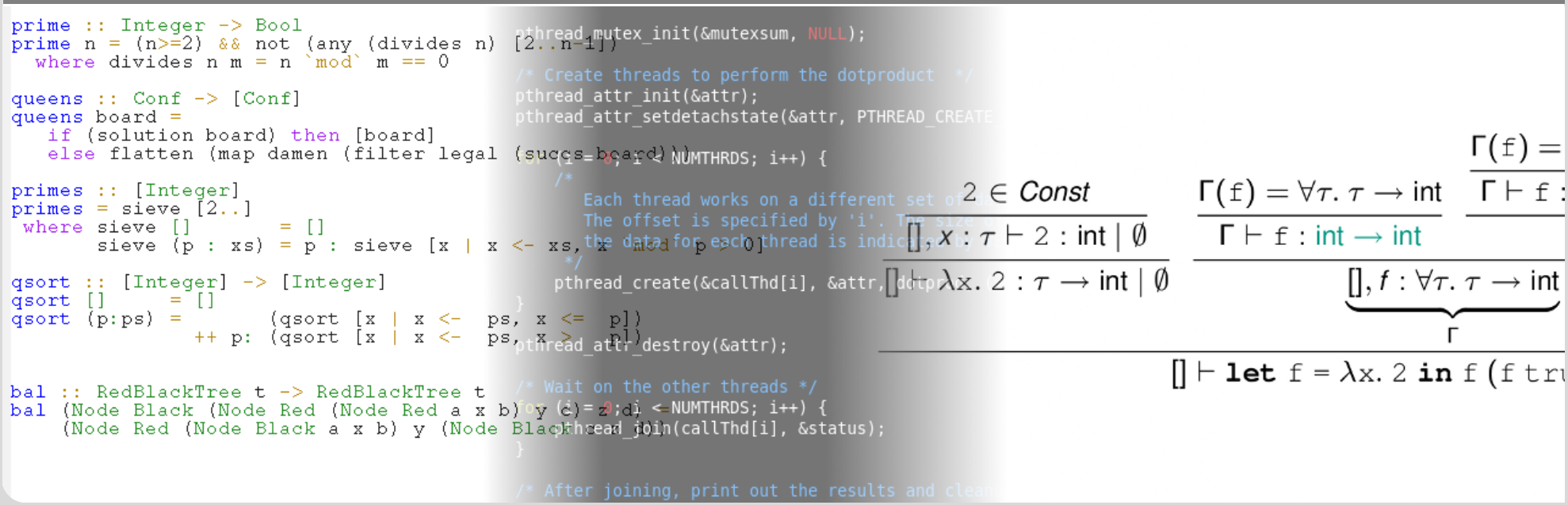
Karlsruher Institut für Technologie

Programmierparadigmen
Tutorium: Erste Schritte in Haskell

Prof. Dr.-Ing. Gregor Snelting | WS 2013/14

LEHRSTUHL PROGRAMMIERPARADIGMEN



Einfache Funktionen in Haskell

Größter gemeinsamer Teiler per *remainder*

Sei $r > 0$ der Rest der Division a/b zweier natürlicher Zahlen $a, b > 0$.
Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, r*

- Definiere zweistellige Haskell-Funktion `gcdr` zur Berechnung des größten gemeinsamen Teilers zweier Zahlen größer 0

Größter gemeinsamer Teiler per *remainder*

Sei $r > 0$ der Rest der Division a/b zweier natürlicher Zahlen $a, b > 0$.
Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, r*

- Definiere zweistellige Haskell-Funktion `gcdr` zur Berechnung des größten gemeinsamen Teilers zweier Zahlen größer 0

Lösungsidee: $gcd\,(a, b) = gcd\,(b, r)$,
außerdem: $gcd\,(a, b) = b$ falls *b teilt a*

Größter gemeinsamer Teiler per *remainder*

Sei $r > 0$ der Rest der Division a/b zweier natürlicher Zahlen $a, b > 0$.
Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, r*

- Definiere zweistellige Haskell-Funktion `gcdr` zur Berechnung des größten gemeinsamen Teilers zweier Zahlen größer 0

