Programmiersprachen

Wintersemester 2005/2006

Alexandru Berlea

Lehrstuhl Prof. Seidl

Institut für Informatik
TU München

1 Allgemeines

Inhalt der Vorlesung: Programmierparadigmen

- funktional
- logisch
- imperativ
- objektorientiert

Organisation

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

2 Einleitung

Motivation

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

Lernziele

Was wir lernen:

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

Was wir nicht lernen:

- einzelne Programmiersprachen im Detail
- Implementierungstechniken für Programmiersprachen

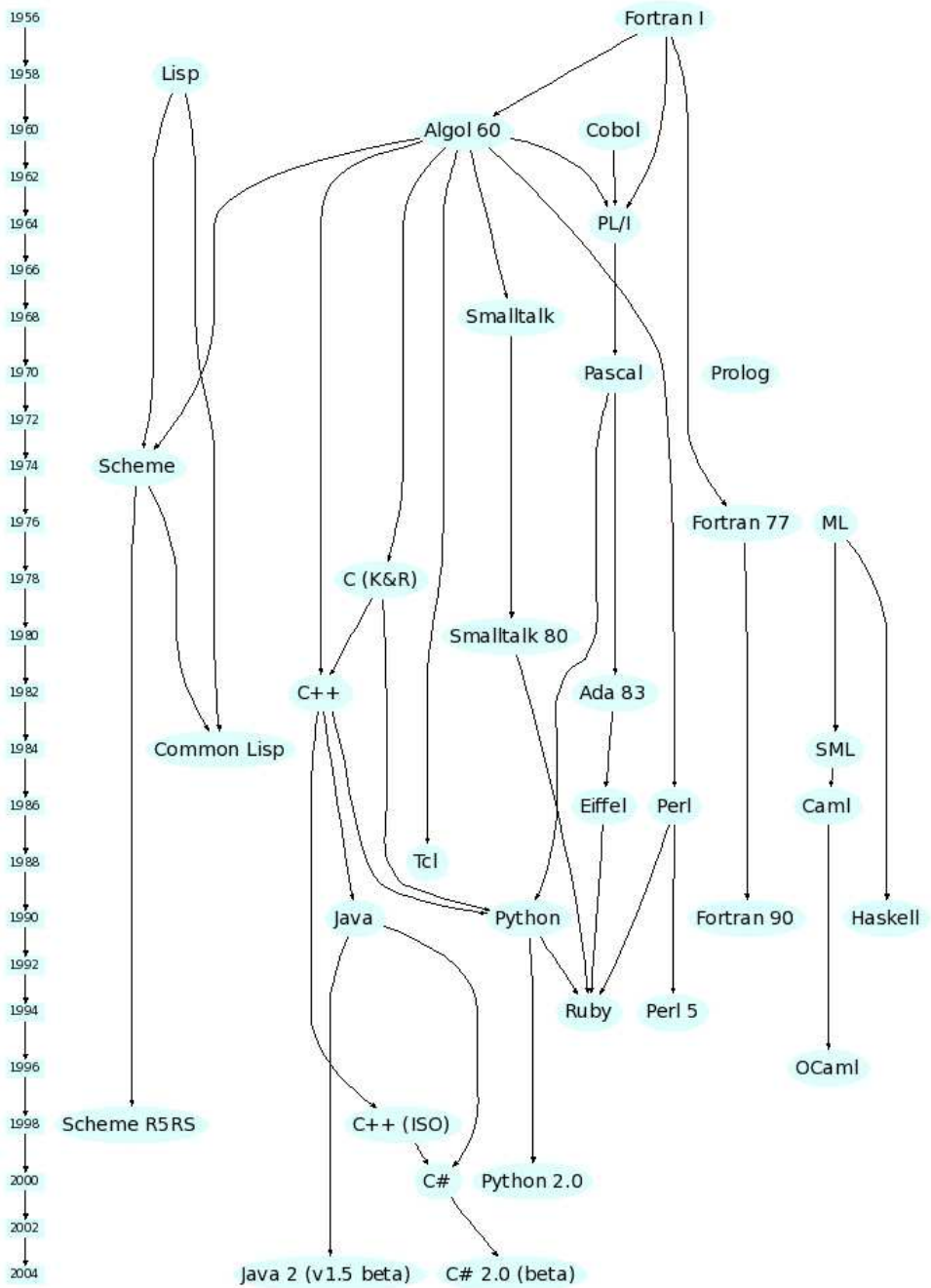
Literatur

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

2.1 Geschichte

Stammbaum

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


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

2.1.1 Imperative Sprachen

Maschinen-Code, Ende 1940er, Anfang 1950er

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

Assembler-Sprachen, Anfang 1950er

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

Plankalkül (1945), Konrad Zuse

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

Fortran, 1954

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

Fortran, 1954

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

ALGOL, Ende 1950

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

COBOL, 1960

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

BASIC, 1964

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

PL/I, 1965

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

SIMULA, 1967

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

Pascal, 1971

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

C, 1971

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

Ada, 1983

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

Smalltalk, 1980

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

C++

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

Java, 1994

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

C#, 2002

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

2.1.2 Logische Sprachen

Prolog, 1972

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

2.1.3 Funktionale Sprachen

LISP, Ende 1950er

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

Scheme, 1975

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

COMMON LISP, 1984

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

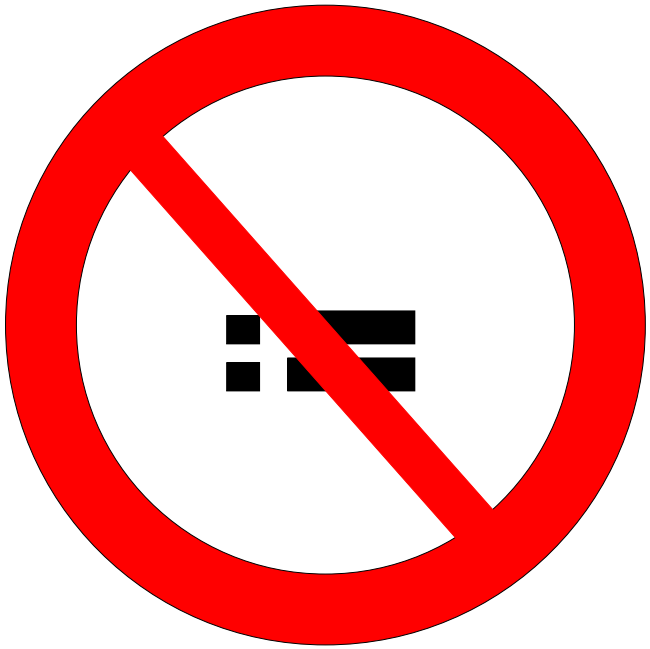
ML, 1980

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

Haskell, 1992

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

3 Funktionale Programmierung



Funtkionale Programmierung vs. traditionelle (prozedurale) Programmierung

- Traditionelle Programmierung
  - ▷ Berechnungsbeschreibung = Folge von Befehlen

    $\Longrightarrow$ imperativ

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

    Zustand $\equiv$ Inhalt der Speicherzellen

    Zustandsänderung durch Zuweisung:

    $$Speicherzelle := Wert$$

    $\Longrightarrow$ zuweisungsorientiert

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

Funtkionale Programmierung vs. traditionalle (prozedurale) Programmierung

- Funktionale Programmierung (FP)
  - ▷ Berechnungsbeschreibung = (mathematische) Funktion
    $\Longrightarrow$ deklarativ

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

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

Funtkionale Programmierung vs. traditionalle (prozedurale) Programmierung

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

3.1 Merkmale der FP

3.1.1 Keine Seiteneffekte

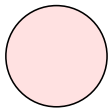
- Der Aufruf einer Funktion mit dem selben Parameter liefert immer das selbe Ergebnis
- Aufruf $\equiv$ Anwendung einer mathematischen Funktion
- 1. Vorteil: Unabhängigkeit von der Auswerttungsreihenfolge (z.B. der Parameter)

Fibonacci-Bäume

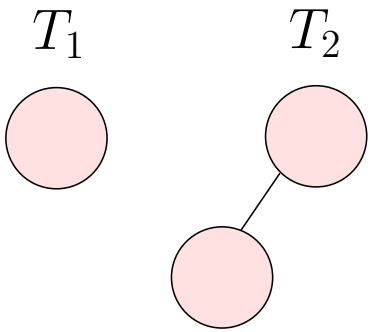
- **Definition**:
  - ▷ Der leere Baum ist ein Fibonacci-Baum der Höhe 0
  - ▷ Ein einzelner Knoten ist ein Fibonacci-Baum der Höhe 1
  - ▷ Sind T_{h-1} und T_{h-2} Fibonacci-Bäume der Höhen $h-1$ und $h-2$, so ist $T_h = k < T_{h-1}, T_{h-2} >$ ein Fibonacci-Baum der Höhe h.
- sind ein Speziallfall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

Fibonacci-Bäume

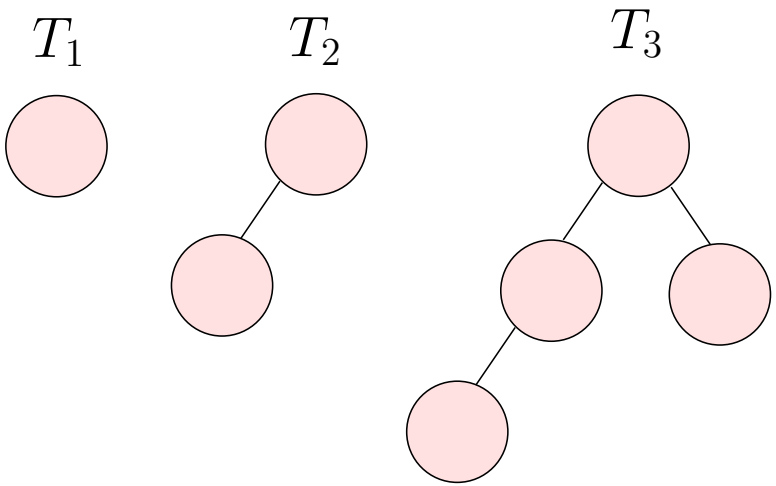
T_1



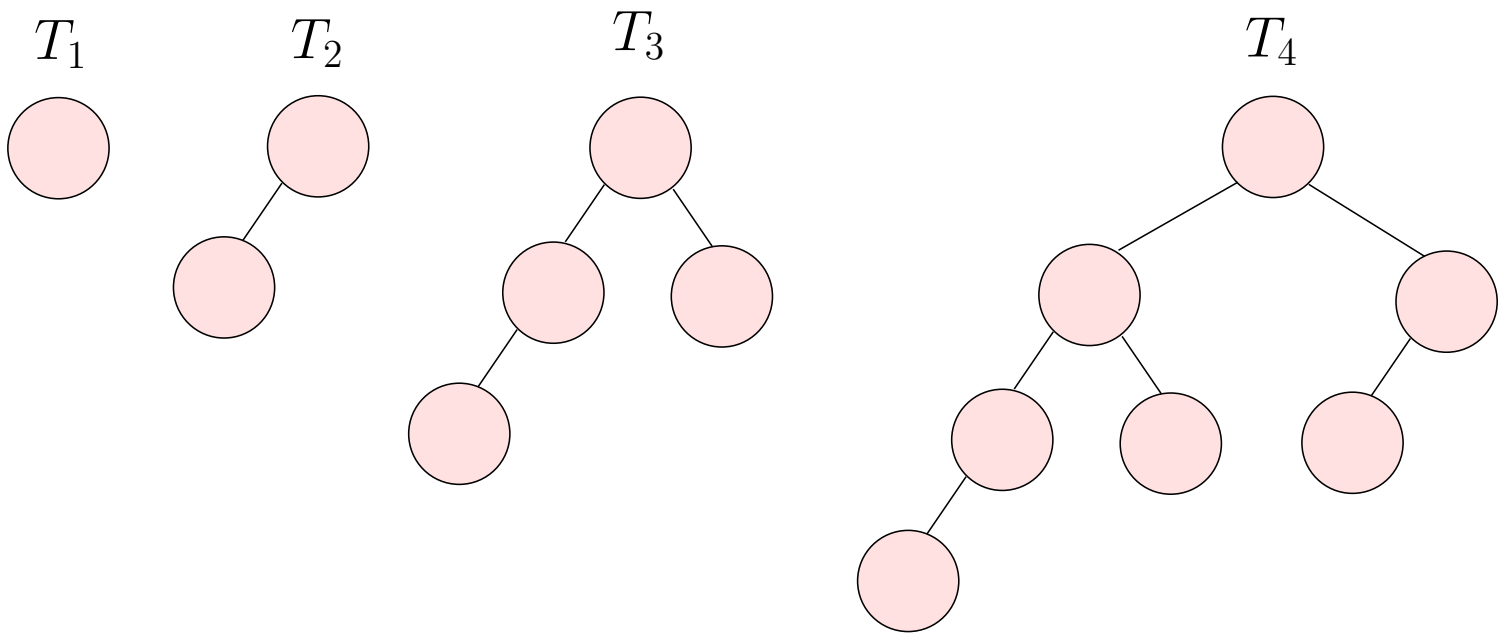
Fibonacci-Bäume



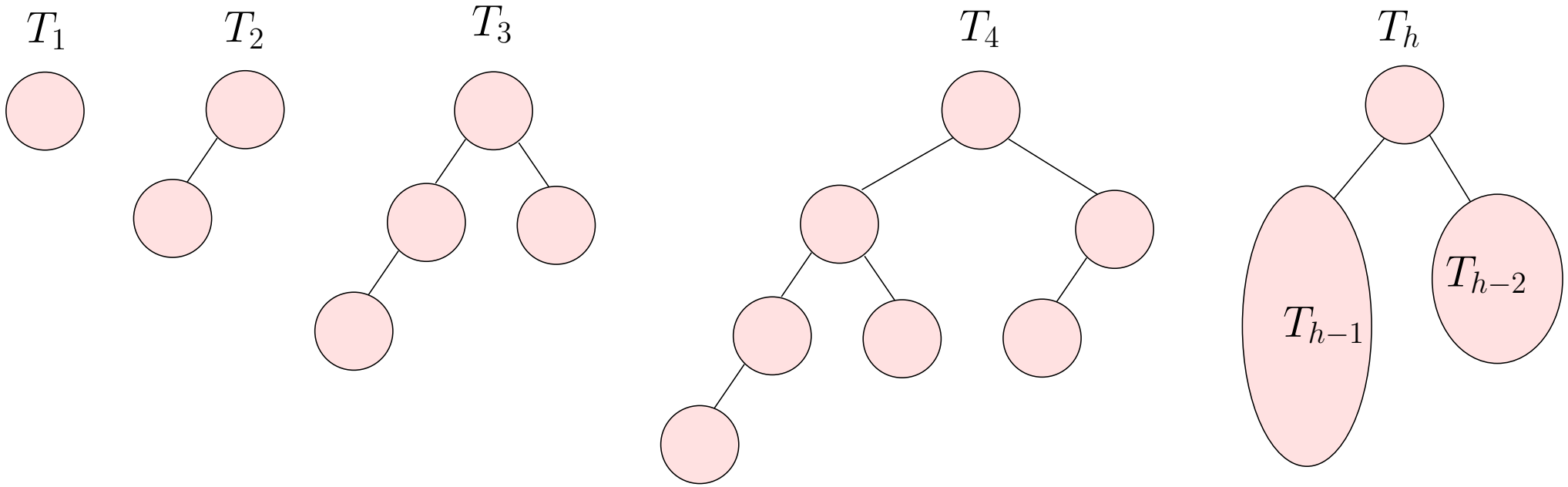
Fibonacci-Bäume



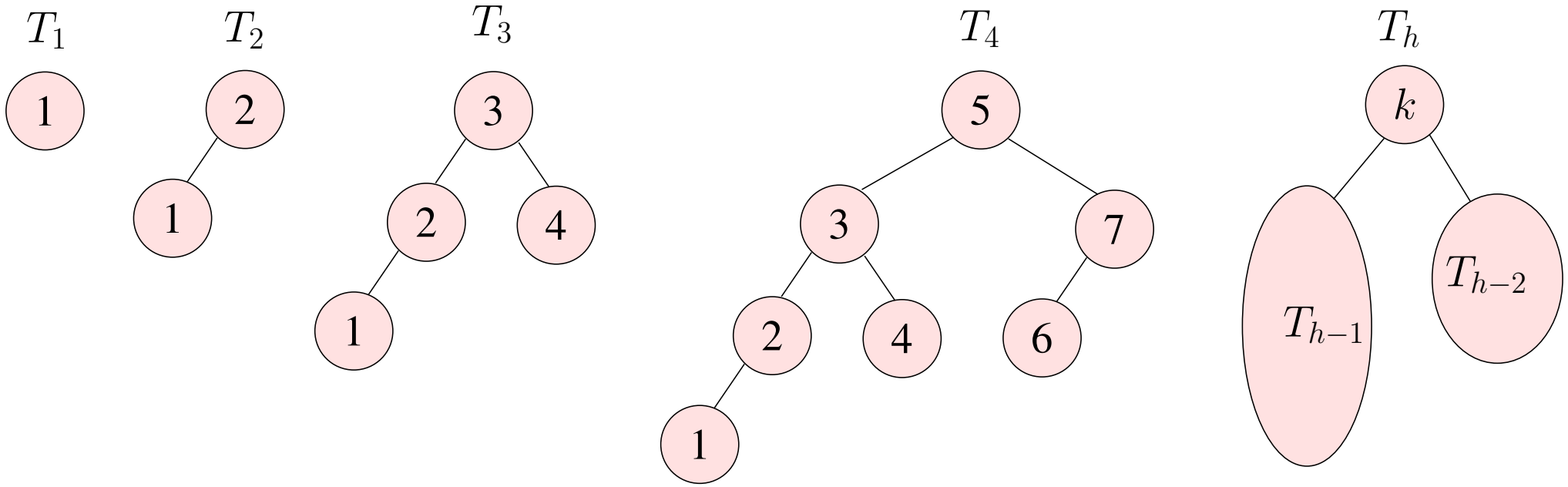
Fibonacci-Bäume



Fibonacci-Bäume



Natürlich annotierte Fibonacci-Bäume



... sind ein Spezialfall von binären Suchbäumen:

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

Natürlich annotierte Fibonacci-Bäume

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

Imperative Lösung (in Java)

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

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

    .......
```

Imperative Lösung (in Java)

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

static int m = 1;

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

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

Probleme der iterativen Lösung

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

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

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

    $\Longrightarrow$ das Ergebnis ist nicht deterministich

Probleme der iterativen Lösung

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

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

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

Beweisidee (für Java)

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

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

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

Funktionale Lösung (immer noch in Java)

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

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

Funktionale Lösung (immer noch in Java)

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

Behauptung: *t(h,k) ist ein Suchbaum.*

Korrektheitsbeweis

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

Beweis: Induktion über h.

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

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

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

$\Longrightarrow$

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

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

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

Beweis:

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

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

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

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

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

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

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

  Folgt direkt per Induktion über h.

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

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

Ursprüngliche Variante

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

Effizientere Variante:

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

Es geht noch effizienter. Idee:

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

  Hilfklasse `Pair`

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

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

Dritte Variante

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

3.1.2 Referenzielle Transparenz

- Abwesenheit von Seiteneffekten sorgt für referenzielle Transparenz (*referential transparency*)
- Referenzielle Transparenz: der Wert eines Programmausdruckes hängt nur von den Werten der Teilausdrücken und nicht von dem Kontext der Auswertung; eine (implizite) Referenz zum (dynamischen) Kontext der Auswertung ist nicht nötig/sichtbar
- Allgemeiner: der Wert eines Funktionsaufrufs hängt nur von den Werten der aktuellen Parameter $\Longrightarrow$ referenzielle Transparenz $\equiv$ keine Seiteneffekte

Referentielle Transparenz/Keine RT

- Funktionale Programmierung (SML):

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

- Imperative Programmierung (Java/C++):

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

Beispiel: Keine Referentielle Transparenz

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

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

Ausgabe: erst 1 dann 2.

Folgen der referentiellen Transparenz

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

3.1.3 Funktionen sind Werte erster Klasse

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

  $\Longrightarrow$ Können als Parameter übergeben werden

  $\Longrightarrow$ Können als Rückggabewerte von Funktionen zurückgeliefert werden

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

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

Beispiel: Funktionskomposition

Versuch in C:

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

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

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

Ausgabe: `res=5`

Beispiel: Funktionskomposition

Versuch in C:

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

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

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

Probleme:

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

Lösung in SML:

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

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

- Ausgabe: `81`

3.1.4 Weitere Merkmale von FP

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 Strukturen zur expliziten Kontrolle des Kontrollflußes
  - ▷ keine Schleifen $\Longrightarrow$ Iteration durch Rekursion ersetzt

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

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:

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

3.1.5 Typische Eigenschaften der FP-Sprachen

- Funktionen sind Werte erster Klasse
- Gute Unterstützung durch das Typ-System (Polymorphie, Typ-Inferenz)
- (Möglichst) keine Seiteneffekte
- Hoher Abstraktionsgrad (Programme sind eher für das Problem als für die Maschine repräsentativ)
- Fall-Unterscheidung für Funktionen durch Muster-Angabe (*pattern matching*)
- Dekomposition eines Wertes durch Pattern-matching
- Automatische Speicherfreigabe (*garbage collection*)

4 Funktionale Programmierung in SML

- ML, 1973 Robin Milner (Meta-Language: 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.

4.1 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)

4.2 Vordefinierte Datentypen

- Eine funktionale Programmiersprache arbeitet mit einer Menge von Typen
- Ein Typ ist eine Menge von Werten (Konstanten)
- Typen werden als Definitions- und Bildbereich für Funktionen (Operatoren) benutzt

Vordefinierte Basis-Typen in SML

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

Typen werden vom Compiler (meist) automatisch erkannt:

```
- 1;
val it = 1 : int

- 1.0;
val it = 1.0 : real

- ~3.0/4.0;
val it =  0.75 : real

- "So" ^ " " ^ "geht" ^ " " ^ "das";
val it = "So geht das" : string

- 1>2 orelse not (2.0<1.0);
val it = true : bool
```

4.3 Variablen

- 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
```

Variablen

- 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
```

4.4 Funktionen

4.4.1 Funktionen definieren (erste Methode)

- 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
```

4.4.2 Funktionsanwendung

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
```

4.4.3 Funktionen definieren (zweite Methode)

Für die Definition von Funktionswerte gibt es auch eine verkürzte Syntax. 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
```

Rekursive Funktionen

- Ein Wert wie `fn x => x` ist eine namenlose (anonyme) Funktion
  $\Longrightarrow$ keine Definition rekursiver Funktionen möglich
- Nicht namenlose Funktionen erlauben die Definition von rekursiven Funktionen:

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

4.5 Definition neuer Typen

Sei MT die Menge der Typen in der Sprache.

Neue Typen können mit Hilfe von Operatoren konstruiert werden (ähnlich wie Ausdrücke aufgebaut werden), die die Form haben:

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

Solche Operatoren heißen Typ-Operatoren.

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

4.5.1 Produkt-Typen

Der Typ-Operator $*$

$$* : MT^n \mapsto MT \text{ mit } n \geq 2$$

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

Beispiele:

- $int * int = \{(x, y) \mid x, y \in int\}$
  $(1, 2) \in int * int$
- der Typ-Operator $*$ steht zwischen Operanden (*infix operator*)

Produkt-Typ: Beispiele

```
- (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
```

Produkt-Typ: Beispiele

```
- (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
```

4.5.2 Records-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",alter="23"};
val p2 = {alter="23",name="Smith",vorName="Jan"}
 : {alter:string, name:string, vorName:string}

- val p3 = {vorName="Jan",name="Smith"};
val p3 = {name="Smith",vorName="Jan"}
 : {name:string, vorName:string}
```

Records

- Zwei Record-Typen sind gleich wenn sie gleich viele Komponente haben, jeweils mit dem selben Namen und dem selben Typ:

```
- p2 = p3;
stdIn:41.1-41.8 Error: operator and operand don't agree [tycon mismatch]
  operator domain: {alter:string, name:string, vorName:string}
                 * {alter:string, name:string, vorName:string}
  operand: {alter:string, name:string, vorName:string}
                 * {name:string, vorName:string}
  in expression:
    p2 = p3
```

Records

- Zwei Record-Werte (≡ Records) sind gleich, wenn sie vom selben Typ sind, und die jeweiligen Komponenten gleich sind

```
- 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",alter="23"};
val p2 = {alter="23",name="Smith",vorName="Jan"}
 : {alter:string, name:string, vorName:string}

- p1 = p2;
val it = false : bool
```

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
```

4.5.3 Summen-Typen

- Da ein Typ eine Menge von Werten ist, kann man ihn angeben, indem man spezifiziert, wie die Werte konstruiert werden.
- Werte eines Typs werden mit Hilfe von (Typ-)Konstruktoren aufgebaut.
- Bei endlichen Typen kann man alle Elemente aufzählen
  $\longrightarrow$ Aufzählungstypen.

