Informatik II für Verkehrsingenieure
Haskell (Kapitel 16)

Janis Voigtländer

Technische Universität Dresden

Sommersemester 2007

Wiederholung — Edit-Distanz

Problem: Berechnung der Distanz zwischen zwei Zeichenfolgen als minimale Anzahl von Edit-Operationen (Löschung, Einfügung oder Änderung)

Analyse:

$$d_{i,j} = \begin{cases} m - j & \text{wenn } i = n \\ n - i & \text{wenn } j = m \\ min\{1 + d_{i+1,j}\,;\, 1 + d_{i,j+1}\,;\, d_{i+1,j+1}\} & \text{wenn } \texttt{s[i]==t[j]} \\ 1 + min\{d_{i+1,j}\,;\, d_{i,j+1}\,;\, d_{i+1,j+1}\} & \text{sonst} \end{cases}$$

Umsetzung:
- Speicherung aller $d_{i,j}$ in Tabelle
- Berechnung in geeigneter Reihenfolge

Nachteile:
- Festlegung der Berechnungsreihenfolge willkürlich (also gewissermaßen überflüssig)
- geringe Nähe zur mathematischen Beschreibung

Wiederholung — Edit-Distanz in C:

```
int min(int x, int y, int z)
{ ... };

int d[n+1][m+1];
int i,j;

for (i=0; i<=n; i++) d[i][m]=n-i;
for (j=0; j<m; j++) d[n][j]=m-j;

for (j=m-1; j>=0; j--)
  for (i=n-1; i>=0; i--)
    { if (s[i]==t[j])
        d[i][j]=min(1+d[i+1][j],1+d[i][j+1],d[i+1][j+1]);
      else
        d[i][j]=1+min(d[i+1][j],d[i][j+1],d[i+1][j+1]);
    }
```


Edit-Distanz in Haskell:

```
min3 (x,y,z) = ...

dt = [ [ d (i,j) | j <- [0..m] ] | i <- [0..n] ]

d (i,j) | i==n          = m-j
        | j==m          = n-i
        | s!!i == t!!j = min3 (1 + dt !! i !! (j+1),
                               1 + dt !! (i+1) !! j,
                               dt !! (i+1) !! (j+1))
        | otherwise     = 1 + min3 (dt !! i !! (j+1),
                                    dt !! (i+1) !! j,
                                    dt !! (i+1) !! (j+1))
```

Wiederholung — Towers of Hanoi

Regeln:
- drei Plätze: A, B und C
- zu Beginn n Scheiben unterschiedlicher Größe auf Platz A
- niemals eine größere auf einer kleineren Scheibe

Ziel: alle Scheiben auf Platz B

Strategie:

$$\begin{aligned} \mathit{towers}(n+1,i,j,k) &= \mathit{towers}(n,i,k,j)\ \ \mathit{move}(i,j)\ \ \mathit{towers}(n,k,j,i) \\ \mathit{towers}(0,i,j,k) &= \varepsilon \end{aligned}$$

In Haskell:

```
towers(n+1,i,j,k) = towers(n,i,k,j) ++ [(i,j)] ++ towers(n,k,j,i)
towers(0,i,j,k)   = []
```


Einfache Typen, Operationen und Funktionen

`Int`:
- ganze Zahlen (`-12`, `0`, `42`, ...)
- Operationen: `+`, `-`, `*`, `^`
- Funktionen: `div`, `mod`, `min`, `max`, ...
- Vergleiche: `==`, `/=`, `<`, `<=`, `>`, `>=`

`Float`:
- Gleitkommazahlen (`-3.7`, `pi`, ...)
- Operationen: `+`, `-`, `*`, `/`
- Funktionen: `sqrt`, `log`, `sin`, `min`, `max`, ...
- Vergleiche: `==`, `/=`, `<`, `<=`, `>`, `>=`

`Bool`:
- boolesche Werte (`True`, `False`)
- Operationen: `&&`, `||`
- Funktionen: `not`

`Char`:
- einzelne Zeichen (`'a'`, `'b'`, `'\n'`, ...)
- Funktionen: `succ`, `pred`
- Vergleiche: `==`, `/=`, `<`, `<=`, `>`, `>=`

