## Anderes Problem: Vermeidung mehrfacher Durchläufe einer Datenstruktur

- Angenommen, wir wollen den Durchschnitt einer Liste berechnen:

```
average :: [Int] → Int
average xs = div (sum xs) (length xs)
```

mit:

```
sum [ ]      = 0
sum (x : xs) = x + sum xs
```

und:

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

- Dabei würden wir gern das doppelte Durchlaufen der Liste vermeiden.

- Eine Idee:

```
average' :: [Int] → Int
average' xs = let (s, n) = sumLength xs
              in  div s n
```

wobei:

```
sumLength xs = (sum xs, length xs)
```

## Anderes Problem: Vermeidung mehrfacher Durchläufe einer Datenstruktur

- Nun Versuch der Herleitung einer alternativen Definition für sumLength:

```
sumLength [ ]      =  ???
sumLength (x : xs) =  ???
```

- Postuliert:

```
sumLength xs = (sum xs, length xs)
```

- Herleitung:

```
sumLength [ ]      = (sum [ ], length [ ])
                   = (0, 0)

sumLength (x : xs) = (sum (x : xs), length (x : xs))
                   = (x + sum xs, 1 + length xs)
                   = (x + s, 1 + n)
                     where (s, n) = sumLength xs
```

## Anderes Problem: Vermeidung mehrfacher Durchläufe einer Datenstruktur

Insgesamt, statt:

```
average :: [Int] → Int
average xs = div (sum xs) (length xs)
```

```
sum [ ]        = 0
sum (x : xs) = x + sum xs
```

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

nun:

```
average' :: [Int] → Int
average' xs = let (s, n) = sumLength xs
              in  div s n
```

```
sumLength [ ]        =  (0, 0)
sumLength (x : xs)  =  (x + s, 1 + n)
                       where (s, n) = sumLength xs
```

# Anderes Problem: Vermeidung mehrfacher Durchläufe einer Datenstruktur

Oder aber:

- Da foldr schon einmal so praktisch war, vielleicht kann es auch hier helfen?

- Schließlich könnten wir ja auch schreiben:

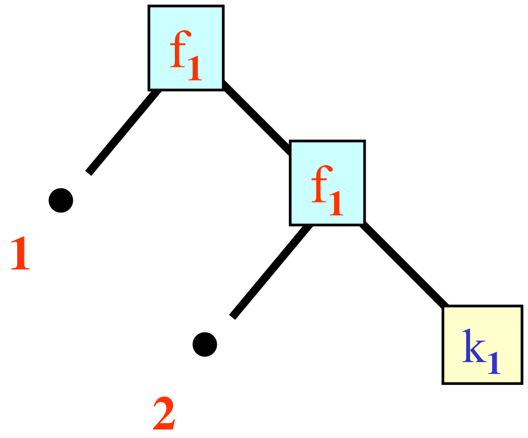
```
average' :: [Int] → Int
average' xs = let (s, n) = (sum xs, length xs)
              in  div s n
```

mit: `sum = foldr (+) 0` und: `length = foldr (\_ y → 1 + y) 0`

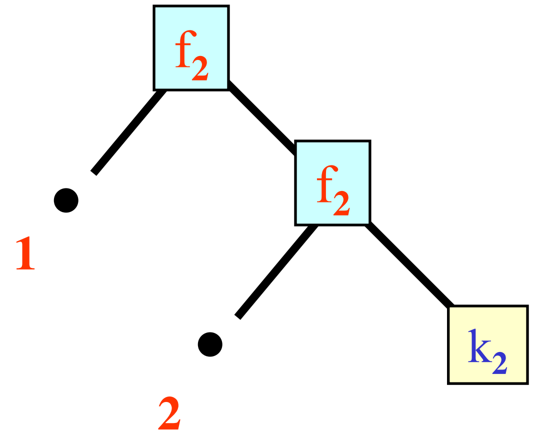
- Und vielleicht gibt es ja eine allgemeine Regel, die uns erlaubt, aus einem Ausdruck der Form ($\text{foldr } f_1\ k_1\ xs$, $\text{foldr } f_2\ k_2\ xs$) einen einzelnen Aufruf von foldr zu machen?

# Ideen für eine allgemeine Tupling-Regel

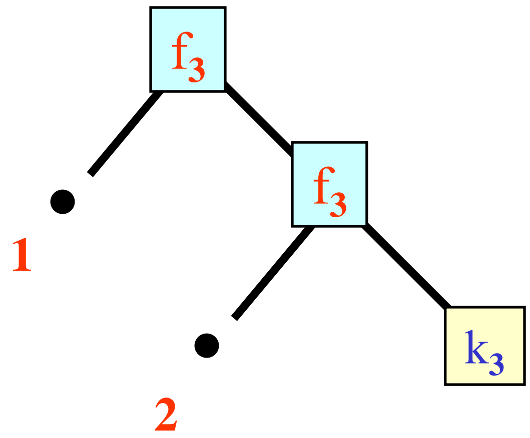
Wie könnte man das Paar von




und



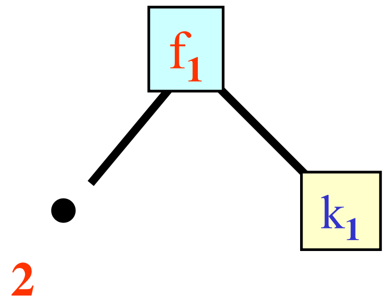

durch nur ein



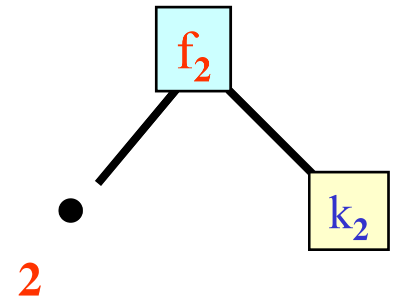

ausdrücken?

## Ideen für eine allgemeine Tupling-Regel

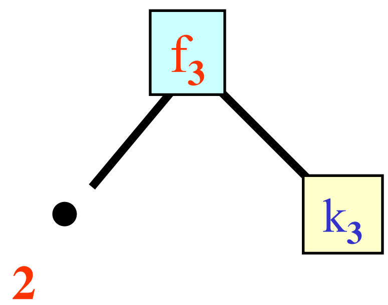
- Noch einfacher, das Paar von




und




durch nur ein




ausdrücken,

- bzw. sogar nur das Paar von

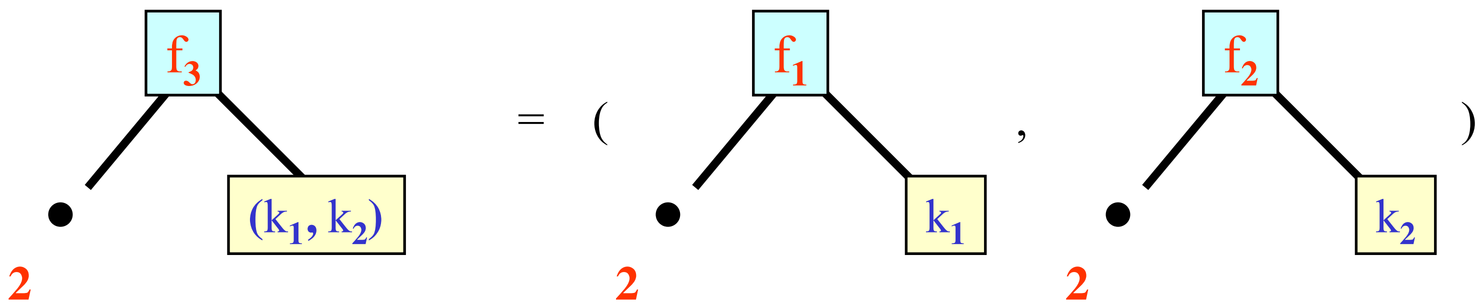
$\boxed{k_1}$ und $\boxed{k_2}$

durch ein $\boxed{k_3}$ ?

# Ideen für eine allgemeine Tupling-Regel

- Aha, offenbar $k_3 = (k_1, k_2)$ !

- Also noch zu leisten: ein $f_3$ finden, so dass




- Idee: $\boxed{f_3\ x\ (y, z) = (f_1\ x\ y,\ f_2\ x\ z)}$

- In der Tat, allgemein:

$$\boxed{(\text{foldr}\ f_1\ k_1\ xs,\ \text{foldr}\ f_2\ k_2\ xs) = \text{foldr}\ (\backslash x\ (y, z) \rightarrow (f_1\ x\ y,\ f_2\ x\ z))\ (k_1, k_2)\ xs}$$

# Anwendung der allgemeinen Tupling-Regel

$$(\text{foldr } f_1\ k_1\ xs, \text{foldr } f_2\ k_2\ xs) = \text{foldr } (\backslash x\ (y, z) \rightarrow (f_1\ x\ y, f_2\ x\ z))\ (k_1, k_2)\ xs$$

- Folglich, Transformation von:

```
average' :: [Int] → Int
average' xs = let (s, n) = (sum xs, length xs)
              in  div s n
```

mit:

```
sum = foldr (+) 0
```

und:

```
length = foldr (\_ y → 1 + y) 0
```

in:

```
average' :: [Int] → Int
average' xs = let (s, n) = foldr (\x (y, z) → (x + y, 1 + z)) (0, 0) xs
              in  div s n
```

# Exkurs: Alternative Datenstrukturen

- Eine günstige Alternative zur Transformation von Programmen ist die Verwendung optimierter Datenstrukturen.

- Um etwa die ineffizienten Vorkommen von (++) zu vermeiden, zum Beispiel bei reverse und flatten, könnte man auch einen neuen Typ verwenden:

```
type L a = …
```

mit Operationen:

```
nil :: L a

cons :: a → L a → L a

app :: L a → L a → L a

snoc :: L a → a → L a

toList :: L a → [a]
```

Forderung: bis auf toList haben alle Operationen nur konstanten Aufwand

# Exkurs: Alternative Datenstrukturen

- Dann könnte etwa:

```
flatten :: BinTree a → [a]
flatten Empty        = [ ]
flatten (Node l a r) = flatten l ++ [a] ++ flatten r
```

optimiert werden durch Umschreiben in:

```
flatten :: BinTree a → L a
flatten Empty        = nil
flatten (Node l a r) = flatten l `app` (cons a nil) `app` flatten r
```

- Aber was wäre denn eine einfache geeignete Implementierung für

```
type L a = …
```

und die Operationen?

# Exkurs: Alternative Datenstrukturen

Unter Benutzung eines Funktionstyps als Datenstruktur:

```
type L a = [a] → [a]

toList :: L a → [a]
toList f  =  f [ ]

nil :: L a
nil  =  id

cons :: a → L a → L a
cons x f  =  (x :) . f

snoc :: L a → a → L a
snoc f x  =  f . (x :)

app :: L a → L a → L a
app f g  =  f . g
```

Macht Sinn wegen:

```
   toList (cons x f)
=  cons x f [ ]
=  ((x :) . f) [ ]
=  (x :) (f [ ])
=  x : toList f
```

## Exkurs: Alternative Datenstrukturen

- Dann entspricht:

```
flatten :: BinTree a → L a
flatten Empty        = nil
flatten (Node l a r) = flatten l `app` (cons a nil) `app` flatten r
```

einfach:

```
flatten :: BinTree a → [a] → [a]
flatten Empty        = id
flatten (Node l a r) = flatten l  .  ((a :) . id)  .  flatten r
```

bzw.:

```
flatten :: BinTree a → [a] → [a]
flatten Empty        = \as → as
flatten (Node l a r) = \as → flatten l (a : flatten r as)
```

# Deskriptive Programmierung

## Motivation/Einführung Logisches Programmieren

## Algorithm = Logic + Control

*„An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used. The logic component determines the meaning of the algorithm whereas the control component only affects its efficiency.*

*The efficiency of an algorithm can often be improved by improving the control component without changing the logic of the algorithm. We argue that computer programs would be more often correct and more easily improved and modified if their logic and control aspects were identified and separated in the program text.“*

Robert Kowalski, 1979

# Geschichte von Prolog

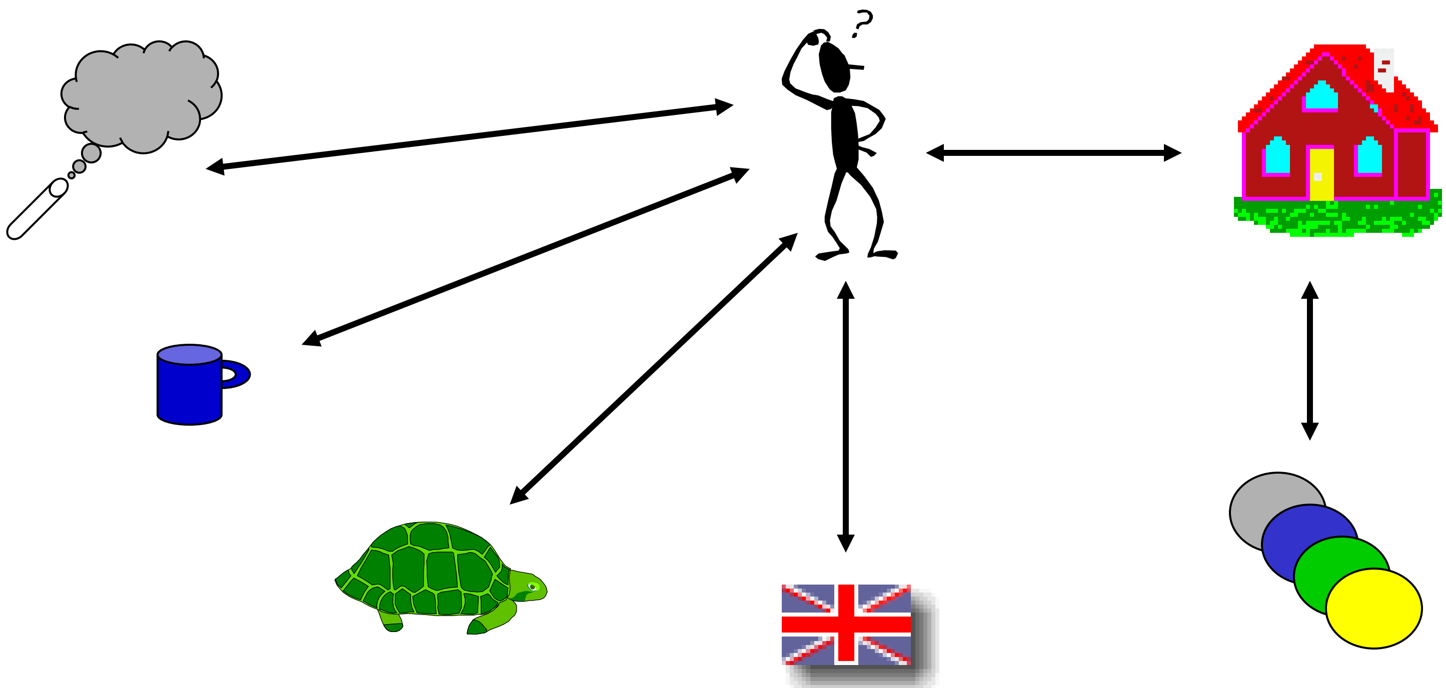
- Prolog steht für “Programming with logic”.
- Es ist die am weitesten verbreitete logische Programmiersprache.
- Ein wenig Geschichte zu Prolog:

| | |
|---|---|
| 1965: | John Alan Robinson legt theoretische Grundlagen für Theorembeweiser mit dem Resolutionskalkül. |
| 1972: | Alain Colmerauer (Marseilles) und seine Gruppe entwickeln Prolog. |
| Mitte 70er: | David D.H. Warren baut den ersten lauffähigen Compiler, wonach sich der spätere DEC-10 Standard richtet (Edinburgh-Standard). |
| 1981–92: | 5th Generation Computer Project in Japan (machte Prolog bekannt) |

# Ein berühmtes logisches Puzzle als deskriptiv spezifiziertes Problem

„There are five houses, each of a different color and inhabitated by a man of a different nationality with a different pet, drink and brand of smokes …“

(das „Zebra Puzzle“ oder „Einstein‘s Riddle“, s. http://en.wikipedia.org/wiki/Zebra_Puzzle)

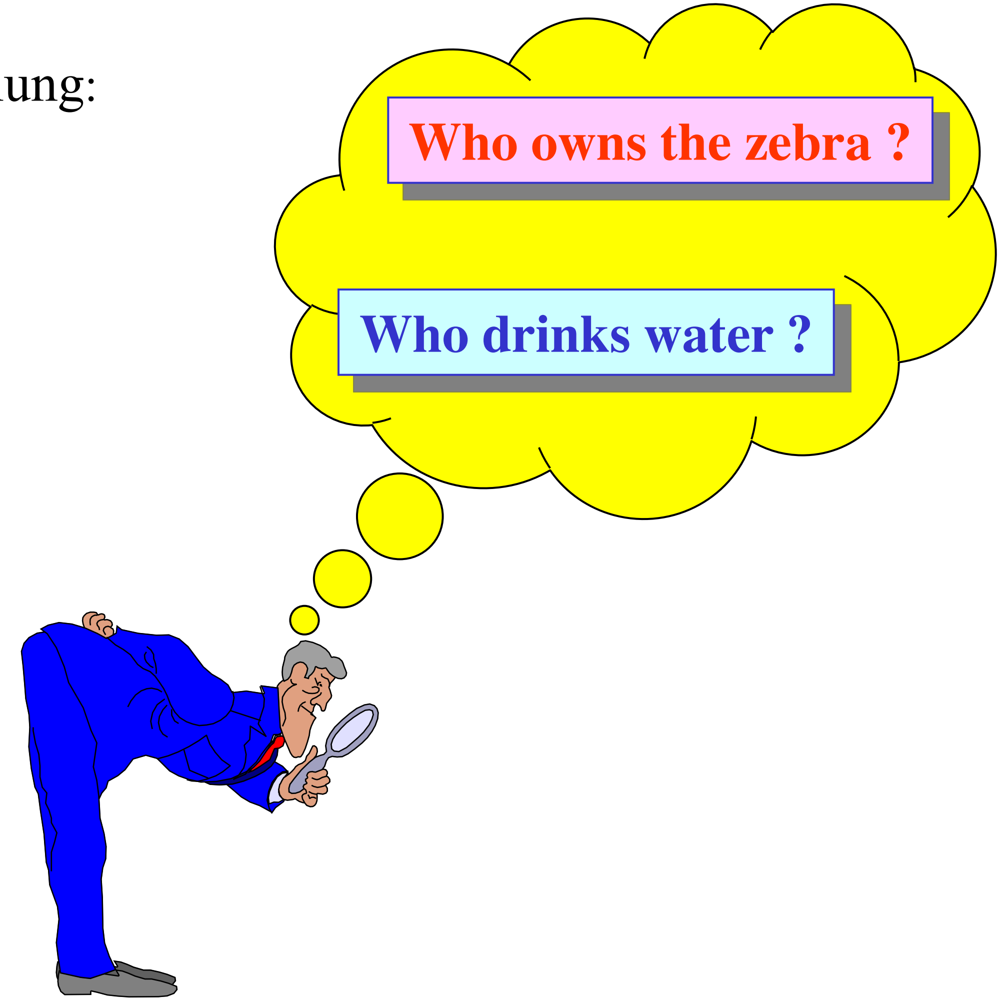


# Puzzle (1)

Insgesamt gelten 14 Regeln (engl. „clues“), die die „Welt“ des Puzzles definieren:

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the leftmost house.
10. The man who smokes Chesterfield lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

Problemstellung:



# Puzzle (3)

Systematische Konstruktion der Lösung (durch den Menschen):

| | | | | | |
|---|---|---|---|---|---|
| | 9. | 14. | 8. | | |
| house | 1 | 2 | 3 | 4 | 5 |
| color | | blue | | | |
| nationality | Norwegian | | | | |
| drink | | | milk | | |
| pet | | | | | |
| smoke | | | | | |

Es gibt drei sichere Zuordnungen – die sollte man zuerst festhalten!

**9. The Norwegian lives in the leftmost house.**
**14. The Norwegian lives next to the blue house.**
**8. Milk is drunk in the middle house.**

# Puzzle (4)

Bedingung 5:

5. **The green house is immediately to the right of the ivory house.**

5.

… lässt nur zwei Möglichkeiten offen:

| house | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| color | | blue | ivory | green ? | green |
| nationality | Norwegian | | | | |
| drink | | | milk | coffee | coffee |
| pet | | | | | |
| smoke | | | | | |

Eine direkte Konsequenz daraus wäre:

3. **Coffee is drunk in the green house.**

?

3.

# Puzzle (5)

Falls die 1. Lösung für ivory-green richtig ist: notwendige Position für red

1.

ebenfalls notwendig

**1. The Englishman lives in the red house.**

| house | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| color | yellow | blue | ivory | green | red |
| nationality | Norwegian | | | | English |
| drink | | | milk | coffee | |
| pet | | horse | | | |
| smoke | Kools | | | | |

**7. Kools are smoked in the yellow house.**

7.

11.

**11. Kools are smoked in the house next to the house where the horse is kept.**

**usw. …**

# Puzzle (6)

Eindeutige Lösung des Rätsels (zu finden durch diverse Backtrackingschritte):

| house | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| color | yellow | blue | red | ivory | green |
| nationality | Norwegian | Ukrainian | English | Spanish | Japanese |
| drink | **water** | tea | milk | juice | coffee |
| pet | fox | horse | snails | dog | **zebra** |
| smoke | Kools | Chesterfield | Winston | Lucky Strike | Parliaments |

# Puzzle: eine mögliche Spezifikation in Prolog

```
right_of(R, L, [ L | [ R | _ ] ]).
right_of(R, L, [ _ | Rest ]) :- right_of(R, L, Rest).
next_to(X, Y, List) :- right_of(X, Y, List).
next_to(X, Y, List) :- right_of(Y, X, List).
zebra(Zebra_Owner) :-
  8. ∧ 9.  Houses = [ [ _, norwegian, _, _, _ ], _, [ _, _, milk, _, _ ], _, _ ],
     1.    member([ red, englishman, _, _, _ ], Houses),
     2.    member([ _, spaniard, _, dog, _ ], Houses),
     3.    member([ green, _, coffee, _, _ ], Houses),
     4.    member([ _, ukrainian, tea, _, _ ], Houses),
     5.    right_of([ green, _, _, _, _ ], [ ivory, _, _, _, _ ], Houses),
     6.    member([ _, _, _, snails, winston ], Houses),
     7.    member([ yellow, _, _, _, kools ], Houses),
    10.    next_to([ _, _, _, _, chesterfield ], [ _, _, _, fox, _ ], Houses),
    11.    next_to([ _, _, _, _, kools ], [ _, _, _, horse, _ ], Houses),
    12.    member([ _, _, juice, _, lucky ], Houses),
    13.    member([ _, japanese, _, _, parliaments ], Houses),
    14.    next_to([ _, norwegian, _, _, _ ], [ blue, _, _, _, _ ], Houses),
     ?     member([ _, Zebra_Owner, _, zebra, _ ], Houses),
     ?     member([ _, _, water, _, _ ], Houses).
```