Lösungsidee: $gcd\,(a, b) = gcd\,(b, r)$,
außerdem: $gcd\,(a, b) = b$ falls *b teilt a*

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

Größter gemeinsamer Teiler per *subtraction*

Sei $a > b > 0$. Dann gilt für $x \in \mathbb{N}$:

$$x \textit{ ist gemeinsamer Teiler von } a, b \text{ genau dann wenn } x \textit{ ist gemeinsamer Teiler von } b, (a - b)$$

Sei $b > a > 0$. Dann gilt für $x \in \mathbb{N}$:

$$x \textit{ ist gemeinsamer Teiler von } a, b \text{ genau dann wenn } x \textit{ ist gemeinsamer Teiler von } a, (b - a)$$

- Definiere zweistellige Haskell-Funktion `gcds` zur Berechnung des größten gemeinsamen Teilers zweier `Zahlen` größer 0

Größter gemeinsamer Teiler per *subtraction*

Sei $a > b > 0$. Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, (a − b)*

Sei $b > a > 0$. Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von a, (b − a)*

- Definiere zweistellige Haskell-Funktion `gcds` zur Berechnung des größten gemeinsamen Teilers zweier `Zahlen` größer 0

Lösungsidee: $gcd\,(a, b) = gcd\,(b, a - b)$ bzw.
$gcd\,(a, b) = gcd\,(a, b - a)$, außerdem: $gcd\,(a, a) = a$

Größter gemeinsamer Teiler per *subtraction*

Sei $a > b > 0$. Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, (a − b)*

Sei $b > a > 0$. Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von a, (b − a)*

- Definiere zweistellige Haskell-Funktion `gcds` zur Berechnung des größten gemeinsamen Teilers zweier `Zahlen` größer 0

Lösungsidee: $gcd\,(a, b) = gcd\,(b, a - b)$ bzw. $gcd\,(a, b) = gcd\,(a, b - a)$, außerdem: $gcd\,(a, a) = a$

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

Terminierung rekursiver Funktionen

Rekursion

Grundidee Rekursion:

- Löse „kleine“ Problem-Instanzen *direkt*
- Reduziere „große“ Problem-Instanzen auf „kleinere“ (auch: mehrere Instanzen, siehe `fib`)

Fakultätsfunktion

```
fak n   = if (n==0) then 1 else n * fak (n-1)
```

- „kleine“ Problem-Instanz: $n = 0$
- „groß“ Problem-Instanzen: $n > 0$, reduziere auf $n - 1$

Fakultätsfunktion (Endrekursiv)

```
fak n a = if (n==0) then a else fak (n-1) (n*a)
```

- genauso

Wichtig: Reduktion „groß“ auf „kleiner“ nicht unendlich oft möglich

- klar bei n nach $n - 1$

Rekursion *gcdr*

Rekursion in `gcdr`: Inwiefern ist das Problem

„Berechne *gcd* von *b*, *r*“

„kleiner“ als

„Berechne *gcd* von *a*, *b*“?

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

Rekursion *gcdr*

Rekursion in `gcdr`: Inwiefern ist das Problem

„Berechne *gcd* von *b*, *r*“

„kleiner“ als

„Berechne *gcd* von *a*, *b*“?

- zweites Argument von *gcd* wird kleiner, denn: $r < b$
- „kleines“ Problem: zweites Argument $= 0$

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

Rekursion *gcds*

Rekursion in `gcds`: Inwiefern ist das Problem

„Berechne *gcd* von $a - b, b$ bzw. von $a, b - a$“

„kleiner“ als

„Berechne *gcd* von a, b“

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

Rekursion *gcds*

Rekursion in `gcds`: Inwiefern ist das Problem

„Berechne *gcd* von $a - b, b$ bzw. von $a, b - a$“

„kleiner“ als

„Berechne *gcd* von a, b“

- eins der beiden Argumente kleiner, *und* das anderes bleibt gleich
- $\Rightarrow$ Problem wird nicht unendlich oft „kleiner“ gemacht!
- „kleine“ Probleme: solche mit $a = b$

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

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.

