));
  consume(callcc (fn k => throw prod (S k))))

fun run () = consume (callcc (fn k => produce (0, S k)))
```

# Continuations-Anwendung: Threads

- Mit zunehmender Anzahl Coroutinen in einem Programm wird die explizite Übergabe der Kontrolle mühsam und unübersichtlich.
- Besser: **Threads**
  - flexibler und modularer Ansatz des Kontrollflusses
  - asynchrone Events können auch behandelt werden
- Eine Coroutine **Scheduler** koordiniert die verschiedenen Threads.
- Threads sind Coroutinen des Schedulers.

# Continuations-Anwendung: Threads

- Scheduler-Funktionalität:
  - verwaltet eine Schlange `ready` von Threads
  - wenn aufgerufen (via `dispatch`), wählt einen Thread und setzt diesen fort
  - Ein Thread ist eine `unit cont`
- Definition eines Typ-Synonyms

```
type thread = unit cont
```

# Continuations-Anwendung: Threads

## Scheduler-Funktionalität:

```
exception NoMoreThreads
type thread = unit cont

val ready : thread Queue.queue = Queue.mkQueue ()

fun dispatch () =
  throw (Queue.dequeue ready) ()
  handle Queue.Dequeue => raise NoMoreThreads
```

# Continuations-Anwendung: Threads

## Threads-Funktionalität:

- `fork : (unit -> unit) -> unit`
  `fork f` erzeugt einen neuen Thread, der die Funktion $f$ ausführt. Der aufrufende Thread wird suspendiert.

- `yield : unit -> unit`
  `yield ()` übergibt die Kontrolle an den Scheduler

- `exit : unit -> 'a`
  `exit ()` beendet den aufrufenden Thread

# Continuations-Anwendung: Threads

## Threads-Funktionalität:

```
fun enqueue t = Queue.enqueue (ready,t)

fun exit () = dispatch ()

fun fork f =
  callcc (fn parentCont =>
              (enqueue parentCont;f(); exit()))

fun yield () =
  callcc (fn cont => (enqueue cont; dispatch ()))
```

# Erzeuger-Verbraucher-Threads

```
val buffer = ref 0

fun producer () =
  (buffer := !buffer + 1;yield();producer ())

fun consumer () =
  (print (Int.toString (!buffer));yield();consumer())

fun run () =
  (init ();fork consumer;producer ())

fun run2 () =
  (init ();fork consumer; fork producer; producer ())
```

# Threads: Pre-emption

- Unser Scheduler ist nicht **pre-emptive**!

- Pre-emption braucht Laufzeitsystem-Unterstützung: `Signals` in SMLofNJ:

```
val setHandler : (signal * sig_action) -> sig_action

eqtype signal

datatype sig_action = IGNORE | DEFAULT
  | HANDLER of (signal * int * unit cont) -> unit cont
```

- Installierung eines Handler für ein Signal:
  `setHandler(signal,HANDLER (fn (signal,n,k) => k'))`
  Unterbrochener Thread und Handler sind Coroutinen. Tritt `signal` auf, dann:
  1. Laufzeitsystem übergibt aktuelle Cont. `k` dem Handler;
  2. Handler liefert eine neue Cont. `k'`;
  3. Das Laufzeitsystem aktiviert `k'`.

# Threads: Pre-emption

- Idee:

```
Signals.setHandler
  (Signals.sigALRM,
   Signals.HANDLER (fn (_,_,k) =>
                       (enqueue k; dispatch())

val granularity = Time.fromMilliseconds 10

SMLofNJ.IntervalTimer.setIntTimer (SOME granularity)
```

- Zugriff auf gemeinsamen Datenstrukturen (z.B. `ready`) muss atomisch sein
- Signal Behandlung ist eine atomische Operation
- Continuations + Signals Handling $\Longrightarrow$ **Concurrent ML** = SML Erweiterung um Nebenläufigkeit

# Andere Anwendungen von Continuations

- Ausnahmen
- Lösungs-Generatoren
- Multi-Agent-Programmierung
- Iteratoren
- Backtracking
- andere benutzer-definierte Kontrollstrukturen

# Überblick

## 10. Modularisierung

Strukturen

Signaturen

Funktoren

Sharing Constraints

# Motivation

- Bis jetzt haben wir alle Deklarationen in der Top-Level-Umgebung eingeführt
  - Abhängigkeit der Deklarationen diktiert Reihenfolge ihrer Einführung/Implementierung

  $\Longrightarrow$ keine unabhängige Entwicklung der Programmteile

  $\Longrightarrow$ separate Kompilierung nicht möglich

- Einkapselung zusammenhängender Deklarationen in Programmeinheiten (**Module**), die relativ unabhängig voneinander behandelt werden können.

# Strukturen

- In SML heißen die Module **Strukturen**. Eine Struktur wird zwischen **struct** und **end** eingeklammert:

```
struct
   type 'a Pair = 'a * 'a
   fun Pair(a,b) = (a,b)
   fun first (a,b) = a
   fun second (a,b) = b
end
```

# Strukturen

- In SML heißen die Module **Strukturen**. Eine Struktur wird zwischen **struct** und **end** eingeklammert:

```
struct
  type 'a Pair = 'a * 'a
  fun Pair(a,b) = (a,b)
  fun first (a,b) = a
  fun second (a,b) = b
end
```

- Eine Struktur kann zu einem Struktur-Bezeichner gebunden werden:

```
structure Pairs =
struct
  type 'a Pair = 'a * 'a
  fun Pair(a,b) = (a,b)
  fun first (a,b) = a
  fun second (a,b) = b
end
```

# Signaturen

Ähnlich wie andere Werte...

- eine Struktur hat einen Typ, genannt **Signatur**
- Auf die Eingabe einer Struktur antwortet der Compiler mit ihrer Signatur:

```
structure Pairs :
    sig
        type 'a Pair = 'a * 'a
        val Pair : 'a * 'b -> 'a * 'b
        val first : 'a * 'b -> 'a
        val second : 'a * 'b -> 'b
    end
```

# Sichtbarkeit

- Die Definitionen innerhalb der Struktur sind außerhalb (insb. top-level) zunächst nicht sichtbar:

```
- first;
stdIn:41.1-41.6 Error: unbound variable or constructor: first
```

- Die Namen innerhalb einer Struktur sind ansprechbar über den Namen der Struktur:

```
- Pairs.first;
val it = fn : 'a * 'b -> 'a
```

# Strukturen als Namespaces

- Strukturen helfen u.a. zur Einführung von Namespaces zur Vermeidung von Namenskonflikten.

- Bsp.: Funktionen für verschiedene Typen mit dem gleichen Namen definieren:

```
structure Triples =
  struct
    datatype 'a Triple = Triple of 'a * 'a * 'a
    fun first (Triple abc) = #1 abc
    fun second (Triple abc) = #2 abc
    fun third (Triple abc) = #3 abc
  end

- Triples.first;
val it = fn : 'a Triples.Triple -> 'a
```

# Öffnen von Strukturen

- Um nicht immer den Strukturnamen verwenden zu müssen, kann man auch alle Definitionen einer Struktur auf einmal sichtbar machen:

```
- open Pairs;
opening Pairs
   type 'a Pair = 'a * 'a
   val Pair : 'a * 'b -> 'a * 'b
   val first : 'a * 'b -> 'a
   val second : 'a * 'b -> 'b
- Pair;
val it = fn : 'a * 'b -> 'a * 'b
- Pair (4,3);
val it = (4,3) : int * int
```

# Öffnen von Strukturen

- Strukturen öffnen sollte man aber möglichst nur lokal tun, um Namenskonflikte mit anderen offenen Modulen zu vermeiden:

- Einschränkung der Sichtbarkeit der Deklarationen:
  - Für Ausdrücke:

```
let open struct
in expr end
```

```
let open Real in (toString 2.2)^"f" end;
val it = "2.2f" : string
```

  - Für Definitionen:

```
local open struct
in expr end
```

```
local open Real
in fun Real2String r = (toString r)^"f" end;
val Real2String = fn : real -> string
```

# Geschachtelte Strukturen

- Strukturen können selbst auch wieder Strukturen enthalten:

```
structure Quads =
  struct
    structure Pairs =
      struct
        type 'a Pair = 'a * 'a
        fun Pair(a,b) = (a,b)
        fun first(a,_) = a
        fun second(_,b) = b
      end
    type 'a Quad = 'a Pairs.Pair Pairs.Pair
    fun Quad (a,b,c,d) =
        Pairs.Pair (Pairs.Pair(a,b),Pairs.Pair(c,d))
    fun first q = Pairs.first(Pairs.first q)
    fun second q = Pairs.second(Pairs.first q)
    fun fourth q = Pairs.second(Pairs.second q)
  end
```

# Geschachtelte Strukturen

```
- Quads.Quad(1,2,3,4);
val it = ((1,2),(3,4)) : (int * int) * (int * int)
- Quads.Pairs.first;
val it = fn : 'a * 'b -> 'a
```

# Signaturen...

- erscheinen nicht nur als Compiler-Antworten;
- können eine erwünschte Funktionalität spezifizieren;
- können zu einem Signatur-Bezeichner gebunden werden.
- Bsp.: Counter-Funktionalität:

```
signature CountSig =
  sig
    val setCounter : int -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> int
  end
```

# Signatur-Einschränkungen

- Mit Hilfe von Signaturen kann man einschränken, was ein Modul nach außen exportiert.

- Bsp.:

```
structure Count =
  struct
    val cnt = ref 0
    fun setCounter n  = cnt := n
    fun getCounter () = !cnt
    fun incCounter () = cnt := !cnt+1
  end

- !Count.cnt;
val it = 0 : int
```

Der Zähler ist nach außen sichtbar, zugreifbar und sogar veränderbar.

# Signatur-Einschränkungen

- Mit einer Signatur kann man eine eingeschränkte Schnittstelle (*view*) einer Struktur ereugen:

```
- structure SafeCount = Count:CountSig;
structure SafeCount : CountSig
- open SafeCount;
opening SafeCount
   val setCounter : int -> unit
   val incCounter : unit -> unit
   val getCounter : unit -> int
- SafeCount.cnt;
stdIn:1.1-1.14 Error: unbound variable or constructor:cnt in path SafeCount.cnt
```

- Die Signatur bestimmt also, welche Definitionen exportiert werden (`CountSig` enthält den Counter selbst nicht).

# Signatur-Einschränkungen

- Eine eingeschränkte Schnittstelle zuweisen geht auch schon direkt bei der Definition:

```
structure Count1 : Count =
  struct
    val cnt = ref 0
    fun setCounter n  = cnt := n
    fun getCounter () = !cnt
    fun incCounter () = cnt := !cnt+1
  end

- !Count1.cnt;
stdIn:196.2-196.12 Error: unbound variable or constructor: cnt in path Count1.cnt
```

# Signatur-Einschränkungen

- Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein: Dadurch werden deren Typen spezialisiert:

```
signature A1 = sig val f : 'a -> 'b -> 'b end
signature A2 = sig val f : int -> char -> int end
structure A = struct fun f x y = x end
- structure A1 = A:A1;
stdin: Error: value type in structure doesn't match signature spec name: f
   spec: 'a -> 'b -> 'b
   actual: 'a -> 'b -> 'a
- structure A2 = A:A2;
structure A2 : A2
- A2.f;
val it = fn : int -> char -> int
```

# Information Hiding

- Aus Gründen der Modularität möchte man oft nicht, dass die Struktur der Typen, die ein Modul zur Verfügung stellt, nach außen bekannt ist. Beispiel:

```
structure ListQueue =
  struct
    exception EmptyQueue
    type 'a Queue = 'a list
    val empty = nil
    fun isempty nil = true
      | isempty _   = false
    fun enqueue nil y = [y]
      | enqueue (x::xs) y = x :: enqueue xs y
    fun dequeue nil = raise EmptyQueue
      | dequeue (x::xs) = (x,xs)
  end
```

# Information Hiding

- Will man verstecken, dass eine Queue eine Liste ist, kann man das mit einer Signatur:

```
signature QueueSig =
  sig
    exception EmptyQueue
    type 'a Queue
    val empty : 'a Queue
    val isempty : 'a Queue -> bool
    val enqueue : 'a Queue -> 'a -> 'a Queue
    val dequeue : 'a Queue -> 'a * 'a Queue
  end
```

# Information Hiding

- Das Einschränken per Signatur genügt nicht, um die wahre Natur des Typs `Queue` zu verschleiern.

```
- structure Queue = ListQueue : QueueSig;
structure Queue : QueueSig

- open Queue;
opening Queue
  exception EmptyQueue
  type 'a Queue = 'a list
  val empty : 'a Queue
  val isempty : 'a Queue -> bool
  val enqueue : 'a Queue -> 'a -> 'a Queue
  val dequeue : 'a Queue -> 'a * 'a Queue
- isempty nil;
val it = true : bool
```

# Information Hiding

- Um die Implementierung der Datentypen zu verstecken, muss man das so genannte *opaque signature matching* (`:>` anstatt `:`) verwenden:

```
- structure HiddenQueue = ListQueue :> Queue;
structure HiddenQueue : Queue
```

- Durch die Verwendung von `:>` werden alle exportierten Typen, deren Definition nicht explizit in der Signatur steht, abstrahiert.

# Information Hiding

```
- open HiddenQueue;
opening HiddenQueue
  exception EmptyQueue
  type 'a Queue
  val empty : 'a Queue
  val isempty : 'a Queue -> bool
  val enqueue : 'a Queue -> 'a -> 'a Queue
  val dequeue : 'a Queue -> 'a * 'a Queue
```

# Information Hiding

- Die Struktur `HiddenQueue` ist ein **abstrakter Datentyp**:

```
- isempty empty;
val it = true : bool

- isempty nil;
stdIn:301.1-301.12 Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z Queue
  operand: 'Y list
  in expression:
    isempty nil
```

# Funktoren: Motivation

- Möglichst unabhängige Entwicklung: Ein Modul hängt nur von den Signaturen anderer Module und nicht von ihren Implementierungen.

- Bsp.: ein Parser kann einen beliebigen Lexer benutzen, dessen Signatur eine Funktion `next` implementiert, die das nächste Token liefert.
  - Problem: Code Duplizierung
    - `struct MyParser = ... MyLexer.next() ... end`
    - `struct YourParser = ... YourLexer.next() ... end`

  - Lösung: parametrisierte Module = **Funktoren**

# Funktoren: Motivation

- Ein Funktor bekommt als Parameter eine Reihe von Werten, Typen oder ganzen Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Parameter des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.
- Bsp:

```
functor Parser(Lexer :
                  sig val next: unit -> int end) =
  struct ... Lexer.next() ... end
```

Instantiiere Parser mit einem bestimmten Lexer = **Funktor-Anwendung**:

- `structure MyParser = Parser(MyLexer)`
- `structure YourParser = Parser(YourLexer)`

# Funktoren: Motivation

- Funktoren unterstützen **generische** Datenstrukturen/Algorithmen:
  - Signatur-Constraints für Argumente erfassen das **Generische**
  - Das Spezifische/Unwesentliche steckt in den übergebenen Strukturen, die mehr als von der Signatur verlangt implementieren können

```
functor SortedList
          (Element:sig val le : 'a*'a -> bool end )=
  struct ... if Element.le (x,y) ... end
```

# Funktoren: Vorgehensweise

- Man legt zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
signature Enum =
  sig
    type Enum
    val null : Enum
    val incr : Enum -> Enum
  end

signature Counter =
  sig
    type Counter
    val setCounter : Counter -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> Counter
  end
```

# Funktoren

```
functor GenCounter (structure Enum : Enum) : Counter =
  struct
    open Enum
    type Counter = Enum
    val cnt = ref null
    fun setCounter x = cnt := x
    fun incCounter () = cnt := incr(!cnt)
    fun getCounter () = !cnt
  end
```

- Die Anwendung des Funktors `GenCounter` auf eine Struktur mit der Signatur `Enum` erzeugt eine neue Struktur mit der Signatur `Counter`.

# Funktoren

```
structure IntEnum =
  struct
    type Enum = int
    val null = 0
    fun incr x = x+1
  end

structure IntCounter =
   GenCounter(structure Enum = IntEnum);
structure IntCounter : Counter

- IntCounter.setCounter 5;
val it = () : unit
- IntCounter.incCounter ();
val it = () : unit
- IntCounter.getCounter ();
val it = 6 : IntCounter.Counter
```

# Funktoren

- Eine erneute Anwendung des Funktors erzeugt eine neue Struktur:

```
structure StringEnum =
  struct
    type Enum = string
    val null = "a"
    fun incr str =
     let val cs = String.explode str
         val new =
           case cs of nil => [#"a"]
                    | #"z"::_ => #"a"::cs
                    | c::cs' => chr(ord c+1)::cs'
     in String.implode new
     end
  end
- structure StringCounter =
    GenCounter (structure Enum = StringEnum);
```

# Funktoren

```
- StringCounter.incCounter ();
val it = () : unit
- StringCounter.getCounter ();
val it = "b" : StringCounter.Counter
```

# Funktoren

- Mit einem Funktor kann man sich auch mehrere Instanzen des gleichen Moduls erzeugen:

```
- structure CountApples =
    GenCounter (structure Enum = IntEnum);
structure CountApples : Counter
- structure CountPears =
    GenCounter (structure Enum = IntEnum);
structure CountPears : Counter
```

# Funktoren

```
- CountApples.incCounter ();
val it = () : unit
- CountApples.incCounter();
val it = () : unit
- CountPears.incCounter ();
val it = () : unit
- CountApples.getCounter ();
val it = 2 : CountApples.Counter
- CountPears.getCounter ();
val it = 1 : CountPears.Counter
```

# Sharing Constraints

```
signature LexerSig = sig structure Symbol : SymbolSig
                         val next : unit -> Symbol.sym
                     end
signature SymTableSig =
  sig structure Symbol : SymbolSig
      structure Value : ValueSig
      type table
      val lookup : table * Symbol.sym -> Value.val
  end
functor Parser(Lexer:LexerSig, SymTable:SymTableSig) =
  struct ... SymTable.lookup(table,Lexer.next ()) ...end
```

- Allein unter Verwendung Informationen aus dem aktuellen Modul (der Argument-Signaturen), kann der Type-Checker nicht überprüfen, ob `SymTable.Symbol.sym` = `Lexer.Symbol.sym`.
- Unsere Absicht, dass `Lexer.Symbol` = `SymTable.Symbol` muss explizit dokumentiert werden.
- Zusätzliche Beziehungen zwischen Funktor-Argumenten muss man explizit mit Hilfe von **sharing constraints** angegeben.

# Sharing Constraints für Typen...

... spezifizieren, dass zwei Typen identisch sein müssen:

```
signature LexerSig =
  sig structure Symbol : SymbolSig
      val next : unit -> Symbol.sym
  end

signature SymTableSig =
  sig structure Value : ValueSig
      structure Symbol : SymbolSig
      type table
      val lookup : table * Symbol.sym -> Value.val
      val update : table * Symbol.sym * Value.val ->table
  end

functor Parser(Lexer:LexerSig, SymTable:SymTableSig
               sharing type SymTable.Symbol.sym = Lexer.Symbol.sym) =
  struct ... SymTable.lookup(table,Lexer.next ()) ...end
```

- Bei der Funktor-Anwendung überprüft der Compiler, dass die zwei Typen gleich sind.

# Sharing Constraints für Strukturen...

... spezifizieren, dass zwei Strukturen identisch sein müssen:

```
signature LexerSig =
  sig structure Symbol : SymbolSig
      val next : unit -> Symbol.sym
  end

signature SymTableSig =
  sig structure Value : ValueSig
      structure Symbol : SymbolSig
      type table
      val lookup : table * Symbol.sym -> Value.val
      val update : table * Symbol.sym * Value.val ->table
  end

functor Parser(Lexer:LexerSig, SymTable:SymTableSig
               sharing SymTable.Symbol = Lexer.Symbol) =
  struct ... SymTable.lookup(table,Lexer.next ()) ...end
```

# Programmieren im Großen

- Eingabe längerer Programme auf der interaktiven Kommando-Zeile ist mühsam.

- $\Longrightarrow$ Lese sie aus Dateien ab
  - `CM.use : string -> unit` liest, übersetzt und macht den Inhalt einer Datei in der SML-Umgebung sichtbar.

```
- use "test.sml";
[opening test.sml]
  type s = int
  type t = bool
  val x = 1 : int
  val y = 2 : int
val it = () : unit
```

# Projekte in mehreren Dateien

- Größere Programmier-Projekte sind jedoch i.A. auf mehrere Dateien und Verzeichnisse verteilt.
- Regeln:
  - jede Struktur bzw. jeder Funktor liegen in einer separaten Datei;
  - mehrfach verwendete Signaturen sollten in eigenen Dateien gesammelt werden;
  - zusammengehörige Projekt-Bestandteile kommen in ein gemeinsames Verzeichnis.

