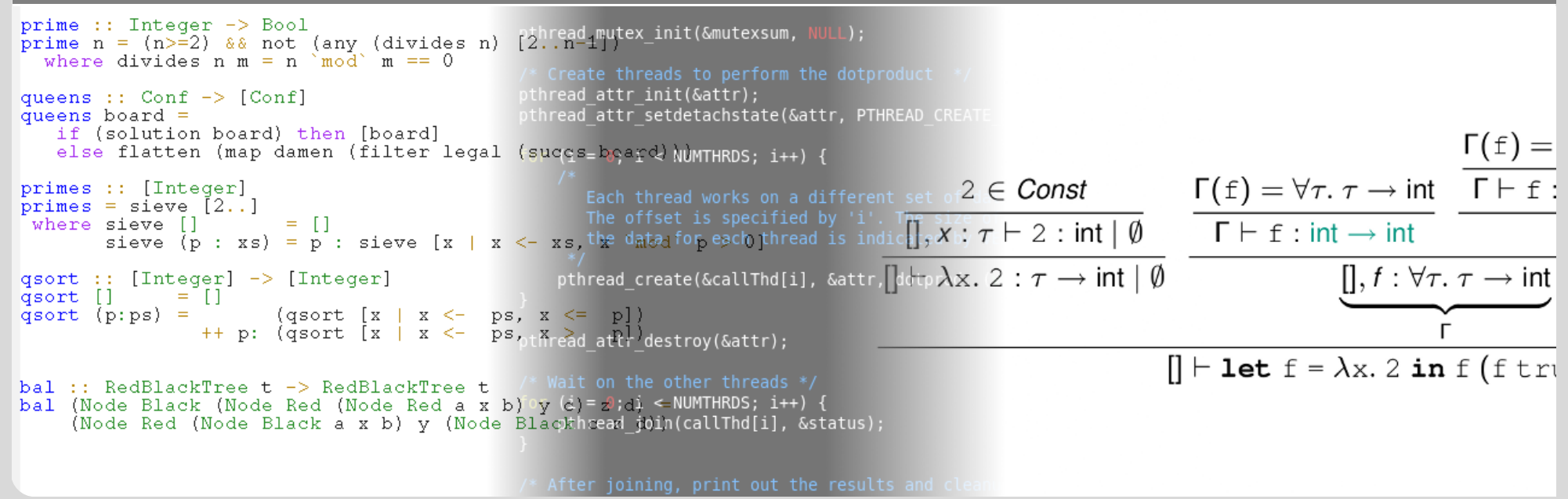
Karlsruher Institut für Technologie

Programmierparadigmen
Tutorium: Listen in Haskell

Prof. Dr.-Ing. Gregor Snelting | WS 2012/13

LEHRSTUHL PROGRAMMIERPARADIGMEN



Teil I

Listen in Haskell

Listen

Listen

Ein Liste ist entweder

1. die leere Liste `[]`, oder
2. eine Liste `(x:xs)`, konstruiert aus Restliste `xs` und Listenkopf `x`

Primitive Rekursion: Rekursion über diese Struktur von Listen

- Genau eine Gleichung pro Listen-Konstruktor (`[]` bzw. `(:)`)

```
app []     r = r
app (x:xs) r = x:(app xs r)
```

allgemeinere Rekursionsschemen: komplexere Konstruktoren-*Pattern*

- Beliebige Schachtelung von (Listen-)Konstruktoren

```
odds []       = []
odds [x]      = [x]
odds (x:y:xs) = x : odds xs
```

```
odds []       = []
odds (x:[])   = [x]
odds (x:y:xs) = x : odds xs
```

Haskell-Style Listen in Java

```
package list;

public abstract class List {
  List() {}
  public abstract boolean isNull();
  public abstract int head();
  public abstract List tail();
}


public final class ListFunctions {
  public static int length(List l) {
    if (l.isNull())  return 0;
    else return 1 + length(l.tail());
  }
}
```

- Instanzen von `Null` oder `Cons` (keine weitere Unterklassen)
- jeweils genau ein Konstruktor
- Erzeugen von Listen: verschachtelte Konstruktoren

```
new Cons(1,new Cons(2, new Cons(3,new Null())));
```

