Ein- und Ausgabe

- Zweite Möglichkeit: Ausgabe direkt auf dem Bildschirm durchführen:

```
fun printTree printA t =
 case t of Leaf a => (print "Leaf "; printA a)
         | Node(l,a,r) =>
                    (print "Node(";
                     printTree printA l;
                     print ",";
                     printA a;
                     print ",";
                     printTree printA r;
                     print ")")
val printTree = fn : ('a -> unit) -> 'a Tree -> unit
```

Ein- und Ausgabe aus Dateien

Zur Ein- und Ausgabe aus Dateien stellt das Modul `TextIO` eine Kollektion von Typen und Funktionen zur Verfügung:

```
type elem = char
type vector = string
type instream
type outstream

val stdIn : instream
val stdOut : outstream
val stdErr : outstream
```

Einlesen aus Text-Dateien

Der typ `instream` repräsentiert Dateien, aus denen man nur lesen kann. Als Sonderfall kann man auch einen String zum Lesen öffnen.

```
val openIn : string -> instream
val openString : string -> instream
val closeIn : instream -> unit
val input1 : instream -> elem option
val inputN : instream * int -> vector
val endOfStream : instream -> bool
```

Schreiben in Text-Dateien

Ein `outstream` dagegen dient zum Schreiben:

```
val openOut : string -> outstream
val openAppend : string -> outstream
val closeOut : outstream -> unit
val output : outstream * vector -> unit
val output1 : outstream * elem -> unit
```

4.12.5 Continuations

Rekursionsarten

Je nach Art der rekursiven Aufrufe unterscheiden wir folgende Arten von Rekursion:

- End-Rekursion(lineare Rekursion) Bei der Funktionsauswertung gibt es nur einen rekursiven Aufruf. Dieser ist gleichzeitig der Rückgabewert.

```
fun fac1 (n,acc) = if n=1 then acc
                   else fac1(n-1,n*acc)

fun loop x = if x<2 then x
             else if x mod 2 = 0 then loop(x div 2)
                  else loop(3*x+1)
```

Rekursionsarten

- Repetitive Rekursion Es gibt nur einen rekursiven Aufruf, der aber nicht der Rückgabewert ist.

```
fun fac n = if n=1 then 1 else n*(fac(n-1))
```

- Baumartige (kaskadenartige) Rekursion Es gibt mehrere, nicht verschachtelte rekursive Aufrufe.