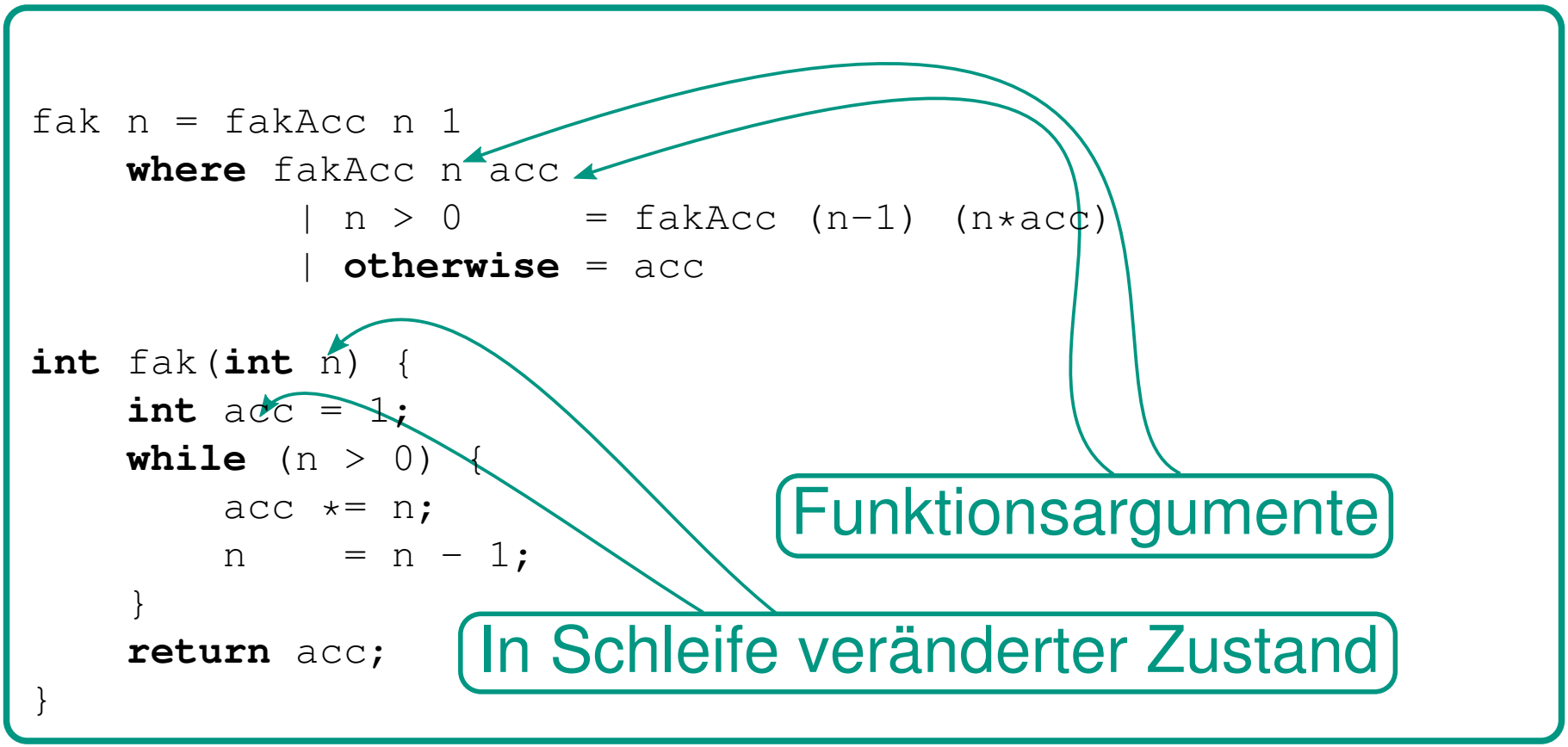
Vorlesung: Endrekursive Variante ist schneller.

```
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;
}
```