Aufzählungstypen (*enumeration types*)

- werden durch das Schlüsselwort `datatype` definiert.
- bestehen aus Konstanten (nullstellige Konstruktoren) mit symbolischen Namen getrennt durch "`|`".

```
- 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
```

Aufzählungstypen (*enumeration types*)

- 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)

Besser als ad-hoc Kodierungen: Aufzählungstypen

```
- 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 | ...`

Verallgemeinerung: Summentypen

- Im Allgemeinen können nicht alle Werte eines Typen aufgezählt werden.
- Stattdessen konstruiert man neue Werte aus Werten eines anderen Typen $\mapsto$ einstellige Konstruktoren.
- Ein neuer Typ wird definiert indem man alle seine (nullstelligen und einstelligen) Konstruktoren spezifiziert $\mapsto$ Summentypen
- werden mit Hilfe von `datatype` eingeführt

  `datatype =` $Konstruktor_1$ `|` $Konstruktor_1$ `|` $\ldots$ `|` $Konstruktor_n$

Summentypen

Ein Konstruktor c eines Typen $T \in MT$ kann als Funtion aufgefasst werden:

- nullstellig: $c : \bullet \mapsto T$
- einstellig: $c : T_1 \mapsto T$ mit $T_1 \in MT$
- Beispiel:

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

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

Summentypen: Beispiel

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

4.5.4 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

$$\texttt{case}\ Ausdruck\ \texttt{of}\ Muster_1 \quad => \quad Ausdruck_1$$
$$\ldots$$
$$|Muster_n \quad => \quad Ausdruck_n$$

- $Muster_1$, $Muster_2 \ldots$, $Muster_n$ spezifizieren die zutreffende Struktur via Konstruktoren und Variablen
- $Ausdruck_1$, $Ausdruck_2 \ldots$, $Ausdruck_n$ müssen alle den selben Typ haben; das ist der Typ des gesamten case-Ausdrucks
- Wenn $Muster_i$ das erste zutreffende Muster (*pattern*) für den Wert von *Ausdruck* ist, ist der Wert des case-Ausdrucks gleich der Wert von $Ausdruck_i$.

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 black"
                    | RGB(80,200,130) => "Possibly Jamaica"

val description = "Possibly Jamaica" : string
```

Der If-Ausdruck

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

```
val 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
```

Der If-Ausdruck

Im Allgemeinen:

`case` *Expr* `of` `true` `=>` $Expr_1$ `|` `false` `=>` $Expr_2$	$\equiv$	`if` *Expr* `then` $Expr_1$ `else` $Expr_2$

If ist ein Ausdruck

```
fun handlePlural number what =
  what^(if number>1 then "s" else "")
val handlePlural = fn : int -> string -> string
- handlePlural 5 "student";
val it = "students" : string
```

In imperativen Sprachen wird zwischen Anweisungen und Ausdrücken unterschieden.

In funktionallen Sprachen sind alle Konstrukte Ausdrücke.

Der If-Ausdruck

```
if Expr then Expr1
else Expr2
```

$Expr_1$ und $Expr_2$ müssen den selben Typ haben.

```
fun inverse x =
   if x>0.0001 then 1.0/x else "a to large number";
stdIn:43.17-43.52 Error: types of rules don't agree [tycon mismatch]
 earlier rule(s): bool -> real
 this rule: bool -> string
 in rule:
 false => "error"
```

Pattern-Matching mit Variablen-Bindung

- Patterns können 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 "white"
                      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
```

4.5.5 Rekursive Typen

Die Definition eines Typen kann rekursiv sein, d.h.
Typ-Konstruktoren dürfen Elemente des zu definierenden Typ erhalten.

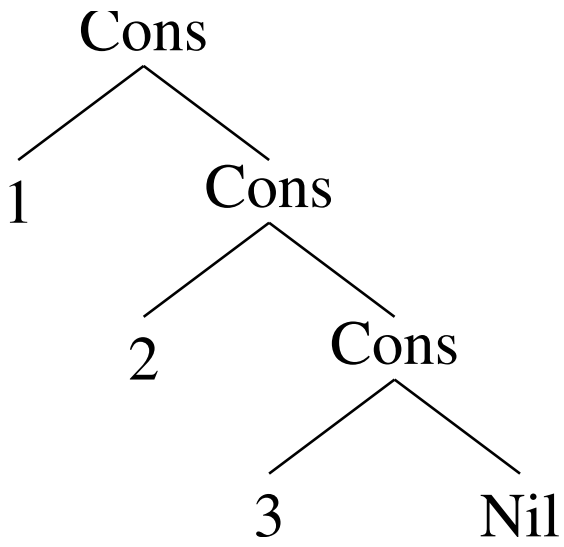
```
datatype IntList = Nil | Cons of (int*IntList);
```

Damit Werte konstruierbar sind, muss midestens 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
```

4.5.6 Polymorphismus

Polymorphe Typen

Listentypen unterscheiden sich nur in den Typ ihrer Elemente:

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

```
datatype RealList = Nil | Cons of (real * RealList);
```

Besser als verschiedene Definitionen: parametrisieren:

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

'a ist ein Bezeichner für einen Typ (Typ-Variable)

Polymorphe Typen

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

Der Typ `List` ist parametrisiert ≡ `List` ist ein (postfix) Typ-Operator.

Diese Art von Polymorphismus heißt parametrischer Polymorphismus.

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)
```

Der Compiler leitet den allgemeinst möglichen Typ ab:

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

Hier ist 'a als *ein beliebiger Typ* zu lesen

$\Longrightarrow$ `length` is polymorph:

Polymorphe Funktionen

```
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 polypmorpher Typ `'a list` ist vordefiniert.

Konstruktoren:

- nullstellig: `nil` (entspricht unserem `Nil`)
- zweistellig: `::` (entspricht unseren `Cons`)
  - ▷ steht zwischen seinen Argumenten: `kopf::rest`
  - ▷ `::` ist rechtsassoziativ:
    `1::(2::(3::nil))` ≡ `1::2::3::nil`
- Alternative Klammernotation:
  - ▷ `[]` ≡ `nil`
  - ▷ `[1,2,3]` ≡ `1::(2::(3::nil))` ≡ `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 `int`s: `[[1,2,3],[4,5],[6],[7,8,9]]`

List-Typen: Beispiele

- Alle Elemente einer Liste müssen vom selben Typ sein:

```
- [1,[1]];
stdIn:93.1-93.8 Error: operator and operand don't agree [literal]
 operator domain: int * int list
 operand: int * int list list
 in expression:
 1 :: (1 :: nil) :: nil
```

- Liste von Listen von `int`s:

```
- [[1,2,3],[4,5],[6],[7,8,9]];
val it = [[1,2,3],[4,5],[6],[7,8,9]] : int list list
```

Polymorphe Typen: Beispiele

- Binäre Bäume mit Informationen nur an Blättern:

```
datatype 'a BinTree = Leaf of 'a
                    | Node of 'a BinTree * 'a BinTree
val t2 = Leaf "text1"
val t2 = Leaf "text1" : string BinTree
val t1 = Node(Leaf 1,Leaf 2);
val t1 = Node (Leaf 1,Leaf 2) : int BinTree
fun height t =
  case t of Leaf _ => 1
          | Node(t1,t2) => 1 + max(height t1,height t2);
val height = fn : 'a BinTree -> int
height t2;
val it = 1 : int
height t1;
val it = 2 : int
```

Polymorphe Typen: Beispiele

- Binäre Bäume mit Informationen an Blättern und inneren Knoten:

```
datatype 'a BinTree1 = Leaf1 of 'a
                     | Node1 of 'a * 'a BinTree1 * 'a BinTree1
fun height1 t =
  case t of Leaf1 _ => 1
          | Node1(_,t1,t2) => 1 + max(height t1,height t2);
val height1 = fn : 'a BinTree1 -> int
```

- Bäume beliebiger Stelligkeit:

```
datatype 'a Baum = Blatt of 'a | Knoten of 'a * 'a list;
```

Pattern-Matching: Einschränkungen

Nur Konstrukoren:

```
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
```

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

```
fun elimDouble l =
  case l of x::x::rest => x::(elimDouble rest);
stdIn:67.20-67.64 Error: duplicate variable in pattern(s): x
```

4.6 Mehr über Variablen

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

4.6.1 Variablendefinitionen

- ... 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

4.6.2 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;
```

Variablengültigkeit

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

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

- Pattern-Variablen sind gültig in der entprechenden 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";
```

Benutzer-definierte 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 Bereich einer Funktionsdefinitionen 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

Static scoping Der Gültigkeitsbereich einer Variablendefinition in SML und in den meisten modernen Programmiersprachen ist definiert durch die (statische) Struktur des Programmtextes:

An welchen Programmpunkten eine Variablen-Definition (zu bestimmten Zeitpunkten) gültig ist, hängt nur von der (statischen) Struktur des Programmtextes ab.

$\Longrightarrow$ *static/lexical scoping*$\equiv$ *static/lexical variable binding*

Gültige Variablen-Definitionen am Programmpunkt •

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

Gültige Variablen-Definitionen am Programmpunkt •

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

Gültige Variablen-Definitionen am Programmpunkt •

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

Dynamischer Gültigkeitsbereich

Dynamic scoping An welchen Programmpunkten eine Variablen-Definition gültig ist, hängt davon ab, wie diese bei der Laufzeit erreicht werden.

Dynamic Scoping: Beispiel

Scheme-Beispiel:

```
(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

$\Longrightarrow$ Die referentielle Transparenz ist verletzt

4.6.3 Variablen-Sichtbarkeit

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
     •P x+1
   end) + •P1 x
end
```

- Beide $x \leftarrow 1$ und $x \leftarrow 2$ sind gültig an P.
  Nur $x \leftarrow 2$ ist sichtbar an P.
- Nur $x \leftarrow 1$ ist gültig und sichtbar an P_1.

Variablen-Sichtbarkeit: Beispiel

```
fun fact n = •P if n=1 then 1
                  else n * (fact (n-1))
fact 2;
```

Sichtbare Variablen am P:

Zeit	Variablen-Definition (implizit)	gültig	sichtbar
t_1:	`(fact 2)`	$n \leftarrow 2$	$n \leftarrow 2$
t_2:	`(fact 1)`	$n \leftarrow 2$, $n \leftarrow 1$	$n \leftarrow 1$

4.6.4 Kontext

Der Kontext eines Programmpunkts P zu einem bestimmten Zeitpunkt t besteht aus der Menge der Variablen-Definitionen die sichtbar am P zum Zeitpunkt t sind.
$\Longrightarrow$ 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 = •P if n=1 then 1
                  else n * (fact (n−1))
```

Kontext von P:

Zeit	Kontext
t_1: `(fact 2)`	$n \leftarrow 2$
t_2: `(fact 1)`	$n \leftarrow 1$

4.7 Mehr über Funktionen

4.7.1 Der Typ-Operator für Funktionstypen

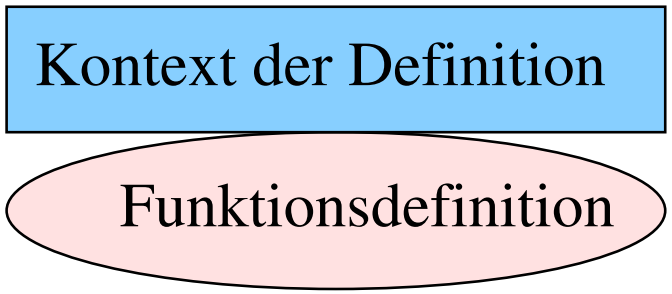
$$-> : MT \times MT \mapsto MT$$
$$\boxed{\alpha -> \beta} = \{f : \alpha \mapsto \beta \mid \alpha, \beta \in MT\}$$

Der Typ-Operator $->$ ist rechtsassoziativ:

$$\alpha \mapsto \beta \mapsto \gamma \equiv \alpha \mapsto \beta \mapsto \gamma$$

4.7.2 Funktionale Abschlüsse

Eine Funktion besteht aus der Funktionsdefinition und der Kontext des Programmpunktes an welchen die Funktion definiert (≡ konstruiert) wird.



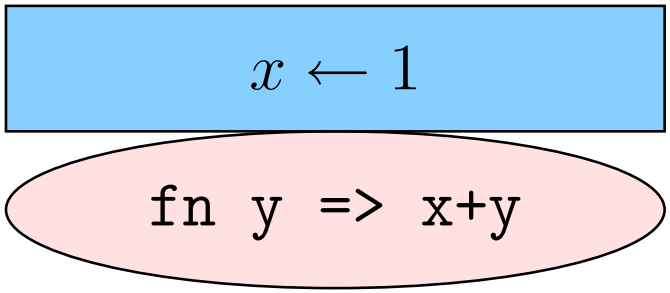
Man sagt, dass Funktionen ihren Kontext zu dem Zeitpunkt ihrer Definition abschließen.

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
```



4.7.3 Currying

- Currying = Methode mit der man Funktionen von mehreren Argumenten konstruieren kann.
- genannnt nach dem Erfinder, Haskell B. Curry

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

`sum` erwartet ein `x` Argument 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 (*curried functions*)

Funktionsanwendung (Aufruf):

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

Die Funktionsanwendung ist linksassoziativ:

`f x1 x2 x3` $\equiv$ `(((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 angewendet werden als in der Deklaration
- Liefern dann eine Funktion zurück, die den Rest der Argumente erwartet

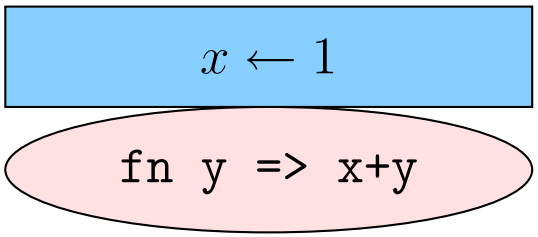
($\Longrightarrow$ Curry-Funktionen sind Funktionen höherer Ordnung)

```
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
```

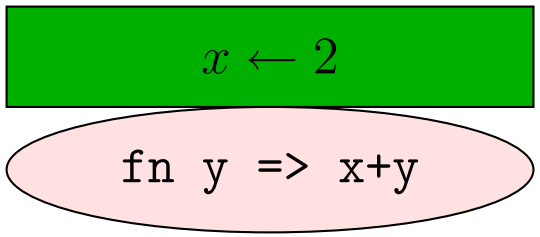
Partielle Anwendung: Beispiel

```
- 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
```

succ



succ2



4.7.4 Funktionen höherer Ordnung: Beispiele

... 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 zurück.

```
- 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: Beispiele

... bekomen Funktionen als Argumente:

- Bsp.: `filter p l` wendet `p` auf jedes Element `x` aus `l` an und liefert die Liste aller `x` zurück, 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: Beispiele

... liefern Funktionen als Ergebnisse zurück:

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

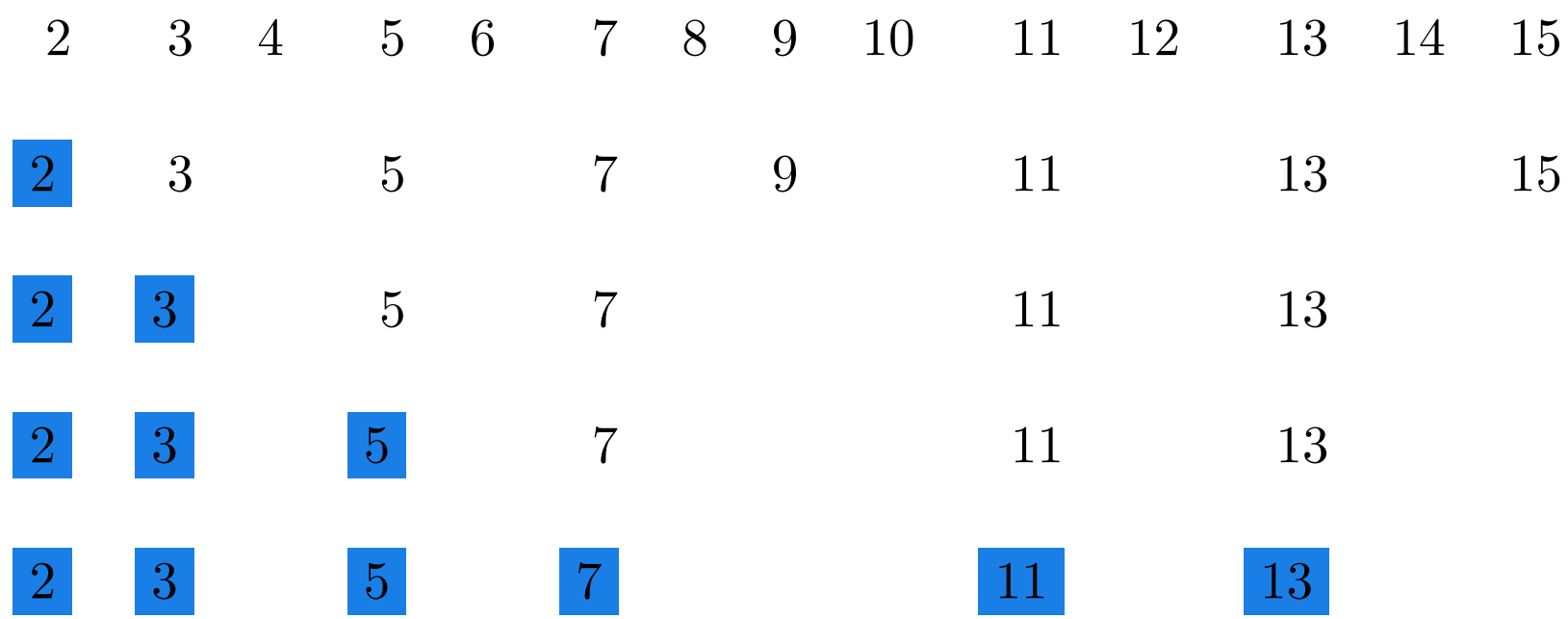
Int.max(2,7);
val it = 7 : int

- 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



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
```

4.8 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
- 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 Aurufes `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 Aufrufes `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 $\Longrightarrow$ **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)

4.8.1 Benutzer-kontrollierte Auswertung

Mit Hilfe funktionaller 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
```

4.8.2 Unendliche Datenstrukturen

Ein Vorteil der verzögerten Auswertung: Darstellung infiniter Datenstrukturen.

Erster Versuch: Datentyp zur Darstellung unendlicher Folgen (*streams*, Ströme):

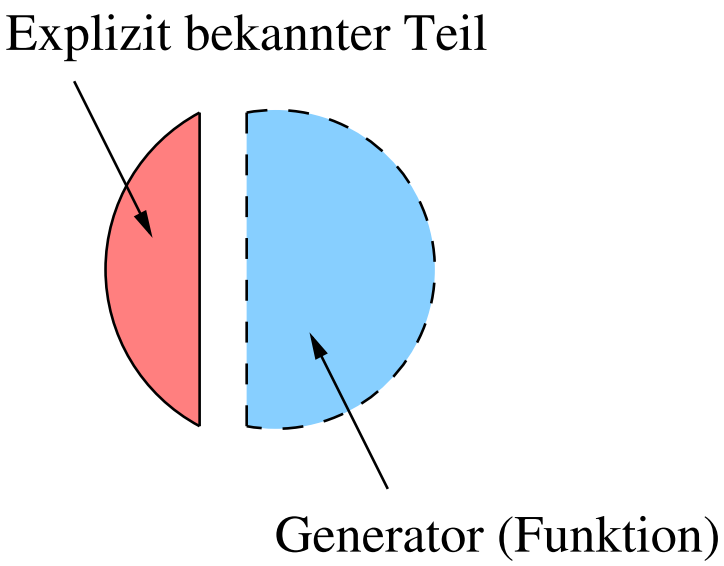
```
- datatype 'a stream = Stream of 'a * 'a stream
  fun generateNat n = Stream (n,generateNat (n+1));
val generateNat = fn : int -> int stream
```

Ein Stream ist ein Paar `Stream(firstTerm,restStream)`:

Wegen der strikten Auswertung terminiert `generateNat 0` nie.

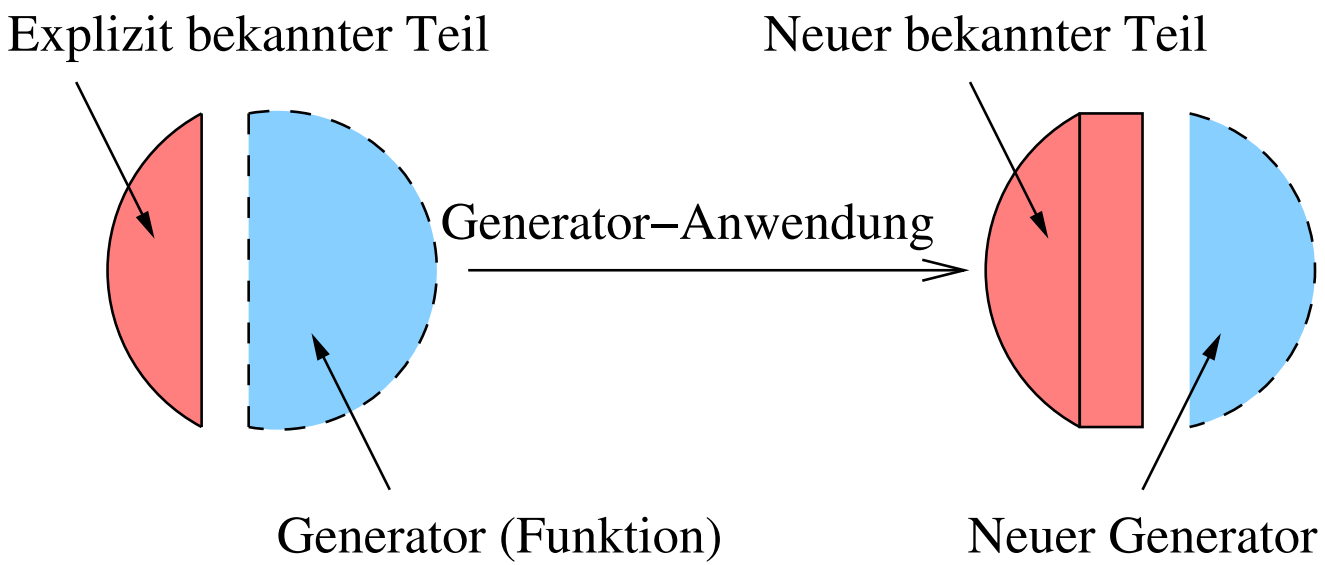
Unendliche Datenstrukturen

Idee:



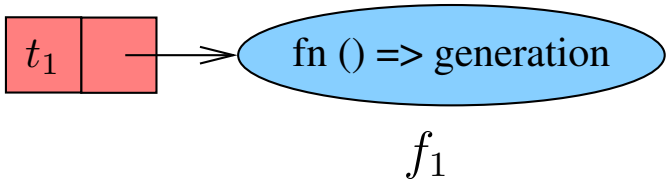
Unendliche Datenstrukturen

Idee:

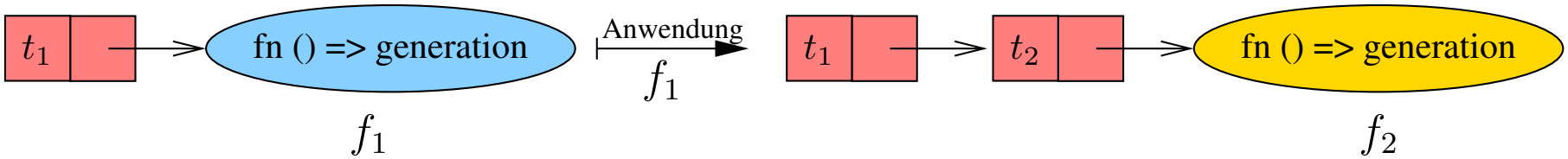


Unendliche Datenstrukturen

Unendliche Datenstrukturen



Unendliche Datenstrukturen



Unendliche Datenstrukturen

Zweiter Versuch: mit funktionallen Abschlüssen:

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

- Dieser Datentyp kann keine endlich großen 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()
  fun nth n s = if n=0 then head s else nth (n-1) (tail s)
- 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

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;

- 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
```

4.9 Typ-Inferenz

Der SML-Compiler erlaubt Typ-Information auszulassen, wenn diese aus dem Kontext herleitbar (inferierbar) ist

$\Longrightarrow$ Typ-Inferenz

- Der Compiler leitet den allgemeinsten (d.h. möglichst polymorphen) Typ her. Z.B. hat der Wert `nil` u.a. die Typen

```
int list
(bool * string list) list
({x: int; y: 'a} list
```

Der allgemeinste Typ ist aber

```
'a list
```

Typ-Ausdrücke

- Typ-Ausdruck = Ausdruck bestehend aus Typ-Konstanten (Basis-Typen), Anwendungen von Typ-Operatoren und Typ-Variablen, die unbekannte, beliebige Typ-Ausdrücke repräsentieren:

▷ `string -> int`

▷ `int * (unit -> int list)`

▷ `'a -> 'a`

▷ `int * 'a`

▷ `'a list * 'b list -> 'a * 'b list`

Typ-Instanzen

- Durch Einsetzen eines Typs für eine Typ-Variable erhält man eine Typ-Instanz:

▷ `int -> int` ist eine Instanz von `'a -> 'a`

▷ `int -> string` ist keine Instanz von `'a -> 'a`

▷ `(int * int) list` ist eine Instanz von `('a * 'b) list`

▷ `(int * ((int -> string) list)) list` ist eine Instanz von `('a * 'b) list`

4.9.1 Typ-Annotationen

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

```
NONE;
val it = NONE : 'a option
NONE : int option;
val it = NONE : int option
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
```

Typ-Annotationen

Der angegebene Typ muss eine Instanz des hergeleiteten Typs sein:

```
[NONE] : 'a list;
stdIn:17.1-17.17 Error: expression doesn't match constraint
  expression: 'Z option list
  constraint: 'a list
in expression: NONE :: 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
```

4.10 Inferenz-Regeln

Typinferenz-Regeln: Beispiel

... besagen wie der Typ eines Ausdrucks aus den Typen seiner Teilausdrüke hergeleitet wird.

