Hinweise zum Abschlusstest

- Mitnahme von maximal 3 handbeschriebenen DIN-A4-Blättern (Vorder- und Rückseite)
- Alle Inhalte der Vorlesung sind relevant
- Orientiert Euch an Übungen und Hausaufgaben

Generelles zur Prüfungsordnung

- Fällt man durch ein Modul durch, muss man sich zum nächsten Versuch erneut anmelden
- Die Anmeldung für diese Prüfung ist für **einen** der beiden Test-Termine gültig. Es kann frei gewählt werden.
- Aus organisatorischen Gründen: Anmeldung für einen der beiden Termine demnächst über ISIS
- Wer zum ersten Termin durchfällt und zum zweiten Termin den nächsten Versuch wahrnehmen möchte, muss sich erneut anmelden
- Wer den nächsten Versuch erst im WS2015/2016 machen will, muss sich Anfang des WS2015/2016 dafür anmelden

Softwaretechnik und Programmierparadigmen

VL12: Funktionale Programmierung

Prof. Dr. Sabine Glesner
FG Programmierung eingebetteter Systeme
Technische Universität Berlin

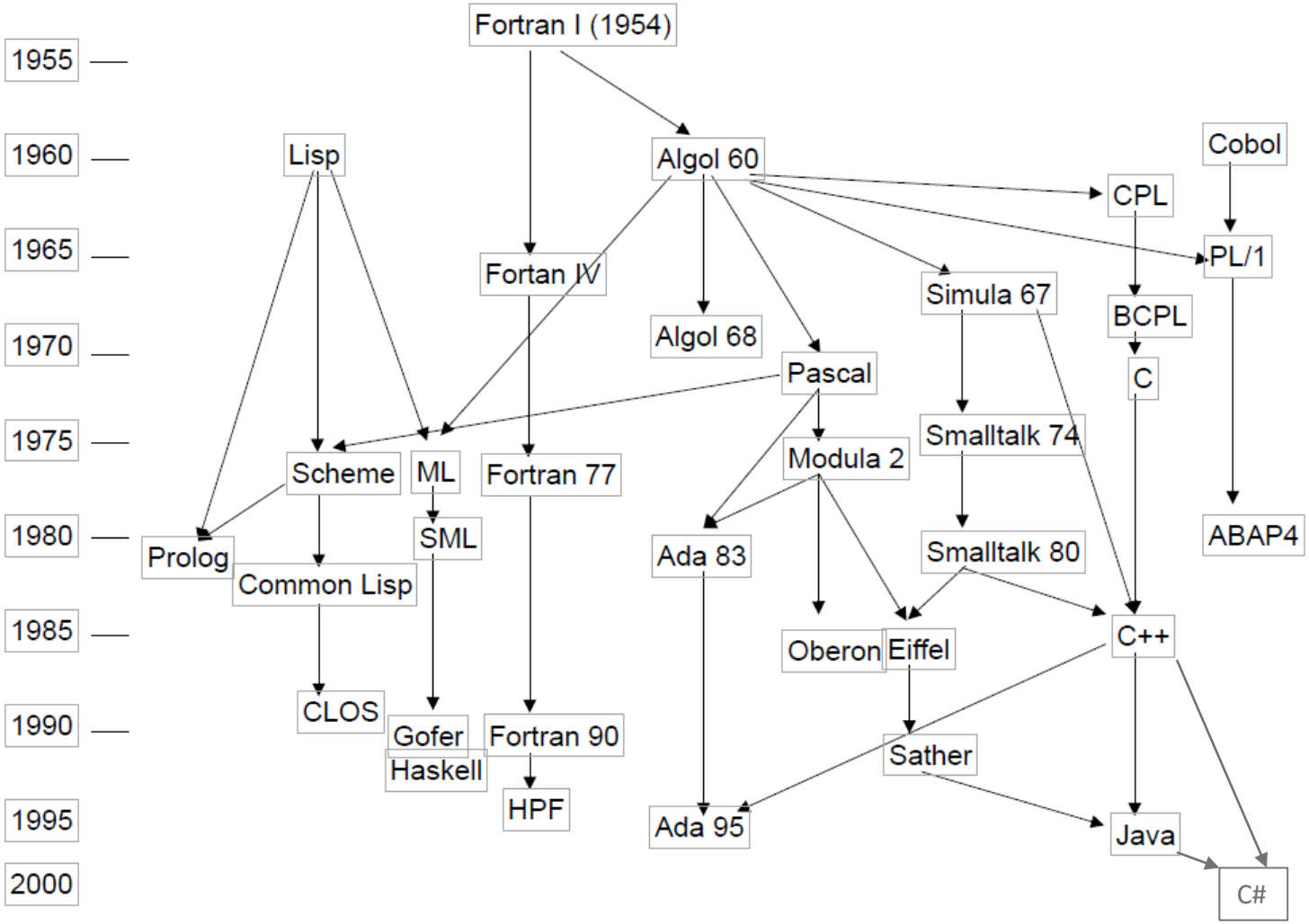
Einordnung

- Bisher in der Vorlesung: Modellierung, Qualitätssicherung und Implementierung für objektorientierte Systeme
- Aber: Nicht immer ist Objektorientierung gut geeignet!

Übersicht

- Einführung Programmierparadigmen
- Funktionale Programmierung mit Haskell
  - Motivation
  - Einfache Funktionen
  - Rekursive Funktionen
  - Abstrakte Datentypen
  - Polymorphie und Listen
  - Weiterführende Konzepte
- Ausblick: Logische Programmierung

Entwicklung von Programmiersprachen



Historische Entwicklung

Funktionale Programmiersprachen:

Praxisnahe Erweiterung eines mathematischen Modell

- Spezifikation, was berechnet werden soll
- implementieren µ-rekursive Funktionen
- µ-rekursive Funktionen: Funktionen, die sich rekursiv aufrufen können, ohne dass die Berechnung unbedingt terminieren muss
- Variablen kann höchstens einmal ein Wert zugewiesen werden

Imperative Programmiersprachen:

Abstraktion „echter“ Computer mit Prozessor, Speicher etc.

- Folge nacheinander ausführbarer Operationen
- Programmvariablen sind Abstraktionen von Speicherstellen
- Turing-Maschinen: einfaches, aber universelles Maschinenmodell
- Variablen können mehrfach neu beschrieben werden bei der Programmausführung

Vergleich funktional -- imperativ

Funktional

- Geschachtelte Funktionsauswertung
- Funktionen können auch Werte, also Parameter für andere Funktionen sein (Funktionale)
- Beispiele für Funktionale:
  - Compiler: Programme, die Programme als Eingabe erwarten
  - Ableitungsoperator: Funktion, die die Ableitung einer anderen Funktion berechnet

Imperativ (prozedural, operational)

- Zustände bei der Programmausführung
- Zustand: z.B. Speicherbelegung oder Werte der Variablen
- Zustandsübergänge (wenn Variablen ihre Werte bei Zuweisungen ändern)

Weiteres Paradigma: Logische Programmierung

- Programm besteht aus einer Menge von logischen Fakten und Schlussregeln (Hornklauseln)
- Anfragen an das Programm: werden mittels Resolution und Unifikation gelöst
- später mehr dazu

Programmierparadigmen

- Imperatives Programmieren (C, Java, Algol, Pascal)
- Funktionales Programmieren (LISP, ML, Haskell)
- Logisches Programmieren (Prolog, Datalog)
- Wissenschaftliches Rechnen (Fortran, HPC)
- Objektorientiertes Programmieren (Simula, Java, C#, Eiffel)
- Wirtschaftsanwendungen (COBOL)
- Skriptsprachen (JavaScript, Python, PHP)
- Modellierungsorientierte Programmiersprachen (Matlab/Simulink/Stateflow, StateChart-Dialekte wie z.B. SafeStateMachines)

Gemeinsame Eigenschaften von Programmiersprachen

- Syntax
  - legt genau fest, welche Programme zu einer Programmiersprache gehören
  - Sprache = Menge von Wörtern
  - Programmiersprache = Menge von Programmen

- Semantik
  - definiert Bedeutung von Programmen
    - was sie berechnen
  - syntaktisch falsche Programme haben keine Bedeutung!

