Notizen zur Vorlesung

Deklarative Programmiersprachen

Wintersemester 2010/2011

Prof. Dr. Michael Hanus

Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Christian-Albrechts-Universität zu Kiel

Inhaltsverzeichnis

Kapitel 0

Einleitung

Die Geschichte und Entwicklung von Programmiersprachen kann aufgefasst werden als Bestreben, immer stärker von der konkreten Hardware der Rechner zu abstrahieren. In der folgenden Tabelle sind einige Meilensteine dieser Entwicklung aufgeführt:

Sprachen/Konzepte:	Abstraktion von:
Maschinencode	
Assembler (Befehlsnamen, Marken)	Befehlscode, Adreßwerte
Fortran: arithmetische Ausdrücke	Register, Auswertungsreihenfolge
Algol: Kontrollstrukturen, Rekursion	Befehlszähler, "goto"
Simula/Smalltalk: Klassen, Objekte	Implementierung ("abstrakte Datentypen") Speicherverwaltung
Deklarative Sprachen: Spezifikation von Eigenschaften	Ausführungsreihenfolge ↪ Optimierung ↪ Parallelisierung ↪ keine Seiteneffekte

Eine der wichtigsten charakteristischen Eigenschaften deklarativer Sprachen ist die **referentielle Transparenz** (engl. **referential transparency**):

- Ausdrücke dienen *nur* zur Wertberechnung
- Der Wert eines Ausdrucks hängt nur von der Umgebung ab und nicht vom Zeitpunkt der Auswertung
- Ein Ausdruck kann durch einen anderen Ausdruck gleichen Wertes ersetzt werden (**Substitutionsprinzip**)

Beispiel: Üblich in der Mathematik:

Wert von $x^2 - 2x + 1$ hängt nur vom Wert von x ab
$\Rightarrow$ - Variablen sind Platzhalter für Werte
- eine Variable bezeichnet immer gleichen Wert

Beispiel: C ist nicht referentiell transparent:

- Variablen = Namen von Speicherzellen
- Werte (Inhalte) können verändert werden:

```
x = 3;
y = (++x) * (x--) + x;
```

  Hier bezeichnen das erste und das letzte Vorkommen von `x` *verschiedene* Werte!
  Außerdem ist der Wert von `y` abhängig von der Auswertungsreihenfolge!

Substitutionsprinzip:

- natürliches Konzept aus der Mathematik
  Beispiel: Ersetze π immer durch 3,14159...
- Fundamental für Beweisführung („ersetze Gleiches durch Gleiches“, vereinfache Ausdrücke)
- vereinfacht:
  - Optimierung
  - Transformation
  - Verifikation

  von Programmen

Beispiel: *Tranformation und Optimierung*

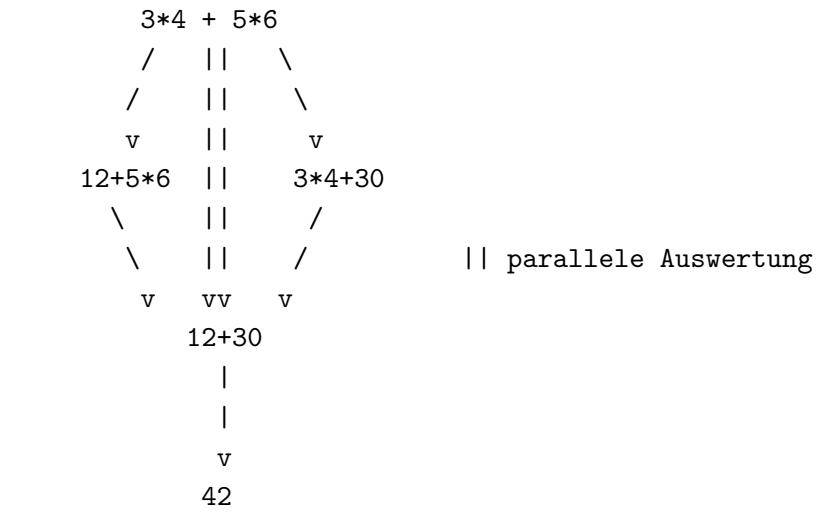
```
function f(n:nat) : nat =
begin
   write("Hallo");
   return (n * n)
end
```

`...z := f(3)*f(3) ...` $\Rightarrow$ `z=81` und `HalloHallo`

Mögliche Optimierung(!?): `...x:=f(3); z:=x*x ...`

$\Rightarrow$ **Seiteneffekte erschweren Optimierung**

Beispiel: Flexible Auswertungsreihenfolge:
Auswertung von `3*4 + 5*6`:



⇒ **Ausnutzung von Parallelrechnern durch referentielle Transparenz**

Wichtiger Aspekt deklarativer Sprachen: **lesbare, zuverlässige Programme**

- Programme müssen so geschrieben werden, damit Menschen sie lesen und modifizieren, und nur nebenbei, damit Maschinen sie ausführen [Abelson/Sussmann 85]
- Programmierung ist eine der schwierigsten mathematischen Tätigkeiten [Dijkstra]

Lesbarkeit durch Abstraktion von Ausführungsdetails:

Beispiel: Fakultätsfunktion:
Imperativ:

```
function fac(n:nat):nat =
begin
   z:=1; p:=1;
   while z<n+1 do begin p:=p*z: z:=z+1 end;
   return(p)
end
```

Potentielle Fehlerquellen:

- Initialisierung (`z:=1` oder `z:=0`?)
- Abbruchbedingung (`z<n+1` oder `z<n` oder `z<=n`?)
- Reihenfolge im Schleifenrumpf (zuerst oder zuletzt `z:=z+1`?)

Deklarative Formulierung: Definiere Eigenschaften von `fac`:

```
fac(0)   = 1
fac(n+1) = (n+1) * fac(n)
```

⇒ kein Zähler (`z`), kein Akkumulator (`p`), Rekursion statt Schleife, weniger Fehlerquellen

Beispiel: **Quicksort**

Imperativ (nach Wirth: Algorithmen und Datenstrukturen):

```
procedure qsort (l, r: index);
var i,j: index; x,w: item
begin
   i := l; j := r;
   x := a[(l+r) div 2]
   repeat
      while a[i]<x do i := i+1;
      while a[j]>x do j := j-1;
      if i <= j then
      begin w := a[i]; a[i] := a[j]; a[j] := w;
            i := i+1; j := j-1;
      end
   until i > j;
   if l < j then qsort(l,j);
   if j > r then qsort(i,r);
end
```

Potentielle Fehlerquellen: Abbruchbedingungen, Reihenfolge

Deklarativ (funktional): Sortieren auf Listen:
(`[]` bezeichnet die leere Liste, `(x:l)` bezeichnet eine Liste mit erstem Element `x` und Restliste `l`)

```
qsort [] = []
qsort (x : l) = qsort (filter (<x) l)
                ++ [x]
                ++ qsort (filter (>=x) l)
```

⇒ klare Codierung der Quicksort-Idee

Datenstrukturen und Speicherverwaltung

Viele Programme benötigen komplexe Datenstrukturen
Imperative Sprachen: Zeiger → fehleranfällig (“core dumped”)

Beispiel: Datenstruktur **Liste**:
- entweder leer (NIL)

- oder zusammengesetzt aus 1. Element (Kopf) und Restliste

Aufgabe: Gegeben zwei Listen $[x_1, \ldots, x_n]$ und $[y_1, \ldots, y_m]$
Erzeuge Verkettung der Listen: $[x_1, \ldots, x_n, y_1, \ldots, y_m]$

Imperativ, z.B. Pascal:

```
TYPE list = ^listrec;
     listrec = RECORD elem: etype; next: list END;

PROCEDURE append(VAR x: list; y:list);
BEGIN
   IF x = NIL
   THEN x := y
   ELSE append(x^.next, y)
END;
```

Anmerkungen:

- Implementierung der Listendefinition durch Zeiger
- Verwaltung von Listen durch explizite Zeigermanipulation
- Verkettung der Listen durch Seiteneffekt

Gefahren bei expliziter Speicherverwaltung:

- Zugriff auf Komponenten von NIL
- Speicherfreigabe von unreferenzierten Objekten:
  - keine Freigabe $\rightsquigarrow$ Speicherüberlauf
  - explizite Freigabe $\rightsquigarrow$ schwierig (Seiteneffekte!)

Deklarative Sprachen abstrahieren von Details der Speicherverwaltung (automatische Speicherbereinigung)
Obiges Beispiel in funktionaler Sprache:

```
data List = [] | etype : List

append([]  , ys) = ys
append(x:xs, ys) = x : append(xs,ys)
```

Vorteile:

- Korrektheit einsichtig durch einfache Fallunterscheidung:
  z.z.: `append`$([x_1, \ldots, x_n], [y_1, \ldots, y_m]) = [x_1, \ldots, x_n, y_1, \ldots, y_n]$
  Durch Induktion über Länge der 1. Liste:
  Ind. anfang: n=0: $\underbrace{\texttt{append}([], \underbrace{[y_1, \ldots, y_m]}_{ys}) = \underbrace{[y_1, \ldots, y_m]}_{ys}}_{\text{Anwenden der ersten Gleichung}}$

Ind.schritt: $n \leq 0$:
Induktionsvoraussetzung: `append`$([x_2, \ldots, x_n], [y_1, \ldots, y_m]) = [x_2, \ldots, x_n, y_1, \ldots, y_m]$
append $([x_1, \ldots, x_n], [y_1, \ldots, y_m])$
= append $(x_1 : \underbrace{[x_2, \ldots, x_n]}_{xs}, \underbrace{[y_1, \ldots, y_m]}_{ys})$
= x_1: append $([x_2, \ldots, x_n], [y_1, \ldots, y_m])$ (Anwendung der 2. Gleichung)
= $x_1 : [x_2, \ldots, x_n, y_1, \ldots, y_m]$ (Ind. vor.)
= $[x_1, \ldots, x_n, y_1, \ldots, y_m]$ (nach Def. von Listen)

- keine Seiteneffekte ($\rightsquigarrow$ einfacher Korrektheitsbeweis)
- keine Zeigermanipulation
- keine explizite Speicherverwaltung
- Universelle Verwendung von `append` durch generische Typen ($\rightarrow$ *Polymorphismus*):

```
data List a = [] | a : List a
```

(`a` Typvariable)

$\Rightarrow$ `append` auf Listen mit beliebigem Elementtyp anwendbar (im Gegensatz zu Pascal)

Klassifikation von Programmiersprachen:

- *Imperative Sprachen:*
  - Variablen = Speicherzellen (veränderbar!)
  - Programm = Folge von Anweisungen (insbesondere: Zuweisung)
  - sequentielle Abarbeitung, Seiteneffekte
- *Deklarative Sprachen*
  - Variablen = (unbekannte) Werte
  - Programm = Menge von Definitionen (+auszuwertender Ausdruck)
  - potentiell parallele Abarbeitung, seiteneffekfrei
  - automatische Speicherverwaltung
  - mathematische Theorie:
    Funktionale Sprachen
    * Funktionen, λ-Kalkül [Church 1941]
    * Reduktion von Ausdrücken
    * Funktionen höherer Ordnung
    * polymorphes Typkonzept
    * "Pattern Matching"
    * "Lazy Evaluation"

Logiksprachen

* Relationen, Prädikatenlogik
* Resolution, Lösen von Ausdrücken [Robinson 1965]
* logische Variablen, partielle Datenstrukturen
* Unifikation
* Nichtdeterminismus

Funktional-logische Sprachen: Kombination der beiden Klassen

Konkrete deklarative Sprachen:

- funktional:
  - LISP, Scheme: eher imperativ-funktional
  - (Standard) ML: funktional mit imperativen Konstrukten
  - Haskell: Standard im Bereich nicht-strikter funktionaler Sprachen
    → praktische Übungen
- logisch:
  - Prolog: ISO-Standard mit imperativen Konstrukten
  - CLP: Prolog mit Constraints, d.h. fest eingebauten Datentypen und Constraint Solver
- funktional-logisch:
  - Mercury (University of Melbourne): Prolog-Syntax, eingeschränkt
  - Oz (DFKI Saarbrücken): Logik-Syntax, nebenläufig
  - Curry: Haskell-Syntax, nebenläufig

Anwendungen deklarativer Sprachen

- prinzipiell überall, da berechnungsuniversell
- Aspekte der Softwareverbesserung:
  - Codereduktion (→ Produktivität):
    * Assembler/Fortran ≈ 10/1
    * keine wesentliche Reduktion bei anderen imperativen Sprachen
    * funktionale Sprachen/imperative Sprachen ≈ $1/(5-10)$
  - Wartbarkeit: lesbarere Programme
  - Wiederverwendbarkeit:
    * polymorphe Typen (vgl. `append`)
    * Funktionen höherer Ordnung $\rightsquigarrow$ Programmschemata

Anwendungsbereiche

- Telekommunikation:
  Erlang (nebenläufige funktionale Sprache):
  - ursprünglich: Programmierung von TK-Software für Vermittlungsstellen
  - Codereduktion gegenüber imperativen Sprachen: 10-20%
- wissensbasierte Systeme: Logiksprachen
- Planungsprobleme (Operations Research, Optimierung, Scheduling):
  CLP-Sprachen
- Transformationen: funktionale Sprachen, z.B. XSLT für XML-Transformationen

Literatur

- P. Hudak: The Haskell School of Expression: Learning Functional Programming through Multimedia, Cambridge University Press, 1999
- S. Thompson: Haskell - The Craft of Functional Programming, Addison-Wesley, 1996
- L. Sterling, E. Shapiro: The Art of Prolog, 2nd Ed., MIT Press, 1994
- M. Hanus: The Integration of Functions into Logic Programming: From Theory to Practice, Journal of Logic Programming, Vol. 19,20, pp. 583-628, 1994
- S. Antoy, M. Hanus: Functional Logic Programming, Communications of the ACM, Vol. 53, No. 4, pp. 74-85, 2010
- Armstrong et al: Concurrent Programming in Erlang, Prentice Hall, 1996

Kapitel 1

Funktionale Programmierung

1.1 Funktionsdefinitionen

Grundidee: Programm	=	Menge von Funktionsdefinitionen (+ auszuwertender Ausdruck)
Funktionsdefinition:		Funktionsname
	+	formale Parameter
	+	Rumpf (Ausdruck)
Allgemein:		$f\ x_1 \ldots x_n\ =\ e$

Ausdrücke:

- elementar: Zahlen `3`, `3.14159`
- elementare Funktion: `3+4`, `1+5*7`
- Funktionsanwendungen:

$$\begin{array}{ccccc} f & & a_1 & \ldots & a_n \\ \uparrow & & \nwarrow & & \nearrow \\ \textit{Funktion} & & \textit{aktuelle} & & \textit{Parameter} \end{array}$$

- bedingte Ausdrücke: `if` b `then` e_1 `else` e_n

Beispiel: Quadratfunktion:

```
square x = x*x
```

"`=`" steht für Gleichheit von linker und rechter Seite
$\Rightarrow$ `square 9 = 9*9 = 81`

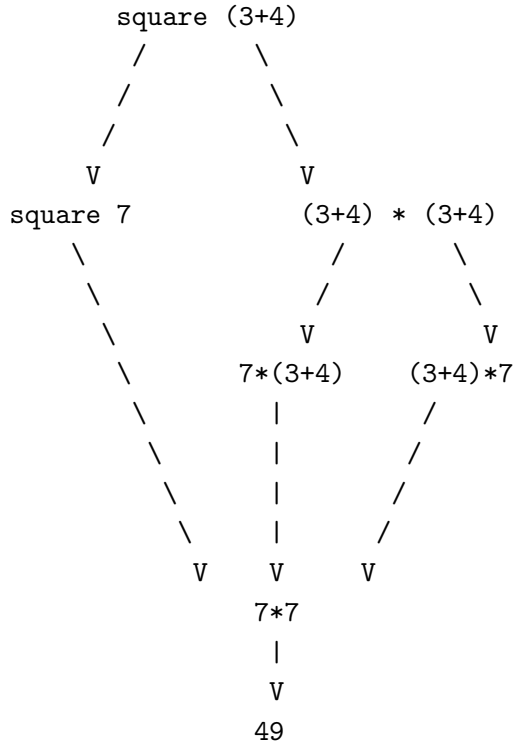
Rechnen: Gleiches durch Gleiches ersetzen bis ein Wert herauskommt:
Wie genau?

Auswerten von Ausdrücken:
Falls Ausdruck elementarer Funktionsausdruck mit ausgewerteten Argumenten: Ersetze durch Ergebnis
Sonst:

1. Suche im Ausdruck eine Funktionsanwendung (auch: **Redex**: **red**ucible **ex**pression)
2. Substitutiere in entsprechender Funktionsdefinition formale Parameter durch aktuelle (auch im Rumpf!)
3. Ersetze Teilausdruck durch Rumpf

Wegen 2: *Variablen* in Funktionsdefinitionen sind *unbekannte Werte*, aber keine Speicherzellen! (Unterschied: imperativ ↔ deklarativ)

Beispiel: Auswertungsmöglichkeiten von (Anwendung elementarer Funktionen: werte vorher Argumente aus)



Konkrete Sprache: Festlegung einer **Auswertungsstrategie** (vgl.1.)

Strikte Sprachen: Wähle immer linken inneren Teilausdruck, der Redex ist (Bsp.: linker Pfad)
(*leftmost innermost, call-by-value, application order reduction, eager reduction*)

+ einfache effiziente Implementierung
− berechnet evtl. keinen Wert, obwohl ein Ergebnis existiert (bei anderer Auswertung)

Nicht-strikte Sprachen Wähle immer linken äußeren Teilausdruck, der Redex ist
(Bsp.: mittlerer Pfad)
(*leftmost outermost, call-by-name, normal order reduction*)
Besonderheit: Argumente sind evtl. unausgewertete Ausdrücke (nicht bei elementaren Funktionen)

+ berechnet Wert, falls einer existiert
+ vermeidet überflüssige Berechnungen
− evtl. Mehrfachauswertung (z.B. `(3+4)`)
  ⇒ Verbesserung durch *verzögerte/faule Auswertung* (*call-by-need*, *lazy evaluation*)

LISP, SML: strikt
Miranda, Haskell: nicht-strikt

Fallunterscheidung: Für viele Funktionen notwendig, z.B.

$$abs(x) = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{sonst} \end{cases}$$

Implementierung mit if-then-else (Beachte: `abs` vordefiniert!):

```
absif x = if x >= 0 then x else -x
```

Aufruf: `absif (-5)` $\rightsquigarrow$ `5`

Implementierung mit *bedingten Gleichungen* (guarded rules)

```
absg x | x >= 0     = x
       | otherwise  = -x
```

Beduetung der Bedingungen:
Erste Gleichung mit erfüllbarer Bedingung wird angewendet (`otherwise` $\approx$ `True`)
⇒ mathematische Notation

Unterschiedliche Möglichkeiten zur Funktionsdefinition:

Beispiel: **Fakultätsfunktion**: $fac(n) \rightsquigarrow n * (n-1) * \cdots * 2 * 1$

Rekursive Definition: $fac(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n * fac(n-1) & \text{falls } n > 0 \end{cases}$

Implementierung:

1. if-then-else:
   `fac n = if n==0 then 1 else n * fac (n-1)`
   Nicht ganz korrekt, weil rekursiver Aufruf auch bei `n<0`

2. bedingte Gleichungen:

```
fac n | n == 0  = 1
      | n>0     = n * fac (n-1)
```

Korrekt, aber etwas umständlich

3. **Muster** im formalen Parametern (*pattern matching*)
   Bisher: formale Parameter: nur Variablen
   Nun auch Muster: Gleichung nur anwendbar, wenn aktueller Parameter zum Muster „paßt“

   ```
   fac 0     = 1
   fac (n+1) = (n+1) * fac n
   ```

   wobei `(n+1)` = Muster für alle positiven Zahlen
   (allgemein: `(n+k)`: Muster für ganze Zahlen $\geq$ `k`)

   Beispielberechnung:

   ```
   fac 3                    3 paßt auf Muster n+1 für n=2 ⇝
    = (2+1) * fac(2)
    = 3 * fac(2)            2 paßt auf Muster n+1 für n=1
    = 3 * (1+1) * fac (1)
    = 3 * 2 * fac (1)
    = 3 * 2 * (1+0) * fac (0)
    = 3 * 2 * 1 * 1   ⇝   6
   ```

Musterorientierter Stil:

- kürzere Definitionen (kein if-then-else)
- leichter lesbar
- leichter verifizierbar (vgl. `append`, Kapitel 0, S. 6)

Vergleiche:

```
and True  x = x
and False x = False
```

mit

```
and x y = if x==True then y else False
```

oder

```
and x y | x==True  = y
        | x==False = False
```

Mögliche Probleme:

- Reihenfolge bei Musteranpassung
  (links-nach-rechts, oben-nach-unten, best-fit?)
- Deklarativ sinnlose Definitionen:

```
f True  = 0
f False = 1
f True  = 2     -- ??????
```

Spezifikationssprache ↔ deklarative Programmiersprache

Deklarative Programmiersprache:
Definition von Eigenschaften statt genauer Programmablauf
$\rightsquigarrow$ Spezifikation $\stackrel{?}{=}$ Programm ?

Nein, denn: Programm: effektiv ausführbar (z.B. Reduktion von Ausdrücken)
Spezifikation: formale Beschreibung, evtl. nicht ausführbar

Programmieren: Überführung einer Spezifikation in eine ausführbare Form, dies wird vereinfacht durch deklarative Sprachen

Beispiel: **Quadratwurzelberechnung nach Newton**
Wurzelfunktion: $\sqrt{x} = y$, so daß $y \geq 0$ und $y^2 = x$

Spezifikation, aber nicht effektiv (Raten von y?)

Approximationsverfahren: *Newtonsches Iterationsverfahren*
Gegeben: Schätzung für y
bessere Schätzung: Mittelwert von y und $\frac{x}{y}$
Beispiel:

$x = 2$	*Schätzung* :	*Mittelwert* :
	1	$\frac{1+2}{2} = 1.5$
	1.5	$\frac{1.5+1.3333}{2} = 1.4167$
	1.4167	$\frac{1.4167+1.4118}{2} = 1.4142$

Funktionale Definition: Iteration mit Abbruchbedingung:

```
iter y x = if (ok y x) then y else iter (improve y x) x
```

(ok y x): *Abbruch*; (improve y x): *Verbesserung*

```
improve y x = (y + x/y) / 2.0
```

("2.0" ist wichtig wegen strikter Unterscheidung zwischen `Int` und `Float`)

```
ok y x = absf (y*y-x) < 0.001
```

`absf` ⇒ Absolutbetrag für Gleitkommazahlen

```
absf x | x >= 0.0  = x
       | otherwise = -x
```

Gesamtlösung:

```
wurzel x = iter 1.0 x

? wurzel 9.0  ⇝  3.00009
```

Nachteil dieser Lösung:

- `wurzel`, `iter`, `improve`, `ok` gleichberechtigte (globale) Funktionen
- Argument `x` muß "durchgereicht" werden
- mögliche Konflikte bei Namen wie `ok`, `improve`

Definition mit **where-Klausel**:

```
wurzel x = iter 1.0
       where iter y    = if ok y then y else iter (improve y)
             improve y = (y + x/y) / 2.0
             ok y      = absf (y*y-x) < 0.001
```

lokale Deklarationen

Gültigkeitsbereich: where-Deklarationen sind nur in der vorhergehenden rechten Gleichungsseite (bzw. Folge von bedingten Gleichungsseiten) sichtbar

Weitere Form lokaler Deklarationen: **let-Ausdrücke**

`let` <Deklarationen> `in` <Ausdruck>

Die Deklarationen sind nur im Ausdruck sichtbar

Beispiel: Definition von $f(x, y) = y(1 - y) + (1 + xy)(1 - y) + xy$

```
f x y = let a = 1-y
            b = x*y
        in  y*a+(1+b)*a+b
```

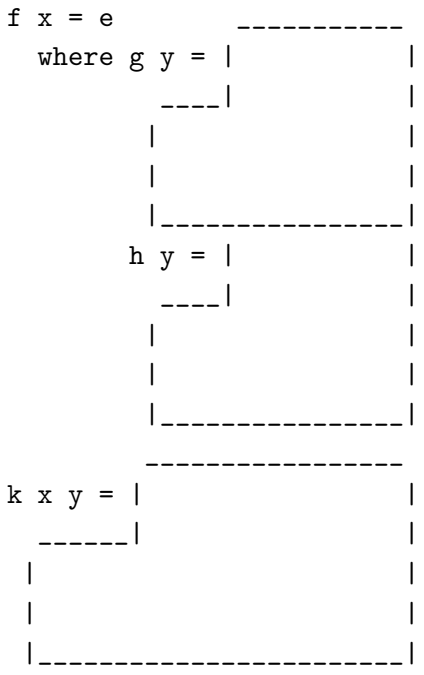
Wann enden Deklarationen?
Formatfrei: *Klammerung:*

```
let {...;...} in ...
f x = e where {...;...}
```

Formatabhängig: *Abseitsregel* (*offside rule*, *layout rule*):

1. Erstes Symbol nach `let`, `where` (und `of` bei case-Ausdrücken) legt *linken Rand* des Geltungsbereiches fest.

2. Beginnt neue Zeile rechts vom Rand ⇒ Fortsetzung der vorigen Zeile
3. Beginnt neue Zeile genau am Rand ⇒ neue Deklaration
4. Beginnt neue Zeile links vom Rand ⇒ Geltungsbereich endet davor, Zeile gehört zum umfassenden Bereich



- gewöhnungsbedürftig, aber:
- erlaubt kompakte, übersichtliche Schreibweise

1.2 Datentypen

Moderne funktionale Sprachen haben *strenges Typkonzept*, d.h.

- alle Werte sind klassifiziert in *Typbereiche* (boolesche Werte, Zeichen, ganze Zahlen, ...)
- jeder zulässige Ausdruck hat einen Typ: beschreibt Wert, der evtl. berechnet wird

Schreibweise: `e :: t` : Ausdruck `e` hat den Typ `t`

Basistypen

Wahrheitswerte: `Bool`

Konstanten: `True`, `False`
Funktionen: `&&` (und) `||` (oder) `not` (nicht)

Ganze Zahlen: `Int`

Konstanten: `0, 1, -42, ...`
Funktionen: `+, -, *, /, div, mod, ...`

Gleitkommazahlen: `Float`

Konstanten: `0.3 1.5 e-2 ...`

Achtung: keine automatische Konversion zwischen Int und Foat:

```
2 == 2.0  ⇝ error
2 +  2.0  ⇝ error
```

Zeichen: `Char`

Konstanten: `'a' '0' '\n'`

Strukturierte Typen:

Listen (Folgen) von Elementen von Typ `t`: `[t]`
Eine Liste ist

- entweder leer: Konstante `[]`
- oder nicht-leer, d.h. bestehend aus einem (Kopf-) Element `x` und einer Restliste `l`: `x:l`

Beispiel: Liste mit Elementen `1,2,3`: `1:(2:(3:[])) :: [Int]`

1. `:` ist rechtsassoziativ: `1 : 2: 3 : []`
2. Direkte Aufzählung aller Elemente: `[1, 2, 3]`
   ⇒ `[1,2,3] == 1:[2,3] == 1:2:[3]`
3. Abkürzungen für Listen von Zahlen:

   - `[a..b]` == `[a, a+1, a+2, ..., b]`
   - `[a..]` == `[a, a+1, a+2, ...` Unendlich!
   - `[a,b..c]` == `[a, a+s, a+2s, ...,c]` mit Schrittweite `s=b-a`
   - `[a,b..]` == `[a, a+s, a+2s, ...` mit Schrittweite `s=b-a`

Operationen auf Listen (viele vordefiniert)
`length xs`: Länge der Liste `xs`

```
length []     = 0
length (x:xs) = 1 + length xs
```

`++`: Verkettung von Listen (rechtsassoziativ)

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Strings, Zeichenketten: `String` identisch zu `[Char]`
Konstanten: `"Hallo" == ['H','a','l','l','o']`
Funktion: wie auf Listen

Tupel: $(t_1, t_2, \ldots, t_n)$: Struktur („Record“) mit Elementen mit Typen $t_1, t_2, \ldots, t_n$
Konstanten: `(1,'a',True) :: (Int,Char,Bool)`

Funktionale Typen: t_1`->`t_2: Funktionen mit Argument von Typ t_1 und Ergebnis von Typ t_2

Beispiel: `square:: Int -> Int`

Konstante: `\` x `->` e

↑	↑	↑
λ(lambda)	*Parameter*	*Rumpf*

äquivalent zu $f\ x = e$, falls f Name dieser Funktion

„λ-Abstraktion“: Definiert „namenlose“ Funktion

Bsp.: `\x -> 3*x+4 :: Int -> Int`

Besonderheit funktionaler Sprachen:
Funktionale Typen haben gleichen Stellenwert wie andere Typen („Funktionen sind Bürger 1. Klasse“, „functions are first class citizens“), d.h. sie können vorkommen in

- Argumenten von Funktionen
- Ergebnissen von Funktionen
- Tupeln
- Listen

⋮

Falls Funktionen in Argumenten oder Ergebnisse anderer Funktionen vorkommen:
⇒ **Funktionen höherer Ordnung**

```
polynomdiff:: (Float -> Float) -> (Float -> Float)
```

Anwendung (Applikation) einer Funktion f auf Argument x:
mathematisch: $f(x)$
In Haskell: Klammern weglassen: $f\ x$ (Anwendung durch Hintereinanderschreiben)

Funktionen mit mehreren Argumenten:
Funktionale Typen haben nur ein Argument.
Zwei Möglichkeiten zur Erweiterung:

1. Mehrere Argumente als Tupel:
   `\(x,y) -> x+y+2 :: (Int,Int) -> Int`
   `add(x,y) = x+y`

2. „Curryfizierung“ (nach Schönfinkel und Curry)

   - Tupel sind kartesische Produkte: $(A, B) \hat{=} A \times B$
   - Beobachtung: Funktionsräume $(A \times B) \to C$ und $A \to (B \to C)$ sind isomorph
     $\Rightarrow \forall f : (A \times B) \to C$ existiert äquivalente Funktion $f' :: A \to (B \to C)$

   Beispiel:

   ```
   add':: Int -> (Int -> Int)
   add' x = \y -> x+y

   add (x,y) = x+y
   (add' x) y = (\y -> x+y) y = x+y
   ```

   Also: Statt Tupel von Argumenten Hintereinanderanwendung der Argumente
   Interessante Anwendung bei partieller Applikation:

   `(add 1)`: Funktion, die eins addiert $\approx$ Inkrementfunktion
   $\Rightarrow$ `(add 1) 3 = 4`

Viele Anwendungen: generische Funktionen ($\to$ Kapitel 1.4, Funktionen höherer Ordnung)

Daher: Standard in Haskell: curryfizierte Funktionen

$f\ x_1\ x_2 \ldots x_n \hat{=} (\ldots((f\ x_1)\ x_2)\ldots x_n)$ (Applikation linksassoziativ)

$t_1 \to t_2 \to \ldots \to t_n \hat{=} t_1 \to (t_2 \to (\ldots \to t_n)\ldots)$ (Funktionstyp rechtsassoziativ)

Daher:

```
add :: Int -> Int -> Int   =  Int -> (Int -> Int)
add x y = x+y
 ⋮
add 1 2 = ((   add 1   ) 2)
            ::Int → Int
```

Achtung: `add` 1 2 $\neq$ `add` $(1, 2)$:
Auf der linken Seite muß `add` den Typ `Int -> Int -> Int` haben
Auf der rechten Seite muß `add` den Typ `(Int, Int) -> Int` haben

Aber: es gibt Umwandlungsfunktionen zwischen Tupeldarstellung und curryfizierter Darstellung (`curry`, `uncurry`)

Benutzerdefinierte (algebraische) Datentypen:

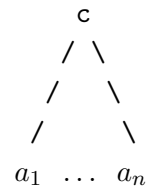
Objekte bestimmten Typs sind aufgebaut aus

Konstruktoren: Funktion mit entsprechendem Ergebnistyp, die nicht reduzierbar ist (frei interpretiert, keine Funktionsdefinition)

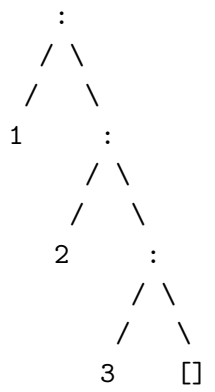
Beispiel:

Typ	**Konstruktoren**
`Bool`	`True False`
`Int`	`0 1 2...-3...`
Listen:	`[] :`

Konstruktion (`C` $a_1 \ldots a_n$) entspricht Baumstruktur:



Beispiel: Liste `1:2:3:[]` entspricht Baumstruktur:



Dagegen ist `++` eine Funktion, die Listen (Bäume) verarbeitet: `[1] ++ [2] = [1,2]`

Konvention in Haskell:

- Konstruktoren beginnen mit Großbuchstaben
- Funktionen beginnen mit Kleinbuchstaben

Definition neuer Datentypen: Festlegung der Konstruktoren

Datendefinition in Haskell:

`data` t `=` C_1 $\mathtt{t}_{11}$ `...` $\mathtt{t}_{1n_1}$ `|` `...` `|` C_k $\mathtt{t}_{k1}$ `...` $\mathtt{t}_{kn_k}$

führt neuen Typ t und Konstruktoren $C_1, \ldots C_k$ mit

$$C_i :: t_{11} \to \ldots \to t_{in_i} \to t$$

ein.
Beispiel: Neue Listen:

```
data List = Nil | Cons Int List
```

definiert neuen Datentyp List und Konstruktoren

```
Nil  :: List
Cons :: Int -> List -> List
```

Spezialfälle:
Aufzählungstypen:

```
data Color = Red | Yellow | Blue
```

```
data Bool = True | False
```

Verbundtypen (Records):

```
data Complex = Complex (Float, Float)
```

```
addc :: Complex -> Complex -> Complex
```

```
addc (Complex (r1,i1)) (Complex (r2,i2)) = Complex (r1 + r2, i1 + i2)
```

Rekursive Typen (variante Records): Listen (s.o.)
Binärbäume mit ganzzahligen Blättern:

```
data True = Leaf Int | Node Tree Tree
```

Addiere alle Blätter:

```
addLeaves (Leaf x) = x
```

```
addLeaves (Node t1 t2) = (addLeaves t1) + (addLeaves t2)
```

Beachte: Kein Baumdurchlauf, sondern Spezifikation!