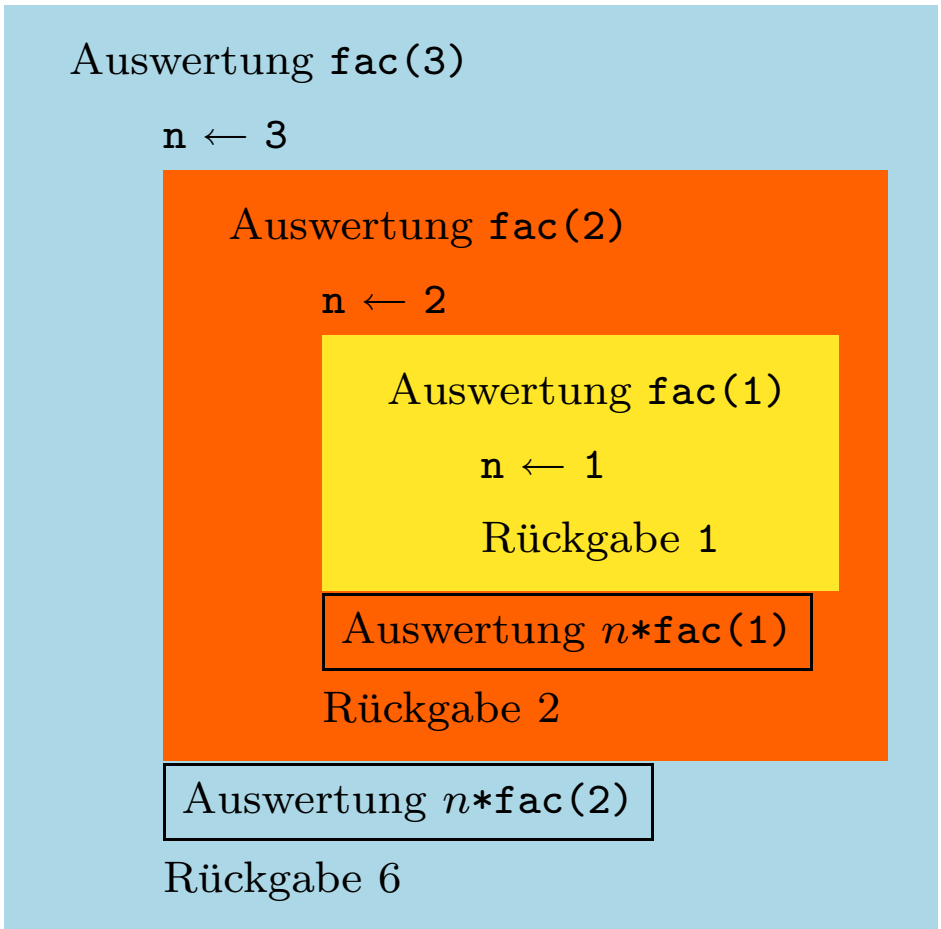
```
fun fib n = if n=0 then 1
            else if n=2 then 1
                 else fib(n-1)+fib(n-2)
```

- Wilde Rekursion: geschachtelte rekursive Aufrufe

```
fun f n = if n<2 then n else f(f(n div 2))
```

Speicherverhalten der repetitiv-rekursiven Funktionen

```
fun fac n = if n=1 then 1 else n*(fac(n-1))
```