Auswertung einfacher Ausdrücke

```
> 5+7
12
```

```
> div 17 3                                   (nicht: div(17,3))
5
```

```
> 17 `div` 3
5
```

```
> pi/1.5
2.0943951023932
```

```
> min (sqrt 4.5) (1.5^3)     (nicht: min(sqrt(4.5),1.5^3))
2.12132034355964
```

```
> 'a'<='c'
True
```

```
> if 12<3 || 17.5/=sqrt 5 then 17-3 else 6
14
```


Komplexere Typen (I)

Listen:

- `[Int]` für `[]` oder `[-12,0,42]`
- `[Bool]` für `[]` oder `[False,True,False]`
- `[[Int]]` für `[[3,4],[],[6,-2]]`
- ...
- Operationen: `:`, `++`, `!!`, z.B.:

```
> 3:[-12,0,42]
[3,-12,0,42]
```

```
> [1.5,3.7] ++ [4.5,2.3]
[1.5,3.7,4.5,2.3]
```

```
> [False,True,False] !! 1
True
```

- Funktionen: `head`, `tail`, `last`, `null`, ...

Zeichenketten:

- `String = [Char]`
- spezielle Notation: `""` für `[]` und `"abcd"` für `['a','b','c','d']`

Komplexere Typen (II)

Tupel:

- `(Int,Int)` für `(3,5)` und `(0,-4)`
- `(Int,String,Bool)` für `(3,"abc",False)`
- `((Int,Int),Bool,[Int])` für `((0,-4),True,[1,2,3])`
- `[(Bool,Int)]` für `[(False,3),(True,-4),(True,42)]`
- ...
- Funktionen: `fst` und `snd` auf Paaren
- Auswertung komplexerer Ausdrücke:

```
> (3-4, snd (head [('a',42),('c',3)]))
(-1,42)
```


Deklaration von Werten

In File:

```
x = 7
y = 2*x
z = (mod y (x+2), tail [1..y])

a = b-c
b = fst z
c = head (snd z)

d = (a,e)
e = [fst d,f]
f = head e
```

Nach Laden:

```
> z
(5,[2,3,4,5,6,7,8,9,10,11,12,13,14])

> a
3

> d
(3,[3,3])
```

Optionale Typangaben

```
x,y :: Int
x = 7
y = 2*x

z :: (Int,[Int])
z = (mod y (x+2), tail [1..y])

a,b,c :: Int
a = b-c
b = fst z
c = head (snd z)

d :: (Int,[Int])
d = (a,e)

e :: [Int]
e = [fst d,f]

f :: Int
f = head e
```


Deklaration von Funktionen

```
min3 :: (Int,Int,Int) -> Int
min3 (x,y,z) = if x<y then (if x<z then x else z)
                      else (if y<z then y else z)

> min3 (5,4,6)
4
```

```
min3' :: Int -> Int -> Int -> Int
min3' x y z  = min (min x y) z

> min3' 5 4 6
4
```

```
isEven :: Int -> Bool
isEven n = (n `mod` 2) == 0

> isEven 12
True
```

Rekursive Funktionen

```
sumsquare :: Int -> Int
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)

> sumsquare 4
30
```

```
fac :: Int -> Int
fac n = if n==0 then 1 else n * fac (n-1)

> fac 5
120
```


Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)

> sumsquare 3
= if 3==0 then 0 else 3*3 + sumsquare (3-1)
= 3*3 + sumsquare (3-1)
= 9 + sumsquare 2
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)
= 9 + 2*2 + sumsquare (2-1)
= 9 + 4 + sumsquare 1
= 9 + 4 + if 1==0 then 0 else 1*1 + sumsquare (1-1)
= 9 + 4 + 1*1 + sumsquare (1-1)
= 9 + 4 + 1 + sumsquare 0
= 9 + 4 + 1 + if 0==0 then 0 else 0*0 + sumsquare (0-1)
= 9 + 4 + 1 + 0
= 14
```

Berechnung durch schrittweise Auswertung