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

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

- Die Anwendung der entsprechenden Inferenz-Regel für einen Ausdruck bestimmt eine Menge von Gleichungen (*constraints*), die die Beziehungen zwischen den Typen der Teilausdrücke beschreibt.
- Für einen Ausdruck e ergibt sich ein Gleichungssystem S.
  - ▷ Hat S eine Lösung $\Longrightarrow$ die Auswertung von e führt nie zu einem Typ-Fehler
  - ▷ Hat S keine Lösung $\Longrightarrow$ die Auswertung von e könnte fehlschlagen

Beispiel: Typ-Inferenz für `a+1`

$$\frac{x : \text{int} \qquad y : \text{int}}{x+y : \text{int}} \qquad \frac{\texttt{a} : \texttt{'a} \qquad \texttt{1} : \texttt{int}}{\texttt{a+1} : \texttt{'b}}$$

Durch *Unifikation* der Hypothesen und der Schlussfolgerungen $\Longrightarrow$ Gleichungssystem mit den Typ-Variablen `'a` und `'b`:

$$\left\{\begin{array}{l}\texttt{'a = int}\\ \texttt{int=int}\\ \texttt{'b = int}\end{array}\right. \Longleftrightarrow \left\{\begin{array}{l}\texttt{'a = int}\\ \texttt{'b = int}\end{array}\right. \Rightarrow \texttt{a : int} \text{ und } \texttt{a+1 : int}.$$

Typinferenz: Allgemeine Regeln

Case: $$\frac{e \texttt{ : 'a} \quad p_1 \texttt{ : 'a} \ldots p_n \texttt{ : 'a} \qquad e_1 \texttt{ : 'b} \ldots e_n \texttt{ : 'b}}{\texttt{case } e \texttt{ of } p_1 \texttt{ => } e_1 \texttt{ | } \ldots \texttt{| } p_n \texttt{ => } e_n \texttt{ : 'b}}$$

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

Anwendung: $$\frac{f \texttt{ : 'a} \mapsto \texttt{'b} \qquad a \texttt{ : 'a}}{f \ a \texttt{ : 'b}}$$

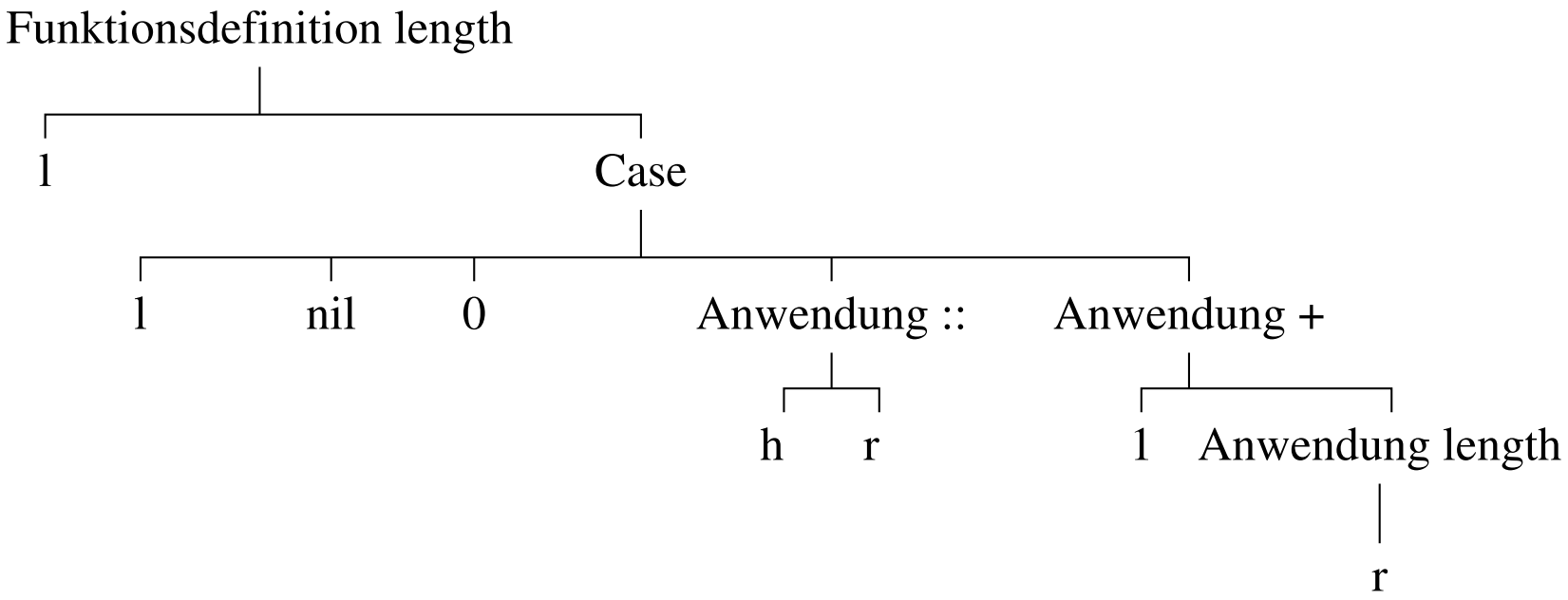
Funktionsdefinition: $$\frac{x \texttt{ : 'a} \qquad e \texttt{ : 'b}}{\texttt{fun } x \texttt{ = } e \texttt{ : 'a} \mapsto \texttt{'b}}$$

Typ-Inferenz

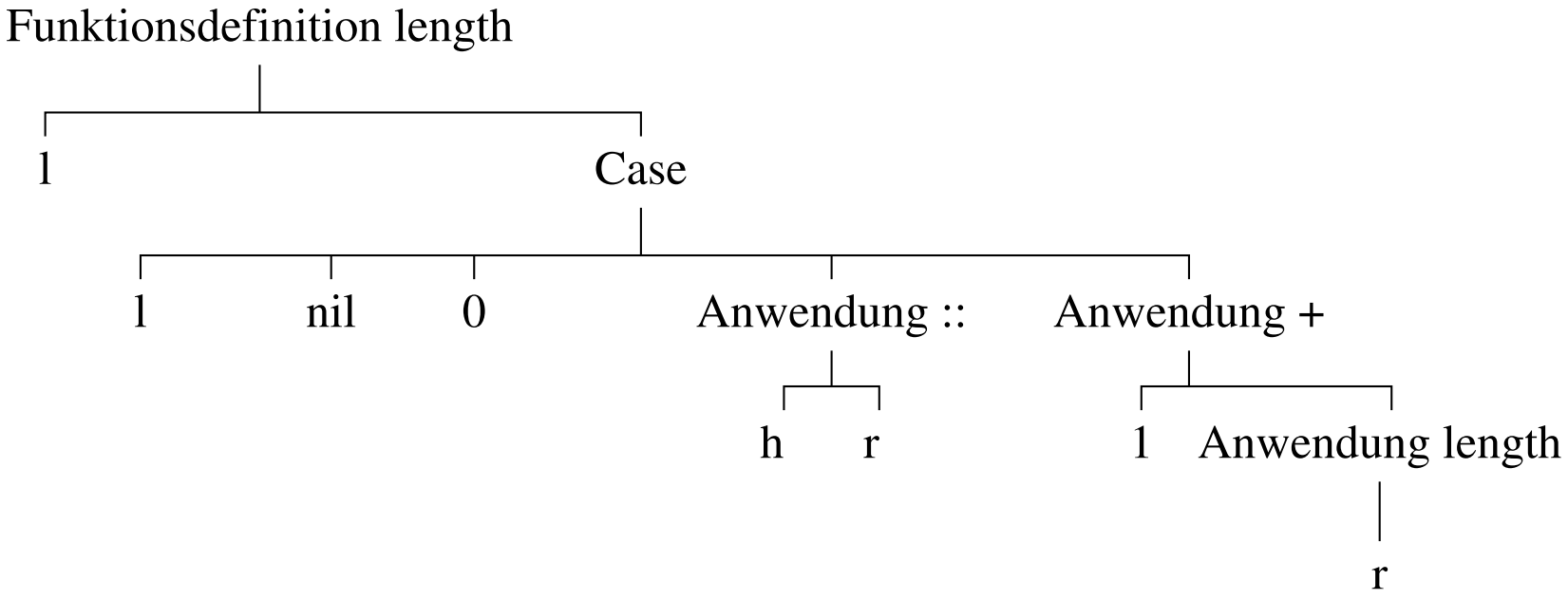
```
fun length l = case l of nil => 0
                       | h::r => 1 + length r
```

Typ-Inferenz

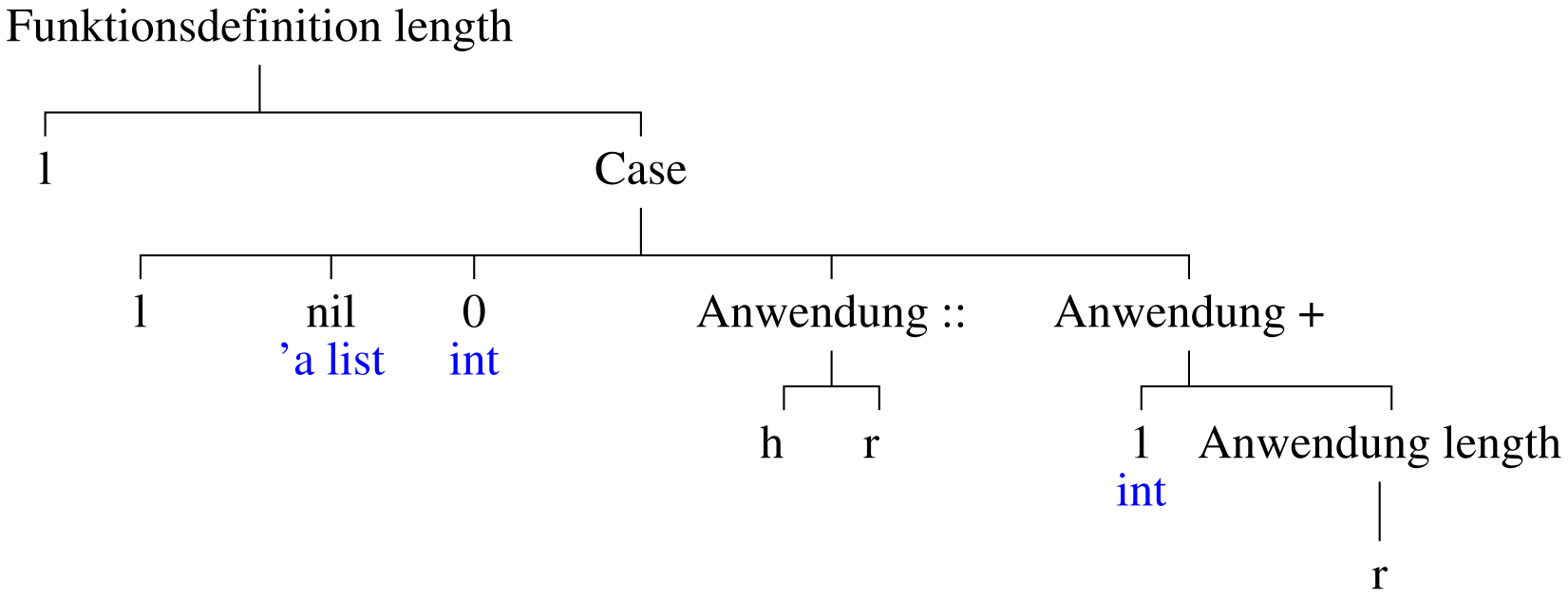
```
fun length l = case l of nil => 0
                       | h::r => 1 + length r
```



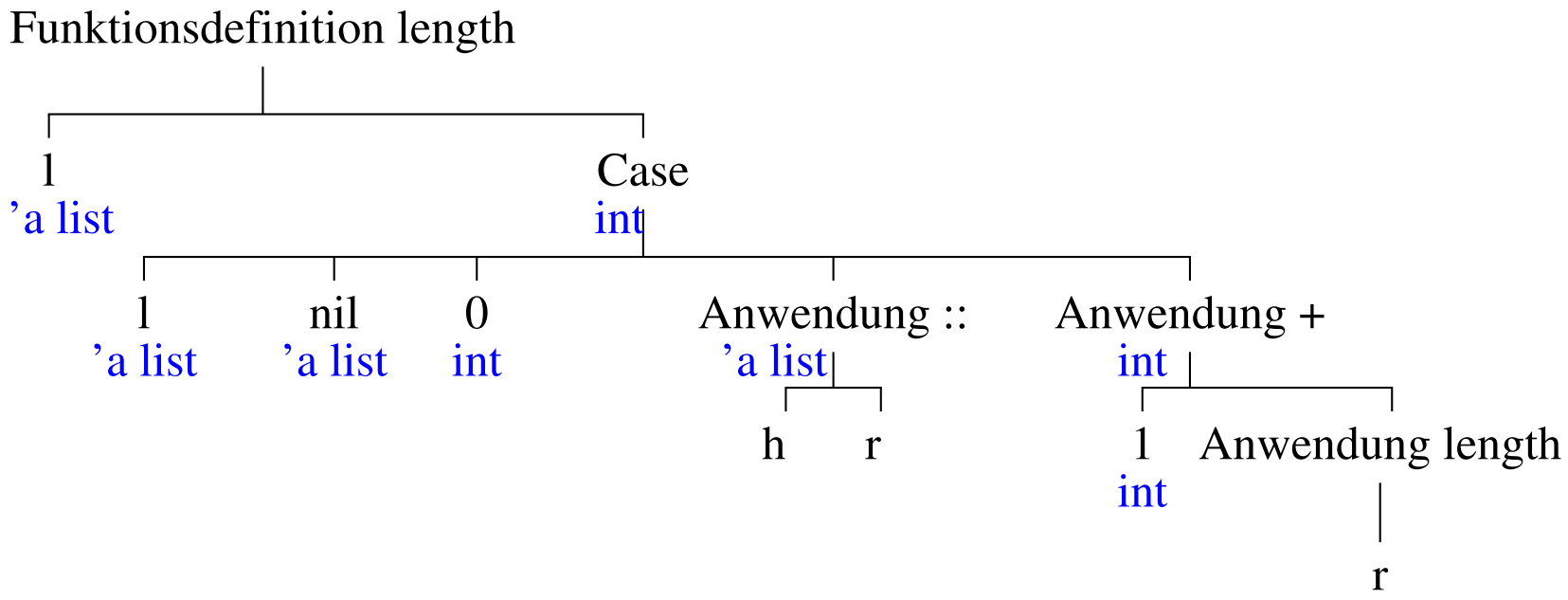
Typ-Inferenz



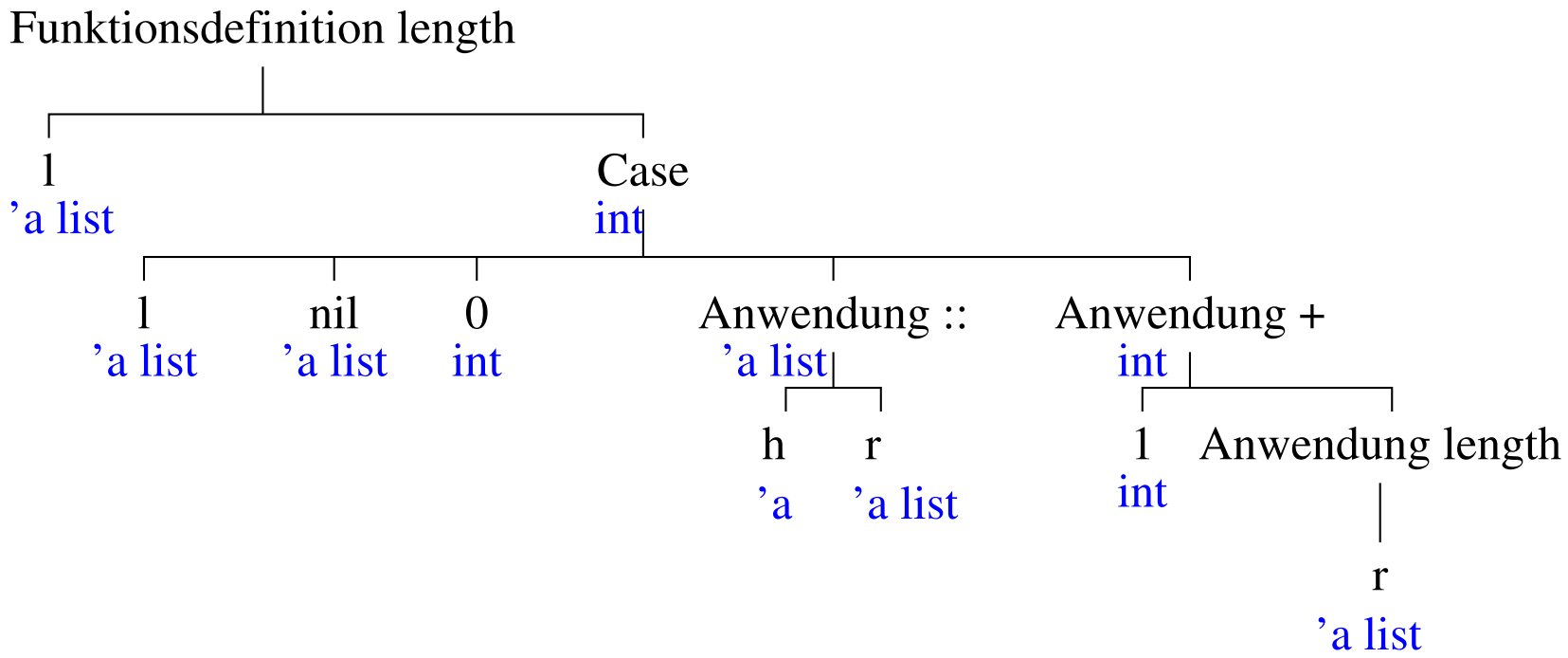
Konstanten


Funktionsdefinition length
l
Case
l
nil
'a list
0
int
Anwendung ::
h
r
Anwendung +
l
int
Anwendung length
r

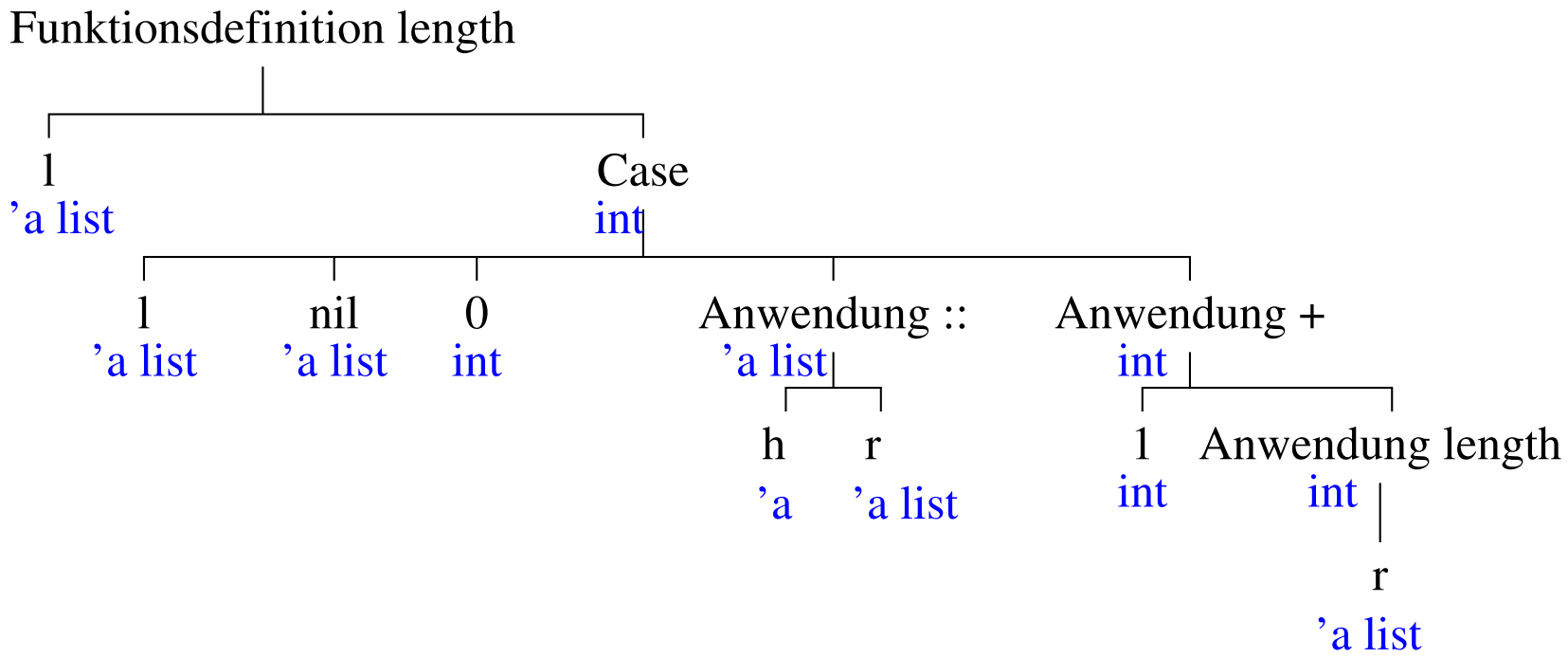
Case


Funktionsdefinition length
l
'a list
Case
int
l
'a list
nil
'a list
0
int
Anwendung ::
'a list
h
r
Anwendung +
int
1
int
Anwendung length
r

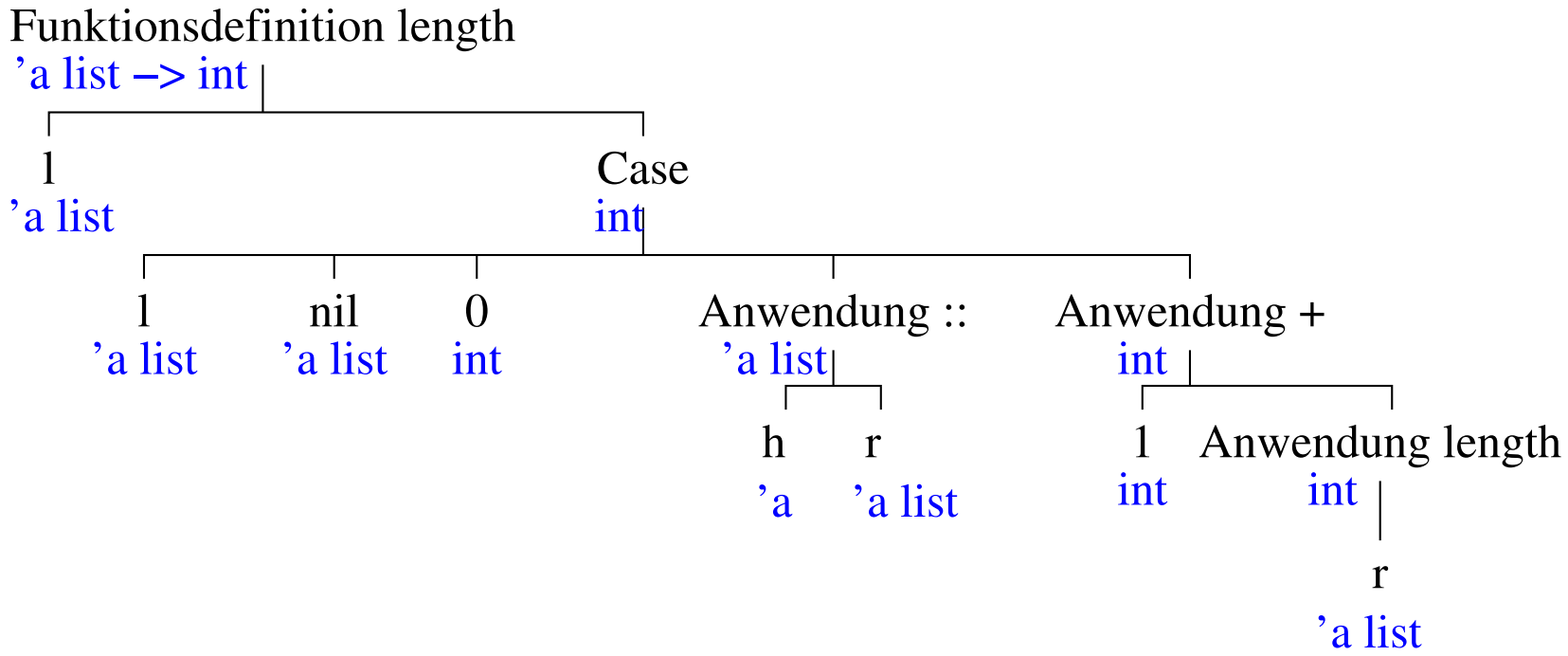
Anwendung des Konstruktors : :