Unterschied Syntax – Semantik
Beispiel: Switch-Case-Statement

Beispiel in Java

```
int cnt = 0;
switch (x) {
        case 1: cnt+=1;
        case 2: cnt+=1;
        default : cnt+=1;
}
```

Beispiel in Visual Basic

```
Dim cnt As Integer
Select Case x
    Case 1: cnt = cnt + 1
    Case 2: cnt = cnt + 1
    Case Else: cnt = cnt + 1
End Select
```

Übersicht

- Einführung Programmierparadigmen
- Funktionale Programmierung mit Haskell
  - Motivation
  - Einfache Funktionen
  - Rekursive Funktionen
  - Abstrakte Datentypen
  - Polymorphie und Listen
  - Weiterführende Konzepte
- Ausblick: Logische Programmierung

Funktionale Programmiersprachen

- Wichtige Vertreter funktionaler Programmiersprachen
  - (Common) LISP
  - ML
  - HASKELL
  - GOFER
  - OPAL

Funktionale Programmierung: Motivation

- Funktionen und Typen im Mittelpunkt
- Typisierung: Nicht nur Werte, sondern Funktionen, Typen, Typkonstruktoren als Argumente
- keine while-Schleifen, aber Rekursion
  - „Variablen“ in funktionalen Sprachen wird max. einmal ein Wert zugewiesen, daher machen Schleifen keinen Sinn!

While-Schleife vs Rekursion

Fakultät imperativ

```
int fac(int n){
int res = 1;

while(n > 0){
  res = res * n;
  n --;
}
return res;
}
```

Fakultät funktional

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

Praktische Dinge am Anfang

- wir verwenden Haskell in dieser Vorlesung
- Haskell-Compiler sind frei verfügbar, können auf praktisch jeder Plattform installiert werden
- empfohlen für die Veranstaltung: glasgow haskell compiler (ghc)
- Tutorials zum Nachlesen:
  - http://learnyouahaskell.com/syntax-in-functions
  - http://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf
  - …

Übersicht

- Einführung Programmierparadigmen
- Funktionale Programmierung mit Haskell
  - Motivation
  - Einfache Funktionen
  - Rekursive Funktionen
  - Abstrakte Datentypen
  - Polymorphie und Listen
  - Weiterführende Konzepte
- Ausblick: Logische Programmierung

Syntax - Modulstruktur

module <ModuleName> where

imports <Imported Module>

data <TypeName> = <TypeDefinition>

<function-name> :: <Input-type> -> <Output-type>
<function-name> <parameters> = <function-definition>

Coding Standard

- Modulnamen werden GROSS geschrieben
- Typnamen werden GROSS geschrieben
- Konstruktoren von einem Typ werden GROSS geschrieben
- Funktionsnamen werden klein geschrieben
- Typparameter werden klein geschrieben

- Haskell-Compiler erzwingen diese Standards!

Erste Haskell-Funktionen: Konstanten

```
goldenRatio :: Double
goldenRatio = 1.618

goldenRatioPrecise :: Double
goldenRatioPrecise = (sqrt 5 + 1) / 2
```

Erste Haskell-Funktionen: „Richtige“ Funktionen

```
doubleMe :: Int -> Int
doubleMe x = x + x

maximum :: Int -> Int -> Int
maximum x y = if x > y then x else y
```

Funkionsaufruf: „maximum 10 5“ → ohne Klammern

Anonyme Funktionen

- Funktionen können auch ohne Angabe eines Namens definiert werden

```
\ x y -> if x < y then y else x
```

- Unbenannte Funktionen können zum Beispiel verwendet werden, um benannte zu definieren

```
min : Int -> Int -> Int
min = \ x y -> if x < y then y else x
```

Dies ist vollkommen gleichwertig zu „min x y = if x < y then x else y“

Currying

- Betrachtet nochmal die Minimumfunktion

```
min :: Int -> Int -> Int
min = \ x y -> if x < y then x else y
```

- Was passiert, wenn man nicht alle Argumente übergibt?

