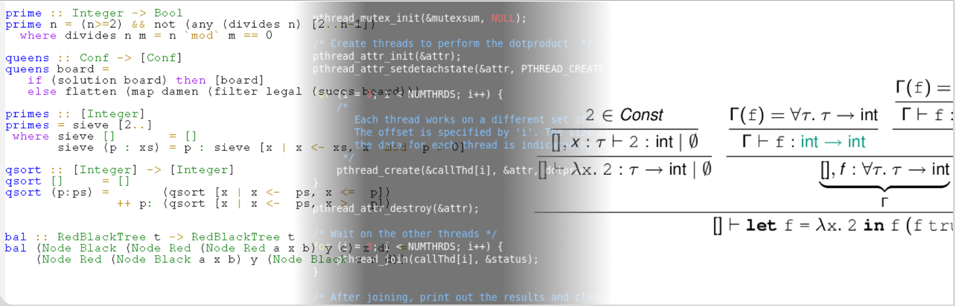
Karlsruher Institut für Technologie

Programmierparadigmen
Tutorium: List-Comprehensions, Backtracking

Prof. Dr.-Ing. Gregor Snelting | WS 2012/13

LEHRSTUHL PROGRAMMIERPARADIGMEN



Teil I

List-Comprehensions, Backtracking

List-Comprehensions

List-Comprehensions: "lediglich" Schreibweise zur Listengenerierung

- jede List-Comprehension prinzipiell auch als Kombination von **`map`**, **`filter`**, `flatten` darstellbar
- aber üblicherweise viel besser lesbar! zum Beispiel:

```
graduates :: Examination -> [Student]
graduates exam = [student | (student,assesment) <- exam, passed assesment ]
```

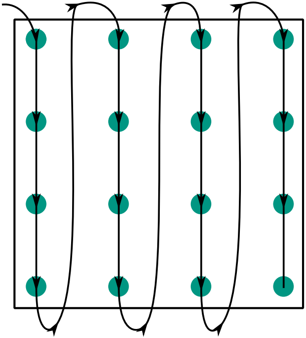
statt

```
graduates exam = map fst (filter (passed . snd) exam)
```

Aufzählung, List-Comprehensions

Menge (unsortiert): $\{(x, y) \mid x \in \{1 \ldots n\} \wedge y \in \{1 \ldots n\}\}$

Liste (spaltenweise):



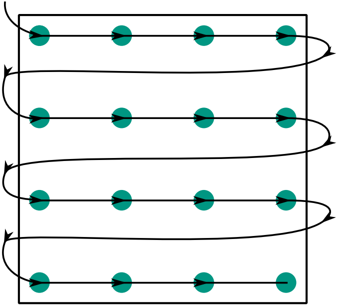
```
pairs     :: Integer -> [(Integer,Integer)]
pairs     n = [(x,y) | x <- [1..n], y <- [1..n]]
```

- Fixiere `x==1`, zähle auf `(1,1),(1,2),(1,3),(1,4)`
- Fixiere `x==2`, zähle auf `(2,1),(2,2),(2,3),(2,4)`
- ...

Aufzählung, List-Comprehensions

Menge (unsortiert): $\{(x, y) \mid x \in \{1 \ldots n\} \wedge y \in \{1 \ldots n\}\}$

Liste (zeilenweise):



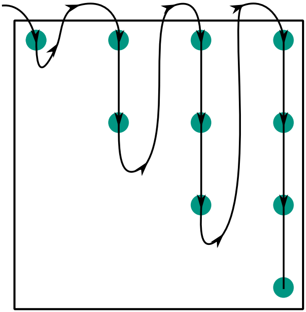
```
pairs    :: Integer -> [(Integer,Integer)]
pairs    n = [(x,y) | y <- [1..n], x <- [1..n]]
```

- Fixiere `y==1`, zähle auf `(1,1),(2,1),(3,1),(4,1)`
- Fixiere `y==2`, zähle auf `(1,2),(2,2),(3,2),(4,2)`
- ...

Aufzählung, List-Comprehensions

Menge (unsortiert): $\{(x, y) \mid x \in \{1 \dots n\} \wedge y \in \{1 \dots x\}\}$

Liste (spaltenweise):

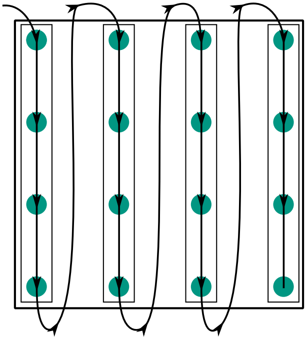


```
triangle :: Integer -> [(Integer,Integer)]
triangle n = [(x,y) | x <- [1..n], y <- [1..x]]
```

- Fixiere `x==1`, zähle auf `(1,1)`
- Fixiere `x==2`, zähle auf `(2,1),(2,2)`
- ...

Aufzählung, List-Comprehensions

`flatten` "Verflache" Listen von Listen



```
flatten :: [[t]] -> [t]
flatten lists = [ x | list <- lists, x <- list ]
```

- Fixiere erste Liste, zähle Elemente auf
- Fixiere zweite Liste, zähle Elemente auf
- ...

Backtracking, List-Comprehensions

Damenproblem: berechne Liste aller von `board` aus erreichbarer Lösungen

```
queens :: Board -> [Board]
queens board =
   if (solution board) then [board]
   else flatten (map queens (filter legal (succs board)))
```

List-Comprehension: macht *backtracking* explizit sichtbar

- Ersetze **`map`**, **`filter`** durch List-Comprehension

```
queens board =
   if   (solution board) then [board]
   else flatten [ queens succ  | succ <- (succs board), legal succ ]
```

- Ersetze `flatten` durch List-Comprehension

```
queens board =
   if   (solution board) then [board]
   else [ sol  | succ <- (succs board), legal succ, sol <- (queens succ) ]
```

Nimm `sol` in Liste von Lösungen auf, falls `succ` legale Nachfolger von `board`, und `sol` eine von `succ` erreichbare Lösung ist