# Projekte in mehreren Dateien

- Um ein Projekt aus mehreren Dateien mit Hilfe von **use** zu kompilieren, müsste man:
  - sich die Datei-Namen merken;
  - die entsprechenden Folgen von `use`-Aufrufen verwalten.
- Dazu stellt SML/NJ die Struktur `CM` , der **Compilation Manager**, zur Verfügung.

# Der Compilation Manager

- Zum Laden von Bibliotheken bietet CM:
  - `CM.make : unit -> unit` sucht eine Datei "sources.cm" und versucht, aus der dort angegebenen Spezifikation eine Bibliothek herzustellen.
  - `make' : string -> unit` benutzt stattdessen den angegebenen String als Datei-Namen.
- Bsp.-Spezifikation:

```
Library
    signature Counter
    structure IntCounter
is
    counter.sig
    intcounter.sml
    foo.sml
```

Die Bibliothek stellt die Signatur `Counter` sowie die Struktur `IntCounter` bereit. Zu deren Herstellung dienen die nach dem `is` aufgelisteten Dateien (die "Members").

# Der Compilation Manager

- Um Bibliotheken effizient herstellen zu können, **verwaltet** CM **Abhängigkeiten** zwischen den in Dateien definierten Strukturen.
- Die **Übersetzung** der Dateien **berücksichtigt diese Abhängigkeiten**. Insbesondere wird eine Datei nur dann neu übersetzt, wenn sie seit der letzten Kompilierung verändert wurde oder von Dateien abhängt, deren Modifizierung einen Einfluss haben könnten.

# Der Compilation Manager

- Die **exportierten Elemente** können auch innerhalb der Bibliothek von den Members benutzt werden.

```
Library
    signature Bar
    structure Foo
is
    bar.sig
    foo.sml
```

# Der Compilation Manager

- Für den lokalen Gebrauch innerhalb von Bibliotheken kann man mehrere Dateien zu einer **Gruppe** zusammen fassen.

```
Library
    signature A
    structure A
is
    a.sig
    a.sml
    utils.cm
```

- `utils.cm` könnte dann z.B. Funktionen aus einer Datei `utils.sml` und Datenstrukturen aus einer Datei `data.sml` zusammenfassen.

```
Group
is
    utils.sml
    data.sml
```

# Der Compilation Manager

```
Group
is
    utils.sml
    data.sml
```

- Sämtliche definierten Elemente werden exportiert.
- Soll der Export eingeschränkt werden, kann man eine **explizite Export-Liste** angeben:

```
Group
    structure Data
is
    utils.sml
    data.sml
```

# Der Compilation Manager

## Regeln:

- Explizit können nur Funktoren, Strukturen und Signaturen exportiert werden.
- Jede Datei darf nur einmal in einer ".cm"-Datei vorkommen.
- Gruppen und Bibliotheken können beliebig oft verwendet werden.

# Der Compilation Manager

## Weitere Features:

- automatischer Aufruf von **Präprozessoren** ("Tools"), die die Source-Datei erst noch erzeugen müssen;
- **bedingter Export** bzw. bedingter Einschluss von Members in Abhängigkeit z.B. von SML/NJ-Version oder Betriebssystem;
- Erzeugung von **Stand-alone-Versionen** (SMLofNJ-Struktur).

# Überblick

# 12.
# Eine nicht strikte funktionale Sprache: Haskell

# Nicht-strikte (verzögerte) Auswertung

Im Unterschied zu SML ist Funktionsauswertung in Haskell **nicht-strikt**, d.h. Argumente werden nur bei Bedarf (**verzögert**) ausgewertet. Betrachte die definition `f x = 1`. Dann liefert:

- In SML:

```
f (1 div 0)
uncaught exception divide by zero
```

- In Haskell:

```
f (1 `div` 0)
1 :: Integer
```

# Basis-Typen in Haskell

| Typ | Bsp.-Werte | Bsp.-Operatoren |
|---|---|---|
| Integer | `0 3 -7` | `+ div mod` : Integer $\times$ Integer $\mapsto$ Integer<br>`-` : Integer $\mapsto$ Integer |
| Double | `-3.0 7.0` | `+ - * /` : Double $\times$ Double $\mapsto$ Double<br>`~` : Double $\mapsto$ Double |
| Bool | `True False` | `not` : Bool $\mapsto$ Bool<br>`\|\|  &&` : Bool $\times$ Bool $\mapsto$ Bool |
| Char | `'a' 'b' '\n'` | |

# Funktionen

- Definition:

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

- Anwendung:

```
fact 4
24 :: Integer
```

# Namenlose Funktionen

```
(\x -> x+1) 2
```
*3 :: Integer*

# Produkt-Typen

- **Produkt-Konstruktor** wie in SML
- **Produkt-Typoperator** syntaktisch ähnlich wie der Produkt-Konstruktor:

```
(1,2)
(1,2) :: (Integer,Integer)
(True,'a')
(True,'a') :: (Bool,Char)
```

# Summen-Typen

- Definition:

$$\begin{array}{l}\texttt{data T } \alpha_1\ \alpha_2\ \ldots\ \alpha_n = \mathit{Konstruktor}_1\ \beta_{11}\ \ldots\ \beta_{1n_1} \\ \qquad\qquad\qquad\qquad\quad \mid \mathit{Konstruktor}_2\ \beta_{21}\ \ldots\ \beta_{2n_2} \\ \qquad\qquad\qquad\qquad\quad \ldots \\ \qquad\qquad\qquad\qquad\quad \mid \mathit{Konstruktor}_m\ \beta_{m1}\ \ldots\ \beta_{mn_m}\end{array}$$

  Damit werden deklariert:

  - der **Typ-Operator** `T` (Prä-fixiert)
  - die **Konstruktor-Funktionen** (curried):

    $\mathit{Konstruktor}_i$ `::` $\beta_{i1}$ `->` $\beta_{i2}$ `->` `...` $\beta_{in_1}$ `->` `T` $\alpha_1$ `...` $\alpha_n$

- Bsp.:
  - `data Tree a = Leaf a | Node (Tree a) a (Tree a)`

# Pattern-Matching

- Ähnlich wie in SML:

```
case Ausdruck of
     Muster_1 -> Ausdruck_1
     Muster_2 -> Ausdruck_2
     ...
     Muster_n -> Ausdruck_n
```

- Bsp.:

```
nodes t =
  case t of
    Leaf _ -> 1
    Node t1 _ t2 -> 1 + (nodes t1) + (nodes t2)
```

- `if` $e_1$ `then` $e_2$ `else` $e_3$ ist eine spezielle Syntax für

```
case e_1 of True -> e_2
            False -> e_3
```

# Listen

- Der Typ von Listen mit Elementen von Typ $\alpha$, `[`$\alpha$`]`, ist ein vordefinierter Summentyp mit:
  - Nullstelliger Konstruktor *nil*: `[]`
  - Einstelliger Konstruktor *cons*: `:` (rechtsassoziativ)
- Bsp.:
  - `1:2:3:[]`
  - Syntaktische Abkürzung: `[1,2,3]`
- Vordefinierte List-Funktionen: `head`, `tail`, `map`, `foldl`, `foldr`, `reverse`, `take`, `drop`, etc.

# Strings

- `String` ist ein anderer Typ-Name (Synonym) für `[Char]`:

```
"hallo"
"hallo" :: String
reverse "hallo"
"ollah" :: [Char]
```

- Benutzer-definierte **Typ-Synonyme** werden mit `type` deklariert:

```
type Coordinate3D = (Integer,Integer,Integer)
```

# Lokale Definitionen

- **Lokale Definitionen** werden mit dem `let`-Ausdruck eingeführt:

```
let  Definition_1
     Definition_2
     ...
     Definition_n
in   Ausdruck
```

  - *Definition*$_i$ ist der Form: *Pattern*$_i$ = *Ausdruck*$_i$
  - Die Definitionen sind in `Ausdruck` sichtbar;
  - Die Definitionen können verschränkt rekursiv sein;
  - Der Wert des `let`-Ausdrucks ist der Wert von *Ausdruck*.

```
fibo n =
  let f1 = 0
      f2 = 1
      fiboHelp f1 f2 n =
        if n<=0 then f1
        else fiboHelp f2 (f1+f2) (n-1)
  in fiboHelp f1 f2 n
```

# Layout

- Keine Schlüsselwörter (wie etwa `val` oder `end` in SML) trennen die Definitionen in einem **let**-Ausdruck.
  $\Longrightarrow$ Sind folgende Definitionen äquivalent?

```
let  x = y
     z  t = x
in  z 1
```

```
let  x = y  z
     t = x
in  z 1
```

- Nein – Haskell verwendet zur Auflösung von Mehrdeutigkeiten beim Parsen die Einrückung (**Layout**) des Quellcodes:
  - Das Ende einer Deklaration wird durch ein nachfolgendes Symbol in einer Spalte <= Anfangsspalte der Deklaration signalisiert.

- Ähnliches gilt für die Trennung der Fälle in einem **case**-Ausdruck

# Layout

- Das Layout ist eine verkürzte Syntax einer Einrückung-unabhängigen Spezifikation, die explizite Separator- und Terminator-Symbole benutzt, um einzelne Deklarationen voneinander abzugrenzen.

- Bsp.: alternative Syntax für **let**-Ausdrücke

```
let { Definition_1 ; Definition_1 ; ... ; Definition_n } in Ausdruck
```

- Die zwei Syntax-Formen können gemischt werden. Bsp.:

```
let x = 1; y = f
    g x = x
in g 1
```

# Polymorphismus in Haskell

- Ähnlich wie SML unterstützt Haskell **parametrischer Polymorphismus**.
  Bsp. `length :: [a] -> Integer`
  - `length` hat eine einzige Definition
  - Keine Voraussetzungen an die Typen der Argumente $\Longrightarrow$ Typvariablen sind universal quantifiziert

- Zusätzlich unterstützt Haskell eine kontrollierte Form des **ad-hoc Polymorphismus** (*overloading*): der selbe Funktionsname steht für verschiedene Definitionen.
  - Bsp. Gleichheitstest-Fkt. `(==)` ist verschieden definiert für jeden verschiedenen Typ.
  - Andere Bsp.: `(+)`, `show`.

# Typklassen

- Überladung stellt neue Herausforderungen an die statische Typ-Überprüfung
- **Problem**: **Typ-Überprüfung** – welchen Typ hat ==?
  - Man darf nicht den Typ einer überladenen Fkt. mit einer universell quantifizierten Typ-Variable parametrisieren.
- **Lösung** bei Haskell: assoziiere Funktionsnamen mit **Typ-Klassen**.

# Typklassen

- **Syntax:** Typklasse = **Kollektion von Funktionsdeklarationen**.

```
class C α where
  f1 :: β1
  f2 :: β2
  ...
  fn :: βn
```

  - $C$ = Typklassen-Name
  - $\alpha$ = Platzhalter für konkrete Typen (**Instanzen**)
  - $\beta_1$, ..., $\beta_n$ = Typ-Ausdrücke, die $\alpha$ enthalten

- **Semantik:** Typklasse $C\ \alpha$ = Prädikat ((*type constraint/context*)), das die Existenz von (überladenen) Funktionsdefinitionen $f_1$, ..., $f_n$ für eine Instanz $\alpha$ vorschreibt.

# Typklassen

- Bsp.: Die Typklasse Eq ist wie folgt definiert:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

- Als Effekt dieser Deklaration, wird == folgenden Typ zugewiesen:
  `(==) :: (Eq a) => a -> a -> Bool`
  Typ-Constraint `(Eq a)` beschränkt den polymorphen Typ a auf Instanzen der Klasse Eq

- Typ-Überprüfung propagiert Typ-Constraints:

```
isIn e l = case l of [] -> False
                     h:r -> (e == h) || isIn e r
isIn :: Eq a => a -> [a] -> Bool
```

# Typinstanzen

- Die Zugehörigkeit eines Typs $\alpha$ zu einer Typklasse $C$ wird wie folgt deklariert:

```
instance C α where
  f1 = Definition1
  f2 = Definition2
  ...
  fn = Definitionn
```

- Bsp.:

```
data State = On | Off

instance Eq State where
  (==) s1 s2 = case (s1,s2) of (On,On) -> True
                               (Off,Off) -> True
                               _   -> False
  (/=) s1 s2 = not (s1 == s2)
```

```
On == On
True :: Bool
```

# Default-Definitionen

- Bei der Deklaration einer Klasse kann man **Default-Definitionen** spezifizieren:

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)

  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

- Jede Typ-Instanz besitzt diese Definition, wenn sie diese nicht überlädt.

- Für `Eq` braucht man nur eine von `(==)` oder `(/=)` definieren.

# Typ-Constraints in instance-Deklarationen

Bei instance-Deklarationen parametrisierter Typen müssen manchmal die Typ-Parameter eingeschränkt werden:

```
instance (Eq a) => Eq (Tree a) where

  (==) t1 t2 = case (t1,t2) of
     (Leaf x, Leaf y) -> (x==y)

     (Node tl1 k1 tr1, Node tl2 k2 tr2) ->
         (tl1==tl2) && (k1==k2) && (tr1==tr2)

     _ -> False
```

# Abgeleitete instance-Deklarationen für Summen-Typen

- Unsere `instance (Eq a) => Eq (Tree a)` Deklaration folgt der rekursiven Struktur des Datentyps `Tree`.

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
instance (Eq a) => Eq (Tree a) where
  (==) t1 t2 = case (t1,t2) of
     (Leaf x, Leaf y) -> (x==y)
     (Node tl1 k1 tr1, Node tl2 k2 tr2) ->
         (tl1==tl2) && (k1==k2) && (tr1==tr2)
     _ -> False
```

- Solche **instance-Deklarationen** für manche (vordefinierte) Typklassen können für Summentypen **automatisch abgeleitet** werden.

- Bsp.: Die abgeleitete `==`-Operation aus `Eq` für einen Summentyp benutzt die Identität der Konstruktoren und die rekursive Gleichheit der Komponenten.

# Abgeleitete instance-Deklarationen für Summen-Typen

- Automatisch abgeleitete instance-Deklarationen werden mit folgender Syntax bei der Definition der Typen eingeführt.

```
data T α1 α2 ... αn = Konstruktor1 β11 ... β1n1
                    | Konstruktor2 β21 ... β2n2
                    ...
                    | Konstruktorm βm1 ... βmnm
        deriving (C1, ..., Cp)
```

- Bsp.:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
      deriving (Eq, Show)
```

# Oberklassen

- Haskell unterstützt Klassenerweiterung: eine Typklasse $C$ vererbt alle Operationen der Klassen $C_1$, $C_2$, ..., $C_n$.
  $\Longrightarrow$ ein Typ $T$ ist eine Instanz der Typklasse $C$ nur wenn er auch schon eine Instanz von $C_1$, ..., $C_n$ ist.

- Syntax:

```
class (C1, ..., Cn) => C α where ...
```

- Bsp.:

```
class (Eq a) => Ord a where ...
```

- Die Klassenhierarchie, die dadurch entsteht muss azyklisch sein.

- In $C_i$ darf lediglich $\alpha$ als Typvariable auftreten.

# Vordefinierte Typklassen

- **Eq a**, wie oben
- **Ord a** mit `(<), (<=) :: a -> a -> Bool`
  - automatisch ableitbare instance-Dekl. für alle Summentypen laut Reihenfolge der Konstruktoren in `data`-Deklaration
- **Show a**, **Num a**, **Enum a**: siehe unten

# Die vordefinierte Typklasse Show

- Für Typen deren Werte eine String-Darstellung haben:

```
class Show a where
  show :: a -> String
  showList :: [a] -> (String -> String)
  -- Default-Definition fuer showList v
```

- Instanzen müssen `show` definieren
- Wird benutzt u.a. von der Interpreterumgebung
- Automatisch ableitbare instance-Dekl. gemäß Konstruktornamen

# Die vordefinierte Typklasse Num

- Alle numerischen Typen sind eine Unterklasse der Typklasse **Num**:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

- Alle Instanzen müssen vergleichbar sein und eine String-Darstellung haben.
- Macht die Überladung numerischer Konstanten möglich: je nach Kontext kann `2` vom Typ `Integer` oder `Double` sein.
- Eine numerische Konstante $k$ ist syntaktischer Zucker für `fromInteger` $k$.

# Die vordefinierte Typklasse Enum

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

- Für Typen, deren Werte aufzählbar sind.
- Defaults für alle Operationen bis auf `toEnum` und `fromEnum`:
  - Bsp.: `succ = toEnum . (+1) . fromEnum`

# Die vordefinierte Typklasse Enum

- Syntaktischer Zucker:
  - `[x..y]` ≡ `enumFromTo x y`

```
> [1 .. 8]
[1,2,3,4,5,6,7,8]
```

  - `[x..]` ≡ `enumFrom x`

```
> take 5 [3..]
[3,4,5,6,7]
```

  - `[x,y..z]` ≡ `enumFromThenTo x y z`

```
> [1,4 .. 20]
[1,4,7,10,13,16,19]
```

# Die vordefinierte Typklasse Enum

- Automatisch ableitbare instance-Dekl. der Klasse `Enum` für Aufzählungstypen : gemäß Reihenfolge der Konstruktoren in data-Deklaration

- Bsp.:

```
data Wert =
  Zwei | Drei | Vier | Fuenf | Sechs | Sieben | Acht |
  Neun | Zehn | Bube | Dame | Koenig | As
 deriving (Eq,Ord,Enum,Show)

> [Sieben .. Dame]
[Sieben,Acht,Neun,Zehn,Bube,Dame]
```

# List-Comprehensions

- List-Comprehensions sind syntaktischer Zucker für die Darstellung von Listen ähnlich wie Mengendefinitionen in der Mathematik:

$$A = \{f(x) \mid x \in B \wedge p(x)\}$$

- Bsp.:
  - Mathematische Notation:

    $Even = \{x \mid x \in \mathbb{N}, x \bmod 2 == 0\}$

  - Haskell-Notation:

```
even = [ x | x <- [1 ..], x `mod` 2 == 0]
```

```
> take 4 even
[2,4,6,8]
```

# List-Comprehensions

- **Syntax**: `[ e | `$q_1$`, `$q_2$`, ..., `$q_n$`]` mit $n \geq 1$ und die Qualifier $q_i$ einer der folgenden Formen sind:
  - **Generatoren**: *pattern*$_i$ `<-` *expr*$_i$ mit *expr*$_i$ vom Typ $[\alpha_i]$
  - **Filter**: Ausdrücke vom Typ `Bool`
- **Semantik** (operational):

```
[ e | v <- [], qs ]      → []
[ e | v <- (x:xs), qs ] →
    [ e | qs ][x/v] ++ [ e | v <- xs, qs ]
[ e | False, qs ]        → []
[ e | True, qs ]         → [ e | qs ]
[ e | ]                  → [ e ]
```

# List-Comprehensions: Beispiele

```
> [(x,y) | x <- [1,2,3], y <- [4,5,6]]
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]

> [(x,y) | x <- [1,2,3], y <- [4,5,6], x+y > 6]
[(1,6),(2,5),(2,6),(3,4),(3,5),(3,6)]

> [x | x <- [-100..100], x^2 + x -12 == 0]
[-4,3]

pythTriples n = [(x,y,z) | x <- [1.. n],
                           y <- [x..n],
                           z <- [y..n],
                           x^2 + y^2 == z^2]
> pythTriples 10
[(3,4,5),(6,8,10)]

quick l = case l of
  [] -> []
  h:r -> (quick [x | x <- r, x <= h]) ++ [h]
         ++ (quick [x | x <- r, x > h])
```

# Unendliche Datenstrukturen

- Die verzögerte Auswertung erlaubt unendliche Datenstrukturen direkt darzustellen.

- Bsp.:

```
onlyOnes = 1 : onlyOnes

natsFrom n = n : natsFrom (n+1)
-- synt. Zucker [n..]

squares = [ x^2 | x <- [1..]]

addCrtNumber l = zip [1..] l
```

Part III

# Logische Programmierung

# Logik als Programmiersprache

- Logik wird als Programmiersprache benutzt.
- Der logische Ansatz zu Programmierung ist (sowie der funktionale) deklarativ.
- Programme können mit Hilfe zweier abstrakten, Maschinen-unabhängigen Konzepte verstanden werden:
  - **Wahrheit**
  - **Logische Deduktion**

# Deklarativität der logischen Programmierung

- Das auszuführende **Programm** wird spezifiziert durch:
  - das Wissen über ein Problem und die Annahmen, die hinreichend für die Lösung sind $\equiv$ **logische Axiomen**;
  - eine zu beweisende Aussage (**Ziel**, **goal statement**) als das zu lösende Problem. ($\approx$ Eingabe)
- Die **Ausführung**:
  - ist der Versuch das goal statement zu beweisen unter den gegebenen Annahmen.

# Hauptkonstrukte

Die Hauptkonstrukte der logischen Programmierung stammen aus der Logik:

- **Aussagen**: Fakten, Anfragen und Regeln
- **Terme** $\equiv$ die einzigen Datenstrukturen

# Fakten

- **Fakten** sind Aussagen, die Beziehungen zwischen Objekten definieren.
- Bsp.:

```
father(abraham,isaac).
```

  ... besagt, dass zwischen `abraham` und `isaac` die Beziehung `father` besteht.