- Listen sind *immutable*

```
public final class Null extends List {
  public Null() {}
  public int head() {
    throw new RuntimeException();
  }
  public List tail() {
    throw new RuntimeException();
  }
  public boolean isNull() { return true; }
}
```

```
public final class Cons extends List {
  private final int x;
  private final List xs;

  public Cons(int x, List xs) {
    assert xs!=null;
    this.x = x; this.xs=xs;
  }
  public int  head() { return x; }
  public List tail() { return xs; }
  public boolean isNull() { return false; }
}
```

Haskell-Listen $\neq$ Arrays

Gegeben: Histogramm von Klausurpunkten, z.b. `[3,0,4,5,8,12,2]`

- 3 mal 0 Punkte, 0 mal 1 Punkt, ...
- Berechne: Gesamtpunkte, Zahl bestandener Prüfungen (Punkte ≥ 5)

Zellen-Zugriff: immer nur auf vorderstes Element von *Teil*liste

$\Rightarrow$Index in *Gesamt*liste unbekannt!

```
static int total(int[] points){
  int t = 0;
  for(int i=0; i<points.length ; i++){
    t+=i*points[i];
  }
  return t;
}
```

```
total []     = 0
total (x:xs) = ???
```

Zellen-Zugriff: nicht beliebig per Index, nur nacheinander

```
static int passed(int[] points){
  int p = 0;
  for(int i=5; i<points.length ; i++){
    p+=points[i];
  }
  return p;
}
```

```
passed []     = 0
passed (x:xs) = ???
```

Haskell-Listen $\simeq$ Iteratoren

Gegeben: Histogramm von Klausurpunkten, z.b. `[3,0,4,5,8,12,2]`

- 3 mal 0 Punkte, 0 mal 1 Punkt, . . .
- Berechne: Gesamtpunkte, Zahl bestandener Prüfungen (Punkte ≥ 5)

Zellen-Zugriff: immer nur auf vorderstes Element von *Teil*liste

$\Rightarrow$Index manuell mitführen

```
static int total(Iterable<Integer> points){
  int t = 0; int i=0;
  for(Integer x : points){
    t+=i*x;
    i++;
  }
  return t;
}
```

Zellen-Zugriff: vorhergehende Elemente „droppen“

```
static int passed(Iterable<Integer> points){
  int p = 0; int i=0;
  Iterator<Integer> iter = points.iterator();
  while(iter.hasNext() && i++ < 5) iter.next();
  while(iter.hasNext()) p+=iter.next();
  return p;
}
```

Haskell-Listen $\simeq$ Iteratoren

Gegeben: Histogramm von Klausurpunkten, z.b. `[3,0,4,5,8,12,2]`

- 3 mal 0 Punkte, 0 mal 1 Punkt, . . .
- Berechne: Gesamtpunkte, Zahl bestandener Prüfungen (Punkte ≥ 5)

Zellen-Zugriff: immer nur auf vorderstes Element von *Teil*liste

$\Rightarrow$Index manuell mitführen

```
static int total(Iterable<Integer> points){
  int t = 0; int i=0;
  for(Integer x : points){
    t+=i*x;
    i++;
  }
  return t;
}
```

```
total points = (wSumFrom 0 points)
 where wSumFrom i []     = 0
       wSumFrom i (x:xs) = i*x +
                           wSumFrom (i+1) xs
```

Zellen-Zugriff: vorhergehende Elemente „droppen“

```
static int passed(Iterable<Integer> points){
  int p = 0; int i=0;
  Iterator<Integer> iter = points.iterator();
  while(iter.hasNext() && i++ < 5) iter.next();
  while(iter.hasNext()) p+=iter.next();
  return p;
}
```

```
passed  points = sum (drop 5 points)
  where drop 0 l      = l
        drop i []     = []
        drop i (x:xs) = drop (i-1) xs
```

Haskell-Listen $\simeq$ Iteratoren

Gegeben: Histogramm von Klausurpunkten, z.b. `[3,0,4,5,8,12,2]`

- 3 mal 0 Punkte, 0 mal 1 Punkt, . . .
- Berechne: Gesamtpunkte, Zahl bestandener Prüfungen (Punkte ≥ 5)

