Grundlagen der Programmierung 2

Haskell: Auswertung

Prof. Dr Manfred Schmidt-Schauß

Sommersemester 2013

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Beispiel: Aufrufhierarchie

```
quadrat x          =  x*x
quadratsumme x y = (quadrat x) + (quadrat y)
```

`quadratsumme` referenziert direkt die Funktion `quadrat`,
`quadratsumme` referenziert direkt die (eingebaute) Funktion $+$
`quadratsumme` referenziert die Funktionen $\{$`quadrat`$, *, +\}$

Aufrufhierarchie und Rekursive Definitionen

Jetzt die Definition dazu:

f, g, f_i seien Haskell-definierte Funktionen.

f **referenziert** g **direkt**,	wenn g im Rumpf von f vorkommt.
f **referenziert** g	wenn f referenziert direkt g, oder es gibt Funktionen $f_1, \ldots, f_n$, so dass gilt: f referenziert direkt f_1, f_1 referenziert direkt f_2, $\ldots$, f_n referenziert direkt g.
f ist **direkt rekursiv**,	wenn f sich selbst direkt referenziert.
f ist **rekursiv**,	wenn f sich selbst referenziert.
Verschränkte Rekursion:	wenn f die Funktion g referenziert und g die Funktion f. (auch für allgemeinere Fälle)

Beispiel: Aufrufhierarchie

```
quadrat x          =  x*x
quadratsumme x y = (quadrat x) + (quadrat y)
```

`quadratsumme` referenziert direkt die Funktion `quadrat`,
`quadratsumme` referenziert direkt die (eingebaute) Funktion $+$
`quadratsumme` referenziert die Funktionen $\{$`quadrat`$, *, +\}$

Die Funktion `quadratsumme` ist somit nicht rekursiv

Beispiel: Fakultät

$$\begin{array}{lcll} 0! & := & 1 & \\ n! & := & n * (n-1)! & \text{wenn } n > 0 \end{array}$$

Anwendung: $n!$ ist die Anzahl aller Permutationen aller Elemente einer n-elementigen Menge.

Beispiel:

Menge mit 3 Elementen $\{A, B, C\}$:
6 Permutationen: $ABC, ACB, BAC, BCA, CAB, CBA$
$3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 * 1 = 6$

Beispiel: Fakultät

rekursive Definition:

```
fakultaet:: Integer -> Integer
fakultaet x = if x <= 0 then 1
              else x*(fakultaet  (x-1))
```

Die Funktion fakultaet ist rekursiv, da sie im Rumpf der Definition sich selbst referenziert

Entwurf rekursiver Funktionen

zwei Fälle sind beim Entwurf rekursiver Funktionen zu beachten

der Basisfall:	keine weitere Rekursion
der Rekursionsfall:	neuer rekursiver Aufruf

Terminierung der Aufrufe für alle Eingabefälle sollte man prüfen.

Entwurf rekursiver Funktionen

Am Beispiel fakultaet:

der Basisfall:	Ergebnis: 0 wenn das Argument $x \leq 1$ ist.
der Rekursionsfall:	Ergebnis: `x*(fakultaet (x-1))`, wenn $x > 1$ ist.

Terminierung:

- Argumente werden mit jedem rekursiven Aufruf kleiner
  `fakultaet x` ruft `fakultaet (x-1)` auf für $x \geq 1$.
- Der Basisfall hat das kleinste Argument

```
*Main> fakultaet 3 ↵
6
*Main> fakultaet 40 ↵
815915283247897734345611269596115894272000000000
```

Eine falsche Definition

```
fakultaet_nt:: Integer -> Integer
fakultaet_nt x  = if x == 0 then 1
                  else x*(fakultaet_nt (x-1))
```

Diese Funktion terminiert nicht bei negativen Eingaben:
`fakultaet_nt (-5)` ruft `fakultaet_nt (-6)` auf usw.

Beispiel: größter gemeinsamer Teiler: ggt

```
ggt:: Integer -> Integer -> Integer
ggt a b =    if   b == 0
             then a
             else ggt b (rem a b)
```

Basisfall: ?
Rekursionsfall: ?