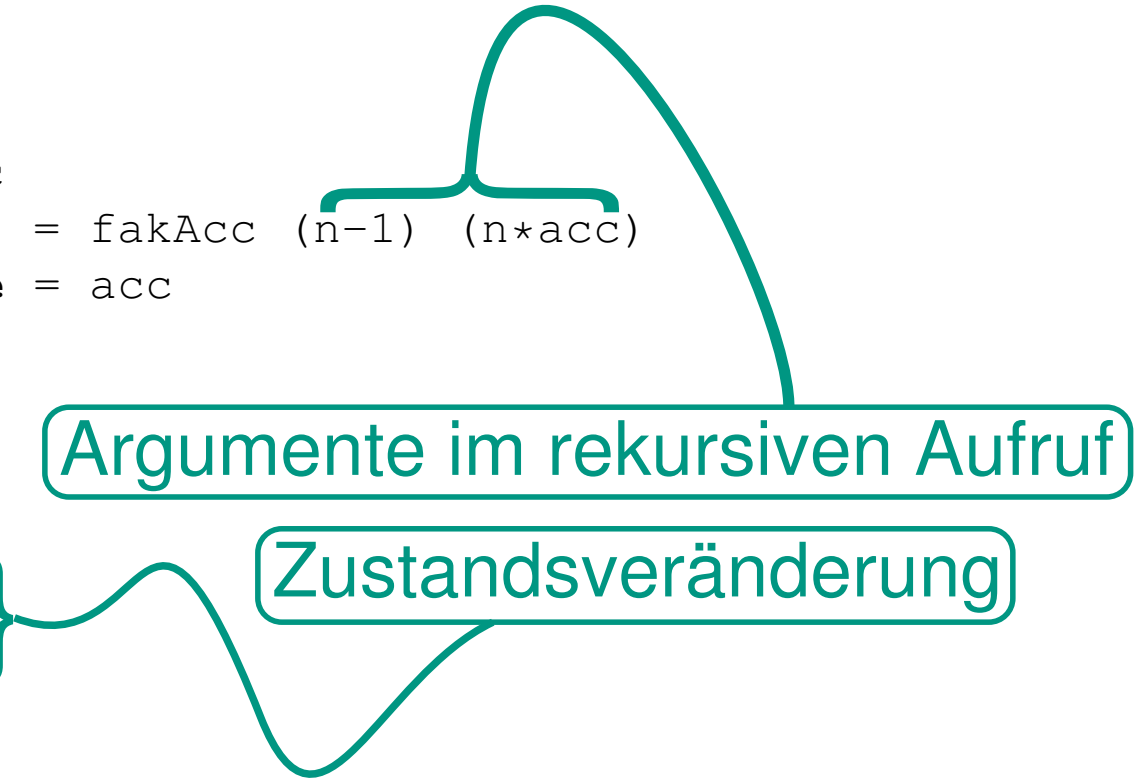


- Endrekursive Funktion $\simeq$ `while`-Schleife
- Akkumulator $\simeq$ lokale Hilfsvariable
- Funktionsargumente $\simeq$ in `while`-Schleife veränderter Zustand

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;
}
```

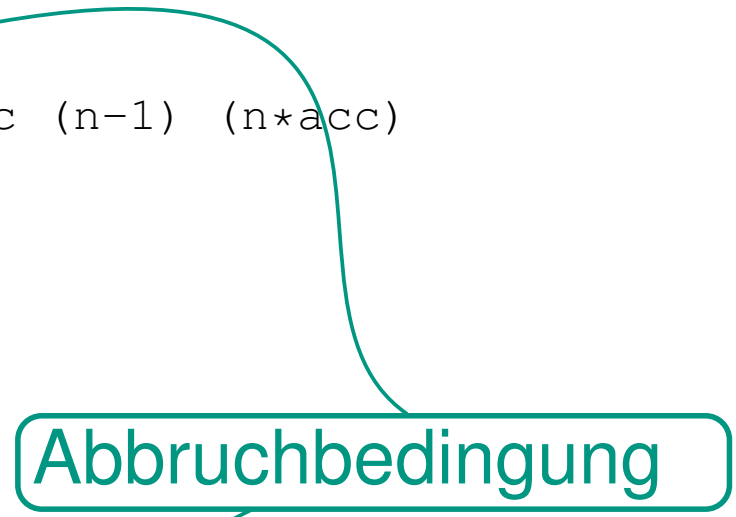


- Endrekursive Funktion $\simeq$ `while`-Schleife
- Akkumulator $\simeq$ lokale Hilfsvariable
- Funktionsargumente $\simeq$ in `while`-Schleife veränderter Zustand

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;
}
```

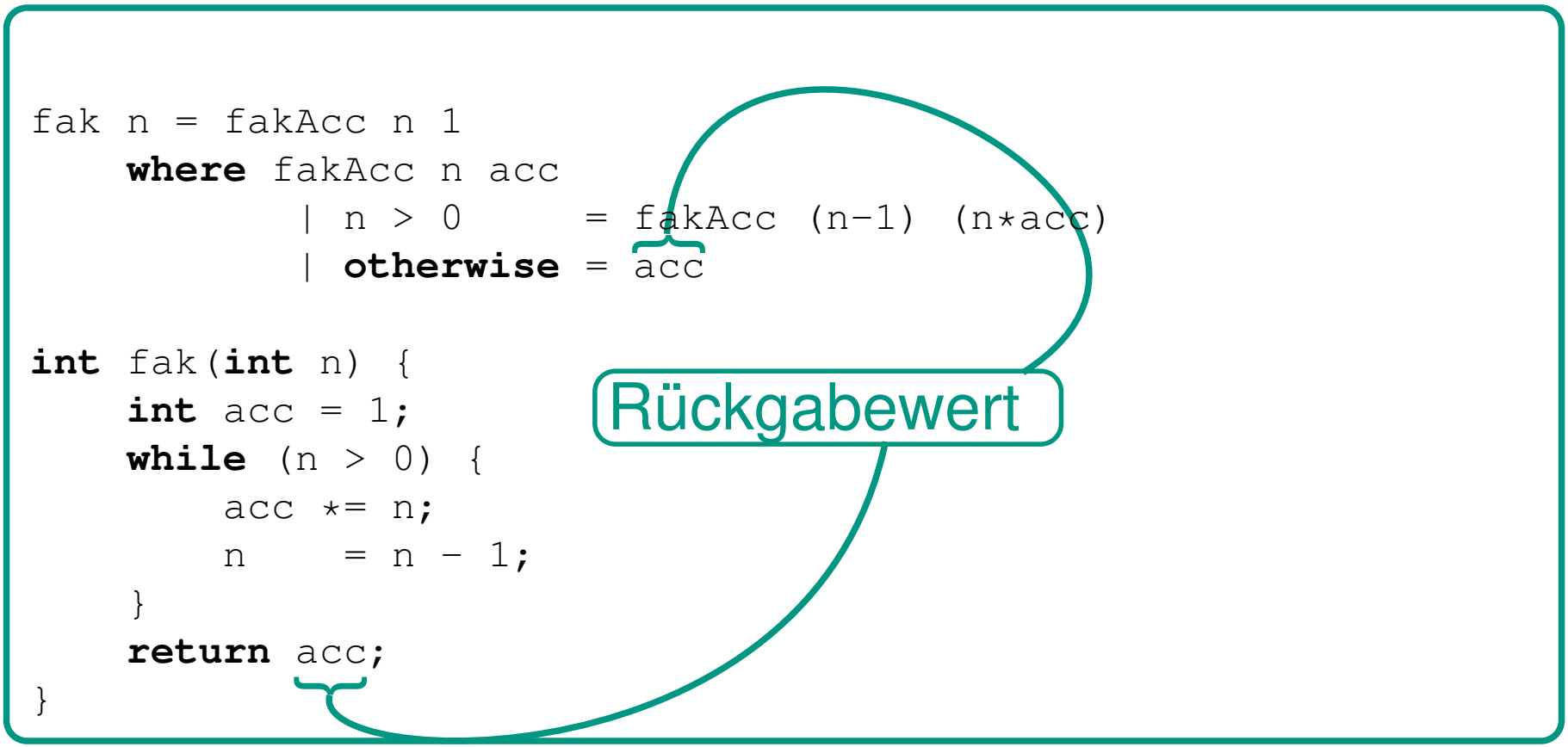


- Endrekursive Funktion $\simeq$ `while`-Schleife
- Akkumulator $\simeq$ lokale Hilfsvariable
- Funktionsargumente $\simeq$ in `while`-Schleife veränderter Zustand

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