1.3 Pattern Matching

Bevorzugter Programmierstil: Funktionsdefinition durch Muster (pattern) und mehrere Gleichungen
Auswertung: Auswahl und Anwenden einer passenden Gleichung
Auswahl → "patttern matching"
Vergleiche: Definition der Konkatenation von Listen:
musterorientiert:

```
append []  ys  =  ys
append (x : xs)  ys  =  x: (append xs ys)
```

Ohne Muster (wie in klassischen Programmiersprachen):

```
append xs ys = if xs == []
               then ys
               else (head xs): (append (tail xs) ys)
```

wobei head, tail: vordefinierte Funktionen auf Listen
head → Kopfelement
tail → Rest der Liste
d.h. es gilt immer:

```
head x:xs = x
tail x:xs =xs
bzw. (head l) : (tail l) = l
```

Andere Sprechweise:
: ist Konstruktor für Datentyp Liste
head, tail sind Selektoren für Datentyp Liste
Somit:
Musterorientierter Stil:

- Schreibe Konstruktoren in Argumente von linken Regelseiten
- haben Wirkung von Selektoren
- linke Seiten: „Prototyp“ für den Funktionsaufruf, bei dem diese Regel anwendbar ist
  Beispiel: `append [] ys = ys` (*)
  ist anwendbar, falls 1. Argument die Form einer leeren Liste hat, 2. Argument beliebig

Wann paßt ein Muster?
Ersetze Variablen in Muster (“binde Variablen“) so durch andere Ausdrücke, daß ersetztes Muster syntaktisch identisch zum gegebenen Ausdruck ist. Ersetze in diesem Fall Ausdruck durch rechte Regelseite (mit den entsprechenden Bindungen!).
Beispiel: Regel (*) und Ausdruck

- `append [] [1,2,3]`: Binde xs an `[1,2,3]`
  ⇒ linke Seite paßt ⇒ Ersetzung durch “`[1,2,3]`“
- `append [1] [2,3]` : Muster in (*) paßt nicht, da Konstruktor [] immer verschieden von : (beachte: `[1] =: (1,[])`)

Was sind mögliche Muster?

Muster	Paßt, falls
`x`	Variable paßt auf jeden Ausdruck und wird an diesen gebunden
`[] True 'a'`	Konstanten passen nur auf gleichen Wert
`(x:xs)(Node` $t_1 t_n$`)`	Konstruktoren passen nur auf gleichen Konstruktor. Zusätzlich müssen die Argumente jeweils passen
(t_1, t_2, t_3)	Tupel paßt mit Tupel gleicher Stelligkeit (Argumentzahl), zusätzlich müssen die Argumente jeweils passen
`(x+k)` (k=natürliche Zahlkonstate)	Paßt auf ganze Zahlen $\geq$ k, wobei dann x an die Zahl minus k gebunden wird Beispiel: `sub3 (n+3) = n`
`v@pat`	Paßt, falls Muster "paßt". Zusätzlich wird v an gesamten Wert gebunden Beispiel: Statt `fac (n+1) = (n+1) * fac n` auch möglich: `fac v@(n+1) = v * fac n` Effekt: Vermeidung des Ausrechnens von (n+1) in der rechten Seite $\Rightarrow$ Effizienzsteigerung
_	Joker, "wildcard pattern": Paßt auf jeden Ausdruck, keine Bindung

Problem: Muster mit mehrfachem Variablenvorkommen:

```
equ x x = True
```

Woran wird Variable x gebunden?
$\Rightarrow$ Solche Muster sind unzulässig!

In Mustern kommt jede Variable höchstens einmal vor

Ausnahme: _ : Jedes Vorkommen steht für eine andere Variable

Problem: Anwendbare Ausdrücke an Positionen, wo man den Wert wissen muß
Beispiel:

```
fac 0 = 1
```
Aufruf: `fac (0+0)`

(0+0) $\rightarrow$ Paßt erst nach Auswertung zu 0

$\Rightarrow$ Falls beim pattern matching der Wert relevant ist (d.h. bei allen Mustern außer Variablen), wird der entsprechende aktuelle Parameter ausgewertet.

$\Rightarrow$ Form der Muster beeinflußt Auswertungsverhalten und damit auch des Terminationsverhalten!

Beispiel: Betrachte folgende Definition der Konjunktion: ("Wahrheitstafel"):

```
and   False   False   = False
and   False   True    = False
and   True    False   = False
and   True    True    = True
```

Effekt der Muster: bei (and $t_1 t_2$) müssen beide Argumente ausgewertet werden, um eine Regel anzuwenden.
Betrachte nichtterminierende Funktion:

```
loop: Bool → Bool
loop = loop   ⇝ Auswertung von loop terminiert nicht
```

Dann: `and False loop`
terminiert nicht

Verbesserte Definition durch Zusammenfassen von Fällen:

```
and1 False x = False
and1 True  x = x
```

Effekt: 2. Argument muß nicht ausgenutzt werden
⇒ `and1 False loop` ⇝ `False`
Weiteres Problem: Reihenfolge der Abarbeitung von Mustern:
Logisches Oder ("parallel or")

```
or True    x    = True       (1)
or  x    True   = True       (2)
or False False  = False      (3)
```

Betrachte: `or loop True`
Regel(2) paßt (mit `x` → `loop`): `True`
Aber: Regel (1) könnte auch passen: Auswertung des 1. Argumentes ⇝ Endlosschleife

In Haskell: Abarbeitung der Muster von links nach rechts, Auswertung eines Arguments, falls ein Muster in einer Regel es verlangt.

Präzisierung durch case-Ausdrücke:
case-Ausdrücke haben die Form

```
case e of
```

$$\begin{array}{llll} c_1 & x_{11} & \ldots & x_{1n_1} \\ \vdots & & & \vdots \\ \vdots & & & \vdots \\ c_k & x_{k1} & \ldots & x_{kn_k} \end{array} \quad \rightarrow \quad \begin{array}{l} e_1 \\ \vdots \\ \vdots \\ e_k \end{array}$$

wobei $c_1, \ldots, c_k$ Konstruktoren des Datentyps von x sind (Anmerkung: Haskell erlauben auch allgemeine Formen).
Beispiel:

```
append xs ys = case xs of
                  []  → ys
                  x:l → x: append l ys
```

Bedeutung des case-Ausdruckes:

1. Werte e aus (bis ein Konstruktor oben steht)
2. Falls die Auswertung $c_i\ a_1 \ldots a_{n_i}$ ergibt (sonst: Fehler), binde x_{ij} an $a_j (j = 1, \ldots, n_i)$ und ersetze case-Ausdrück durch e_i

Obiges Beispiel zeigt: Pattern matching in case-Ausdrücken übersetzbar
Im folgenden: Definition der Semantik von Pattern matching durch Übersetzung in case-Ausdrücke.

Gegeben: Funktionsdefinition
$f\ p_{11} \ldots p_{1n} = e_1$
$\vdots$
$\vdots$
$f\ p_{m1} \ldots p_{mn} = e_m$

Übersetzung mittels Funktion match:

```
f x1 ... xn =
⟦match [x1, ..., xn] [([p11, ..., p1n], e1),
                      :
                      :
                      ([pm1 ..., pmn], em)]
                      ERROR⟧
```

allgemeine Form von match: `match xs eqs E`
xs → Liste von Argumentvariablen
eqs → Liste von Pattern-Rumpf Paaren
E → Fehlerfall, falls kein Pattern aus eqs passt

Definition durch vier Transformationsregeln:

1. **Alle Muster fangen mit einer Variablen an:**

```
match (x:xs) [(v1 : vs1,  e1),
                  ⋮
              (vm : vsm,  em)] E
⇒ match xs [(vs1,  e1[v1  ↦  x]),
                  ⋮
            (vsm,  em[vm  ↦  x])] E
```

wobei $e[v \mapsto t]$ steht für: Ersetze in e alle Vorkommen der Variablen v durch t
Somit: keine Argumentauswertung bei Variablenmustern
Anmerkung: auch anwendbar falls m=0 (leere Gleichungsliste)

2. **Alle Muster fangen mit Konstruktor an:**
Dann Argument auswerten + Fallunterscheidung über Konstrukten

$$
\begin{array}{l}
\texttt{match (x:xs) eqs E} \quad \texttt{(wobei eqs} \neq \texttt{[])} \\
\texttt{= match (x:xs) (eqs}_1 \texttt{ ++ ... eqs}_k\texttt{) E} \\
\qquad\qquad \Rightarrow \texttt{Umsortierung von eqs, so dass alle} \\
\qquad\qquad\quad \texttt{Pattern in eqs}_i \texttt{ mit Konstruktor c}_i \texttt{ beginnen, wobei} \\
\qquad\qquad\quad \texttt{c}_1, \ \ldots, \texttt{ c}_k \texttt{ alle Konstruktoren vom Typ von x} \\
\qquad\qquad\quad \texttt{d.h. eqs}_i \texttt{ = [}((C_i \ p_{i,1,1} \ \ldots \ p_{i,1,n_i}) : \ ps_{i,1}, \ e_{i,1})\texttt{,} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad ((C_i \ p_{i,m_i,1} \ \ldots \ p_{i,m_i,n_i}) : \ ps_{i,m_i}, \ e_{i,m_i})\texttt{]}
\end{array}
$$

(Beachte: falls in eqs Konstruktor c_j nicht vorkommt $\Rightarrow eqs_j = []$)

$$
\begin{array}{l}
\texttt{= case x of} \\
\quad C_1 \ x_{1,1} \ \ldots \ x_{1,n_1} \ \rightarrow \texttt{match ([}x_{1,1}, \ \ldots, \ x_{1,n_1}\texttt{]} \ ++ \ xs) \ eqs'_1 \ E \\
\quad\ \vdots \\
\quad\ \vdots \\
\quad C_k \ x_{k,1} \ \ldots \ x_{k,n_k} \ \rightarrow \texttt{match ([}x_{k,1}, \ \ldots, \ x_{k,n_k}] ++ xs) \ eqs'_k \ E \\
\texttt{wobei } eqs'_i \ = \ [([p_{i,1,1}, \ \ldots, \ p_{i,1,n_i}] ++ \ ps_{i,1}, \ e_{i,1})\texttt{,} \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad ([p_{i,m_i,1} \ \ldots \ p_{i,m_i,n_i}] ++ \ ps_{i,m_i}, \ e_{i,m_i})\texttt{]} \\
\qquad x_{i,j} \texttt{ : neue Variablen}
\end{array}
$$

Beispiel: Übersetzung von `append`:

$$
\begin{array}{l}
\texttt{match } [x_1, \ x_2][([[], \ ys,], \ ys), \ ([x : l, \ ys], \ x : \ \texttt{append } l \ ys)] \ \texttt{ERROR} \\
\texttt{= case } x_1 \texttt{ of} \\
\quad \texttt{[]} \ \rightarrow \ \underbrace{\texttt{match } [x_2] \ [([ys], \ ys)]\texttt{ERROR}}_{(x)} \\
\quad x_3 : x_4 \ \rightarrow \ \texttt{match } [x_3, \ x_4, \ x_2][([x, l, ys], \ x : \ \texttt{append } \ l \ ys)] \ \texttt{ERROR}
\end{array}
$$

(x) = 1. Transformation

```
match [] [([], x2)] ERROR
  !
  = x2
  ↑
Transformation f"ur leere Musterliste
```

3. **Musterliste ist leer:**

   ```
   match [] [([] e)] E = e (genau 1 Regel anwendbar)
   match [] [] E = E       (keine Regel anwendbar: Fehleralternative)
   ```

   Anmerkung: Theoretisch kann evtl. auch mehr als 1 Regel übrigbleiben, z.B. bei

   ```
   f True  = 0
   f False = 1
   f True  = 2
   ```

   In diesem Fall: Fehlermeldung oder Wahl der ersten Alternative

4. **Muster fangen mit Konstruktoren als auch Variablen an:**
   Gruppiere diese Fälle:

   ```
   match xs eqs E =
     match xs eqs1 (match xs eqs2 (...(match xs eqsn E) ...))
   ```

   wobei: `eqs = eqs1 ++ eqs2 ++ ...++ eqsn`
   und in `eqsi` ($\neq$`[]`) beginnen alle Muster entweder mit Variablen oder mit Konstruktoren; falls in eqs_i alle Muster mit Variablen anfangen, dann fangen in eqs_{i+1} alle Muster mit Konstruktoren an (oder umgekehrt)

Eigenschaften dieses Algorithmus:

1. **Vollständigkeit**: Jede Funktionsdefinition wird in case-Ausdrücke übersetzt (klar wegen vollständiger Fallunterscheidung)

2. **Terminierung**: Die Anwendung der Transformationen ist endlich.
   Definiere:

   - Größe eines Musters: Anzahl der Symbole in diesem
   - Größe einer Musterliste: Summe der Mustergrößen

   Dann: jeder match-Aufruf wird in einer Transformation ersetzt durch Aufrufe mit kleinerer Musterlistengröße $\Rightarrow$ Terminierung

Beispiel: Übersetzung von Oder:

```
   or  True   x     = True    (1)
   or  x      True  = True    (2)
   or  False  False = False   (3)
⇒
or x y =   ⟦ match [x,y] [((True, x), True),
                          ((x, True), True),
                          ([False, False], False)] ERROR ⟧
```

3. Transformation:

```
match [x,y][([True, x], True)]
  (match [x,y][([x, True], True)]
    (match[x,y][([False, False], False)] ERROR))
```

Transformation

```
2,1,3       case x of
              True  → True
1,2,3         False → case y of
                  True → True
2,3,2,3,3         False → case x of
                     True  → ERROR
                     False → case y of
                           True  → ERROR
                           False → False
```

Nun ist klar, wie `or loop True` abgearbeitet wird: zunächst case über 1. Argument ⇒ Endlosschleife
keine Endlosschleife bei Vertauschung von (1) + (2)!
Ursache: Regeln (1) + (2) "überlappen", d.h. für das Muster `or True True` sind bei Regeln anwendbar

⇒ **Bei Funktionen mit überlappenden Mustern hat die Regelreihenfolge Einfluß auf Erfolg des Pattern Matching.**

⇒ **Vermeide überlappende linke Regelseiten!**

Ausreichend für Reihenfolgeunabhängigkeit?
Nein! Betrachte:

```
diag x      True   False = 1
diag False  x      True  = 2
diag True   False  x     = 3
```

Gleichungen nicht überlappend
Aufruf: `diag loop True False` $\rightsquigarrow$ 1
Gleichungen in umgekehrter Reihenfolge $\rightsquigarrow$ Endlosschleife

Prinzipielles Problem: es existiert nicht für jede Funktionsdefinition eine sequentielle Auswertungsstrategie, die immer einen Wert berechnet. (vgl. dazu Arbeiten von [Huet/Levy '79]

Ursache für Probleme: Tranformation 4, bei der Auswertung und Nichtauswertung von Argumenten gemischt ist $\Rightarrow$
Funktionsdefinition heißt *uniform*, falls Transformation 4 bei Erzeugung von case-Ausdrücken nicht benötigt wird.
Intuitiv: Keine Vermischung von Variablen und Konstruktoren an gleichen Positionen.

Satz Bei uniformen Definitionen hat die Reihenfolge der Regeln keinen Einfluß auf das Ergebnis.

Beispiel:

```
or False x     = x
or True  False = True
or True  True  = False
```

ist uniform (trotz Vermischung im 2. Argument) wegen links-nach-rechts Pattern-Matching

```
or' x     False = x
or' False True  = True
or' True  True  = False
```

ist nicht uniform. Es wäre aber uniform bei Pattern matching von rechts nach links.
Andere Pattern Matching Strategien denkbar, aber: übliche funktionale Sprachen basieren auf links$\rightarrow$rechts Pattern Matching.
Ausweg: Nicht-uniforme Definitionen verbieten? Manchmal ganz praktisch: Umkehrung einer 2-elementigen Liste (und nur diese!):

```
rev2 [x,y] = [y,x]
rev2  xs   =  xs
```

Als uniforme Definition:

```
rev2  []      = []
rev2  [x]     = [x]
rev2  [x,y]   = [y,x]
rev2  x:y:z:l = x:y:z:l
```

Andere Alternativen: nicht-sequentielle Auswertung: statt leftmost outermost: Parallel outermost: vollständig, aber aufwendig

1.4 Funktionen höherer Ordnung

Funktionen: Bürger 1. Klasse ⇒ können auch als Parameter auftreten
Anwendung:

- generische Funktionen
- Programmschemata

⇒ Wiederverwendbarkeitn, Modularität

Altes Konzept aus der Mathematik:
Beispiel: Ableitung: Funktion deriv:: $\underbrace{(Float \to Float)}_{Eingabefunktion} \to \underbrace{(Float \to Float)}_{Ausgabefunktion}$

Numerische Berechnung der 1. Ableitung:

$$f'x = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx}$$

Näherung (wähle kleines dx):

```
deriv f = f'
      where f'x = (f(x+dx)-f x)/dx
            dx = 0.00001
```

$\underbrace{\texttt{deriv sin 0.0}}_{\cos}$ ⇝ 1.0

$\underbrace{\texttt{deriv (\ x → x*x)}}_{2*x}$ 1.0 ⇝ 2.00272

Wichtige Anwendung: Definition von *Programmierschemata*
Beispiel:

- Quadrieren aller Elemente einer Zahlenliste:

```
sqlist []     = []
sqlist [x:xs] = (square x): sqlist xs
```

sqlist [1,2,3,4] ⇝ [1,4,9,16]

- Chiffrieren von Strings durch zyklische Verschiebung um 1 Zeichen:

```
code c | c=='Z'    = 'A'
       | c=='z'    = 'a'
       | otherwise = chr (ord c + 1)
```

Hierbei ist:
chr :: Int → Char
ord :: Char → Int

Anwendung auf Strings:

```
codelist  []    = []
codelist (c:cs) = (code c): codelist cs
codelist 'ABzXYZ' → 'BCaYZA'
```

Anmerkung:

- `sqlist` und `codelist` haben ähnliche Struktur
- Unterschied: „`code`“ statt „`squense`“

Verallgemeinerung durch eine Funktion mit Funktionsparameter:

```
map::(a → b) → [a] → [b]
```

wobei: `a` und `b` = Typvariablen, ersetzbar durch beliebeige Typen wie z.B. `Int` oder `Char` (⇝ Kap 1.5 “Typsystem“)

```
map f []     = []
map f (x:xs) = f x: map f xs
```

Nun erhalten wir obige Funktion als Spezialisierung von map:

```
sqlist xs   = map square xs
codelist cs = map code cs
```

Vorteil: `map` universell einsetzbar:

- Andere Codierungsfunktion für Zeichen, z.B.
  `rot13:: Char → Char` (verschiebe um 13 Zeichen)
  `rot13list cs = map rot 13 cs`
- Inkrementierung einer Zahlenliste um 1:
  `inclist xs = map` $\underbrace{\texttt{(\x → x+1)}}_{Lambda-Abstraktion, anonyme Funktion}$ xs

Auffallend: Bei allen Definitionen ist letztes Argument `(xs, cs)` identisch ⇒ Schreibe „Funktionsgleichungen“

```
sqlist   = map square
codelist = map code
inclist  = map (\ x → x+1)
           ⇒ Partielle Applikation, d.h. es fehlen Argumente
```

Nützlich auch in anderen Anwendungen:

```
add x y = x+y
inc =       add 1
```
Partielle Anwendung

Daher: `inc` ist keine Konstante, sondern eine Funktion `::Int →Int`
Voraussetzung für partielle Anwendung: Funktion muß curryfizierten Typ $t_1 \to t_2 \to \ldots \to t_n$ haben (vgl. Kapitel 1.2)

$f\ x$ steht immer für die (partielle) Anwendung von f auf ein Argument, d.h. falls
$f ::$ $t_1 \to t_2 \to \ldots \to t_n$ und $x :: t_1$, dann ist $fx :: t_2 \to \ldots \to t_n$

Sonderfall: Infixoperationen (+, -, *, /, ++, ≤, ...)
Zwei partielle Anwendungen:
Präfixapplikation : $(2 <) \hat{=} \backslash x \to 2 < x$
Postfixapplikation: $(> 2) \hat{=} \backslash x \to x < 2$

Folgende Ausdrücke sind daher äquivalent: `a/b` `(/) a b` `(a/)b` `(/b) a`
(/) ⇒ Klammern notwendig!

In Kap 1.2: `a → b → c` und `(a,b) → c` sind isomorph. Tatsächlich ist es möglich, die Funktionen ineinander zu übersetzen:

„Curryfizieren“:

```
curry :: ((a,b) → c) → (a → b → c)
curry f x y = f(x,y)
oder auch: curry f = \ x → \ y → f(x,y)
```

Umkehrung:

```
uncurry :: (a → b → c) → ((a,b) → c)
uncurry f (x,y) = f x y
```

Es gilt:

```
uncurry (curry f) = f
curry (uncurry g) = g
```

Allgemeiner formuliert:

```
uncurry.curry = id
curry.uncurry = id
```

wobei `id x = x` („ Identität“)
und `f.g = \x → f (g x)` („Komposition“)
Beispiel:

```
(+) :: Int → Int → Int
(uncurry (+)) :: (Int, Int) → Int
```

Gegeben: Liste von Zahlenpaaren, z.B. `[(2,3),(5,6)]`
Addieren aller Zahlen:

```
(sum.map  (uncurry (+)))  [(2,3), (5,6)]   ⇝ 16
          = \ (x,y) → x+y
wobei  sum [] = 0
       sum (x:xs) = x + sum xs
```

⇒ Summierung aller Elemente einer Zahlenliste

Multiplizieren aller Zahlen:

```
(prod.map (uncurry (*))) [(2,3),(5,6)]    ⇝ 180
wobei: prod [] = 1
       prod (x:xs) = x * prod xs
```

⇒ Multiplikation aller Listenelemente

Auffällig: `sum + prod` haben gleiches Schema ⇒ Verallgemeinerung durch ein einziges Schema:

`foldr ::` $\underbrace{(a \to b \to b)}_{\text{bin"are Operation}} \to \underbrace{b}_{\text{neutrales Element}} \to \underbrace{[a]}_{\text{Wertliste}} \to \underbrace{b}_{\text{Ergebnis}}$

```
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Damit werden `sum + prod` zu Spezialfällen:

```
sum  = foldr (+) 0
prod = foldr (x) 1
```

Einfache weitere Anwendung:

```
foldr (&&) True : Konjunktion boolescher Werte
foldr (++) []   : Konkatenation von Listen von Listen
```

Vorgehen bei (funktionaler) Programmierung:

1. Suche nach allgemeinen Programmschemata (häufig: Rekursionsschema bei rekursiven Datenstrukturen)
2. Realisiere Schema durch Funktion höherer Ordnung

`map, foldr`: Rekursionsschemata für Listen
Weiteres Schema: `filter :: (a → Bool) → [a] → [a]`
Filtere alle Elemente, die ein Prädikat erfüllen

```
filter p [] = []
filter p (x:xs) | p x       = x: filter p xs
                | otherwise = filter p xs
```

Anwendung: Umwandlung Liste → Menge (Entfernen aller Duplikate)

```
rmdups [] = []
rmdups (x:xs) = x: rmdups (filter (x /=) xs)
                                    /= → ungleich
```

Anwendung: Sortieren von Zahlenlisten mittels Quicksort:

```
qsort [] = []
qsort (x:xs) = qsort (filter (<= x) xs)
               ++ [x] ++ qsort (filter (> x) xs)
qsort [3,1,5,3] ⇝ [1,3,4,5]
```

Schema: Längster Präfix, dessen Elemente ein Prädikat erfüllen

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p [] = []
takeWhile p (x:xs) | p x       = x : takeWhile p xs
                   | otherwise = []
```

≈ **Kontrollstrukturen** (auch andere möglich)

Vorteil funktionaler Sprachen: Definition eigener Kontrollstrukturen durch Funktionen (Schemata)

Beispiel: until-do-Schleife: Besteht aus

- Abbruchbedingung (Prädikat `::a`→ Bool)
- Rumpf: Transformation auf Werte (Funktion `::a → a`)
- Anfangswert

Beispiel:

```
x:=1                        Anfangswert: 1
until x > 100               Abbruch: p x = x>100
do    x:= 2*x               Transformation: t x = 2*x
```

Implementierung in funktionaler Sprache:

```
until:: (a → Bool) → (a → a) → a → a
until  p f x | p x       = x
             | otherwise = until p f (f x)
```

wobei:
p = Abbruch
f = Transformation
x = Anfangswert

Obige Schleife: `until (>100) (2*) 1` ⇝ `128`
Beachte:

- keine Spracherweiterung notwendig!
- auch andere Kontrollstrukturen möglich
- gute Programmiersprache:
  - einfaches Grundkonzept
  - Techniken zur flexiblen Erweiterung
  - Bibliotheken
  - Negativbeispiel: PL/I (komplexe Sprache, enthält viele Konstrukte)

Anwendung: Newtonsches Verfahren zur Nullstellenberechnung
Gegeben: Funktion f, Schätzwert x für Nullstelle
Methode: $x - \frac{f(x)}{f'(x)}$ ist bessere Näherung
Quadratwurzel von a: Wende Verfahren auf $f(x) = x^2 - a$ an

Iterationsverfahren, Realisierung durch Schleife:

```
newton:: (Float → Float) →  Float  → Float
          \_____f_____/     \_Schätzwert_/
newton f = until ok improve
   where ok x = abs (f x) < 0.001
         improve x = x - f x / deriv f x
sqrt x  = newton f x   where f y = y*y-x
cubrt x = newton f x   where f y = y*y*y-x
sqrt 2.0   ⇝ 1.41421
cubrt 27.0 ⇝ 3.00001
```

wobei `sqrt x` = Quadratwurzel
und `cubrt x` = Mobilwurzel

Funktionen als Datenstrukturen

Was ist Datenstruktur?
Objekt mit Operationen zur

- Konstruktion (z.B. [], : bei Listen)
- Selektion (z.B. head, tail bei Listen)

(- Verknüpfung (z.B. ++ bei Listen))

Wichtig ist also nur die Funktionalität (Schnittstelle), nicht die Repräsentation ($\leadsto$ "abstrakter Datentyp")
Daher: Datenstruktur $\widehat{=}$ Satz von Funktionen

Beispiel: Felder mit Elementen vom Typ t:

Konstruktion: `emptyarray :: Feld t` (leeres Feld)
`putidx :: Feld t→ Int → t → Feld t`
`putidx a i v` $\widehat{=}$ neues Feld, wobei an Indexposition `i` der Wert `v` steht, alle anderen Positionen unverändert
(Beachte: `a` kann nicht verändert werden wegen Seiteneffektfreiheit!)
Selektion `getidx :: Feld t→ Int → t`
`getidx a i :` Element an Position `i` im Feld `a`

Mehr muß ein Anwender von Feldern nicht wissen!