Auswertung von Programmen

Zeichen	Program
Bedeutung?	Effekte?

Auswertung von Haskell-Programmen

operationale Semantik von Haskell:

Transformationen eines Ausdrucks
(bzw. des main-Ausdrucks des Programms)

bis ein einfacher Ausdruck erreicht ist.

⇒ **Transformations-Semantik**

Eindeutige Festlegung der Ausführung durch eine Strategie:
normale Reihenfolge der Auswertung,
(Haskell benutzt eine verbesserte Variante)
applikative Reihenfolge der Auswertung,
(z.B. in Python, ML, OCaml, Lisp)

Auswertung von Haskell-Programmen

Prinzip der Berechnung in Haskell:

Auswertung	=	Folge von Transformationen eines Ausdrucks • in normaler Reihenfolge (Normalordnung) • unter Benutzung der Funktions-Definitionen bis ein Basiswert erreicht ist

Fehler-Möglichkeiten für Transformationsfolgen:

- Transformation bleibt stecken, aber kein Basiswert wird erreicht
  (i.a. verhindert durch Typisierung)
- Transformation terminiert nicht

Definitionseinsetzung (δ-Reduktion)

$f\ x_1 \ldots x_n = \mathrm{R}_f$ sei die Haskell-Definition von f

Auswertung:

$$(f\ t_1 \ldots t_n) \quad \to \quad (\mathtt{R}_f[t_1/x_1, \ldots t_n/x_n])$$

paralleles (unabhängiges) Ersetzen
von x_i durch t_i für $i = 1, \ldots, n$

Arithmetische Auswertungen

Zahl + Zahl	Ergebnis durch „Ausrechnen“ mittels vorprogrammierter Operationen
Zahl $*$ Zahl	genauso
...	

Beachte: Wenn $s\ op\ t$ arithmetisch ausgewertet werden soll,
dann müssen zuerst die Ausdrücke s und t
zu Zahlen ausgewertet werden!

Beispiel:

Programm:

```
main =  quadrat 5
quadrat x = x*x
```

Auswertung als Folge von Transformationen:

```
main
→            (Definitions-Einsetzung)
quadrat 5
→            (Definitions-Einsetzung)
5 * 5
→            (arithmetische Auswertung)
25
```

Boolesche Auswertungen

v `&&` w oder v `||` w oder `not` w

Definitionen von und, oder, nicht,
äquivalent zu den vordefinierten:

```
x && y = if x then y     else False
x || y = if x then True  else y
not x  = if x then False else True
```


Auswertung der Fallunterscheidung

Fallunterscheidung (if-Reduktion)
Definition:

$$\begin{array}{lcl} (\texttt{if True then } e_1 \texttt{ else } e_2) & \rightarrow & e_1 \\ (\texttt{if False then } e_1 \texttt{ else } e_2) & \rightarrow & e_2 \end{array}$$

Beachte

$$(\texttt{if b then } e_1 \texttt{ else } e_2)\text{-Auswertung}$$

erfordert automatisch, dass zuerst `b` ausgewertet werden muss.

Transformationen, Reduktionen

Wir nennen eine Transformation auch *Reduktion*
und eine Folge von Programmtransformationen auch
Reduktionsfolge (oder *Auswertung*).

Beachte: Reduktionen / Transformationen sind zunächst überall im Ausdruck erlaubt.
Erst eine Auswertungs-Strategie macht die Auswertung eindeutig.

Transformationen, Reduktionen

Motivation für diese Transformationssemantik:

- **eindeutige Festlegung der (Haskell-)Auswertung**
- **Umgang mit nichtterminierenden Argumenten**
- **Unendliche Listen**

Motivation für Normalordnung (s.u.):

- korrekte **Programmtransformationen**
- **einfache Programm-Logik**
- **Hohe Modularität ist möglich**

Beispiel

```
x && y = if x then y     else False
x || y = if x then True  else y
bot    = bot
```

Beispiel-Auswertungen dazu

```
True && True  → if True then True else False  → True
True && False → if True then False else False → False
True || True  → if True then True else True   → True
True || False → if True then True else False  → True

True && bot   → if True then bot else False   → bot → bot ...
True || bot   → if True then True else bot    → True
```