„min 6“ → alle freien Vorkommen von x werden durch 6 ersetzt
→ min 6 = \ y -> if 6 < y then 6 else y

- (min 6) :: Int -> Int

Damit ist “min“ eine Funktion, die für ein übergebenes Integer eine Funktion als Ergebnis liefert!

Übersicht

- Einführung Programmierparadigmen
- Funktionale Programmierung mit Haskell
  - Motivation
  - Einfache Funktionen
  - Rekursive Funktionen
  - Abstrakte Datentypen
  - Polymorphie und Listen
  - Weiterführende Konzepte
- Ausblick: Logische Programmierung

Rekursion: Potenzieren als erstes Beispiel

```
-- pre: n >=0
potenz :: Double -> Int -> Double
potenz x n = if n == 0 then 1
                       else x * potenz x (n-1)
```

- Alternative Variante mit Pattern Matching:

```
potenz :: Double -> Int -> Double
potenz x 0 = 1
potenz x n = x * potenz x (n-1)
```

Fibonacci-Funktion

```
-- pre: n >= 0
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Ende 29.1.

Übersicht

- Einführung Programmierparadigmen
- Funktionale Programmierung mit Haskell
  - Motivation
  - Einfache Funktionen
  - Rekursive Funktionen
  - Abstrakte Datentypen
  - Polymorphie und Listen
  - Weiterführende Konzepte
- Ausblick: Logische Programmierung

Definition Abstrakter Datentyp

- Ein abstrakter Datentyp besteht aus
  - Konstruktoren C1, …, Cn
  - Die Anzahl der Parameter gibt die Stelligkeit eines Konstruktors an
- Induktive Definition der Elemente des Datentyps:
  - Wenn Ci eine nullstellige Konstruktorfunktion ist, dann ist Ci auch ein Element des ADT.
  - Wenn Ci m-stellig ist und Argumente der abstrakten Datentypen S1, …, Sm erwartet und t1, t2, …, tm Elemente der entsprechenden ADTs sind, dann ist auch (Ci t1 t2 …. tm) ein Element des ADT.
  - Alle Terme, die auf diese Weise gebildet werden können, und sonst nichts! (das heißt „induktiv definiert“)

(Abstrakte) Datentypen

- Basistypen wie Bool, Int, String, Real, etc. vordefiniert
- Eigene Datentypen können mit „data“ definiert werden

- Typ für Farben

```
data color = Orange | Red | Green | Blue | Yellow
```

- Typ für geometrische Objekte

```
data Shape = Circle Double
           | Rectangle Double Double
```

Alternative Definition:

```
data Shape = Circle { radius :: Double}
           | Rectangle { width :: Double, height :: Double}
```

Automatische Erzeugung von Selektoren, z.B. radius :: Shape -> Double

Übersicht

- Einführung Programmierparadigmen
- Funktionale Programmierung mit Haskell
  - Motivation
  - Einfache Funktionen
  - Rekursive Funktionen
  - Abstrakte Datentypen
  - Polymorphie und Listen
  - Weiterführende Konzepte
- Ausblick: Logische Programmierung

Problem

- Datentyp für Paare
  - data PairIntInt = PairConstructorIntInt Int Int
  - data PairIntDouble = PairConstructorIntDouble Int Double
  - data PairStringInt = PairConstructorString Int String Int
  - …

Das geht besser!

Polymorphie (1): Polymorphe Typen

- Wunsch: Paare für beliebige zwei Typen definieren

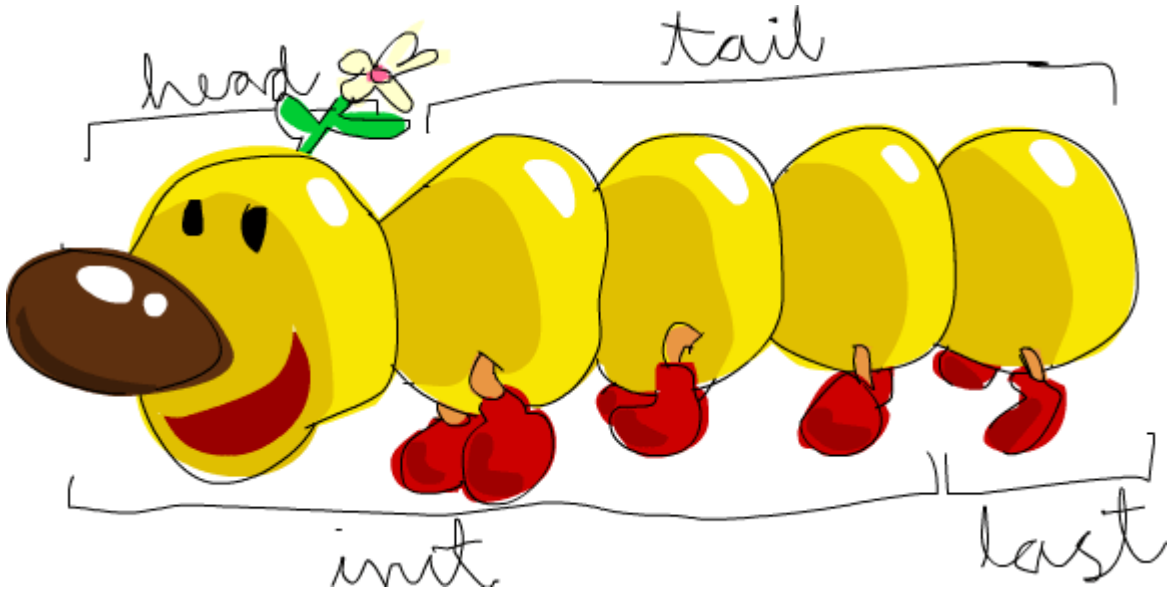
data Pair a b = PairConstructor a b

a und b sind Typparameter, d.h. repräsentieren beliebige Typen

- „PairConstructor (5::Int) (6::Int)“ ist vom Typ „Pair Int Int“
- „PairConstructor (5::Double) (6::Int)“ ist vom Typ „Pair Double Int“

- In Haskell gibt es einen vordefinierten Typ für Paare „(a,b)“ mit dem Konstruktur (_,_), d.h., „(5::Double,6::Int)“ ist vom Typ „(Double,Int)“

Rekursive Datentypen: Listen



siehe http://learnyouahaskell.com/starting-out

Abstrakter Polymorpher Datentyp für Listen

```
data List a = Nil
            | Cons a (List a)
```

Der vordefinierte Listendatentyp in Haskell wird als „[a]“ geschrieben, mit „[] : [a]“ als die leere Liste und „: a -> [a] -> [a]“ als Listenkonstruktor.

- Die Liste <5,7,3> kann in Haskell als „(5::Int) : 7 : 3 : []“ geschrieben werden (oder auch kurz [(5::Int),7,3])

Einfache Operationen auf Listen

- Head:: [a] -> a
  gibt das erste Element einer Liste zurück
- tail:: [a] -> [a]
  entfernt das erste Element aus einer Liste und gibt die Restliste zurück
- init:: [a] -> [a]
  entfernt das letzte Element einer Liste und gibt die resultierende Liste zurück
- last:: [a] -> a
  gibt das letzte Element einer Liste zurück

Rekursive Funktionen auf Listen

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

length :: [a] -> Int
length [] = 0
length (x : xs) = 1 + (length xs)

last [x] = x
last (x:xs) = last xs
```

In Haskell ist „++ :: [a] -> [a] -> [a]“ vordefiniert.

Für leere Listen nicht definiert!

Higher Order Functions: map

Wende auf jedes Element einer Liste eine Funktion an (vergleichbar mit OCL-Operation collect).

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

- Beispiel: map doubleMe [4,7,2]
  = [doubleMe 4, doubleMe 7, doubleMe 2]
  = [8,14,4]

Higher Order Functions: filter

Nur Elemente einer Liste behalten, die einer bestimmten Bedingung genügen (vergleichbar mit OCL-Operation select)

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

- weitere Notation: filter p xs == [x | x <- xs , p x]
- Beispiel: filter (\x -> x `mod` 2 = 0) [4,7,2] = [4,2]

Higher Order Functions: reduce

```
reduce :: (a -> b -> b) -> b -> [a] -> b
reduce op e [] = e
reduce op e (x:xs) = x `op` reduce op e xs
```

