Nebenläufige und verteilte Programmierung mit Haskell — Plan

- 08.05.01 — Erinnerung an Haskell, funktionale I/O
- 10.05.01 — Einführung in *Concurrent Haskell*
- 15.05.01 — Abstraktionen in *Concurrent Haskell*
- 17.05.01 — Socket-Programmierung
- 22.05.01 — Ereignisbasierte Programmierung in *Concurrent Haskell*
- Später: graphische Benutzerschnittstellen mit Haskell

  Sonstiges:

  Alle Beispiele sind auf der Webseite!

  Dort auch Links zu weiterführender Doku.

Erinnerungen an Haskell

- Funktionale Sprache
- Typen
- Definition von Funktionen durch Pattern-Matching
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Typklassen
- *Neu:* Ein/Ausgabe
- *Neu:* Fehlerbehandlung

Haskell: Funktionen und Typen

- *Programme* sind *Funktionen*, die Eingabe auf Ausgabe abbilden.

```
fac n   = if n <= 0 then 1 else n*fac (n-1)

hallo x = if x == 0 then "" else "hallo! " ++ hallo (x-1)
```

  Keine Variablen, kein Zustand, Rekursion

- Einschränkung der Werte durch *Typisierung*:

```
fac   :: Int -> Int
hallo :: Int-> String
```

  Vordefinierte Basistypen: `Int`, `Integer`, `Double`, `String`, `Char`, `Bool`, ...

Algebraische Datentypen und Polymorphie

- Selbstdefinierte Datentypen (mit *Konstruktoren* `Leaf`, `Tree`):

```
data Tree a = Leaf a | Tree (Tree a) (Tree a)    deriving (Show, Eq)
```

  Bäume sind *polymorph* über `a`:

```
Leaf "foo" :: Tree String
Tree (Leaf 4) (Tree (Leaf 4) (Leaf 10)) :: Tree Int
```

  Nicht typkorrekt: `Tree (Leaf "Vier") (Leaf 5)`, `Tree (Leaf True) 4`, ...

- Funktionsdefinition durch *pattern matching*

```
height (Leaf _)     = 0
height (Tree t1 t2) = max (height t1) (height t2)+ 1
```

Vordefinierte Datentypen

- Listen: `data [a] = [] | a : [a]`
  ```
  length []     = 0
  length (x:xs) = 1+ length xs
  ```
- Tupel: `data (a, b) = (a, b)`
  ```
  f :: (Int, Int)-> Int
  f (a, b)= a+ b
  ```
  Sonderfall: das leere Tupel `() :: ()` (Typ mit genau einem Wert)
  NB: `(a, b)-> c` $\cong$ `a-> b-> c` (isomorph, nicht gleich)
- Partialität: `data Maybe a = Just a | Nothing`
  ```
  inv :: Double -> Maybe Double
  inv x = if x == 0 then Nothing else Just (1 / x)
  ```


Funktionen höherer Ordnung

- Funktionen als Argumente von Funktionen:
  ```
  twice :: (a-> a)-> a-> a
  twice f x = f (f x)

  infixr 9 .
  (.)   :: (b-> c)-> (a-> b)-> a-> c
  (f. g) x = f (g x)
  ```
- Priorität einiger Infix-Operationen:
  `^` 8 `*`, `/` 7 `+`, `-` 6 `++` 5 `==`, `<=` 4 `&&` 3 `||` 2
- Einige vordefinierte Funktionen für Listen:
  ```
  map    :: (a-> b)-> [a]-> [b]
  filter :: (a-> Bool)-> [a]-> [a]
  ```


Beispiel: Das Nim-Spiel

- Zwei Spieler nehmen abwechselnd von einem Haufen Streichhölzer 1,2 oder 3 Hölzer; wer das letzte nimmt, hat verloren.
- Frage: kann ein Spieler ein Spiel mit 21 Hölzchen gewinnen?
- Ich kann *gewinnen*, wenn ich einen Zug machen kann, nach dem mein Gegner verlieren muss.
- Der Gegner muss verlieren, wenn
  - entweder nur noch ein Hölzchen übrig ist
  - oder ich bei allen seinen möglichen Zügen gewinnen kann.

Nim auf gut Haskell

```
-- Alle möglichen Züge für x Hölzchen:
moves :: Int-> [Int]
moves x = let valid :: Int-> Bool
              valid x = 0 < x
          in  filter valid ([x-1, x-2, x-3])

-- Spieler am Zug kann gewinnen
canwin :: Int-> Bool
canwin x = any mustlose (moves x)

-- Spieler am Zug muß verlieren
mustlose :: Int-> Bool
mustlose x = x==1 ||  all canwin (moves x)
```

Nim: Wie kann ich denn jetzt gewinnen?

`wins x` ergibt `Just m` gdw. m Hölzchen nehmen gewinnt

```
-- Alle möglichen Züge für x Hölzchen:
moves :: Int-> [Int]
moves x = filter ((<) 0) (map (\n-> x- n) [1,2,3])

-- Spieler am Zug kann mit gewinnen
wins :: Int-> Maybe Int
wins x = case filter mustlose (moves x) of
             m:_ -> Just (x-m)
             [] -> Nothing

-- Spieler am Zug muß verlieren
mustlose :: Int-> Bool
mustlose x = x==1 || all (isJust.wins) (moves x)
```


Harnessing the Power of Logic

- Bleibt ein Hölzchen über, habe ich verloren;
  wenn mein Zug den Gegner verlieren läßt, habe ich gewonnen.
- Für $n = 1$, verloren; für $n = 2, \ldots, 4$ Gewinnzüge klar; für andere n Übergang zum *Restkörper*

  Gewinnzug bei n Hölzchen: $m = (n \bmod 4 - 1) \bmod 4 = (n - 1) \bmod 4$

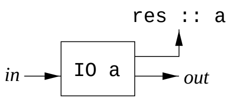
  Unmöglich zu gewinnen, falls $m = 0$

```
wins :: Int-> Maybe Int
wins n =
  let m = (n- 1) `mod` 4
  in  if m == 0 then Nothing else Just m
```


Monadische I/O

- *Problem:* Eingabe kann nicht als Funktion `readLine ::() -> String` modelliert werden (keine *referentielle Transparenz*)
- *Lösung:* „Einmal I/O, immer I/O" — Abhängigkeit von Umwelt am Typ erkennbar
- Abstrakter Datentyp `IO a` („Monadische I/O"):

```
(>>=)  :: IO t-> (t-> IO u)-> IO u  -- "then"
return :: t-> IO t                  -- "return"
```



`IO a` ist eine Aktion („Funktion auf der Umwelt"), die auch einen Wert vom Typ `a` berechnet.