Transformationsmöglichkeiten

3 verschiedene Auswertungen für `quadrat (4+5)` :

$\texttt{quadrat}(4+5) \to (4+5)*(4+5) \to 9*(4+5) \to 9*9 \to 81$

$\texttt{quadrat}(4+5) \to (4+5)*(4+5) \to (4+5)*9 \to 9*9 \to 81$

$\texttt{quadrat}(4+5) \to \texttt{quadrat}\ 9 \to 9*9 \to 81$

Beobachtungen:

- Ergebnis ist gleich
- Anzahl der Reduktionen verschieden

Reduktionsstrategien

Wichtige *Reduktionsstrategien*

Applikative Reihenfolge:
Argumentauswertung vor Definitionseinsetzung

Normale Reihenfolge:
Definitionseinsetzung vor Argumentauswertung

Verzögerte Reihenfolge:
Normale Reihenfolge mit Sharing

Eindeutigkeit der Ergebnisse

Satz

Sei t ein Ausdruck in Haskell.

Wenn

$$t \xrightarrow{*,appl.R.} a,$$
$$t \xrightarrow{*,norm.R.} b$$
$$t \xrightarrow{*,verz.R.} c$$

und a, b, c sind Basiswerte (d.h. Integer, Boolesche Werte).

Dann gilt $a = b = c$.

Applikative Reihenfolge der Auswertung

- Argumentauswertung vor Definitionseinsetzung
  wird in den meisten Programmiersprachen benutzt
  z.B. in Python, C, ML, ...
- wird nicht in Haskell verwendet.
- Diskussion der Vor- / Nachteile später ...

Applikative Reihenfolge der Auswertung

Werte den Ausdruck t **applikativ** aus! Fälle:

- t ist Basiswert. fertig.
- $t \equiv s_1\ t_1$,
  Wenn s_1 kein Funktionsname und keine Anwendung,
  dann werte s_1 applikativ aus (rekursiv)
- $t \equiv f\ t_1 \ldots t_n$.
  Wenn $ar(f) \leq n$, dann (rekursiv) applikativ auswerten:
  t_i, $1 \leq i \leq ar(f)$ von links nach rechts.
  Wenn $ar(f) \leq n$ und alle t_i Basiswerte, dann Definitionseinsetzung (δ-Red
  Wenn $n < ar(f)$, dann fertig: keine Reduktion.
- $t \equiv$ `if` b `then` e_1 `else` e_2.
  Wenn b Basiswert, dann if-Reduktion
  Wenn b kein Basiswert, dann werte b applikativ aus (rekursiv)

Beispiel: applikative Auswertung

Wo im Ausdruck **applikativ** auswerten?

$\texttt{quadrat}\ (\texttt{quadrat}\ ((\texttt{quadrat}\ 2) + (\texttt{quadrat}\ 3)))$

$\texttt{quadrat}\ (\texttt{quadrat}\ ((\texttt{quadrat}\ 2) + (\texttt{quadrat}\ 3)))$

Beispiel für applikative Reihenfolge

```
quadrat x = x*x
```

2 Auswertungen (applikative Reihenfolge)

$\texttt{quadrat}(4+5) \ \rightarrow\ \texttt{quadrat}\ 9 \ \rightarrow\ 9*9 \ \rightarrow\ 81$

$\texttt{quadrat}(\texttt{quadrat}(2+3)) \ \rightarrow\ \texttt{quadrat}(\texttt{quadrat}\ 5) \ \rightarrow$
$\texttt{quadrat}\ 25 \ \rightarrow\ 25*25 \ \rightarrow\ 625$

fakultaet: applikative Reihenfolge

```
main = fakultaet 4
fakultaet x = if x <= 1 then 1
                else x*(fakultaet (x-1))
```

applikativ

```
fakultaet 4
if  4 <= 1  then  1  else  4*(fakultaet  (4-1))
if False   then 1  else  4*(fakultaet  (4-1))
4*(fakultaet (4-1))
4*(fakultaet  3)
4*(if 3  <=  1 then  1 else 3*(fakultaet  (3-1)))
```