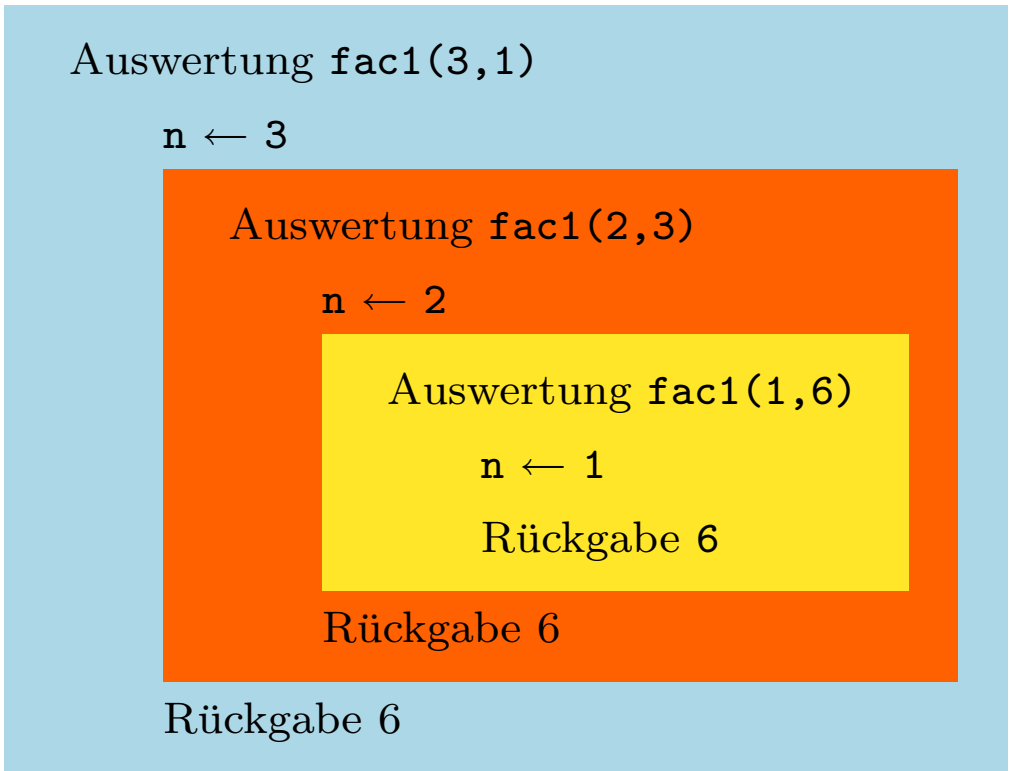


Bei jedem neuen Funktionsauruf muss der aktuelle Aufruf seine lokale Variablen speichern, um diese nach dem Aufruf benutzten zu können

$\Longrightarrow$ Speicherverbrauch wächst mit Anzahl geschachtelter Aufrufe.

Speicherverhalten der tail-rekursiven Funktionen

```
fun fac1 (n,acc) = if n=1 then acc else fac1(n-1,n*acc)
```



Rekursiver Aufruf = Rückgabewert
$\Longrightarrow$ Keine Berechnung nach denm Aufruf
$\Longrightarrow$ Lokale Variablen werden nicht mehr gebraucht
$\Longrightarrow$ müssen nicht gespeichert werden

$\Longrightarrow$ Tail-Rekursion hat den besten Speicherverhalten. Es wird zur wilden Rekursion hin immer schlechter.

Eliminierung der repetitiven Rekursion

Der typische Fall von repetitiver Rekursion hat die folgende Form:

```
fun f x = if x = <x0> then <v0> else <e(x,f(g(x)))>
```

Zum Beispiel bei Fakultät:

```
fun fac x = if x = 1 then 1 else x*(fac(x-1))
```

Hier ist x_0=1, v_0=1, g(x)=x-1 und e(x,y)=x*y. Wie wir schon wissen, gibt es auch eine end-rekursive Variante:

```
fun fac1(x,a) = if x=1 then a else fac1(x-1,x*a)
```

Im allgemeinen Fall kann man f ersetzen durch:

```
fun f1(x,a) = if x = <x0> then a else f1(g(x),e(x,a))
```

Eliminierung der repetitiven Rekursion

```
fun f x = if x = <x0> then <v0> else <e(x,f(g(x)))>
fun f1(x,a) = if x = <x0> then a else f1(g(x),e(x,a))
```

$$\begin{aligned} f(x) &= e(x, e(g(x), e(g(g(x), \ldots e(g^n(x), v_0) \ldots))) \\ &= x \,\square_e\, (g(x) \,\square_e\, (g^2(x) \,\square_e \ldots (g^n(x) \,\square_e\, v_0) \ldots)) \end{aligned}$$

$$\begin{aligned} f1(x, v_0) &= f1(g(x), e(x, v_0)) = f1(g(g(x)), e(g(x), e(x, v_0))) = \ldots \\ &= e(g^n(x), e(g^{n-1}(x), \ldots e(g(x), e(x, v_0)) \ldots)) \\ &= g^n(x) \,\square_e\, (g^{n-1}(x) \,\square_e \ldots (g(x) \,\square_e\, (x \,\square_e\, v_0)) \ldots) \end{aligned}$$

$\Longrightarrow$ $fx = f1(x, v_0)$, wenn `e` = assoziativ, kommutativ, z.B.:

```
n*((n-1)*(...*(2*1))) = 1*(2*(...*((n-1)*n)))
```

Der zusätzliche Parameter `a` heißt auch Akkumulator.

Rekursionsarten

Vorsicht bei Listenfunktionen:

```
fun f  x      = if x=0 then nil else x::f(x-1)
fun f1 (x,a) = if x=0 then a   else f1(x-1,x::a)
```

Der Listenkonstruktor `::` is weder kommutativ noch assoziativ.

