Programmierkonzepte und Algorithmen – Listen

Seminaristischer Unterricht

Prof. Dr.-Ing. Hendrik Gärtner

Gliederung

- Übungsaufgaben
- Pattern Matching
- Modellierung verketteter Listen
  - Abstrakte Klassen
  - Vererbung
  - Überschreibung von Methoden
- Case-Klassen
  - Motivation
  - Pattern Matching mit Case-Klassen
  - Beispiele

Beispiel Rekursionen

Berechnung der 10001. Primzahl! Vorgehensweise?

1. Schreiben eines Primzahlentests
2. Entwerfen einer Funktion, die den Primzahlentest so lange aufruft, bis er 10001 mal true geliefert hat

```
def is_prim(X:Int)= calcPrim(X, 2, math.sqrt(X).toInt+1)

def calcPrim(X:Int, i:Int, Max:Int):Boolean =
   if (i>=Max) true
      else if (X % i == 0) false
          else calcPrim(X,i+1,Max)
```

Aufgabe 1

Berechnung der 10.001. Primzahl - Vorgehensweise?

```
def primNr(X:Int) = getPrim(X,2)


def getPrim(nr:Int, currentNr:Int):Int =
   if (nr<=0) currentNr-1
     else if (is_prim(currentNr)) getPrim(nr-1,currentNr+1)
       else getPrim(nr,currentNr+1)
```

Fibonacci Zahlen

```
def fibo (x:Int):Int= x match {
      case 0 => 0
      case 1 => 1
      case x => fibo(x-1)+fibo(x-2)
}



def fibo_h(in:Int, count:Int, fib1:Double, fib2:Double):Double = {

    if (in>=count) fibo_h(in, count+1, fib1+fib2, fib1)
          else fib1+fib2

}

def fiboDynProg(x:Int):Double = x match {

      case 0 => 0
      case 1 => 1case x => fibo_h(x,2,1,0)

}
```

Pattern Matching

```
def multiple2(x:Double):Double = x match {
    case 0 => 0
    case y  if ((y % 3 ==0) || (y % 5 ==0)) => (y + multiple2(y-1))
    case y if (y>0) => multiple(y-1)
    case _ => throw new Error("No negative Numbers!")
}
```

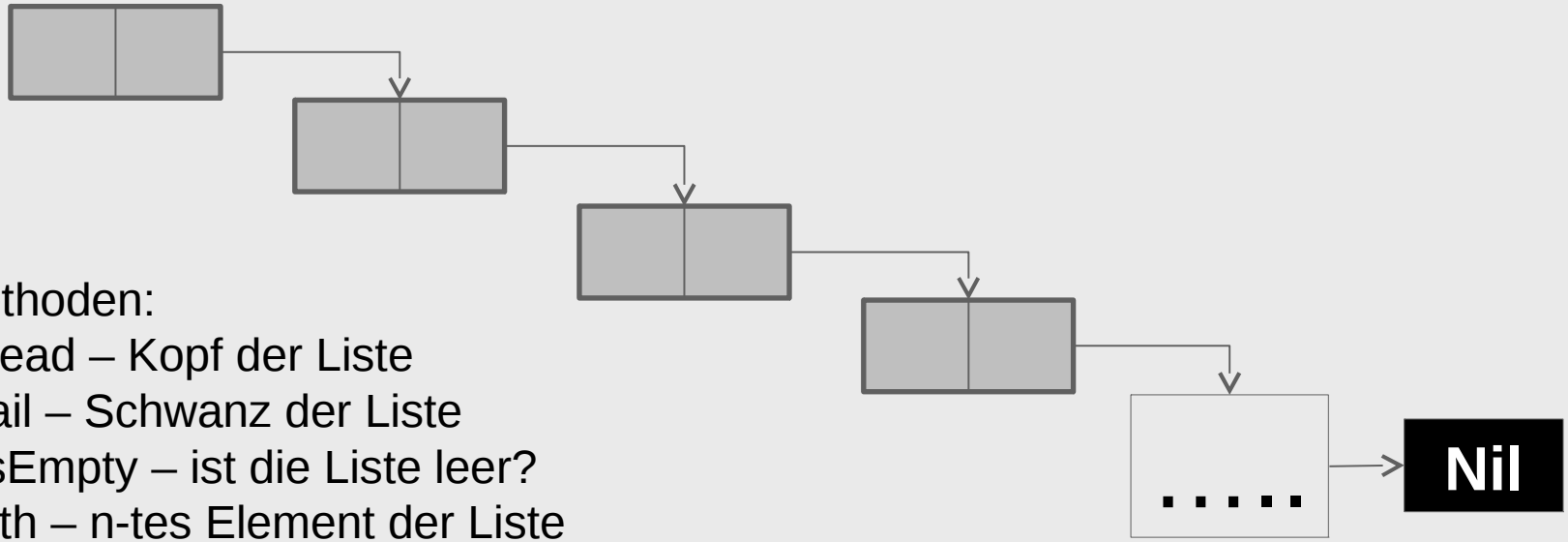
- Pattern Matching kann viel Code sparen – funktioniert ähnlich wie die switch-Anweisung in Java (ist aber mächtiger)
- Patterns können bspw. Konstanten sein (wie in dem Beispiel die 0 – es könnte aber auch eine Konstante in einer Variable sein, die dann aber groß geschrieben werden müsste z.B. Pi)
- Pattern können Variablen enthalten, die zugewiesen werden (y)
- Patterns können sog. *Guards* (if Bedingungen) enthalten
- Patterns können Wild Cards („_“) beinhalten – sie ignorieren den Wert