fakultaet: applikative Reihenfolge (2)

```
4*(if False then  1 else 3*(fakultaet (3-1)))
4*(3*(fakultaet (3-1)))
4*(3* (fakultaet  2))
4*(3*(if 2  <=  1  then 1 else 2*(fakultaet (2-1))))
4*(3*(if False    then  1 else  2*(fakultaet(2-1))))
4*(3*(2*(fakultaet (2-1))))
4*(3*(2*(fakultaet  1)))
4*(3*(2*(if 1  <=  1  then  1 else  1*(fakultaet(1-1)))))
4*(3*(2*(if True     then 1  else  1*(fakultaet(1-1)))))
4*(3*(2*1))
4*(3*2)
(4*6)
24
```

18 Auswertungsschritte

Beispiel: applikative Reihenfolge der Auswertung

```
main = const 5 (fakultaet 4)
fakultaet x = if x <= 1 then 1
                else x*(fakultaet (x-1))
const x y = x
```

main $\xrightarrow{1}$ const 5 (fakultaet 4) $\xrightarrow{18}$ const 5 24 $\xrightarrow{1}$ 5

Anzahl der Reduktionen: 20

Normale Reihenfolge der Auswertung

- Definitionseinsetzung vor Argumentauswertung
- wird in Haskell benutzt

Beschreibung: normale Reihenfolge

werte t in **normaler** Reihenfolge aus. Fälle:

- t ist Basiswert. fertig.
- $t \equiv s_1\ t_1$,
  Wenn s_1 kein Funktionsname und keine Anwendung.
  Dann normale R. auf s_1
- $t \equiv f\ t_1 \dots t_n$ und f keine eingebaute Funktion,
  Wenn $ar(f) \leq n$, dann Definitionseinsetzung auf $f\ t_1 \dots t_{ar(f)}$.
  Wenn $ar(f) > n$: keine Reduktion.
- $t \equiv f\ t_1 \dots t_n$ und f ist eingebaute Funktion
  Wenn $ar(f) \leq n$ und Argumente von f keine Basiswerte,
  dann normale R. auf $ar(f)$ Argumente von links nach rechts.
  Wenn $ar(f) \leq n$, und $ar(f)$ Argumente von f sind Basiswerte,
  dann eingebaute Funktion aufrufen.
  Wenn $ar(f) > n$: keine Reduktion.
- $t \equiv$ `if` b `then` e_1 `else` e_2.
  Wenn b Basiswert, dann if-Reduktion
  Wenn b kein Basiswert, dann normale R. auf b

Beispiel: normale Auswertung

Wo im Ausdruck **normal** auswerten?

$(\texttt{quadrat}\ (2+3)) * (\texttt{quadrat}\ (2+3))$

$(\texttt{quadrat}\ (2+3)) * (\texttt{quadrat}\ (2+3))$

Beispiel für normale Reihenfolge

```
quadrat x = x*x
```

2 Auswertungen

$\texttt{quadrat}(4+5) \to (4+5)*(4+5) \to 9*(4+5) \to 9*9 \to 81$

$\texttt{quadrat}\ (\texttt{quadrat}(2+3)) \to$
$(\texttt{quadrat}(2+3)) * (\texttt{quadrat}(2+3)) \to$
$((2+3)*(2+3)) * (\texttt{quadrat}(2+3)) \to$
$(5*(2+3)) * (\texttt{quadrat}(2+3)) \to$
$(5*5) * (\texttt{quadrat}(2+3)) \to$
$25 * (\texttt{quadrat}(2+3)) \to$
$25 * ((2+3)*(2+3)) \to$
$25 * (5*(2+3)) \to$
$25 * (5*5) \to$
$25 * 25 \to$
625

Beispiel: Auswertung

```
main = fakultaet 4
fakultaet x = if x <= 1 then 1
              else x*(fakultaet (x-1))
```

Auswertung (in normaler Reihenfolge:)

fakultaet 4
if 4 <= 1 then 1 else 4*(fakultaet(4-1))
if False then 1 else 4*(fakultaet(4-1))
4*(fakultaet(4-1))