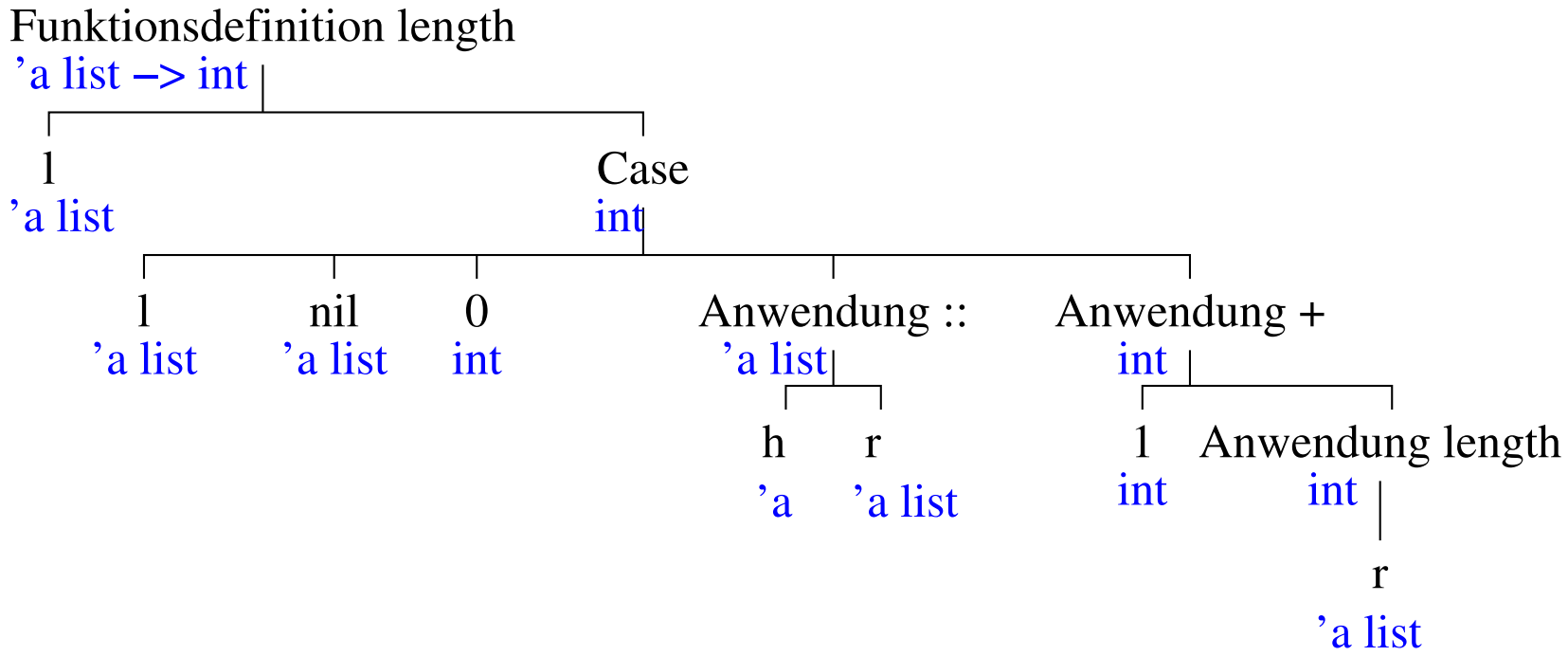
Anwendung der Addition



Anwendung der Funktion `length`



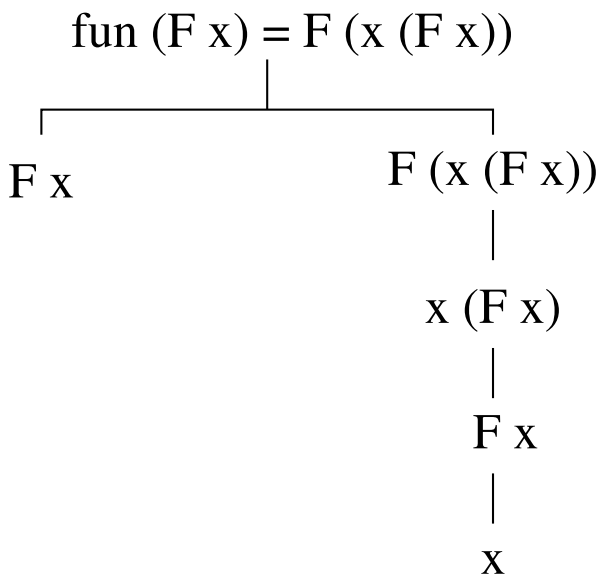
Funktionsdefinition `length`



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

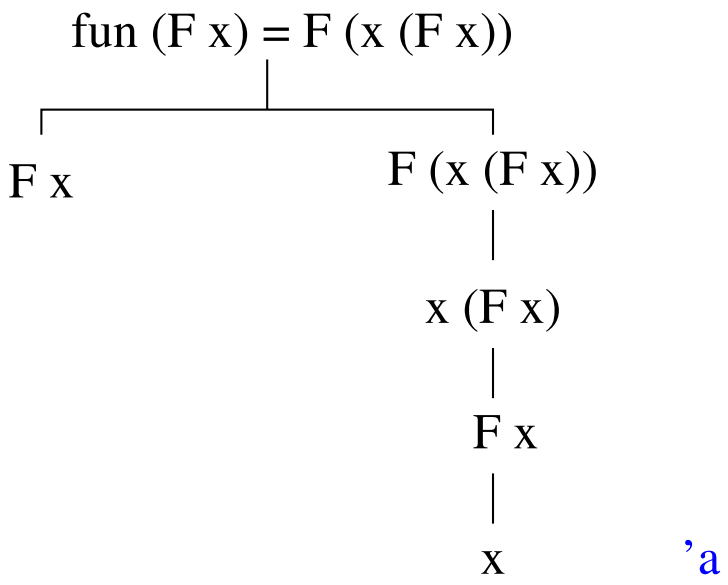
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

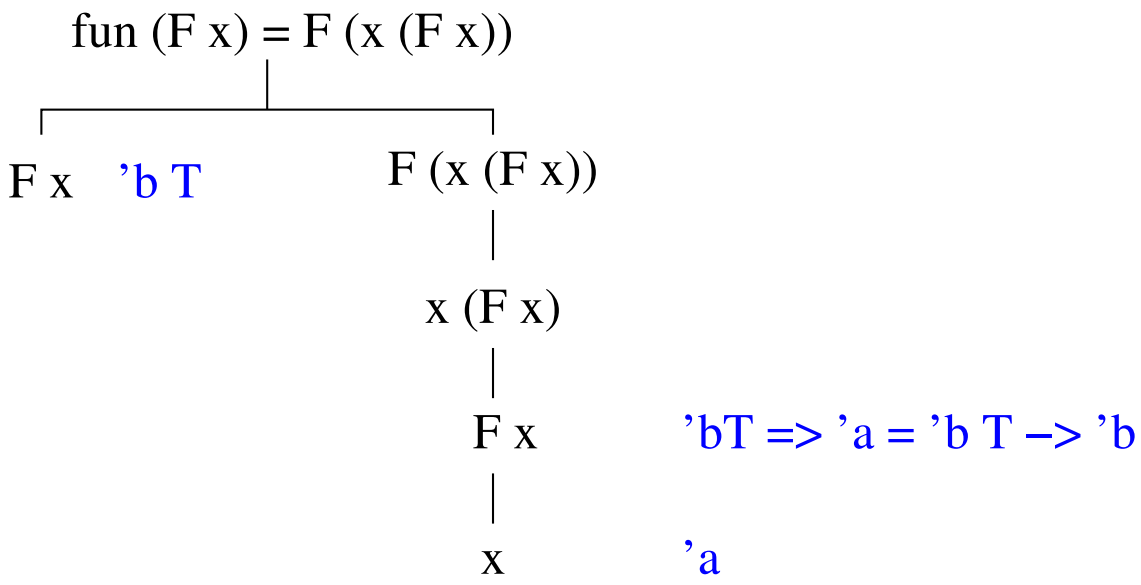
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

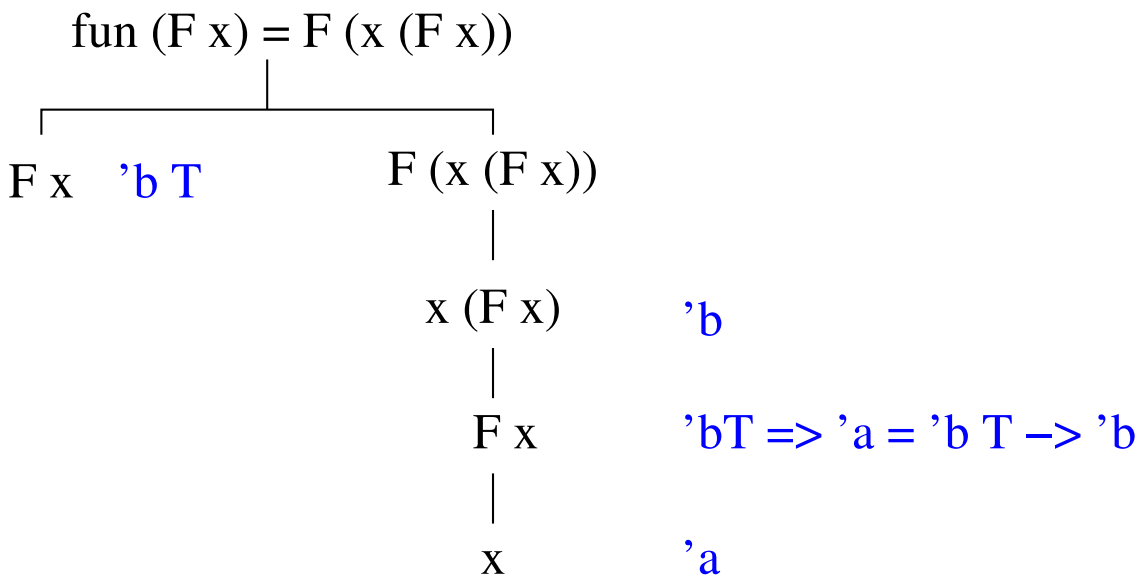
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

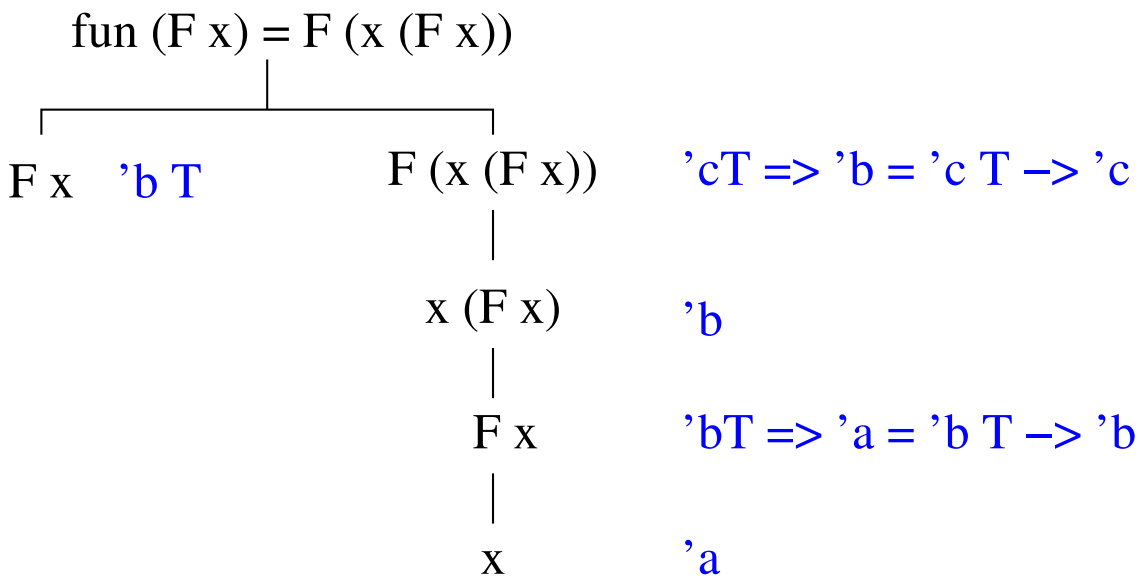
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

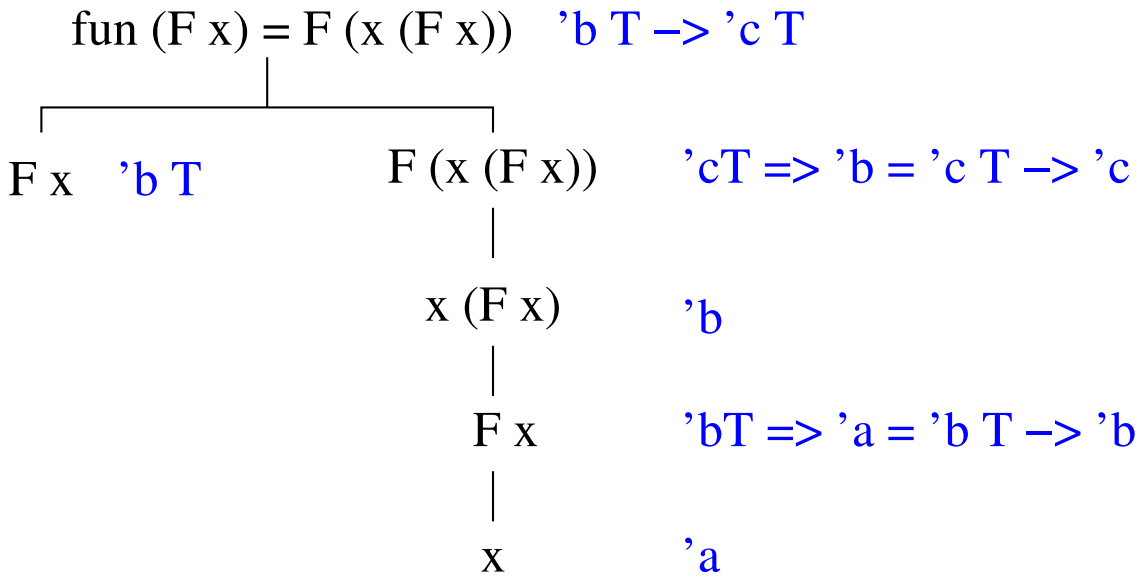
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

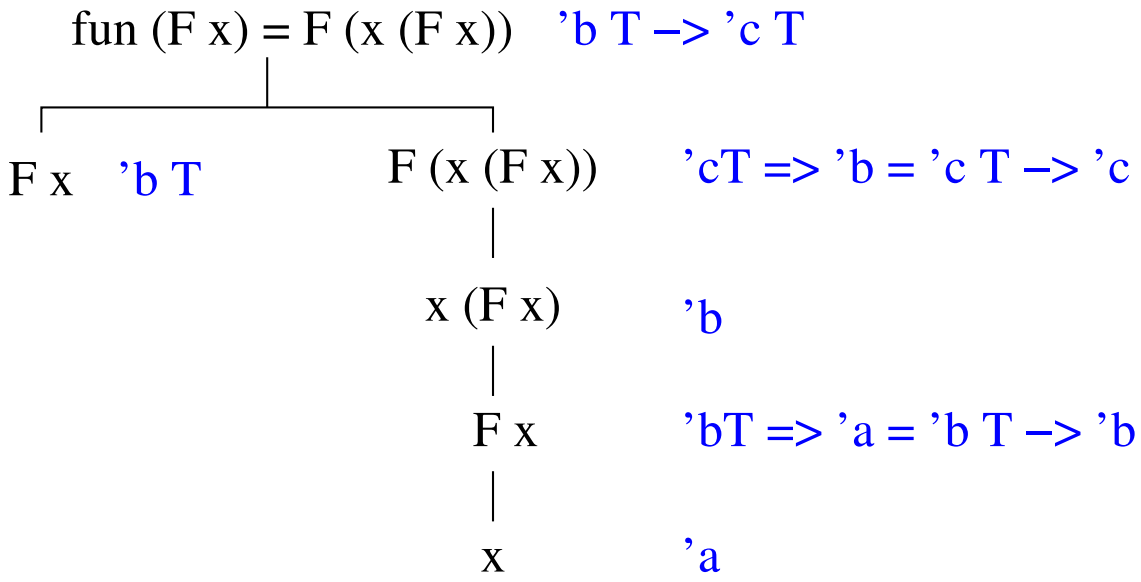
$\Longrightarrow$ `fun f (F x) = F (x (F x)) : 'b T -> 'c T`



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

$\Longrightarrow$ `fun f (F x) = F (x (F x)) : ('c T -> 'c) T -> 'c T`



Typ-Inferenz: Beispiel

```
fun f (F x) = F (x (F x)) : ('cT -> 'c) T -> 'c T
```

Durch Umbennen der Typ-Variable `'c`:

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

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

4.11 Ausnahmen

Ausnahmen 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 Exceptions 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`	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 eingebetten Wertes.

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

$\Longrightarrow$ **exn** ist kein polymoprher 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
```

- `raise`
  - ▷ 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 polymorphisch 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

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

- 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 den 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 ungefangene Ausnahmen.

Ausnahmen Verwenden

Ausnahmen können verwendet werden:

- zur Fehlerbehandlung
- als “lange Sprünge” (*longjumps*)
- als Berechnungsmechanismen

4.11.1 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
```

4.11.2 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 $[a, b, c]$

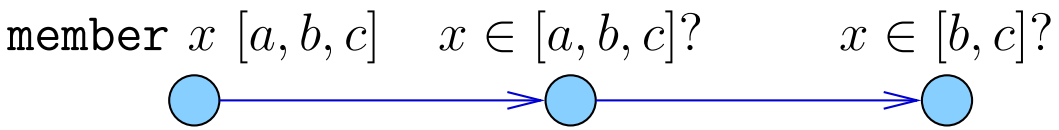
Ausnahmen als Longjumps

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

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

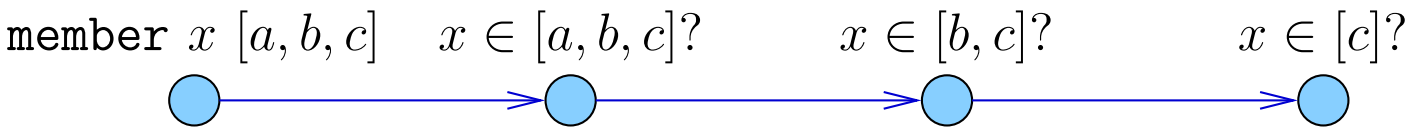
Ausnahmen als Longjumps

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



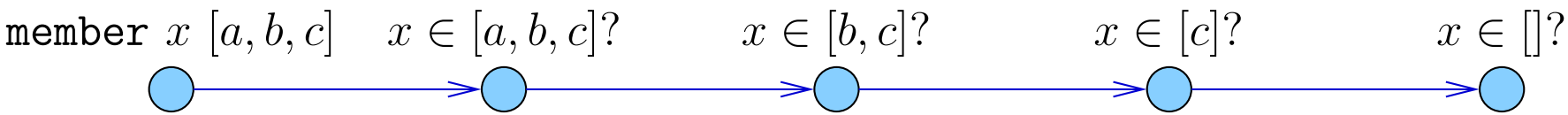
Ausnahmen als Longjumps

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



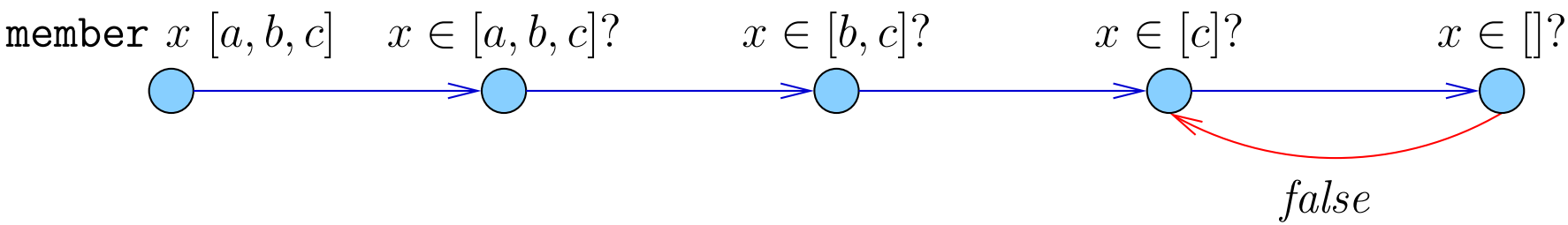
Ausnahmen als Longjumps

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



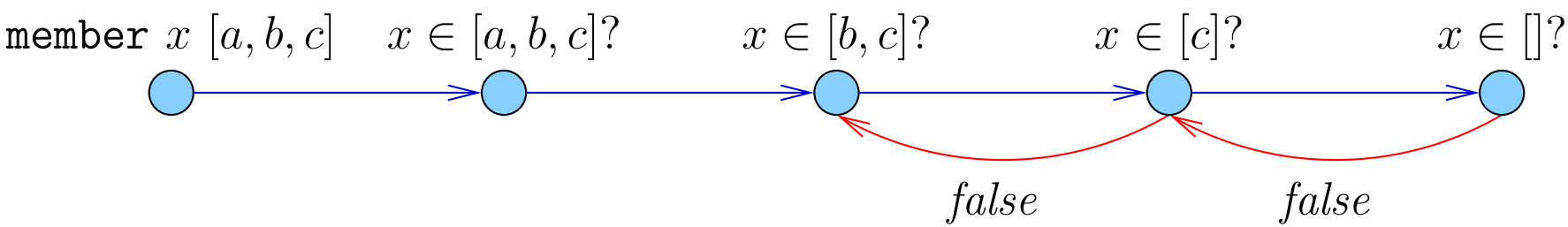
Ausnahmen als Longjumps

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



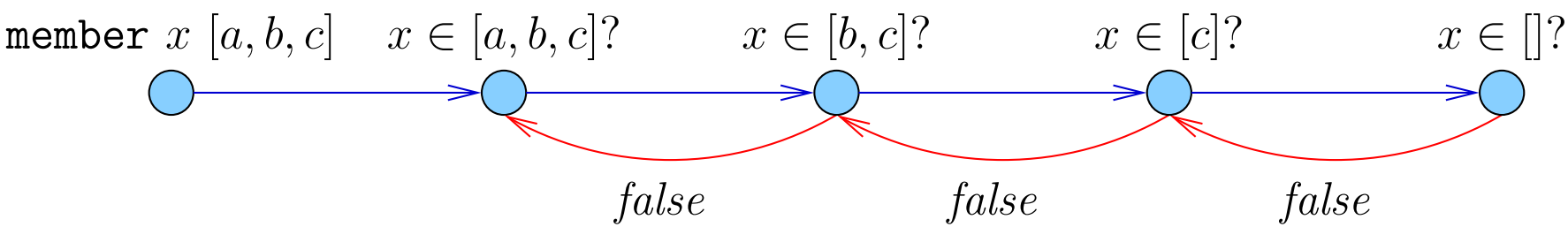
Ausnahmen als Longjumps

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



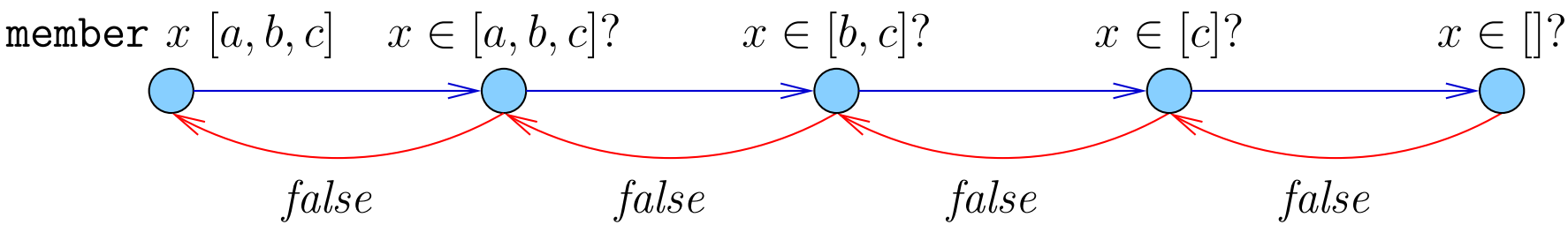
Ausnahmen als Longjumps

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



Ausnahmen als Longjumps

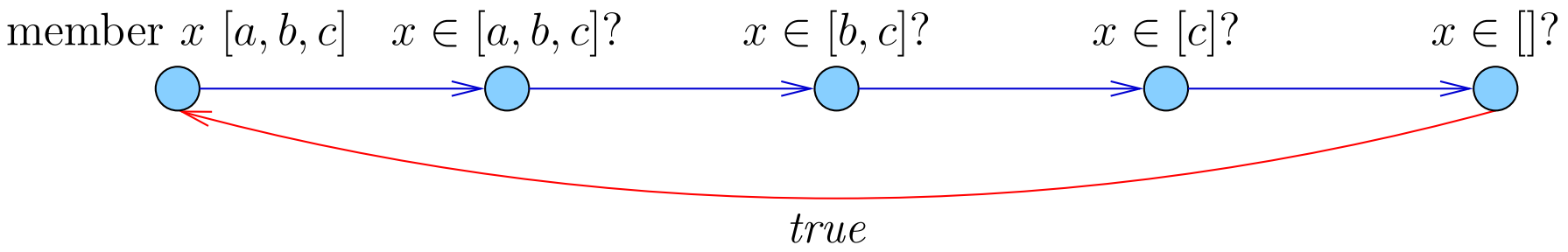
```
fun member x l =  case l of nil => false
                        | h::r => if x=h then true
                                  else member x r
```



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
```



4.11.3 Ausnahmen als Berechnungsmechanismus

```
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
```

4.12 Imperative Features

4.12.1 Referenzen

Der `ref`-Typ-Operator

Der postfixierte einstellige Typ-Operator `ref`

`ref : MT` $\mapsto$ `MT`

konstruiert ein Typ α `ref` aus einem Typ α.

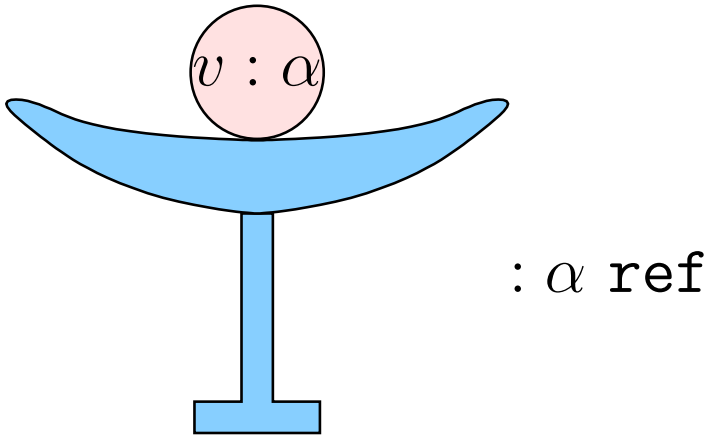
Z.B. `int ref`

`(int * bool) ref`

`int ref ref` (der Typ-Operator `ref` ist links-assoziativ)

Der `ref`-Konstruktor

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




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

Der `ref`-Konstruktor

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

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

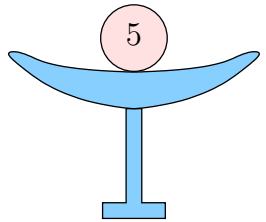
Eine andere Möglichkeit ist der Dereferenzierungs-Operator `!`:

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

Zuweisungen

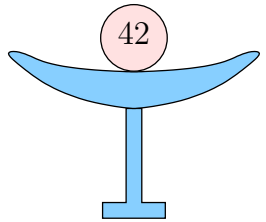
- Der

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
```

Sequenzen

Beim Arbeiten mit Seiteneffekten ist die Ausführungsreihenfolge wichtig.