Implementierung von Feldern:
Feld: Abbildung von Indizes im Werte, d.h. Feld `t` $\widehat{=}$ `Int → t`
wobei `Int` = Indizes
in Haskell: `type` Feld `t = int → t` (Typdeklaration, Typsynonym)

Dann ist Implementierung der Schnittstelle klar:

```
emptyarray i = error ''subscript out of range"
getidx a i = a I
puitdx a i v = a' ← neues Feld, d.h. neue Funktion
    where a' j | i==j      = v
               | otherwise = a j
```

Somit:

```
getidx (putidx emptyarray 2 'b') 2 ⇝ 'b'
               ''                1 ⇝ error...
```

Datenstrukturen ≈ Funktionen
Vorteil: konzeptuelle Klarheit (≈ Spezifikation)
Nachteil: Zugriffszeit: abhängig von Anzahl der `putidx`-Ops. (Verbesserung möglich, falls "altes" Feld nicht mehr benötigt wird.)

1.5 Typsystem und Typinferenz

Goofer, Haskell: "Streng getypt mit polymorphem Typsystem"
Streng getypte Sprache (auch Pascal (unvollständig wergen fehlender Typisierung von Prozedurparametern), Modula):

- Jedes Objekt (Wert, Funktion) hat einen *Typ*
- Typ ≈ Menge von möglichen Werten:

  `Int` ≈ Menge der ganzen Zahlen bzw. endliche Teilmenge,

  Bool ≈ {True, False}

  `Int → Int` ≈ Menge aller Funktionen, die ganze Zahlen auf ganze Zahlen abbilden

- "Objekt hat Typ" ≈ Objekt gehört zur Wertmenge des Typs
- *Typfehler:* Anwendung einer Funktion auf Werte, die nicht ihrem Argumenttyp entsprechen
  Beispiel: `3 + 'a'`
  Aufruf von + ohne Typprüfung der Argumente ⇒ unkontrolliertes Verhalten (Systemabsturz, Sicherheitslücken)
- Eigenschaft streng getypter Sprachen:
  Typfehler können nicht auftreten ("well-typed programs do not go wrong", Milner)
  Daher: `3 + 'a'` wird durch Compiler zurückgewiesen, ebenso `foldr (+) 0 [1, 3, 'a', 5]`

Schwach getypte Sprachen (Scheme, Smalltalk):

- Objekte haben Typ (werden zur Laufzeit annotiert)
- Funktionen prüfen zur Laufzeit, ob Argumente typkorrekt

Vorteile stark getypter Sprachen:

- Programmiersicherheit: Typfehler treten nicht auf
- Programmiereffizienz: Compiler meldet Typfehler, bevor Programm ausgeführt wird (Compiler prüft "Spezifikation"]
- Laufzeiteffizienz: Typprüfung zur Laufzeit unnötig
- Programmdokumentation: Typangaben können als (automatisch überprüfbare!) Teilspezifikation angesehen werden

Nachteile:

- Typsystem schränkt Flexibilität ein, um automatische Prüfung zu ermöglichen
- nicht jedes Programm ohne Laufzeittypfehler ist zulässig
  Beispiel: `takeWhile (<3) [1, 2, 3, 'a']`
  unzulässig, obwohl kein Laufzeittypfehler

Wichtiges Ziel bei Entwicklung von Typsystemen

- Sicherheit
- Flexibilität (möglichst wenig einschränken)
- Komfort (möglichst wenig explizit spezifizieren)

In funktionalen Sprachen erreicht durch

Polymorphismus: Objekte (Funktionen) können mehrere Typen haben

Typinferenz: nicht alle Typen müssen deklariert werden, sondern Typen von Variablen und Funktionen werden inferiert

1.5.1 Typpolymorphismus

Unterscheidung:

Ad-hoc Polymorphismus: Funktionen, die mehrere Typen haben, verhalten sich auf Typen unterschiedlich
Beispiel: Overloading, überladene Bezeichner: ein Bezeichner für unterschiedliche Funktionen
Java: "+" steht für

- Addition auf ganzen Zahlen
- Addition auf Gleitkommazahlen
- Stringkonkatenation

Parametrischer Polymorphismus Funktionen haben gleiches Verhalten auf allen ihren zulässigen Typen
Beispiel: Listenlänge:
`length [] = 0`
`length (x:xs) = 1+length xs`
- unabhängig vom Typ der Elemente
⇒ `length :: [a] → Int`
`a`=Typvariable, durch jeden anderen Typ ersetzbar
⇒ `length` anwendbar auf `[Int], [Float], [[Int]], [(Int, Float)], ...`
z. B. ist `length [0,1] + length ['a', 'b', 'c']` zulässig!
Aber: length arbeitet auf allen zulässigen Typen gleich
Beachte: Pascal oder C haben keinen parametrischen Polymorphismus
⇒ für jeden Listentyp muß eigene length-Funktion (mit identischer Struktur) definiert werden
Beispiel: Identität: `id x = x`
Allgemeinster Typ: `id:: a → a`
Also: `id` auf jedes Argument anwendbar, aber: Ergebnistyp = Argumenttyp
⇒ `id` $\underbrace{\underbrace{\texttt{[1]}}_{\texttt{[Int]}}}_{\texttt{[Int]}}$ == $\underbrace{\texttt{['a']}}_{\texttt{[Char]}}$ Typfehler

Typinferenz Herleitung von allgemeinsten Typen, so daß Programm noch *typkorrekt* ist

Vorteil:

1. Viele Programmierfehler werden als Typfehler entdeckt
   Beispiel: Vertauschung von Argumenten:
   `foldr 0 (+) [1, 3, 5]` → Typ error ...

2. Auch, falls kein Typfehler auftritt, kann allgemeiner Typ auf Fehler hinweisen:

```
revf [] = []
revf (x:xs) = revf xs ++ x   -- (statt [x])
```

Allgemeinster Typ: `[[a]] → [a]`
Allerdings ist der Argumenttyp `[[a]]` nicht beabsichtigt!

Was bedeutet *typkorrekt*?
Keine allgemeine Definition, sondern durch Sprache festgelegt. Forderung: „Typkorrekte“ Programme haben keine Laufzeittypfehler
Im folgenden: Typkorrektheit á la Hindley/Milner (beachte: Haskell basiert auf Verallgemeinerung mit „Typklassen“)
Typausdrücke τ werden gebildet aus:

1. *Typvariablen* (a, b, c, ...)
2. *Basistypen* (Bool, Int, Float, Char, ...)
3. *Typkonstruktoren* (Funktionen auf Typen, bilden neue Typen aus gegebenen:) $\cdot \to \cdot$, `[·]`, `Tree` $\cdot$
   Bsp.: τ = `Int → [Tree Int]`

monomorpher Typ: enthält keine Typvariablen
polymorpher Typ: enthält Typvariablen

(Typ-)**Substitution:** Ersetzung von Typvariablen durch Typausdrücke

Notation: $\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$
bezeichnet Abbildung Typvariablen → Typausdrücke mit $\sigma(a) = \begin{cases} \tau_i & \text{falls } a = a_i \\ a & \text{sonst} \end{cases}$
Fortsetzung auf Ausdrücken:
σ (b) = b $\forall$ Basistypen b
σ (k($\tau_1, \ldots, \tau_n$)) = k($\sigma(\tau_n)$, ..., $\sigma(\tau_n)$)
$\forall$ n-stelligen Typkonstruktoren und Typen $\tau_1, \ldots, \tau_n$
Bsp.: Falls $\sigma = \{\texttt{a} \mapsto \texttt{Int}, \texttt{b} \mapsto \texttt{Bool}\}$, dann ist $\sigma(\texttt{a} \to \texttt{b} \to \texttt{a}) = \texttt{Int} \to \texttt{Bool} \to \texttt{Int}$

Falls Funktion (polymorphen) Typ $\tau_1 \to \tau_2$ hat, dann hat sie auch jeden speziellen Typ $\underbrace{\sigma(\tau_1 \to \tau_2)}_{\text{„}Typinstanz\text{“}}$
$\Longrightarrow$ Polymorphismus von Funktionen

Beispiel:
`length :: [a] → Int`
hat auch die Typen
`[Int] → Int` ($\sigma = \{\texttt{a} \mapsto \texttt{Int}\}$)
`[Char] → Int` ($\sigma = \{\texttt{a} \mapsto \texttt{Char}\}$)

Einschränkung: keine Instanzbildung bei Definition derselben Funktion
Beispiel:
`f:: a → a`
`f =` $\underbrace{\texttt{f}}_{(b\to b)\to(b\to b)}$ $\underbrace{\texttt{(f)}}_{b\to b}$ unzulässig! Ursache: Typinferenz sonst unentscheidbar

Daher: Unterscheide Funktion mit und ohne Instanzbildung

$$\underset{\text{Typschema}}{} \quad \underset{\text{Typvariablen}}{\forall a_1 \quad \ldots \quad a_n} : \quad \underset{\text{Typausdruck}}{t} \qquad \text{(Beachte: n=0} \Rightarrow \text{Typausdruck)}$$

generische Instanz des Typschemas $\forall a_1 \ldots a_n : \tau : \sigma(t)$ wobei $\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$
Vordefinierte Funktionen (alle Funktionen, deren Definition festliegt) haben Typschema,
zu definierende Funktionen haben Typausdrücke

Vorgehen bei Typprüfung

1. Nehme für zu definiernde Funktionen und deren Parameter Typausdrücke an und prüfe (s.u.) alle Regeln für diese.
2. Bei Erfolg: Fasse diese Funktionen von nun an als vordefiniert auf, d.h. abstrahiere die Typvariablen zu einem Typschema.

Voraussetzung: Parameter in verschiedenen Regeln haben verschiedene Namen ($\rightarrow$ Umbenennung)
Typannahme: Zuordnung Namen $\rightarrow$ Typschemata
Typprüfung: Beweisen von Aussagen der Form A $\vdash$ e :: τ „Unter der Typannahme A hat Ausdruck e den Typ τ“
Beweis durch Inferenzsystem (Lese $\frac{P_1 \ldots P_n}{Q}$: falls $P_1 \ldots P_n$ richtig ist, dann auch Q)

Axiom $\quad \dfrac{}{A \vdash x :: \tau}$ falls τ generische Instanz von $A(x)$

Applikation $\quad \dfrac{A \vdash e_1 :: \tau_1 \rightarrow \tau_2, A \vdash e_2 :: \tau_1}{A \vdash e_1 e_2 :: \tau_2}$

Abstraktion $\quad \dfrac{A[x \mapsto \tau] \vdash e :: \tau'}{A \vdash \backslash x \rightarrow e :: \tau \rightarrow \tau'}$ wobei τ Typausdruck

$$A[x \mapsto \tau](y) = \begin{cases} \tau & \text{, falls } y = x \\ A(y) & \text{, sonst} \end{cases}$$

Bedingung $\quad \dfrac{A \vdash e_1 :: \texttt{Bool}, A \vdash e_2 :: \tau, A \vdash e_3 :: \tau}{A \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 :: \tau}$

Gleichung $l = r$ typkorrekt bzgl. $A :\Leftrightarrow A \vdash l :: \tau$ und $A \vdash r :: \tau$ für Typausdruck τ
Falls alle Gleichungen für alle zu definierenden Funktionen aus A typkorrekt sind, abstrahiere deren Typen zu Typschemata (durch Quantifizierung der Typvariablen)

Bsp.: Vordefiniert:

$$\begin{aligned} A(+) &= \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int} \\ A([]) &= \forall a.[a] \\ A(:) &= \forall a.a \rightarrow [a] \rightarrow [a] \\ A(not) &= \texttt{Bool} \rightarrow \texttt{Bool} \end{aligned}$$

Definition der Funktion twice: $twice\ f\ x = f(f\ x)$

$$\begin{array}{rlcl} \text{Annahme:} & A(f) & = & a \to a \\ & A(x) & = & a \\ & A(\texttt{twice}) & = & (a \to a) \to a \to a \end{array}$$

Typkorrektheit:

$$\dfrac{\dfrac{\dfrac{}{A \vdash \texttt{twice} :: (a \to a) \to a \to a} \quad \dfrac{}{A \vdash f :: a \to a}}{A \vdash \texttt{twice} f :: a \to a} \quad \dfrac{}{A \vdash x : a}}{A \vdash \texttt{twice} f x :: a}$$

$$\dfrac{\dfrac{}{A \vdash f : a \to a} \quad \dfrac{\dfrac{}{A \vdash f :: a \to a} \quad \dfrac{}{A \vdash x :: a}}{A \vdash (fx) :: a}}{A \vdash f(fx) :: a}$$

$\Longrightarrow$ Gleichung typkorrekt

Abstrahiere: `twice`:: $\forall a : (a \to a) \to a \to a$
Nun polymorphe Anwendung möglich:

```
twice (+2) 3
twice not True
```

τ heißt *allgemeinster Typ* eines Objektes (Funktion), falls

- τ korrekter Typ
- Ist τ korrekter Typ für dasselbe Objekt, dann $\tau' = \sigma(\tau)$ für eine Typsubstitution σ

Bsp.: `loop 0 = loop 0`
Allgemeinster Typ: `loop:: int` $\to$ a
`0:: Int`
Typkorrekt: $\dfrac{\dfrac{}{A \vdash \ \texttt{loop} :: \texttt{Int} \to a} \quad \dfrac{}{A \vdash 0 :: \texttt{Int}}}{A \vdash \texttt{loop}\ 0 :: a}$
$\Rightarrow$ Beide Gleichungsseiten haben Typ a

Der Typ `Int`$\to a$ deutet auf Programmierfehler hin, da die Funktion beliebige Ergebnisse erzeugen kann, was nicht möglich ist.

Typprüfung für mehrere Funktionen:

```
length [] = 0
length (x:xs) = 1 + length xs
f xs ys = length xs + length ys
```

Vordefiniert:

```
[] :: ∀ a. [a]
(:):: ∀ a. a→ [a] → [a]
0,1:: Int
```

```
(+)  :: Int → Int → Int
```

Annahme:

```
length :: [a] → Int
  x    :: a
  xs   :: [a]
```

Damit sind Regeln für `length` wohlgetypt
Nun vordefiniert: `length:: ∀a. [a] → Int`
Annahme:

```
f  :: [a] → [b] → Int
xs :: [a]
ys :: [b]
```

In den folgenden Ableitungen lassen wir „$A \vdash$“ weg:

Mit dieser Typannahme erhalten wir folgende Ableitung für den Typ der linken Seite der Definition von `f`:

$$\dfrac{\dfrac{\dfrac{}{\texttt{f :: [a] → [b] → Int}} \quad \dfrac{}{\texttt{xs :: [a]}}}{\texttt{f xs :: [b] → Int}} \quad \dfrac{}{\texttt{ys :: [b]}}}{\texttt{f xs ys :: Int}}$$

In ähnlicher Weise erhalten wir eine Ableitung für den Typ der rechten Seite der Definition von `f` (beachte, dass wir hier generische Instanzen des Typschemas von `length` verwenden):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{length :: [a] → Int}} \quad \dfrac{}{\texttt{xs :: [a]}}}{\texttt{length xs :: Int}} \quad \dfrac{}{\texttt{+ :: Int → Int → Int}}}{\texttt{length xs + :: Int → Int}} \quad \dfrac{\dfrac{}{\texttt{length :: [b] → Int}} \quad \dfrac{}{\texttt{ys :: [b]}}}{\texttt{length ys :: Int}}}{\texttt{length xs + length ys :: Int}}$$

Somit erhalten wir das folgende Typschema für `f`:

```
f :: ∀a,b : [a] → [b] → Int
```

Daher ist

```
f [0,1] "Hallo"                                                    (*)
```

wohlgetypt (mit der generischen Instanz $\{\texttt{a} \mapsto \texttt{Int}, \texttt{b} \mapsto \texttt{Char}\}$)

Ohne Typschemabildung für `length` würden wir für `f` den Typ `[a] → [a] → Int` erhalten, der zu speziell ist, d.h. mit dem der Ausdruck `(*)` nicht typkorrekt wäre.

Aus diesem Verfahren ergeben sich unmittelbar die folgenden Fragen zur praktischen Anwendung:

- Wie findet man allgemeinste Typen?
- Wie muß man Typannahmen raten?

Diese Fragen sollen im nächsten Kapitel beantwortet werden.

1.5.2 Typinferenz

Aufgabe: Suche bzgl. vordefinierter Funktionen allgemeinste Typen für neu definierte Funktion.
Grundidee (nach Damas/Milner, POPL'82):
Statt Typ raten:

- Setze für unbekannten Typ zunächst Typvarable ein
- Formuliere Gleichungen (Bedingungen) zwischen Typen
- Berechne allgemeinste Lösung für das Gleichungssystem

zunächst: Typinferenz für eine neue Funktion.
Vorgehen:

1. **Benenne Variablen** in verschiedenen Gleichungen so **um**, dass verschiedene Gleichungen keine identischen Varablen enthalten:

   ```
   app [] x = x                    ⇝ app [] z = z
   app (x:xs) y = x : app xs y
   ```

2. **Erzeuge Ausdruck/Typ-Paare:**
   Für jede Regel $l(|c) = r$
   erzeuge
   $l :: a$
   $r :: a$
   (c::`Bool`)
   wobei a eine *neue* Typvariable ist

3. **Vereinfache Ausdruck/Typ-Paare:** Ersetze

   - $e_1\ e_2 :: \tau$ durch $e_1 :: a \to \tau,\ e_2 :: a$ (hierbei ist a eine neue Typvariable)
   - `if` e_1 `then` e_2 `else` $e_3 :: \tau$ durch $e_1 ::$ `Bool`$, e_2 :: \tau, e_3 :: \tau$
   - $e_1 \circ e_2 :: \tau$ durch $e_1 :: a, e :: b, o :: a \to b \to \tau$ (a, b neue Typvariablen)
     (hierbei ist $\circ$ ein vordefinierter Infixoperater)
   - $\backslash x \to e :: \tau$ durch $x :: a, e :: b, \underbrace{\tau \doteq a \to b}_{Typgleichung}$ (a, b neue Typvariablen)
   - $f :: \tau$ durch $\tau \doteq \sigma(\tau')$
     falls f vordefiniert mit Typschema $\forall a_1 \ldots a_n : \tau'$ und $\sigma = \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ wobei $b_1, \ldots b_n$ neue Typvariablen

4. **Erzeuge Gleichungen:**
   Vor.: alle Ausdruck/Typ-Paare von der Form $x :: \tau, x$ Bezeichner
   Dann: setze alle Typen für einen Bezeichner gleich, d.h. erzeuge Gleichung $\tau_1 = \tau_2$ für alle

$x : \tau_1, x : \tau_2$

Beispiel :	`twice`$fx = f$	(fx)	
$\Rightarrow$	$\underbrace{\texttt{twice}\ fx :: a}$	$\underbrace{f(fx) :: a}$	
	$\underbrace{\texttt{twice}\ f :: b \to a,}\, x :: b$	$f :: d \to a,$	$\underbrace{fx :: d}$
	`twice` $:: c \to b \to a, f : c$		$f :: e \to d, x :: d$
$\Rightarrow$	*Gleichungen* :	$c \doteq d \to a$	
		$c \doteq e \to d$	
		$b \doteq d$	

5. Löse Gleichungssystem, d.h. finde „allgemeinsten Unifikator“ σ. Dann ist, für alle Ausdruck/Typ-Paare $x :: \tau$, $\sigma(\tau)$ der allgemeinste Typ für x. Falls kein allg. Unifikator existiert, dann sind die Regeln nicht typkorrekt.

   $\Rightarrow$ Für typkorrekte Regeln existiert immer ein allgemeinster Typ! (Nicht trivial, z.B. falsch bei Overloading)

Definition:

- Substitution σ heißt *Unifikator* für ein Gleichungssystem $E \Leftrightarrow \forall l \doteq r \epsilon E$ ist $\sigma(l) = \sigma(r)$
- Substitution σ heißt *allgemeinster Unifikator (mgu, most general unifier)* für Gleichungssystem E
  $:\Leftrightarrow \forall$ Unifikatoren σ' für $E \exists$ Substitution φ mit $\sigma' = \varphi \circ \sigma$
  (wobei $\varphi \circ \sigma(\tau) = \varphi(\sigma(\tau))$, d.h. alle anderen Unifikatoren sind Spezialfälle von σ)

Satz(Robinson, 1965) Falls Unifikator existiert, existiert auch ein mgu, der effektiv berechenbar ist.

Hier: mgu-Berechnung durch Transformation von E (Martelli/Montanari, ACM TOPLAS 4(2), 1982
Wende folgende Transformationsregeln auf E an:

Decomposition: $\dfrac{\{k\ s_1 \dots s_n \doteq k\ t_1 \dots t_n\} \cup E}{\{s_1 \doteq T_1, \dots, s_n \doteq t_n\} \cup E}$ k Typkonstruktor

Clash: $\dfrac{\{k\ s_1 \dots s_n \doteq k'\ t_1 \dots t_m\} \cup E}{\texttt{fail}}$ $k \neq k$ Typkonstruktoren

Elimination: $\dfrac{\{x \doteq x\} \cup E}{E}$ x Typvariable

Swap: $\dfrac{\{k\ t_1 \dots t_n \doteq x\} \cup E}{\{x \doteq k\ t_1 \dots t_n\} \cup E}$ x Typvariable

Replace: $\dfrac{\{x \doteq \tau\} \cup E}{\{x \doteq \tau\} \cup \sigma(E)}$ x Typv., kommt in E aber nicht in τ vor, $\sigma = [x/\tau]$

Occur check: $\dfrac{\{x \doteq \tau\} \cup E}{\texttt{fail}}$ x Typvariable $x \neq \tau$, kommt in τ vor

Satz: Falls $E \rightsquigarrow \{x_1 \doteq \tau_1, \dots, x_n \doteq \tau_n\}$ und keine Regel anwendbar, dann ist $\sigma = \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\}$ ein mgu für E. Falls $E \rightsquigarrow \texttt{fail}$, dann existiert kein Unifikator für E.

Beispiel:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\boxed{c \doteq d \to a},\ c \doteq e \to d,\ b \doteq e}{c \doteq d \to a,\ \boxed{d \to a \doteq e \to d},\ b \doteq e}\ \text{Replace}}{c \doteq d \to a,\ \boxed{d \doteq e},\ a \doteq d,\ b \doteq e}\ \text{Decomposition}}{c \doteq e \to a,\ d \doteq e,\ \boxed{a \doteq e},\ b \doteq e}\ \text{Replace}}{c \doteq e \to e,\ d \doteq e,\ a \doteq e,\ b \doteq e}\ \text{Replace}$$

$\Rightarrow$ `twice` $:: (e \to e) \to e \to e$ ist allgemeinster Typ
Da `twice` typkorrekt ist, hat `twice` Typschema $\forall e.(e \to e) \to e \to e$

Typinferenz für beliebige rekursive Funktionen

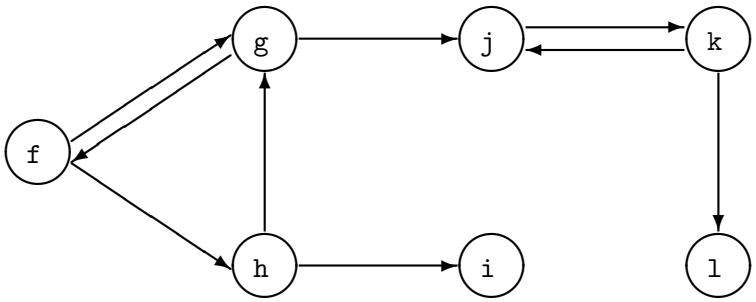
Idee:

1. Sortiere Funktionen nach Abhängigkeiten
2. Inferiere Basisfunktionen. Falls typkorrekt $\Rightarrow$ Umwandlung zu Typschema
3. Inferiere darauf aufbauende Funktionen $\Rightarrow$ Umwandlung zu Typschema
4. ...

Präsisierung durch *statischen Aufrufgraph:*
Knoten: Funktionen
Kante $f \to g :\Leftrightarrow g$ wird in Definition von f verwendet.
Beispiel:

```
f n  =  g n + h n
g n  =  j n + f n
h n  =  i n + g n
i n  =  n
j n  =  k n
k n  =  j n + l n
l n  =  5
```

Statischer Aufrufgraph für dieses Beispiel:



Relation „gegenseitig abhängig“: $f \leftrightarrow g :\Leftrightarrow$ es existiert Pfad von f nach g und Pfad von g nach f

Feststellung: $\leftrightarrow$ ist eine Äquivalenzrelation

Daher können wir die Äquivalenzklassen bzgl. $\leftrightarrow$ bilden (diese werden auch *starke Zusammenhangskomponenten* genannt; im obigen Beispiel sind dies z.B. die Knotenmengen $\{\texttt{f}, \texttt{gh}\}$ und $\{\texttt{j}, \texttt{k}\}$)

Sortiere Äquivalenzklassen in Liste $[A_1, A_2, \ldots, A_n]$
wobei: wenn Pfad von A_i-Knoten nach A_j-Knoten ($i \neq j$) existiert, dann ist $i \geq j$ (A_i definiert mit Hilfe von A_j)
Beispiel: $[\{\texttt{i}\}, \{\texttt{l}\}, \{\texttt{j}, \texttt{k}\}, \{\texttt{f}, \texttt{gh}\}]$
Nun: Für $i = 1, \ldots, n$ tue folgendes:

1. Berechne allgemeinsten Typ für alle Funktionen in A_i
2. Abstrahiere diese Typen durch Allquantifizierung aller vorkommenden Typvariablen

Beispiel:

```
i     :: ∀a : a → a
l     :: ∀a : a → Int
```

```
j,k   :: ∀a : a → Int
f,g,h :: Int → Int
```

Beispiel: nicht typisierbar ist `f x = x x` ("`x x`" wird auch als *Selbstanwendung* bezeichnet)

Inferenz ergibt nach der Vereinfachung: `f::`$b \to a$`, x::`b`, x::`$c \to a$`, x::`c

Gleichungssystem aufstellen und mgu berechnen:

$$\dfrac{\dfrac{b \doteq c \to a, b \doteq c}{c \doteq c \to a, b \doteq c}}{\texttt{fail}}$$

Somit ist dieses Gleichungsssytem nicht lösbar.

Nicht typisierbar sind aber auch sinnvollere Funktionen:

```
funsum f l1 l2 = f l1 + f l2
```

Mit dieser Definition führt der Ausdruck

```
funsum length [1,2,3] "abc"
```

zu einem **Typfehler**!
(bei einer ungetypten Sprache würde das Ergebnis `6` herauskommen)

Die Ursache liegt im inferierten Typ von `funsum` :

$$\forall a : (a \to \texttt{Int}) \to a \to a \to \texttt{Int}$$

Bei Aufruf von `funsum` muss man eine generische Instanz festlegtn, z.B. $a =$ `[Int]`$], b =$ `Int`, was aber zu einem Typfehler bei `"abc"` führt.

Ein geeigneter Typ wäre:

`funsum` $\forall b, c :\ (\forall a : a \ \to\ \texttt{Int}) \ \to\ b \ \to\ c \ \to\ \texttt{Int}$

d.h. der erste Parameter ist eine polymorph verwendbare Funktion.
Solche Typen sind prinzipiell denkbar ($\rightsquigarrow$ Polymorphismus 2. Ordnung), aber in dem hier vorgestellten Typsystem gilt:

Parameter sind nicht mehrfach typinstantiierbar

Praktisch ist dies aber kein Problem, da mehrfache Aufrufe wegtransformiert werden können:

```
funsum :: (a -> Int) -> (b -> Int) -> a -> b -> Int
funsum  f1 f2 l1 l2 = f1 l1 + f2 l2
```

Mit dieser Definition ist der Ausdruck

```
funsum length length [1,2,3] "abc"
```

typisierbar und ergibt `6`

1.6 Lazy Evaluation

Redex (*re*ducible *ex*pression): Ausdruck (Funktionsaufruf), auf den die linke Seite einer Gleichung paßt ($\leadsto$ 1.3 Pattern Matching)

Normalform: Ausdruck ohne Redex

Reduktionsschnitt: Ersetze Redex durch entsprechende rechte Gleichungsseite

Ziel einer Auswertung: Berechne Normalform durch Anwendung von Reduktionsschritten
Problem: Ausdruck kann viele Redexe enthalten: Welche Redexe soll man reduzieren?
„Sichere“ Strategie: Reduziere alle Redexe $\leadsto$ zu aufwendig
Strikte Sprachen: Reduziere linken inneren Redex (*LI-Reduktion*)
Nichtstrikte Sprachen: Reduziere linken äußeren Redex (*LO-Reduktion*
$\rightarrow$ Nachteil: Mehrfachauswertung von Argumenten

Beispiel:
`double` $x = x + s$
`double` $(1 + 2) \rightarrow (1 + 2) + (1 + 2) \rightarrow 3 + (1 + 2) \rightarrow 3 + 3 \rightarrow 6$

Vermeidung durch *Graphrepräsentation:* Mehrfache Vorkommen von Variablen duplizieren keine Terme, sondern sind Verweise auf einen Ausdruck:

`double` $(1 + 2) \rightarrow \;\; \cdot + \cdot \;\; \rightarrow \;\; \cdot + \cdot \;\; \rightarrow \;\; 6$
$\downarrow \;\; \downarrow \qquad\quad \downarrow \;\; \downarrow$
$(1 + 2) \qquad\quad 3$

LO-Reduktion auf Graphen: *lazy evaluation*
call-by-need- Reduktion

Charakteristik:

1. Berechne Redex nur, falls er „benötigt“ wird
2. Berechne jeden Redex höchstens einmal

„benötigt“:

1. Durch Muster: $fx : xs = \ldots$
   Aufruf $f \qquad \underbrace{(g \ldots)}$ Genauer: Übersetzung Muster $\rightarrow$ case-Ausdrücke:
   Wert (Konstrukter) wird benötigt
   Auswertung von `case` t `of`: Wert von t wird benötigt
2. Durch strikte Grundoperationen: Arithmetische Operationen und Vergleiche benötigen Wert ihrer Argumente
   Dagegen: `if` e_1 `then` $e2$ `else` e_3 benötigt nur Wert von e_1
3. Ausgabe: Drucken der benötigten Teile

Bei 1: Benötigten Redex nicht komplett anwenden, sondern nur bis zur *schwachen Kopfnormalform* (SKNF) *weak head normal form,* WHNF), das ist:

$b \epsilon$ `Int` $\cup$ `Float` $\cup$ `Bool` $\cup$ `Char`	
$Ce_1 \ldots e_k$:	C $n-$ stelliger Konstruktor, $k \leq n$
$(e_1, \ldots, e_k)$	
$\backslash x \rightarrow e$	
$fe_1 \ldots e_k$	f $n-$ stellige Funktion, $k \leq n$

SKNF-Ausdruck: kein Redex, d.h. nicht weiter reduzierbar (an der Wurzel)
Wichtig: Beim Pattern matching ($\simeq$ case-Ausdrücke) nur Reduktion bis SKNF erreicht
Bsp: `take` n xs : Erste n Elemente von xs

```
take 0 xs = []
take (n+1)(x:xs)= x : take n xs
```

```
take 1 ([1,2] ++[3,4])
          benötigt
→ take 1 (1:([2] ++ [3,4]))
              Redex
→ 1 : take 0 ([2] ++ [3,4])
              Redex
→ 1 : [] (≃ [1])
```

Beachte: 2. Argument wurde nicht komplett ausgewertet ($\rightarrow$ „lazy“ evaluation)
Lazy evaluation hat Vorteile bei sog. nicht-strikten Funktionen (Vermeidung von überflüssigen Berechnungen)
n-stellige Funktion f heißt *strikt im i-ten Argument* ($1 \geq i \geq n$), falls $\forall x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ gilt: $fx_1, \ldots, x_{i-1}, \perp x_{i-1} \ldots x_n = \perp$
Hierbei bezeichnet $\perp$ eine nichtterminierende Berechnung (undefinierter Wert)
`if-then-else`:: `Bool`$\rightarrow a \rightarrow a \rightarrow a$: Strikt im 1. Argument, aber nicht im 2. und 3.