Beispiel: normale Reihenfolge der Auswertung (2)

```
4*(fakultaet(4-1)))
4*(if (4-1)<= 1 then 1 else (4-1)*(fakultaet((4-1)-1)))
4*(if 3 <= 1 then 1 else (4-1)*(fakultaet((4-1)-1)))
4*(if False then 1 else (4-1)*(fakultaet((4-1)-1)))
4*((4-1)*(fakultaet((4-1)-1)))
4*(3*(fakultaet((4-1)-1)))
4*(3*(if ((4-1)-1) <= 1 then 1 else ((4-1)-1)*(fakultaet(((4-1)-1)-1))))
4*(3*(if(3-1)<= 1 then 1 else ((4-1)-1)*(fakultaet(((4-1)-1)-1))))
4*(3*(if 2 <= 1 then 1 else ((4-1)-1)*(fakultaet(((4-1)-1)-1))))
4*(3*(if False then 1 else ((4-1)-1)*(fakultaet(((4-1)-1)-1))))
4*(3*(((4-1)-1)*(fakultaet(((4-1)-1)-1))))
4*(3*((3-1)*(fakultaet(((4-1)-1)-1))))
4*(3*(2*(fakultaet(((4-1)-1)-1))))
4*(3*(2*(if (((4-1)-1)-1) <= 1 then 1 ...)))
```


Beispiel: normale Reihenfolge der Auswertung (4)

```
4*(3*(2*(if (((4-1)-1)-1) <= 1 then 1 ...)))
4*(3*(2*(if ((3-1)-1) <= 1 then 1 ...)))
4*(3*(2*(if 2-1 <= 1 then 1 ...)))
4*(3*(2*(if 1 <= 1 then 1 ...)))
4*(3*(2*(if True then 1 ...)))
4*(3*(2*1))
4*(3*2)
4*6
24
```

Das sind 24 Auswertungsschritte

Beispiel: normale Reihenfolge der Auswertung

```
main = const 5 (fakultaet 4)
akultaet x = if x <= 1 then 1
             else x*(fakultaet (x-1))
const x y = x
```

$\texttt{main} \xrightarrow{1} \texttt{const 5 (fakultaet 4)} \xrightarrow{1} 5$

Anzahl der Reduktionen: 2

(20 bei applikativer Reihenfolge)

Beispiel für verschiedene Reduktionsstrategien

```
quadrat x = x*x
```

3 Auswertungen für `quadrat (4+5)` :

1. $\texttt{quadrat}(4+5) \to (4+5)*(4+5) \to 9*(4+5) \to 9*9 \to 81$
   normale Reihenfolge der Auswertung
2. $\texttt{quadrat}(4+5) \to (4+5)*(4+5) \to (4+5)*9 \to 9*9 \to 81$
   Irgendeine Auswertung
3. $\texttt{quadrat}(4+5) \to (\texttt{quadrat}\ 9) \to 9*9 \to 81$
   applikative Reihenfolge der Auswertung

Verzögerte Reihenfolge der Auswertung

Definition *verzögerte Reihenfolge der Auswertung* (lazy reduction):

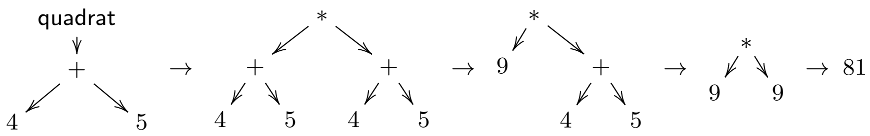
- wie normale Reihenfolge
- aber: gerichteter Graph statt Syntax-Baum
- Vermeidung von unnötiger Doppelauswertung durch gemeinsame Unterausdrücke (Sharing)
- Die gemeinsamen (d.h. shared) Unterausdrücke sind durch die Funktionsrümpfe festgelegt.