```
a = 3

d = (a,e)

e = [fst d,f]

f = head e
```

```
> d
= (a,e)
= (3,e)
= (3,[fst d,f])
= (3,[fst (3,[fst d,f]),f])
= (3,[3,f])
= (3,[3,head e])
= (3,[3,head [3,head e]])
= (3,[3,3])
```


Pattern-Matching

Statt:
```
power :: Int -> Int
power n = if n==0 then 1 else 2 * power (n-1)
```

Auch:
```
power :: Int -> Int
power 0     = 1
power (m+1) = 2 * power m
```

Statt:
```
prod :: [Int] -> Int
prod l = if null l then 1
                   else head l * prod (tail l)
```

Auch:
```
prod :: [Int] -> Int
prod []     = 1
prod (x:xs) = x * prod xs
```

Komplexes Pattern-Matching

```
risers :: [Int] -> [[Int]]
risers []       = []
risers [x]      = [[x]]
risers (x:y:zs) = if x<=y then (x:s):ts else [x]:(s:ts)
    where (s:ts) = risers (y:zs)
```

```
> risers [1,2,0]
= if 1<=2 then (1:s):ts else [1]:(s:ts)
    where (s:ts) = risers (2:[0])
= (1:s):ts
    where (s:ts) = risers (2:[0])
= (1:s):ts
    where (s:ts) = [2]:(s':ts')
                     where (s':ts') = risers (0:[])
= (1:s):ts
    where (s:ts) = [2]:[[0]]
= [[1,2],[0]]
```


Komplexes Pattern-Matching

```
unzip :: [(Int,Int)] -> ([Int],[Int])
unzip []         = ([],[])
unzip ((x,y):zs) = (x:xs,y:ys)
    where (xs,ys) = unzip zs
```

```
> unzip [(1,2),(3,4)]
= (1:xs,2:ys)
    where (xs,ys) = unzip [(3,4)]
= (1:xs,2:ys)
    where (xs,ys) = (3:xs',4:ys')
                      where (xs',ys') = unzip []
= (1:xs,2:ys)
    where (xs,ys) = (3:xs',4:ys')
                      where (xs',ys') = ([],[])
= ([1,3],[2,4])
```

Pattern-Matching über mehreren Argumenten

```
drop :: Int -> [Int] -> [Int]
drop 0     xs     = xs
drop n     []     = []
drop (n+1) (x:xs) = drop n xs
```

```
> drop 0 [1,2,3]
[1,2,3]
> drop 5 [1,2,3]
[]
> drop 3 [1,2,3,4,5]
[4,5]
```


Reihenfolge beim Pattern-Matching

```
zip :: [Int] -> [Int] -> [(Int,Int)]
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
zip xs     ys     = []
```

```
> zip [1..3] [10..15]
[(1,10),(2,11),(3,12)]
```

```
zip :: [Int] -> [Int] -> [(Int,Int)]
zip xs     ys     = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

```
> zip [1..3] [10..15]
[]
```

Einfache Ein- und Ausgabe

```
module Main where

prod :: [Int] -> Int
prod []     = 1
prod (x:xs) = x * prod xs

main = do n <- readLn
          m <- readLn
          print (prod [n..m])
```

Nach Übersetzung:

```
5
8
1680
```


Algebraische Datentypen (I)

```
data Days = Monday | Tuesday | Wednesday | Thursday |
            Friday | Saturday | Sunday
```

- Typ `Days` hat mögliche Werte `Monday`, `Tuesday`, ...
- kann beliebig als Komponente in anderen Typen auftreten, etwa `[(Days,Int)]` mit z.B. `[]`, `[(Sunday,-5)]` und `[(Monday,1),(Wednesday,3),(Monday,0),(Friday,5)]` als möglichen Werten
- Berechnung mittels Pattern-Matching möglich:

```
workingday :: Days -> Bool
workingday Saturday = False
workingday Sunday   = False
workingday day      = True
```

Algebraische Datentypen (II)

```
data Date = Date Int Int Int
data Time = Hour Int
data Connection = Flight String Date Time Time |
                  Train Date Time Time
```

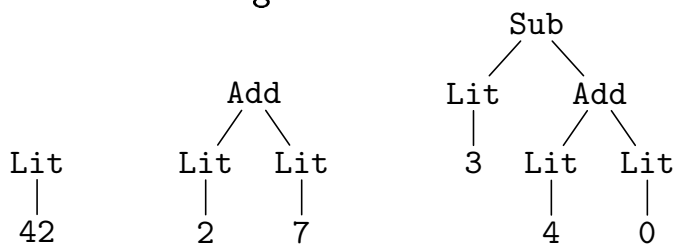
- mögliche Werte für `Connection`:
  `Flight "DBA" (Date 20 06 2007) (Hour 9) (Hour 11)`,
  `Train (Date 21 06 2007) (Hour 9) (Hour 13)`, ...
- Berechnung mittels Pattern-Matching:

```
travelTime :: Connection -> Int
travelTime (Flight _ _ (Hour d) (Hour a)) = a-d+2
travelTime (Train _ (Hour d) (Hour a))    = a-d+1
```