Typische Benutzungen:

vor Berechnung	nach Berechnung

`let val _ = <effect>`
`    val x = <value>`
`in x`
`end`
	`let val x = <value>`
	`    val _ = <effect>`
	`in x`
	`end`

Abkürzung

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

Beispiel: Objekte mit Hilfe von Referenzen und Abschlüsse

```
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 Hilfe von Referenzen und Abschlüsse

```
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.

4.12.2 Vektoren

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
```

4.12.3 Arrays

Vektoren kann man, wenn sie einmal erzeugt sind, nicht mehr verändern. Dafür muß man Arrays verwenden. Arrays kann man im Gegensatz zu Vektoren nicht direkt hinschreiben:

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

4.12.4 Ein- und Ausgabe

Die einfachste Funktion zur Bildschirmausgabe ist
`print: string -> unit`:

```
print "Palim-palim!\n";
Palim-palim!
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 sich selber entsprechende Funktionen schreiben.

- Erste Möglichkeit: Eine Funktion die einen String produziert benutzen:

```
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: Ausgabe direkt auf dem Bildschirm durchführen:

```
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 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
```

4.12.5 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 Es gibt 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 Es gibt mehrere, nicht verschachtelte rekursive Aufrufe.

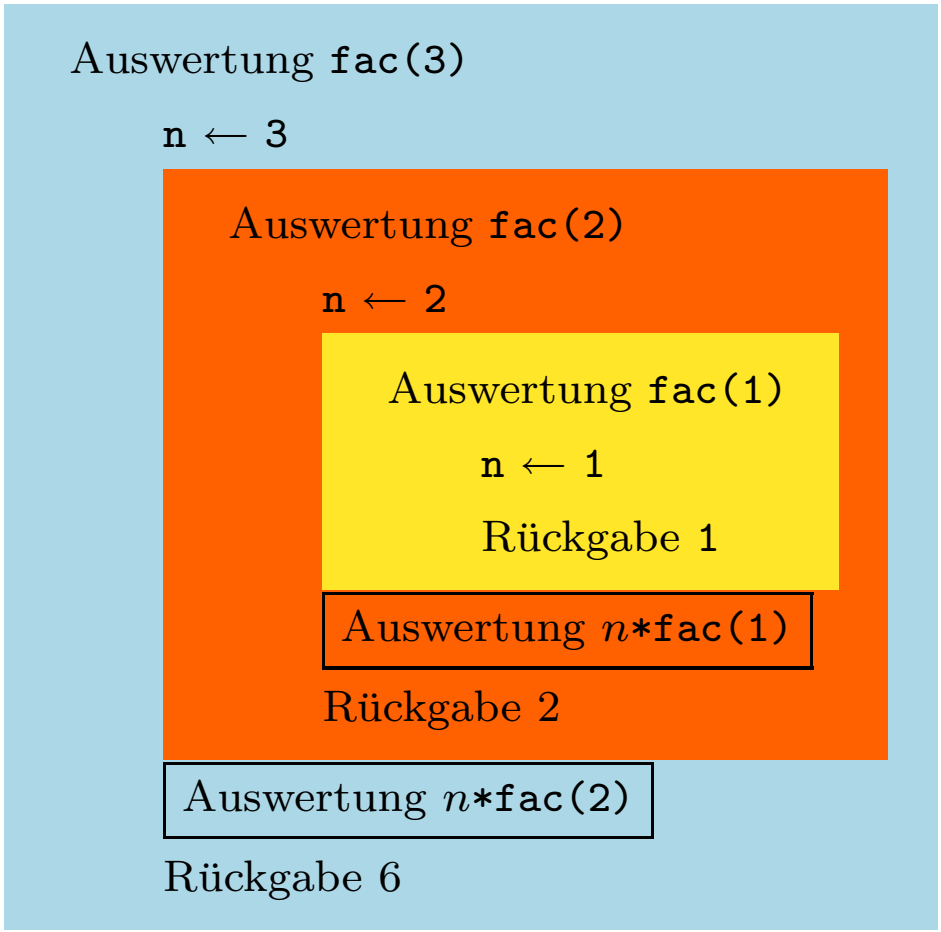
```
fun fib n = if n=0 then 1
            else if n=2 then 1
                 else fib(n-1)+fib(n-2)
```

- Wilde Rekursion: geschachtelte rekursive Aufrufe

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

Speicherverhalten der repetitiv-rekursiven Funktionen

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

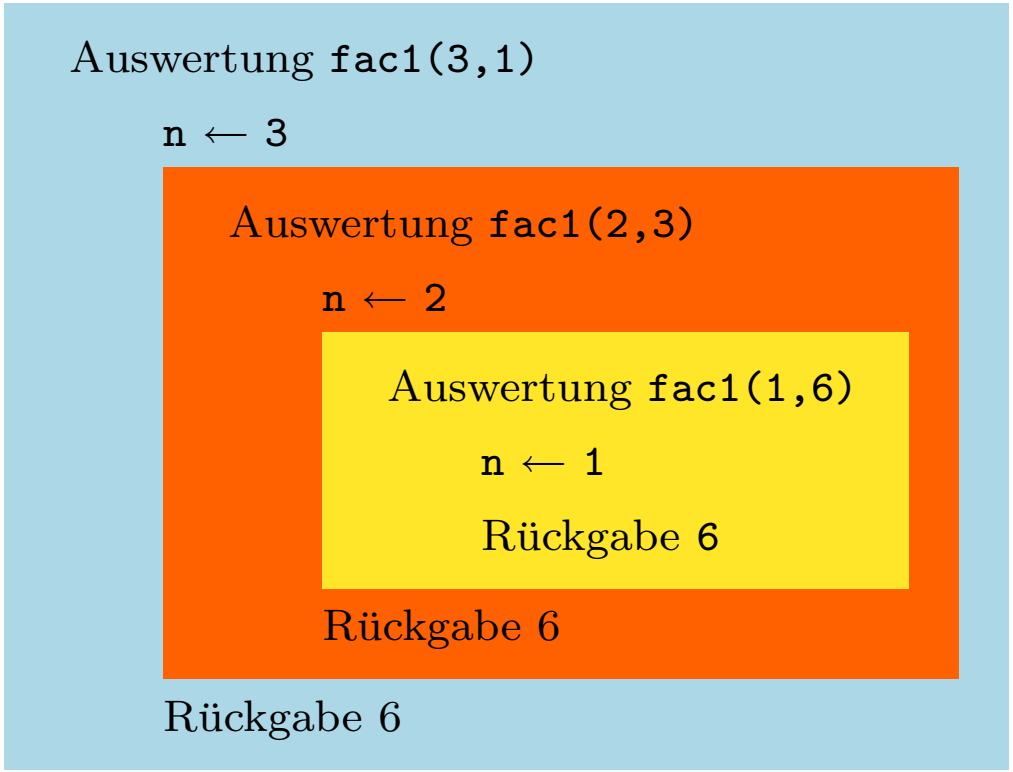


Bei jedem neuen Funktionsauruf muss der aktuelle Aufruf seine lokale Variablen speichern, um diese nach dem Aufruf benutzten zu können

$\Longrightarrow$ 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 denm Aufruf
$\Longrightarrow$ Lokale Variablen werden nicht mehr gebraucht
$\Longrightarrow$ müssen nicht gespeichert werden

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

Eliminierung der repetitiven Rekursion

Der typische Fall von repetitiver Rekursion hat die folgende Form:

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

Zum Beispiel bei Fakultät:

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

Hier ist x_0=1, v_0=1, g(x)=x-1 und e(x,y)=x*y. Wie wir schon wissen, gibt es auch eine end-rekursive Variante:

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

Im allgemeinen Fall kann man f ersetzen durch:

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

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))
```

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

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

$\Longrightarrow$ $fx = f1(x, v_0)$, wenn `e` = assoziativ, kommutativ, z.B.:

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

Der zusätzliche Parameter `a` heißt auch Akkumulator.

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 `::` is weder kommutativ noch assoziativ.

Deshalb berechnen `f x` und `f1 (x,nil)` 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)

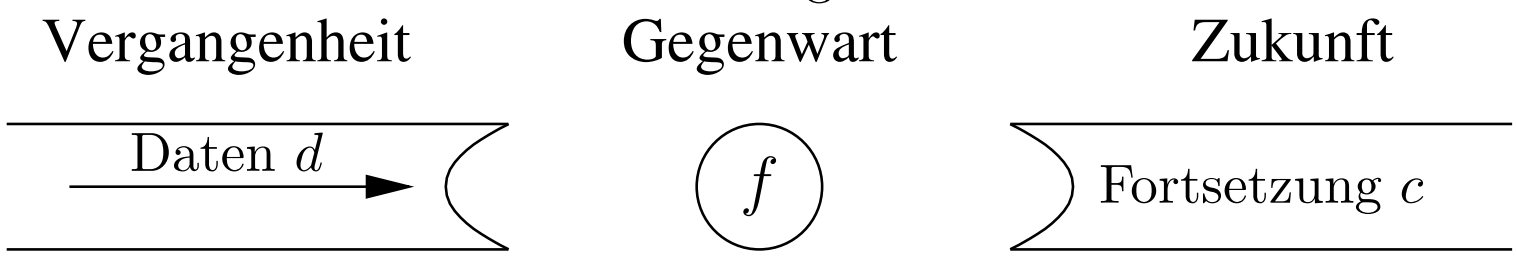
fun fib1 n =
    let fun iter (m,f1,f2) =
            if m = n then f1 else iter(m+1,f2,f1+f2)
    in iter(0,0,1)
    end
```

Das läßt sich aber nicht so einfach verallgemeinern!

Continuation-passing Style

Continuation-passing Style (CPS) (dt. etwa Fortsetzungen-Durchreichen-Programmierstil)

CPS ≡ Abstraktion einer Berechnung:



Vergangenheit = Daten d die der aktuellen Verarbeitungsfunktion übergeben werden

Gegenwart = Eine *aktuelle* Verarbeitungsfunktion f

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

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

Continuation-passing Style

Anstelle der Rückgabe eines Wertes als Ergebnis einer Funktion f, wird eine Funktion k mit diesem Wert aufgerufen, die explizit angibt, wie die Berechnung fortgesetzt werden soll.

f bekommt die Fortsetzungfunktion(en) 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 Continuation-Passing-Style:

```
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

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 der Kontrolflusses
  - ▷ benutzer-definierte Kontrollstrukturen durch explizite Modellierung des Kontrolflusses: z.B. Threads, Korutinen, Ausnahmen

CPS und Effizienz

Im CPS sind die Rückgaben aus Funktionen unnötig.

$\Longrightarrow$ Der Aufruf einer Continuation (Die Aktivierung einer Continuation) braucht nicht die Kontrolle an den Aufrufenden zurückzugeben.

$\Longrightarrow$ Programme in CPS können effizient implementiert werden.

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*)

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

Betrachten wir den Ausdruck:

```
1 + 2 * 3
```

Die impliziten Continuations für jeden Teilausdruck sind unten dargestellt:

Wert	Aktuelle 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 benutzen

In SML of New Jersey (wie auch Scheme, Python, u.a.) ist die aktuelle Continuation ein "first class value":

- Man kann auf die aktuelle Continuation zugreifen und sie als "first class value" manipulieren
- Man kann Continuations aktivieren

Implizite Continuations benutzen

Das Modul `SMLofNJ` 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 Continuations kann mit Hilfe von throw aktiviert werden.

Implizite Continuations benutzen

- `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 benutzen

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

- Wenn `e` die Continuation `k` nicht aktiviert:

$\Longrightarrow$ `callcc (fn k => e)` liefert den Wert von `e`

$\Longrightarrow$`1 +` den Wert von `e` wird zurückgeliefert.

- Wenn bei der Auswertung von `e` die Continuation `k` via `throw k e'` aktiviert:

$\Longrightarrow$ die Berechnung fortgesetzt als hätte `callcc (fn k => e)` den Wert von `e'` zurückgeliefert

$\Longrightarrow$`1 +` den Wert von `e'` zurückgeliefert.

Continuations

```
1 + callcc (fn k => 2);
val it = 3 : int
1 + callcc (fn k => throw k 3);
val it = 4 : int
```

```
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]` berechnet `1*(2*(3*(4*(5*0))))`.

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]` liefert direkt 0.

Continuations-Anwendung: Coroutinen

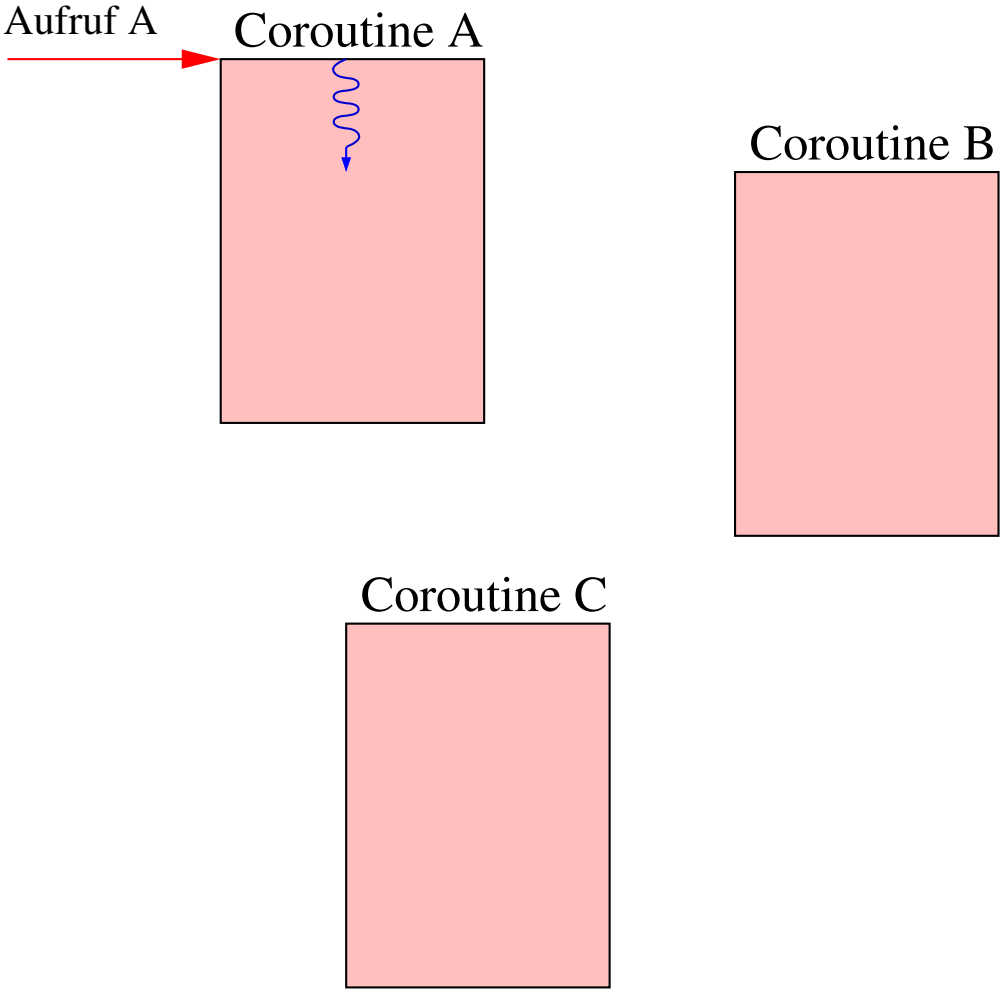
Coroutine = Funktion, die, nach dem sie einen Wert zurückliefern, in dem zuletzt verlassenen Zustand fortgesetzt werden kann.

- Der erste Startpunkt der Coroutinen ist der Anfangspunkt der Coroutine
- Nach einer Rückgabe setzt die Berechnung bei einem neuen Aufruf nach dem Rückgabepunkt fort.

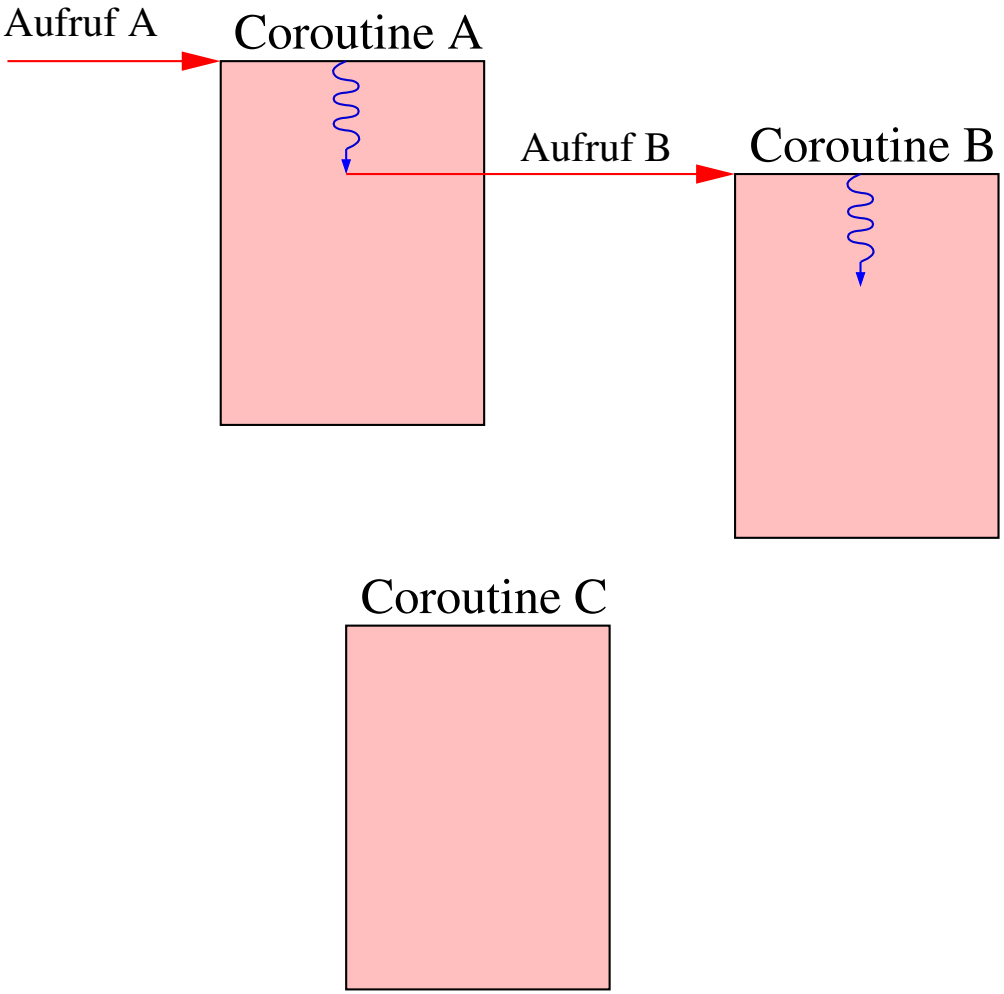
$\Longrightarrow$ Verallgemeinerung von Funktionen:

- mehrere Eingangspunkte
- mehrere Rückgaben aus einer einzigen Funktionsinstanz

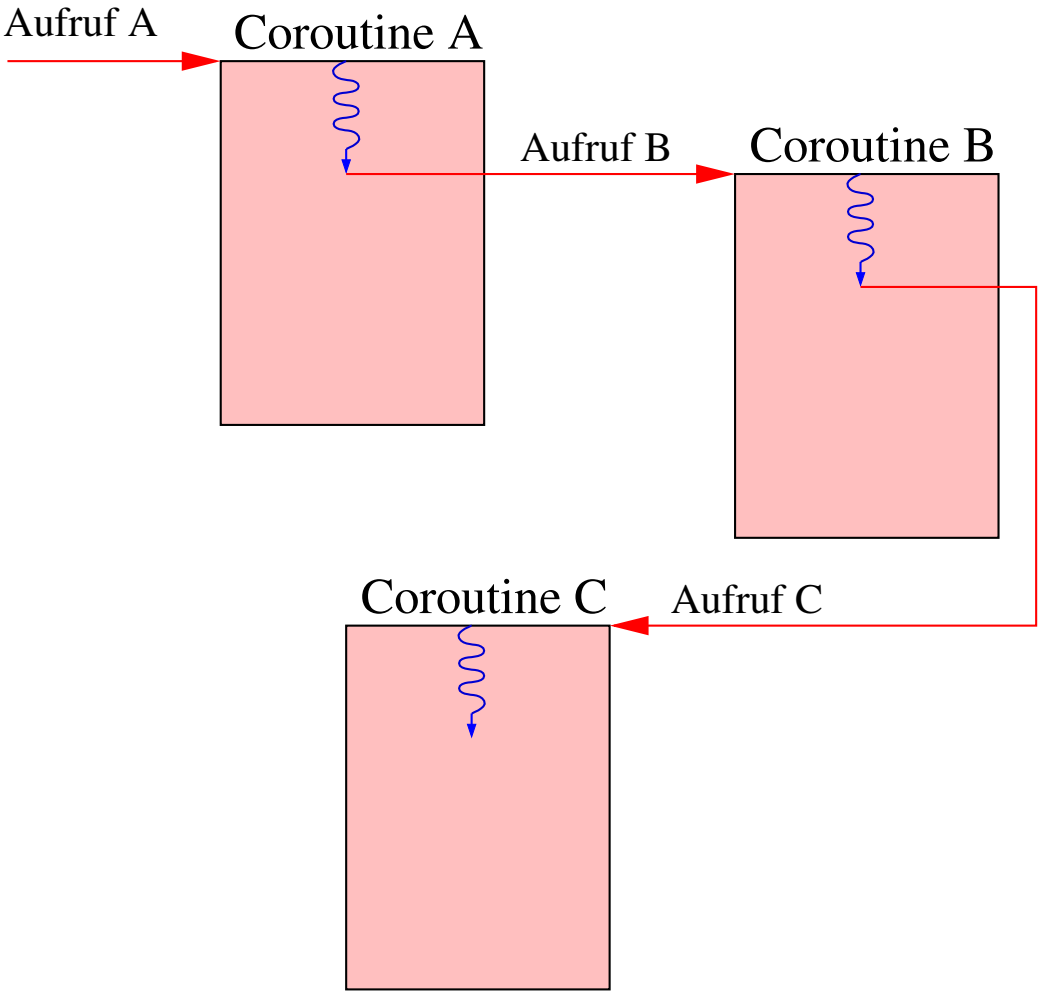
Coroutinen



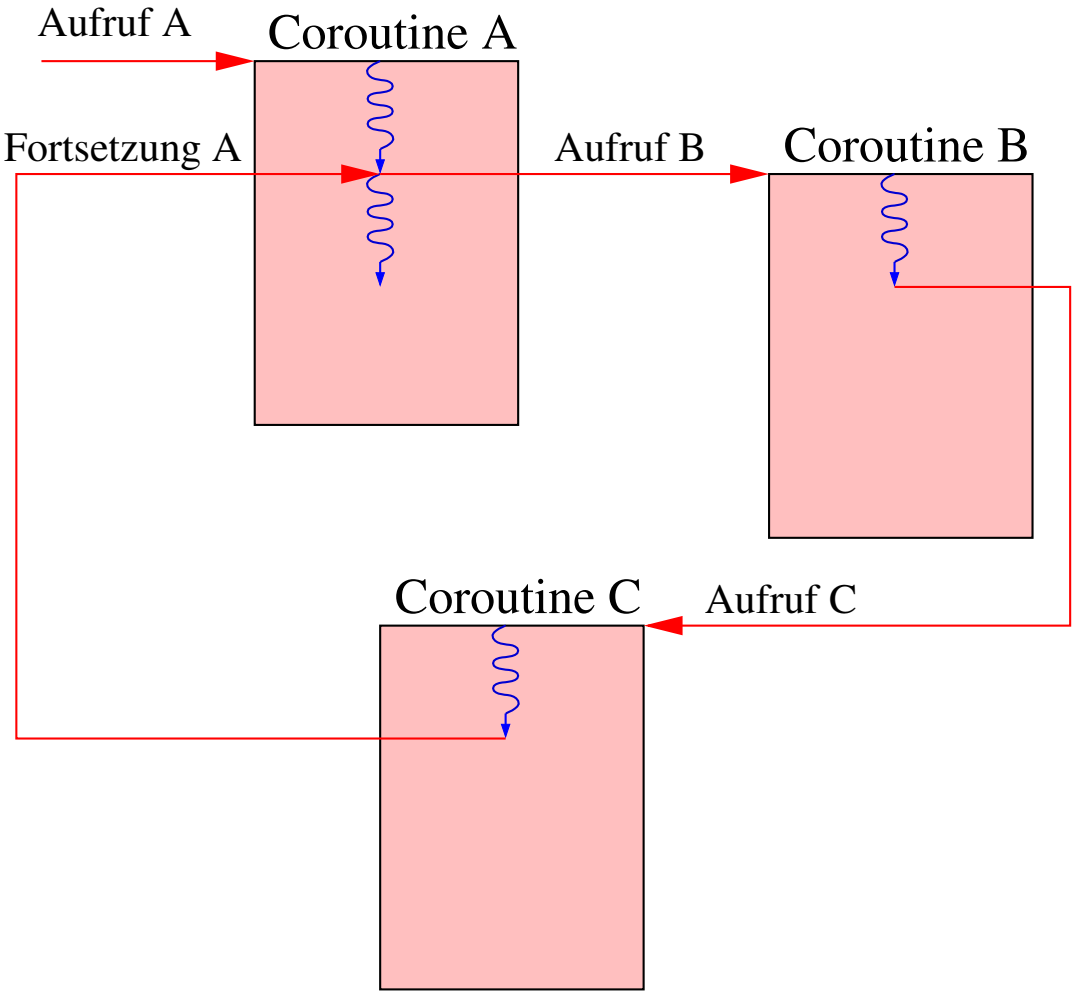
Coroutinen



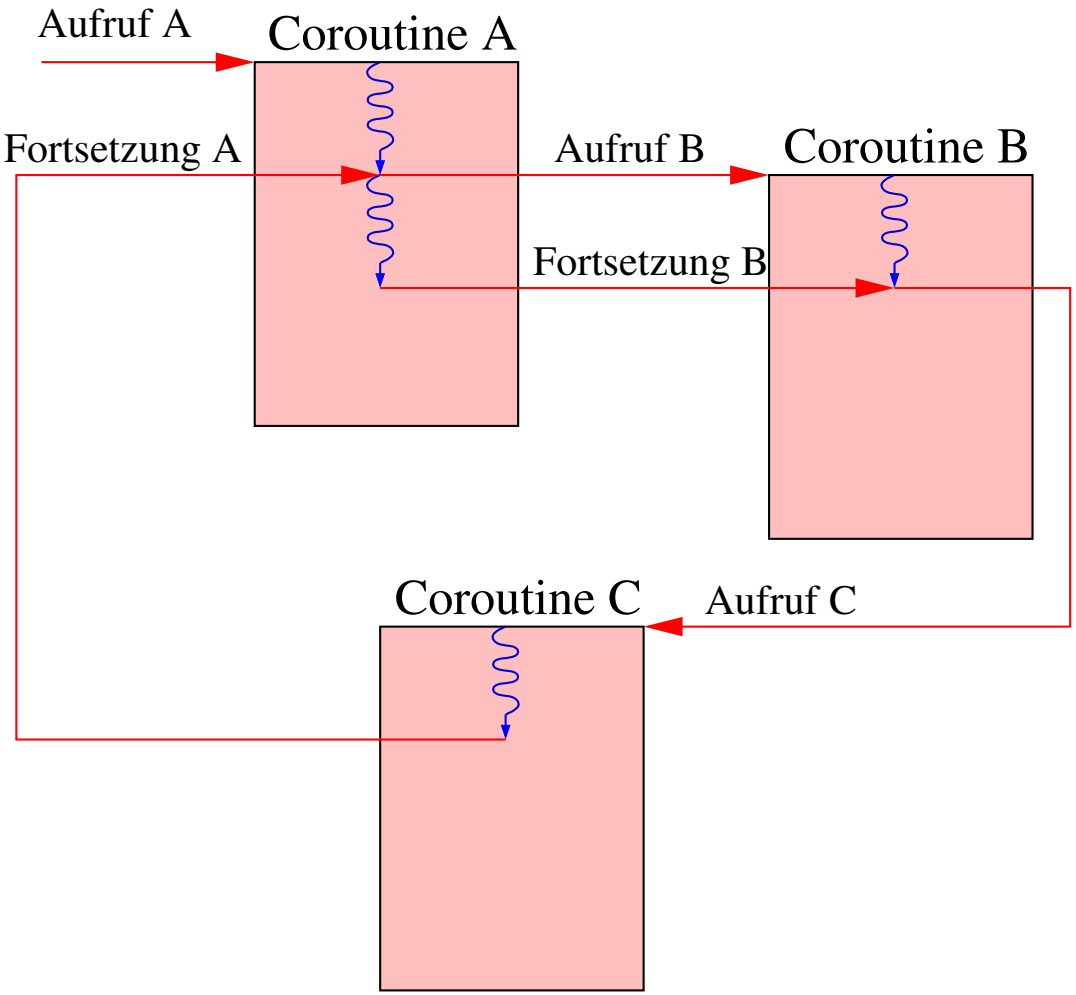
Coroutinen


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

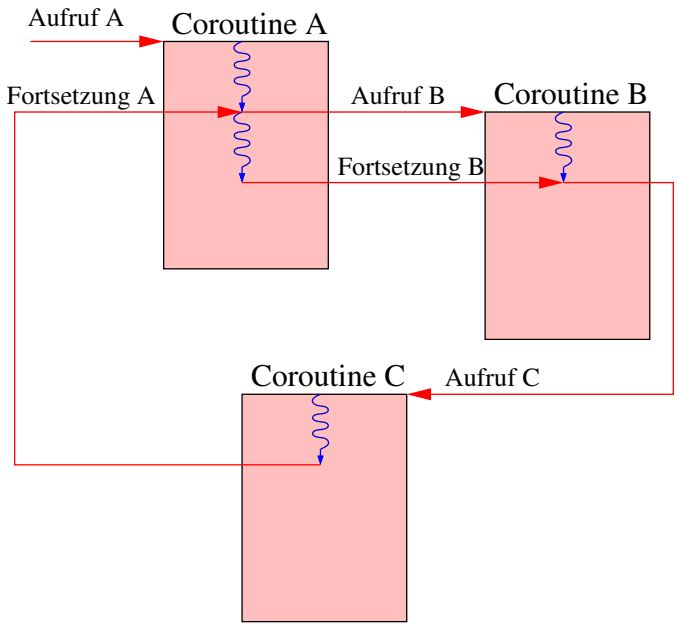
Coroutinen


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

Die Erzeuger-Verbraucher-Architektur mit Coroutinen

Die Ergeuger-Verbraucher-Architektur:

- Der Erzeuger stellt eine Resource zur Verfügung
- Der Verbraucher benutzt die Resource
- Die Kontrolle liegt alternativ beim Erzeuger bzw. Verbraucher

Die 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

Die Erzeuger-Verbraucher-Architektur mit Coroutinen

Implementierung der Kontrollübergabe:

```
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))
```

Die Erzeuger-Verbraucher-Architektur mit Coroutinen

Äquivalent dazu:

```
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))))
```

Die 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.

Die 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

```
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 ())
```

Continuations: Fazit

- Da die aktuelle Continuation "first class" ist, lassen sich grundlegende Konstrukte der Nebenläufigkeit leicht implementieren
- Andere Konstrukte lassen sich ebenso mit Hilfe von "first class" Continuations definieren, z.B.: Ausnahmen, Lösungs-Generatoren, Multi-Agent-Programmierung, Iteratoren, Backtracking, andere benutzer-definierte Kontrollstrukturen ($\longrightarrow$ Übung)

4.13 Das Modul-System von SML

4.13.1 Strukturen

In SML heissen die Module Strukturen:

```
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
```

Strukturen

Auf die Eingabe einer Struktur antwortet der Compiler mit dem Typ der Struktur, einer 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
```

Strukturen

Die Definitionen innerhalb der Struktur sind außerhalb zunächst nicht sichtbar:

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

Über ihren Namen kann man in das Innere einer Struktur hineinsehen:

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

Strukturen

So kann man z.B. 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:

- Für Ausdrücke:

```
let open <struct>
in <expr>
end
```

- Für Definitionen:

```
local open <struct>
in <defs>
end
```

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
```

4.13.2 Signaturen

Mithilfe von Signaturen kann man einschränken, was ein Modul nach außen exportiert:

```
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.

Signaturen

Will man, daß nur über die von der Struktur selbst definierten Funktionen auf ihn zugegriffen werden kann, tut man dies mit einer Signatur:

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

Die Signatur enthält den Counter selbst nicht.

Signaturen

Nun kann man mit dieser Signatur einer Struktur eine eingeschränkte Schnittstelle zuweisen:

```
- structure SafeCount = Count:Count;
structure SafeCount : Count
- 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.

Signaturen

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
```

Signaturen

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 Queue =
  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 : Queue;
structure Queue : Queue

- 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 sogenannte *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
```

4.13.3 Funktoren

Funktoren sind parametrisierte Modulen:

- 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.

Funktoren

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
```

4.13.4 Sharing Constraints

Manchmal ist es notwendig, dass zwei Typen, die in verschiedenen Eingabe-Strukturen eines Funktors definiert sind, identifiziert werden:

```
signature Printable =
  sig
    type Printable
    val print : Printable -> unit
  end
signature PrCounter =
  sig
    type Counter
    val setCounter : Counter -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> Counter
    val prtCounter : unit -> unit
  end
```

Sharing Constraints

- Wir wollen jetzt einen Funktor definieren, der eine Struktur produziert, die nicht nur zählen, sondern auch den Zählerstand ausdrucken kann:

```
functor GenPrCounter (structure Enum : Enum
                      structure Prt  : Printable)
  : PrCounter =
  struct
    structure Cnt = GenCounter (structure Enum = Enum)
    open Cnt
    fun prtCounter() = Prt.print (getCounter())
  end
```

Sharing Constraints

- Das System antwortet mit der folgenden Fehlermeldung:

```
stdIn:845.26-845.50 Error: operator and operand
don't agree [tycon mismatch]
  operator domain: ?.Printable
  operand:         Counter
  in expression:
    Prt.print (getCounter ())
```

- Weil `Enum` und `Prt` zwei verschiedene Strukturen sind, weiß der Compiler nicht, dass die Typen `Enum.Enum` und `Prt.Printable` gleich sein sollen. Das muss man ihm mithilfe von Sharing Constraints explizit mitteilen.

Sharing Constraints

```
functor GenPrCounter (structure Enum : Enum
                                and Prt  : Printable
        sharing type Enum.Enum = Prt.Printable)
  : PrCounter =
  struct
    structure Cnt = GenCounter (structure Enum = Enum)
    open Cnt
    fun prtCounter () = Prt.print (getCounter())
  end
```

Der Funktor darf nun allerdings nur noch auf solche Strukturen angewendet werden, bei denen die Constraints erfüllt sind.

4.14 Programmieren im Großen

Längere Programme geben wir nicht auf der interaktiven Kommando-Zeile ein, sondern legen sie aus Dateien ab. Den Inhalt einer Datei können wir z.B. mithilfe der Funktion `use : string -> unit` einlesen, übersetzen und in der SML-Umgebung sichtbar machen:

```
- 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, wenn nicht gar mehrere 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

- Besteht ein Projekt aus mehr als zwei Dateien, ist es offenbar mühsam, sich die Datei-Namen zu merken und die entsprechenden Folgen von `use`-Aufrufen zu verwalten. Dazu dient die Struktur `CM` , der Compilation Manager, der bei der SML-Implementierung von New Jersey dabei ist.

Bibliotheken

```
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").

Zum Laden von Bibliotheken bietet CM folgende Funktionen an:

```
make : unit -> unit
make' : string -> unit
```

Bibliotheken

- Die Funktion `make` sucht eine Datei "sources.cm" und versucht, aus der dort angegebenen Spezifikation eine Bibliothek herzustellen. `make'` funktioniert genauso, benutzt stattdessen aber den angegebenen String als Datei-Namen.
- 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.

Bibliotheken

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

```
Library
    signature BAR
    structure Foo
is
    bar.sig
    foo.sml
    cml.cm
```

Bibliotheken

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 `utils.sml` und Datenstrukturen aus `data.sml` zusammenfassen...

Bibliotheken

```
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
```

Bibliotheken

Soll die Gruppe nur innerhalb der Bibliothek oder Gruppe `a.cm` verwendet werden, kann man diese explizit mitteilen:

```
Group (a.cm)
    structure Data
is
    utils.sml
    data.sml
```

Bibliotheken

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.

Bibliotheken

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-Version oder Betriebssystem;
- Erzeugung von Stand-alone-Versionen.

5 Logische Programmierung

- 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 Program wird spezifiziert durch
  - ▷ das Wissen über ein Problem und die Annahmen, die hinreichend für die Lösung sind ≡ logische Axiomen;
  - ▷ eine zu beweisende Aussage (Ziel, goal statement) als das zu lösende Problem. (≈ Eingabe)
- Die Ausführung
  - ▷ ist der Versuch das goal statement zu beweisen unter den gegebenen Annahmen.

5.1 Grundlegende Konstrukte

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

  ```
  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:

```
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 Frage ist die logische Folge eines identischen Fakten.

  $\Longrightarrow$ `father(abraham,isaac)?` ist wahr, angenommen der Fakt `father(abraham,isaac).` ist Teil des Programms.

Logische Variablen

Statt nur `wahr` oder `falsch` als Antworten zu bekommen, möchte man manchmal auch Objekte mit einer bestimmten Eigenschaft herausfinden. Z.B.:

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, Variablen benutzen, 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.
  - ▷ Variable: Variable (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 heißt Funktor, t_1 bis t_n Argumente; n ist die Stelligkeit des Funktors
- 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 $X_i \neq X_j$ für alle $i \neq j$ und X_i tritt in keinem t_j auf für alle i und j.
  - ▷ Bsp.: {`X=isaac`}
- Die Anwendung einer Substitution θ auf einen Term A, $A\ \theta$, ist der Term, den man erhält, indem man jedes Auftreten von X durch t ersetzt für jedes Paar $X = t$ in θ.
- A ist eine Instanz von B, wenn eine Substitution θ 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

▷ `father(abraham,X)?`

- Anfragen mit Variablen haben eine existentielle Interpretation:

  Existiert eine Substitution, für welche die Anfrage eine logische Folge des Programms ist?

$\Longrightarrow$ Verallgemeinerung als Deduktionsregel:

Eine existentielle Anfrage $P?$ ist eine Folge einer Instanz $P\ \theta$ für jedes θ.

▷ `father(abraham,X)` ist eine Folge von `father(abraham,isaac).`

Variablen in Anfragen

`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.

$\Longrightarrow$ Instanziierung als Deduktionsregel: Aus einer universell quantifizierten Aussage P folgt eine Instanz $P\ \theta$ für jede Substitution θ.

- ▷ Bsp.: `likes(lot,pomegranates)` folgt aus dem Fakt `likes(X,pomegrantes).`

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.
  - ▷ `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 Wert 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 Speziallfall $n = 0$. Im Allgemeinen ist ein Programm eine endliche Menge von Regeln (statt eine Menge von Fakten).
- Fakten, Anfragen und Regeln heißen auch Horn Klauseln/Klauseln.

Regeln

- Variablen in Regeln sind (wie in Fakten) universell quantifiziert.
  - ▷ `son(X,Y)` $\leftarrow$ `father(Y,X), male(X).`
  - ▷ `daughter(X,Y)` $\leftarrow$ `father(Y,X), female(X).`
  - ▷ `grandfather(X,Y)` $\leftarrow$ `father(X,Z), father(Z,Y).`
- "$\leftarrow$" 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 Instanziierung sind Spezialfälle des Modus ponens.

Prozedurale vs. logische Interpretation der Regeln

▷ `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.*

5.2 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 Basisrelationen 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)` $\leftarrow$
    `father(Parent,Child).`
    `parent(Parent,Child)` $\leftarrow$
    `mother(Parent,Child).`
    ($\longrightarrow$ Mehrere Regeln mit dem selben Kopf definieren eine Disjunktion.)
  - `procreated(Man,Woman)` $\leftarrow$
    `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.
```

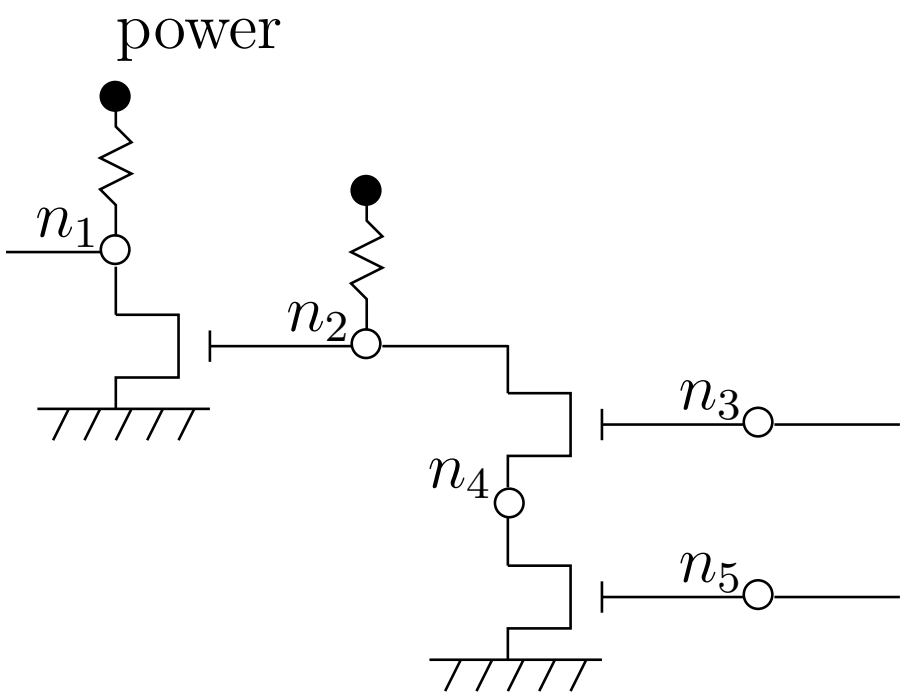
Beispiele

- `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).`

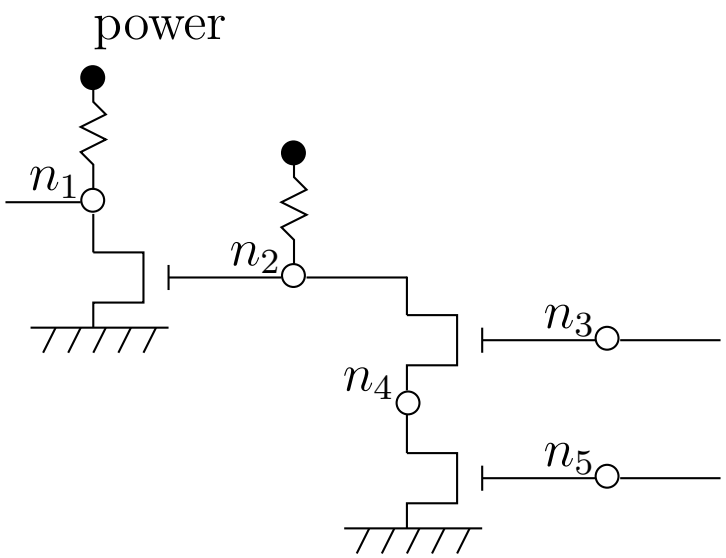
Beispielanfragen

- 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

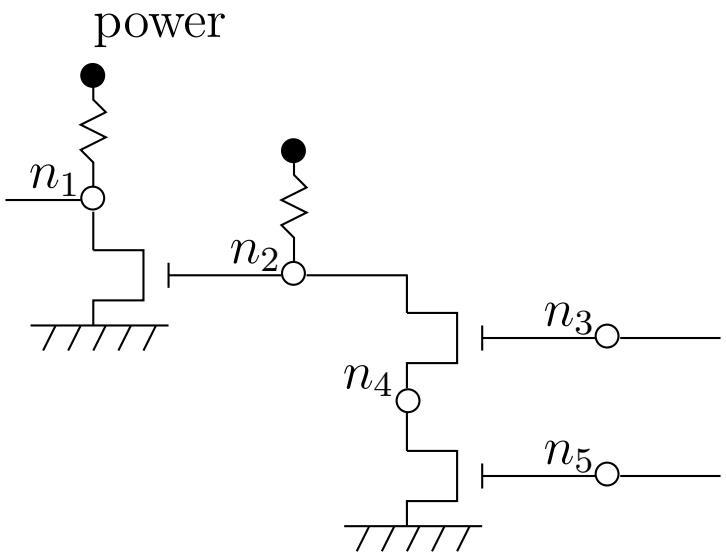



```
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).
```

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)?`

$\Longrightarrow$ `{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,time(Day,S,F),Lecturer,Loc).
```

Rekursive Regeln

```
ancestor(Ancestor,Descendant) ←
      parent(Ancestor,Descendant)

ancestor(Ancestor,Descendant) ←
      parent(Ancestor,Person), ancestor(PersonDescendant)
```

5.3 Logische Programme zur Bearbeitung rekursiver Datenstrukturen

5.3.1 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 das 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.
```

5.3.2 Listen

- Listen sind ein Spezialfall von Binärbäumen und können syntaktisch definiert werden mit zwei Konstrukten:
  - ▷ Leere Liste: `[]`
  - ▷ Nicht-leere Liste: : `[X|Y]`

    mit `X` das erste Element und `Y` der Rest der Liste.
- Syntakische Konvention: `[a|[b|[c|[]]]]` ≡ `[a,b,c]`

Bsp.: member

```
member(X, [X|Xs]).
member(X, [Y|Ys]) ← member(X, Ys).
```

Bsp.: prefix, suffix, sublist

```
prefix([],Ys).
prefix([X|Xs],[X|Ys]) ← prefix(Xs,Ys).

suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) ← suffix(Xs,Ys).

sublist(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)?` gibt aus: `Xs=[a,b,c,d,e]`.
- `append(Xs,[d,e],[a,b,c,d,e])?` gibt aus: `Xs=[a,b]`.
- `append(As,Bs,[a,b,c,d])?` gibt aus 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

- Erster Versuch:

```
reverse([],[]).
reverse([X|Xs],Zs) ← reverse(Xs,Ys), append(Ys,[X],Zs).
```

Ineffizient: quadratische Anzahl von Deduktionsschritten ($\longrightarrow$ Berechnungsmodell der logischen Programme)

- Besser:

```
reverse(Xs,Ys) ← reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys) ← reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

5.4 Das Berechnungsmodel der Logikprogrammierung

5.4.1 Unifikation

- Eine Substitution θ heißt Unifikator für zwei Termen T_1 und T_2 falls $T_1\ \theta = T_2\ \theta$.
  - ▷ Z.B. ist θ = {`X=1, Xs=[2,3], Ys=[3,4], List=[1|Zs]`} ein Unifikator für `append([1,2,3],[3,4],List)` und `append([X|Xs],Ys,[X|Zs])`.
- Gibt es einen Unifikator für T_1 und T_2, dann heißen diese unifizierbar.

Allgemeinster Unifikator

- Ein Unifikator heißt allgemeinster Unifikator (*most general unifier* = *mgu*) für T_1 und T_2, falls es für jeden Unifikator σ eine Substitution β gibt, so dass $\sigma = \theta\ \beta$ (Die Komposition der Substitutionen θ und β).
- Es kann gezeigt werden, dass unter den Unifikatoren von Termen einen allgemeinsten gibt (wenn es überhaupt einen gibt), und dass dieser bis auf die Umbennenung von Variablen (Variantenbildung) eindeutig ist.

Berechnung des allgemeinster Unfikators

- Basiert auf Lösung von Gleichungen zwischen Termen;
- Nutzt einen Keller, um noch nicht gelöste Gleichungen zu speichern und eine Liste von Substitutionen θ, die die Substitutionen für die Ausgabe sammelt.

Eingabe: Zwei Terme T_1 und T_2

Ausgabe: *failure*, wenn T_1 und T_2 nicht unifizierbar sind oder ihr *mgu* sonst.

Algorithm zur Berechnung des allgemeinster Unfikators

```
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 X durch Y im stack und in θ
      füge X = Y zu θ hinzu
    Y ist eine Variable, die in X nicht auftritt:
      ersetze Y durch X im stack und in θ
      füge Y = X zu θ hinzu
    X und Y sind identische Konstante oder Variable: continue
    X ist f(X1, ..., Xn) und Y ist f(Y1, ..., Yn) mit f=Funktor:
      push(stack, X1 = Y1, X2 = Y2, ..., Xn = Yn)
    sonst:
      failure := true
if failure then output failure else output θ
```

mgu-Berechnung: Beispiel

- Zu unifizieren: `append([a|[b]],[c|[d]],Ls)` und
  `append([X|Xs],Ys,[X|Zs])`.
- Initialisierung:
  Stack $s = [$`append([a|[b]],[c|[d]],Ls)` $=$ `append([X|Xs],Ys,[X|Zs])`$]$;
  Substitution $\theta = \{\}$

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]`$\}$

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 `s(X)`;
- wird aus Effizienzgründen in Implementierungen wie Prolog weggelassen.

5.5 Logik als Berechnungsmodel

- 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):

Atome:	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.
- Das Ziel $\top$ heißt top/true/leeres Ziel ($\longrightarrow$ Erfolg der Berechnung). Es gilt das Identitätsgesetz: $G \wedge \top \equiv G$.
- Das Ziel $\bot$ heißt bottom/false ($\longrightarrow$ Scheitern der Berechnung). Es gilt das Absorptiongesetz: $G \wedge \bot \equiv \bot$
- Wir schreiben $\wedge$ für logische Konjunktion.

5.5.1 LP-Kalkül

- Ein Zustand ist ein Paar $< G, \theta >$, wobei G ein Ziel und θ eine Substitution ist. G wird Resolvente genannt.
- Ein Anfangszustand ist ein Zustand der Form $< G, \epsilon >$, wobei ϵ 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.

LP-Kalkül

- 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, Inferrenzregel) von der Form dar:

$$\frac{\begin{array}{l}\text{Bedingung 1}\\ \ldots\\ \text{Bedingung n}\end{array}}{S \mapsto S'}$$

LP-Reduktionsregel

- Entfalten

$$\frac{(B \leftarrow H) \in P \qquad \beta \text{ ist allgemeinster Unifikator von } B \text{ und } A\ \theta}{< A \wedge G, \theta > \mapsto_{Entfalten} < H \wedge G, \theta\ \beta >}$$

- Scheitern

$$\frac{\text{Es gibt keine Klausel } (B \leftarrow H) \in P, \qquad \text{so dass ein Unifikator von } B \text{ und } A\ \theta \text{ existiert}}{< A \wedge G, \theta > \mapsto_{Scheitern} < \bot, \epsilon >}$$

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 θ wird Antwort eines Zieles G genannt, falls es eine erfolgreiche Ableitung $< G, \epsilon > \mapsto \ldots \mapsto < \top, \beta >$ gibt, so dass θ die eingeschränkte Substitution von β 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, oder *no*, sonst

Resolvente := G; $\theta := \epsilon$;

while Resolvente $\neq \top$
  sei Resolvente $= C_1 \wedge \ldots C_i \wedge A \wedge C_{i+1} \ldots C_m$
  if existiert eine (umbennante) Klausel $(A' \mapsto B_1, \ldots, B_n) \in P$,
    so dass β der mgu von A und A' ist
  then Resolvente := $(C_1 \wedge \ldots C_i \wedge B_1 \ldots B_n \wedge C_{i+1} \ldots C_m)\ \beta$
    $\theta := \theta\ \beta$
  else break

if Resolvente $= \top$
  then output θ
  else output *no*

Nichtdeterminismus im LP-Kalkül

- In unserem LP-Kalkül ist die Auswahl der Klausel innerhalb eines Programms und die Auswahl des Atoms A aus der Resolventen nicht deterministisch.
- Eine LP-Sprache (z.B. Prolog) muss den Nichtdeterminismus nach einem Schema (*scheduling policy*) auflösen.
  - ▷ Die Selektion des Atoms 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	Ziel	
`append([],Ys,Ys)` $\leftarrow \top$.	(1)	`append([a,b],[c,d],Ls)`
`append([X|Xs],Ys,[X|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`.	(2)	

$<$ `append([a,b],[c,d],Ls)`, $\epsilon >$

$\mapsto_{Entfalten(2)}$ $<$ `append(Xs,Ys,Zs)`,
`{X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs]}` $>$

$\mapsto_{Entfalten(2)}$ $<$ `append(Xs1,Ys1,Zs1)`,
`{X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs],X1=b,Xs1=[],Ys1=[c,d],Zs=[b|Zs1]}` $>$

$\mapsto_{Entfalten(1)}$ $< \top$,
`{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],Xs1=[c,d]}` $>$

$\Longrightarrow$ Antwort: `{Ls=[a,b,c,d]}`.

5.5.2 Negation durch Scheitern

- Manchmal ist es natürlich negative Bedingungen zu spezifizieren.
  Z.B.:
  `bachelor(X)` $\leftarrow$ `male(X), not married(X).`

$\Longrightarrow$ Syntax und Semantik der LP muss erweitert werden

- Syntax:

Atome:	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 Atome

Reduktionsregeln für negierte Atome

- Scheitern NaF

$$\frac{< A, \theta > \mapsto^* < \top, \beta >}{< \neg A \wedge G, \theta > \mapsto < \bot, \epsilon >}$$

- Erfolg NaF

$$\frac{\text{Jede Ableitung von } A \text{ scheitert endlich: } < A, \theta > \mapsto^* < \bot, \epsilon >}{< \neg A \wedge G, \theta > \mapsto < G, \theta >}$$

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).`$\} \implies$ `p(b)` scheitert. $\implies \neg$`p(b)` ist erfolgreich.

  $P = \{$`p(a).;p(b).`$\} \implies$ `p(b)` ist erfolgreich. $\implies \neg$`p(b)` scheitert.

5.6 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 (Atom oder negiertes Atom) eines Zieles 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)
- Erfolgreiche Ableitungen werden 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 Anwendung der Entfalten-Reduktionsregel:

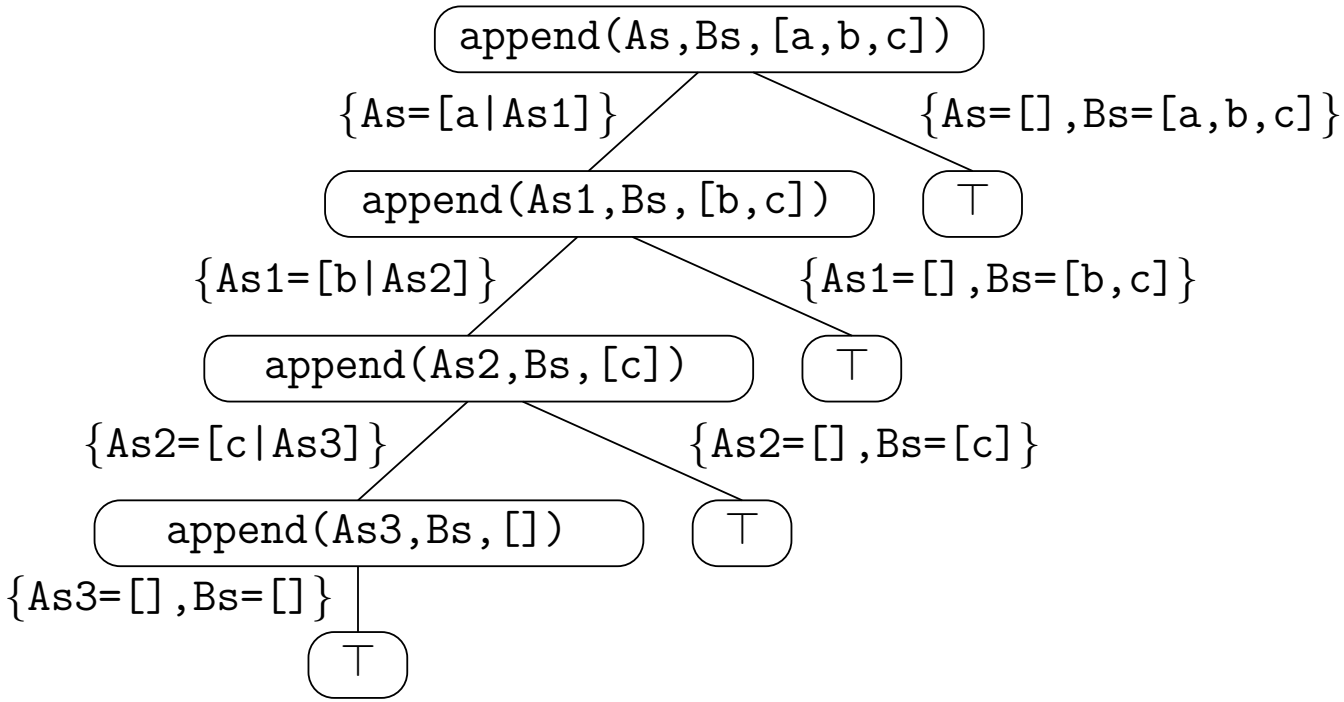
$$\frac{(B \leftarrow H) \in P \qquad \beta \text{ ist allgemeinster Unifikator von } B \text{ und } A\theta}{< A \wedge G, \theta > \mapsto_{Entfalten} < H \wedge G, \theta\beta >}$$

  existiert eine mit β beschrifteter Kante vom Knoten $A \wedge G$ zum Knoten $H \wedge G$.

- Durch die Auswahlstrategie von Prolog entspricht jedem Ziel genau ein Suchbaum.

Suchbäume

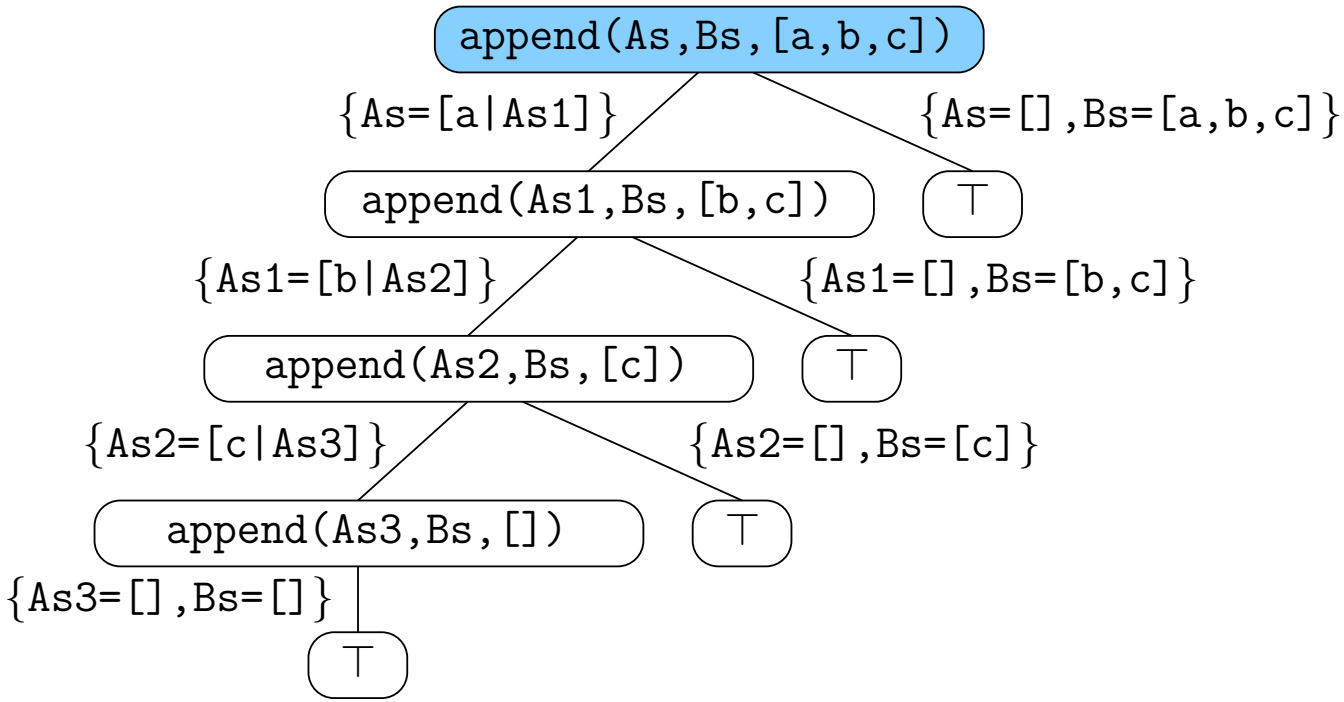
```
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
append([],Ys,Ys).
```



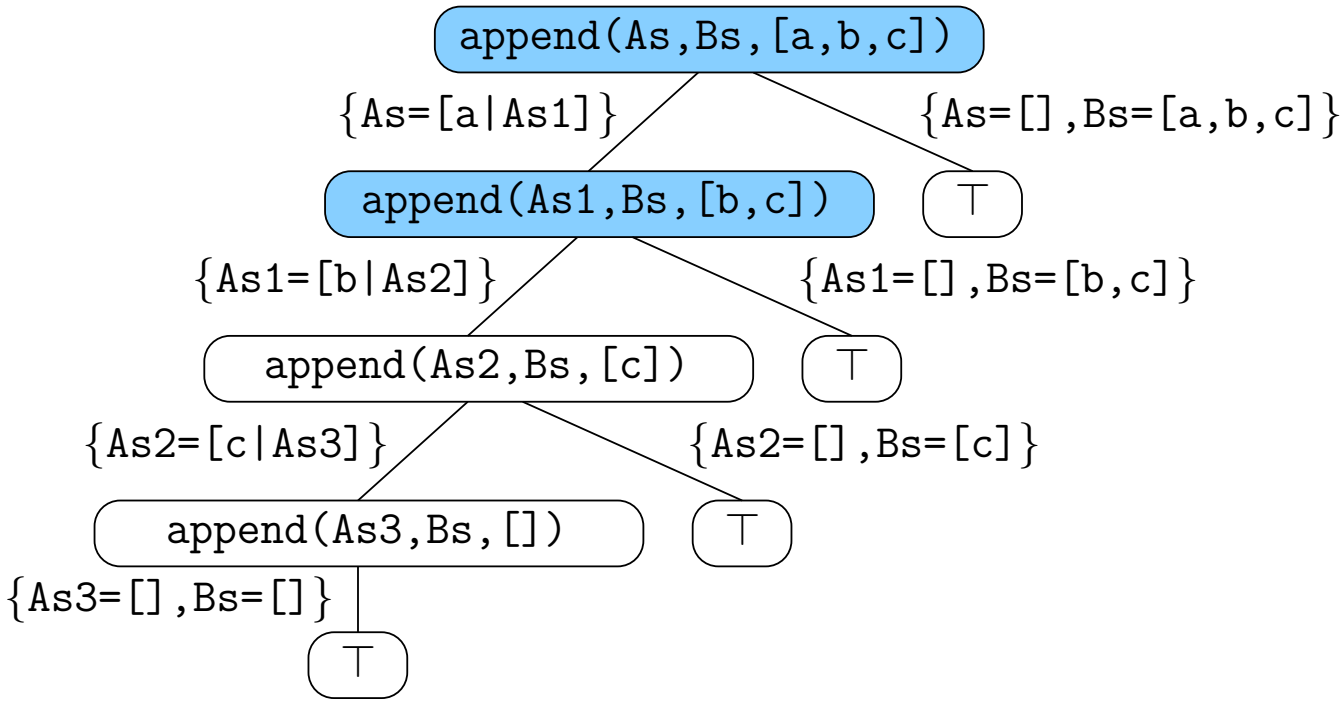
Suchbäume

- 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 erfolgt in Prolog durch einen Tiefendurchlauf über den entsprechenden Suchbaum.

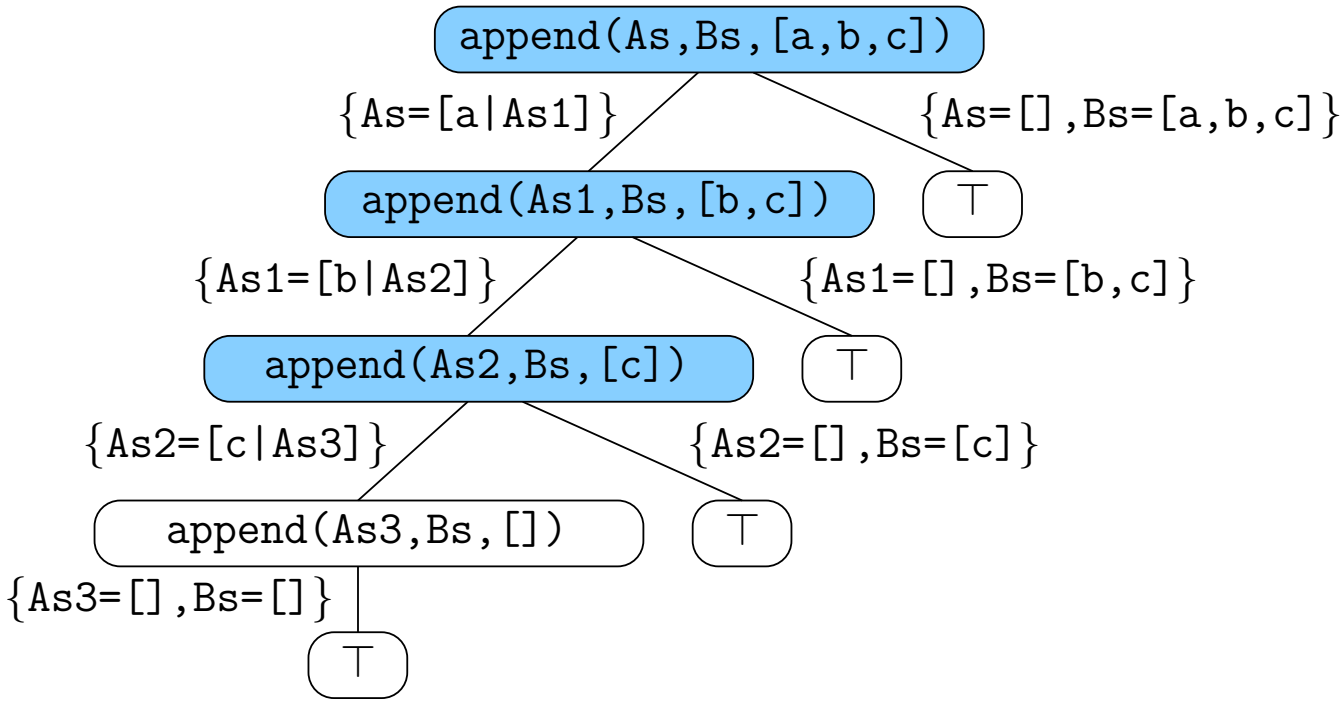
Prolog's Suchstrategie



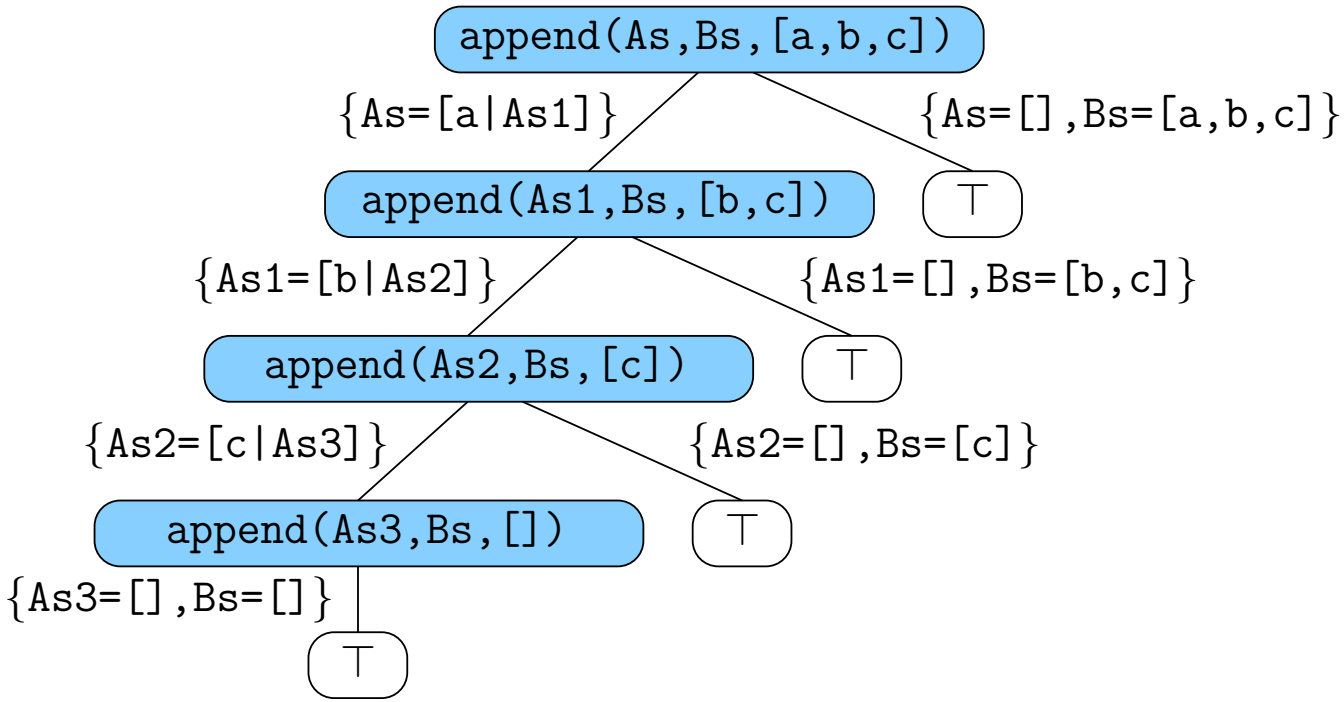
Prolog-Suchbaumdurchläufe



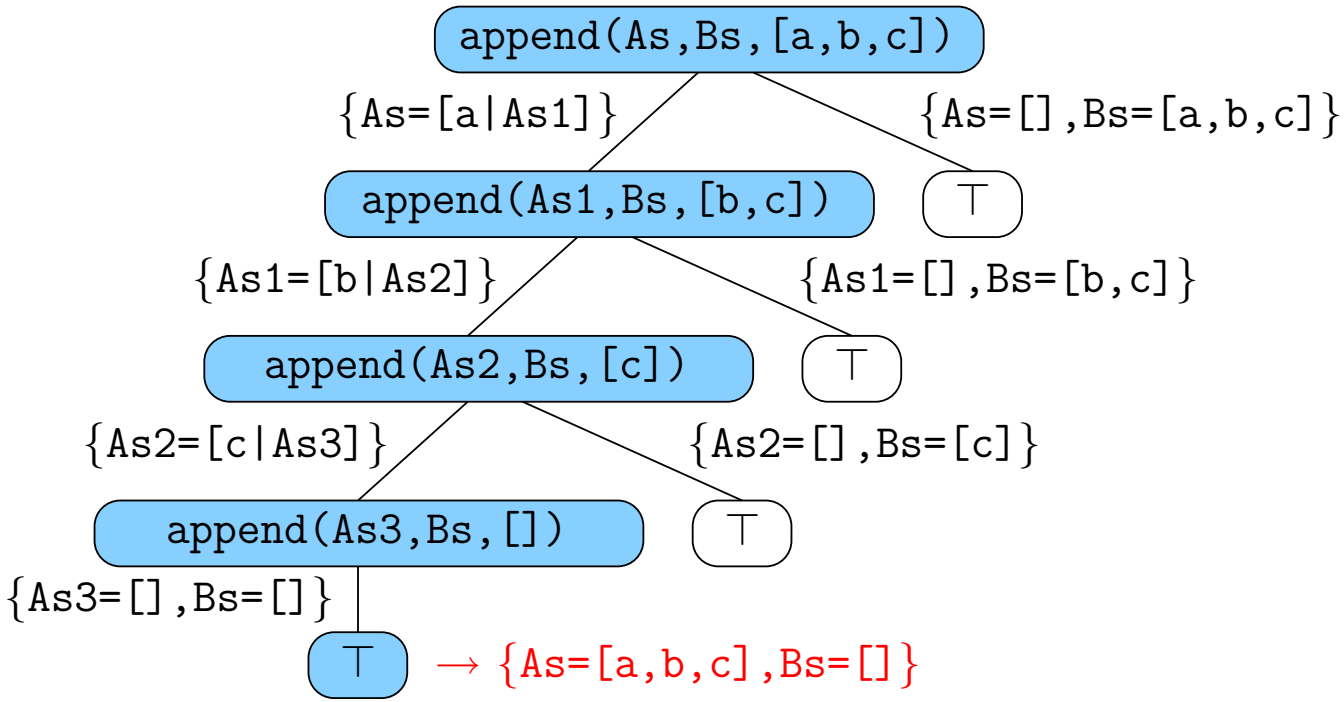
Prolog-Suchbaumdurchläufe



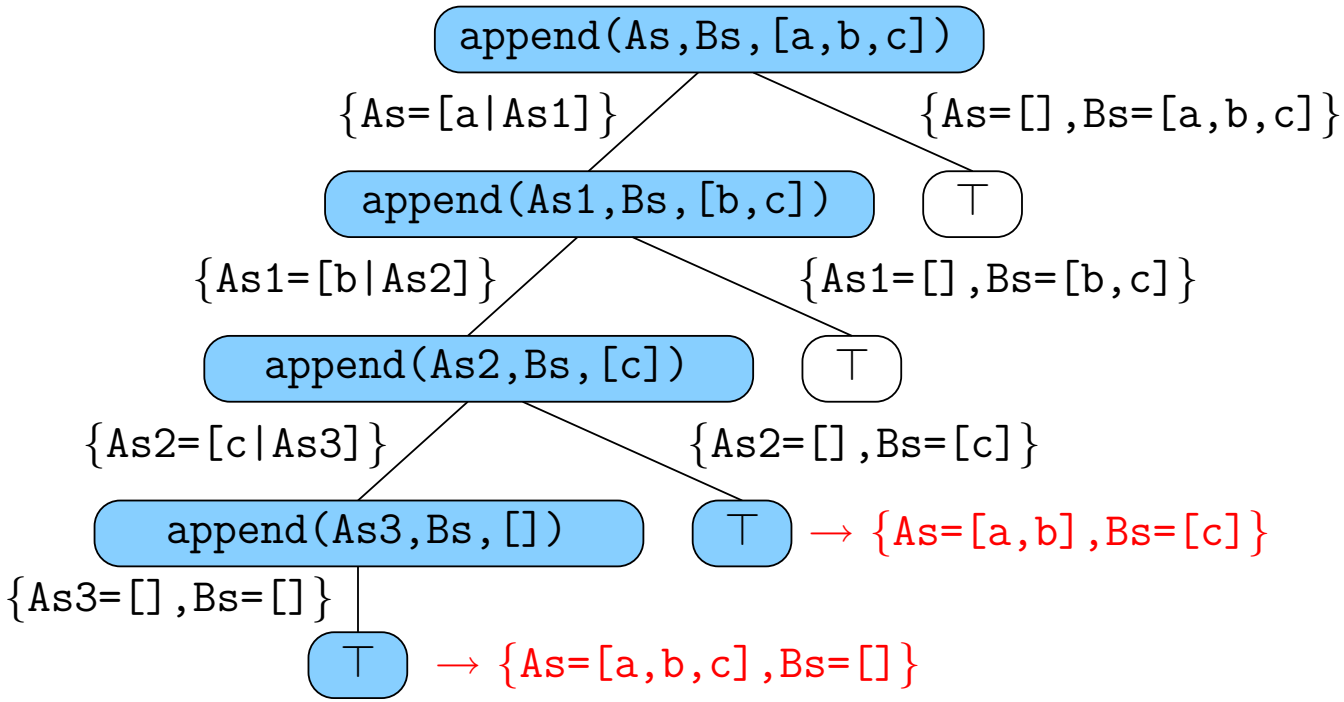
Prolog-Suchbaumdurchläufe



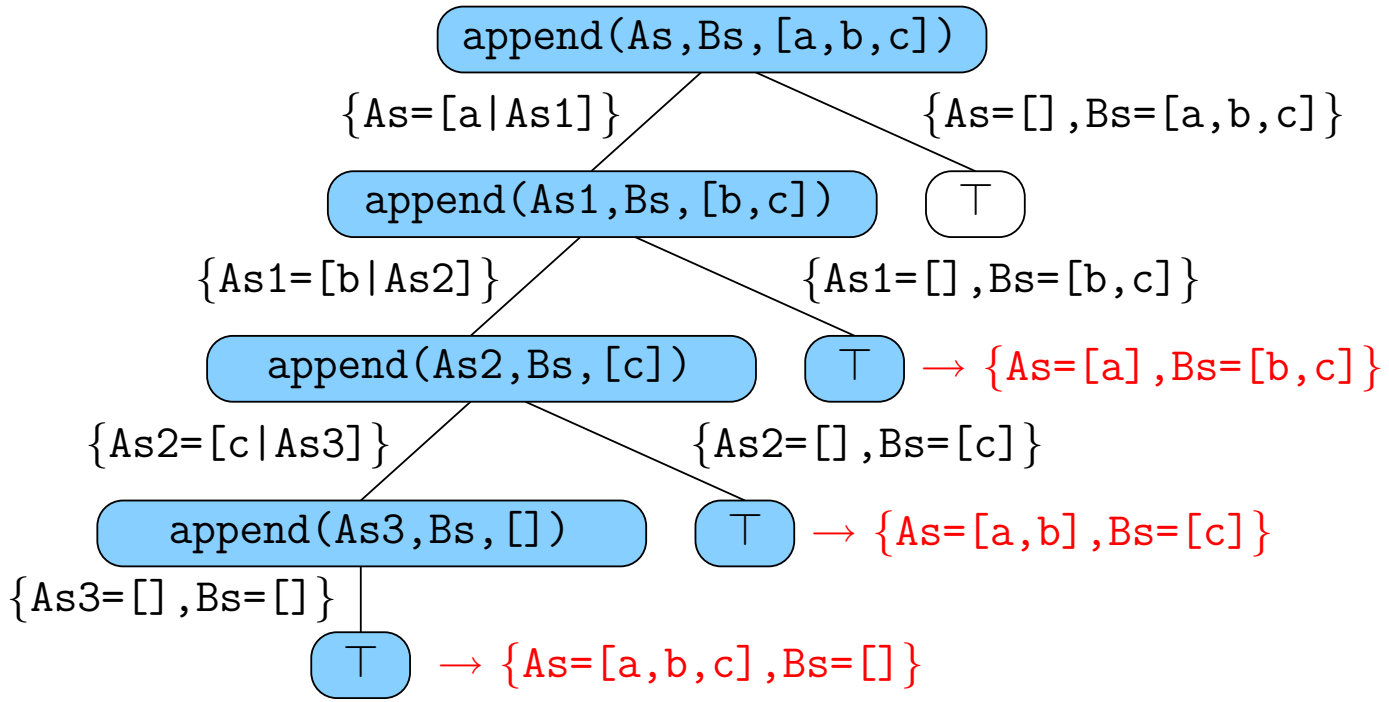
Prolog-Suchbaumdurchläufe



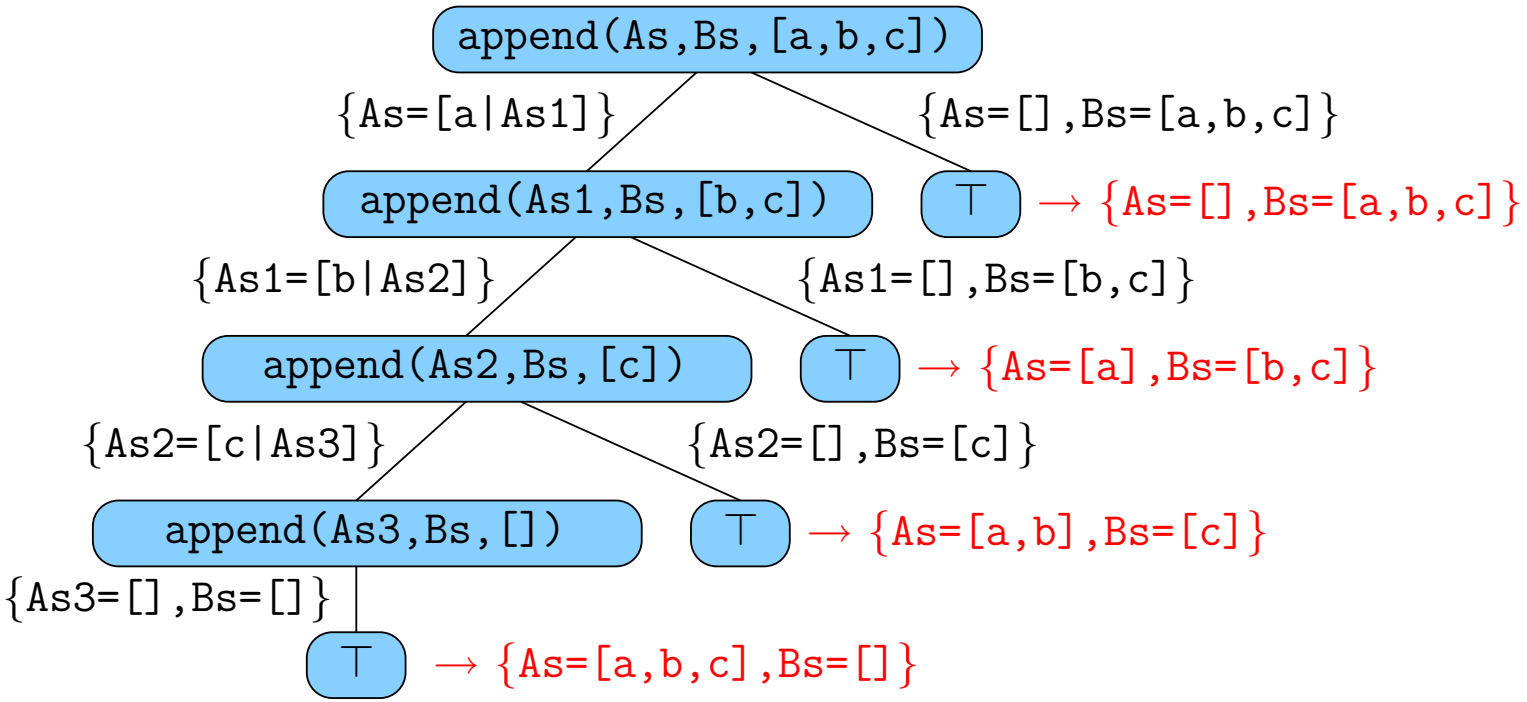
Prolog-Suchbaumdurchläufe



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.

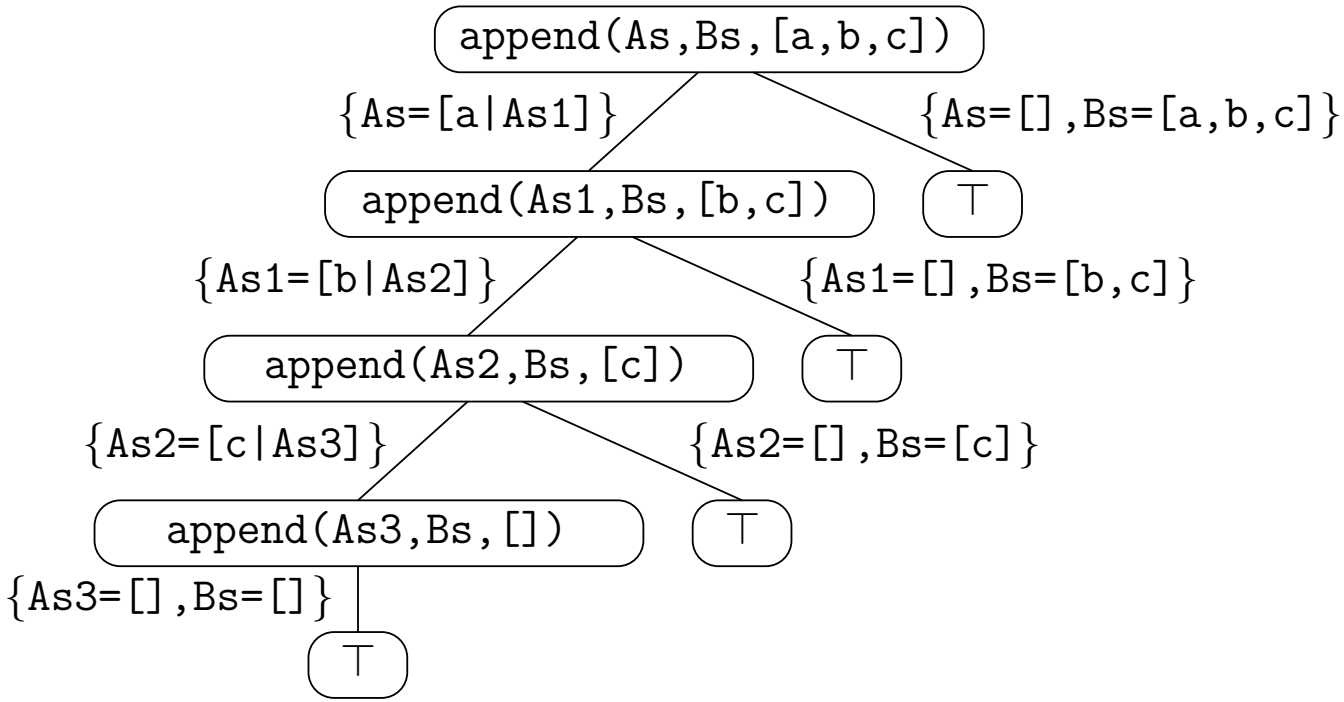
$\Longrightarrow$ Deklarativität der Logikprogrammierung wird (zugunsten der Effizienz) verletzt.

5.6.1 Folgen der Auswahlstrategie in Prolog

- 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.

$\Longrightarrow$ 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.

Beispiel: 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. ($\longrightarrow$ 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.

Rednundante Lösungen

Prolog gibt für jede erfolgreiche Ableitung eine Antwort aus, wenn mehrere Klauseln für den selben Fall zuständig sind, 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
```