Zellen-Zugriff: immer nur auf vorderstes Element von *Teil*liste

$\Rightarrow$Listenelemente mit Index annotieren

```
static int total(Iterable<Integer> points){
  int t = 0; int i=0;
  for(Integer x : points){
    t+=i*x;
    i++;
  }
  return t;
}
```

```
total points =
  sum (zipWith (*) [0..] points)

total points =
  sum [i*x | (i,x) <- zip [0..] points]
```

Zellen-Zugriff: vorhergehende Elemente „droppen“

```
static int passed(Iterable<Integer> points){
  int p = 0; int i=0;
  Iterator<Integer> iter = points.iterator();
  while(iter.hasNext() && i++ < 5) iter.next();
  while(iter.hasNext()) p+=iter.next();
  return p;
}
```

```
passed  points = sum (drop 5 points)
  where drop 0 l      = l
        drop i []     = []
        drop i (x:xs) = drop (i-1) xs
```

Teil II

Endrekursion

Programm Beschleunigen

Gegeben: Fakultätsfunktion

```
fak n
    | n == 0     = 1
    | otherwise = n * fak (n - 1)
```

Ziel: Effiziente Berechnung.

Programm Beschleunigen

Gegeben: Fakultätsfunktion

```
fak n
    | n == 0     = 1
    | otherwise = n * fak (n - 1)
```

Ziel: Effiziente Berechnung.

Vorlesung: Endrekursive Variante ist schneller.

```
fak n = fakAcc n 1
    where fakAcc n acc
        | n == 0     = acc
        | otherwise = fakAcc (n-1) (n*acc)
```

Programm Beschleunigen

Gegeben: Fakultätsfunktion

```
fak n
    | n == 0     = 1
    | otherwise = n * fak (n - 1)
```

Ziel: Effiziente Berechnung.

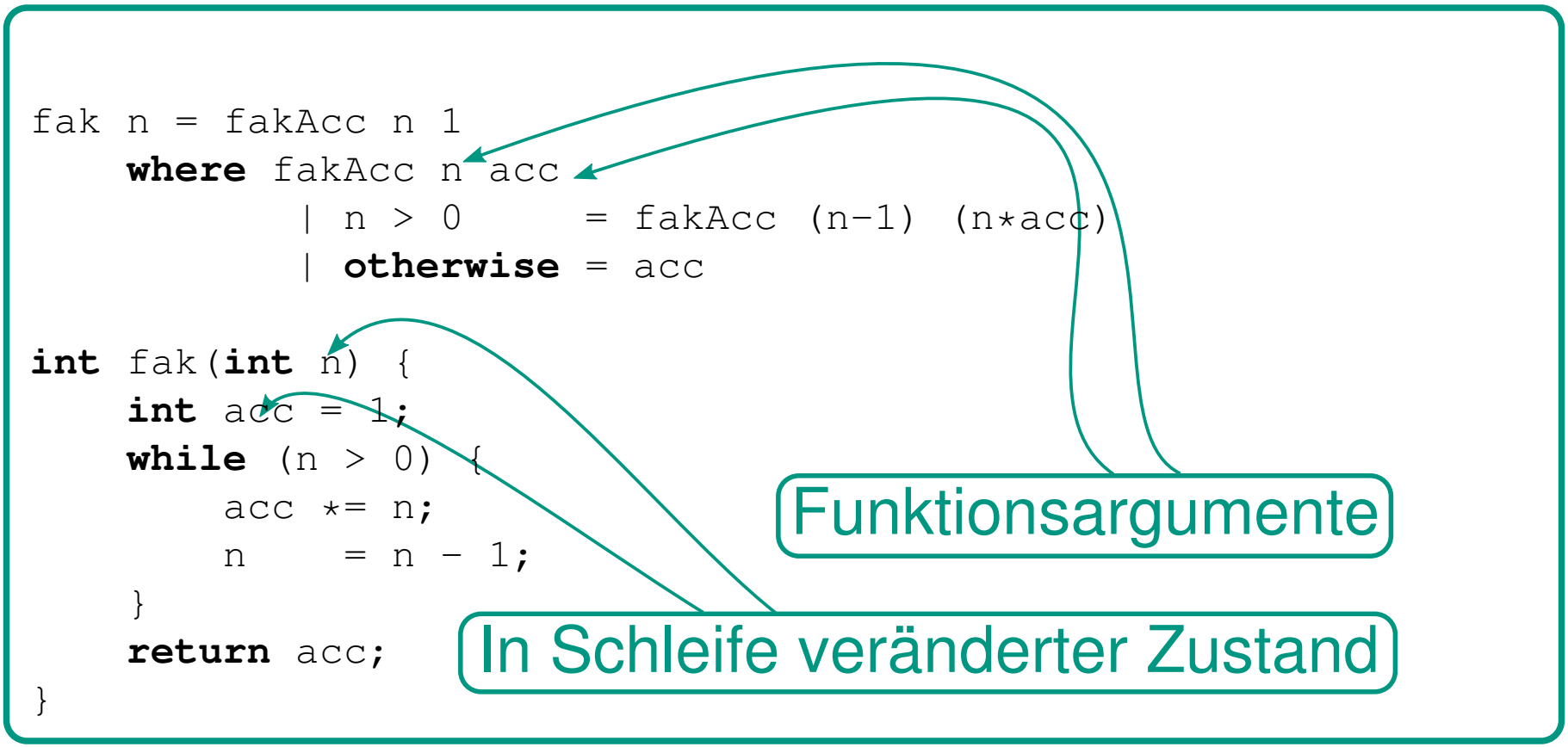
Vorlesung: Endrekursive Variante ist schneller.

```
fak n = fakAcc n 1
    where fakAcc n acc
            | n > 0      = fakAcc (n-1) (n*acc)
            | otherwise = acc
```

