Lists, PatternMatching, CaseClasses & Objects

Funktion auf Listen

- Wiederholung:

  ähnliche Notation wie in Prolog jedoch ohne Klammern

  Liste kann leer sein: `Nil`

  Liste nicht leer: `head::tail`

  Bsp.: `1::2::3::Nil`

- gesucht Funktion, die Länge einer Liste berechnet
- 1. Ansatz: Verwendung obiger Listenstruktur

```
def listlength(xs:List[Any]) : Int =
  if   (xs == Nil)          0
  //xs = head::tail
  else  listlength(tail) + 1

  listlength(1::2::3::Nil) ->  Error
```

- Problem: tail nicht vorhanden

listLength auf OO-Listenstruktur von Scala

- 2. Ansatz:

  verwende OO-Listenstruktur von Scala (siehe OOS Api)

```
Nil :        isEmpty
X::Xs :      head -> X, tail -> Xs

def listlength(xs:List[Any]) : Int =
if (xs.isEmpty) 0
else listlength(xs.tail) + 1


listlength(1::2::3::Nil) ->  Int = 3
```

- wollen aber keine OO-Konzepte verwenden
- darum: Operiere auf der internen Listenstruktur mittels:

Pattern Matching
unidirektionale Unifikation

- 3. Ansatz:
  - nicht Teile von Listen durch Methoden ermitteln (wie in OOS),
  - sondern direkt auf interne Listenstruktur zugreifen (wie in Prolog)

```
def listlengthPM(xs:List[Any]) : Int = xs match {
  case Nil => 0
  case head :: tail => 1 + listlengthPM(tail)
}
listlength(1::2::3::Nil) ->  Int = 3
```

Pattern Matching
Definition

- Def. (Pattern Matching): (aus Wikipedia)

"Pattern Matching (englisch für *Musterabgleich*) oder **musterbasierte Suche** ist ein Begriff für symbolverarbeitende Verfahren, die anhand eines vorgegebenen Musters diskrete Strukturen oder Teilmengen einer diskreten Struktur identifizieren."

Pattern Matching Bsp.

- Einfügen eines Ints an richtiger Stelle in sortierter Liste

```
def ins(x:Int, xs:List[Int]) : List[Int] = xs match
{
  case Nil => x :: Nil
  case y :: ys => if (x<=y) x::xs else y :: ins(x,ys)
}

insert(2,1::3::Nil)   ->   List[Int] = List(1, 2, 3)
```

- Übung: InsertionSort unter Verwendung von insert
- Wie Pattern Matching bei Datenstrukturen, z.B. Binärbäume, für die es keine vordefinierte Struktur gibt?

- Lösung: Struktur selbst definieren mittels:

Case Classes und Case Objects

- in TILO Listen definiert mit:
  - leere Liste: -> `nil`, (Konstante)
  - zusammengesetzt: -> `list(head,tail)` (2-stell. Funktor)

- geht in Scala auch durch:
  - Konstante = Case Object `Nil`
  - 2-stell. Funktor = Case Class `List`
    - im 1. Argument Eintrag `head`
    - im 2. Argument Restliste `tail`
- Bsp.: Definition einer Case Class `MyList`

```
abstract class MyList
case object Nil extends MyList
case class List(  head:Int,
                  tail:MyList) extends MyList
```

Pattern Matching auf Case Class

- Bsp.: `listlength` auf `MyList`:

```
def listlengthCC(xs : MyList) : Int = xs match {
  case Nil => 0
  case List(head, tail) => listlengthCC(tail) + 1
}

listlengthCC(List(1,List(2,Nil)))   ->  Int = 2
```

Pattern Matching
allgemein

- Bisher: Pattern Matching bei
  - Interner Listenstruktur

```
xs match {
      case Nil =>
      case head :: tail =>
```

  - Case Objects & Classes

```
xs match {
      case Nil =>
      case List(head, tail) =>
```

- Dabei immer: Bereitstellung von Variablen
- Einzige Anwendung in rein funktionalen Sprachen

- In Scala aber viel allgemeiner definiert

Pattern Matching
als Ersatz für switch/case

Switch/case (Java)	Pattern Matching (Scala)

int i = ...;
switch (i) {
  case 2:
  break;
  case 3: case 4: case 7:
  break;
  default:
  break;
}
	val i = ...
	i match {
	  case 2 =>
	  case 3 | 4 | 7 =>
	  case _ =>
	}

- Bei Pattern Matching immer nur 1. zutreffender Fall ausgeführt -> kein break vorhanden
- Mehrere Überprüfungen in einem möglich
- Default-Fall _ muss vorhanden sein, falls was fehlt
- _ steht für alles (Wildcard)

Pattern Matching
Beliebige Typen & Pattern

```
val any: Any = ...
val matched = any match {
  case n : Int => "the number: " +n
  case "hello" => "a string"
  case true | false => "a boolean"
  case 45.35 => "a double"
  case _ => "an unknown value"
}
```

- Brauchen keine Klammern innerhalb eines Matchs
- Typinferenz ermittelt Typ

```
def matching(xs: List[Int]) = xs match {
  case _ :: n :: _ => „2. element of List is "+n
  case 5 :: 3 :: Nil => "List contains 5 and 3"
  case Nil => "Nil"
}
```

- Gibt noch viele weitere Möglichkeiten, siehe: https://scalatutorial.wordpress.com