- Eine Beziehung kann man als ein **Prädikat** auffassen: das Prädikat `father` gilt für `abraham` und `isaac`.

# Fakten

- Die *Plus*-Beziehung:

```
plus(0,0,0).  plus(0,1,1).  plus(0,2,2).  plus(0,3,3).
plus(1,0,1).  plus(1,1,2).  plus(1,2,3).  plus(1,3,4).
plus(2,0,2).  plus(2,1,3).  plus(2,2,4).  plus(2,3,5).
```

- Eine endliche Menge von Fakten ist das einfachste Programm.

# Anfragen

- **Anfragen** (*queries*) erlauben es zu testen, ob eine Beziehung zwischen Objekten besteht, und somit Informationen aus einem Programm abzufragen.

- Bsp.:

```
father(abraham,isaac)?
```

  ... fragt, ob die Beziehung `father` zwischen `abraham` und `isaac` besteht.

# Anfragen

- Um eine **Anfrage** für ein gegebenes Programm **zu beantworten**, muss man **bestimmen**, **ob** die **Anfrage** eine **logische Folge des Programms** (d.h. der spezifizierten Axiomen) laut der **Deduktionsregeln** ist.

- Die einfachste Deduktionsregel ist **Identität**:

  **aus** $P$ **folgt** $P$

  D.h. eine Anfrage ist die logische Folge eines identischen Fakten.

- Bsp.: `father(abraham,isaac)?` ist wahr, angenommen der Fakt `father(abraham,isaac).` Teil des Programms ist.

# Logische Variablen

- Statt nur `wahr` oder `falsch` als Antworten zu bekommen, möchte man manchmal auch Objekte mit einer bestimmten Eigenschaft herausfinden.

- Bsp.: *Wessen Vater ist Abraham?*
  - 1. Möglichkeit: Wiederholte Anfragen

    `father(abraham,lot)?`, `father(abraham,milcah)?`, ..., `father(abraham,isaac)?`, ...

    bis man die Antwort `wahr` erhält.

  - 2. Möglichkeit: Besser, benutze **Variablen** , um über unbekannte Objekte zu sprechen...

# Terme

- Die Objekte eines Programms werden als **Terme** spezifiziert. Induktive Definition:
  - **Atome:** Konstanten (z.B. `abraham`, `lot`, `0`, `1`) sind Terme.
  - **Variablen:** Variablen (z.B `X`, `Y`) sind Terme.
  - **Zusamengesetzte Terme:** Wenn $t_1$, $t_2$, ..., $t_n$ Terme sind, und $f$ ein Name ist, dann ist $f(t_1, t_2, \ldots, t_n)$ ein Term.
    - $f$ = **Funktor**
    - $t_1$ bis $t_n$ = **Argumente**
    - $n$ = **Stelligkeit** des Funktors
- Können als Bäume aufgefasst sein:
  - Blätter = Atome, Variablen
  - innere Knoten = Funktoren
- Syntaktische Konvention: Nur Variablennamen werden großgeschrieben.

# Terme

- Beispiele:
  - `name(john,smith)`
  - `name(X,Y)`
  - `person(name(john,smith),age(23))`
  - `person(name(X,smith),age(23))`
  - `person(Y,age(23))`
  - `node(node(leaf(a),leaf(b)),leaf(c))`
  - `s(0)`
- Terme sind die einzige Datenstruktur in logischen Programmen.

# Substitutionen und Instanzen

- Eine **Substitution** ist eine endliche Menge von Paaren der Form $X_i = t_i$, mit $X_i$ eine Variable, $t_i$ ein Term und:
  - $X_i \neq X_j \ \forall i \neq j$
  - $X_i$ tritt in keinem $t_j$ auf $\forall i, j$

  Bsp.: {X=isaac}

- Die **Anwendung einer Substitution** $\theta$ auf einen Term $A$, $A\ \theta$, ist der Term, den man erhält, indem man für jedes Paar $X = t$ in $\theta$ jedes Auftreten von $X$ durch $t$ ersetzt .

- $A$ ist eine **Instanz** von $B$, wenn eine Substitution $\theta$ existiert, so dass $A = B\ \theta$.
  - Bsp.: `father(abraham,isaac)` ist eine Instanz von `father(abraham,X)` unter der Substitution `{X=isaac}`.

# Variablen in Anfragen

- **Existentielle Anfragen** = Anfragen mit Variablen
  - Bsp.: `father(abraham,X)?`
  - Haben eine **existentielle** Interpretation:
    ***Existiert*** *eine Substitution, für welche die Anfrage eine logische Folge des Programms ist?*
- **Verallgemeinerung** als Deduktionsregel:

  **Eine existentielle Anfrage $P?$ ist eine Folge einer Instanz $P\ \theta$ für jedes $\theta$.**
  - Bsp.: `father(abraham,X)?` ist eine Folge von `father(abraham,isaac).`

# Variablen in Anfragen

- Bsp.: `father(abraham,X)?`
- Der Beweis einer Anfrage ist **konstruktiv**, d.h. falls die Anfrage mit *ja* beantwortet werden kann, wird eine Substitution ausgegeben, für die die Aussage aus dem Programm deduzierbar ist.
- *Antwort*: *yes*, $\{X \mapsto isaac\}$

# Variablen in Fakten

- Variablen in Fakten sind **universal quantifiziert**. Ein Fakt $p(t_1, t_2, \ldots, t_n)$ besagt, dass **für jedes** $X_1, \ldots, X_k$, mit $X_i$ eine Variable die im Fakt auftritt, $p(t_1, \ldots, t_n)$ wahr ist.
  - Bsp.: `likes(X,pomegranates)` besagt, dass für alle `X`, `X` Granatäpfel mag.

- **Instantiierung** als Deduktionsregel:

  **Aus einer universell quantifizierten Aussage $P$ folgt eine Instanz $P\ \theta$ für jede Substitution $\theta$.**

  - Bsp.: `likes(lot,pomegranates)` folgt aus dem Fakt `likes(X,pomegranates).`

# Konjunktive Anfragen

- Eine Aussage $p(t_1, t_2, \ldots, t_n)$ heißt auch **Ziel** (engl. *goal*).
- Eine Anfrage $p(t_1, t_2, \ldots, t_n)?$, die aus nur einem Ziel besteht heißt **einfach**.
- Eine Anfrage kann auch eine Konjunktion aus mehreren Zielen sein $\Longrightarrow$ **konjunktive Anfragen**.
- Eine konjunktive Anfrage ist die Folge eines Programms, wenn jedes Ziel eine Folge des Programms ist.
- Bsp. `father(abraham,isaac),male(lot)?`

# Konjunktive Anfragen mit Variablen

- Wenn eine Variable in verschiedenen Zielen in einer konjunktiven Anfrage erscheint, dann bezieht sie sich immer auf das selbe Objekt.
  - Bsp.: `father(haran,X),male(X)`
- Eine konjunktive Anfrage ist die Folge eines Programms, wenn jedes Ziel eine Folge des Programms ist, wobei jede Variable mit dem selben Term in verschiedenen Zielen ersetzt wird.

# Regeln

- **Regeln** erlauben, neue Beziehungen mit Hilfe existierender Beziehungen zu definieren.

- Regeln sind Aussagen der Form:

$$A \leftarrow B_1, B_2, \ldots, B_n$$

  - $A$ heißt **Kopf** der Regel.
  - Die Konjunktion von Zielen $B_1, B_2, \ldots, B_n$ heißt **Rumpf** der Regel.
  - Fakten sind Regeln im Spezialfall $n = 0$. Im Allgemeinen ist ein Programm eine endliche Menge von Regeln (statt eine Menge von Fakten).

- Fakten, Anfragen und Regeln heißen auch **(Horn) Klauseln**.

# Regeln

- Variablen in Regeln sind (wie in Fakten) universell quantifiziert.
  - `son(X,Y) ← father(Y,X), male(X).`
  - `daughter(X,Y) ← father(Y,X), female(X).`
  - `grandfather(X,Y) ← father(X,Z), father(Z,Y).`
- "←" stellt logische Implikation dar
  $\Longrightarrow$**Modus ponens** als Deduktionsregel:

  Aus $R = (A \leftarrow B_1, B_2, \ldots, B_n)$ und $B'_1, B'_2, \ldots, B'_n$ folgt $A'$ wenn $A' \leftarrow B'_1, B'_2, \ldots, B'_n$ eine Instanz von $A \leftarrow B_1, B_2, \ldots, B_n$ ist.

  - Identität und Instantiierung sind Spezialfälle des Modus ponens.

# Prozedurale vs. logische Interpretation der Regeln

Bsp.: `grandfather(X,Y)` ← `father(X,Z), father(Z,Y).`

- Prozedurale I.: *Um die Anfrage "ist X der Großvater von Y?" zu beantworten, beantworte die konjunktive Anfrage "ist X der Vater von Z und Z der Vater von Y?".*

- Logische I.: *Für alle X, Y und Z, X ist der Großvater von Y, wenn X der Vater von Z und Z der Vater von Y ist.*

# Logische Programme als deduktive Datenbanken

- Außer Relationen enthält eine **deduktive/logische Datenbank** Regeln, die erlauben, neue Relationen aus bestehenden Relationen zu gewinnen.
  $\Longrightarrow$ Logische Programme können als deduktive Datenbanken betrachtet werden.
  - Grundrelationen werden via Fakten spezifiziert.
  - Regeln entsprechen der logischen Regeln.

# Beispiele

- Als Grundrelationen nehmen wir `father/2`, `mother/2`, `male/1` und `female/1` an, wobei die Zahl die Stelligkeit der jeweiligen Relation angibt.

- Wir definieren neue Relationen mit Hilfe der bestehenden Relationen via Regeln:
  - `parent(Parent,Child)` ← `father(Parent,Child).`
    `parent(Parent,Child)` ← `mother(Parent,Child).`
    (Mehrere Regeln mit dem selben Kopf definieren eine Disjunktion.)

  - `procreated(Man,Woman)` ←
    `father(Man,Child), mother(Woman,Child).`

# Beispiele

```
brother(Brother,Sib) ←
   parent(Parent,Brother),parent(Parent,Sib),male(Brother).
```

- Problem: `brother(X,X)?` gilt für jedes männliche Kind `X`.
- Lösung: Wir nehmen an, es existiert ein vordefiniertes Prädikat `≠(Term1,Term2)`, das wahr ist, wenn `Term1` verschieden von `Term2` ist. Wir schreiben das Prädikat infixiert: `Term1 ≠ Term2`.

$\Longrightarrow$

```
brother(Brother,Sib) ←
   parent(Parent,Brother),parent(Parent,Sib),
   male(Brother),Brother≠Sib.
```

# Beispiele

- `uncle(Uncle,Person)` $\leftarrow$
  `brother(Uncle,Parent),parent(Parent,Person)`

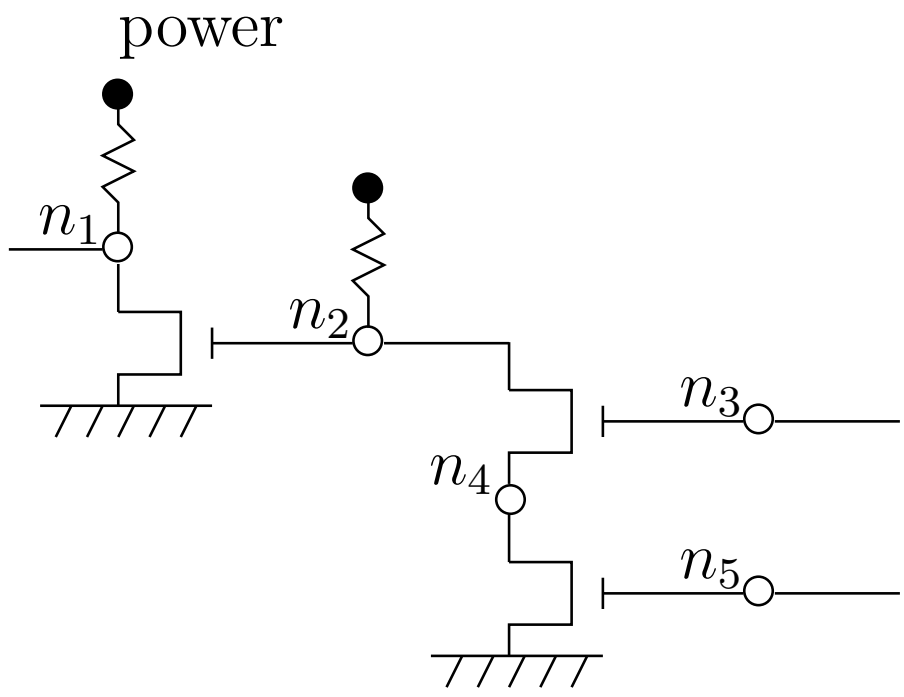
- `sibling(Sib1,Sib2)` $\leftarrow$
  `parent(Parent,Sib1),parent(Parent,Sib2),Sib1`$\neq$`Sib2.`

- `cousin(Cousin1,Cousin2)` $\leftarrow$
  `parent(Parent1,Cousin1),parent(Parent2,Cousin2),`
  `sibling(Parent1,Parent2).`

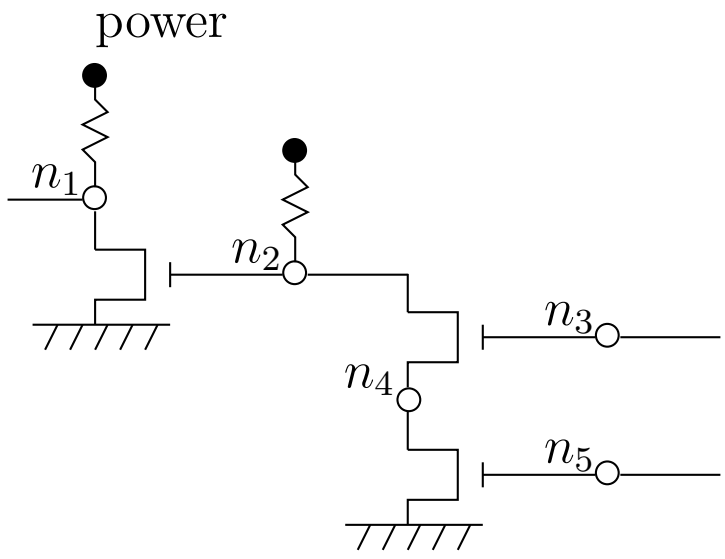
# Beispielanfragen

- Sind Hanah und Isaac Geschwister: `sibling(hanah,isaac)?`
- Wer ist der Onkel von Hanah: `uncle(X,hanah)?`
- Welche Geschwisterpaare gibt es: `sibling(X,Y)?`

# Beispiel



# Beispiel




```
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).
```

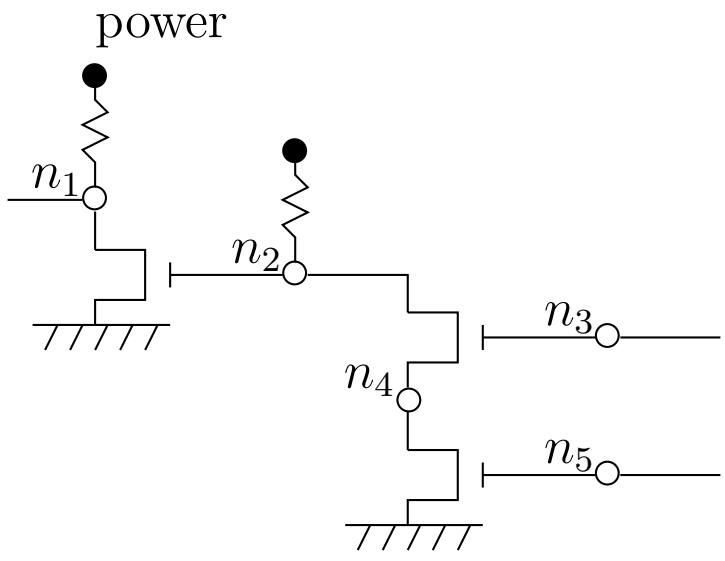
# Beispiel

- ```
  not(Input,Output) ←
      transistor(Input,ground,Output),
      resistor(power,Output).
  ```

- ```
  nand(Input1,Input2,Output) ←
      transistor(Input1,X,Output),
      transistor(Input2,ground,X),
      resistor(power,Output).
  ```

- ```
  and(Input1,Input2,Output) ←
      nand(Input1,Input2,X),
      not(X,Output).
  ```
  ```

# Beispielanfragen




```
and(In1,In2,Out)?
⟹{In1=n3,In2=n5,Out=n1}
```

# Strukturierte Daten und Datenabstraktion

- Flache Darstellung einer Vorlesung:

```
course(compilerbau,montag,9,11,helmut,seidl,MW,1001).
```

  $\longrightarrow$ Kompakte aber unübersichtliche Darstellung

- Strukturierte D.:

```
course( compilerbau,
        zeit(montag,9,11),
        lecturer(helmut,seidl),
        room(MW,1001)).
```

  $\longrightarrow$ Abstraktion der für die jeweiligen Ziele unwesentlichen Details, z.B.:

```
lecturer(Lecturer,Course) ← course(Course,Time,Lecturer,Loc).
teaches(Lecturer,Day)← course(Course,zeit(Day,S,F),Lecturer,Loc).
```

# Rekursive Regeln

```
ancestor(Ancestor,Descendant) ←
     parent(Ancestor,Descendant)

ancestor(Ancestor,Descendant) ←
     parent(Ancestor,Person), ancestor(Person,Descendant)
```

# Logische Programme zur Bearbeitung rekursiver Datenstrukturen

Bäume:

- ```
  bintree(void).
  bintree(tree(Element,Left,Right)) ←
     bintree(Left), bintree(Right).
  ```

- ```
  member(X,tree(X,Left,Right)).
  member(X,tree(Y,Left,Right)) ← member(X,Left).
  member(X,tree(Y,Left,Right)) ← member(X,Right).
  ```

# Bsp.: Baumisomorphie

- Zwei Bäume $t_1$ und $t_2$ sind isomorph, wenn $t_2$ durch eine Umordnung der Zweige in $t_1$ erhalten werden kann.
- Bäume in denen die Reihenfolge der Söhne unwichtig ist = ungeordnete Bäume

```
isotree(void,void).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←
  isotree(Left1,Left2), isotree(Right1,Right2).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←
  isotree(Left1,Right2), isotree(Right1,Left2).
```

# Bsp.: Substitution in Bäumen

- Problem: Ersetze im Baum $T_1$ alle Vorkommen von $X$ durch $Y$.

- Lösung: Die Eingabedaten und der Ergebnisbaum $T_2$ sind Argumente eines Prädikats, `substitute(X,Y,`$T_1$`,`$T_2$`)`, definiert wie folgt:

```
substitute(X,Y,void,void).
substitute(X,Y,tree(Info,Left,Right),
           tree(Info1,Left1,Right1)) ←
  replace(X,Y,Info,Info1),
  substitute(X,Y,Left,Left1),
  substitute(X,Y,Right,Right1).

replace(X,Y,X,Y).
replace(X,Y,Z,Z) ← X ≠ Z.
```

# Listen

- Listen sind ein Spezialfall von Binärbäumen und können syntaktisch definiert werden mit zwei Konstrukten:
  - **Leere Liste**: `[]`
  - **Nicht-leere Liste**: : `[X|Y]`
    - `X` = das erste Element
    - `Y` = der Rest der Liste.
- Syntaktische Konvention: `[a|[b|[c|[]]]]` ≡ `[a,b,c]`

# Bsp.: member

```
member(X,[X|Xs]).
member(X,[Y|Ys]) ← member(X,Ys).
```

# Bsp.: prefix, suffix, suli

```
prefix([],Ys).
prefix([X|Xs],[X|Ys]) ← prefix(Xs,Ys).

suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) ← suffix(Xs,Ys).

suli(Xs,Ys) ← prefix(Ps,Ys), suffix(Xs,Ps).
```

# Bsp.: append

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
```

- `append([a,b,c],[d,e],Xs)?` ergibt *Xs=[a,b,c,d,e]*.
- `append(Xs,[d,e],[a,b,c,d,e])?` ergibt *Xs=[a,b]*.
- `append(As,Bs,[a,b,c,d])?` ergibt die verschiedenen möglichen Aufspaltungen der Liste `[a,b,c,d]`.
  $\longrightarrow$ `append` kann benutzt werden, um Listen aufzuspalten...