Endrekursion und `while`

```
fak n = fakAcc n 1
    where fakAcc n acc
            | n > 0     = fakAcc (n-1) (n*acc)
            | otherwise = acc

int fak(int n) {
    int acc = 1;
    while (n > 0) {
        acc *= n;
        n    = n - 1;
    }
    return acc;
}
```

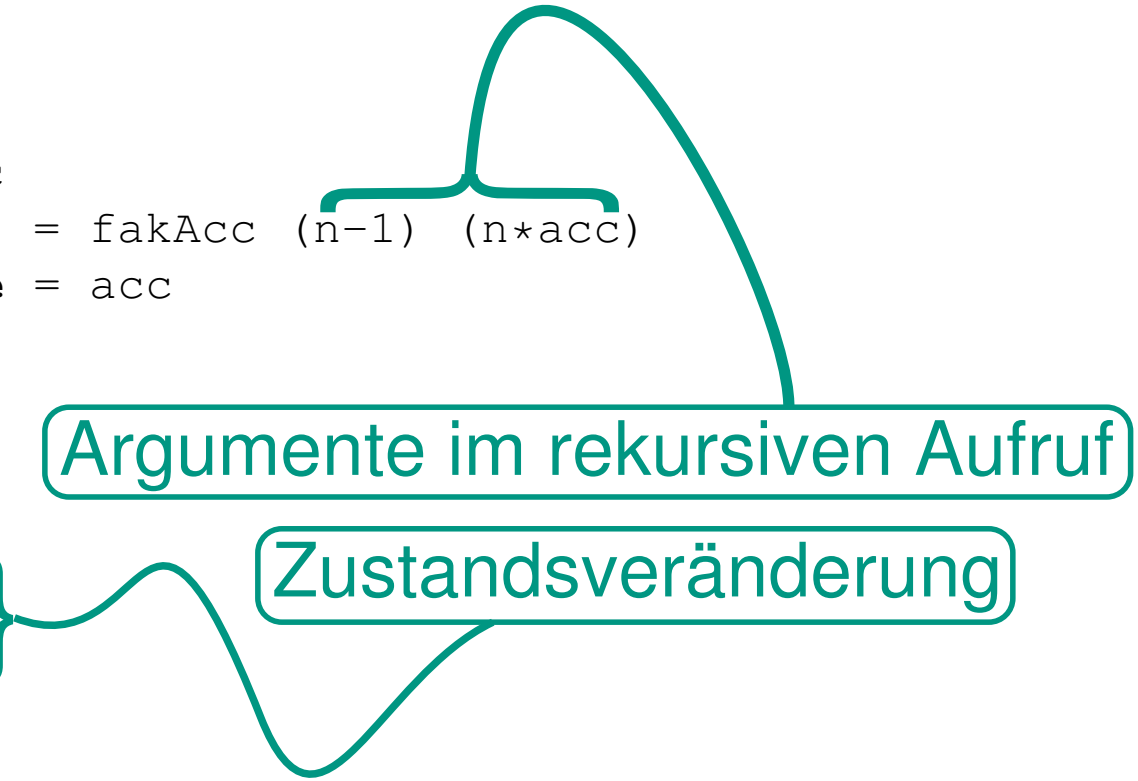


- Endrekursive Funktion $\simeq$ `while`-Schleife
- Akkumulator $\simeq$ lokale Hilfsvariable
- Funktionsargumente $\simeq$ in `while`-Schleife veränderter Zustand

Endrekursion und `while`

```
fak n = fakAcc n 1
    where fakAcc n acc
            | n > 0     = fakAcc (n-1) (n*acc)
            | otherwise = acc

int fak(int n) {
    int acc = 1;
    while (n > 0) {
        acc *= n;
        n    = n - 1;
    }
    return acc;
}
```

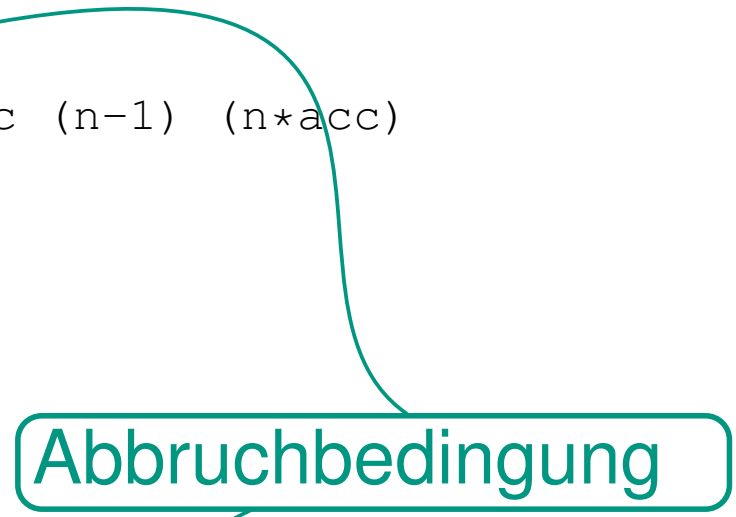


- Endrekursive Funktion $\simeq$ `while`-Schleife
- Akkumulator $\simeq$ lokale Hilfsvariable
- Funktionsargumente $\simeq$ in `while`-Schleife veränderter Zustand

Endrekursion und `while`

```
fak n = fakAcc n 1
    where fakAcc n acc
            | n > 0      = fakAcc (n-1) (n*acc)
            | otherwise = acc

int fak(int n) {
    int acc = 1;
    while (n > 0) {
        acc *= n;
        n    = n - 1;
    }
    return acc;
}
```