Operationen dürfen infix verwendet werden, wenn sie in Hochkommas gestellt werden

- In Haskell als „foldr“ vordefiniert
- Beispiel: foldr (+) 0 [4,7,2] = 4 + (7 + (2 + 0)) = 13
- Analog gibt es auch „foldl“

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

Anwendung: quicksort

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = let smaller = [y | y <- xs , y < x]
                       equal = [y | y <- xs , y == x]
                       bigger = [y | y <- xs , y > x]
                   in quicksort smaller ++ equal ++ quicksort bigger
```

Polymorphie (2): Typklassen

Warum konnten wir quicksort nicht polymorph deklarieren?

Weil angenommen werden muss, dass ein Vergleichsoperator ‚<' auf dem Typen „a" existiert.

→ Verwendung von Typklassen: Typparameter + Abstrakte Operationen

Definition von Typklassen

```
class Eq a where
        (==) :: a -> a -> Bool
```

Erben von Typklassen

```
class Eq a => Ord a where
        (<),(<=),(>=),(>)  :: a -> a -> Bool
        max, min           :: a -> a -> a
```

Instantiierung von Typklassen

```
instance Eq Int where
        x == y = intEq x y
```

Bedingte Instantiierung

```
instance Eq a => Eq [a] where
        [] == [] = True
        (x:xs) == (y:ys) = x == y && xs == ys
        xs == ys = False                -- sonst
```

Funktionen auf Listen: Polymorphes Quicksort

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = let smaller = [y | y <- xs , y < x]
                       equal = [y | y <- xs , y == x]
                       bigger = [y | y <- xs , y > x]
                   in quicksort smaller ++ equal ++ quicksort bigger
```

Übersicht

- Einführung Programmierparadigmen
- Funktionale Programmierung mit Haskell
  - Motivation
  - Einfache Funktionen
  - Rekursive Funktionen
  - Abstrakte Datentypen
  - Polymorphie und Listen
  - Weiterführende Konzepte
- Ausblick: Logische Programmierung

Map-Reduce-Schema

MapReduce bei Google, Facebook & Co.

- Zahlen aus dem Jahr 2008:
  - mehr als 10.000 MapReduce-Programme bei Google in den Jahren 2004-2008
  - durchschnittlich 100.000 MapReduce-Jobs ausgeführt auf Google-Clustern jeden Tag, wodurch insgesamt mehr als 20 Petabytes Daten pro Tag produziert werden
  - Anwendungen: Ermitteln von z.B. Menge der häufigsten Anfragen pro Tag, Anzahl Wörter in Webseiten, …
- Was sind eigentlich MapReduce-Programme?

MapReduce als Programmierschema

- große Mengen (Listen) an Eingabedaten
- map-Operation angewendet auf Liste solcher Eingabedaten (key/value pairs)
- liefert Liste von Ausgabedaten
- selektiere daraus die interessanten Daten
- auf diese selektierten Daten wird reduce-Operation angewendet

Implementierung von MapReduce

- Schwierigkeit beim Implementieren:
  - nicht die Operationen an sich, die sind trivial
  - sondern die Größe der Datenmenge
- Lösung: Parallelisierung, d.h. Verteilung auf z.B. PC-Cluster
  - so dass der Programmierer sich nicht um die Verteilung kümmern braucht

MapReduce effizient implementieren

- Funktionales Denken hilft, weil ohne Seiteneffekte
- das allein reicht aber nicht: Beispiel:
  - Problem, wenn einer der parallelen Prozesse nicht fertig wird (weil Festplatte des entsprechenden PCs nicht korrekt arbeitet, weil andere Jobs auf dem Rechner laufen, …)
  - Lösung: sobald fast alle Berechnungen abgeschlossen sind, werden noch fehlende dupliziert, also doppelt ausgeführt als Backup-Tasks
  - reduziert Berechnungszeit erheblich
- siehe Papier:
  Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing
  On Large Clusters. Communications of the ACM January 2008, Vol. 51, No.1.
- Online-Zugang ACM: http://www.acm.org/dl