Algebraische Datentypen (III)

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

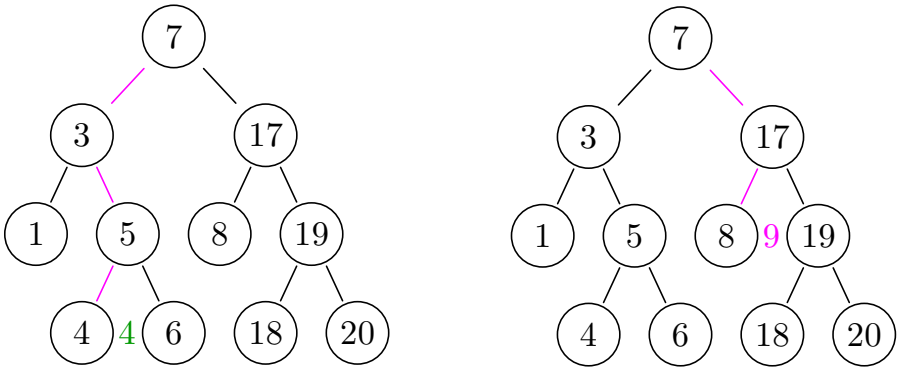
- mögliche Werte: `Lit 42`, `Add (Lit 2) (Lit 7)`,
  `Sub (Lit 3) (Add (Lit 4) (Lit 0))`, ...
- Baumdarstellung:



- Berechnung:

```
eval :: Expr -> Int
eval (Lit n)     = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

Wiederholung — Suchbäume



Wiederholung — Suchbäume in C

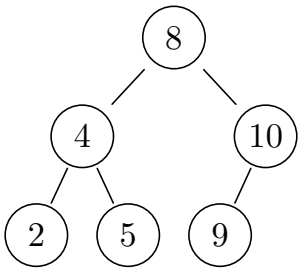
```
typedef struct Nodeelem *Ptr;

typedef struct Nodeelem { int key;
                          Ptr left, right;
                        } Node;

int search(Ptr t,int x)
{ if (t==NULL) return 0;
  if (t->key == x) return 1;
  if (t->key < x) return search(t->right,x);
  return search(t->left,x);
}
```

Suchbäume in Haskell

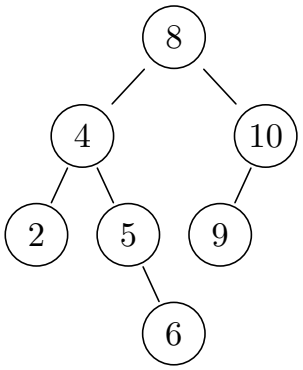
```
data Tree = Node Int Tree Tree | Nil

search :: Int -> Tree -> Bool
search x Nil                    = False
search x (Node key left right) = if key==x
                                   then True
                                   else if key<x
                                        then search x right
                                        else search x left
```


Wiederholung — Einfügen in Suchbäume

6



Wiederholung — Einfügen in Suchbäume



Wiederholung — Einfügen in Suchbäume in C:

```
void insert(Ptr *t,int x)
{ if (*t==NULL)
    { *t=(Ptr) malloc(sizeof(Node));
      (*t)->key=x;
      (*t)->left=NULL;
      (*t)->right=NULL;
    }
  else
    { if ((*t)->key < x) insert(&((*t)->right),x);
      else if ((*t)->key > x) insert(&((*t)->left),x);
    }
}
```

Einfügen in Suchbäume in Haskell:

```
insert :: Int -> Tree -> Tree
insert x Nil                    = Node x Nil Nil
insert x (Node key left right) =
    if key<x
    then Node key left (insert x right)
    else if key>x
         then Node key (insert x left) right
         else Node key left right
```


Simultan-rekursive algebraische Datentypen

