Funktionale Programmierung
Formen der Rekursion

Programm

1. Endrekursion (tail recursion)
   - Akkumulator Pattern

Endrekursion (tail recursion)

In der funktionalen Programmierung werden viele, in anderen Paradigmen typischerweise iterativ formulierte, Algorithmen rekursiv ausgedrückt. Aus dieser Präferenz folgt, dass funktionale Sprachen[1] Optimierungen anbieten, um Rekursion effizient zu übersetzen. Wichtig ist dabei insbesondere, nicht für jeden rekursiven Funktionsaufruf den Aufrufstapel zu vergrössern. Dadurch können Stapelüberläufe auch bei tiefer Rekursion verhindert werden. Die "tail-call" Optimierung behandelt genau diesen Fall.

[1] Genauer Compiler für funktionale Sprachen

Endrekursion (tail recursion)

- Eine Deklaration liegt in endrekursiver Form oder “tail-recursive” vor, wenn Resultate von rekursiven Aufrufen direkt[2] zurückgegeben werden.
- Die meisten (Compiler von) Funktionalen Sprachen übersetzen endrekursive Funktionen so, dass bei deren Ausführung konstanter Stapelspeicher in Anspruch genommen wird. Man spricht dabei von “tail-call” Optimierung.

[2] D.h. ohne weitere Modifikation oder Verwendung.

Endrekursion (tail recursion)

Die Deklaration

```
1 sum_ :: [Integer] -> Integer
2 sum_ [] = 0
3 sum_ (x:xs) = x + (sum_ xs)
```

ist nicht endrekursiv, weil das Resultat des rekursiven Funktionsaufrufes als Summand weiterverwendet wird. Dieselbe Funktionalität lässt sich endrekursiv deklarieren:

```
1 sumTR_ :: Integer -> [Integer] -> Integer
2 sumTR_ acc [] = acc
3 sumTR_ acc (x:xs) = sumTR_ (x + acc) xs
4
5 sumTR = sumTR_ 0
```

Endrekursion (tail recursion)

Akkumulator Pattern

Das eben gesehene Muster um eine rekursive Funktion in eine endrekursive Form zu bringen besteht darin, das Zwischenresultat der Rekursion explizit in einem "Akkumulator" mitzuführen. Daher heisst diese Vorgehensweise "Akkumulator Pattern".

Endrekursive Form der Fakultätsfunktion mit einem Akkumulator:

```
1 fakTR :: Integer -> Integer
2 fakTR = fakTR_ 1
3     where
4         fakTR_ :: Integer -> Integer -> Integer
5         fakTR_ acc 0 = acc
6         fakTR_ acc n = fakTR_ (n * acc) (n-1)
```

Endrekursion (tail recursion)

Akkumulator Pattern

Aufgabe

Bringen Sie folgende Funktionen in eine endrekursive Form:

```
1 pow x y
2     | y < 1 = 1
3     | otherwise = x * pow x (y - 1)
```

Endrekursion (tail recursion)

Akkumulator Pattern

Aufgabe

Bringen Sie folgende Funktionen in eine endrekursive Form.

```haskell
isPalindrome :: String -> Bool
isPalindrome w
    | l < 2 = True
    | otherwise =
        w0 == wE && isPalindrome w'
    where
        l  = length w
        w0 = head w
        wE = last w
        w' = tail $ init w
```

Endrekursion (tail recursion)

Continuation Pattern

- Nicht alle rekursiven Funktionsdeklarationen lassen sich mittels eines (einfachen) Akkumulators in endrekursive Form bringen.

Endrekursion (tail recursion)

Continuation Pattern

- Nicht alle rekursiven Funktionsdeklarationen lassen sich mittels eines (einfachen) Akkumulators in endrekursive Form bringen.
- Manchmal ist es einfacher, anstelle eines Akkumulators, der die bereits geleistete Arbeit repräsentiert, einen Funktionsparameter mitzuführen, der die noch zu leistende Arbeit darstellt.

Endrekursion (tail recursion)

Continuation Pattern

Die Fakultätsfunktion mit dem Continuation - Pattern umgesetzt:

```
1 fakC :: Integer -> Integer
2 fakC = fakC_ (const 1)
3   where
4     fakC_ f n
5         | n < 1 = f n
6         | otherwise = fakC_ (\x -> n * (f x)) $
              n - 1
```

Aufgabe

Bringen Sie mithilfe des Continuation - Patterns die Funktion `map` in eine endrekursive Form.

```
1 myMap :: (a -> b) -> [a] -> [b]
2 myMap f [] = []
3 myMap f (x:xs) = (f x):(myMap f xs)
```

Aufgabe

Bringen Sie mithilfe des Continuation - Patterns die Funktion `map` in eine endrekursive Form.

```haskell
myMapC :: (a -> b) -> [a] -> [b]
myMapC f = mmc id
  where
    mmc g [] = g []
    mmc g (x:xs) = mmc (\as -> g ((f x):as)) xs
```