Vorteile von lazy evaluation:

- Vermeidung überflüssiger Auswertungen
  ($\rightsquigarrow$ *optimale Reduktionen*, Huet/Levy '79)
- Rechnen mit unendlichen Datenstrukturen
  ($\rightsquigarrow$ Trennung von Daten und Kontrolle $\rightarrow$ Modularisierung)

Unendliche Datenstrukturen

Erzeugung einer (endlichen) Liste von ganzen Zahlen:

```
fromTo f t | f> t      = []
           | otherwise = x : fromTo (f+1) t
fromTo 1 5  ⇝  [1, 2, 3, 4, 5]
```

Falls „$t = \infty$“: unendliche Liste, weil $f > t$ nie wahr. Spezialisierung von `fromTo` für $t = \infty$:

```
from f = x : from (f+1)
from 1  ⇝  [1, 2, 3, 4, ...] ^C
```

Obwohl `from` 1 unendliches Objekt darstellt, kann man es wie andere Listen verwenden:
`take` 5 (`from` 1) $\rightsquigarrow [1,2,3,4,5]$

Somit ist `fromTo` Spezialfall:
`fromTo` $ft = \underbrace{\texttt{take } (t-f+1)}_{\uparrow}\underbrace{(\texttt{ from } f)}_{\uparrow}$
Vorteil:
Trennung von Kontrolle und Datenerzeugung
Vermischt im ersten `fromTo`
Häufig: unabhängige Datenerzeugung einfacher definierbar
Auswertung unendlicher Datenstrukturen erst durch lazy evaluation möglich:
`take` 2 $\underbrace{(\texttt{from } 1)}_{Lazy:benötigt}$ Eager evaluation: Auswertung von `from` 1: Endlosschleife
$\rightarrow$ `take` 2 $\underbrace{(1 : \texttt{from } 2)}_{in SKNF}$ ($\Rightarrow 2 =$ eigentlich: $1+1$)
$\rightarrow 1 :$ `take` 1 $\underbrace{(\texttt{from } 2)}_{benötigt}$
$\rightarrow 1 :$ `take` $1(2 :$ `from` $3)$
$\rightarrow 1 : 2 :$ `take` $0($`from` $3)$
$\rightarrow 1 : 2 : []$

Bsp.: Erzeugung aller *Fibonacci-Zahlen*
n. Fib. Zahl: Summer der beiden vorigen
$\Rightarrow$ Im Gegensatz zu `from`: Zwei Parameter (vorige Fib. zahlen)

```
fibgen n1 n2 = n1 : fibgen n2 (n1 + n2)
fibs = fibgen 0 1
```

$\Rightarrow$ keine exponentielle Laufzeit!
`fibs` $\rightsquigarrow 1, 1, 2, 3, 5, \ldots$
Fibonacci-Zahlen < 50 :

```
takeWhile (<50) fibs  ⇝ [1, 1, 2, 3, 5, 7, 13, 21, 34]
```

Achtung: Filtern hier nicht möglich:

```
filter (<10) fibs   ⇝ [0, 1, 2, 3, 5, 8,...   Endlosschleife!
```

Beispiel: Primzahlberechnung durch *Sieb des Eratosthemes*:
Idee:

1. Erstelle Liste aller natürlicher Zahlen ≥ 2
2. Streiche darin alle Vielfachen bereits gefundenen Primzahlen
3. Nächste kleinste Zahl ist prim

Beispiel:
$\underline{2}, \underline{3}, \not{4}, \underline{5}, \not{6}, \underline{7}, \not{8}, \not{9}, \not{10}, \underline{11}, \not{12}, \underline{13}, \ldots$

```
sieve :: [Int] -> [Int]
```

Hierbei gelten die folgende Invarianten für `sieve`:

1. Das erste Element x der Eingabeliste ist prim und die restliche Eingabeliste enthält keine Vielfachen von Primzahlen $< x$
2. Die Ergebnisliste besteht nur aus Primzahlen

Mit diesen Überlegungen kann `sieve` wie folgt implementiert werden:

```
sieve (x:xs) = x : sieve (filter (\ y -> y `mod` x > 0) xs)
```

Hierbei liefert das Prädikat in `filter` den Wert `False`, falls `y` ein Vielfaches von `x` ist. Insgesamt erhalten wir unter Benutzung von `sieve` die folgende Definition für die Liste aller Primzahlen:

```
primes = sieve (from 2)
```

Erste 100. Primzahlen: `take 100 primes`

Bsp.: **Hamming-Zahlen:** Aufsteigende Folge von Zahlen der Form $2^a * 3^b * 5^c, a, b, c, \geq 0$
d.h. alle Zahlen mit Primfaktoren $2, 3, 5$

Idee:

1. 1 ist kleinste Hamming-Zahl
2. Falls n Hamming-Zahl, dann ist auch $2n$, $3n$, $5n$ eine Hamming-Zahl
3. Wähle aus den nächsten möglichen Hamming-Zahlen kleinste aus und gehe nach 2.

Realisierung Erzeuge unendliche Liste der Hamming-Zahlen durch Multiplikation der Zahlen mit $2, 3, 5$, und Mischen der multiplizierten Listen

Mischen zweier unendlicher Listen in aufsteigender Folge:

```
ordMerge (x:xs) (y:ys)  |  x==y   =  x : ordMerge xs     ys
                        |  x < y  =  x : ordMerge xs     (y:ys)
                        |  x > y  =  y : ordMerge (x:xs) ys
```

Mischen dreier Listen "`l1`", `l2`, `l3` durch `ordMerge l1 (ordMerge l2 l3)`
Somit:

```
hamming = 1 : ordMerge (map (2*) hamming)
                       (ordMerge (map (*3) hamming)
                                 (map (*5) hamming))
```

Effizienz: `ordMerge + map` konstanter Aufwand für nächstes Element $\Rightarrow 0(n)$ für n. Element
Erste 15 Hammingzahlen:

```
take 15 hamming  ⇝  [1,2,3,4,5,6,8,9,10,12,15,16,18,20,24]
```

1.7 Deklarative Ein/Ausgabe

To do: Einfügen

1.8 Zusammenfassung

Vorteile funktionaler Sprachen:

- einfaches Modell: Ersetzung von Ausdrücken
- überschaubare Programmstruktur:
  Funktionen, definiert durch Gleichungen für Spezialfälle (pattern matching)
- Ausdrucksmächtig: Programmschemata mittels Funktionen höherer Ordnung
- Sicherheit: keine Laufzeittypfehler
- Wiederverwendung: Polymorphismus + Funktion höherer Ordnung
- Verifikation und Wartbarkeit:
  keine Seiteneffekte: erleichtert Verifikation und Transformation

Kapitel 2

Grundlagen der funktionalen Programmierung

Ziel dieses Kapitels: Formalisierung wichtiger Begriffe der funktionalen Programmierung
Hierbei sind drei Bereiche relevant:

- Reduktionssysteme:
  formalisieren Reduktionsfolgen, Normalformen, Terminierung...
- Termersetzung:
  formalisiert Rechnen mit Pattern matching und entsprechende Strategien

2.1 Reduktionssysteme

Rechnen in funktionalen Sprachen: Anwenden von Reduktionsschritten
Hier: *allgemeine (abstrakte) Reduktionssysteme*
(Begriffe werden bei λ-Kalkül, Termersetzung und Logikprogrammierung wiederverwendet.)

Definition:
M Menge, $\rightarrow$ zweistellige Relation auf M
(d.h. $\rightarrow \subseteq M \times M$, Notation: $e_1 \rightarrow e_2 :\Leftrightarrow (e_1, e_2) \subseteq \rightarrow$).
Dann heißt $(M, \rightarrow)$ *Reduktionssystem*
Erweiterung von $\rightarrow$:

$$\begin{array}{lll} x \rightarrow^0 y & :\Leftrightarrow & x = y \\ x \rightarrow^i y & :\Leftrightarrow & \exists z \quad x \rightarrow^{i-1} z \text{ und } z \rightarrow y \end{array} \Rightarrow \text{i-faches Produkt von } \rightarrow$$

$$x \rightarrow^+ y \quad :\Leftrightarrow \quad \exists i > 0 : x \rightarrow^i y \text{(transitiver Abschluß)}$$
$$x \rightarrow^* y \quad :\Leftrightarrow \quad \exists i \geq 0 : x \rightarrow^i y \text{(reflexiv-transitiver Abschluß)}$$

Abgeleitete Relationen:

$$x \leftarrow y \quad :\Leftrightarrow \quad y \rightarrow x$$
$$x \leftrightarrow y \quad :\Leftrightarrow \quad x \rightarrow y \text{ oder } x \leftarrow y$$

Insbesondere $x \leftrightarrow^* y$: symmetrischer, reflexiv-transitiver Abschluß, kleinste Äquivalenzsrelation, die $\rightarrow$ enthält.

Def.: Sei $(M, \to)$ festes Reduktionssystem
$x \in M$ *reduzierbar* $:\Leftrightarrow \exists y$ mit $x \to y$
$x \in M$ *irreduzibel* oder *in Normalform*, falls x nicht reduzierbar
y *Normalform von* x $:\Leftrightarrow$ $x \to^* y$ und y irreduzibel
(falls x nur eine Normalform y hat: $x\!\downarrow := y$)
$x \downarrow y$ (x und y sind *zusammenführbar*) $:\Leftrightarrow \exists z$ mit $x \to^* z$ und $y \to^* z$

Wichtig: Funktionen sollen „sinnvoll“ definiert sein
$\Rightarrow$ es soll höchstens eine Normalform existieren.
Hierzu muß das Reduktionssystem weitere Forderungen erfüllen:

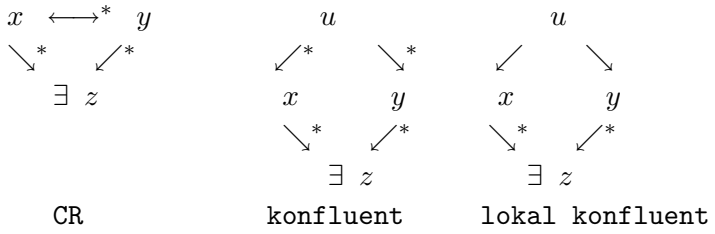
Def.: Sei $(M, \to)$ Reduktionssystem

1. $\to$ heißt *Church-Rosser (CR)*, falls $\leftrightarrow^* \subseteq \downarrow$
(d.h. $x \leftrightarrow^* y$ impliziert $x \downarrow y$)

2. $\to$ heißt *konfluent*, falls $\forall u, x, y$ mit $u \to^* x$ und $u \to^* y$
ein z existiert mit $x \to z$ und $y \to z$

3. $\to$ heißt *lokal konfluent*, falls $\forall u, x, y$ mit $u \to x$ und $u \to y$
ein z existiert mit $x \to^* z$ und $y \to^* z$

Graphisch:



CR: Äquivalenz von Ausdrücken entscheiden durch gerichtete Reduktionen
Konfluenz: Unterschiedliche Berechnungswege spielen keine Rolle
Lokale Konfluenz: Lokale Berechnungswege spielen keine Rolle

Beachte: lokal konfluent i.allg. leichter prüfbar als konfluent
CR und Konfluenz wünschenswert für effiziente Implementierung funktionaler Sprachen. Es gilt:
Satz 1. $\to$ ist CR $\leftrightarrow$ $\to$ ist konfluent
2. $\to$ ist konfluent $\Rightarrow \to$ ist lokal konfluent
Umkehrung von 2 gilt leider nicht:

Beispiel:
Sei $\to$ definiert durch $b \to a$
$b \to c$
$c \to b$
$c \to d$
Graphisch: $a \leftarrow b \leftrightarrows c \to d$

Lokale Divergenzen: $b \nearrow a$, $b \searrow c$ $\rightarrow$ $b \searrow^{0} a$, $b \nearrow a$ $\Rightarrow$ lokal konfluent

$c \nearrow b$, $c \searrow d$ $\rightarrow$ $c \longrightarrow^{0} d$

Aber nicht konfluent: $b \nearrow^{*} a$, $b \searrow^{*} d$, jedoch nicht $a \downarrow d$

Problem hier: „Schleife“ zwischen b und c

Def.: Sei $(M, \rightarrow)$ Reduktionssystem

1. $\rightarrow$ heißt *terminierend* oder *Noethersch*, falls keine unendlichen Ketten $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots$ existieren
2. $\rightarrow$ heißt *konvergent* (eindeutig terminierend), falls $\rightarrow$ konfluent und terminierend ist.

Satz: Sei $(M, \rightarrow)$ terminierend. Dann gilt:
$\rightarrow$ konfluent $\Leftrightarrow$ $\rightarrow$ lokal konfluent
(Newman-Lemma)

Somit: Für terminierende Reduktionssysteme Konfluenz „einfach“ prüfbar: Betrachte alle 1-Schritt-Divergenzen (evtl. nur endlich viele) und berechne Normalformen der Elemente (existieren wegen Terminierung):
falls identisch $\Rightarrow$ (lokal) konfluent

Aber: Reduktionen in funktionalen Sprachen nicht immer terminierend
($\rightarrow$unendliche Datenstrukturen)

2.2 Termersetzungssysteme

Ursprüngliche Motivation: Rechnen mit Gleichungsspezifikationen

Beispiel: Gruppenaxiome:

$$\begin{array}{lcll} x * 1 & = & x & (1) \\ x * x^{-1} & = & 1 & (2) \\ (x * y) * z & = & x * (y * z) & (3) \end{array}$$

Rechnen: Ersetze Gleiches durch Gleiches
Bsp.: Wir wollen zeigen: $1 * x = x$ gilt für alle Gruppen:

$$
\begin{array}{rcll}
1 * x & = & (x * x^{-1}) * x & (2) \\
& = & x * (x^{-1} * x) & (3) \\
& = & x * (x^{-1} * (x * 1)) & (1) \\
& = & x * (x^{-1} * (x * (x^{-1} * (x^{-1})^{-1}))) & (2) \quad \text{für } x^{-1} \text{ statt } x \\
& = & x * (x^{-1} * ((xxx^{-1}) * (x^{-1})^{-1})) & (3) \\
& = & x * (x^{-1} * (1 * (x^{-1})^{-1})) & (2) \\
& = & x * ((x^{-1} * 1) * (x^{-1})^{-1}) & (3) \\
& = & x * (x^{-1} * (x^{-1})^{-1}) & (1) \\
& = & x * 1 & (2) \\
& = & x & (1)
\end{array}
$$

Aspekte:

- Terme, Anwendungen von Gleichungen in Teiltermen
- „Pattern matching“: Ersetzen von Variablen in Gleichungen durch andere Terme
- Gleichungen in beide Richtungen anwenden

Termersetzung als Modell zum Rechnen mit Funktionen:

- Gleichungen nur von links nach rechts anwenden
- Einschränkung der linken Seiten, damit effektive Strategien möglich sind

Definition:

- **Signatur:** $\Sigma = (S, F)$ mit
  - S: Menge von *Sorten* ($\approx$ Basistypen Nat, Bool,...)
  - F: Menge von *Funktionssymbolen* $f :: s_1, \ldots, s_n \to s, n \geq 0$
    ($c ::\to s$ heißen auch *Konstanten*)
- **Variablen** haben auch Sorten, d.h. sei $V = \{x :: s \mid x \text{ Variablensymbol}, s \in S\}$
  mit $x :: s, x :: s' \in V \Rightarrow s = s'$ (kein Überladen)
- **Terme** über Σ und V der Sorte $s : T(\Sigma, V)_s$:
  $x \in T(\Sigma, V)_s$ falls $x :: s \in V$
  $f(t_1, \ldots, t_n) \in T(\Sigma, V)_s$ falls $f :: s_1, \ldots, s_n \to s \in F$ (wobei $\Sigma = (S, F)$) und
  $t_i \in T(\Sigma, V)_{s_i} (i = 1, \ldots, n)$
  Notation: Schreibe c statt $c()$ für Konstanten
- Variablen in einem Term t: $\mathcal{V}ar(t)$:
  $\mathcal{V}ar(x) = \{x\}$
  $\mathcal{V}ar(f(t_1, \ldots, t_n)) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$
- f heißt **Grundterm**, falls $\mathcal{V}ar(t) = \emptyset$
- f heißt **linear**, falls keine Variable in t mehrfach vorkommt (Bsp.: $f(x, y)$ linear, $f(x, x)$ nicht)

- **Termersetzungssystem (TES):** Menge von *Regeln* der Form $l \to r$ mit $l, r \in T(\Sigma, V)_s$ (für ein $s \in S$) und $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$
  Hierbei wird auch l als **linke Seite** und r als **rechte Seite** der Regel bezeichnet.

Beispiel:
Addition auf natürliche Zahlen (aufgebaut aus 0 und s(uccessor)):
$S = \texttt{Nat}, F = \{0 ::\to \texttt{Nat},\ s :: \texttt{Nat} \to \texttt{Nat}, + :: \texttt{Nat,Nat} \to \texttt{Nat}\}$

$$\begin{array}{lrcl} \text{Regeln:} & 0 + n & \to & n \\ & s(m) + n & \to & s(m + n) \end{array}$$

Semantik: Regeln sind Gleichungen, jedoch nur von links nach rechts anwenden
Ziel: Beschreibung des Rechnens mit $TES's:\ s(0) + s(0) \to s(s(0))$
Dazu notwendig: Anwendung einer Regel (*Substitution*) an beliebiger *Position*:

Definition:

- **Substitution** $\sigma : V \to T(\Sigma, V)$ mit $\sigma(x) \in T(\Sigma, V)_s$ für $x :: s \in V$
  wobei der $\mathcal{D}om(\sigma) = \{x \mid \sigma(x) \neq x\}$ endlich ist
  Notation: $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ bezeichnet $\sigma(x) = \begin{cases} t_i & \text{falls } x = x_i \\ x & \text{sonst} \end{cases}$
- **Positionen** in einem Term: Bezeichner für Teilterme, ausgedrückt durch Zahlenfolgen
  $\mathcal{P}os(t)$: Menge aller Positionen im Term t:
  - $\epsilon \in \mathcal{P}os(t)$ (Wurzelposition)
  - Falls $p \in \mathcal{P}os(t_i)$, dann ist auch $i \cdot p \in \mathcal{P}os(f(t_1, \ldots, t_n))$

  Teilterm bzw. **Unterterm** von t an Position $p \in \mathcal{P}os(t)$: $t|_p$, wobei
  $t|_\epsilon = t$
  $f(t_1, \ldots, t_n)|_{i \cdot p} = t_i|_p$
  $\underline{t[s]_p}$: **Ersetzung** des Teilterms an Position p im Term t durch s:
  $\overline{t[s]_\epsilon} = s$
  $f(t_1, \ldots, t_n)[s]_{i \cdot p} = f(t_1, \ldots, t_{i-1}, t_i[s]_p, t_{i+1}, \ldots, t_n)$

  Vergleich von Positionen:
  - $p \leq q$ (p ist *über* q) falls p Präfix von q, d.h. $\exists r$ mit $p.r = q$
  - p und q sind *disjunkt* $:\Leftrightarrow$ es gilt weder $p \leq q$ noch $q \leq p$
  - p heißt *links* von $q :\Leftrightarrow p = p_0 \cdot i \cdot p_1$ und $q = p_0 \cdot j \cdot q_1$ mit $i < j$

Nun können wir die Ersetzungsrelation bzgl. eines TES R definieren:
Definition:
Sei R ein TES. Dann ist die Reduktionsrelation (bzgl. R) $\to_R \subseteq T(\Sigma, V) \times T(\Sigma, V)$ wie folgt definiert:
$t_1 \to_R t_2 :\Leftrightarrow \exists l \to r \in R$, Position $p \in \mathcal{P}os(t)$ und Substitution σ mit $t_1|_p = \sigma(l)$ und $t_2 = t_1[\sigma(r)]_p$

$t_1 \to_R t_2$ wird auch **Reduktionsschnitt** oder **Ersetzungsschnitt** genannt. $\sigma(l)$ ist eine „passende“ linke Regelseite. In diesem Fall wird $t_1|_p$ auch **Redex** genannt.
Notation:

$t_1 \to_{p,l\to r} t_2$	falls Position und Regel wichtig
$t_1 \to t_2$	falls R fest gewählt ist

Beispiel:

$$\begin{array}{lllll} R: & 0+n & \to & n & (1) \\ & s(m)+n & \to & s(m+n) & (2) \end{array}$$

Für dieses Termersetzungssystem gibt es folgende Reduktionsschritte:

$$\begin{array}{lll} s(s(0))+s(0) & \to_{\varepsilon,(2)} & s(s(0)+s(0)) \\ & \to_{1,(2)} & s(s(0+s(0))) \\ & \to_{1.1,(1)} & \underbrace{s(s(s(0)))}_{\text{Normalform}} \end{array}$$

Wünschenswert: eindeutige Normalformen
Hinreichend: (Kap.2.1) Konfluenz bzw. CR von $\to_R$
Das obige +-Beispiel ist konfluent, wohingegen das Gruppenbeispiel nicht konfluent ist. Betrachte hierzu die Regeln:

$$\begin{array}{lllll} x * x^{-1} & \to & 1 & (1) \\ (x*y)*z & \to & x*(y*z) & (2) \end{array}$$

Dann gibt es für den Term $(a * a^{-1}) * b$ die folgenden Reduktionsschritte:

$$\begin{array}{lll} (a*a^{-1})*b & \to_{\epsilon,(2)} & a*(a^{-1}*b) \\ (a*a^{-1})*b & \to_{1,(1)} & 1*b \end{array}$$

Die jeweils letzten Terme sind verschiedene Normalformen des Terms $(a * a^{-1}) * b$. Daher sind die Regeln nicht konfluent.
Die Konfluenz ist also abhängig vom jeweiligen TES:
Hinreichende Kriterien? (z.B. wichtig für Compiler)
Knuth/Bendix (1970): Betrachte *kritische Paare* (Überlappungen linker Seiten)
Idee: Falls $t \begin{array}{c} \nearrow \\ \searrow \end{array} \begin{array}{c} t_1 \\ t_2 \end{array}$, dann muß gelten:

$$\exists p_1, \sigma_i, l_i \to r_i \in R \text{ mit } t|_{pi} = \sigma_i(l_i) \text{ und } t_i = t[\sigma_i(r_i)]p_i \text{ für } i = 1, 2$$

Unterscheide folgende Fälle:

- p_1 und p_2 sind disjunkt: To do: Bild
  Führe t_1 und t_2 zusammen durch Anwendung von $l_i \to r_i$ an Stelle p_i in t_{3-i} $(i = 1, 2)$

- p_1 über p_2, aber nicht innerhalb von l_1:
  To do: Bild

  t_1 und t_2 ebenfalls zusammenführbar
  (Beachte: p_2 innerhalb einer Variablen in l_1; somit $l_2 \to r_2$ auch anwendbar in allen diesen Variablen in r_2)

- p_2 über p_1 , aber nicht innerhalb von l_2: analog

- echte Überlappung von l_1 und l_2 $\quad \leadsto$ kritisches Paar

Definition:
Seien $l_1 \to r_1, l_2 \to r_2 \in R$ mit $Var(l_1) \cap Var(l_2) = \varnothing$, $p \in Pos(l_1)$ mit $l_1|_p \notin Var$ und σ mgu (vgl. Kap. 1.5.2, Typinferenz) für $l_1|_p$ und l_2. Falls $p = \epsilon$, dann sei auch $l_1 \to r_1$ keine Variante (gleich bis auf Variablenumbenennung) von $l_2 \to r_2$. Dann heißt

$$\langle \sigma(r_1), \sigma(l_1)[\sigma(r_2)]_p \rangle$$

kritisches Paar von $l_1 \to r_1$ und $l_2 \to r_2$.
Weiterhin bezeichnen wir mit $CP(R)$ die Menge aller kritischer Paare in R.

Erklärung:
$\sigma(l_1) \begin{array}{ll} \to_\epsilon & \sigma(r_1) \\ \to_p & \sigma(l_1)[\sigma(r_2)]_p \end{array}$ ist eine kritische Divergenz
(da $\sigma(l_1)|_p = \sigma(l_2)$)

Kritisches-Paar-Lemma (Knuth/Bendix 1970): Sei R ein TES.
Falls $t \to t_1$ und $t \to t_2$, dann ist $t_1 \downarrow_R t_2$ oder $t_1 \leftrightarrow_{CP(R)} t_2$.

D.h. alle echten lokalen Divergenzen kommen von kritischen Paaren $\Rightarrow$
Satz: Sei R TES. Dann gilt: R lokal konfluent $\Leftrightarrow t_1 \downarrow_R t_2 \; \forall \langle t_1, t_2 \rangle \in CP(R)$
Falls also R endlich und $\to_R$ terminierend: Entscheide Konfluenz durch Zusammenführung aller kritischen Paare
Und bei nichtterminierenden TES?

Untersuchung kritischer Paare nicht ausreichend:

$$R: \begin{array}{lll} f(x,x) & \to & a \\ f(x,c(x)) & \to & b \\ a & \to & c(a) \end{array} \quad \Rightarrow CP(R) = \varnothing$$

Lokal konfluent, aber nicht konfluent:

$$f(a,a) \begin{array}{ll} \nearrow & a \\ \searrow & f(a,c(a)) \to b \end{array}$$

Beachte: Erste linke Regelseite ist nicht-linear!
Definition: Ein TES R mit l linear $\forall l \to r \in R$ („links-linear“) heißt

- *orthogonal* falls $CP(R) = \varnothing$

- *schwach orthogonal* falls für alle $\langle t, t' \rangle \in CP(R)$ gilt: $t = t'$
  (nur triviale kritische Paare)

Es gilt:
Satz: (Schwach) orthogonale TES sind konfluent.
Somit: Hinreichende Kriterien für Konfluenz:

- bei terminierenden TES: Zusammenführung kritischer Paare
  terminierend = teilweise prüfbar mit sog. Terminierungsordnungen
- im allg.: Prüfe Linkslinearität und Trivialität von Überlappungen

Problem: wie findet man die Normalform (falls sie existiert) eines Terms bzgl. eines konfluenten TES?
Bei terminierenden TES: beliebige Folge von Reduktionsschnitten
Im allgemeinen: Durch geeignete Reduktionsstrategie (im folgenden: R konfluentes TES):

Definition:
Reduktionsstrategie $S : T(\Sigma, V) \to 2^{Pos(T(\Sigma,V))}$, wobei $S(t) \subseteq Pos(t)$ mit $t|_p$ ist Redex $\forall p \in S(t)$ und $\forall p_1, p_2 \in S(t)$ mit $p_1 \neq p_2$ gilt: p_1 und p_2 sind disjunkt.