Konstante oder Variable

```
scala> import Math.{E,Pi}
import Math.{E,Pi}
scala> E match {
        case Pi => “strange math? Pi=“+Pi
        case _ => “OK“}
res10: java.lang.String=OK


scala> val pi= Math.Pi
Pi: Double = 3.141592653589793
scala> E match {
        case pi => “strange math? Pi=“+pi}
res10: java.lang.String= strange math? Pi= 2.71828…
```

Verkettete Listen

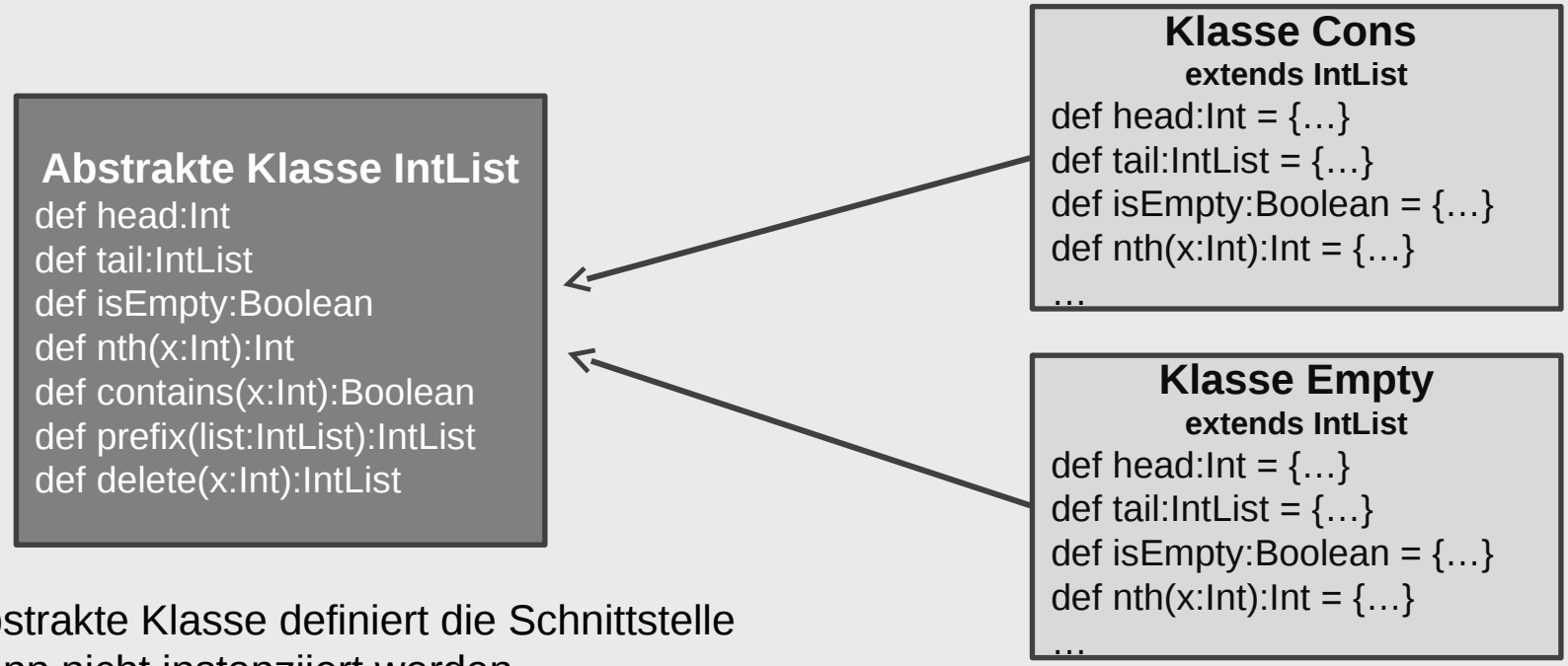


Methoden:

- head – Kopf der Liste
- tail – Schwanz der Liste
- isEmpty – ist die Liste leer?
- nth – n-tes Element der Liste
- contains – beinhaltet die Liste ein Element?
- ++ - Konkatenation von zwei Listen
- deleteElem – Löschen eines Elements

für eine Liste aus Zahlen IntList

Klassenstruktur



- Abstrakte Klasse definiert die Schnittstelle
- Kann nicht instanziiert werden
- Kann Code enthalten
- Kann Typ einer Variablen sein, aber in der konkreten Ausprägung vielgestaltig (Polymorphie)

- Implementieren die Schnittstelle
- Müssen mindestens die abstrakten Methoden aus IntList implementieren

Zentrale Frage: In welcher Klasse sollen welche Methoden implementiert werden?

Aufbau einer Schnittstelle

```
abstract class IntList {
    def isEmpty:Boolean
    def head:Int
    def tail:IntList
}
object Empty extends IntList{
    def isEmpty=true
    def head= throw new Error("head.nil")
    def tail= throw new Error("tail.nil")
    override def toString= "Empty"
}
class Cons (val head:Int, val tail:IntList) extends IntList{
    def isEmpty=false
    override def toString= "List("+ head+ ","+ tail + ")“
}
```

Terminologie

- Empty und Cons erweitern (*extends) die Klasse IntList*
- Das impliziert, dass Empty und Cons konform sind zum Typ IntSet

→ Ein Objekt vom Typ Empty oder Cons kann immer dann genutzt werden, wenn ein Objekt vom Typ IntList erforderlich ist

- IntSet wird *Superklasse* von Empty und Con genannt
- Empty und Cons sind *Subklassen* von IntList
- In Scala erweitern alle vom User definierten Klassen eine andere Klasse
- Ist keine Superklasse explizit genannt, so wird von der Java Standard-Klasse Object aus package java.lang abgeleitet
- Alle direkten und indirekten Superklassen werden *Basisklassen* genannt
- Basisklassen von Cons sind also IntList und Object

Implementierung und Überschreiben

- Die Definition von head und tail in den Klassen Empty und Cons überschreiben die abstrakten Funktionen der Klasse IntList
- Es ist ebenfalls möglich implementierte Klassen zu überschreiben (es muss override verwendet werden)

Beispiel

```
abstract class Base {
def foo = 1
def bar: Int
}
```

```
class Sub extends Base {
override def foo = 2
def bar = 3
}
```

Objekt-Definitionen

Die Klasse Empty ist unveränderlich, das heißt es macht keinen Sinn, Instanzen der Klasse mehrfach zu erzeugen.

Daher sollte Empty als Objekt definiert werden*:*

```
object Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def insert(x: Int): IntSet =
        new Cons(x, Empty)
}
```

- Singleton Objekte werden einmalig beim ersten Gebrauch erzeugt
- Müssen nicht mit new instanziiert werden

IntLists erzeugen

```
object Empty extends IntSet {
…
   def insert(x: Int): IntSet =
         new Cons(x, Empty)
}
class Cons extends IntSet {
…
   def insert(x: Int): IntSet =
         new Cons(x, this)
}
```

Listen erzeugen:

```
val x= Empty.insert(4).insert(7).insert(9).insert(11) oder
val y= new Cons(4, new Cons(7, new Cons(9, new Cons(11,Empty))))
```

Dynamic Method Invocation

- Object-oriented Sprachen (auch Scala) benutzen den Ansatz des “*dynamic method dispatch*“
- Das bedeutet, dass zur Laufzeit entschieden wird, welche Methode aufgerufen wird
- Dabei wird von der Klasse, die erzeugt wurde, ausgehend die Vererbungshierarche so lange durchsucht, bis eine konkrete Methode gefunden worden ist