```
data T1 = A T2 | E
data T2 = B T1
```

- mögliche Werte für `T1`: `E`, `A (B E)`, `A (B (A (B E)))`, `A (B (A (B (A (B E)))))`, ...
- mögliche Werte für `T2`: `B E`, `B (A (B E))`, `B (A (B (A (B E))))`, ...
- Berechnung:

```
as :: T1 -> Int
as (A t) = 1 + (as' t)
as E     = 0

as' :: T2 -> Int
as' (B t) = as t
```

Polymorphe Typen

```
append :: [Int] -> [Int] -> [Int]
append []     l = l
append (x:xs) l = x:(append xs l)
```

```
append' :: [Bool] -> [Bool] -> [Bool]
append' []     l = l
append' (x:xs) l = x:(append' xs l)
```

```
append'' :: String -> String -> String
append'' []     l = l
append'' (x:xs) l = x:(append'' xs l)
```


Polymorphe Typen

```
append :: [a] -> [a] -> [a]
append []     l = l
append (x:xs) l = x:(append xs l)

> append [1,2,3] [4,5,6]
[1,2,3,4,5,6]

> append [True] [False,True,False]
[True,False,True,False]

> append "abc" "def"
"abcdef"
```

Polymorphe Typen

```
drop :: Int -> [Int] -> [Int]
drop 0     xs     = xs
drop n     []     = []
drop (n+1) (x:xs) = drop n xs
```

```
drop :: Int -> [a] -> [a]
drop 0     xs     = xs
drop n     []     = []
drop (n+1) (x:xs) = drop n xs
```


Polymorphe Typen

```
append :: [a] -> [a] -> [a]
append []     l = l
append (x:xs) l = x:(append xs l)
```

```
> append "abc" [True]
    Couldn't match 'Char' against 'Bool'
        Expected type: Char
        Inferred type: Bool
    In the list element: True
    In the second argument of 'append', namely '[True]'
```

Polymorphe Typen

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
zip xs     ys     = []
```

```
> zip "abc" [True,False,True]
[('a',True),('b',False),('c',True)]
```

```
> :t "abc"
"abc" :: [Char]
```

```
> :t [True,False,True]
[True,False,True] :: [Bool]
```

```
> :t [('a',True),('b',False),('c',True)]
[('a',True),('b',False),('c',True)] :: [(Char,Bool)]
```


Polymorphe Typen

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

- mögliche Werte: `Nil`,
  `Node 4 Nil Nil :: Tree Int`,
  `Node 'a' Nil (Node 'b' Nil Nil) :: Tree Char`,
  ...
- aber nicht: `Node 4 (Node 'a' Nil Nil) Nil`
- Berechnung:

```
height :: Tree a -> Int
height Nil            = 0
height (Node n t1 t2) = 1 + (max (height t1)
                                 (height t2))
```

Funktionen höherer Ordnung

```
prod :: [Int] -> Int
prod []     = 1
prod (x:xs) = x * prod xs
```

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

```
foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int
foldr f k []     = k
foldr f k (x:xs) = f x (foldr f k xs)
```


Funktionen höherer Ordnung

```
data Tree = Node Int Tree Tree | Nil

insert :: Int -> Tree -> Tree
insert x Nil                    = Node x Nil Nil
insert x (Node key left right) = ...

buildTree :: [Int] -> Tree
buildTree []     = Nil
buildTree (x:xs) = insert x (buildTree xs)
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k []     = k
foldr f k (x:xs) = f x (foldr f k xs)
```

Funktionen höherer Ordnung

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k []     = k
foldr f k (x:xs) = f x (foldr f k xs)

buildTree :: [Int] -> Tree
buildTree xs = foldr insert Nil xs

prod :: [Int] -> Int
prod xs = foldr (*) 1 xs

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

...
```


Funktionen höherer Ordnung

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x):(map f xs)

isEven :: Int -> Bool
isEven n = (n `mod` 2) == 0

> map isEven [1..6]
[False,True,False,True,False,True]
```

Funktionen höherer Ordnung

```
filter :: (a -> Bool) -> [a] -> [a]
filter test []     = []
filter test (x:xs) = if (test x)
                     then x:(filter test xs)
                     else filter test xs

> filter isEven [1..6]
[2,4,6]
```