# Bsp.: append zum Listenaufspalten

```
prefix(Xs,Ys) ← append(Xs,As,Ys).

suffix(Xs,Ys) ← append(As,Xs,Ys).

member(X,Ys) ← append(As,[X|Xs],Ys).

last(X,Xs) ← append(As,[X],Xs).
```

# Bsp.: Listen Umdrehen

- Eine mögliche Lösung:

```
reverse([],[]).
reverse([X|Xs],Zs)← reverse(Xs,Ys),append(Ys,[X],Zs).
```

- Andere Lösung:

```
reverse(Xs,Ys) ← reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys) ← reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

- Um die zwei Lösungen bezüglich der Effizienz vergleichen zu können, müssen wir den zugrunde liegenden Berechnungsmodell kennen.
  (☞ Nächster Abschnitt: Berechnungsmodell der logischen Programme)
  Im Bsp., Anzahl von Deduktionschritten:
  - Erste Lösung: quadratisch
  - Zweite Lösung: linear

# Unifikation

Grundoperation des Berechnungsmodells ist Unifikation
(☞ Typ-Inferenz). Zur Erinnerung

- **Unifikation** = Lösen von Systemen von Term-Gleichungen
  - Lösung = **Substitution** (Unifikator) der Variablen, die die Terme **strukturell gleich** machen.
  - Bsp.:
    - `app([a|[b]],[c|[d]],Ls) = app([X|Xs],Ys,[X|Zs])`
    - Lösung:
      `{X` $\mapsto$ `a, Xs` $\mapsto$ `b, Ys` $\mapsto$ `[c|[d]], Ls` $\mapsto$ `[a|Zs]}`

# Unifikation

- Satz: Ein System von Term-Gleichungen hat entweder:
  - **keine** Lösung
  - **eine eindeutige** Lösung
  - **beliebig viele** Lösungen. In diesem Fall gibt es eine allgemeinste Lösung (**mgu**).

# Algorithmus zur Berechnung des mgu

$\theta$ := $\emptyset$; push(stack, $T_1 = T_2$); failure = false;
while not empty(stack) and not failure do
  hole X = Y vom stack runter
  case
    **X ist eine Variable, die in Y nicht auftritt:**
      ersetze X durch Y im stack und in $\theta$
      füge $X \mapsto Y$ zu $\theta$ hinzu
    **Y ist eine Variable, die in X nicht auftritt:**
      ersetze Y durch X im stack und in $\theta$
      füge $Y \mapsto X$ zu $\theta$ hinzu
    **X und Y sind identische Konstanten oder Variablen:** continue
    **X ist $f(X_1, \ldots, X_n)$ und Y ist $f(Y_1, \ldots, Y_n)$ mit $f$=Funktor:**
      push(stack, $X_1 = Y_1, X_2 = Y_2, \ldots, X_n = Y_n$)
    **sonst:**
      failure := true
if failure then output *failure* else output $\theta$

# Der *occurs check* Test

```
θ := ∅; push(stack, T1 = T2); failure = false;
while not empty(stack) and not failure do
  hole X = Y vom stack runter
  case
    ....
    X ist eine Variable, die in Y nicht auftritt:
      ersetze Y durch X im stack und in θ
      füge X = Y zu θ hinzu
    ....

if failure then output failure else output θ
```

- ...stellt sicher, dass die Unifikation terminiert; Z.B. gibt es keine endliche gemeinsame Instanz von `X` und `[1,X]`;
- wird aus Effizienzgründen in Implementierungen wie Prolog weggelassen.

# *mgu*-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

**0.** Initialisierung:
$s = [$`append([a|[b]],[c|[d]],Ls)` $=$ `append([X|Xs],Ys,[X|Zs])`$]$;
$\theta = \{\}$

# *mgu*-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

**0.** Initialisierung:
$s = [$`append([a|[b]],[c|[d]],Ls)` $=$ `append([X|Xs],Ys,[X|Zs])`$]$;
$\theta = \{\}$

**1.** $s = [$`[a|[b]]` $=$ `[X|Xs]`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$

# *mgu*-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

**0.** Initialisierung:
$s = [\texttt{append([a|[b]],[c|[d]],Ls)} = \texttt{append([X|Xs],Ys,[X|Zs])}]$;
$\theta = \{\}$

**1.** $s = [\texttt{[a|[b]]} = \texttt{[X|Xs]}, \texttt{[c|[d]]} = \texttt{Ys}, \texttt{Ls} = \texttt{[X|Zs]}]$; $\theta = \{\}$

**2.** $s = [\texttt{a} = \texttt{X}, \texttt{[b]} = \texttt{Xs}, \texttt{[c|[d]]} = \texttt{Ys}, \texttt{Ls} = \texttt{[X|Zs]}]$; $\theta = \{\}$

# *mgu*-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

**0.** Initialisierung:
$s = [$`append([a|[b]],[c|[d]],Ls)` $=$ `append([X|Xs],Ys,[X|Zs])`$]$;
$\theta = \{\}$

**1.** $s =$ `[[a|[b]]` $=$ `[X|Xs]`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$

**2.** $s =$ `[a` $=$ `X`, `[b]` $=$ `Xs`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$

**3.** $s =$ `[[b]` $=$ `Xs`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[a|Zs]`$]$; $\theta = \{$`X=a`$\}$

# *mgu*-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

**0.** Initialisierung:
$s = [$`append([a|[b]],[c|[d]],Ls)` $=$ `append([X|Xs],Ys,[X|Zs])`$]$;
$\theta = \{\}$

**1.** $s =$ `[[a|[b]]` $=$ `[X|Xs]`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$

**2.** $s =$ `[a` $=$ `X`, `[b]` $=$ `Xs`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$

**3.** $s =$ `[[b]` $=$ `Xs`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[a|Zs]`$]$; $\theta =$ {`X=a`}

**4.** $s =$ `[[c|[d]]` $=$ `Ys`, `Ls` $=$ `[a|Zs]`$]$; $\theta =$ {`X=a`, `Xs=[b]`}

# *mgu*-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

**0.** Initialisierung:
$s = [\texttt{append([a|[b]],[c|[d]],Ls)} = \texttt{append([X|Xs],Ys,[X|Zs])}]$;
$\theta = \{\}$

**1.** $s = [\texttt{[a|[b]]} = \texttt{[X|Xs]}, \texttt{[c|[d]]} = \texttt{Ys}, \texttt{Ls} = \texttt{[X|Zs]}]$; $\theta = \{\}$

**2.** $s = [\texttt{a} = \texttt{X}, \texttt{[b]} = \texttt{Xs}, \texttt{[c|[d]]} = \texttt{Ys}, \texttt{Ls} = \texttt{[X|Zs]}]$; $\theta = \{\}$

**3.** $s = [\texttt{[b]} = \texttt{Xs}, \texttt{[c|[d]]} = \texttt{Ys}, \texttt{Ls} = \texttt{[a|Zs]}]$; $\theta = \{\texttt{X=a}\}$

**4.** $s = [\texttt{[c|[d]]} = \texttt{Ys}, \texttt{Ls} = \texttt{[a|Zs]}]$; $\theta = \{\texttt{X=a}, \texttt{Xs=[b]}\}$

**5.** $s = [\texttt{Ls} = \texttt{[a|Zs]}]$; $\theta = \{\texttt{X=a}, \texttt{Xs=[b]}, \texttt{Ys=[c|[d]]}\}$

# *mgu*-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

**0.** Initialisierung:
$s = [$`append([a|[b]],[c|[d]],Ls)` $=$ `append([X|Xs],Ys,[X|Zs])`$]$;
$\theta = \{\}$

**1.** $s =$ `[[a|[b]]` $=$ `[X|Xs]`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$

**2.** $s =$ `[a` $=$ `X`, `[b]` $=$ `Xs`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$

**3.** $s =$ `[[b]` $=$ `Xs`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[a|Zs]`$]$; $\theta = \{$`X=a`$\}$

**4.** $s =$ `[[c|[d]]` $=$ `Ys`, `Ls` $=$ `[a|Zs]`$]$; $\theta = \{$`X=a`, `Xs=[b]`$\}$

**5.** $s =$ `[Ls` $=$ `[a|Zs]`$]$; $\theta = \{$`X=a`, `Xs=[b]`, `Ys=[c|[d]]`$\}$

**6.** $s = []$; $\theta = \{$`X=a`, `Xs=[b]`, `Ys=[c|[d]]`, `Ls=[a|Zs]`$\}$

# Logik als Berechnungsmodell

- Ein **logischer Kalkül** ist die Erweiterung logischer Formeln um den Ableitungsbegriff. Mittels eines Kalküls kann man auch die operationale Semantik einer Programmiersprache beschreiben, d.h. formal angeben, wie Berechnungen in der Programmiersprache erfolgen.

- Syntax unserer Logikprogrammiersprache (LP-Sprache):

| | | | |
|---|---|---|---|
| **Atomische Ziele:** | $A, B$ | $::=$ | $p(t_1, \ldots, t_n)$ |
| **Ziele:** | $G, H$ | $::=$ | $\top \mid \bot \mid A \mid G \wedge H$ |
| **Klauseln:** | $K$ | $::=$ | $A \leftarrow G$ |
| **Programme:** | $P$ | $::=$ | $\{K_1, \ldots, K_m\}$ |

# Bemerkungen

- Die obigen Klauseln (genannt **Horn- o. definite Klauseln**) sind Speziallfälle von Formeln der Prädikatenlogik erster Stufe.
- Wir schreiben $\wedge$ für logische Konjunktion.
- Das Ziel $\top$ heißt top/true/leeres Ziel (☞ Erfolg der Berechnung). Es gilt das **Identitätsgesetz**: $G \wedge \top \equiv G$.
- Das Ziel $\bot$ heißt bottom/false (☞ Scheitern der Berechnung). Es gilt das **Absorptiongesetz**: $G \wedge \bot \equiv \bot$

# Zustände

- Ein **Zustand** ist ein Paar $< G, \theta >$, wobei $G$ ein Ziel und $\theta$ eine Substitution ist. $G$ wird **Resolvente** genannt.
- Ein **Anfangszustand** ist ein Zustand der Form $< G, \epsilon >$, wobei $\epsilon$ die leere Substitution ist.
- Ein Zustand heißt **erfolgreicher Endzustand**, falls er von der Form $< \top, \theta >$ ist.
- Ein Zustand heißt **erfolgloser Endzustand**, falls er von der Form $< \bot, \epsilon >$ ist.

# Reduktionen

Eine **Reduktion** (ein Zustandsübergang, Ableitungsschritt) von einem Ausgangszustand $S$ zu einem Folgezustand $S'$ kann erfolgen, wenn bestimmte Reduktionsbedingungen erfüllt sind. Man stellt das als **Reduktionsregel** (Ableitungsregel, Inferenzregel) von der Form dar:

$$\frac{\begin{array}{l}\text{Bedingung 1}\\ \ldots\\ \text{Bedingung n}\end{array}}{S \mapsto S'}$$

# LP-Reduktionsregel

- **Entfalten**

$$\frac{(B \leftarrow H) \in P \qquad \beta \text{ ist allgemeinster Unifikator von } B \text{ und } A\ \theta}{< A \wedge G, \theta > \mapsto_{\mathit{Entfalten}} < H \wedge G, \theta\ \beta >}$$

- **Scheitern**

$$\frac{\text{Es gibt keine Klausel } (B \leftarrow H) \in P, \text{ so dass ein Unifikator von } B \text{ und } A\ \theta \text{ existiert}}{< A \wedge G, \theta > \mapsto_{\mathit{Scheitern}} < \bot, \epsilon >}$$

# LP-Kalkül

- Eine **Berechnung** (engl. *computation*) ist eine Sequenz $S_0 \mapsto S_1 \mapsto \ldots \mapsto S_n$ von Reduktionen.
- Eine **Ableitung** (engl. *derivation*) ist eine Berechnung, die entweder in einem Endzustand endet oder unendlich ist.
- Eine Ableitung ist
  - **erfolgreich**, wenn ihr Endzustand erfolgreich ist;
  - **erfolglos**, wenn ihr Endzustand erfolglos ist;
  - **unendlich**, wenn sie keinen Endzustand hat.

# LP-Kalkül

- Ein Ziel $G$ ist
  - **erfolgreich**, wenn es eine erfolgreiche Ableitung beginnend mit $< G, \epsilon >$ gibt;
  - **erfolglos** (endlich gescheitert), wenn es nur erfolglose Ableitungen beginnend mit $< G, \epsilon >$ hat.
- Eine Substitution $\theta$ wird **Antwort eines Zieles** $G$ genannt, falls es eine erfolgreiche Ableitung $< G, \epsilon > \mapsto \ldots \mapsto < \top, \beta >$ gibt, so dass $\theta$ die eingeschränkte Substitution von $\beta$ auf die Variablen von $G$ ist.

# Eine Implementierung des LP-Kalküls

**Input:** Ein Ziel $G$ und ein Programm $P$

**Output:** Eine berechnete Antwort des Zieles G , wenn es eine gibt; *no*, sonst

Resolvente := G; $\theta := \epsilon$;

while Resolvente $\neq \top$
  sei Resolvente $= C_1 \wedge \cdots \wedge C_i \wedge A \wedge C_{i+1} \wedge \cdots \wedge C_m$
  if $\exists$ eine (umbenannte) Klausel $(A' \leftarrow B_1 \wedge \cdots \wedge B_n) \in P$ ,
    so dass $\beta$ der mgu von $A$ und $A'$ ist
  then Resolvente := $(C_1 \wedge \ldots C_i \wedge B_1 \wedge \cdots \wedge B_n \wedge C_{i+1} \wedge \cdots \wedge C_m)\,\beta$
       $\theta := \theta\,\beta$
  else break

if Resolvente $= \top$
  then output $\theta$
  else output *no*

# Nichtdeterminismus im LP-Kalkül

- In unserem **LP-Kalkül ist** die Auswahl der Klausel innerhalb eines Programms und die Auswahl des atomischen Ziels $A$ aus der Resolventen **nicht deterministisch**.

- Eine LP-Sprache (z.B. Prolog) muss den Nichtdeterminismus nach einem Schema (*scheduling policy*) auflösen.
  - Die Selektion des atomischen Ziels $A$ kann (nur) die Länge der Ableitung beeinflußen (im schlimmsten Fall unendlich).
  - Die Selektion der Klausel kann über Erfolg oder Scheitern und über die berechnete Antwort entscheiden.

# LP-Kalkül: Beispiel

Programm:
```
append([],Ys,Ys) ← ⊤.                          (1)
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).   (2)
```

Ziel:
```
append([a,b],[c,d],Ls)
```

Berechnung:

$< \texttt{append([a,b],[c,d],Ls)}, \epsilon >$

# LP-Kalkül: Beispiel

Programm: `append([],Ys,Ys)` $\leftarrow \top$. (1)

`append([X|Xs],Ys,[X|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`. (2)

Ziel: `append([a,b],[c,d],Ls)`

Berechnung:

$<$ `append([a,b],[c,d],Ls)`$, \epsilon >$

$\mapsto_{\textit{Entfalten}(2)}$ $<$ `append(Xs,Ys,Zs)`$, \{$`X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs]`$\} >$

# LP-Kalkül: Beispiel

Programm: `append([],Ys,Ys)` $\leftarrow \top$. (1)

`append([X|Xs],Ys,[X|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`. (2)

Ziel: `append([a,b],[c,d],Ls)`

Berechnung:

$<$ `append([a,b],[c,d],Ls)`, $\epsilon >$

$\mapsto_{\textit{Entfalten}(2)}$ $<$ `append(Xs,Ys,Zs)`, {`X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs]`} $>$

$\mapsto_{\textit{Entfalten}(2)}$ $<$ `append(Xs1,Ys1,Zs1)`,
{`X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs],X1=b,Xs1=[],Ys1=[c,d],Zs=[b|Zs1]`} $>$

# LP-Kalkül: Beispiel

Programm: `append([],Ys,Ys)` $\leftarrow \top$. (1)

`append([X|Xs],Ys,[X|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`. (2)

Ziel: `append([a,b],[c,d],Ls)`

Berechnung:

$<$ `append([a,b],[c,d],Ls)`, $\epsilon >$

$\mapsto_{\textit{Entfalten}(2)}$ $<$ `append(Xs,Ys,Zs)`, {`X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs]`} $>$

$\mapsto_{\textit{Entfalten}(2)}$ $<$ `append(Xs1,Ys1,Zs1)`,
{`X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs],X1=b,Xs1=[],Ys1=[c,d],Zs=[b|Zs1]`} $>$

$\mapsto_{\textit{Entfalten}(1)}$ $< \top$,
{`X=a,Xs=[b],Ys=[c,d],Ls=[a,b,c,d],X1=b,Xs1=[],Ys1=[c,d],Zs=[b,c,d],`
`Ys2=[c,d],Zs1=[c,d]`} $>$

# LP-Kalkül: Beispiel

Programm: `append([],Ys,Ys)` $\leftarrow \top$. (1)

`append([X|Xs],Ys,[X|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`. (2)

Ziel: `append([a,b],[c,d],Ls)`

Berechnung:

$<$ `append([a,b],[c,d],Ls)`, $\epsilon >$

$\mapsto_{\textit{Entfalten}(2)}$ $<$ `append(Xs,Ys,Zs)`, {`X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs]`} $>$

$\mapsto_{\textit{Entfalten}(2)}$ $<$ `append(Xs1,Ys1,Zs1)`,
{`X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs],X1=b,Xs1=[],Ys1=[c,d],Zs=[b|Zs1]`} $>$

$\mapsto_{\textit{Entfalten}(1)}$ $< \top$,
{`X=a,Xs=[b],Ys=[c,d],Ls=[a,b,c,d],X1=b,Xs1=[],Ys1=[c,d],Zs=[b,c,d],`
`Ys2=[c,d],Zs1=[c,d]`} $>$

$\Longrightarrow$ Antwort: {`Ls=[a,b,c,d]`}.

# Negation

- Manchmal ist es natürlich negative Bedingungen zu spezifizieren. Z.B.:

  ```
  bachelor(X) ← male(X), not married(X).
  ```

$\Longrightarrow$ Syntax und Semantik der LP muss erweitert werden

- Syntax:

| | | | |
|---|---|---|---|
| **Atomische Ziele:** | $A, B$ | $::=$ | $p(t_1, \ldots, t_n)$ |
| **Ziele:** | $G, H$ | $::=$ | $\top \mid \bot \mid A \mid \neg A \mid G \wedge H$ |
| **Klauseln:** | $K$ | $::=$ | $A \leftarrow G$ |
| **Programme:** | $P$ | $::=$ | $\{K_1, \ldots, K_m\}$ |

# Negation durch Scheitern: Semantik

- Ein Ziel $\neg A$ ist erfolgreich genau dann, wenn das Ziel $A$ endlich scheitert.
- Diese Art der Negation wird **Negation durch Scheitern** genannt (*Negation as Failure, NaF*).

$\Longrightarrow$ Reduktionsregeln für negierte Ziele

# Reduktionsregeln für negierte Ziele

- **Scheitern NaF:** *es gibt eine erfolgreiche Ableitung von A*

$$\frac{< A, \theta > \mapsto^* < \top, \beta >}{< \neg A \wedge G, \theta > \mapsto < \bot, \epsilon >}$$

- **Erfolg NaF:** *es gibt keine erfolgreiche UND keine unendliche Ableitung von A*

$$\frac{\text{Jede Ableitung von } A \text{ scheitert endlich: } < A, \theta > \mapsto^* < \bot, \epsilon >}{< \neg A \wedge G, \theta > \mapsto < G, \theta >}$$

# Negation durch Scheitern: Bemerkungen

- NaF ist eine eingeschränkte Form der Negation aus der Prädikatenlogik erster Stufe.
- NaF führt unter Umständen zu semantischen Problemen in Zusammenhang mit Vollständigkeit und Terminierung.
- NaF zerstört die Eigenschaft des LP-Kalküls ohne Negation, dass erfolgreiche Ableitungen erhalten bleiben, wenn man dem Programm Klauseln hinzufügt. Z.B.:

  $P = \{$`p(a).`$\} \Longrightarrow$ `p(b)` scheitert. $\Longrightarrow \neg$`p(b)` ist erfolgreich.

  $P = \{$`p(a).`
  `p(b).`$\} \Longrightarrow$ `p(b)` ist erfolgreich. $\Longrightarrow \neg$`p(b)` scheitert.

# Prolog

- ... ist die bekannteste Implementierung einer LP-Sprache;
- wurde Anfang der 1970er von Alain Colmerauer (Marseille) und Robert Kowalski (Edinburgh) entwickelt.
- konkretisiert den vorgestellten LP-Kalkül zur Bearbeitung von Zielen durch:
  - Auflösung des Nichtdeterminismus der Auswahl von Klauseln und Atomen nach einem festen Schema;
  - Erlauben der Negation durch Scheitern nur für zur Laufzeit unter den aktuellen Substitutionen variablenfreie Ziele.

  $\Longrightarrow$ SLDNF-Resolution (Linear resolution with Selection function for Definite clauses with Negation as Failure)

# Auswahl von Klauseln und Atomen in Prolog

- In Prolog wird immer das am weitesten links stehende **Literal** (atomisches Ziel oder negiertes atomisches Ziel) eines Ziels selektiert und ganz entfaltet.

- Klauseln werden in textueller Reihenfolge ausgewählt.
  - Scheitert eine Ableitung mit einer bestimmten Klausel, versucht man eine neue Ableitung für das Literal mit Hilfe der nächsten Klausel. (Rücksetzen, **backtracking**)

  - Wird eine erfolgreiche Ableitung gefunden, werden weitere erfolgreiche Ableitungen gesucht, indem man immer das zuletzt gewählte Literal, bei dem noch Klauseln zur Auswahl stehen, erneut zu entfalten versucht.

- Die Auswahl der Klauseln bei der Suche durch Rücksetzen wird als **don't know Nichtdeterminismus** bezeichnet.

# Suchbäume

- Ein **Suchbaum** eines Zieles $G_{start}$ bezüglich eines Programms $P$ ist wie folgt definiert:
  - Knoten sind Ziele.
  - Die Wurzel des Baumes ist $G_{start}$.
  - Für jede anwendbare Entfalten-Reduktionsregel:

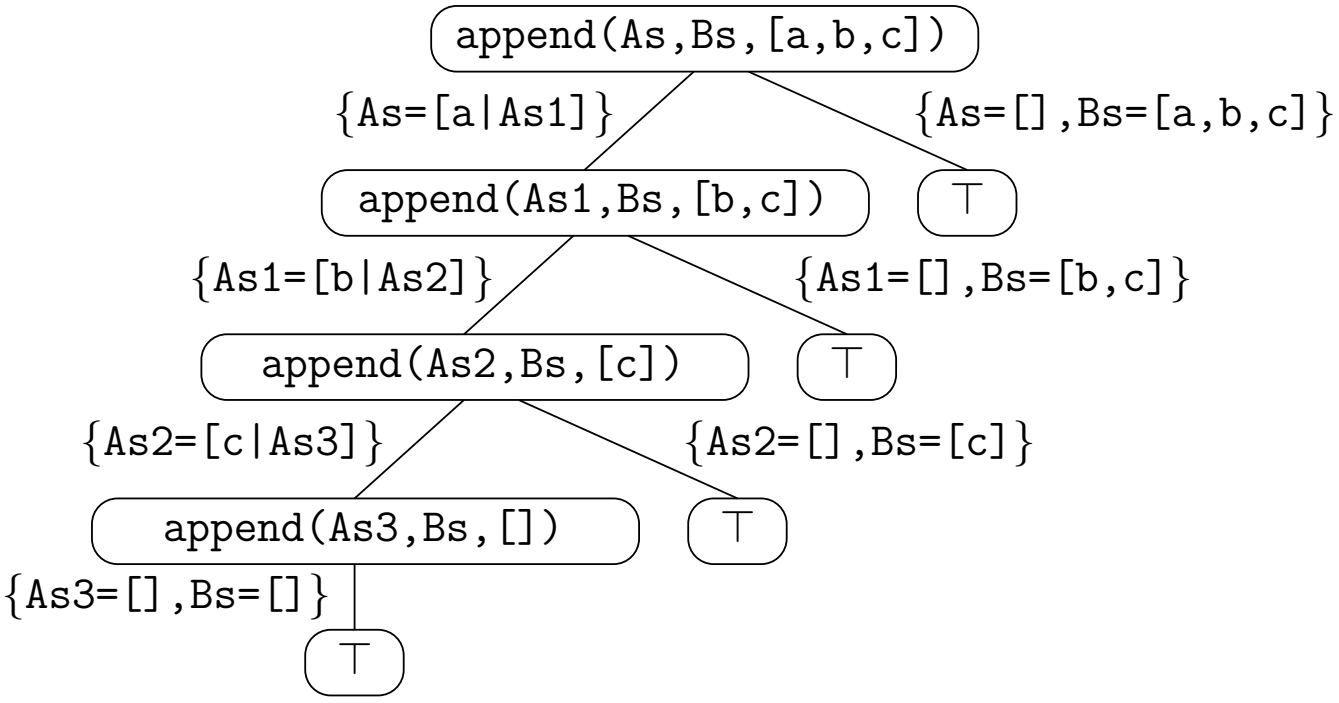
$$\frac{(B \leftarrow H) \in P \quad \beta \text{ ist allgemeinster Unifikator von } B \text{ und } A\theta}{< A \wedge G, \theta > \mapsto_{Entfalten} < H \wedge G, \theta\beta >}$$

    existiert eine mit $\beta$ beschrifteter Kante vom Knoten $A \wedge G$ zum Knoten $H \wedge G$.

- Durch die Auswahlstrategie von Prolog entspricht jedem Ziel genau ein Suchbaum.

# Suchbäume

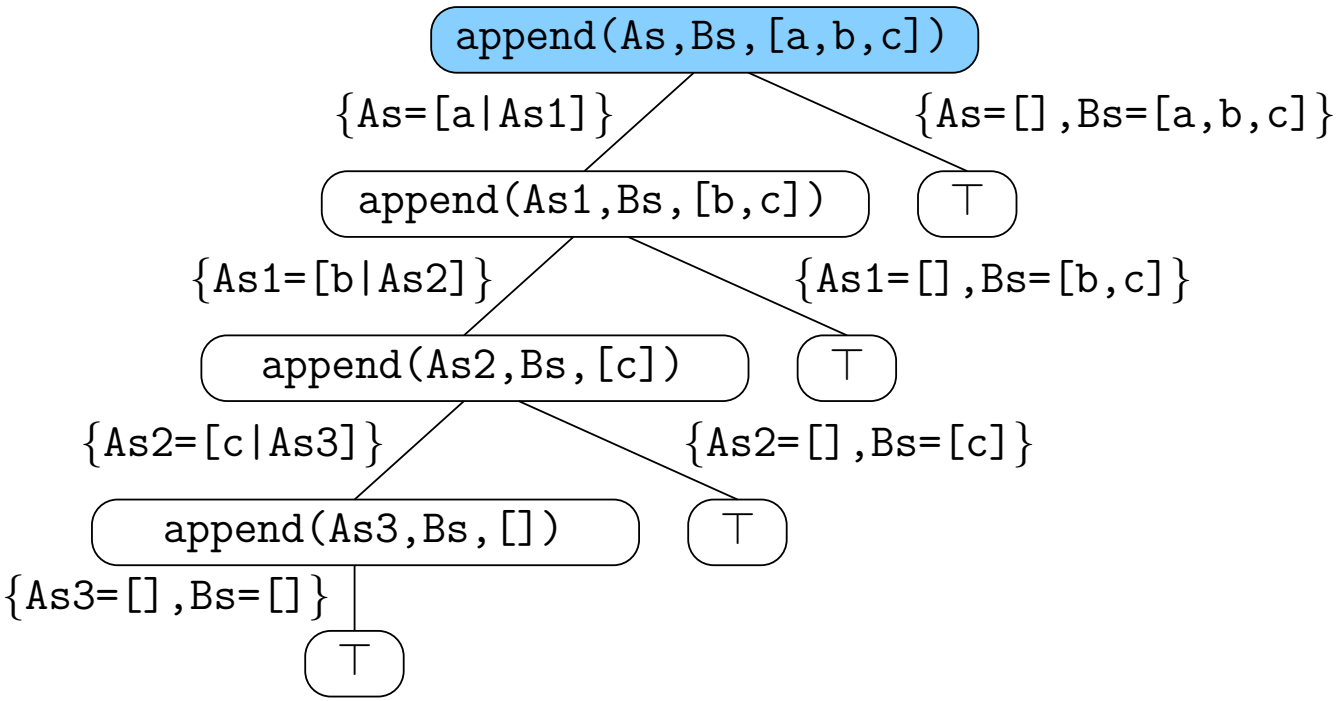
```
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
append([],Ys,Ys).
```



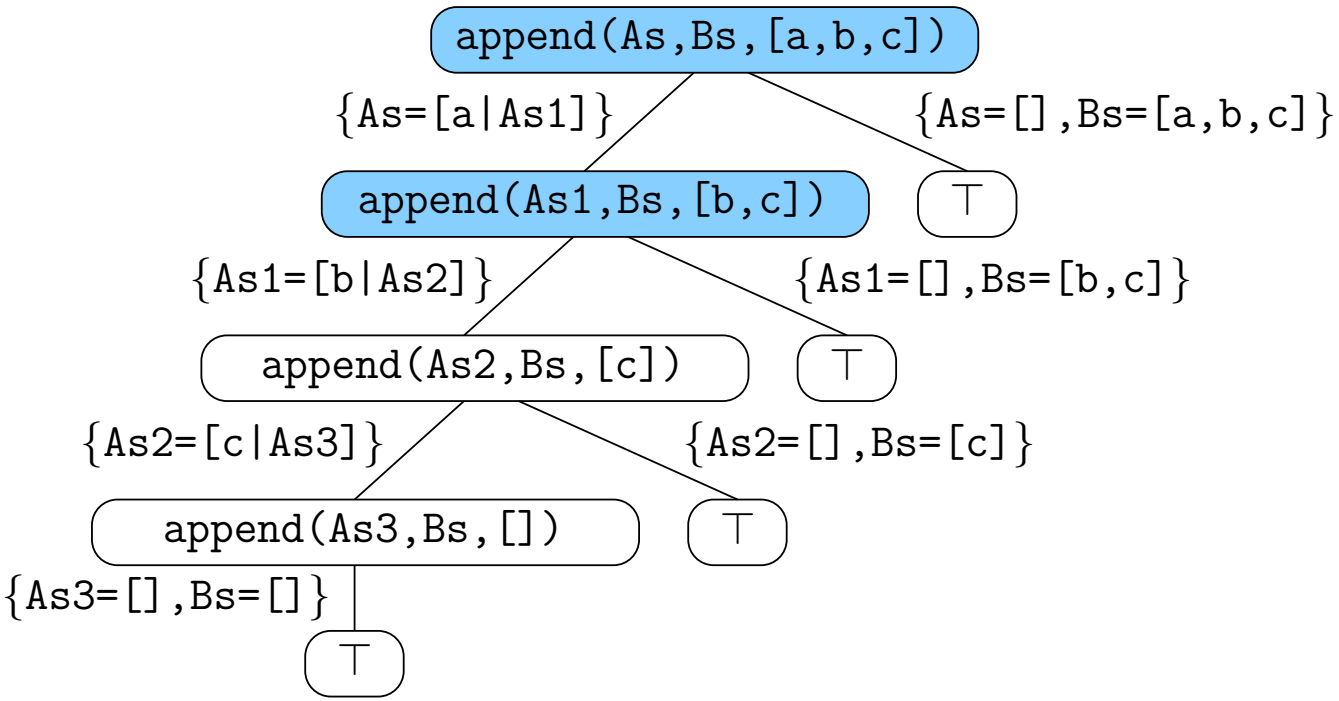
# Suchbäume

- Die Blätter eines Suchbaumes, wo das leere Ziel erreicht wird, heißen **Erfolgsknoten**.
- Der Pfad zu einem Erfolgsknoten entspricht der Berechnung einer Antwort.
- Erfolgsknoten entsprechen einer berechneten Antwort.
- Die übrigen Blätter heißen **Scheiternknoten**.
- Die Auswertung eines Zieles in Prolog entspricht einem Tiefendurchlauf über den entsprechenden Suchbaum. Wird ein Erfolgsknoten erreicht, so werden die entsprechenden Substitutionen für die Variablen in der Anfrage ausgegeben. Der Benutzer hat die Möglichkeit nach weiteren Lösungen suchen zu lassen.

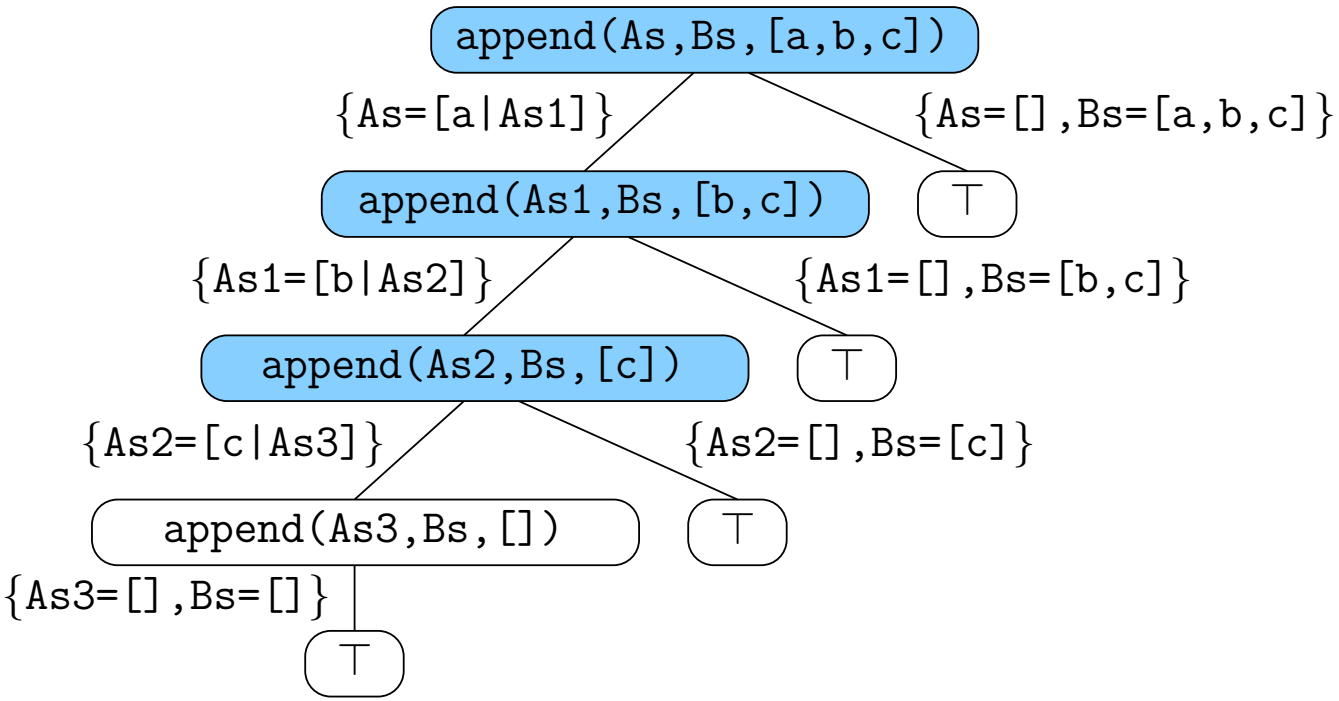
# Prolog's Suchstrategie



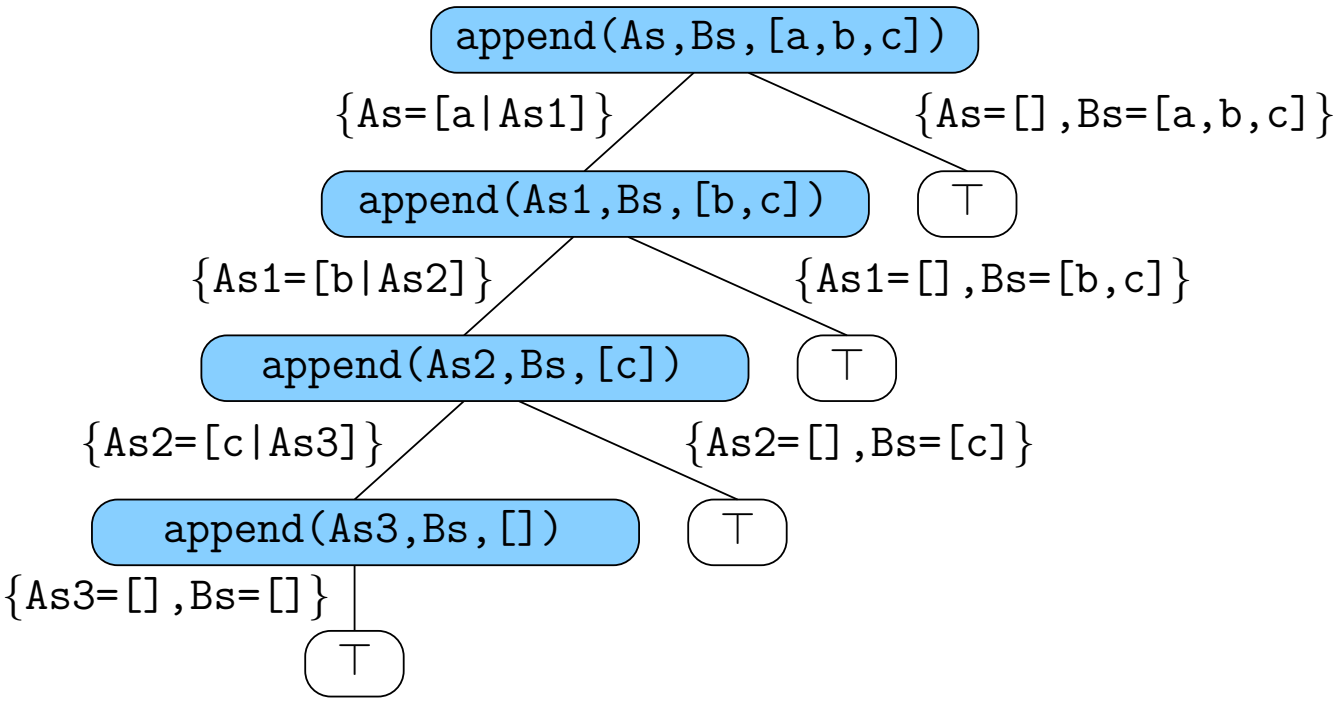
# Prolog-Suchbaumdurchläufe



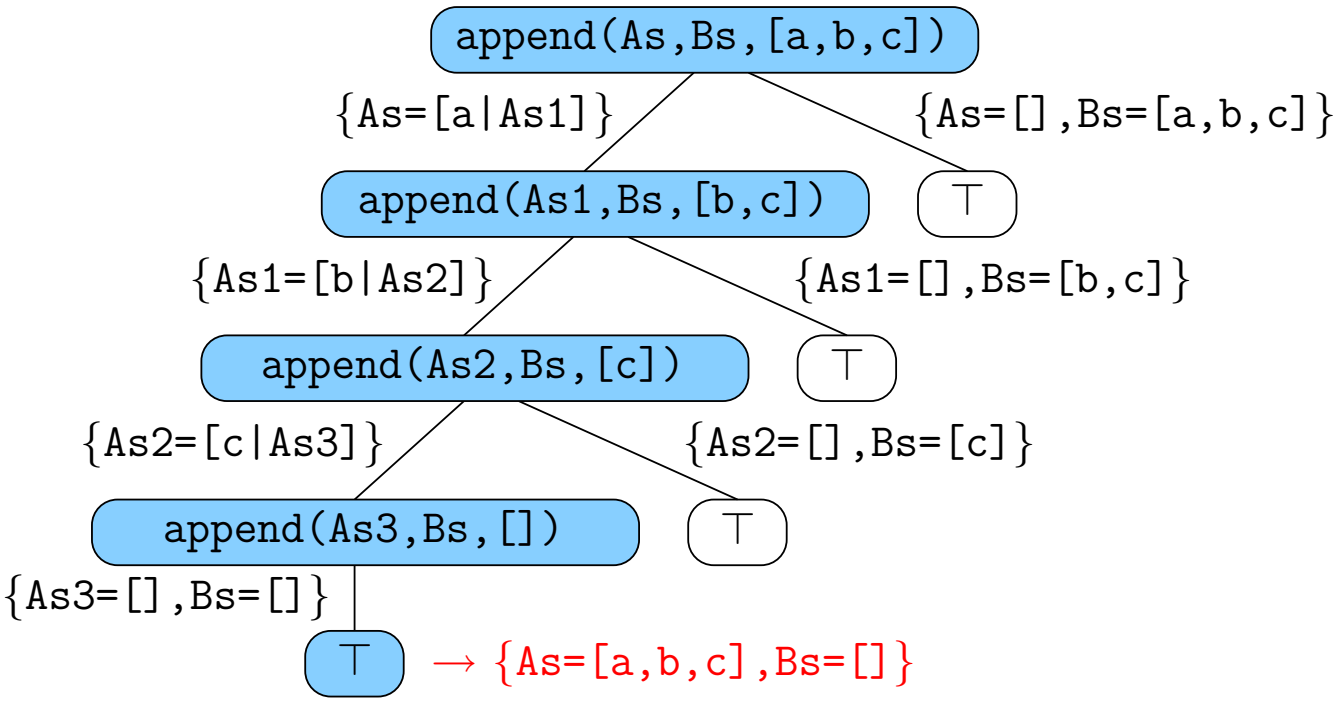
# Prolog-Suchbaumdurchläufe



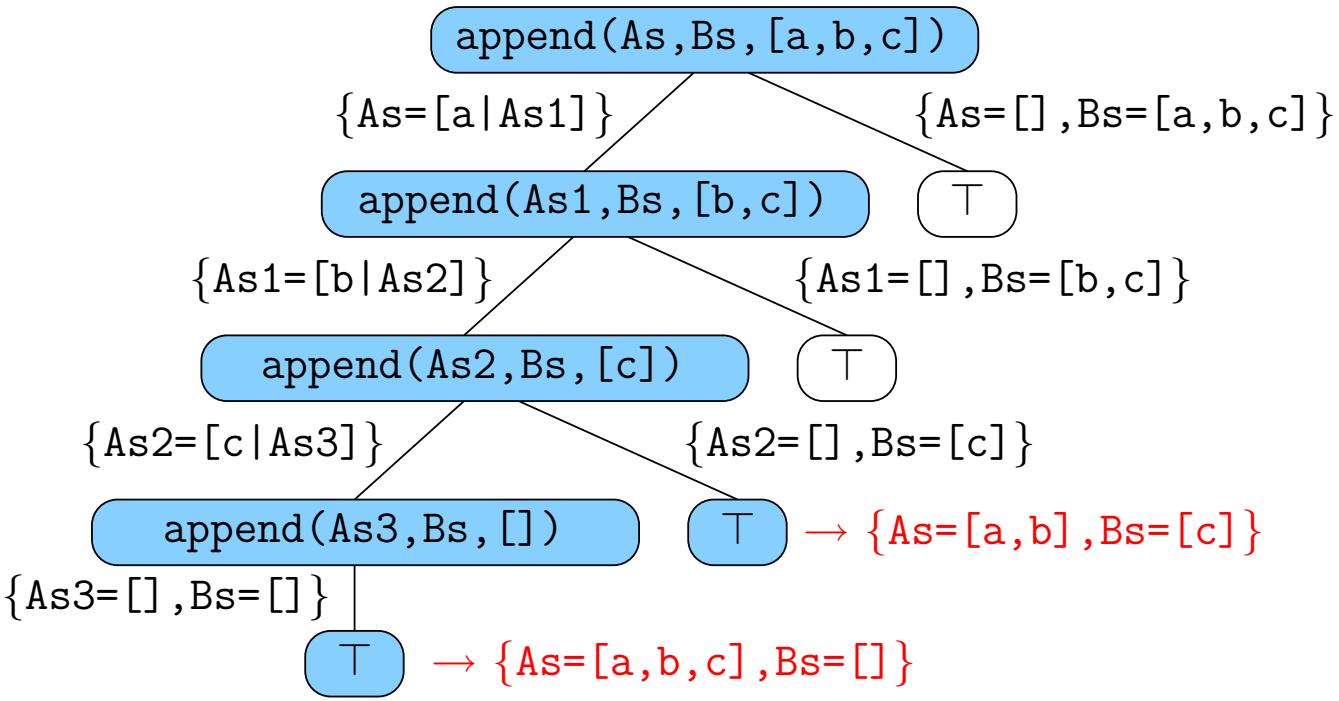
# Prolog-Suchbaumdurchläufe



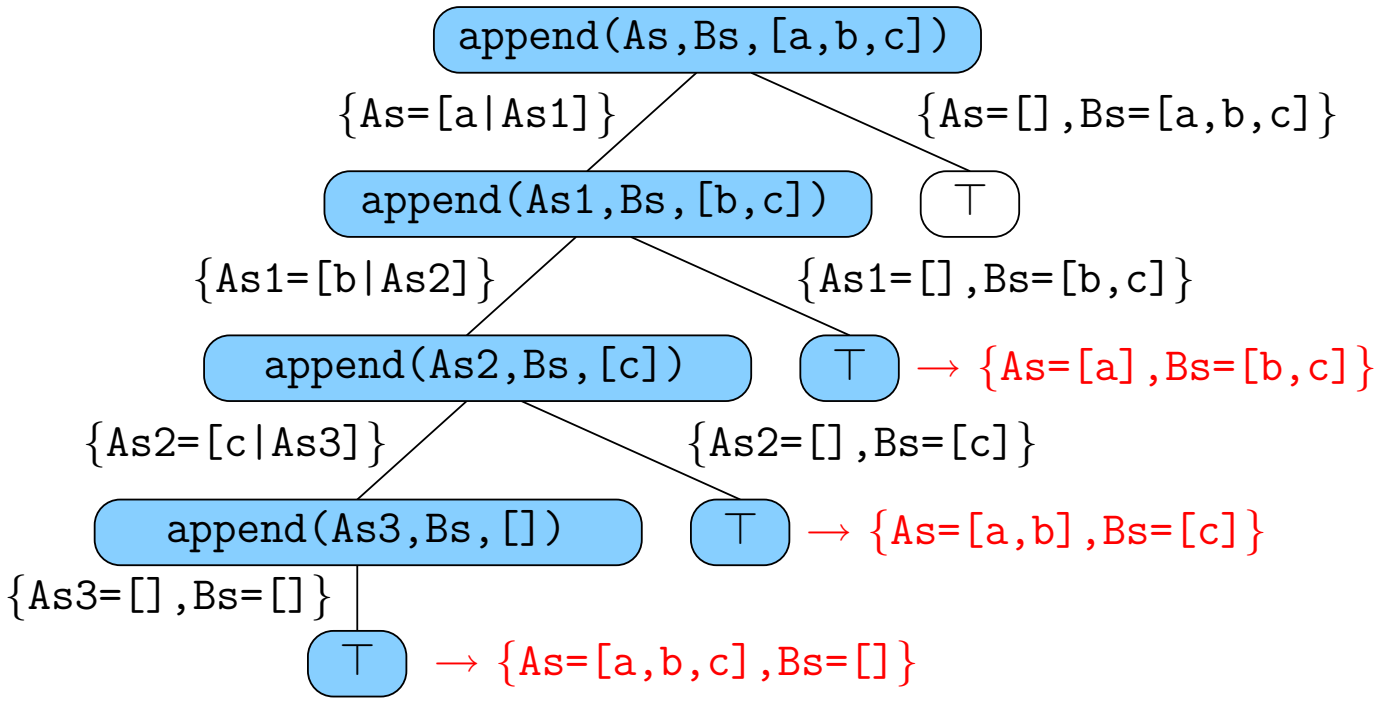
# Prolog-Suchbaumdurchläufe



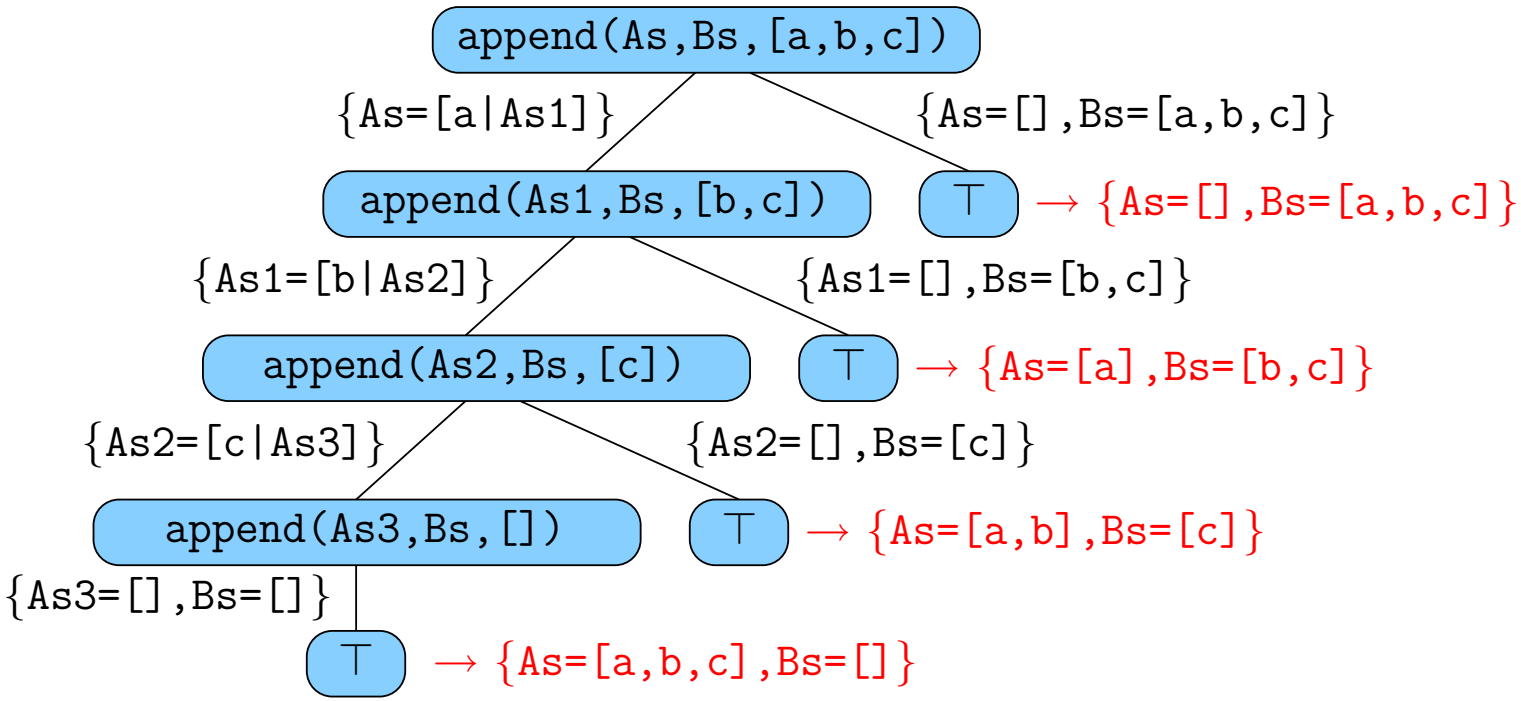
# Prolog-Suchbaumdurchläufe



# Prolog-Suchbaumdurchläufe



# Prolog-Suchbaumdurchläufe



# Folgen der Tiefensuche

- Die Tiefensuche ist effizient und einfach zu implementieren, aber...
- Unter Umständen ist der zuerst gefolgte Pfad unendlich, so dass andere eventuell existierende Erfolgsknoten nicht mehr erreicht werden können.
- Durch die Änderung der Reihenfolge der Klauseln und Literale kann erreicht werden, dass unendliche Berechnungen vermieden werden oder später auftretten.

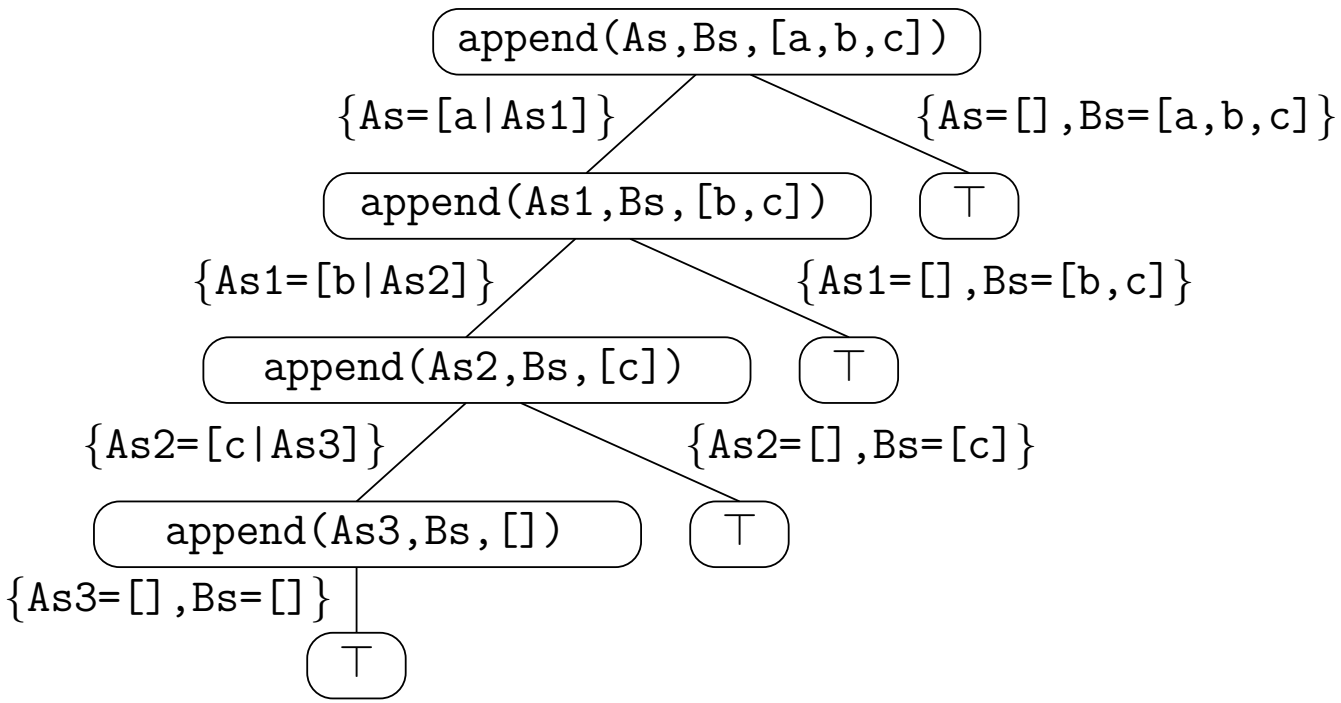
$\Longrightarrow$ Deklarativität der Logikprogrammierung wird (zugunsten der Effizienz) verletzt.

# Fazit

- Aus Sicht der LP-Programmierung ist die Reihenfolge der Klauseln und der Ziele irrelevant.
- Die Effizienz der Prolog-Programme allerdings hängt oft maßgeblich von dieser Reihenfolge ab.
- Im Extremfall (oft) terminieren korrekte LP-Programme nicht für eine bestimmte Anordnung der Klauseln und der Ziele.

# Reihenfolge in der Lösungen gefunden werden

```
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
append([],Ys,Ys).
```



# Reihenfolge in der Lösungen gefunden werden

```
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
append([],Ys,Ys).
```

```
?- append(As,Bs,[a,b,c]).
As = [a, b, c]
Bs = [] ;

As = [a, b]
Bs = [c] ;

As = [a]
Bs = [b, c] ;

As = []
Bs = [a, b, c] ;
No
```

# Reihenfolge in der Lösungen gefunden werden

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

```
?- append(As,Bs,[a,b,c]).
As = []
Bs = [a, b, c] ;

As = [a]
Bs = [b, c] ;

As = [a, b]
Bs = [c] ;

As = [a, b, c]
Bs = [] ;
No
```

# Terminierung

Rekursive Klauseln können zu Nichtterminierung führen.

Bsp.: 1. Versuch, eine kommutative Relation zu definieren.

```
married(X,Y) :- married(Y,X).
married(abraham,sarah).
```

```
?- married(abraham,sarah).
Nichtterminierende Berechnung
```

Grund:

```
married(abraham,sarah)
   married(sarah,abraham)
      married(abraham,sarah)
         married(sarah,abraham)
            ...
```

# Terminierung

Rekursive Klauseln können zu Nichtterminierung führen.

Bsp.: 2. Versuch, eine kommutative Relation zu definieren.

```
married(abraham,sarah).
married(X,Y) :- married(Y,X).
```

```
?- married(abraham,sarah).
Yes
?- married(sarah,abraham).
Yes
?- married(lot,sarah).
Nichtterminierende Berechnung
```

# Terminierung

Kommutative Relationen können mit einem neuen Prädikat definiert werden, das eine Klauseln für jede Permutation der Argumente der Relation hat:

```
are_married(X,Y) :- married(X,Y).
are_married(X,Y) :- married(Y,X).
married(abraham,sarah).
```

```
?- are_married(abraham,sarah).
Yes
?- are_married(sarah,abraham).
Yes
?- are_married(sarah,lot).
No
```

# Anordnungen der Literale

Die Anordnungen der Literale bestimmen den Prolog-Suchbaum.
(im Unterschied zur Anordnung der Klausel, die nur die Reihenfolge ändert, in der Teilbäume besucht werden sollen.)

- kann die Effizienz maßgeblich beeinflußen.

- kann bestimmen, ob eine Berechnung terminiert oder nicht.

# Anordnung der Literale und Effizienz

Bsp.: Berechnung für `son(X,lot)`

- 1. Möglichkeit: `son(X,Y) :- male(X),parent(Y,X).`
  $\longrightarrow$ man betrachtet die Männer nacheinander und prüft, ob sie Kinder von `lot` sind.

- 2. Möglichkeit: `son(X,Y) :- parent(Y,X),male(X).`
  $\longrightarrow$ man betrachtet die Kinder von `lot` nacheinander und prüft, ob sie Männer sind.

# Anordnung der Literale und Effizienz

Bsp.: Berechnung für `son(X,lot)`

- 1. Möglichkeit: `son(X,Y) :- male(X),parent(Y,X).`
  $\longrightarrow$ man betrachtet die Männer nacheinander und prüft, ob sie Kinder von `lot` sind.

- 2. Möglichkeit: `son(X,Y) :- parent(Y,X),male(X).`
  $\longrightarrow$ man betrachtet die Kinder von `lot` nacheinander und prüft, ob sie Männer sind.

$\Longrightarrow$ Die zweite Anordnung ist günstiger für die Anfrage `son(X,lot)` aber die erste ist besser für `son(sarah,X)`

$\Longrightarrow$ Die optimale Anordnung hängt von der beabsichtigten Benutzung ab.

# Anordnung der Literale und Effizienz

$\Longrightarrow$ Heuristik: Literale deren Ableitung effizient ist (z.B. arithmetische Teste), sollten möglichst links, vor anderen Literalen (insbesondere vor rekursiven) Atomen stehen.

Bsp.: Eine Prozedur `partition(Liste,Pivot,Kleinere,Groessere)` kann benutzt werden, um eine Liste in zwei Listen der Elemente, die kleiner bzw. größer als ein Pivot sind. (☞ Quicksort).

- Eine Klausel, die die Prozedur definiert könnte so aussehen:

```
partition([X|Xs],Y,[X|Ks],Gs) :- X<=Y,partition(Xs,Y,Ks,Gs)
```

- Diese führt i.A. zu effizienteren Berechnungen als:

```
partition([X|Xs],Y,[X|Ks],Gs) :- partition(Xs,Y,Ks,Gs),X<=Y
```

# Anordnung der Literale und Terminierung

Die Anordnung der Literale kann über Terminierung entscheidend sein.

```
quicksort([X|Xs],Ys) :-
        partition(Xs,X,Kleinere,Groessere),
        quicksort(Kleinere,Ls),
        quicksort(Groessere,Bs),
        append(Ls,[X|Bs],Ys).
```

$\Longrightarrow$ terminiert, weil die rekursive Sortierung auf die kleineren Listen `Kleinere` bzw. `Groessere` angewendet wird.

```
quicksort([X|Xs],Ys) :-
        quicksort(Kleinere,Ls),
        quicksort(Groessere,Bs),
        partition(Xs,X,Kleinere,Groessere),
        append(Ls,[X|Bs],Ys).
```

$\Longrightarrow$ terminiert nicht.

# Redundante Lösungen

Prolog gibt für jede erfolgreiche Ableitung eine Antwort aus. U.U. (wenn mehrere Klauseln für den selben Fall zuständig sind) kann eine Antwort sich wiederholen, z.B.:

```
append([],Ys,Ys).
append([X],Ys,[X|Ys]).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

```
?- append([1],[2,3],X).
X = [1, 2, 3] ;
X = [1, 2, 3] ;
No
```

# Systemprädikate

- **Systemprädikate** (*builtin Prädikate, bips*) sind Prädikate, die vom implementierenden System direkt unterstützt werden, statt mit Hilfe von Klauseln definiert zu sein.

$\Longrightarrow$ Effizienz, dafür Eingeschränkungen bezüglich ihrer Benutzung.

- **Arithmetische Systemprädikate** liefern Zugang zur effizienten, maschinenunterstützten arithmetischen Funktionalität.

# Auswertung arithmetischer Ausdrücke: das Prädikat `is`

Zur Evaluierung eines arithmetischen Ausdrucks benutzt man das infixierte Prädikat `is(Wert, Ausdruck)` d.h.: `Wert is Ausdruck`

- Prolog-Interpretierung des Zieles:
  1. Wenn `Ausdruck` unter der aktuellen Variablensubstitution zu einem Wert $v$ ausgewertet werden kann, liefert die Unifikation von $v$ und `Wert` das Ergebnis der Ableitung des Zieles.
  2. Sonst gibt es einen Laufzeitfehler.

- Beispiele:

| | |
|---|---|
| `X is 1+2` | Antwort: `X=3`. |
| `3 is 1+2` | Antwort: `yes`. |
| `1+2 is 1+2` | Antwort: `no`. (Grund: `1+2` und 3 unifizieren nicht.) |

# Auswertung arithmetischer Ausdrücke: das Prädikat `is`

Gründe warum ein `Ausdruck` in `Wert is Ausdruck` nicht auswertbar sein könnte:

- `Ausdruck` ist kein arithmetischer Ausdruck, z.B. `1+x`
  $\Longrightarrow$ **Das Ziel scheitert.** (*failure*)

- `Ausdruck` benutzt Variablen, die bei der Auswertung (noch) nicht belegt sind, z.B. `1+Y`, wenn noch keine Substitution von `Y` im Laufe der aktuellen Ableitung vorliegt.
  $\Longrightarrow$ **Laufzeitfehler** (*error condition*)

# Das Prädikat `is`

- Vorsicht: `is` dient nicht der Zuweisung eines Wertes an eine Variable.
- X is X+1 schlägt fehl oder führt zu einem Laufzeitfehler – immer.

# Arithmetische Vergleiche

- `1+2 <= 6-3`
  - Linke Seite wird ausgewertet $\longrightarrow$ 3;
  - Rechte Seite wird ausgewertet $\longrightarrow$ 3;
  - 3 und 3 unifizieren $\longrightarrow$ Antwort `yes`.
- Andere Vergleichsoperatoren: `>=`,`<`, `>`, `=:=` (Gleichheit), `=/=` (Ungleichheit).

# Arithmetik in Prolog: Beispiel

```
factorial(N,F) :-
            N>0, N1 is N-1, factorial(N1,F1), F is N*F1.
factorial(0,1).
```

```
?- factorial(3,X).
X = 6 ;
No

?- factorial(X,6).
ERROR: Arguments are not sufficiently instantiated
```

# Arithmetik in Prolog: Beispiel

- Alternative Implementierung der Fakultätsfunktion:

```
factorial(N,F) :- factorial(0,N,1,F).
factorial(I,N,T,F) :-
        I < N, I1 is I+1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

- Welche Variante is vorzuziehen? (☞ Übung)

# Explizite Kontrolle der Zielauswertung

- Prolog stellt ein Systemprädikat, die das Backtracking bei der Suche von Prolog steuern kann: *cut*, geschrieben !.

- Der Sinn eines **cut** ist, den Suchaufwand für eine Berechnung zu reduzieren, indem man einen Zweig des Suchbaumes abschneidet.
  - green cuts: schneiden Zweige ab, die keine Lösungen enthalten $\Longrightarrow$ erhöhen die Effizienz;
  - red cuts: schneiden Zweige ab, die Lösungen enthalten.

# Cuts in Prologprogrammen

- Da cuts die Bedeutung eines Programms von der prozeduralen Interpretierung zusätzlich abhängig machen, verletzten sie die strebenswerte Deklarativität.
  - green cuts: nützlich als Kompromiss zwischen Effizienz und Deklarativität.
  - red cuts: eher unerwünscht.

# Cuts: Bedeutung

**Cut ist** ein nullstelliges Prädikat, das **immer erfüllt** ist. Wird das Cut im Laufe der Ableitung eines aktuellen Zieles $A'$ mit Hilfe einer Klausel

$$A \leftarrow A_1, \ldots, A_k, !, A_{k+1}, \ldots, A_n$$

erfüllt (wobei $A$ und $A'$ unifizieren), so werden Alternative zur Erfüllung von $A$, $A_1$, $\ldots$, $A_k$ im Laufe der aktuellen Ableitung von $A'$ ausgeschlossen. D.h.:

- alternative Klauseln, deren Kopf mit $A'$ unifizieren werden ignoriert;
- Wenn die Ableitung von $A_i$ mit $i \geq k+1$ im Laufe der weiteren Ableitung von $A'$ fehlschlägt, werden im Laufe des Backtracking alternative Ableitungen nur soweit zurückverfolgt bis zum !.

# Green Cuts

Klauseln zum Mischen zweier geordneten Listen:

```
merge([X|Xs],[Y|Ys],[X|Zs]) :-
                          X < Y, merge(Xs,[Y|Ys],Zs).      (1)

merge([X|Xs],[Y|Ys],[X,Y|Zs]) :-
                          X =:= Y, merge(Xs,Ys,Zs).        (2)

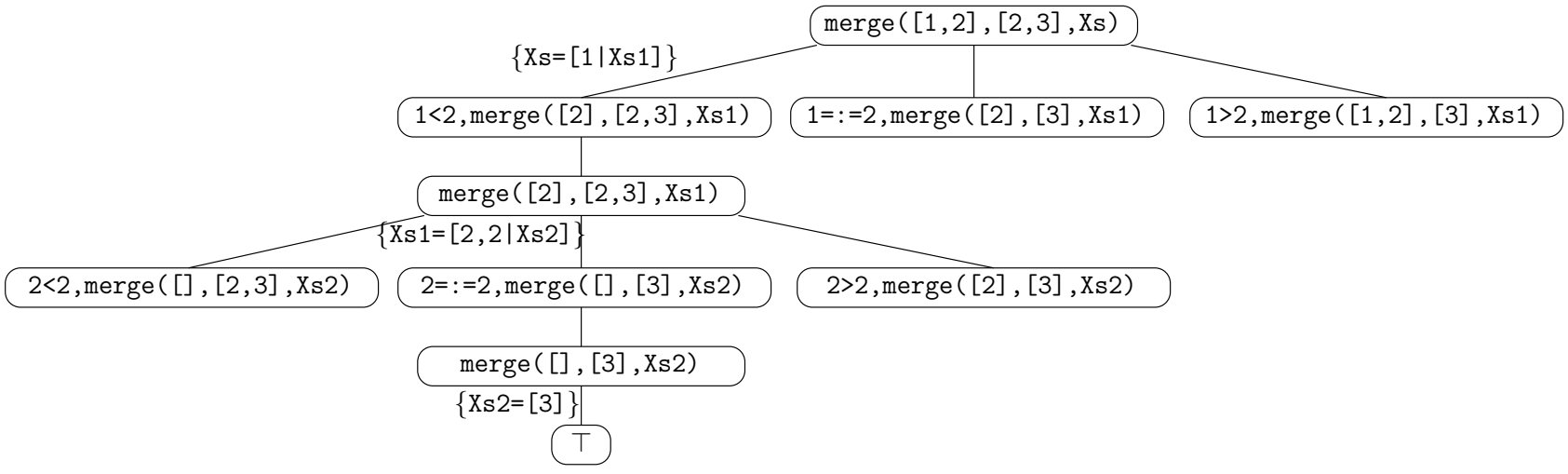
merge([X|Xs],[Y|Ys],[Y|Zs]) :-
                          X > Y, merge([X|Xs],Ys,Zs).      (3)

merge([],[Y|Ys],[Y|Ys]).                                   (4)
merge(Xs,[],Xs).                                           (5)
```

- Das Programm ist deterministisch: es gibt für jedes Ziel höchstens eine Klausel, die zur erfolgreichen Ableitung des Zieles führt;

- Ob die Auswahl einer der Klauseln (1), (2), (3) zum Erfolg führt, hängt ausschließlich von den Testen `X<Y`, `X=:= Y` bzw. `X > Y`.

# Green Cuts


merge([1,2],[2,3],Xs)
{Xs=[1|Xs1]}
1<2,merge([2],[2,3],Xs1)
1=:=2,merge([2],[3],Xs1)
1>2,merge([1,2],[3],Xs1)
merge([2],[2,3],Xs1)
{Xs1=[2,2|Xs2]}
2<2,merge([],[2,3],Xs2)
2=:=2,merge([],[3],Xs2)
2>2,merge([2],[3],Xs2)
merge([],[3],Xs2)
{Xs2=[3]}
⊤

# Green Cuts

Wenn (1) zur Erfüllung eines Zieles gewählt wird, braucht man nach dem Test `X =< Y` keine weitere Klauseln betrachten. Ähnliches gilt für den Test `X =:= Y` in (2). Zur Vermeidung der Suche unnötiger Ableitungen kann man hier Cuts einsetzen:

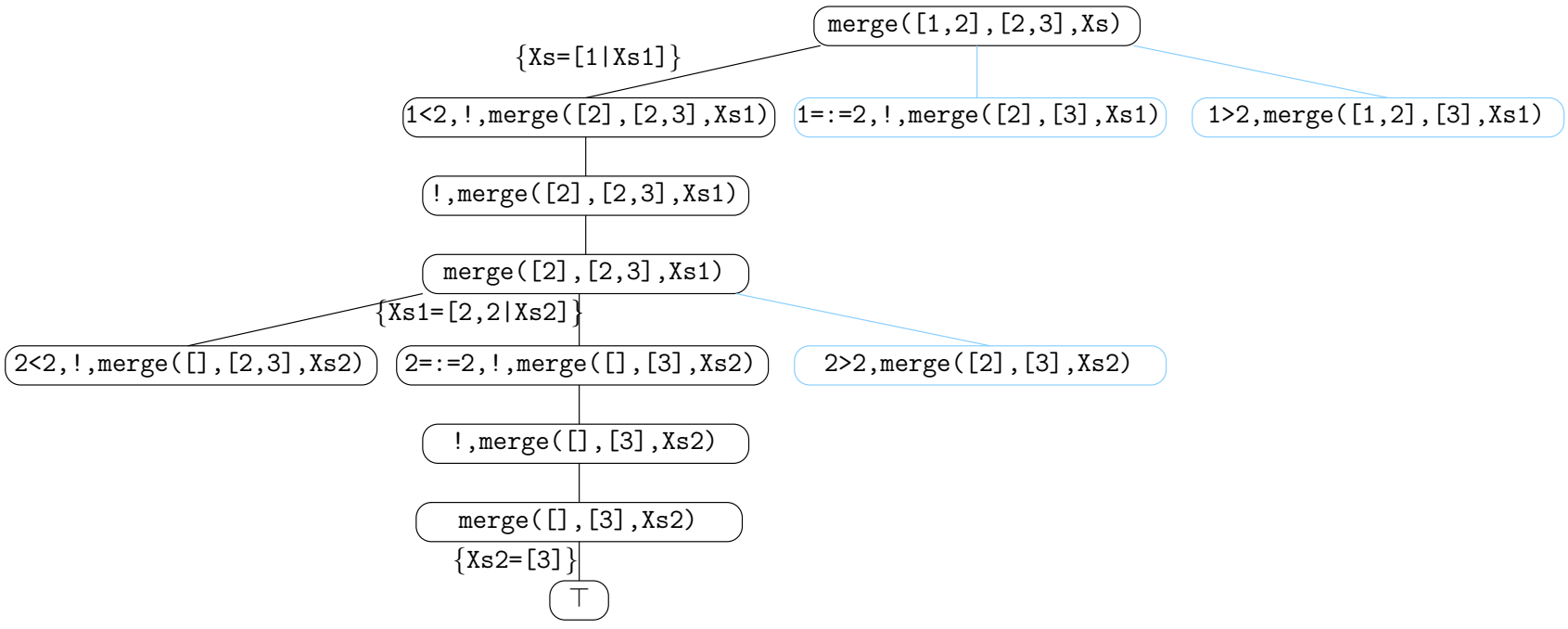
```
merge([X|Xs],[Y|Ys],[X|Zs]) :-
                        X < Y, !, merge(Xs,[Y|Ys],Zs).     (1)

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
                        X =:= Y, !, merge([X|Xs],Ys,Zs).   (2)

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
                        X > Y, merge([X|Xs],Ys,Zs).        (3)

merge([],[Y|Ys],[Y|Ys]).                                   (4)
merge(Xs,[],Xs).                                           (5)
```

# Green Cuts


merge([1,2],[2,3],Xs)
{Xs=[1|Xs1]}
1<2,!,merge([2],[2,3],Xs1)
1=:=2,!,merge([2],[3],Xs1)
1>2,merge([1,2],[3],Xs1)
!,merge([2],[2,3],Xs1)
merge([2],[2,3],Xs1)
{Xs1=[2,2|Xs2]}
2<2,!,merge([],[2,3],Xs2)
2=:=2,!,merge([],[3],Xs2)
2>2,merge([2],[3],Xs2)
!,merge([],[3],Xs2)
merge([],[3],Xs2)
{Xs2=[3]}
⊤

# Die Prolog-Auswertung

- Die Auswertung eines Zieles in Prolog benötigt einen Stack für den Tiefendurchlauf über den Suchbaum. Darin muss man i.A. beim Absteigen durch die Auswahl einer Klausel für das aktuelle Ziel die übrigen alternativen Klauseln merken, um später die Suche via Backtracking fortsetzen zu können.
  $\Longrightarrow$ Informationen über die jeweils zuletzt gewählten Klauseln (*choice points*) müssen in jedem Kellerrahmen gespeichert werden.

- Insbesondere, in der prozeduralen Auslegung:
  - Eine Klausel $A \leftarrow B_1, \ldots, B_n$ entspricht der Definition einer Prozedur $A$.
  - Im Unterschied zu prozeduralen Programmiersprachen hat $A$ statt eine, so viele Definitionen, wieviele Klauseln $A$ definieren. Ein Interpreter muss i.A. alle Definitionen der Reihe nach betrachten.

# Die Last-call-Optimierung

**Last-call-Optimierung:** $A \leftarrow B_1, \ldots, B_{n-1}, B_n$

- Benutze für die Auswertung von $B_n$ den Kellerrahmen für die Auswertung von $A$ wieder.

  Notwendige Bedingung: es gibt keine Alternative Berechungen der Ziele $A$, $B_1$, $\ldots$, $B_{n-1}$. Manche Gelegenheiten zur Last-call-Optimierung können automatisch erkannt werden.

- Cuts können solche Optimierungen zusätzlich unterstützen z.B.:

$$A \leftarrow B_1, \ldots, B_{n-1}, !, B_n$$

# Die Last-call-Optimierung

Die letzten (rekursiven) Prädikate in den untenstehenden Klauseln eignen sich für die Last-call-Optimierung (Tail-Recursion-Optimierung):

```
merge([X|Xs],[Y|Ys],[X|Zs]) :-
            X < Y, !, merge(Xs,[Y|Ys],Zs).

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
            X =:= Y, !, merge([X|Xs],Ys,Zs).

merge([X|Xs],[Y|Ys],[Y|Zs]) :-
            X > Y, merge([X|Xs],Ys,Zs).

merge([],[Y|Ys],[Y|Ys]).
merge(Xs,[],Xs).
```

# Negation in Prolog

Negation in Prolog wird mit Hilfe des vordefinierten Prädikats `not` implementiert, das wie folgt definiert ist:

```
not(X) :- call(X), !, fail
not(X).
```

- Ein Feature von Prolog ist, dass Terme benutzt werden können, um beides Programme und Daten zu repräsentieren. Daten können in Programme transformiert werden (und umgekehrt): `call(X)` transformiert `X` in einem Ziel und versucht dieses abzuleiten. Syntaktisch: `call(X)` $\equiv$ `X` **als Ziel**.

- `fail` ist ein Prädikat, das immer scheitert.

# Negation in Prolog vs. NaF

- **Das Metavariable Feature** ist eine vereinfachte Syntax, die erlaubt, `call` wegzulassen

```
not(X) :- X, !, fail
not(X).
```

- `not` ist eine ungenaue Implementierung der Negation durch Scheitern.
  - $not(X)$ ist erfolgreich in der LP-Semantik, wenn alle Pfade in allen Suchbäumen endlich sind und zu Scheiternknoten führen.
  - `not(X)` ist erfolgreich in Prolog, wenn alle Pfade im Prolog-Suchbaum endlich sind und zu Scheiternknoten führen.

# Negation in Prolog vs. NaF

```
p(X) :- p(X).
q(a).
```

- $not(q(b), p(a))$ ist nicht definiert in der LP-Semantik.
- `not((q(b),p(a)))` ist **erfolgreich** in Prolog.

# Negation und Ziele mit Variablen

**Bsp.:**

```
braucht_schein(X) :- not(diplom(X)), student(X).
student(peter).
student(martina).
diplom(martina).
```

Die Anfrage `braucht_schein(X)` scheitert, obwohl die Antwort `{X=peter}` unter einer Interpretierung von `not` als logische Negation erwartet wird.

$\Longrightarrow$ Man muss sicherstellen, dass die Variablen in einem negierten Ziel bei seiner Auswertung belegt sind.

# Negation und Ziele mit Variablen

**Bsp.:**

```
braucht_schein(X) :- student(X), not(diplom(X)).
student(peter).
student(martina).
diplom(martina).
```

$\Longrightarrow$ Antwort {X=peter}.

# Red Cuts

**Bsp.:**

```
min(X,Y,X) :- X =< Y.
min(X,Y,Y) :-  X > Y.
```

Cuts deren Auftritt in einem Programm die Semantik des Programmes ändern heißen red cuts.

```
min(X,Y,X) :- X =< Y, !.
min(X,Y,Y).
```

Die Anfrage `min(1,2,2)` liefert (unerwünschterweise) *yes*.

# Red Cuts

**Bsp.:** Eine Prozedur zur Entfernung von Elementen aus einer Liste:

```
delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- Z \== X, delete(Xs,Z,Ys).
delete([],X,[]).
```

# Red Cuts

- Delete mit green Cuts:

```
delete([X|Xs],X,Ys) :- !, delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- Z\==X,!,delete(Xs,Z,Ys).
delete([],X,[]).
```

- Delete mit red Cuts:

```
delete([X|Xs],X,Ys) :- !, delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- !, delete(Xs,Z,Ys).
delete([],X,[]).
```

Das Programm mit red Cuts funktioniert richtig, ist dafür bei minimal besserer Effizienz viel unleserlicher geworden.

# Selbst-modifizierende Programme

- Prolog erlaubt laufende Programme bei der Laufzeit zu analysieren und zu ändern.
- Das Ziel `clause(Kopf,Rumpf)` bietet Zugang zu den Klauseln des laufenden Programms.
- `Kopf` darf keine unbelegte Variable sein.
- Die erste Klausel, deren Kopf mit `Kopf` unifiziert wird gefunden und `Rumpf` wird mit dessen Rumpf unifiziert.
- Alle Klauseln deren Kopf mit `Kopf` unifizieren werden via Backtracking gefunden.
- Fakten haben `true` als ihren Rumpf.

# Selbst-modifizierende Programme

## Bsp.:

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
```

Anfrage `clause(member(X,Ys),Rumpf)` liefert die Antworten:

- `{Ys=[X|Xs], Rumpf=true};`
- `{Ys=[Y|Ys1], Rumpf=member(X,Ys1)}.`

# Selbst-modifizierende Programme

Systemprädikate zum Hinzufügen und Entfernen von Klauseln zum (laufenden) Programm:

- `assertz(Klausel)`: fügt `Klausel` als letzte Klausel der entsprechenden Prozedur.

  **Bsp.:** `assertz((mammal(X) :- whale(X))).`

- `asserta(Klausel)`: fügt `Klausel` als die erste Klausel der entsprechenden Prozedur.

- `retract(K)`: entfernt die erste Klausel, die mit `K` unifiziert.

  **Bsp.:** Eine Klausel `a :- b,c` kann mit `retract((a :- X))` entfernt werden.

# Selbst-modifizierende Programme

Das dynamische Hinzufugen und Entfernen von Klauseln macht einen Unterschied zwischen **dynamischen** und **statischen** benutzerdefinierten **Prädikaten**.

- Statische Prädikate können kompiliert werden $\Longrightarrow$ können effizienter ausgewertet werden.
- Dynamische Prädikate müssen als solche deklariert werden.
- Dynamische Prädikate machen den Code von Seiteneffekten abhängig $\Longrightarrow$ weniger deklarativ und leserlich.
- Allerdings können dynamische Prädikate u.U. die Effizienz unterstützen, z.B. für *dynamische Programmierung*.

# Beispiel: Dynamische Programmierung

- **Dynamische Programmierung**
  - Idee: speichere partielle Ergebnisse während einer Berechnung, die später wieder benutzt werden können.
  - Bsp.: Berechnung der Binomialkoeffizienten (☞ Übung)
- **Mögliche Umsetzung in Prolog**

  Versuche ein Ziel abzuleiten, und wenn dies möglich ist, speichere dieses Ergebnis als Fakt und vermeide, dass später alternative Ableitungen betrachtet werden.

```
lemma(Z) :- Z, asserta((Z :- !)).
```

# Beispiel: Dynamische Programmierung

Anwendung: Die Türme aus Hanoi

```
:- dynamic hanoi/5.

hanoi(1,A,B,C,[move(A,B)]).
hanoi(N,A,B,C,Moves) :-
        N > 1, N1 is N -1,
        lemma(hanoi(N1,A,C,B,Ms1)),
        hanoi(N1,C,B,A,Ms2),
        append(Ms1,[move(A,B)|Ms2],Moves).

lemma(P):- P, asserta((P :- !)).

test_hanoi(N,Pegs,Moves) :-
        hanoi(N,A,B,C,Moves), Pegs = [A,B,C].
```

# Beispiel: Dynamische Programmierung

Anwendung: Die Türme aus Hanoi

```
:- dynamic hanoi/5.

hanoi(1,A,B,C,[move(A,B)]).
hanoi(N,A,B,C,Moves) :-
        N > 1, N1 is N -1,
        lemma(hanoi(N1,A,C,B,Ms1)),
        hanoi(N1,C,B,A,Ms2),
        append(Ms1,[move(A,B)|Ms2],Moves).

lemma(P):- P, asserta((P :- !)).

test_hanoi(N,Pegs,Moves) :-
        hanoi(N,A,B,C,Moves), Pegs = [A,B,C].


?- test_hanoi(3,[a,b,c],X).
X = [move(a, b),move(a, c),move(b, c),move(a, b),move(c, a),move(c, b),move(a, b)]
```

# Mehr Prolog

Prolog bietet mehr an, z.B.:

- Prädikate zum **Testen und Manipulieren der Struktur der Terme**;
- Mehr meta-logische Prädikate z.B. zum **Testen des Zustands der Ableitung**;
- Mehr extra-logische Prädikate, die Seiteneffekte bei ihrer "Ableitung" haben, z.B für **Ein- und Ausgabe** oder für die **Schnittstelle zum Betriebssystem**.
- Es gibt Prolog-Erweiterungen, z.B. für Constraint-Programmierung...