Methode nth-Element und contains

- Möglichkeit die Methoden in der Klasse IntList (als einmalige Definition) und in Cons/Empty zu implementieren
- Verhalten der Methoden ist bei den Klassen Cons/Empty unterschiedlich z.B.
  - Ist die Liste leer, dann kann weder das Element noch gefunden werden, noch kann das Element mit nth zugegriffen werden
  - Ist die Liste nicht leer, so kann weiter gesucht werden
- Unterschiedliche Verhaltensweisen sollten in den Klassen Empty und Cons implementiert werden
- Das Laufzeitsystem ruft dann automatisch die richtige Methode auf

Definition der Methode contains und nth

In Objekt Empty:

def contains(elem:Int):Boolean=false

def nth(index:Int)= throw new Error("IndexOutOfBound")

In Objekt Cons:

```
def nth(index:Int):Int= index match{
    case 0 => head
    case i => tail.nth(i-1)}
def contains(elem:Int):Boolean= elem match{
    case y if (y==head) => true
    case _ => tail.contains(elem)}
```

Entwerfen Sie eine Methode delete, die das erste Vorkommen eines Elements aus einer Liste löscht

def delete(elem:Int):IntList

Definition von Funktionen innerhalb der Abstrakten Klasse

Aufgabe: Entwerfen Sie eine Funktion prefix(l:ListInt), die den prefix l vor die aktuelle Liste setzt!

Variante 1

```
def prefix(list:IntList):IntList = {
   if (list.isInstanceOf[Listen.Cons])
         new Cons(list.head,this.prefix(list.tail))
              else this
 }
```

- Verwendung der Funktion isInstanceOf, um zu überprüfen, um welche Art von Objekt es sich handelt (Cons/Empty)
- Generell gilt: Solche Abfragen sowie Type-Casts mit asInstanceOf[Type] zu vermeiden – sie sind sehr fehlerträchtig

Variante 2

```
def prefix(list1:IntList):IntList = list1.isEmpty match{
   case true => this
   case false => new Cons(list1.head,this.prefix(list1.tail))
 }
```

- Verwendung der Funktion isEmpty, um zu testen, ob das Ende der Liste erreicht worden ist
- Erweitern des Ausdrucks der „match“-Klausel
- Eleganter, aber noch nicht optimal

Besser: Verwendung von Case-Classes

Case-Classes (1/3)

Einfügen des Schlüsselworts case vor class:

```
case class Cons (val head:Int, val tail:IntList) extends IntList{…}
case class Empty{…}  // war vorher vom Typ Object
```

Folgen:

1. Automatische Generierung einer Factory-Methode mit dem Klassennamen
   Objekte können über diese Methode erzeugt werden ohne das Schlüsselwort new zu verwenden

z.B. val nums=Cons(4,Cons(3,Cons(1,Empty()))) anstatt
val nums= new Cons(4,new Cons(3,new Cons(1,Empty())))

Case Classes (2/3)

2. Der Compiler addiert zu jeder Variable im Konstruktor der Klasse das val, so dass diese zu öffentlichen Eigenschaften werden (keine Relevanz für das Beispiel)
   case class Cons (val head:Int, val tail:IntList) extends IntList{…}

3. Der Compiler fügt Methoden für toString, hashCode und equals hinzu
   Cons(3,Cons(4,Empty()))==Cons(3,Cons(5,Empty())) → false
   Cons(3,Cons(4,Empty()))==Cons(3,Cons(4,Empty())) → true
   Rekursive Überprüfung

Apropo == und equals: == und equals tun das selbe – nämlich auf Gleichheit der Werte überprüfen. Objektreferenzen können mit *.eq* und *.ne* getestet werden

Aufgabe

Implementieren Sie eine Funktion traverse, die einen Liste umdreht. Implementieren Sie diese in der Abstrakten Klasse IntList.

```
def prefix(l:IntList):IntList=l match {
    case Empty => this
    case Cons(head, tail) => new Cons(head,this.prefix(tail))
  }
```

Vielen Dank für

Ihre Aufmerksamkeit

Prof. Dr.-Ing. Hendrik Gärtner

Aufgabe 2

Schreiben Sie eine Funktion, die überprüft, ob innerhalb eines Ausdrucks (eine Liste von Character) eine valide Klammerung existiert. So soll bspw.:

- Dies ist ein (kleiner) (((()Test)) - true zurückgeben und
- Dies)(() ist falsch - false.

```
def balance(chars: List[Char]): Boolean = {
    def help(balVal: Int, chars:List[Char]):Boolean = (balVal,chars) match {
      case (value, Nil) => (0==value)
      case (value, x::xs) => if (x=='(') help(value+1,xs) else {
       if (x==')' && value<1) false else if (x==')') help(value-1,xs) else help(value,xs)
     }
    }
  help(0,chars)
}
```