5.6.2 Arithmetik in Prolog

- 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 schlägt das Ziel fehl.
- Beispiele:

  `X is 1+2` $\longrightarrow$ Antwort: `X=3`.

  `3 is 1+2` $\longrightarrow$ Antwort: `yes`.

  `1+2 is 1+2` $\longrightarrow$ 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

Effizientere 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).
```

5.6.3 Explizite Kontrolle der Zielauswertung

- Prolog stellt ein Systemprädikat, die das Backtracking bei der Suche von Prolog steuern kann: *cut*, geschrieben !.
- Ziel eines cut ist den Suchaufwand für einer Berechnung zu reduzieren, indem man einen Zweig des Suchbaumes abzuschneidet.
  - ▷ 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 !.

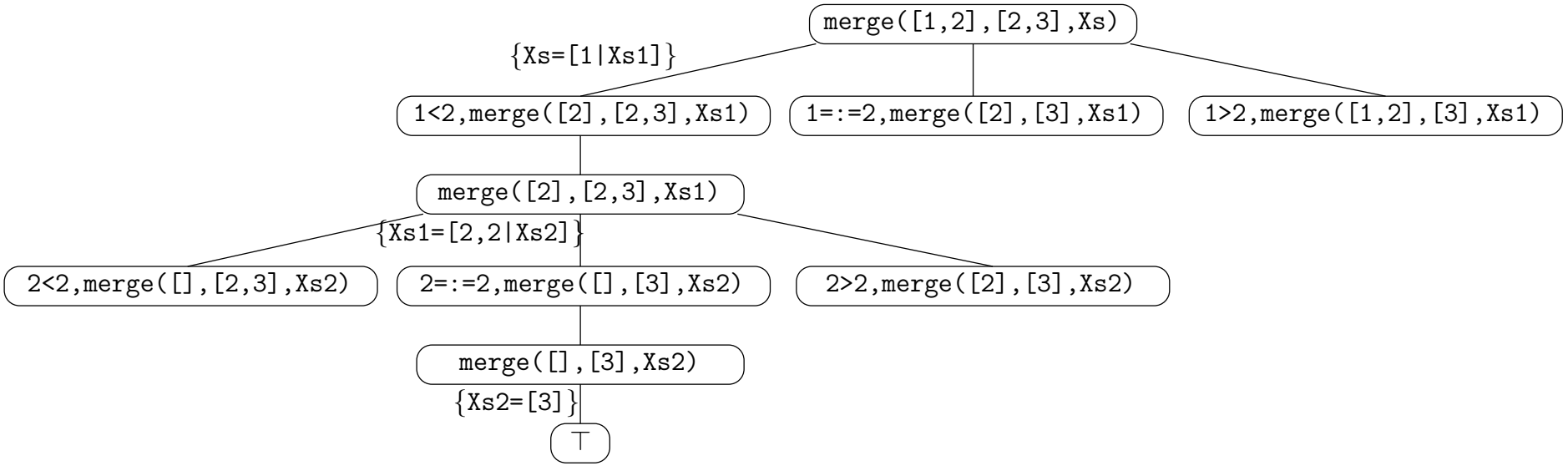
Green Cuts

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

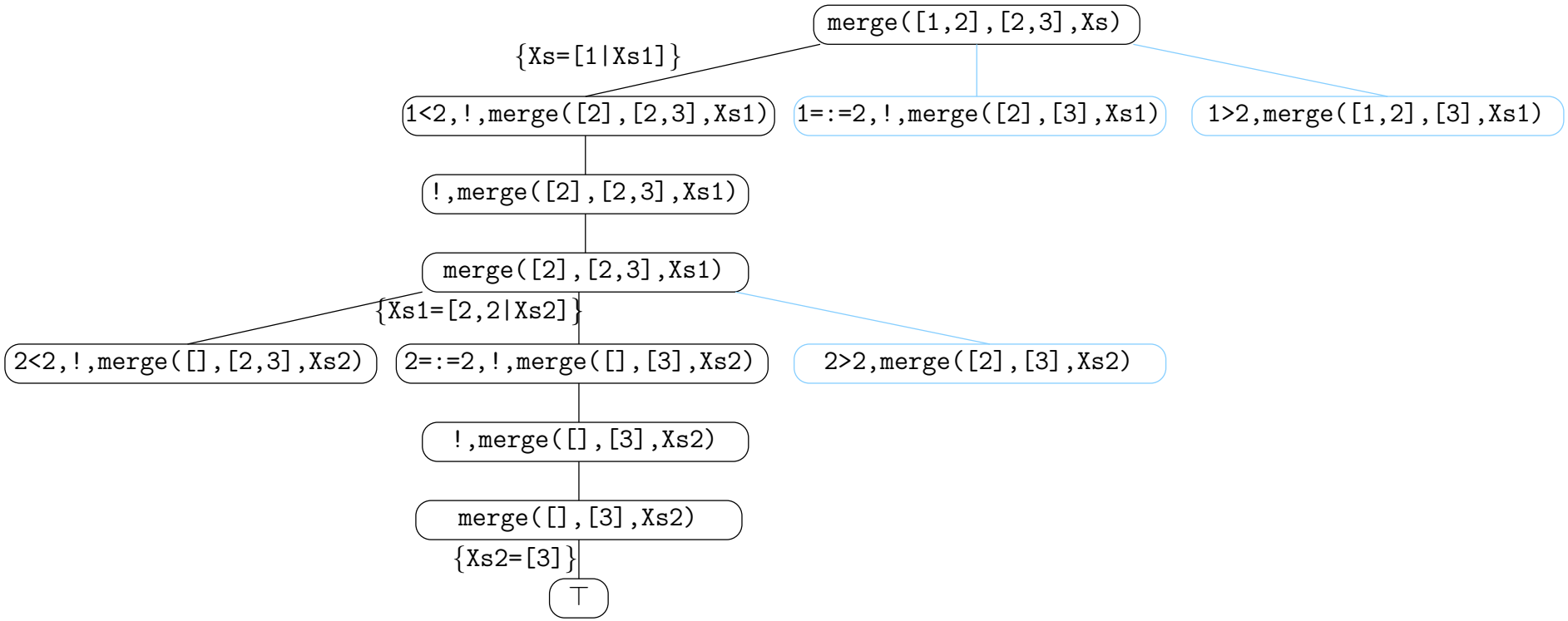

merge([1,2],[2,3],Xs)
{Xs=[1|Xs1]}
1<2,merge([2],[2,3],Xs1)
1=:=2,merge([2],[3],Xs1)
1>2,merge([1,2],[3],Xs1)
merge([2],[2,3],Xs1)
{Xs1=[2,2|Xs2]}
2<2,merge([],[2,3],Xs2)
2=:=2,merge([],[3],Xs2)
2>2,merge([2],[3],Xs2)
merge([],[3],Xs2)
{Xs2=[3]}
⊤

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 Last-call-Optimierung

- 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.
$\Longrightarrow$ Informationen über die jeweils zuletzt gewählten Klauseln (*choice points*) muss in jedem Kellerrahmen gespeichert werden.

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 (rekursive) 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)` ≡ `X` als Ziel.
- ▷ `fail` ist ein Prädikat, das immer scheitert.

Negation in Prolog vs. NaF

```
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

```
braucht_schein(X) :- not(diplom(X)), student(X).
student(piotr).
student(tina).
diplom(tina).
```

Die Anfrage `braucht_schein(X)` scheitert, obwohl die Antwort {`X=piotr`} unter einer Interpretierung von `not` als logische Negation erwartet wird.

$\Longrightarrow$ Prolog erlaubt Negation durch Scheitern nur für zur Laufzeit unter den aktuellen Substitutionen variablenfreie Ziele.

Negation und Ziele mit Variablen

```
braucht_schein(X) :- student(X), not(diplom(X)).
student(piotr).
student(tina).
diplom(tina).
```

$\Longrightarrow$ Antwort {`X=piotr`}.

Der Programmierer muss sicherstellen, dass die Variablen in einem negierten Ziel bei seiner Auswertung belegt sind.

Red Cuts

```
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

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

Beispiel:

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
```

Die Antwort zur Anfrage `clause(member(X,Ys),Rumpf)` liefert die Antworten {`Ys=[X|Xs], Rumpf=true`} und {`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 die letzte Klausel der entsprechenden Prozedur. Z.B. `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. Z.B. kann eine Klausel `a :- b,c` 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: partielle Ergebnisse während einer Berechnung speichern, die später wieder benutzt werden können. ($\longrightarrow$ Übung, die Berechnung der Binomialkoeffizienten).

Idee: versuche ein Ziel abzuleiten, und wenn es möglich ist, speichere dieses Ergebnis als einen Fakt und vermeide, dass es später alternative Ableitungen betrachtet werden.

```
lemma(Z) :- Z, asserta((Z :- !)).
```

Beispiel: Dynamische Programmierung

Anwendung: Die Türme aus Hanoi

```
:- op(100, xfx, [to]).
:- dynamic hanoi/5.

hanoi(1,A,B,C,[A to 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,[A to B|Ms2],Moves).

lemma(P):- P, asserta((P :- !)).

test_hanoi(N,Pegs,Moves) :-
      hanoi(N,A,B,C,Moves), Pegs = [A,B,C].
```

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...

6 Constraint-Programmierung

- Alle in einem rein logischen Programm manipulierten Objekte sind syntaktische Konstruktionen (Terme), denen keine Semantik zugewiesen wird. Man sagt, dass die Funktor-Symbolen nicht-interpretiert (d.h die dargestellten Objekte nicht-interpretierte Strukturen) sind:
  - ▷ Das Ziel `X=2+3` bewirkt nur die Bindung von `X` an den Term `2+3`, weil das Funktionssymbol `+` nicht interpretiert wird.

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`.
- Gleichungen spezifizieren implizit Relationen zwischen semantischen Objekten. Im Allgemeinen heißen Relationen zwischen semantischen Objekten Constraints.

Constraint-Programmierung als Erweiterung der logischen Programmierung

- Die Constraints werden als syntaktisch ausgezeichnete Prädikate dargestellt, und statt mittels SLDNF-Resolution durch spezielle Algortithmen 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 beliebiegen 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 $\longrightarrow$ 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 Variable
  - ▷ Interval-Constraints über endliche Bereiche
  - ▷ Lineare Gleichungssysteme über reele Zahlen

Constraint-Löser

Von einem Constraint-Löser zu erfüllenden Berechnungsdienste:

- Konsistenztest(Erfüllbarkeittest) (*satisfiability/consistency test*): sind die Constraints erfüllbar? Ein Constraint-Löser ist vollständig, wenn er die Erfüllbarkeit jeder beliebiegen 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.

6.1 Das Berechnungsmodell der Constraint-Programmierung

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	$$\frac{(B \leftarrow H) \in P \qquad (B = A) \wedge C \text{ ist erfüllbar}}{< A \wedge G, C > \mapsto_{Entfalten} < H \wedge G, (B = A) \wedge C >}$$
Scheitern	$$\frac{\text{Es gibt keine Klausel } (B \leftarrow H) \in P, \text{ so dass } (B = A) \wedge C \text{ erfüllbar ist}}{< A \wedge G, C > \mapsto_{Scheitern} < \bot, false >}$$
Vereinfachen	$$\frac{C \wedge D_1 \equiv D_2}{< C \wedge G, D_1 > \mapsto_{Vereinfachen} < G, D_2 >}$$

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`.
- Im Allgemeinen kann eine Antwort unbestimmt (*indefinite*) sein, d.h. eine undenliche Menge von Lösungen repräsentieren.
  - ▷ Z.B. liefert `X≤Y,Y≤Z,Z≤X` die Antwort `X=Y=Z`.

6.2 Lineare arithmetische Constraints

Arithmetische Ausdrücke:

$$\text{t} \quad ::= \quad \text{Zahl} \mid \text{Variable} \mid t_1 \odot t_2 \text{ mit } \odot \in \{+, -, *, /\}$$

Linearität:

- Höchstens ein Term einer Multiplikation enthält eine Variable.
- Der Teiler in einer Division enthält keine Variable.

Arithmetische Constraints:

$$\text{C} \quad ::= \quad \text{true} \mid \text{false} \mid C \wedge C \mid t_1 \ \mathcal{R} \ t_2 \text{ mit } \mathcal{R} \in \{<, \leq, =, >, \geq, \neq\}$$

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

$$
\begin{array}{ll}
 & < \texttt{lightmeal(A,M,D);true} > \\
\mapsto & < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} > \\
\mapsto^* & < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} > \\
\mapsto & < \texttt{main(M,J),dessert(D,K);A=radishes,I=1,J>0,K>0,1+J+K<=10} >, \\
\mapsto & < \texttt{meat(M1,I1),dessert(D,K);} \\
 & \qquad\qquad\qquad\qquad \texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} > \\
\mapsto & < \texttt{dessert(D,K);M=beef,J=5,M1=beef,I1=5,A=radishes,I=1,K>0,1+5+K<=10} > \\
\mapsto & < \top\texttt{;D=fruit,K=2,M=beef,J=5,M1=beef,I1=5,A=radishes,I=1} >
\end{array}
$$

Antwort: `A=radishes,M=beef,D=fruit`.

Beispiel: gescheiterte Berechnung

$<$ lightmeal(A,M,D);true $>$

$\mapsto$ $<$ I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true $>$

$\mapsto$ $<$ appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10 $>$

$\mapsto$ $<$ main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10 $>$,

$\mapsto$ $<$ meat(M1,I1),dessert(D,K);
M=M1,J=I1,A=pasta,I=6,J>0,K>0,6+J+K<=10 $>$

$\mapsto$ $<$ dessert(D,K);M=beef,J=5,M1=beef,I1=5,A=pasta,I=6,K>0,6+5+K<=10 $>$

$\mapsto$ $< \perp, false >$

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...

Sicstus-Prolog Syntax

```
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 = 39570.50372439585
```

- Wie hoch ist die Rate, um das Darlehen in 30 Jahren vollständig zu zahlen?

```
 ?- darlehen(200000,360,0.004,X,0.0).
X = 1049.3307086826705
```

- 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}.
S = -836.0650151022006, X = 404.0
```

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=0.005246653543413345*D}
```

- Bei welchem Zinssatz bezahlt man die Schuld in genau 20 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:

$$D_1 = D + D \cdot Z - R$$
$$D_2 = D_1 + D_1 \cdot Z - R$$
$$\vdots$$

6.3 Constraints über endliche Wertebereiche (FD-Constraints)

Constraints:

$$\begin{aligned} \text{C} \quad ::= \quad & \text{true} \mid \text{false} \mid C \wedge C \mid X \text{ in } n..m \\ & \mid X \text{ in } [k_1, \ldots, k_n] \mid X + Y = Z \mid X \; \mathcal{R} \; Y \end{aligned}$$

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 Sicstus-Prolog:

```
| ?- X in 1..5, T in 3..13, X+Y #= T.

X in 1..5,
T in 3..13,
Y in -2..12 ?

yes
```

Constraint Satisfaction Probleme

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 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:

1. Ansatz (Sicstus-Prolog):

```
solve(S,E,N,D,M,O,R,Y) :-
        domain([S,E,N,D,M,O,R,Y], 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 4..7, N in 5..8,
D in 2..8, R in 2..8, Y in 2..8
```

Beispiel:

2. Lösung (Sicstus-Prolog) mit Suche: Zum Enumerieren benutzt man das Prädikat (hier `labeling`), das Variablen alle Werte aus ihren Variablenbereichen nach einer vorgegebenen Strategie zuordnet.

```
solve(S,E,N,D,M,O,R,Y) :-
        domain([S,E,N,D,M,O,R,Y], 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,
        labeling([], [S,E,N,D,M,O,R,Y]).
```

```
| ?- solve(S,E,N,D,M,O,R,Y).
D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2
```