Intuition: S legt die Positionen fest, bei denen als nächstes reduziert wird

Weiter sei:

- $t_1 \to_R t_2 \to_R t_3 \to_R \ldots$ heißt S-Reduktion, falls für $i = 1, 2, 3 \ldots$ gilt: t_{i+1} entsteht aus t_i durch Anwendung von Reduktionsschritten an allen Positionen aus $S(t_i)$
- S heißt *sequentiell* $:\Leftrightarrow \forall t \in T(\Sigma, V)$ gilt: $|S(t)| \leq 1$
- S heißt *normalisierend* $:\Leftrightarrow \forall t, t' \in T(\Sigma, V)$ mit $t' = t \downarrow_R$ gilt: jede S-Reduktion $t \to_R t_1 \to_R \ldots$ endet in t'
  (Strategie S berechnet immer die Normalform, falls sie existiert)

Beispiele für Reduktionsstrategien (im folgenden nehmen wir an, dass $t \in T(\Sigma, V)$ nicht in Normalform ist):

Leftmost innermost (LI): $S(t) = \{p\}$ falls $t|_p$ Redex und $\forall q \neq p$ mit $t|_q$ Redex gilt: q rechts oder über p

Leftmost outermost (LO): $S(t) = \{p\}$ falls $t|_p$ Redex und $\forall q \neq p$ mit $t|_q$ Redex gilt: p über oder links von q

Parallel outermost (PO): $S(t) = \{p \mid t|_p$ Redex und $\forall q \neq p$ mit $t|_q$ Redex gilt: $q \not\leq p\}$

LI ist nicht normalisierend:

$$\begin{aligned} R: \quad & 0 * x \to 0 \\ & a \to a \end{aligned}$$

$(0 * a) \downarrow_R = 0$, aber :
$0 * \underline{a} \to_R^{LI} 0 * \underline{a} \to_R^{LI} 0 * a \ldots$

LO ist nicht normalisierend für orthogonale TES:[1]

$$\begin{array}{lll} R: & a & \to b \\ & c & \to c \\ & f(x,b) & \to d \end{array}$$

$f(c, \underline{a}) \to_R \underline{f}(c,b) \to_R d$, d.h. $f(c,a) \downarrow_R = d$, aber:
$f(\underline{c}, a) \to_R^{LO} f(\underline{c}, a) \to_R^{LO} f(\underline{c}, a) \ldots$
Dagegen: $f(\underline{c}, \underline{a}) \to_R^{PO} \underline{f}(c,b) \to_R^{PO} d$

Allgemein:
Satz (O'Donnell, LNCS'77): PO ist normalisierend für (schwach) orthogonale Systeme.
Aber: PO aufwendiger zu implementieren als LO

Voriges Beispiel: Rightmost outermost auch normalisierend.
Manchmal ist aber parallele Reduktion essentiell:

Beispiel: „Paralleles Oder":

$$\begin{array}{lllll} \texttt{R:} & \texttt{true} & \vee\ x & \to & \texttt{true} \\ & x & \vee\ \texttt{true} & \to & \texttt{true} \\ & \texttt{false} & \vee\ \texttt{false} & \to & \texttt{false} \end{array}$$

Gegeben: $t_1 \vee t_2$:
Gefahr bei sequentieller Strategie: Falls t_1 (oder t_2) versucht wird zu reduzieren, könnte dies zu unendlicher Ableitung führen, wohingegen t_2 (oder t_1) zu `true` reduzierbar ist.
Daher: „sicheres" Vorgehen: t_1 und t_2 parallel auswerten.

Hier: R nicht orthogonal, da Regel 1+2 überlappen
Daher: Orthogonale Systeme sequentiell normalisierbar?
Leider nicht so einfach!

Beispiel: (Berry)

$$\begin{array}{lll} R: & f(x,0,1) & \to 0 \\ & f(1,x,0) & \to 1 \\ & f(0,1,x) & \to 2 \end{array}$$

Dieses System ist orthogonal, aber:
$f(t_1, t_2, t_3)$: Welches t_i zuerst reduzieren?
⇒ Interessante Frage: Gibt es für bestimmte Klassen von TES sequentielle (und evtl. optimale) Reduktionsstrategien?

[1] Beachte: Die Strategie LO ist nicht identisch zur Pattern-Matching-Strategie aus Kapitel 1.3!

Definition: TES R heißt *linksnormal (left-normal)*, falls für alle Regeln $l \to r \in R$ gilt: in l steht „hinter“ einer Variablen kein Funktionssymbol
Beispiele:
Das System

$$\begin{array}{lcl} 0 * x & \to & 0 \\ a & \to & a \end{array}$$

ist linksnormal. Dagegen ist die Regel

$$f(x, b) \quad \to \quad d$$

nicht linksnormal.

Satz (O'Donnell): LO ist normalisierend für linksnormale orthogonale TES.

Anwendung Kombinatorlogik (und TES R mit $Var(l) = \varnothing \forall l \to r \in R$) sind linksnormal
$\Rightarrow$ LO normalisierend
($\rightsquigarrow$ Implementierung funktionaler Sprachen durch Kombinatoren kann LO statt PO sein)

Weitere wichtige Klasse von TES:

Definition: Sei R ein TES bzgl. Signatur $\Sigma = (S, F)$.

- $f : s_1, \ldots, s_n \to s \in F$ heißt *definierte Funktion*, falls eine Regel $f(t_1, \ldots, t_n) \to r$ existiert.
  $D = \{f | f \text{ definierte Funktion } \} \subseteq F$
- C=F $\setminus$ D heißt Menge der *Konstruktoren*
- $f(t_1, \ldots, t_n)(n \geq 0)$ heißt *Muster (pattern)*, falls $f \in 0$ und $t_1, \ldots, t_n$ enthalten keine definierte Funktion (sog. *Konstruktorterme*)
- R heißt *konstruktorbasiert*, falls $\forall l \to r \in R$ l ein Muster ist

Beispiel:

$$R: \quad \begin{array}{ll} 0 + n & \to n \\ s(m) + n & \to s(m + n) \end{array} \quad \Rightarrow D = \{+\}, C = \{0, s\}, \text{ konstruktorbasiert}$$

Gruppenaxiome: nicht konstruktorbasiert

Konstruktoren: bauen Datenstrukturen auf
Defnierte Funktionen: rechnen auf Datenstrukturen
konstrukturbasierte TES: $\sim$ funktionale Programme, jedoch keine Reihrnfolge der Regeln ($\to$ gleichungsorientiert)

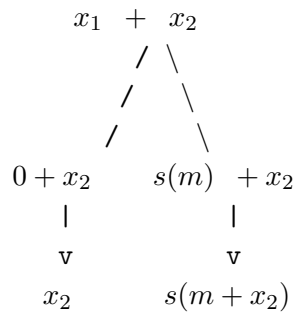
Induktiv-sequentielle TES [Antoy 1992 (LNCS 632)]: intuitiv: Funktionen sind induktiv über den Datenstrukturen definiert
Beispiel: Addition +: Fallunterscheidung (Induktion) im 1. Argument.

Formal:
Definition:

- Ein *definierender Baum (definitional tree)* ist ein Baum, bei dem jeder Knoten mit einem Muster markiert ist. Dabei gibt es zwei Sorten von def. Bäumen mit Muster π:
  - *Regelknoten* der Form $l \to r$ mit $\pi = l$
  - *Verzweigungsknoten* der Form branch $(\pi, p, T_1, \ldots, T_k)$, wobei:
    - $p \in Pos(\pi)$ mit $\pi/p \in Var$
    - $T_i (i = 1, \ldots, k)$ ist def. Baum mit dem Muster $\pi[c_i(x_1, \ldots, x_{mi})]_p$, wobei $x_1, \ldots, x_{mi}$ neue Variablen sind, und $c_1, \ldots, c_n$ sind verschiedene Konstruktoren

    pattern (T): Muster des def. Baumes T
  - Sei R ein TES.
    - T heißt definierender Baum für Funktion f, falls T endlich ist und das Muster $f(x_1, \ldots, x_n)$ hat ($x_1, \ldots, x_n$ verschiedene Variablen), jeder Regelknoten ist Variante einer Regel aus R, und jede Regel $f(t_1, \ldots, t_n) \to r \in R$ kommt in T genau einmal vor. In diesem Fall heißt f *induktiv-sequentiell*.
    - R heißt *induktiv-sequentiell*, falls alle $f \in D$ induktiv-sequentiell sind.

Beispiel:
R: $0 + n \to n$
$s(m) + n \to s(m + n)$
Definierender Baum (graphisch):



Dagegen:

`true`	$\vee$	x	$\to$	`true`
x	$\vee$	`true`	$\to$	`true`
`false`	$\vee$	`false`	$\to$	`false`

nicht induktiv-sequentiell

Satz: Jedes induktiv-sequentielle TES ist orthogonal und konstruktor-basiert (aber nicht umgekehrt!)

Für induktiv-sequentielle TES existiert einfache sequentielle Reduktionsstrategie, die Konstruktor-normalformen berechnet:

Definition Reduktionsstrategie φ sei wie folgt definiert:

Sei t ein Term, o Position des linkesten äußersten definierten Funktionssymbols in t (d.h. $t|_o = f(t_1, \ldots, t_n)$ mit $f \in D$) und T ein def. Baum für f. Dann ist:

$\varphi(t) = \{o \cdot \varphi(t|_o, T)\}$

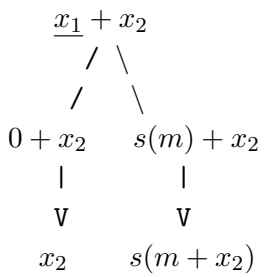
Jier ist zu beachten, dass $\varphi(t|_o, T)$ eventuell undefiniert sein kann; in diesem Fall ist auch $\varphi(t)$ undefiniert). Weiterhin ist
$\varphi(t, l \to r) = \epsilon$ (d.h. wende Regel an)

$$\varphi(t, branch(\pi, p, T_1, \ldots, T_k)) = \begin{cases} \varphi(t, T_j) & \text{falls } t|_p = c(\cdots)\text{und pattern}(T_j)|_p = c(\cdots) \\ p \cdot \varphi(t|_p, T) & \text{falls } t|_p = f(\ldots)\text{mit } f \in D \text{ und T def. Baum für } f \end{cases}$$

Intuitiv: Auswertung einer Funktion durch Analyse des zugehörigen def. Baumes:

- Bei Regelknoten: Anwenden der Regel
- Bei Verzweigungsknoten: Aktueller Wert an Verzweigungsposition ist
  - Konstruktor $\rightsquigarrow$ betrachte entsprechenden Teilbaum
  - Funktion $\rightsquigarrow$ Werte Funktion aus

Beispiel:
Regeln für + (s.o.), Term:
$t = (s(0) + 0) + 0 : \varphi(t) = 1 \cdot \varphi(s(0) + 0, \underline{x_1} + x_2) = 1 \cdot \varphi(s(0) + 0, s(m) + x_2) = 1 \cdot \epsilon = 1$
wobei:



$$\begin{array}{rlll} \Rightarrow \quad t & \to_R^\varphi & s(0+0)+0 & \\ & \to_R^\varphi & s((0+0)+0) & \text{da } \varphi(s(0+0)+0) = \epsilon \\ & \to_R^\varphi & s(0+0) & \text{da } \varphi(s((0+0)+0) = 1 \,.\, 1 (\approx 0+0) \\ & \to_R^\varphi & s(0) & \text{da } \varphi(s(0+0)) = 1 \end{array}$$

φ berechnet evtl. nicht die Normalform:

1. Term enthält Variablen: $(x + 0) * (0 * 0) : \varphi$ hierfür undefiniert,
   Normalform: $(x + 0) + 0$
2. Partielle Funktionen:
   Betrachte zusätzliche Regel: $f(s(m), n) \to 0$
   Dann φ undefiniert auf $f(0, 0 + 0)$ (aber Normalform: $f(0, 0)$

Definition Funktion f heißt *vollständig definiert*, falls ein def. Baum für f existiert, wobei in jedem Verzweigungsknoten jeder Konstruktor (der entsprechenden Sorte) in einem Teilbaum vorkommt.

Beispiel: Konstruktoren von Nat: $0, s \Rightarrow +$ vollständig definiert, f jedoch nicht.
Nun gilt:
Satz: Ist R ein induktiv-sequentielles TES und sind alle Funktionen vollständig definiert, dann ist φ normalisierend auf Grundtermen.

Ausreichend für das Rechnen in funktionalen Sprachen, da

- nur Ausrechnen von Grundtermen
- Normalformen mit partiellen Funktioene $\rightsquigarrow$ Fehlermeldung

Vergleich: Pattern matching (Kap. 1.3) $\leftrightarrow$ Strategie φ
Pattern matching:

- abhängig von Reihenfolge der Regeln
- im allg. nicht normalisierend
- normalisierend auf uniformen Programmen
- erlaubt überlappende linke Regelseiten ($\rightsquigarrow$ Ergebnis von Reihenfolge abhängig!)

Strategie φ:

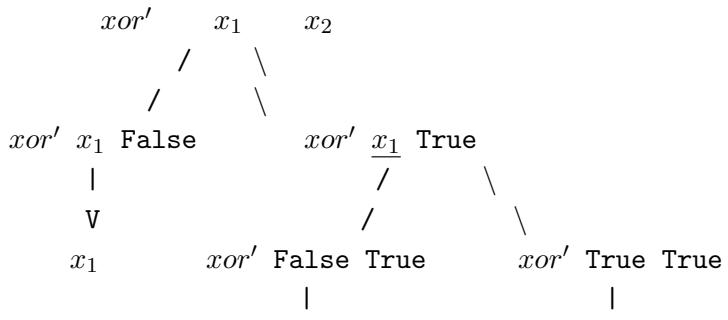
- unabhängig von Regelreihenfolge
- normalisierend
- erlaubt keine Überlappungen

Satz: Jede uniformale Funktionsdefinition ist induktiv-sequentiell.

Umkehrung gilt nicht:
Beispiel: (vgl. Kap. 1.3)

```
xor'  x      False = x
xor'  False  True  = True
xor'  True   True  = False
```

Nicht uniform, aber induktiv-sequentiell (durch Verzweigung über 2. Argument):



```
V                V
True             False
```

$\Rightarrow \varphi$ berechnungsstärker als einfaches Links-rechts Pattern Matching

Weitere Aspekte: Einfache Erweiterung von def. Bäumen auf überlappende Regeln mit paralleler Auswertung ($\rightsquigarrow$ Antoy 1992)

Kapitel 3

Rechnen mit partieller Information: Logikprogrammierung

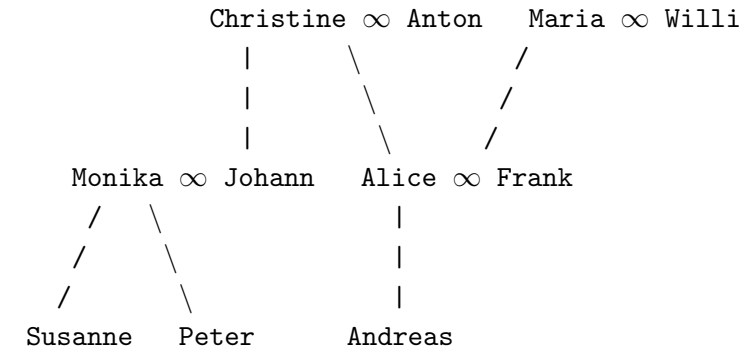
3.1 Motivation

Prinzipiell Funktionale Sprachen sind berechnungsuniversell
⇒ alles programmierbar, keine Erweiterungen notwendig

Jedoch Programmierstil und Lesbarkeit der Programme wichtig
⇝ Erweiterungen wünschenswert, falls dies zu besser strukturierten Programmen führt.

Charakteristik funktionaler Sprachen: Variablen als Unbekannte in Regeln, nicht jedoch in zu berechnenden Ausdrücken, d.h. letztere sind immer *Grundterme.*
Lockerung dieser Einschränkung wünschenswert bei *Datenbankanwendungen*

Beispiel: Betrachte Verwandtschaft, d.h. Personen + Beziehungen:
∞ : verheiratet
/ : Mutter-Kind-Beziehung



Ziel: Allgemeine Definition abgeleiteter Beziehungen wie z.B. Tante, Großvater,...

Modellierung in Haskell:
Datentyp der Personen (könnte auch String o.ä. sein):

```
data Person = Christine | Anton | Maria | Willi | ... | Andreas
```

„verheiratet“-Beziehung als Funktion

```
ehemann :: Person   → Person
ehemann   Christine = Anton
ehemann   Maria     = Willi
ehemann   Monika    = Johann
ehemann   Alice     = Frank
```

„Mutter-Kind“-Beziehung als Funktion

```
mutter :: Person → Person
mutter   Johann  = Christine
mutter   Alice   = Christine
mutter   Frank   = Maria
mutter   Susanne = Monika
mutter   Andreas = Alice
```

Aus diesen grundlegenden Beziehungen können wir weitere allgemein ableiten:
Vater ist Ehemann der Mutter (idealerweise!):

```
vater:: Person → Person
vater kind = ehemann (mutter kind)
```

Großvater-Enkel-Beziehung ist im allgemeinen eine *Relation*:

```
grossvater :: Person → Person → Bool
grossvater g e | g == vater (vater e)  = True
               | g == vater (mutter e) = True
```

Hiermit z.B. folgende Berechnungen möglich:

- Wer ist der Vater von Peter?
  `vater Peter` $\leadsto$ `Johann`
- Ist Anton Großvater von Andreas?
  `grossvater Anton Andreas` $\leadsto$ `True`

Aber nicht:

1. Welche Kinder hat Johann?
2. Welche Großväter hat Andreas?
3. Welche Enkel hat Anton?

Möglich, falls *Variablen in Ausdrücken* zulässig wären:

1. `vater k == Johann` $\leadsto$ `k = Susanne` oder `k = Peter`

2. `grossvater g Andreas` $\leadsto$ `g = Anton` oder `g = Willi`

3. `grossvater Anton e` $\leadsto$ `e = Susanne` oder `e = Peter` oder `e = Andreas`

Idee: Falls Variablen in Ausdrücken vorkommen $\leadsto$ Suche nach passenden Werten, so daß Berechnung möglich
Nicht erlaubt in funktionalen Sprachen $\leadsto$ erweitere Programme (definiere Umkehrrelation)

Erlaubt in Logiksprachen wie z.B. Prolog
Charakteristik: Berechnung von *Lösungen*, d.h. Belegung der Variablen, so daß Ausdruck reduzierbar.
Hauptproblem: Wie findet man konstruktiv die Lösungen? → später

Neue Aspekte:

1. **Constraints** (Einschränkungen) statt Gleichheitstests:
   Betrachte z.B. die Definition von `grossvater`: Hier ist man nur an positiven Resultaten (d.h. Auswertungen zu `True`) interessiert, aber nicht an Auswertungen, die z.B. besagen, wann jemand kein Großvater einer Person ist. Dies ist im allgemeinen häufig der Fall. Aus diesem Grund führen wir neben der Gleichheit "`==`", die zu `True` oder `False` auswertet, auch ein **Constraint** der Form

   e_1 `=:=` e_2

   ein. Hierbei können e_1 und e_2 Variablen enthalten. Die Bedeutung eines solchen Constraint ist, dass es *gelöst* werden soll, d.h. es werden Werte für die Variablen in e_1 und e_2 bzw. eine Substitution σ gesucht, so dass $\sigma(e_1)$ und $\sigma(e_2)$ die gleichen Werte haben.

   Anmerkungen zum Unterschied zwischen "`==`" und "`=:=`":

   e_1 `==` e_2 ist ein **Test**: prüfe, ob e_1 und e_2 gleich oder verschiedene Werte haben. Daher hat "`==`" den Typ `a -> a -> Bool`.

   e_1 `=:=` e_2 ist eine **Einschränkung** (**Constraint**): e_1 und e_2 *müssen* gleiche Werte haben (z.B. find Werte der Variablen in e_1 und e_2, so dass die Werte gleich sind). Aus diesem Grund hat der Ausdruck e_1 `=:=` e_2 nicht den Typ `Bool` (da `False` nie ein Ergebnis dieses Ausdrucks ist).
   Konsequenz: e_1 `=:=` e_2 hat den speziellen Typ `Success`, der keine Werte besitzt, sondern nur das Ergebnis einer Einschränkung bezeichnet, d.h. erfolgreich ist).

   Basisoperationen für Einschränkungen:

```
(=:=) :: a -> a -> Success              -- Gleichheitsconstraint
success :: Success                      -- immer erfuellbares Constraint
(&) :: Success -> Success -> Success    -- Konjunktion von Constraints
failed :: a                             -- Fehlschlag
```

Beispiel: Der Constraint

```
xs++ys =:= [1,2,2] & zs++ys =:= xs
```

kann zu folgenden zwei Antworten erfolgreich ausgewertet werden:

```
xs=[1,2], ys=[2], zs=[1]
xs=[1,2,2], ys=[], zs=[1,2,2]
```

2. **Nichtdeterministische Berechnungen** (auch bei Konfluenz!), da im allgemeinen mehr als eine Lösung existiert

3. **Flexiblere Programmierung:** Umkehrfunktion/-relationen stehen automatisch zur Verfügung
   Definiere `f x = ...`
   Benutze Umkehrung: `f y == ...` $\leadsto$ `y = ...`

4. **Automatische Lösungssuche** statt expliziter Programmierung der Suche nach Lösungen

5. **Variablen in initialen Ausdrücken** (auch **Anfragen** genannt) sind existenzquantifiziert, d.h.
   `vater k == Johann` bedeutet $\exists$`k: father k == Johann`
   Dagegen sind Variablen in Regeln allquantifiziert, d.h.
   `vater k = ehemann (mutter k)` bedeutet $\forall$`k: vater k = ehemann (mutter k)`
   d.h. hier steht `k` für einen beliebigen Wert

6. **Variablen in Regeln**: l `|` c `=` r
   Beachte: c kann vom Typ `Bool` (wie in Haskell) als auch ein Constraint vom Typ `Success` sein!
   Forderung in funktionalen Sprachen: Alle Variablen in c oder r kommen auch in l vor. Ähnlich zu initialen Ausdrücken können hier aber auch **Extravariablen** (Variablen in c oder r, die nicht in l vorkommen) sinnvoll sein. Hierzu einige Beispiele:

   - Letztes Element einer Liste:

     ```
     last l | xs++[e]==l = e
                          where xs,e free
     ```

     wobei `xs` und `e` Extravariablen sind und daher explizit deklariert werden müssen.

   - Element einer Liste:

     ```
     member e l | xs++(e:ys)==l = True
                                  where xs,ys free
     ```

   - Ist eine Liste Teil einer anderen Liste?

     ```
     sublist s l | xs++s++ys==l = True
                                  where xs,ys free
     ```

Intuitive Bedeutung von Extravariablen in Regeln:

- Extravariablen sind existenzquantifiziert
- Finde Belegung für diese, so dass Bedingung beweisbar ist.

In Logiksprachen wie Prolog ist dies möglich. Allerdings ist in reinen Logiksprachen die Syntax eingeschränkter, denn dort kann man ist nur die Definition positiver Relationen erlaubt, d.h. rechte Seite ist immer `success`
⇒ jede Gleichung hat die Form

l `|` c_1 `&` `...` `&` c_n `=` `success`

wobei: $c_1, \cdots, c_n$ Literale sind (Anwendung eines Prädikats auf Argumente).
Schreibweise in Prolog:

l `:-` c_1`,...,`c_n`.`

Im folgenden betrachten wir **logisch-funktionale Sprachen**, insbesondere Curry[1], d.h. funktionale Sprachen + existenzquantifizierte Variablen (auch „freie Variablen“ oder „logische Variablen“ genannt).
Somit: Logiksprachen als Spezialfall
(Ausnahme: In l`:-`$l_1, \ldots, l_n$ muß l nicht linear sein im Gegensatz zu funktionalen Sprachen.)
Bevor wir uns mit Auswertungsstrategien logisch-funktionaler Sprachen beschäftigen, zeigen wir noch einige Beispiele zur Programmierung.

Logisch-funktionale Sprachen sind gut geeignet zum Lösen von Suchproblemen, falls man also keinen direkten Algorithmus hat, die Lösung eines Problems zu berechnen.

Beispiel: Spiel 24

Aufgabe: Bilde arithmetische Ausdrücke genau mit den Zahlen 2,3,6,8, so dass das Ergebnis der Wert 24 ist (Anmerkung: Division ist ganzzahlig). Beispiele für solche Ausdrücke: $(3 * 6 - 2) + 8$, $3 * 8 + 8 - 2$,...
Lösungsidee:

1. bilde Pemutationen der Liste `[2,3,6,8]`
2. erzeuge arithmetische Ausdrücke über diesen Permutationen
3. prüfe, ob der Wert 24 ist

Berechnung von Permutationen einer Liste:

```
permute [] = []
permute (x:xs) | u++v =:= permute xs = u++[x]++v where u, v free
```

Beachte, dass `permute` *eine* beliebige Permutation liefert, z.B. wertet `permute [1,2,3]` zu den Werten

[1] `www.curry-language.org`

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

aus. Da diese Funktion nichtdeterministisch einen dieser Werte liefert, spricht man auch von einer **nichtdeterministischen Funktion**.
Zur Repräsentation von Ausdrücken definieren wir einen Datentyp:

```
data Exp = Num Int      -- Zahl
         | Add Exp Exp
         | Mul Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
```

Nun definieren wir eine Operation, die einen Ausdruck testet, ob dieser nur Ziffern einer Zahlenliste enthält, und dabei den Wert des Ausdrucks berechnet:

```
test :: Exp -> [Int] -> Int
test (Num y) [z] | y =:= z = y
test (Add x y) l | split u v l = test x u + test y v where u, v free
test (Sub x y) l | split u v l = test x u - test y v where u, v free
test (Mul x y) l | split u v l = test x u * test y v where u, v free
test (Div x y) l | split u v l = opdiv (test x u) (test y v)
  where
    u, v free
    opdiv a b = if b == 0 || a `mod` b /= 0 then failed else a `div` b
```

Hierbei ist `split` ein Constraint, dass erfüllt ist, wenn die ersten beiden Listen nicht-leer sind und zusammen gleich zur dritten Liste ist:

```
split (u:us) (v:vs) l | (u:us)++(v:vs) =:= l = success
```

Nun können wir sehr einfach durch Lösen eines Gleichheitsconstraint Lösungen für unser Problem berechnen:

```
test e (permute [2,3,6,8]) =:= 24
```

Damit erhalten wir folgende Lösungen:

```
e = Add (Sub (Mul (Num 3) (Num 6)) (Num 2)) (Num 8)
e = Add (Mul (Num 3) (Num 6)) (Sub (Num 8) (Num 2))
e = Add (Mul (Num 3) (Sub (Num 8) (Num 2))) (Num 6)
e = Add (Sub (Mul (Num 6) (Num 3)) (Num 2)) (Num 8)
e = Add (Num 6) (Mul (Num 3) (Sub (Num 8) (Num 2)))
e = Add (Mul (Num 6) (Num 3)) (Sub (Num 8) (Num 2))
e = Add (Num 6) (Mul (Sub (Num 8) (Num 2)) (Num 3))
```

```
e = Add (Sub (Num 8) (Num 2)) (Mul (Num 3) (Num 6))
e = Add (Mul (Sub (Num 8) (Num 2)) (Num 3)) (Num 6)
e = Add (Num 8) (Sub (Mul (Num 3) (Num 6)) (Num 2))
e = Add (Sub (Num 8) (Num 2)) (Mul (Num 6) (Num 3))
e = Add (Num 8) (Sub (Mul (Num 6) (Num 3)) (Num 2))
e = Sub (Mul (Add (Num 2) (Num 8)) (Num 3)) (Num 6)
e = Sub (Mul (Num 3) (Num 6)) (Sub (Num 2) (Num 8))
e = Sub (Add (Mul (Num 3) (Num 6)) (Num 8)) (Num 2)
e = Sub (Mul (Num 3) (Add (Num 2) (Num 8))) (Num 6)
e = Sub (Mul (Num 3) (Add (Num 8) (Num 2))) (Num 6)
e = Sub (Num 6) (Mul (Num 3) (Sub (Num 2) (Num 8)))
e = Sub (Mul (Num 6) (Num 3)) (Sub (Num 2) (Num 8))
e = Sub (Add (Mul (Num 6) (Num 3)) (Num 8)) (Num 2)
e = Sub (Num 6) (Mul (Sub (Num 2) (Num 8)) (Num 3))
e = Sub (Num 8) (Sub (Num 2) (Mul (Num 3) (Num 6)))
e = Sub (Mul (Add (Num 8) (Num 2)) (Num 3)) (Num 6)
e = Sub (Add (Num 8) (Mul (Num 3) (Num 6))) (Num 2)
e = Sub (Num 8) (Sub (Num 2) (Mul (Num 6) (Num 3)))
e = Sub (Add (Num 8) (Mul (Num 6) (Num 3))) (Num 2)
```

Diese könnte man dann noch natürlich noch schöner ausdrucken...

Nichtdeterministische Funktionen

Wir haben gesehen, dass logisch-funktionale Sprachen die Definition nichtdeterministischer Funktionen erlauben (wie z.B. `permute`), d.h. Funktionen, die zu mehr als einen Wert auswerten. Dies ist in vielen Anwendungen sinnvoll, wie wir oben gesehen haben. Der Prototyp einer nichtdeterministischen Funktion ist eine Funktion, die eines ihrer Argumente zurück gibt. Diese ist in Curry vordefiniert durch die Regeln

```
x ? y = x
x ? y = y
```

Falls wir also eine Operation `coin` durch

```
coin = 0 ? 1
```

definierten, wird der Ausdruck `coin` zu 0 oder 1 ausgewertet. Die Verwendung solcher Operationen wird auch in folgendem Beispiel ausgenutzt.

Reguläre Ausdrücke

Wir definieren einen Datentyp fuer reguläre Ausdrücke über einem Alphabet `a`:

```
data RE a = Lit a
          | Alt  (RE a) (RE a)
          | Conc (RE a) (RE a)
```

```
          | Star (RE a)
```

Als Beispiel können wir z.B. den Ausdruck $(a|b|c)$ definieren:

```
abc = Alt (Alt (Lit 'a') (Lit 'b')) (Lit 'c')
```

Ein weiteres Beispiel ist der Ausdruck (ab^*):

```
abstar = Conc (Lit 'a') (Star (Lit 'b'))
```

Wir können aber auch einfach die Sprache der regulären Ausdrücke um weitere Konstrukte erweitern, wie z.B. einen plus-Operator, der eine mindestens einmalige Wiederholung bezeichnet:

```
plus re = Conc re (Star re)
```

Die Semantik regulärer Ausdrücke kann direkt durch eine Semantik-Funktion definiert werden, wie man dies auch in der Theorie der regulären Ausdrücke macht. Somit wird durch die Semantik jedem regulären Ausdruck ein dadurch beschriebenes Wort zugeordnet. Da es mehrere mögliche Worte geben kann, beschreiben wir dies durch eine nichtdeterministische Funktion.

```
sem :: RE a -> [a]
sem (Lit c)    = [c]
sem (Alt  a b) = sem a ? sem b
sem (Conc a b) = sem a ++ sem b
sem (Star a)   = [] ? sem (Conc a (Star a))
```

Beispielauswertungen:

`sem abc` $\leadsto$ `a` oder `b` oder `c`

Wenn wir einen String gegen einen regulären Ausdruck matchen wollen, können wir dies einfach durch folgendes Constraint beschreiben:

```
match :: RE a -> [a] -> Success
match r s = sem r =:= s
```

Ein Constraint, das ähnlich wie Unix's grep feststellt, ob ein regulärer Ausdruck irgendwo in einem String enthalten ist, können wir wie folgt definieren:

```
grep :: RE a -> [a] -> Success
grep r s = xs ++ sem r ++ ys =:= s  where xs,ys free
```

Beispielauswertungen:

`grep abstar "dabe"` $\leadsto$ `Erfolg!`

Beachte, dass der letzte Ausdruck eine endliche Auswertung hat, obwohl es unendliche viele Wörter gibt, zu denen `abstar` auswerten kann.

Partielle Datenstrukturen:

Datenstrukturen mit freien Variablen ⇒ Platzhalter für Werte, die evtl. später festgelegt werden
Mehr Flexibilität mit partiellen Datenstrukturen (wichtig z.B. in KI-Anwendungen: Aufsammeln partieller Informationen)

Beispiel:
Maximumbaum:

Gegeben:	Binärbaum mit ganzzahligen Blättern
Gesucht:	Baum mit gleicher Struktur, wobei in jedem Blatt das Maximum aller Blätter des Eingabebaums steht.

Definition des Datentyps eines Binärbaumes:

```
data BTree a = Leaf a | Node (BTree a) (BTree a)
```

Zu implementieren ist eine Funktion `maxtree` mit folgendem Verhalten:

```
maxtree (Node (Leaf 0) (Node (Leaf 1) (Leaf 2)))
⇝ (Node (Leaf 2) (Node (Leaf 2) (Leaf 2)))
```

Triviale Lösung: Zwei Baumdurchläufe:

1. Berechnung des Maximums
2. Konstruktion des Maximumbaumes

Mit logischen Variablen: Ein Durchlauf, wobei gleichzeitig Baum konstruiert und das Maximum berechnet wird.
Der Baum enthält als Blätter zunächst eine freie Variable anstelle des noch unbekannten Maximums.

Implementierung:

```
maxtree :: BTree Int → BTree Int
maxtree t  |     pass t mx     =:= (mt,mx)    =  mt
```

berechne Maximumbaum + Maximum

```
  where
    mt,mx free

    pass (Leaf n)     mx = (Leaf mx, n)
    pass (Node t1 t2) mx = (Node mt1 mt2, max m1 m2)
           where (mt1, m1) = pass t1 mx
                 (mt2, m2) = pass t2 mx
```

Beachte: Bei der Berechnung von "`pass t mx`" ist "`mx`" zunächst eine freie Variable, deren Wert erst noch berechnet wird.

Ablauf einer Berechnung:

```
maxtree (Node (Leaf 0) (Leaf 1))
⇝   pass (Node (Leaf 0)(Leaf 1)) mx    =:= (mt,mx)
  ⇝ ((Node (Leaf mx)(Leaf max)), 1)
                 = mt            = max
⇝ mt = (Node (Leaf 1) (Leaf 1))
```

3.2 Rechnen mit freien Variablen

In diesem Kapitel: Wie kann man mit freien Variablen rechnen?

Funktionale Programmierung, Termersetzung:
Rechnen durch *Reduktion* von Ausdrücken

1. Suche zu reduzierenden Teilausdruck
2. Suche „passende“ Regel (Fkt.gleichung)
3. Ersetze Regelvariablen durch „passende“ Ausdrücke (pattern matching)

Formal: $e \to e[\sigma(r)]_p$ falls $l \to r$ Regel und σ Substitution mit $\sigma(l) = e|_p$

Beispiel:

```
f 0 x = 0
f 1 x = 2
```

Mit Reduktion berechenbar:

```
f 0 1 → 0
f 1 2 → 2
f 0 x → 0
```

wobei `x` eine freie Variable ist
Nicht jedoch: `f z 1 =:= 1`
Eine Lösung wäre: $\sigma = \{\texttt{z} \mapsto 1\}$, denn $\sigma(\texttt{f z 1 =:= 1}) = \texttt{f 1 1 =:= 1} \to^* \texttt{success}$

Somit: Falls freie Variablen in Ausdrücken vorkommen:
Belegung der Variablen, so daß Ausdruck reduzierbar wird.

Problem: Wie findet man passende Belegung?

- Raten: zu ineffizient, i.allg. unendlich viele Möglichkeiten
- Konstruktiv: Ersetze Matching durch Unifikation
  ⇝ **Narrowing** (Verengen des Lösungsraumes)

Im folgenden: Termersetzungssysteme als Programme (da Funktionen höherer Ordnung nicht so relevant)
Sei R ein vorgegebenes TES (das eingegebene Programm)
Def.: Sind t und t' Terme, dann heißt $t \leadsto_\sigma t'$ **Narrowingschritt** (bzgl. R), falls gilt:

1. p ist eine nichtvariable Position in t (d.h. $t|_p \notin V$)
2. $l \to r$ ist Variante einer Regel aus R
   (d.h. ersetze in der Regel Variablen durch neue Variablen, so dass Namenskonflikte mit freien Variablen vermieden werden)
3. σ ist ein mgu (allgemeinster Unifikator, vgl. Kap. 1.5.2) für $t|_p$ und l (d.h. $\sigma(t|_p) = \sigma(l)$)
4. $t' = \sigma(t[r]_p)$

Weitere Notationen:

- $t \leadsto_{p,l\to r,\sigma} t'$ falls Position und/oder Regel wichtig
- $t \leadsto_{\sigma'} t'$ mit $\sigma'(x) = \begin{cases} \sigma(x) & \text{falls } x \in Var(t) \\ x & \text{sonst} \end{cases}$
  (in diesem Fall schreiben wir auch $\sigma' = \sigma|_{Var(t)}$) falls Belegung der Regelvariablen unwichtig
- $t_0 \leadsto_\sigma^* t_n$ falls $t_0 \leadsto_{\sigma_1} t_1 \leadsto_{\sigma_2} t_2 \leadsto \cdots \leadsto_{\sigma_n} t_n$ und $\sigma = \sigma_n \circ \ldots \circ \sigma_1$
  (hierbei bezeichnet $\circ$ die Komposition von Funktionen)

Im obigen Beispiel gibt es zwei mögliche Narrowing-Schritte:

```
f z 1 ⇝{z↦0} 0
      ⇝{z↦1} 1
```

Unterschiede zu Reduktion:

1. Belegung der freien Variablen durch Unifikation
   (beachte: freie Variablen können mehrfach vorkommen, daher **nicht** $t[\sigma(r)]_p$ in 4.)
2. Im allg. mehrere mögliche Belegungen $\leadsto$ **Nichtdeterminismus**

Beispiel:Verwandtschaft (→ Kap. 3.1)

```
vater k =:= Johann (Welche Kinder hat Johann?)
→ ehemann (mutter k) =:= Johann
  ⇝{k↦Susanne} ehemann Monika =:= Johann  →  Johann =:= Johann  →  success
  ⇝{k↦Peter}   ehemann Monika =:= Johann  →  Johann =:= Johann  →  success
```

Nichtdeterminismus muss aufwändiger implementiert werden ⇒
Forderung: möglichst wenig Nichtdeterminismus
$\leadsto$ Strategien (später)

Beobachtung: Narrowing ist Verallgemeinerung der Reduktion:
Falls Reduktionsschritt möglich, d.h.
$t \to t[\sigma(r)]_p$ mit $\sigma(l) = t|_p$ für Regel $l \to r$
Annahme: $\mathtt{Var}(l) \cap \mathtt{Var}(t) = \emptyset$ ($l \to r$ Variante!)
und $\mathtt{Dom}(\sigma) \subseteq \mathtt{Var}(l)$
$\Rightarrow$ $\sigma(t) = t$ und σ mgu für l und $t|_p$
$\Rightarrow$ $t \leadsto_\sigma \underbrace{\sigma(t[r]_p)}_{=t[\sigma(r)]_p}$ ist Narrowing-Schritt
Somit: Jeder Reduktionsschritt ist auch Narrowing-Schritt
(aber nicht umgekehrt!)

Ziel von Reduktionen: Berechnung einer (bzw. der) Normalform
Ziel von Narrowing: Finden von Lösungen (Variablenbelegungen), so daß Normalform berechenbar wird
Ausgangspunkt bei Narrowing: i.d.R. **Gleichungen**

Definition:
Eine **Gleichung** $s \doteq t$ heißt **gültig** (bzgl. R) (hier benutzen wir nicht den Gleichheitsoperator "==" von Haskell, da dessen Bedeutung unterschiedlich ist, wie wir noch sehen werden), falls $s \leftrightarrow_R^* t$ (d.h. s kann in t überführt werden)

Beispiel:
Addition auf natürlichen Zahlen (hierbei :: s: Nachfolgefkt.)
$0 \quad +n \quad \to \quad n$
$s(m) \quad +n \quad \to \quad s(m+n)$
Gleichung $s(0) + s(0) \doteq s(s(0))$ ist gültig, denn $s(0) + s(0) \to s(0 + s(0)) \to s(s(0))$

Falls R konfluent und terminierend:
$s \leftrightarrow_R^* t \;\Leftrightarrow \exists$ Normalform u mit $s \to^* u$ und $t \to^* u$
$\Rightarrow$ Berechne Normalformen beider Seiten um eine Gleichung zu prüfen.

Mittels Narrowing kann man Gleichungen **lösen**:
Substitution σ heißt Lösung der Gleichung $s \doteq t$, falls $\sigma(s) \doteq \sigma(t)$ gültig ist.

Beispiel:
$z + s(0) \doteq s(s(0))$ („Löse $z + 1 = 2$“)
$\leadsto_{[z/s(m)]} \quad s(m + s(0)) \quad = \quad s(s(0))$
$\leadsto_{[m/0]} \quad s(s(0)) \quad = \quad s(s(0))$
Lösung ist Komposition aller Substitutionen eingeschränkt auf ursprüngliche freie Variablen:
$\underbrace{[z/s(0)]}_{=([s/s(m)] \circ [m/0])|_{\{z\}}}$

Forderung an Lösungsverfahren:

1. **Korrektheit:** Berechne nur richtige Lösungen

2. **Vollständigkeit:** Berechne alle (bzw. Repräsentanten aller) richtigen Lösungen

Allgemeines Narrowing ist im folgenden Sinn korrekt und vollständig:
Satz (Hullot, 1980): Sei R TES so daß $\rightarrow_R$ konfluent und terminierend ist, und $s \doteq t$ eine Gleichung.

Korrektheit: Falls $s \doteq t \rightsquigarrow_\sigma^* s' \doteq t'$ und φ ist mgu für s' und t', dann ist $\varphi \circ \sigma$ Lösung von $s \doteq t$ (d.h. $\varphi(\sigma(s)) \doteq \varphi(\sigma(t))$ ist gültig)

Vollständigkeit: Falls σ' Lösung von $s \doteq t$ ist, dann existiert Narrowing-Ableitung $s \doteq t \rightsquigarrow_\sigma^* s' \doteq t'$, ein mgu φ für s' und t', und eine Substitution τ, so dass $\sigma'(x) \doteq \tau(\varphi(\sigma(x)))$ gültig ist $\forall x \in Var(s \doteq t)$.

Zur Vollständigkeit: Nur Repräsentanten, aber nicht alle Lösungen berechenbar:
Beispiel:

$$\begin{array}{lcl} f(a) & \rightarrow & b \\ g & \rightarrow & a \\ i(x) & \rightarrow & x \end{array}$$

1. Gleichung: $f(x) \doteq b$ Eine Lösung ist: $\sigma' = \{x \mapsto g\}$
   Narrowing berechnet: $f(x) \doteq b \rightsquigarrow_{\{x \mapsto a\}} b \doteq b$
   $\Rightarrow$ berechnete Lösung: $\sigma = \{x \mapsto a\}$
   Jedoch gilt: $\sigma(x) \doteq \sigma'(x)$ ist gültig

2. Gleichung: $i(z) \doteq z$. Eine Lösung ist: $\sigma' = \{x \mapsto a\}$
   Berechnete Lösung: $\sigma = \{\}$ (Identität)
   Jedoch gilt: σ' ist Spezialfall von σ

$\Rightarrow$ Narrowing berechnet allgemeine Repräsentanten aller möglichen Lösungen.

Dazu jedoch notwendig (wegen Existenz einer Ableitung): Berechne alle möglichen Narrowing-Ableitungen (d.h. rate Position **und** Regel in jedem Schritt!)
$\Rightarrow$ Verbesserung durch spezielle Strategien ($\rightsquigarrow$ später).

Forderung an TES:

1. Konfluenz: sinnvoll (vgl. funktionale Programmierung)

2. Terminierung: Einschränkung:

   - Wie überprüfen?
   - keine unendlichen Datenstrukturen

Narrowing auch vollständig bei nichtterminierenden Systemen, allerdings nur für **normalisierte Substitutionen** (d.h. in 2. muß σ' normalisiert sein, d.h. $\sigma'(x)$ ist in Normalform $\forall x \in Dom(x)$)

Unvollständigkeit bei nicht-normalisierten Substitutionen:
Beispiel:
$f(x,x) \rightarrow 0$
$g \rightarrow c(g)$
Gleichung: $f(y,c(y)) \doteq 0$: kein Narrowing-Schritt möglich, aber $\{y \mapsto g\}$ ist Lösung, denn

$$f(g,c(g)) \rightarrow f(c(g),c(g)) \rightarrow 0$$

Problem bei unendlichen Datenstrukturen: Sinnvolle Definition der Gleichheit
Bisherige Definition: $\doteq$ ist reflexiv, d.h. $t \doteq t$ ist immer gültig
($\rightsquigarrow$ „$\doteq$“ heißt **reflexive Gleichheit**)

Beispiel:	f	$\rightarrow$	$0 : f$	$\Rightarrow$	$f \doteq f$ gültig
Aber gilt:	g	$\rightarrow$	$0 : g$	$\Rightarrow$	$f \doteq g$ gültig?

Semantisch gültig, aber Gleichheit unendlicher Objekte im allg. unentscheidbar!

Beispiel:

$h(x) \rightarrow h(x)$
$k(x) \rightarrow k(x)$

Ist $h(0) \doteq (0)$ gültig? Beide Berechnungen terminieren nicht!

$\Rightarrow$ Sinnvoll bei nichtterminierenden TES:
Statt reflexiver Gleichheit: **Strikte Gleichheit** $\equiv$
$\equiv$: Gleichheit auf endlichen Strukturen
$t_1 \equiv t_2$ gültig, falls t_1 und t_2 reduzierbar zu **Grundkonstruktorterm** u
(d.h. u enthält keine Variablen und keine definierten Funktionen)
Obige Beispiele: $f \equiv f, f \equiv g, h(0) \equiv k(0)$: nicht gültig

Strikte Gleichheit $\equiv$ entspricht `==` in Haskell, wobei in Haskell keine freien Variablen zugelasen sind
(Beachte: Falls $f(x) \rightarrow 0$, dann ist $f(x) \equiv f(x)$ gültig!)

Interessanter Aspekt: $\equiv$ ist (im Gegensatz zu $=$) durch ein funktionales Programm definierbar:
$c \equiv c \rightarrow$ `success` (für alle 0-stelligen Konstruktoren c)
$d(x_1,\ldots,x_n) \equiv d(y_1,\ldots,y_n) \rightarrow s_1 \equiv y_1 \& \ldots \& s_n \equiv y_n$ (für alle n-stelligen Konstruktoren d)
`success` & `x` $\rightarrow$ `x`
Dann gilt:
Satz: $s \equiv t$ gültig $\Leftrightarrow s \equiv t \rightarrow^*$ `success` mittels obiger Zusatzregeln
Beachte: $x \doteq x \rightarrow$ `success`: unzulässige Regel in Haskell (weil linke Seite nicht linear)

3.3 Spezialfall: Logikprogrammierung

Wie wir in der Einleitung schon gesehen haben, gibt es zwei wichtige Klassen deklarativer Sprachen:

- Funktionale Sprachen
- Logische Sprachen

Beide können als Spezialfälle von *logisch-funktionalen Sprachen* (d.h. funktionale Programmierung mit freien Variablen und Nichtdeterminismus) gesehen werden:

Funktionale Sprachen:	keine freien Variablen erlaubt
Logische Sprachen:	keine (geschachtelten) Funktionen erlaubt, nur Prädikate

Logikprogramm:

- alle Funktionen haben Ergebnistyp `Success`
- alle Regeln haben die Form "l `= success`" bzw. "l `|` c `= success`"
  (nur positive Aussagen, c ist Konjunktion von Constraints)
  $\Rightarrow$ syntaktische Vereinfachung: Lasse „`= success`“ weg
  im Prolog-Syntax: $p(t_1, \ldots, t_n).$ statt $p\ t_1 \ldots t_n$ `= success`
  l'_0 `:-` $l'_1, \ldots, l'_n.$ statt l_0 `|` l_1 `&...&` l_n `= success`
  (wobei: l_i die Form $p_i\ t_{i1} \ldots t_{in_i}$ und
  l'_i die Form $p_i(t_{i_1}, \ldots, t_{i_{n_i}})$ hat)
  „Literale“
- Erweiterungen:
  - *Extravariablen* in Bedingung (da freie Variablen erlaubt)
  - *nichtlineare linke Seiten* (z.B. „`eq(X,X).`“ zulässig in Prolog)

Extravariablen:

- Notwendig wegen flacher Struktur der Bedingungen
  Bsp.: Statt

```
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)

rev []     = []
rev (x:xs) = rev xs ++ [x]
```

  in Prolog (hier schreibt man "`[X|Xs]`" anstatt "`X:Xs`"):

```
append([], Ys, Ys).   % Hierbei ist Ys Zusatzargument für Ergebnis
append([X|Xs], Ys, [X|Zs]) :- append (Xs, Ys, Zs).
rev([],[]).
```

```
rev([X|Xs], Ys) :- rev(Xs, Rs), append(Rs, [X], Ys).
% hierbei ist Rs Extravariable zum "'Durchreichen"' des Ergebnisses
```

⇒ Übersetzung Funktionen → Prädikate:

- Zusatzargument für Ergebnis ($f :: t_1 \to \ldots \to t_n \Rightarrow f :: t_1 \to \ldots \to t_n \to$ `Bool`)
- „Abflachen“ („flattening“) von geschachtelten Aufrufen
  (hierdurch: Verlust der lazy-Auswertung!)

- Praktisch verwendbar z.B. bei transitiven Abschlüssen:
  Vorfahrrelation in Verwandtschaftsbespiel (vgl. Kap. 3.1)

```
vorfahr v p | v =:= mutter p   = success
vorfahr v p | v =:= vater  p   = success
vorfahr v p | v =:= mutter p1 & vorfahr p1 p = success   where p1 free
vorfahr v p | v =:= vater  p1 & vorfahr p1 p = success   where p1 free
```

(in den letzten beiden Regeln ist `p1` eine Extravariable)

Bedingte Regeln

- Wichtig in Logikprogrammen
  (sonst nur triviale Regeln der Form $p\ t_1 \ldots t_n =$ `success`, $\sim$ relationale Datenbank)

- Rechnen mit bedingten Regeln $l \mid c = r$:
  Falls l „passt“ (bzgl. pattern matching der Unifikation):
  Berechne c: Falls reduzierbar zu success: Ergebnis: r
  sonst: probiere andere Regel

- Formalisierbar durch Transformation $l \mid c = r$ in „unbedingte“ Regel $l =$ `cond` $c\ r$
  wobei `cond` „definiert“ durch: `cond success x = x`
  Beispiel:

```
f x | x=:=0 = 0          f x = cond (x=:=0) 0
f x | x=:=1 = 1    ⇒     f x = cond (x=:=1) 1
```

  (Achtung: Funktioniert nur, falls alle Regeln gleichzeitig (nichtdeterministisch) angewendet werden
  (also nicht in Haskell), da transformierte Regeln evtl. nicht konfluent!)

Reduktion: `f 0` → `cond (0=:=0) 0` → `cond success 0` → `0`

Logikprogramme:

- enthalten i.d.R. freie Variablen
  ⇒ Abarbeitung durch Narrowing

- Regeln haben die Form l_0 `|` l_1 `&...&` l_n `= success`

⇒ Vereinfachung der „`cond`“ -Transformation zu l_0 `=` l_1 `&...&` l_n
hierbei: & definiert durch

```
success & x = x
```

⇒ links→rechts-Auswertung von `&`

⇒ Vereinfachung von Narrowing bei Logikprogrammen zu:

Resolution:
Gegeben: Konjunktion („*Anfrage*“, „*Ziel*“, „*goal*“) l_1 `&...&` l_n
Falls $l_0 = r_0$ (r_0 „`success`“ oder Konjunktion) eine Variante einer Regel ist und σ mgu für l_i ($i > 0$) und l_0, dann ist

$$l_1 \texttt{ \& } \ldots \texttt{ \& } l_n \quad \leadsto_\sigma \quad \sigma(l_1 \texttt{ \& } \ldots \texttt{ \& } l_{i-1} \texttt{ \& } r_0 \texttt{ \& } l_{i+1} \texttt{ \& } \ldots \texttt{ \& } l_n)$$

ein **Resolutionsschritt**.

Somit:

1. Unifiziere erstes Literal in der Anfrage mit linker Seite einer Regel.
2. Falls erfolgreich: Ersetze erstes Literal durch rechte Seite und wende Unifikator an.

Beispiel:

```
p a = success
p b = success
q b = success
```

Anfrage: `p x & q x` $\leadsto_{\{\texttt{x}\mapsto\texttt{a}\}}$ `success & q a` $\leadsto_{\{\}}$ `q a` : Fehlschlag, Sackgasse
$\leadsto_{\{\texttt{x}\mapsto\texttt{b}\}}$ `success & q b` $\leadsto_{\{\}}$ `q b` $\leadsto_{\{\}}$ `success`

Notwendig zur Lösungsfindung: Ausprobieren **aller** Regeln
Realisierung in den meisten existierenden Sprachen:

Backtracking:

- Probiere zuerst erste passende Regel und rechne weiter
- Falls dies in eine Sackgasse führt: Probiere (bei letzter Alternative) nächste passende Regel

Potentielle Probleme. Kein Auffinden von Lösungen bei unendlichen Berechnungen

Beispiel:

```
p a = p a
p b = success
```

Anfrage: `p x` $\leadsto_{\{\texttt{x}\mapsto\texttt{a}\}}$ `p a` $\rightarrow$ `p a` $\rightarrow$ `p a` $\rightarrow$ ...
⇒ Endlosschleife, Lösung $\{\texttt{x} \mapsto \texttt{b}\}$ wird nicht gefunden

Hier trivial, aber häufiges Problem bei rekursiven Datenstruktoren:

```
last (x:xs) e | last xs e = success
last [x]    e | x =:= e   = success
```

Anfrage: `last xs 0` $\leadsto_{\{xs \mapsto (x1:xs1)\}}$ `last xs1 0`
$\leadsto_{\{xs1 \mapsto (x2:xs2)\}}$ `last xs2 0`
$\leadsto \ldots$

$\Rightarrow$ Vorsicht bei Suche nach potentiell unendlichen Strukturen!

Bei endlichen Strukturen kann Suche sinnvolle Programmiertechnik sein:
Beispiel: Färben einer Landkarte:

Färbung mit Rot, Gelb, Grün, Blau, wobei aneinandergrenzende Länder verschiedene Farben haben

1. Aufzählung der Farben:

```
data  farbe = Rot | Gelb | Gruen | Blau


farbe Rot   = success
farbe Gelb  = success
farbe Gruen = success
farbe Blau  = success
```

2. Mögliche Färbung der Darte:

```
faerbung l1 l2 l3 l4 = farbe l1 & farbe l2 & farbe l3 & farbe l4
```

3. Korrekte Färbung: Angrenzende Länder haben verschieden Farben:

```
korrekt l1 l2 l3 l4 = diff l1 l2 & diff l1 l3 & diff l2 l3 &
                      diff l2 l4 & diff l3 l4


diff x y = (x==y) =:= False
```

Gesamtlösung: Anfrage `faerbung l1 l2 l3 l4` & `korrekt l1 l2 l3 l4`

`faerbung l1 l2 l3 l4`: Aufzählung möglicher Lösungen

`korrekt l1 l2 l3 l4`: Prüfen der Lösungen

$\leadsto$ `[l1/Rot, l2/Gelb, l3/Gruen, l4/Rot]`

Programmiertechnik „**Aufzählung des Suchraumes**“ („**generate + test**“)

Allgemeines Schema: *generate* l `&` *test* l
sinnvoll bei endlich (möglichst wenigen) vielen Möglichkeiten für l

3.4 Narrowing-Strategien

Reduktion (funktionale Sprachen): Berechnung von Werten
Strategien: strikt, lazy
Narrowing (logisch-funktionale Sprachen): Berechnung von Werten + Lösungen für freie Variablen

Vollständigkeit (d.h. finde alle Lösungen):
(Hullot-Satz, Kap. 3.2):
Probiere - alle Regeln
- an allen passenden Stellen
Ausprobieren aller Regeln i. allg. nicht vermeidbar:
$f(a) \to b$
$f(b) \to b$

Gleichung: $f(x) = b$
$\to$ 1. Regel: $\sigma = \{x \mapsto a\}$
$\to$ 2. Regel: $\sigma = \{x \mapsto b\}$
unabhängige Lösungen

Ausprobieren aller Positionen: vermeidbar?
Reduktion: ja ($\leadsto$ lazy evaluation)
Narrowing: nicht so einfach wegen freier Variablen
verbesserte Strategien: dieses Kapitel

3.4.1 Strikte Narrowing-Strategien

Problem der Strategie: innerste bzw. äußerste Position evtl. nicht eindeutig

Beispiel:
Regel: $rev(rev(l)) \to l$
Term: $rev(rev(x))$: Welche Position ist „innerste“?
$\leadsto_{1,\ldots,\{\}} x$
$\leadsto_{1,\ldots,\{x \mapsto rev(s_1)\}} rev(x_1)$

Vermeidung durch konstrukturbasierte TES ($\to$ Kap. 2.3):

$\forall l \to r$ gilt: $l = \underset{\uparrow}{f} \underset{\nwarrow}{(t_1,} \quad \ldots, \underset{\nearrow}{t_n)}$ ← „Muster“

Funktion *Konstruktoren und Variablen*

Nicht zulässig:

$rev(rev(l)) \to l$

$(x + y) + z \to x + (y + z)$

- keine echte Einschränkung für praktische Programme:
  funktionale Programme immer konstruktorbasiert
- Nicht konstruktorbasierte TES ≈ Spezifikation, keine effektiven Programme

Im folgenden: Annahme: **alle TES sind konstrukorbasiert**

Def.: Narrowing-Ableitung

$$t_0 \leadsto_{p_1,\sigma_1} t_1 \leadsto_{p_2,\sigma_2} \ldots \leadsto_{p_n,\sigma_n} t_n$$

heißt **innermost** (≈ strikt), falls $t_{i-1}|_{p_i}$ ein Muster ist $(i = 1, \ldots, n)$.

Bsp.: $0 + n \to n$

$s(m) + n \to s(m + n)$

$x + (y + z) \doteq 0 \quad \leadsto_{1\cdot 2,\{y \mapsto 0\}} \quad x + z \doteq 0 \leadsto_{1,\{x\mapsto 0\}} z \doteq 0$ **innermost**

$\leadsto_{1,\{x\mapsto 0\}} \quad y + z \doteq 0 \leadsto_{1,\{y\mapsto 0\}} z \doteq 0$ **nicht innermost**

Innermost Narrowing: - entspricht strikter Auswertung in funkt. Sprachen
- effizient implementierbar

Nachteile: nur eingeschränkt vollständig:

1. Lösungen sind evtl. zu speziell:
   Bsp.: $R: \quad f(x) \to a$
   $f(a) \to a$
   Gleichung: $f(g(z)) \doteq a$
   Mögliche Lösung: $\{\}$ (Identität), da $f(g(z)) \to_R a$
   Aber: nur eine innermost Narrowing-Ableitung:
   $f(\mathbf{g(z)}) \doteq a \leadsto_{1\cdot 2,\{z\mapsto a\}} f(a) \doteq a \leadsto_{a,\{\}} a \doteq a$
   $\Rightarrow$ innermost Narrowing berechnet speziellere Lösung $\{z \mapsto a\}$
2. Fehlschlag bei partiellen Funktionen:
   Bsp.: $R: \quad f(a, z) \to a$
   $g(b) \to b$
   Gleichung: $f(x, g(x)) \doteq a$
   Innermost Narrowing: $f(x, \underline{g(x)}) \doteq s \leadsto_{\{x\mapsto b\}} f(b, b) \doteq a$: Fehlschlag
   Allgemeines Narrowing: $\underline{f(x, g(x))} \doteq a \leadsto_{\{x\mapsto a\}} a \doteq a$: Erfolg!

Vollständigkeit von innermost Narrowing (Fribourg 1985):

- R TES so daß $\rightarrow_R$ konfluent und terminisierend
- $\forall$ Grundterme t gilt: t in Normalform $\Rightarrow$ t enthält nur Konstruktoren
  ($\sim$ alle Funktionen sind total definiert)

Dann ist innermost Narrowing vollständig für alle Grundsubstitutionen, d.h. es gilt:
Falls σ' Grundsubstitution (d.h. $\sigma(x)$ ist Grundterm $\forall x \in Dom(\sigma)$) und Lösung von $s \doteq t$, dann existiert innermost Narrowing-Ableitung $s \doteq t \rightsquigarrow_\sigma^* s' \doteq t'$, ein mgu φ für s' und t' und eine Substitution τ mit $\sigma'(x) \doteq \tau(\varphi(\sigma(x)))$ ist gültig $\forall x \in Var(s \doteq t)$

Viele Funktionen sind total definiert, falls nicht:

Verbesserung von innermost Narrowing für partielle Funktionen:
Innermost Basic Narrowing
Idee: - Erlaube das „Überspringen“ innerer Funktionsaufrufe
- Falls Aufruf übersprungen $\rightarrow$ $\underbrace{\text{ignoriere auch in nachfolgenden Schritten}}_{\text{basic narrowing}}$

Für formale Beschreibung notwendig:
Unterscheidung zwischen auszuwertenden und übersprungenen Funtkionsaufrufen

$\Rightarrow$ Darstellung von Gleichungen als Paar: $\langle E;\ \sigma\rangle \approx \sigma(E)$
(Skelett: E; Umgebung: σ)

- Auswertung: nur Aufrufe in E
- „Überspringen“: verschiebe Aufruf von E nach σ
- Initiale Gleichung $E \approx \langle E; []\rangle$

Bsp.: $R : f($ $a, z) \rightarrow a$
$g(b) \rightarrow b$
Gleichung: $f(x, g(x)) \doteq a$

In neuer Darstellung:

$\langle f(x, g(x)) \doteq a; []\rangle$	$\rightsquigarrow$	$\langle f(x, y) \doteq a; \{y \mapsto g(x)\}\rangle$	„Überspringen“
	$\rightsquigarrow_{\{x \mapsto a\}}$	$\langle a \doteq a; \{y \mapsto g(a), x \mapsto a\}\rangle$	„Innermost Narrowing im Skelett“

Lösung: $\{x \mapsto a\}$

Also: Mit “Überspringen“ ist innermost Narrowing vollständig

Formal: Innermost basic Narrowing besteht aus zwei alternativen Ableitungsschritten:

Narrowing:
1. $p \in Pos(e)$ mit $E|_p$ ist Muster
2. $l \to r$ neue Variante einer Regel
3. σ' mgu für $\sigma(E|_p)$ und l

Dann: $\langle E; \sigma \rangle \rightsquigarrow \langle E[r]_p; \sigma' \circ \sigma \rangle$

„innermost basic Narrowing-Schritt“

Innermost reflection:
1. $p \in Pos(e)$ mit $E|_p$ ist Muster
2. x neue Variable mit $\sigma' = \{x \mapsto \sigma(E|_p)\}$

Dann $\langle E; \sigma \rangle \rightsquigarrow \langle E[x]_p; \sigma' \circ \sigma \rangle$

„Überspringen“

Anmerkung:
- In beiden Fällen kann p auch linkeste Position sein.
- Innermost basic Narrowing vollständig für konfluente und terminierende TES

Nachteil von reinen innermost Narrowing-Verfahren (Unterschied zu innermost Reduktion!): Endlosschleifen bei rekursiven Datenstrukturen (auch für terminierende TES!)