Deshalb berechnen `f x` und `f1 (x,nil)` nicht den gleichen Wert:

```
- f 5;
val it = [5,4,3,2,1] : int list
- f1(5,nil);
val it = [1,2,3,4,5] : int list
```

Rekursionsarten

Selbst baumartige Rekursion kann manchmal linearisiert werden:

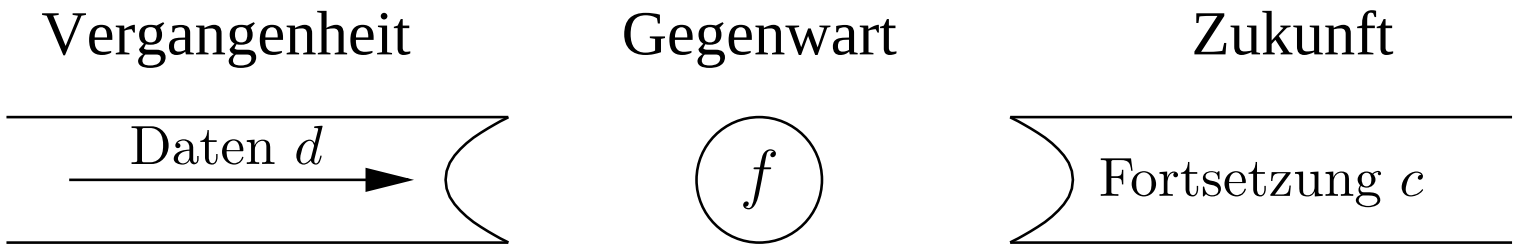
```
fun fib n = case n of 0 => 0
                    | 1 => 1
                    | n => fib (n-1) + fib (n-2)

fun fib1 n =
    let fun iter (m,f1,f2) =
            if m = n then f1 else iter(m+1,f2,f1+f2)
    in iter(0,0,1)
    end
```

Das läßt sich aber nicht so einfach verallgemeinern!

Continuation-passing Style

Continuation-passing Style (CPS) (dt. etwa Fortsetzungen-Durchreichen-Programmierstil)

CPS ≡ Abstraktion einer Berechnung:



Vergangenheit = Daten d die der aktuellen Verarbeitungsfunktion übergeben werden

Gegenwart = Eine *aktuelle* Verarbeitungsfunktion f

Zukunft = Eine Fortsetzungsfunktion c, die den Rest der Berechnung beschreibt (continuation)

- Die Berechnung liefert $c(f(daten))$.

Continuation-passing Style

Anstelle der Rückgabe eines Wertes als Ergebnis einer Funktion f, wird eine Funktion k mit diesem Wert aufgerufen, die explizit angibt, wie die Berechnung fortgesetzt werden soll.

f bekommt die Fortsetzungfunktion(en) als zusätzliche Parameter.

Berechnung von $a + b * c$ ohne Continuations:

```
fun add (x,y) = x + y
fun mult (x,y) = x * y
fun compute (a,b,c) = add(a,mult(b,c))
```

Berechnung von $a + b * c$ im Continuation-Passing-Style:

```
fun add1 ((x,y),k) = k (add (x,y))
fun mult1((x,y),k) = k (mult (x,y))
fun compute ((a,b,c),k) = mult1((b,c),fn x => add1((a,x),k))
```

Continuation Passing Style: Beispiel

Berechnung von $a * b + c * d$ mit CPS:

```
fun add1 ((x,y),k) = k (add (x,y))
val add1 = fn : (int * int) * (int -> 'a) -> 'a
fun mult1((x,y),k) = k (mult (x,y))
val mult1 = fn : (int * int) * (int -> 'a) -> 'a

fun compute ((a,b,c,d),k) =
  mult1((a,b),
        fn x => mult1((c,d),
                      fn y => add1((x,y),k)))
val compute = fn : (int * int * int * int) * (int -> 'a) -> 'a

compute ((1,2,3,4),fn x=> x);
val it = 14 : int
```