Typinferenz

Typinferenz

- Typen von Funktionen müssen in Haskell nicht angegeben werden
  - Es empfiehlt sich aber für die Lesbarkeit und Dokumentation
- Compiler kann zu jedem Ausdruck ohne Typannotation den Typ berechnen (Typinferenz)
  - Genauer: Berechnung des allgemeinsten Typs
- Siehe auch: Hindley-Milner-Typinferenz

Evaluierungsstrategien

Eager Evaluation vs Lazy Evaluation

- Ursprung in Auswertungsstrategien im Lambda-Kalkül
- Eager Evaluation oder „call-by-value“: Beim Aufruf einer Funktion werden zunächst die Argumente vollständig ausgewertet, bevor der Funktionsrumpf ausgewertet wird
- Lazy Evaluation oder „call-by-need“: (Teil-) Ausdrücke werden nur dann ausgewertet, wenn sie benötigt werden
- Beispiel: mult (x,y) == if x=0 then 0 else x*y

  - (Common) LISP: Eager Evaluation
  - ML: Eager Evaluation
  - **HASKELL: Lazy Evaluation**
  - GOFER: Lazy Evaluation
  - OPAL: Eager Evaluation

Beispiel: Unendliche Listen

```
zeros :: [Int]
zeros = 0 : zeros

ones :: [Int]
ones = map (\x -> x+1) zeros

nats :: [Int]
nats = 0 : map (\x -> x + 1) nats
```

take 5 zeros = ??, drop 5 zeros = ?? [0,0,0,0,0] [0,0,0,…]
take 5 ones = ??, drop 5 ones = ?? [1,1,1,1,1] [1,1,1,…]
take 5 nats = ??, take 5 nats = ?? [0,1,2,3,4] [5,6,7,…]

Beispiel: Sieb des Eratosthenes

- Idee: Zähle alle Primzahlen auf

Algorithmus:

1. Schreibe alle natürlichen Zahlen ab 2 in einer Liste A auf
2. Nimm das erste Element von A und füge es der Liste von Primzahlen P hinzu
3. Lösche alle Vielfachen dieser Zahl aus A und mach mit Schritt 2 weiter

Beispielablauf

	A = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,…]	P = []
→	A = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,…]	P = [2]
→	A = [3,5,7,9,11,13,15,…]	P = [2]
→	A = [3,5,7,9,11,13,15,…]	P = [2,3]
→	A = [5,7,11,13,…]	P = [2,3]
→	A = [5,7,11,13,…]	P = [2,3,5]
→	A = [7,11,13,…]	P = [2,3,5]
→	A = [7,11,13,…]	P = [2,3,5,7]
→	A = [11,13,…]	P = [2,3,5,7]
→	A = [11,13,…]	P = [2,3,5,7,11]
→	A = [13,…]	P = [2,3,5,7,11]
→	A = [13,…]	P = [2,3,5,7,11,13]
→	A = […]	P = [2,3,5,7,11,13]
→	…	

Und nun in Haskell

- Schreibe alle natürlichen Zahlen ab 2 in einer Liste A auf

  map (\x -> x + 2) nats

  primes = sieve (map (\x -> x + 2) nats)

- Nimm das erste Element von A und füge es der Liste von Primzahlen P hinzu

  sieve (p:as) = p : sieve (…)

- Lösche alle Vielfachen dieser Zahl aus A und mach mit Schritt 2 weiter

  sieve (p:as) = p : sieve (filter (\x -> x `mod` p > 0) as)

  Alternativnotation: sieve (p:as) = p : sieve ([x | x <- as , x `mod` p > 0])

Verwendung der unendlichen Primzahlliste

```
printFirstPrimes :: Int -> [Int]
printFirstPrimes n = take n primes
```

Übersicht

- Einführung Programmierparadigmen
- Funktionale Programmierung mit Haskell
  - Motivation
  - Einfache Funktionen
  - Rekursive Funktionen
  - Abstrakte Datentypen
  - Polymorphie und Listen
  - Weiterführende Konzepte
- Ausblick: Logische Programmierung in der nächsten Vorlesung

Literatur

- http://learnyouahaskell.com/syntax-in-functions
- http://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf