

# Programmierparadigmen

## Tutorium: List-Comprehensions, Backtracking

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

### LEHRSTUHL PROGRAMMIERPARADIGMEN

```

prime :: Integer -> Bool
prime n = (n>=2) && not (any (divides n) [2..n-1])
  where divides n m = n `mod` m == 0

queens :: Conf -> [Conf]
queens board =
  if (solution board) then [board]
  else flatten (map damen (filter legal (successors board) NUMTHRS; i++))

primes :: [Integer]
primes = sieve [2..]
  where sieve [] = []
        sieve (p : xs) = p : sieve [x | x <- xs, x < p]

qsort :: [Integer] -> [Integer]
qsort [] = []
qsort (p:ps) = (qsort [x | x <- ps, x <= p])
              ++ p: (qsort [x | x <- ps, x > p])

bal :: RedBlackTree t -> RedBlackTree t
bal (Node Black (Node Red (Node Red a x b) y (z) di) < NUMTHRS; i++) {
  (Node Red (Node Black a x b) y (Node Black thread join(callThd[i], &status);
  }

-- After joining, print out the results and clean up

```

$$\begin{array}{c}
 \Gamma(f) = \\
 \frac{\Gamma(f) = \forall \tau. \tau \rightarrow \text{int} \quad \Gamma \vdash f : \text{int} \rightarrow \text{int}}{\Gamma \vdash f : \text{int} \rightarrow \text{int}} \quad \frac{\Gamma \vdash f : \forall \tau. \tau \rightarrow \text{int}}{\Gamma} \\
 \Gamma \vdash \text{let } f = \lambda x. 2 \text{ in } f (f \text{ tr}
 \end{array}$$

# Teil I

## List-Comprehensions, Backtracking

## 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,assessment) <- exam, passed assesment ]
```

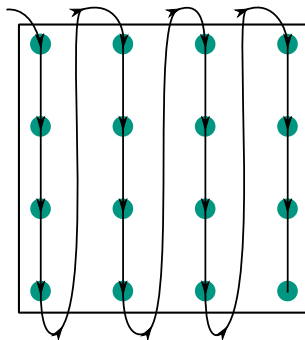
statt

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

# Aufzählung, List-Comprehensions

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

Liste (spaltenweise):



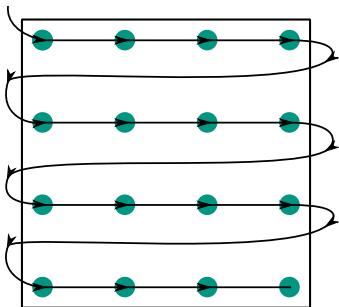
```
pairs    :: Integer -> [(Integer, Integer)]
pairs    n = [(x,y) | x <- [1..n], y <- [1..n]]
```

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

Liste (zeilenweise):



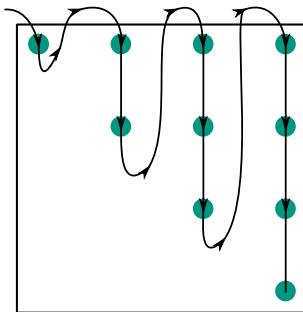
```
pairs      :: Integer -> [(Integer, Integer)]  
pairs n = [(x,y) | y <- [1..n], x <- [1..n]]
```

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

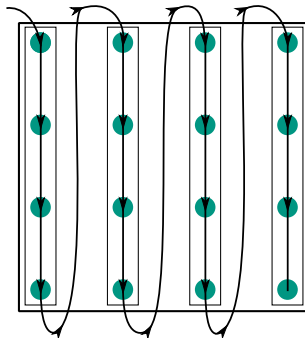
Liste (spaltenweise):



```
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  $(2, 1), (2, 2)$
- ...

`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
- ...

**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*