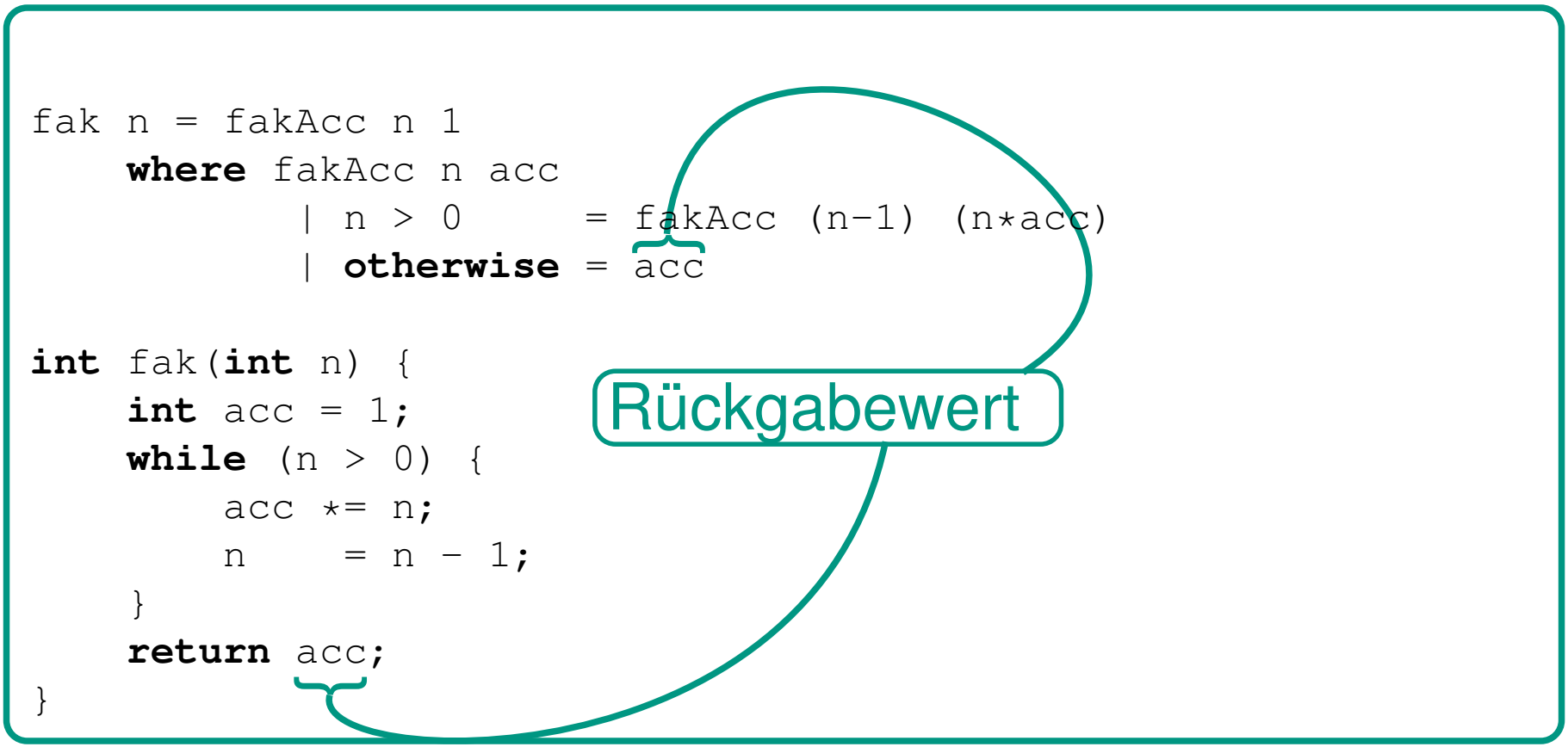


- Endrekursive Funktion $\simeq$ `while`-Schleife
- Akkumulator $\simeq$ lokale Hilfsvariable
- Funktionsargumente $\simeq$ in `while`-Schleife veränderter Zustand

Endrekursion und `while`

```
fak n = fakAcc n 1
    where fakAcc n acc
            | n > 0      = fakAcc (n-1) (n*acc)
            | otherwise = acc

int fak(int n) {
    int acc = 1;
    while (n > 0) {
        acc *= n;
        n    = n - 1;
    }
    return acc;
}
```



- Endrekursive Funktion $\simeq$ `while`-Schleife
- Akkumulator $\simeq$ lokale Hilfsvariable
- Funktionsargumente $\simeq$ in `while`-Schleife veränderter Zustand

Endrekursion – Warum?

Ohne Akkumulator

```
Stack s = new Stack();

int fak(int n) {
    while (n > 0) {
        push(n);
        n = n - 1;
    }

    int res = 1;

    while (!s.empty()) {
        res *= s.pop();
    }

    return res;
}
```

Mit Akkumulator

```
int fak(int n) {
    int acc = 1;

    while (n > 0) {
        acc *= n;
        n    = n - 1;
    }

    return acc;
}
```

Endrekursion – Zusammenfassung

Zusammenfassung:

- Endrekursion + Akkumulator $\simeq$ while + lokale Variablen
- Reduzierter Speicherverbrauch