Beispiel:

$$\begin{array}{ll} 0 + n & \to n \\ s(m) + n & \to s(m + n) \end{array} \Rightarrow \text{konfluent, terminierend, total definiert}$$

Innermost Narrowing:

$$x + y \doteq 0 \rightsquigarrow_{\{x \mapsto s(x_1)\}} s(x_1 + y) \doteq 0 \quad \underbrace{\rightsquigarrow_{\{x_1 \mapsto s(x_2)\}} s(s(x_2 + y)) \doteq 0 \rightsquigarrow \ldots}$$

überflüssig, da linke und rechte Seite nie gleich werden (s und 0 verschiedene Konstruktoren)

Verbesserung: (hierbei: $\texttt{Head}(t) = f :\Rightarrow t = f(t_1 \ldots, t_n)$)

Rückweisung: („rejection“):
Falls $s \doteq t$ zu lösende Gleichung und $p \in Pos(s) \cup Pos(t)$ mit $Head(s|_p) \neq Head(t|_p)$ und $Head(s|'_p), Head(t|'_p) \in C$ für alle Positionen $p' \leq p$ (C ist die Menge der Konstruktoren), dann hat $s \doteq t$ keine Lösung ($\Rightarrow$ sofortiger Abbruch mit Fehlschlag)

Auch mit Rückweisungsregel noch viele überflüssige Ableitungen:

$$\begin{array}{lll} \underline{(x + y)} + z \doteq 0 & \rightsquigarrow_{\{x \mapsto s(s_1)\}} & s(\underline{x_1 + y}) + z \doteq 0 \\ & \rightsquigarrow_{\{x_1 \mapsto s(x_2)\}} & s(s(\underline{x_2 + y})) + z \doteq 0 \\ & \rightsquigarrow \ldots & \text{keine Rückweisung, da + definierte Funktion!} \end{array}$$

Verbesserung: Reduziere zur Normalform **vor** jedem Narrowingschritt
$\Rightarrow$ **Normalisierendes Innermost Narrowing**
[Fay 79, Fribourg 85]

Obiges Beispiel:

$$\begin{array}{lll} \underbrace{(x + y) + z \doteq 0}_{in\ Normalform} & \rightsquigarrow_{\{x \mapsto s(x_1)\}} & s(x_1 + y) + z \doteq 0 \\ & \to & s((x_1 + y) + z) \doteq 0 \\ & & \text{Rückweisung: Fehlschlag!} \end{array}$$

Normalisierendes Innermost Narrowing:

- Normalisierung ist deterministischer Prozess:
  Priorität für Normalisierung (**vor** Narrowing)
  ⇒ geringere Laufzeit und Speicherplatzverbrauch
  Beispiel:
  $0 * n \to 0$
  $n * 0 \to 0$
  Gleichung : $x * 0 \doteq 0$
  Innermost Narrowing: $x * 0 \doteq 0 \leadsto_{\{x \mapsto 0\}} 0 \doteq 0$
  $x * 0 \doteq 0 \leadsto_{\{\}} 0 \doteq 0$
  Innermost Narowing mit Normalisierung:$x * 0 \doteq 0 \to 0 \doteq 0$: deterministisch!
- Vermeidung unnötiger Ableitungen
- effizient implementierbar (→ [Hanus 90, 91, 92])
  (analog zu rein funktionalen/rein logischen Sprachen)

Nachteil rein logischer Programme:

- flache Struktur
- keine funktionalen Abhängigkeiten

Voraussetzung für alle innermost-Strategien und auch für Normalisierung: Terminierung des TES
Nachteil: - schwer zu prüfen (unentscheidbar!)
- verbietet bestimmte funktionale Programmiertechniken
(unendliche Datenstrukturen, Kap. 1.6)
⇒ lazy-Strategien auch bei Narrowing

3.4.2 Lazy Narrowing-Strategien

Strikte Strategien werten alle Teilausdrücke aus
⇒ - unvollständig bei nichtterminierenden Funktionen
- i.d.R. nicht optimal
Lazy Strategie: wertet nur aus, $\underbrace{\text{falls es notwendig ist}}_{\text{was bedeutet dies?}}$
Formal schwierig zu definieren, daher:
1. Ansatz: Lazy $\hat{=}$ outermost:

Definition:
Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ heißt **outermost**, falls für alle Narrowing-Schritte $t \leadsto_{p',\sigma'} t''$ gilt: $p' \not< p$ (p' nicht oberhalb von p). Analog: **leftmost outermost**: $p' \not< p$ und p' nicht links von p.

Aber:
Outermost Narrowing ist im allgemeinen unvollständig (auch wenn es nur eine outermost Position gibt: Gegensatz zu Reduktion!)

Beispiel:

$R:$ $f(0,0) \to 0$
$f(s(x),0) \to s(0)$ konfluent, terminierend total definiert
$f(x,s(y)) \to s(0)$

Gleichung: $f(f(i,j),k) \doteq 0$

Innermost narrowing:	$f(f(i,j),k) \doteq 0$	$\leadsto_{1\cdot 1,\{i\mapsto 0,j\mapsto 0\}}$	$f(0,k) \doteq 0$
		$\leadsto_{1,\{k\mapsto 0\}}$	$0 \doteq 0$
	$\Rightarrow$ Lösung: $\{i \mapsto 0, j \mapsto 0, k \mapsto 0\}$		
outermost narrowing:	$f(f(i;j),k) \doteq 0$	$\leadsto_{1,\{k\mapsto s(y)\}}$	$s(0) \doteq 0$
		$\uparrow$	
		einzig möglicher outermost Schritt	

Weitere Restriktionen notwendig (Echahed 1988, Padawitz 1987):
(Outermost) **Narrowing vollständig** (im Sinn Hullet-Satz, Kap. 3.2), falls TES konfluent + terminierend und jeder Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ (bzgl. dieser Strategie) ist **uniform**, d.h. $\forall \varphi$ mit $\varphi(x)$ in Normalform $\forall x \in DOM(\varphi)$ exist. ein Narrowing-Schritt für $\varphi(t) \leadsto_{p,\sigma'} t''$ (d.h. Narrowing-Positionen invariant unter Normalforminstantiieurng)

Beispiel:	Outermost Narrowing nicht uniform, da
	$f(f(i,j),k) \doteq 0 \leadsto_{1\cdot 1,\{k\mapsto s(y)\}} 0 \doteq 0$,
aber	$f(f(i,j),0) \doteq 0 \leadsto_{1\cdot 1,\ldots}$ nicht möglich

Nachteile:

- Uniformität schwer zu prüfen
- harte Einschränkung (stärker als total definiert!)
- Terminierung verhindert bestimmte funktionale Programmiertechniken

Im folgenden: Verzicht auf Terminierungsforderung (aber $\underbrace{\text{konstruktorbasiert und schwach orthogonal}}_{\text{KB-SO}}$)

Beachte: bei nichtterminierenden TES reflexive Gleichheit nicht sinnvoll (vgl. Kap. 3.2)
$\Rightarrow$ im folgenden: strikte Gleichheit $\equiv$ (entspricht "`=:=`" in Curry)

Definition (informell) Eine Narrowing-Position ist **lazy**, falls dies die Wurzel ist oder der Wert an dieser Stelle notwendig, um eine Regel an einer darüberliegenden Position anzuwenden.
(Formale Def.: Moreno/Rodriguez, JLP 1992)

Beispiel:
R wie oben
$f(f(i,j),k)$
1. Äußerses Vorkommen von f = lazy, weil Wurzel
2. Inneres Vorkommen von f = lazy, weil von Wurzelposition bzgl. Regel 1+2 verlangt

Definition Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ heißt **lazy**, falls p eine lazy Position in t ist.

Satz (Moreno/Rodriguez 1992): Lazy Narrowing ist vollständig für KB-SO TES für strikte Gleichungen.

Somit: Lösen von Gleichungen mit unendlichen Strukturen möglich:

Beispiel:
`from` $(n) \to n :$ `from` $(s(n))$
`first` $(0, xs) \to []$
`first` $(s(n), x : xs) \to x :$ `first` (n, xs)
`first` $(x,$ `from` $(x)) \equiv [o]$
$\leadsto_{\{\}}$ `from` $(x, y :$ `from` $(s(y))) \equiv [0]$
$\leadsto_{[x \mapsto s(x_1)]} y :$ `first` $(s_1,$ `from` $(s(y))) \equiv 0 : []$
$\leadsto_{\{\}} y \equiv 0$ 1 `first` $(s_1,$ `from` $(s(y))) \equiv []$
$\leadsto_{[y \mapsto 0]}$ `success` 1
$\leadsto_{\{\}}$ `first` $(s_1,$ `from` $(s(y))) \equiv []$
$\leadsto_{[x_1 \mapsto 0]} [] \equiv []$
$\leadsto_{\{\}}$ `success`

$\Rightarrow$ berechnete Lösung: $[x \mapsto s(0), y \mapsto 0]$

Anmerkung: Alle strikten Strategien + basic Narrowing: Endlosschleife!

Beachte. Für Term t gibt es nicht nur eine lazy Position, sondern evtl. mehrere ($\neq$ funktionale Sprachen)

$\Rightarrow$ Problem: manche lazy Narrowing-Schritte überflüssig bzgl. bestimmter Substitutionen (wg. Interaktion: Variablenbindung $\leftrightarrow$ Positionsauswahl)

Bsp.:R :

$0 \leq n$	$\to$ `True`	$0 + n \to n$
$s(m) \leq$	$\to$ `False`	$s(m) + n \to s(m + n)$
$s(m) \leq s(m)$	$\to m \leq n$	

Anfrage: $x \leq y + z \leadsto_{[x \mapsto 0]}$ `True`

$\uparrow \quad \uparrow$

lazy Positionen $\leadsto_{[y \mapsto 0]} x \leq z \leadsto_{[x \mapsto s(x), z \mapsto 0]}$ `False`

auch möglich $\leadsto_{[y \mapsto 0]} x \leq z \leadsto_{[x \mapsto 0]}$ `True`

Vergleich:
1. Ableitung: berechnete Lösung: $\{x \mapsto 0\}$ Schritte: 1
2. Ableitung: berechnete Lösung: $\{x \mapsto 0, y \mapsto 0\}$ Schritte: 2

$\Rightarrow$ 3. lazy Narrowing Ableitung: überflüssiger Schritt, zu spezielle Lösung:

lazy Narrowing ist nicht lazy!

„Fehler“ in 3. Ableitung: $y + z$ wurde ausgewertet, weil 2. oder 3. Regel für $\leq$ später angewendet werden sollte, d.h. x sollte später eigentlich an $s(x_1)$ (und nicht an 0) gebunden werden

$\Rightarrow$ Vermeidung des Problems durch frühzeitige Bindung von x

Strategie hierfür:
1. Binde x an 0 oder $s(x_1)$
2. Wende, je nach Bindung, 1. $\leq$-Regel an ($x \mapsto 0$) oder Werte $y + z$ aus ($x \mapsto s(s_1)$)

Verbesserte Strategie: **Needed Narrowing** [Antoy/Echahed/Hanus 94]

Grundidee bei der Auswertung des Aufrufs $f(t_1, \ldots, t_n)$:

1. Bestimme ein Argument t_1, dessen Wert von allen Regeln für f verlangt wird (Bei $\leq$: 1. Argument, nicht jedoch 2.)
2. Falls t_1
   - Konstruktor $c(\ldots)$: wähle passende Regel mit diesem Konstruktor
   - Funktionsausruf $g(\ldots)$: werte diesen aus
   - Variable: binde diese an verschiedene Konstruktoren (nicht deterministisch) und mache weiter

Beispiel:

$$
\begin{array}{llll}
x \leq y + z & \overset{Binde}{\longmapsto}_{[x \mapsto 0]} \quad 0 \leq y + z & \rightarrow_R & \texttt{True} \\
\uparrow & & & \\
\text{„verlangt“} & & & \\
\text{„needed“} \quad \overset{Binde}{\longmapsto}_{[x \mapsto s(x_1)]} & s(x_1) \leq \underbrace{y + z}_{*} \quad \overset{Binde}{\longmapsto}_{[y \mapsto o]} & & s(x_1) \leq 0 + z \rightarrow_R s(x_1) \leq z \\
 & & & \qquad\qquad\qquad\qquad\qquad\qquad \uparrow \\
 & & & \overset{Binde}{\longmapsto}_{[z \mapsto 0]} \; s(x_1) \leq 0 \rightarrow_R \texttt{False} \\
 & & & \overset{Binde}{\longmapsto}_{[z \mapsto s \mapsto z_1)]} \cdots \\
 & & \overset{Binde}{\longmapsto}_{[y \mapsto s(y_1)]} & \cdots
\end{array}
$$

* = auswerten, dabei y verlangt

Beachte: obige 3. Ableitung nicht möglich

Beachte: Der Schritt

$$x \leq y + z \quad \leadsto_{\{x \mapsto s(x_1), y \mapsto 0\}} \quad s(x_1) \leq z$$

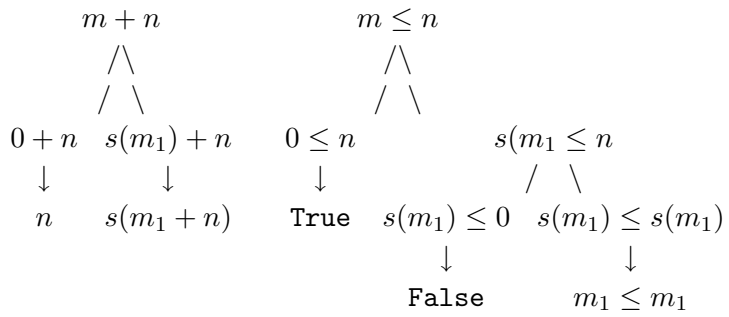
ist kein Narrowing-Schritt im bisherigen Sinn, da $\{x \mapsto s(x_1), y \mapsto 0\}$ kein **mgu** für $y + z$ und linker Regelseite ist!

$\Rightarrow$ Berechnung speziellerer Unifikatoren essentiell!

Problem: Wie findet man die „verlangten“ Argumente?
(Beachte: unentscheidbares Problem für othogonde TES)

Lösung: für induktiv-sequentielle Systeme (vgl. Kap. 2.3) einfach realisierbar mittels definierender Bäume:

Beispiel:Definierende Bäume für $\leq$ und $+$:



z.B. legt Baum für $\leq$ folgende Strategie zur Auswertung von $t_1 \leq t_2$ fest

1. Fallunterscheidung über 1. Argument
   $\Rightarrow$ 1. Argument verlangt, d.h. falls t_1 Funktionsaufruf: werte ihn aus
2. Falls $t_1 = 0$: wende Regel an
3. Falls $t_1 = s(\ldots)$ Fallunterscheidung über 2. Argument:
   - Falls t_2 Funktionsaufruf: werte ihn aus
   - Falls $t_2 = 0$ oder $= s(\ldots)$: wende Regel an
   - Falls t_2 Variable: Binde t_2 an 0 oder $s(\ldots)$ und wende Regel an
4. Falls t_1 Variable: Binde t_1 an 0 oder $s(\ldots)$ und fahre fort mit 2. oder 3.

Formal (hierbei: R induktiv-sequentielles TES):

Definition (Needed Narrowing Schritt) Sei S ein Term, o Position des linkesten äußersten definierten Funktionssysmbols in o (d.h. $S/o = f(t_1, \ldots, t_n)$) und $\mathcal{T}_f$ ein definierender Baum für f. Die needed Narrowing-Strategie λ berechnet Tripel der Form $(p, l \to r, \sigma)$, so daß $s \leadsto_{p,l\to r,\sigma} \sigma(s[r]p)$ ein Narrowing-Schritt ist (beachte: σ nicht unbedingt ein **mgu**). λ ist wie folgt definiert:
$\lambda(s) = \{0 \cdot p, l \to r, \sigma) | (p, l \to r, \sigma) \in \lambda(s|_0, \mathcal{T}_f)\}$
wobei: $\lambda(t, \mathcal{T})$ kleinste Tripelmenge mit

$$\lambda(t,\mathcal{T}) \supseteq \begin{cases} \{(\mathcal{E}, l \to r, mgu(t,l))\} & \text{falls} \mathcal{T} = l \to r \\ \lambda(t, \mathcal{T}_i) & \text{falls} \mathcal{T} = \text{branch}(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_n) \\ & \text{und } t \text{ und pattern}(\mathcal{T}) \text{ unifizierbar} \\ \{(p \cdot p', l \to r, \sigma o \tau)\} & \text{falls} \mathcal{T} = \text{branch}(\pi, p\mathcal{T}_1, \ldots, \mathcal{T}_n), \\ & t|_p = f(\ldots) \text{mit} f \in D,\ \tau = mgu(t, \pi) \\ & \mathcal{T} \text{ def. Baum für } f \text{ und } (p', l \to r, \sigma) \in \lambda(\tau(t_p), \mathcal{T}') \end{cases}$$

Anmerkung: τ bei letzten Alternativen notwendig wegen ortl. Mehrfachvorkommen von Variablen:
$\underline{x} \leqq x + x \longmapsto_{[x \mapsto s(x_1)]} s(x_1) \leqq s(x_1) + s(x_1) \to s(x_1) \leq s(x_1 + s(x_1))$
d.h. $(2, s(m_1) + n \to s(m_1 + n), [x \mapsto s(x_1), m_1 \mapsto x_1, n \mapsto s(x_1)]) \in \lambda(x \subseteqq x + x)$

Anmerkung Definition etwas unhandlich aber needed Narrowing einfach implementierbar analog zu pattern matching (branch-Knoten $\hat{=}$ case-Ausdrücke)

Satz: Needed Narrowing ist vollständig für induktiv-sequentielle TES bzgl. strikten Gleichungen.

Weitere wichtige Eigenschaften von Needed Narrowing:

1. **Optimalität:**
   - Jeder Schritt ist notwendig („needed“), d.h. falls eine Lösung berechnet wird, war kein Schritt überflüssig
   - Needed Narrowing-Ableitungen haben minimale Länge (falls identische Terme nur einmal berechnet werden → sharing).
   - Minimale Lösösungsmenge: Falls σ und σ' Lösung berechnet durch verschiedene Ableitungen, dann sind σ und σ unabhängig (d.h. $\sigma(x)$ und $\sigma'(x)$ nicht unifizierbar für eine Variable x).
2. **Determinismus:**
   Bei Auswertung von variablenfreien Termen ist jeder Schritt deterministisch
   (⇒ funktionale Programme werden determinstisch berechnet, Nichtdeterminsmus nur bei freien Variablen)

Nicht erlaubt bei Needed Narrowing: **Überlappende Regeln**
Beispiel:

```
      'Parralleles Oder'            'Even'
R: True  v x     → True       ev(0)       → True
     x   v True  → True       ev(s(0))    → False
   False v False → False      ev(s(s(n))) → ev(n)
```

Problem: Auswertung von $t_1 \vee t_2$: Zuerst t_1 oder zuerst t_2?

„Verlangtes Argument“ ($\approx$ lazy Position) abhängig von Regel:

1. v-Regel verlangt 1. Argument
2. v˙Regel verlangt 2. Argument

⇒ evtl. unnötige Auswertungen bei “ falscher“ Regel:

`True` $vev\ (n)$ $\leadsto_{\{\}}$ `True`, aber auch:
`True` $v\underline{ev}\ (n)$ $\leadsto_{[n \mapsto s/s(n_1)]}$ `True` $v\ ev(n_1)$
$\leadsto_{[n_1 \mapsto s(s(n_2))]}$ `True` $v\ ev(n_2)$
wobei: $\underline{ev}$ ⇒ lazy Position bzgl. 2. Regel

Wie vermeiden? ⇝ 3 Längsansätze

1. **„Dynamic Cut“** („dynamische Beschneidung des Suchraumes“)
   [Loogen/Winkler PLILP'91]
   Idee: Falls eine Regel anwendbar ohne Instantiierung von freien Variablen,
   dann können Alternativen an dieser Stelle ignoriert werden (aufgrund der Konfluenz)
   Präzise:
   Lemma: Seien $R_1 = l_1 \to r_2$ und $R_2 = l_2 \to r_2$ zwei Regeln und t ein Term
   (mit unterschiedlichen Variablen).
   Falls $\sigma(l_1) = t$ (d.h. t reduzierbar ohne Variableninstantiierung), dann kann Regel R_2
   ignoriert werden, weil R_2 nicht anwendbar oder Anwendung von R_2 ergibt Instanz von R_1 A
   Beispiel: R wie oben
   $\texttt{True}\ v\ ev(n) \leadsto_{\{\},\texttt{True}\ x \to \texttt{True}}\ \texttt{True}$
   $\Rightarrow$ ignoriere Alternativen für diesen Term („dynamic cut“)
   $\Rightarrow$ endlicher Suchraum

   Einfache Verbeserung, aber etwas zu eingeschränkt:

   (a) Abhängigkeit von Regelreihenfolge:
   $ev(n)v\ \texttt{True} \leadsto_{[n \mapsto s(s(n_1))]} ev(n_1)v\texttt{True} \leadsto \ldots$
   Hier hat dynamic cut keinen Effekt! (jedoch bei anderer Reihenfolge der V-Regeln!)

   (b) Dynamic cut nur erfolgreich bei „Top-level-“Anwendbarkeit:
   $(\texttt{True}\ v\ ev(n))v\ ev(m)$ aber möglich: $\to_1 \texttt{True}\ v\ ev(m)$
   $\to_\epsilon \texttt{True}$
   $\uparrow$ kein Narrowing-Schritt notwendig
   hierfür Lemma nicht anwendbar
   $\Rightarrow$ Verbesserung

2. **Lazy Narrowing with Lazy Simplification** [Hanus PLILP'94]
   Idee: Wie bei normalisierendem Narrowing:
   Reduziere den Term vor Narrowing-Schritten, jedoch:

   (a) Benutze eine terminierende Teilmenge von Regeln zur Simplifikation (andernfalls: endlose Simplifikation)

   (b) Wende Reduktionen nur an lazy-Positionen an, d.h. keine vollständige Normalformberechnung (andernfalls: überflüssige Reduktionsschritte)

   Beispiel:
   $(ev(n)v\ \texttt{True})v\ ev(m) \to_1 \texttt{true}\ v\ ev(m)$
   $\to_\epsilon \texttt{True}$
   lazy Positionen

   Im Gegensatz zu simplen lazy Narrowing: keine Auswertung von $ev(n)$ und $ev(m)$!
   Beachte: Reduktion muß immer terminieren, sonst unvollständig:
   Bsp.: Zusätzliche Regel $f(\texttt{True}) \to f(\texttt{True})$
   $xvf(\texttt{True}) \leadsto_{[x \mapsto \texttt{True}]} \texttt{True}$
   Falls neue Regel zur Simplifikation benutzt würde:

$x\ v\ f(\texttt{True} \rightarrow_2 x\ v\ f(\texttt{True})$
$\rightarrow_2 x\ v\ f(\texttt{True})$
:
:
Unvollständig!

Nachteil sequentieller Anwendung von Regeln (d.h. lazy Narrowing wie auch lazy evaluation in Haskell!):
Potentielle Endlosschleife bei Wahl der falschen Regel:

Bsp.: Paralleles v + $f(\texttt{True}) \rightarrow f(\texttt{True})$
$g(\texttt{True}) \rightarrow \texttt{True}$

$f(\texttt{True})v\ g(\texttt{True})$: Welches Argument zuerst auswerten?
1. Argument: $\rightarrow_1 f(\texttt{True})v\ g(\texttt{True}) \rightarrow_1 \ldots$
2. Argument: $\rightarrow_2 f(\texttt{True})v\texttt{True} \rightarrow_\epsilon \texttt{True}$

Analog: $f(x)v\ g(y)$: Binde x oder y? $\Longrightarrow$

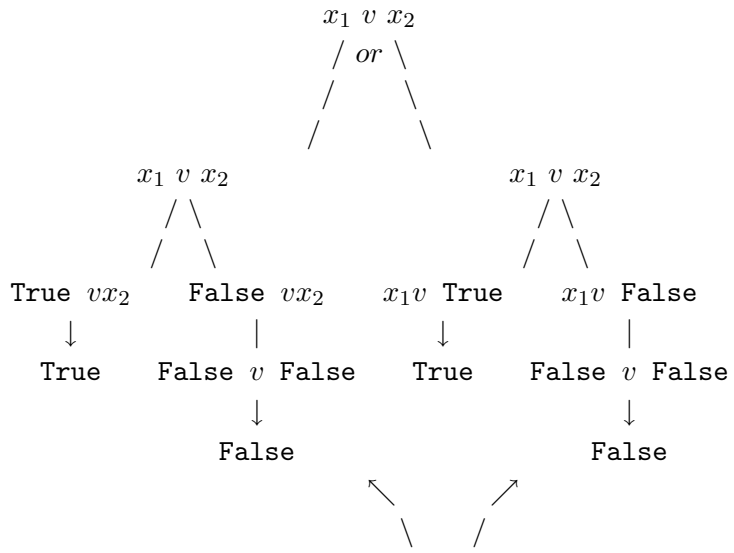
3. **Parallel Narrowing** [Antoy/Echahed/Hanus 97]

Idee:	Reduziere (nach Variablenbindung) mehrere outermost redexes gleichzeitig
Beispiel:	$f(\texttt{True})v\ y(\texttt{True}) \rightarrow_{\{1,2\}} f(\texttt{True})v\ \texttt{True} \rightarrow_\epsilon \texttt{True}$
Im folgenden:	vereinfachte Version
Problem:	Welche Positionen sollen (parallel) reduziert werden?
Lösung:	analog zu needed narrowing: Benutze definierende Bäume
Dazu notwendig:	Erweiterung der Bäume um **Alternativen**

Definition:

1. **Erweiterter definierender Baum**:
   - Regelknoten, Verzweigungsknoten (wie bisher)
   - **Oder-Knoten** or $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$, wobei jedes $\mathcal{T}_i$ erweiterter definierender Baum mit Muster π
2. **Paralleler definierender Baum:**
   Erw. def. Baum, wobei in jedem Oderknoten or $(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ jeder Baum $\mathcal{T}_i$ Verzweigungknoten an der Wurzel hat und diese Verzweigung an paarweise verschiedenen Positionen stattfinden.

Bsp.: paralleler def. Baum für V(graphisch):



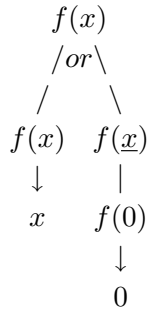
Regeln dürfen auch mehrfach vorkommen

Anmerkungen:

1. Für jedes Konstruktor-basierte TES existiert immer ein erweiterter def. Baum, für KB-SO TES ohne redundante Regeln existiert sogar immer in paralleler def. Baum.

2. TES mit redundanten Regeln: $f(x) \to x$
   $f(0) \to 0$

   Baum:

   nicht parallel

3. Erweiterte Bäume relevant bei bedingten Regeln ($\to$ später)

   $f(0) = 0\ 0$
   $f(x) \mid x \geq 0 = 1$

   nicht redundant, gleiche linke Seite wie oben

Idee für parallele Reduktionsstrategie:

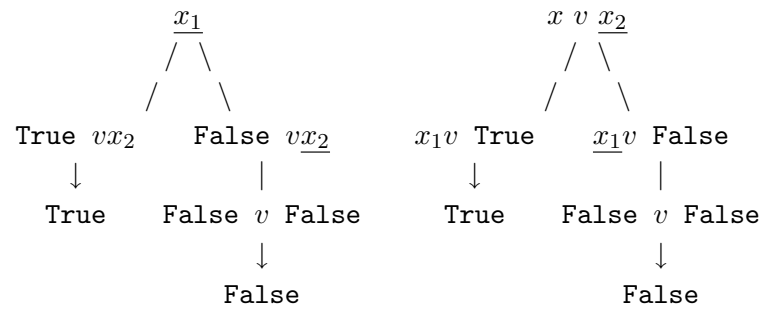
- paralleler definierende Baum $\approx$ Menge von def. Bäumen
- jeder definierende Baum legt Reduktionsschritt fest

$\Rightarrow$ paralleler def. Baum legt Menge von Reduktionsschritten fest.
Führe diese gleichzeitig aus (genauer: nur die unabhängigen)

Definition:
Sequentielle Komponenten $sK(\mathcal{T})$ eines parallelen def. Baumes $\mathcal{T}$:

$sK(l \to r) = \{l \to r\}$
$sK(\texttt{branch}(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_n)) = \{\texttt{branch}(\pi, p, \mathcal{T}_1', \ldots, \mathcal{T}_n') | \mathcal{T}_i' \in \ sK(\mathcal{T}_i), i = 1, \ldots, n\}$
$sK(\texttt{or}(\mathcal{T}_1, \ldots, \mathcal{T}_n)) = \{\mathcal{T} \in sK(\mathcal{T}_i) | 1 \leq i \leq /1\}$

Beispiel: $sK(\leq$ Baum für



Kap. 2.3: Reduktionsstrategie φ definiert bzgl. def. Bäume

Def.: Parallele Reduktionsstrategie $\overline{\varphi}$:
Sei $t = f(t_1, \ldots, t_n)$ und $\mathcal{T}$ paralleler def. Baum für f.
$P = \{p | \mathcal{T}' \in sk(\mathcal{T})$ und $\varphi(t, \mathcal{T}') = p\}$
Sei $P' \subseteq P$ mit: 1. $\forall p \in P \exists\ p' \in P'$ mit $p' \leq p$
2. $\forall p_1, p_2 \in P$ sind p_1 und p_2 disjunkt
(d.h. P' disjunkte äußerste Position aus P)
Dann reduziert $\overline{\varphi}$ gleichzeitig alle Redexe $t|_p$ mit $p \in P'$.