# Part IV

# Constraint-Programmierung

# Constraint-Programmierung (CP)

Einschränkungen der logischen Programmierung:

- Alle in einem puren logischen Programm manipulierten Objekte (die **Terme**) **sind rein syntaktische Konstruktionen**, denen keine Semantik zugewiesen wird.

- D.h. die **Funktor-Symbolen sind nicht-interpretiert**.

- **Bsp.:**
  - Das Ziel `X=2+3` bewirkt nur die Bindung von `X` an den Term `2+3`, weil das Funktionssymbol `+` nicht interpretiert wird.

  - Das Ziel `1+4=2+3` scheitert.

# Constraints

- Zwei Objekte sind in LP nur dann gleich, wenn sie syntaktisch gleich sind.
- Idee: erweitere die rein syntaktische Gleichheit in LP zur Gleichung, die zu lösen ist:
  - Das Ziel `X+Y=8,X-Y=2` erhält in einer CP-Sprache die Antwort `X=5,Y=3`.
- Allgemeiner: spezifiziere (implizite) Relationen zwischen semantischen Objekten. I.A. heißen Relationen zwischen semantischen Objekten **Constraints**.

# CP als Erweiterung der logischen Programmierung

- Die Constraints werden als syntaktisch ausgezeichnete Prädikate dargestellt, und statt mittels Resolution durch spezielle Algorithmen über bestimmte Wertebereiche mit Hilfe eines **Constraint-Löser** gelöst.

- LP ist CP, wobei das einzige Constraint die syntaktische Gleichheit zwischen Termen ist, und der Unifikationsalgorithm zur Lösung solcher Constraints benutzt wird.

# Constraint-Löser

- Programmierung mit Constraints ist in beliebigen Programmiersprachen möglich, vorausgesetzt dass ein Constraint-Löser, evt. als eine Erweiterungs-Bibliothek zur Verfügung gestellt wird, z.B. für Java, C++, Prolog, Lisp.

- Meist werden LP-Sprachen um Constraints erweitert (☞ Constraint-Logikprogrammierung), z.B. Eclipse-, Sicstus-, SWI-Prolog, Mozart/Oz, etc.

- Effektive Constraint-Löser gibt für verschiedene Constraint-Arten, z.B.:
  - Logische Formeln über boolsche Variablen
  - Intervall-Constraints über endliche Bereiche
  - Lineare Gleichungssysteme über reale Zahlen

# Constraint-Löser

Von einem Constraint-Löser zu erfüllenden Berechnungsdienste:

- **Konsistenztest**(Erfüllbarktest) (*consistency/satisfiability test*): sind die Constraints erfüllbar? Ein Constraint-Löser ist **vollständig**, wenn er die Erfüllbarkeit jeder beliebigen Menge von Constraints entscheiden kann.

- **Vereinfachung**: Die Constraints in eine einfachere Normalform darstellen können. Zur effizienten Vereinfachung soll der Löser **inkrementell** sein: Vereinfachung der Constraints zusammen mit einem neu hinzukommendes Constraint ohne die Vereinfachung der bisherigen Constraints.

- **Determination**: Erkennen, wenn eine Variable nur noch einen bestimmten Wert haben kann. (z.B. $X \geq 1$, $X \leq 1 \Rightarrow X = 1$.)

# Vorteile der Constraint-Logikprogrammierung

- Zusätzlich zur erhöhten Deklarativität kann CP die Effizienz der LP-Sprachen erhöhen.

- Constraints können benutzt werden, um den *Nicht-Determinismus* der LP-Suche nach Lösungen einzuschränken, indem man Teilbäume von der Suche ausschliesst, die keine Lösung der Constraints enthalten können (durch Konsistenzteste).
  $\Longrightarrow$ CP wird eingesetzt, um kombinatorische Probleme zu lösen, die meist exponentielle Komplexität haben.

# Syntax einer CP-Sprache

| | | | |
|---|---|---|---|
| Atome: | $A, B$ | $::=$ | $p(t_1, \ldots, t_n)$ |
| **Constraints:** | $C, D$ | $::=$ | $c(t_1, \ldots, t_n) \mid C \wedge D$ |
| Ziele: | $G, H$ | $::=$ | $\top \mid \bot \mid A \mid C \mid G \wedge H$ |
| Klauseln: | $K$ | $::=$ | $A \leftarrow G$ |
| Programme: | $P$ | $::=$ | $\{K_1, \ldots, K_m\}$ |

# Berechnungszustände

- Ein **Zustand** ist ein Paar $< G, C >$, wobei $G$ ein Ziel und $C$ ein Constraint.
- $G$ heißt **Zielspeicher** (die noch zu lösende Ziele), $C$ heißt **Constraintspeicher** (die bereits aufgetretene Constraints).
- Ein **Anfangszustand** ist ein Zustand der Form $< G, \mathit{true} >$.
- Ein Zustand heißt **erfolgreicher Endzustand**, falls er von der Form $< \top, C >$ ist.
- Ein Zustand heißt **erfolgloser Endzustand**, falls er von der Form $< G, \mathit{false} >$ ist.

# CP-Kalkül

**Entfalten** (*unfold*)

$$\frac{(B \leftarrow H) \in P \qquad (B \stackrel{*}{=} A) \wedge C \text{ ist erfüllbar}}{< A \wedge G, C > \mapsto_{Entfalten} < H \wedge G, (B \stackrel{*}{=} A) \wedge C >}$$

**Scheitern** (*failure*)

$$\frac{\text{Es gibt keine Klausel } (B \leftarrow H) \in P, \text{ so dass } (B \stackrel{*}{=} A) \wedge C \text{ erfüllbar ist}}{< A \wedge G, C > \mapsto_{Scheitern} < \bot, \mathit{false} >}$$

**Vereinfachen** (*solve*)

$$\frac{C \wedge D_1 \equiv D_2}{< C \wedge G, D_1 > \mapsto_{Vereinfachen} < G, D_2 >}$$

wobei $B \stackrel{*}{=} A$ gilt gdw. $A = p(t_1, \ldots, t_n), B = p(s_1, \ldots, s_n)$ und $t_1 = s_1 \wedge \ldots \wedge t_n = s_n$.

# Die Antwort einer CP-Berechnung

- Die Antwort einer Berechnung, die einen erfolgreichen Endzustand $< \top, C >$ erreicht, ist $C$.
- Eine Antwort heißt **bestimmt** (*definite*), wenn er eine Gleichung `X=Konstante` für jede Variable in der Anfrage enthält.
  - Z.B. `X+Y=10,X-Y=6` liefert die Antwort `X=8,Y=2`.
- I.A. kann eine Antwort **unbestimmt** (*indefinite*) sein, d.h. eine unendliche Menge von Lösungen repräsentieren.
  - Z.B. liefert `X≤Y,Y≤Z,Z≤X` die Antwort `X=Y=Z`.

# Lineare arithmetische Constraints

- **Arithmetische Ausdrücke:**

  t ::= Zahl | Variable | $t_1 \odot t_2$ mit $\odot \in \{+, -, *, /\}$

  **Linearität**:
  - Höchstens ein Term einer Multiplikation enthält eine Variable.
  - Der Teiler in einer Division enthält keine Variable.

- **Arithmetische Constraints:**

  C ::= true | false | $C \wedge C$ | $t_1 \ \mathcal{R} \ t_2$ mit $\mathcal{R} \in \{<, \leq, =, >, \geq, \neq\}$

# Beispiel: Menu-Berechner

```
appetiser(radishes,1). appetiser(pasta,6).
meat(beef,5). meat(pork,7).
fish(sole,2). fish(tuna,4).
dessert(fruit,2). dessert(icecream,6).

main(M,I) :- meat(M,I).
main(M,I) :- fish(M,I).

lightmeal(A,M,D) :-
   I>0, J>0, K>0,
   I+J+K <= 10,
   appetiser(A,I), main(M,J), dessert(D,K).
```

# Beispiel: erfolgreiche Berechnung

$< \texttt{lightmeal(A,M,D)};\texttt{true} >$

# Beispiel: erfolgreiche Berechnung

```
      < lightmeal(A,M,D);true >
↦     < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
```

# Beispiel: erfolgreiche Berechnung

$$< \texttt{lightmeal(A,M,D);true} >$$

$$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$$

$$\mapsto^* \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} >$$

# Beispiel: erfolgreiche Berechnung

$$
\begin{array}{ll}
 & \langle\, \texttt{lightmeal(A,M,D)};\texttt{true} \,\rangle \\
\mapsto & \langle\, \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K)};\texttt{true} \,\rangle \\
\mapsto^{*} & \langle\, \texttt{appetiser(A,I),main(M,J),dessert(D,K)};\texttt{I>0,J>0,K>0,I+J+K<=10} \,\rangle \\
\mapsto & \langle\, \texttt{main(M,J),dessert(D,K)};\texttt{A=radishes,I=1,J>0,K>0,1+J+K<=10} \,\rangle
\end{array}
$$

# Beispiel: erfolgreiche Berechnung

$< \texttt{lightmeal(A,M,D)};\texttt{true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K)};\texttt{true} >$

$\mapsto^* \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K)};\texttt{I>0,J>0,K>0,I+J+K<=10} >$

$\mapsto \quad < \texttt{main(M,J),dessert(D,K)};\texttt{A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

$\mapsto \quad < \texttt{meat(M1,I1),dessert(D,K)};$
$\texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

# Beispiel: erfolgreiche Berechnung

$< \texttt{lightmeal(A,M,D)};\texttt{true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K)};\texttt{true} >$

$\mapsto^* \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K)};\texttt{I>0,J>0,K>0,I+J+K<=10} >$

$\mapsto \quad < \texttt{main(M,J),dessert(D,K)};\texttt{A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

$\mapsto \quad < \texttt{meat(M1,I1),dessert(D,K)};$
$\texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

$\mapsto \quad < \texttt{dessert(D,K)};\texttt{M1=beef,I1=5,M=beef,J=5,A=radishes,I=1,K>0,1+5+K<=10} >$

# Beispiel: erfolgreiche Berechnung

$$
\begin{array}{ll}
 & \langle \texttt{lightmeal(A,M,D)};\texttt{true} \rangle \\
\mapsto & \langle \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K)};\texttt{true} \rangle \\
\mapsto^{*} & \langle \texttt{appetiser(A,I),main(M,J),dessert(D,K)};\texttt{I>0,J>0,K>0,I+J+K<=10} \rangle \\
\mapsto & \langle \texttt{main(M,J),dessert(D,K)};\texttt{A=radishes,I=1,J>0,K>0,1+J+K<=10} \rangle \\
\mapsto & \langle \texttt{meat(M1,I1),dessert(D,K)}; \\
 & \qquad \texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} \rangle \\
\mapsto & \langle \texttt{dessert(D,K)};\texttt{M1=beef,I1=5,M=beef,J=5,A=radishes,I=1,K>0,1+5+K<=10} \rangle \\
\mapsto & \langle \top;\texttt{D=fruit,K=2,M1=beef,I1=5,M=beef,J=5,A=radishes,I=1} \rangle
\end{array}
$$

# Beispiel: erfolgreiche Berechnung

$$
\begin{array}{ll}
 & \langle \texttt{lightmeal(A,M,D)};\texttt{true} \rangle \\
\mapsto & \langle \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K)};\texttt{true} \rangle \\
\mapsto^{*} & \langle \texttt{appetiser(A,I),main(M,J),dessert(D,K)};\texttt{I>0,J>0,K>0,I+J+K<=10} \rangle \\
\mapsto & \langle \texttt{main(M,J),dessert(D,K)};\texttt{A=radishes,I=1,J>0,K>0,1+J+K<=10} \rangle \\
\mapsto & \langle \texttt{meat(M1,I1),dessert(D,K)}; \\
 & \qquad \texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} \rangle \\
\mapsto & \langle \texttt{dessert(D,K)};\texttt{M1=beef,I1=5,M=beef,J=5,A=radishes,I=1,K>0,1+5+K<=10} \rangle \\
\mapsto & \langle \top;\texttt{D=fruit,K=2,M1=beef,I1=5,M=beef,J=5,A=radishes,I=1} \rangle
\end{array}
$$

Antwort: *A=radishes,M=beef,D=fruit*.

# Beispiel: gescheiterte Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

# Beispiel: gescheiterte Berechnung

```
    < lightmeal(A,M,D);true >
↦   < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
```

# Beispiel: gescheiterte Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$

$\mapsto \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} >$

# Beispiel: gescheiterte Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$

$\mapsto \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} >$

$\mapsto \quad < \texttt{main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10} >,$

# Beispiel: gescheiterte Berechnung

```
    < lightmeal(A,M,D);true >
↦   < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
↦   < appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10 >
↦   < main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10 >,
↦   < meat(M1,I1),dessert(D,K);
                              M=M1,J=I1,A=pasta,I=6,J>0,K>0,6+J+K<=10 >
```

# Beispiel: gescheiterte Berechnung

```
    < lightmeal(A,M,D);true >
↦   < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
↦   < appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10 >
↦   < main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10 >,
↦   < meat(M1,I1),dessert(D,K);
                              M=M1,J=I1,A=pasta,I=6,J>0,K>0,6+J+K<=10 >
↦   < dessert(D,K);M1=beef,I1=5,M=beef,J=5,A=pasta,I=6,K>0,6+5+K<=10 >
```

# Beispiel: gescheiterte Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$

$\mapsto \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} >$

$\mapsto \quad < \texttt{main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10} >,$

$\mapsto \quad < \texttt{meat(M1,I1),dessert(D,K);}$
$\texttt{M=M1,J=I1,A=pasta,I=6,J>0,K>0,6+J+K<=10} >$

$\mapsto \quad < \texttt{dessert(D,K);M1=beef,I1=5,M=beef,J=5,A=pasta,I=6,K>0,6+5+K<=10} >$

$\mapsto \quad < \bot, \mathit{false} >$

# Beispiel: Finanzberater

```
darlehen(Darlehenshoehe, Monate, Zinssatz, Rate, Restschuld)
 :- Monate=0, Darlehenshoehe=Restschuld

darlehen(Darlehen, Monate, Zinssatz, Rate, Restschuld) :-
    Monate>0,
    Monate1=Monate-1,
    Darlehen1=Darlehen+Darlehen*Zinssatz-Rate,
    darlehen(Darlehen1, Monate1, Zinssatz, Rate, Restschuld).
```

# Beispiel: Finanzberater

SWI-/Sicstus-Prolog-Syntax:

```
:- use_module(library(clpr)).


darlehen(Darlehenshoehe,Monate,Zinssatz,Rate,Restschuld)
 :- {Monate=0},Darlehenshoehe=Restschuld.

darlehen(Darlehens,Monate,Zinssatz,Rate,Restschuld) :-
   {Monate>0,
   Monate1=Monate-1,
   Darlehens1=Darlehens+Darlehens*Zinssatz-Rate},
   darlehen(Darlehens1,Monate1,Zinssatz,Rate,Restschuld).
```

# Beispiel: Finanzberater...

- Welcher Restschuld bleibt nach 30 Jahren für ein Darlehen von €200000 bei einer Rate von €1000 und einem monatlichen Zinssatz von 0.4%.

```
?- darlehen(200000,360,0.004,1000,S).
S = 39570.5
```

- Wie hoch ist die Rate, um das Darlehen in 30 Jahren vollständig zu zahlen?

```
?- darlehen(200000,360,0.004,X,0.0).
X = 1049.33
```

- Wieviele Monate muss man zahlen, um die Schulden zu bezahlen bei einer monatlicher Rate von €1000?

```
?- darlehen(200000,X,0.004,1000,S),{S<0.0}.
X = 404.0, S = -836.065
```

# Beispiel: Finanzberater

- Welches ist das Verhältnis zwischen Darlehenshöhe und Rate, wenn man in 30 Jahren die Schulden komplett Zahlen möchte?

```
?- darlehen(D,360,0.004,R,0.0).
{R=0.00524665*D}
```

- Bei welchem Zinssatz bezahlt man die Schuld in genau 30 Jahren?

```
darlehen(200000,360,Z,1000,0.0).
```

...kann nicht von einem linearer Gleichungslöser behandelt werden.
Grund: die im zweiten Schritt aufgestellte Gleichung ist nicht mehr linear:

$$D_1 = D + D \cdot Z - R$$
$$D_2 = D_1 + D_1 \cdot Z - R$$
$$\vdots$$

# Constraints über endliche (ganzzahlige) Wertebereiche (FD-Constraints)

## Constraints:

$$C \quad ::= \quad \text{true} \mid \text{false} \mid C \wedge C \mid X \text{ in } n..m \mid$$

$$X \text{ in } [k_1, \ldots, k_l] \mid X + Y = Z \mid X \; \mathcal{R} \; Y$$

mit $n, m, k_1, \ldots, k_l \in \mathbb{N}$

$\mathcal{R} \in \{<, \leq, =, >, \geq, \neq\}$

$X, Y, Z$ Variablen oder ganze Zahlen

# Beispiel

FD-Constraints in SWI-Prolog:

```
?- use_module(library('clp/bounds')).
Yes

?- X in 1..2, Y in 3..5, Z #= X+Y.
X = _G137389{1..2}
Y = _G137395{3..5}
Z = _G137401{4..7}
```

# Constraint Satisfaction Probleme (CSPs)

- Ein CSP ist definiert durch:
  - Eine Menge von Variablen $X_1 \in D_1, \ldots, X_n \in D_n$, mit $D_1, \ldots, D_n$ endlichen Wertebereichen
  - Eine Menge von Constraints, die von den Variablen zu erfüllen sind.
- Eine Lösung eines CSP ist eine Variablenbelegung, die die Constraints erfüllt.

# Lösung eines CSP mit CP(FD)

Der Ansatz zur Lösung eines CSP mit Constraints über endliche Bereiche besteht typischerweise aus drei Komponenten:

1. Deklaration der Wertebereiche der Variablen.
2. Aufsetzen der Constraints
3. Suche einer Lösung (Backtracking, Branch-and-Bound)

# Beispiel: CSP Problem

- Finde eine Belegung, so dass die Folgende Berechnung (zur Basis 10) wahr ist!

$$\begin{array}{cccccc} & & S & E & N & D \\ & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

- Suchraum hat die Größe $10^8$. Exhaustive Suche ist praktisch nicht einsetzbar.

- Ein Mensch würde das Problem lösen, indem er Constraints dynamisch ableitet. Z.B. muss M gleich 1 sein. Es folgt, dass S gleich 9 oder 8 ist. U.s.w...

# Beispiel: Send more money

1. Ansatz (SWI-Prolog):

```
:- use_module(library('clp/bounds')).

solve(S,E,N,D,M,O,R,Y) :-
      [S,E,N,D,M,O,R,Y] in 0..9,
      S#>0, M#>0,
      all_different([S,E,N,D,M,O,R,Y]),
                      1000*S + 100*E + 10*N + D
      +               1000*M + 100*O + 10*R + E
      #= 10000*M + 1000*O + 100*N + 10*E + Y.
```

```
?- solve(S,E,N,D,M,O,R,Y).
M = 1, O = 0, S = 9, E in 2..8, N in 5..8,
D in 2..8, R in 2..8, Y in 2..8
```

# Suche

- Ein Constraint-Löser schließt Werte aus, die nicht zu einer Lösung beitragen können:

```
X in 1..3, Y in 4..6, Y #= 2*X.
X in 2..3 Y in 4..6
```

- I.A. muss man noch eine Suche über diese Werte durchführen, um eventuelle eindeutige Lösungen zu finden.
  Man braucht ein Prädikat, welches Variablen alle Werte aus ihren Wertebereichen nach einer vorgegebenen Strategie zuordnet.

```
X in 1..3, Y in 4..6, Y #= 2*X, label([X,Y]).
X = 2 Y = 4 ;
X = 3 Y = 6 ;
No
```

# Beispiel: Send more money

2. Lösung mit Suche:

```
solve1(S,E,N,D,M,O,R,Y) :-
      [S,E,N,D,M,O,R,Y] in 0..9,
      S#>0, M#>0,
      all_different([S,E,N,D,M,O,R,Y]),
                     1000*S + 100*E + 10*N + D
      +              1000*M + 100*O + 10*R + E
      #= 10000*M + 1000*O + 100*N + 10*E + Y,
      label([S,E,N,D,M,O,R,Y]).
```

```
?- solve1(S,E,N,D,M,O,R,Y).
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2
```