Beispiel:
$(\texttt{True}\ v(\texttt{True}\ v\ \texttt{True}))v(xv(\texttt{False}\ v\ \texttt{False}))$
$\xrightarrow{\overline{\varphi}}_{\{1,2\cdot 2\}} \texttt{True}\ v(xv(\ \texttt{False})$
$\xrightarrow{\overline{\varphi}}_{\epsilon} \texttt{True}$

Wichtige Eigenschaft von $\overline{\varphi}$:

Satz: Falls $t \to^* c$ (c Konstante) dann reduziert $\overline{\varphi}$ den Term t nach c. D.h. $\overline{\varphi}$ ist normalisierend für KB-SO TES (im Gegensatz zu LO-Reduktion).

Weiterer Vorteil von $\overline{\varphi}$ im Vergleich zu lazy Narrowing:
$\overline{\varphi}$ ist deterministisch $\Rightarrow$ Kombiniere $\overline{\varphi}$ mit Narrowing

1. Schritt: Erweiterung von needed Narrowing auf KB-SO TES:

   Definition: Weakly Needed Narrowing: Sei $t = f(t_1, \ldots, t_n)$ und $\mathcal{T}$ paralleler def. Baum für f.
   $\overline{\lambda}(t, \mathcal{T}) = \{(p, l \to r, \sigma) | (p, l \to, \sigma) \in \lambda(t, \mathcal{T}')$ mit $\mathcal{T}' \in sk(\mathcal{T})\}$ definiert alle weakly needed Narrowing-Schritte für t.

   Beispiel:
   Regeln für v und $f(a) \to \texttt{True}$ (R_4)

$t = f(x) v\ f(x)$
Dann: $\overline{\lambda}(t) = \{(1, R_4, [x \mapsto a]), (2, R_4, [x \mapsto a])\}$
$\Rightarrow$ $t \leadsto_{[x \mapsto a]} \texttt{True}\ v\ f(a)$
$t \leadsto_{[x \mapsto a]} f(a)\ v\ \texttt{True}$
sind alle möglichen weakly needed Narrowing-Schritte für t

Satz: Weakly needed Narrowing ist korrekt und vollständig für KB-SO TES bzgl. strikter Gleichheit.

2. Schritt: Kombiniere $\overline{\lambda}$ mit $\overline{\varphi}$:

Definition: Paralleles Narrowing $\overline{\overline{\lambda}}$:
Sei $t = f(t_1, \ldots, t_n)$ und $\mathcal{T}$ par. def. Baum für t.
Sei $S = \{\sigma /\ (p, l \to r, \sigma) \in \overline{\lambda}(t, \mathcal{T})\}$
(Menge aller Narrowing-Substitutionen)
Dann ist $t \overset{\overline{\overline{\lambda}}}{\leadsto}_\sigma t'$ ein paralleler Narrowing-Schritt falls $\sigma \in S$ und $\sigma(t) \xrightarrow{\overline{\varphi}} t'$

Beispiel: Regeln wie oben.
$\underbrace{f(x) v\ f(x)}_{s=\{[x \mapsto a]\}} \overset{\overline{\overline{\lambda}}}{\leadsto}_{[x \mapsto a]} \texttt{True}\ v\ \texttt{True} \leadsto_{\{\}} \texttt{True}$
Einzige mögliche $\overline{\overline{\lambda}}$-Ableitung!

Satz:

1. Paralleles Narrowing ist korrekt und vollständig für KB-SO TES bzgl. strikter Gleichheit.
2. Paralleles Narrowing ist deterministisch (normalisierend) auf Grundtermen.
3. Für induktiv-sequentielle TES ist paralleles Narrowing identisch zu needed Narrowing (und damit optimal).

Anmerkung: Folgende Verbesserungen sind noch möglich:

1. Minimalisierung der Substitutionsmenge S
(z.B. streiche Substitution, falls diese Spezialfall einer anderen ist)
2. Kombination mit Simplifikation wie bei lazy Narrowing möglich

3.4.3 Zusammenfassung

Narrowing
- Erweiterung von Reduktion um Instantiierung freier Variablen
- vollständige Auswertungsstrategie
(findet immer Lösung bei Untersuchung aller Ableitungswege)

Reflexive Gleichheit (Mathematik) und terminierende + konfluente TES:
Bassic Narrowing, bei konstruktor-basiert: innermost (basic) Narrowing + Normalisierung

Strikte Gleichheit ($\approx$ Berechnung von Datenobjekten) + KB-SO TES:

- needed Narrowing (induktiv-sequentielle TES)
- weakly needed Narrowing (nicht ind. seq. TES)

\+
- dynamic cut, (Implementierung einfach)
- Simplifikation, (Impl. aufwendiger)
- parallele Reduktion (Impl. aufwendiger)

Charakteristik von Narrowing (im Gegensatz zu Reduktion):

- Raten von Werten für freie Variablen
- evtl. nichtdeterministische Auswertung von Funktionen ($\leadsto$ unendlich viele Möglichkeiten)

Alternative: Verzögerung der Auswertung statt Raten

3.5 Residuation

Idee von Residuation: Rechne mit Funktionen nur deterministisch $\Rightarrow$
Keine Funktionsauswertung, falls Argumente geraten werden müssen

Falls Raten + Nichtdeterminismus erwünscht
$\Rightarrow$ erlaube nichtdet. Auswertung in **Prädikaten** (=boolesche Funktion)

Beispiel:

$$\begin{array}{cc} 0+n \to 0 & \texttt{nat}\ (0) \to\ \texttt{True} \\ s(m)+n \to s(m+n) & \texttt{nat}\ (s(n)) \to\ \texttt{nat}\ (n) \\ \uparrow & \uparrow \\ \text{Funktion} & \text{Prädikat} \end{array}$$

Anfrage:	$\underbrace{x+0}\equiv s(0)$	$\wedge \underbrace{\texttt{nat}\ (x)}$
	nicht auswerten!	auswerten, Nichtdeterminismus zulässig
	$\leadsto_{[x\mapsto s(x_1)]}$	$\underbrace{s(x_1)+0}\ \equiv s(0) \wedge\ \texttt{nat}\ (x_1)$
		jetzt auswerten
	$\to$	$s(x_1+0) \equiv s(0)\ \wedge\ \texttt{nat}\ (x_1)$
	$\to$	$\underbrace{x_1+0}\equiv 0\ \wedge\ \texttt{nat}\ (x_1)$
		nicht auswerten
	$\leadsto_{[x_1\mapsto 0]}$	$\underbrace{0+0}\equiv 0\ \wedge\ \texttt{True}$
		auswerten
	$\to$	$0 \equiv 0\ \wedge\ \texttt{True}$
	$\to$	$\texttt{True} \wedge \texttt{True}$
	$\to$	$\texttt{True}$

Vorteile von Residuation:

- Funktionen werden „funktional“ verwendet
- unterstützt nebenläufige Programmierung
  (Funktionen $\approx$ Konsumenten, Prädikate $\approx$ Erzeuger)
  Synchronisation über logische Variablen (einfaches Konzept, gestützt auf Logik)
- einfacher Anschluß zu externen Funktionen
  (Bsp.: C-Bibliothek: Behandle externe Funktionen wie normale Funktionen, Aufruf jedoch verzögern bis alle Argumente gebunden sind)
  Bsp.: Arithmetik: Statt s-Terme und explizite Funktionsdefinitionen:
  Zahlkonstanten $\hat{=}$ Konstruktoren (unendlich viele)
  x+y: externe Funktion, wird erst aufgerufen, wenn x und y an Konstanten gebunden sind
  $\leadsto$ Arithmetik in Curry
  Somit: $x \equiv 3 \wedge y \equiv 5 \wedge z \equiv x * y \leadsto [x \mapsto 3, y \mapsto 5, z \mapsto 15]\texttt{True}$
  $x \equiv y + 1 \wedge y \equiv 2 \leadsto [y \mapsto 2, x \mapsto 3]\texttt{True}$

Nachteil von Residuation:

- unvollständig: $x + 0 \equiv s(0)$: Residuation: nicht weiter ausrechenbar
  Narrowing: $\leadsto_{[x \mapsto s(0)]}$ `True`

  Vollständig ausrechenbar nur, falls alle Funktionsargumente im Berechnungsverlauf gebunden werden.
- unendliche Ableitungen durch verzögerte Auswertung von Bedingungen:
  Beispiel:
  Alternative Definition von reverse (als Prädikat):
  `rev` [] [] = `True`.
  `rev` $l(x : xs)/lx + +[x] == l \wedge$ `rev` (lx, xs)
  Unendliche Ableitung durch Anwender der 2. Klausel (mit Verzögerung von ++):

`rev` $[0]l$		
$\leadsto_{[l \mapsto x:xs]}$	$lx + +[x] === [0]$	$\wedge\underline{\texttt{rev}\ (lx, xs)}$
$\leadsto_{[xs \mapsto x_1:xs_1]}$	„	$\wedge\ lx_1 + +[x_1] === lx \wedge \underline{\texttt{rev}\ (lx_1, xs_1)}$
$\leadsto_{[xs_1 \mapsto x_2:xs_2]}$	$\cdots$	

  Ursache: Erzeugung neuer Bedingungen im rekursiven Fall, die alle verzögert werden ($\approx$ passive constraints)
  Wird ++ jedoch mit lazy Narrowing ausgewertet, dann ist der Suchraum endlich (da bei Bindung von lx an zu lange Listen Bedingungen fehlschlagen)
  Somit: Verzögerung des „Ratens“ von Funktionsargumenten nicht immer besser als nichtdeterministisches Raten.

Zusammenfassung			
Narrowing:		**Residuation:**	
+	vollständig	-	unvollständig
-	Nichtdeterminismus bei Funktionen	+	Determinismus bei Funktionen
+	Optimalität bei induktiv seq. Funktionen	-	??
-	Anschluß von externen Funktionen nicht möglich	+	Anschluß von externen Funktionen trivial
		+	nebenläufige Programmierung
$\Longrightarrow$ **Kombination sinnvoll!**			

Exkurs: Nebenläufige Programmierung durch Verzögerung

Bsp.: Bankkonto als nebenläufiges Objekt:

- hat Zustand (Kontostand)
- versteht Nachrichten (einzahlen, auszahlen, Stand)

Modellierung:
Bankkonto $\approx$ Prädikat mit Nachrichtenstrom und Kontostand als Parameter

In Curry:

```
data Nachricht = Einzahlen Int / Auszahlen Int / Stand Int
default residuate -- Verzögere alle Aufrufe, d.h. warte auf Nachrichten
Konto [Nachricht] Int → Bool   wobei: [Nachricht] = Nachrichtenstrom
                                      Int = aktueller Zustand
Konto [] _ = True  – keine Nachrichten mehr
Konto (( Einzahlen x):ns) b = Konto ns (b+x)
Konto ((Auszahlen x):ns) b = Konto ns (b-x)
Konto ((Stand x):ns) b = b === x ∧ Konto ns b
Konto ns 0 0 ∧                ns=== [Einzahlen 100, Einzahlen 50, Stand b]
Erzeuge neues Konto,                 ⇝ b = 150
wartet auf Nachrichten in ns  hier repräsentiert ns ein Server-Objekt, daß man durch
                              Nachrichten aktivieren kann: Schicke Nachricht m an ns
                              ≙ ns===(m:ns')
```

3.6 Ein einheitliches Berechnungsmodell für deklarative Sprachen

1. Narrowing + Residuation haben Vorteile: Wie kombinieren?
2. Bei Residuation: Falls Funktionsaufruf verzögert wird: mit welchem anderen soll man weiterrechnen?

Lösung: einfache Erweiterung definierender Bäume

Lazy evaluation:

- Vorteilhaft bei funktionaler und logischer Programmierung:
  - Rechnen mit unendlichen Datenstrukturen
  - Vermeidung unnötiger Berechnungen
  - normalisieren
- Präzise Beschreibung mit
  - case-Ausdrücken (funktionale Programmierung)
  - definierenden Bäumen (case-Ausdrücke + Parallelismus/oder-Knoten)
    + geeign. für Narrowing, aber Residuation?

Unterschied Narrowing $\leftrightarrow$ Residuation:
Falls verlangtes Funktionsargument freie Variable:

Narrowing:	binde Variable
Residuation:	verzögere Aufruf

Mache diesen Unterschied explizit in branch-Knoten:
Erweiterte branch-Knoten:

`branch` $(\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$:	wie `branch` $(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)$. wobei $r \in$ {`rigid, flex`} (`rigid` = verzögern, `flex` = binde Variablen)

`flex` $\approx$ wie bisher (vgl. needed narrowing)
`rigid`: Ergebnis der Auswertung: `suspend` (spezielle Konstante, zeigt Verzögerung an)

Formal: Erweitere Def. von λ (Kap. 3.4.3) um:

$\lambda(t, \mathcal{T}) =$ `suspend`	falls $\mathcal{T} =$ `branch` $(\tau, p, rigid, \mathcal{T}_1, \ldots calT_n)$ und $t\|_p$ ist Variable

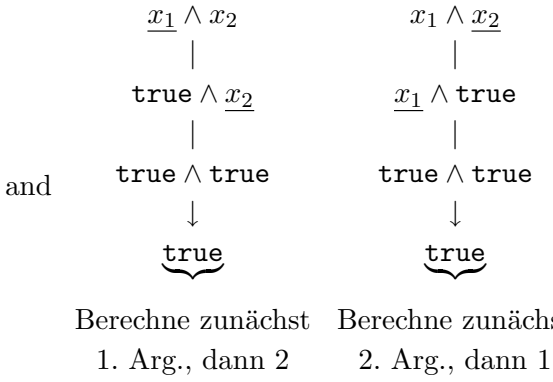
Frage: Wie weiterrechnen, falls Ergebnis = `suspend`?

Obiges Beispiel: $\underbrace{x + 0 \equiv s(0)1}$ $\underbrace{\texttt{nat}(x)}$
Auswertung=`suspend`$\Rightarrow$ hier weitermachen
Definition von 1: `true` 1 `true` $\rightarrow$ `true`
$\Rightarrow$ 1. Zur Auswertung von 1 müssen **beide** Argumente ausgewertet werden.
2. Falls Auswertung des 1. Argumentes suspendiert $\Rightarrow$ werte 2. Argument aus

Präzisierung durch Erweiterung def. Bäume:

Und-Knoten and $(\mathcal{T}_1, \mathcal{T}_2)$:	wie (`or`$\mathcal{T}_1, \mathcal{T}_2$), jedoch enthalten $\mathcal{T}_1$ und $\mathcal{T}_2$ die gleichen Regeln, jedoch andere Verzweigungsknoten

Beispiel:



and

Berechne zunächst 1. Arg., dann 2

Berechne zunächst 2. Arg., dann 1

Und-Knoten:
- Prinzipiell nicht notwendig (vgl. Kap. 3.4.3)
- zeigen ptentiell nebenläufige Berechnung an
- Strategie (sequentielle Simulation der Nebenläufigkeit):
  $\mathtt{and}(\mathcal{T}_1, \mathcal{T}_2)$: - Rechne erst wie mit $\mathcal{T}_1$
  - Falls dies suspendiert: rechne wie mit $\mathcal{T}_2$
  Beispiel: siehe Def. von 1 und Berechnung von $x + 0 \equiv s(0) \wedge \ \mathtt{nat}(x)$!

Gesamtstrategie

Definition:
Antwortausdrücke: Paare $\sigma[]e$
σ = Substitution (bisher berechnete Antwort) ⇒ „logischer Anteil“
e = Ausdruck (bisher berechneter Ausdruck) ⇒ „funktionaler Anteil“
$\sigma[]e$ **gelöst**: ⇔ e enthält keine Funktionsaufrufe
disjunktive Ausdrücke: $\sigma_1[]e_1 v \ldots v \sigma_n[]e_n$ (≈ Nichtdeterminismus)

Beispiel:
$p(a) \rightarrow \mathtt{True}$
$p(b) \rightarrow \mathtt{True}$
Berechnungsschritte:
$p(b) \rightsquigarrow$ `True` (deterministisch)
$p(x) \rightsquigarrow \underbrace{[x \mapsto a]\ []\mathtt{True}\ \ v[x \mapsto b]\ []\mathtt{True}}$ (nichtdeterministisch)

Curry-Notation: $\{x = a\}\mathtt{True}|\{x = b\}\mathtt{True}$

Berechnungsschritt: Transformation auf disjunktiven Ausdrücken
Sei $\sigma_1[]e_1 v \ldots v \sigma_k[]e_k$
Sei $\sigma_1[]e_1$ ungelöst (sonst nehme nächsten ungelösten Ausdruck) und $e_1 = f(t_1, \ldots, t_n)$ (sonst nehme linkesten äußersten Funktionsaufruf in e_1)
Dann ersetze $\sigma_1[]e_1$ durch $\sigma_1' \circ \sigma_1[]e_1' v \ldots v \sigma_m' \circ \sigma_1[]e_m'$, wobei $cs(e_1, \mathcal{T}) = \sigma_1'[]e_1' v \ldots v \sigma_m'[]e_m'$
wobei: $\mathcal{T}$ = definierender Baum für f

cs: Ausdrücke x def. Bäume → disjunktive Ausdrücke v {suspend} definiert durch

$$
\begin{aligned}
cs(e, l \to r) &= []\Box\sigma(r) \text{ wobei } \sigma(l) = e \text{ (wende Regel an)([] = Identität)} \\
cs(e, \texttt{ and } (\mathcal{T}_1, \mathcal{T}_2)) &= \begin{cases} cs(e, \mathcal{T}_1) & \text{falls } cs(t, \mathcal{T}_1) \neq \texttt{suspend} \\ cs(e, \mathcal{T}_2) & \text{sonst} \end{cases} \\
cs(e. \texttt{ or } (\mathcal{T}_1, \ldots \mathcal{T}_k)) &= \begin{cases} \bigvee_{i=1}^{k} cs(e, \mathcal{T}i) & \text{falls } cs(e, \mathcal{T}_i) \neq \texttt{ suspend } , E1, \ldots, k) \text{ (vereinige Alternativ} \\ \texttt{suspend} & \text{sonst} \end{cases}
\end{aligned}
$$

$cs(e, \texttt{ branch } (\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_n)$

$$
= \begin{cases} cs(e, \mathcal{T}_i) & \text{falls } e|_p = c(\ldots) \text{ und pattern } (\mathcal{T}_i)|_p = c(\ldots) \\ \emptyset & \text{falls e—p = c(...) (und pattern } \neq c(\ldots) \forall i = 1, \ldots, k \\ \texttt{suspend} & \text{falls } e|_p = x \text{ und } r = \texttt{ rigid} \\ \bigvee_{i=1}^{k} \texttt{ compose } (\sigma_1(e), \mathcal{T}_i, \sigma_i) & \text{falls } e|_p = x, r = \texttt{flex } , \sigma_i = [x \mapsto \texttt{ pattern } (\mathcal{T}_i)|_p] \\ \texttt{replace } (e, p, cs(t)_p, \mathcal{T}') & \text{falls } e|_p = f(\ldots) \text{ und } \mathcal{T}' \text{ def. Baum für } f \end{cases}
$$

wobei:

$$
\texttt{compose } (e, \mathcal{T}, \sigma) = \begin{cases} \sigma\Box e & \text{falls } cs(e, \mathcal{T}) = \texttt{ suspend} \\ \sigma_1 \circ \sigma\Box e_1 v \ldots v \sigma_n \circ \sigma\Box e_n & \text{falls } cs(e, \mathcal{T}) = \sigma_1\Box e_1 v \ldots v \sigma_n \Box e_n \end{cases}
$$

$\texttt{replace } (e, p, \texttt{suspend }) = \texttt{suspend}$
$\texttt{replace } (e, p, \sigma_1\Box e_1 v \ldots v \sigma_n \Box e_n) = \sigma_1\Box\sigma_1(e)[e_1]_p \; v \ldots v \sigma_n\Box\sigma_n(e)[e_n]_p$

Eigentschaften der Strategie:

1. **Korrektheit:** Jeder Teil einer Disjunktion enthält korrekte Antwort
2. **Vollständigkeit:** Jede korrekte Antwort in einem Teil der Disjunktion enthalten; falls keine rigid-branch-Knoten: jede korrekte Antwort wird vollständig berechnet

Dies ist die Auswertungsstrategie von Curry:

- Definierende Bäume werden automatisch erzeugt (analog zu links-rechts Pattern-matching in Haskell)
- Def. Bäume können auch explizit annotiert werden:
  `add eval 1:flex.`
  (branch über 1. Argument), `flex`= flexible Verzweigung
  `add(z,N) = N.`
  `add(s(M),N) = s(add(M,N)).`

  ⇒ Taste Curry

- Alle branch-Knoten sind
  `flex` = bei Prädikaten (Ergebnistyp Bool)
  `rigid`= sonst

- Compiler-Pragmas:
  default narrow: Im folgenden sollen alle branch-Knoten flex sein
  default residuate: Im folgenden sollen alle branch-Knoten rigid sein
  default standard: Standard (wie oben)

Dieses Modell subsumiert andere bekannte Berechnungsstrategien:

Strategie	Einschränkungen auf def. Bäumen
Needed Narrowing	nur flexible branch-Knoten und Regeln
Weakly needed narrowing (**Babel**)	nur Regeln, flexible branch- und Oder-Knoten
einfaches lazy Narrowing und Resolution (**Prolog**)	für jede Regel ein Baum mit Regel und flexiblen branch-Knoten, alle verbunden mit Oder-Knoten Beispiel $p(a,x) \to \ldots$ $p(x,b) \to \ldots \Rightarrow$ $p(c,x) \to \ldots$ $or($ $p(x_1,x_2)$, $p(x_1,x_2)$, $p(x_1,x_2))$ $\vert$ $\vert$ $\vert$ $p(a,x_2)$ $p(x_1,b)$ $p(c,x_2)$ $\downarrow$ $\downarrow$ $\downarrow$ $\ldots$ $\ldots$ $\ldots$
lazy reduction (Funktionale Sprachen, (**Haskell**)	def. Bäume mit links-rechts Pattern-matching initialer Ausdruck hat keine freien Variablen
Residuation (**Life, Escher, Oz**)	flex-branch für Prädikate rigid-branch sonst

3.7 Erweiterungen des Basismodells

3.7.1 Gleichheit

Notwendig wegen Nichtterminierung:
Strikte Gleichheit
Offen: Gleichheit zwischen freien Variablen: $\underbrace{\text{Binden oder verzögern!}}_{\text{Je nach Anwendung}}$

1. **Testen** der Gleichheit zweier Ausdrücke: e˙1==e˙2 ($\sim$ Haskell)
   def. durch $cx_1 \ldots x_n == y_1 \ldots y_n \to x_1 == y_1 \&\& \ldots \&\& x_n == y_n$
   $c == c \to$ `True` (0-stelliger Konstr. c)
   $cx_1 \ldots x_n == d\ y_1 \ldots y_n \to$ `False` $\forall$ Konstruktoren $c \neq d$
   `True` $\&\& x \to x$
   `False` $\&\& x \to$ `False` $\Rightarrow$ sequentielle Konjunktion

Anmerkungen:

(a) == rigid in beiden Argumenten ⇒ verzögere bei unbekannten Argumenten

(b) Argmente (auswertbar zu) Grundkonstruktortermen
⇒ Ergebnis `True` oder `False`: Verwendung in Fallunterscheidung

```
... if f x == 0 then ... else ...
 wobei  if True then x elxe y → x
        if False then x else y → y
```

2. Gleichheit als **Bedingung (Einschränkung, Constraint)**, die erfüllt sein muß (bzw. gelöst werden muß): $e_1 === e_2$
Anwendung in Bedingungen von Regeln, kann (und soll!) Varaible binden.
Beispiel: `last l |` $\underbrace{\texttt{xs ++ [e]===l}}$ `= e`
Löse Gleichung und binde dadurch e

Definition von ===:
$c === c \rightarrow$ `True` (0-stelliger Konstr. c)
$c\ x_1 \ldots x_n === c\ y_1 \ldots y_n \rightarrow x_1 === y_1 /\backslash \ldots /\backslash x_n === y_n$ ∀ n-stelliger Konstr.
`True` ∧ `True` → `True` (nebenläufige Konjunktion, vgl. Kap. 3.5)

Anmerkungen:

1. === flexibel in beiden Argumenten ⇒ binde Variablen
2. $e_1 === e_2$ liefert nur `True` („partielles“ Prädikat)
⇒ Verwendung in Bedingungen (s.u.), nicht in Fallunterscheidung
3. $x === y$: statt Bindung von x an y: unendlich viele Lösungen der Form $[x \mapsto t, y \mapsto t]$ mit t Grundkonstruktorterm

⇒ Verbesserung: statt $[x \mapsto t, y \mapsto t]$ („x und y haben gleichen Wert“) $[x \mapsto y]$ („x und y gleich, aber Wert noch nicht festgelegt“)

„Meta“definition von ===:

1. 1. und 2. Argument zur Kopfnormalform ausrechnen („branch übeer 1+2“)
2. - $x === y$: binde x an y, d.h. Ergebnis: $[x \mapsto y]$
   - $x === c(t_1, \ldots, t_n)$: Falls x höchstens nur in Funktionsaufrufen in $t_1, \ldots, t_n$ vorkommt (sonst: Fehlschlag wegen occurcheck):
     binde $[x \mapsto c(x_1, \ldots, x_n)]$ und löse $x_1 === t_1 \wedge, ldots \wedge x_n === t_n$
     wobei x_1 und x_n = neue Variablen
   - $c(t_1, \ldots, t_n) === x$: Löse $x === c(t_1, \ldots, t_n)$
   - $c(s_1, \ldots, s_n) === c(t_1, \ldots, t_n)$: Löse $s_1 === t_1 \wedge \ldots \wedge s_n === t_n$

Anmerkungen:

1. $s === t$ mit s, t Konstruktorterme: Berechnung des mgu von s und t
2. $s === t$ allgemein: mgu-Berechnung mit gleichzeitiger Funktionsauswertung

3.7.2 Bedingte Regeln

$l|c = r$: Falls Bedingung $\underbrace{c \text{ beweisbar}}_{\text{ableitbar zur True}}$: Wende $\underbrace{\text{Regel } l = r}_{\text{mit Variablenbindung aus c!}}$ an.

Implementierung einfach:
Interpretiere $l|c = r$ als Regel $l = \underbrace{\texttt{cond}\ (c, r)}_{\text{bedingter Ausdruck}}$
wobei: $\texttt{cond}\ (\texttt{True}\ , x) = x$

Beispiel:

$$\begin{array}{lll} f\ x & |\ x === a & = a \\ & |\ x === b & = b \end{array} \Leftrightarrow \begin{array}{ll} f\ x & = \texttt{cond}\ (x === a, a) \\ f\ x & = \texttt{cond}\ (x === b, b) \end{array}$$

Berechnung von $f\ x$:

$$\begin{array}{lll} f(x) & \to \texttt{cond}(\underline{x === a},)v\ \texttt{cond}(x === b, b) & \\ & \to [x \mapsto a]\texttt{cond}(\texttt{True}, a) & v\ \texttt{cond}(x === b, b) \\ & \to [x \mapsto a]a & v \\ & \to [x \mapsto a]a & v\ [x \mapsto b]\ \texttt{cond}(\ \texttt{True}, b) \\ & \to [x \mapsto a]a & v\ [x \mapsto b]b \end{array}$$

Anmerkung: Falls c Extravariablen enthält, dann ist neue Regel keine „normale“ Regel mehr
$f\ x|p\ x\ y = 0 \rightsquigarrow f\ x = \ \texttt{cond}\ (p\ x\ y, 0)$ wobei y= Extravariable
$\Rightarrow$ Korrektheit + Vollständigkeit aufwendiger ($\rightsquigarrow$ Arbeiten von Middeldorp)

3.7.3 Funktionen höherer Ordnung

Implementierung der Funktionsapplikation $f\ t_1 \ldots t_m$:

- Falls f definierte m-stellige Funktion: Werte wie üblich aus
- Falls f Variable oder n-stellig mit $m < n$:
  Verzögere Auswertung (analog zu Residuation)
  (Alternative: Rate passende Funktion $\rightsquigarrow$ Unifikation höherer Ordnung, aufwendig, unentscheidbar aber aufzählbar)

Realisierung: Sei @ Funktionsapplikation

Erweitere Auswertungsstrategie cs (Kap. 3.6) wie folgt:

$$\begin{array}{lll} cs(x@e) & = \texttt{suspend} & \text{(verzögere unbekannte Apllikation)} \\ xs(f(t_1, \ldots, t_m)Qe) & = f(t_1, \ldots, t_m, e) & \text{(erweitere partielle Applikation)} \end{array}$$

falls f n-stellig und $m < n$

Beispiel:
$(+@1)12) \to \ +(1)@2 \to \ +(1, 2) \to 3$

Zusammenfassung

Deklarative Sprachen:

- definiere Eigenschaften der Objekte
- abstrahiere vonAusführungsdetails (insbesondere: Speicherverwaltung)
- mehr Freiheitsgrade bei Ausführung (aber: konkrete Sprache legt Strategie fest!)
- ⇒ „gute“ Strategien wichtig

In dieser Vorlesung:

- Funktionale Sprachen: Reduktionsstrategien
- (Funktionale) Logische Sprachen: Reduktionsstrategien + Variablenbindung
- Aspekte für flexible, wiederverwendbare SW-Entwicklung:
  - Funktionen höherer Ordnung
  - flexible Typsysteme
- Aspekte für zuverlässige und wartbare Software:
  - lokale Definition (Gleichungen, Pattern Matching)
  - keine Seiteneffekte bzw. explizite globale Effekte

    I/O in dekl. Sprachen? ⇒ monadische I/O
    (I/O nur auf Top-level, Top-level = Sequenz von I/O-Operationen)