Beispiele

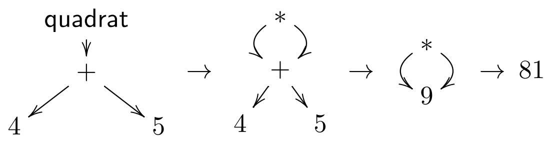
- 4 Reduktionen: (normale Reihenfolge)
  $\texttt{quadrat}(4+5) \to (4+5)*(4+5) \to 9*(4+5) \to 9*9 \to 81$
- 3 Reduktionen (verzögerte Reihenfolge)
  $\texttt{quadrat}(4+5) \to (4+5)^{(1)} * (4+5)^{(1)} \to 9*9 \to 81$

Beispiel in gerichteter-Graph-Darstellung

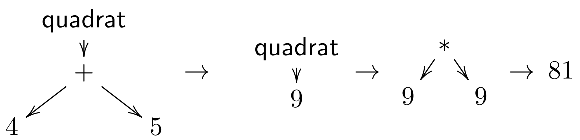
Normale Reihenfolge:



Verzögerte Reihenfolge:



Applikative Reihenfolge:



Verzögerte Auswertung: Beispiel

```
fakultaet x  = if  x  <= 1 then 1
                  else  x*(fakultaet (x-1))
```

Rot: die Stelle, die reduziert wird
Grün: die Stelle, die identisch mit der roten ist

```
fakultaet 4
if 4 <= 1  then  1 else 4*(fakultaet(4-1))
if False  then  1 else  4*(fakultaet(4-1))
4*(fakultaet(4-1))
4*(if (4-1) <=  1  then  1 else (4-1)*(fakultaet((4-1)-1)))
4*(if  3   <=  1   then 1  else  3   *(fakultaet( 3 -1)))
```

Beispiel ...

```
4*(if 3 <= 1 then 1 else 3*(fakultaet(3-1)))
4* (if False then 1 else 3*(fakultaet(3-1)))
4*(3*(fakultaet(3-1)))
4*(3*(if (3-1)<= 1 then 1 else (3-1)*(fakultaet((3-1)-1))))
4*(3*(if 2 <= 1 then 1 else 2*(fakultaet(2-1))))
4*(3*(if False then 1 else 2*(fakultaet(2-1))))
4*(3*(2*(fakultaet(2-1))))
4*(3*(2*(if (2-1)<= 1 then 1 else (2-1)*(fakultaet ((2-1)-1)))))
4*(3*(2*(if 1 <= 1 then 1 else 1*(fakultaet(1-1)))))
4*(3*(2*(if True then 1 else 1*(fakultaet(1-1)))))
4*(3*(2*1))
4*(3*2)
4*6
24
```
(18 Auswertungsschritte)

Übersicht Reduktionsanzahl zu Beispielen

	verzögerte R.	applikative R.	normale R.
`(fakultaet 4)`	18	18	24
`main`	2	20	2

```
main = const 5 (fakultaet 4)
```


Optimale Anzahl der Reduktionen

Es gilt: verzögerte Reduktion
hat optimale Anzahl von Reduktionsschritten. !

- Es gilt immer:
  $\boxed{\#\text{ verzögerte R.} \leq \#\text{ normale R.}}$

  $\boxed{\#\text{ verzögerte R.} \leq \#\text{ applikative R.}}$

- Im allgemeinen gilt:
  # applikative R. und # normale R. sind unvergleichbar

- Wenn alle Reduktionsschritte für das Ergebnis benötigt werden:
  $\boxed{\#\text{ verzögerte R.} \leq \#\text{ applikative R.} \leq \#\text{ normale R.}}$

Reduktions-Strategien und Korrektheit

In Programmiersprachen (z.B. Haskell) mit verzögerter Reihenfolge der Auswertung:

Reduktionen zur Compile-Zeit sind korrekte Programmtransformationen d.h. die Semantik bleibt erhalten

Das ist i.a. falsch in Programmiersprachen, die die applikative Reihenfolge verwenden.

Auswertungsprozesse

Rekursive Auswertungsprozesse

Wir betrachten jetzt Auswertungsprozesse, die durch eine einzige rekursive Funktion erzeugt werden

Wir betrachten bei der Analyse von Auswertungsprozessen nur die **applikative Reihenfolge**

Beispiel: Auswertung der rekursiven Fakultätsfunktion

$$\begin{array}{lcll} 0! & := & 1 & \\ n! & := & n * (n-1)! & \text{wenn } n > 1 \end{array}$$

```
fakultaet x  = if x <= 1 then 1
               else x*(fakultaet (x-1))
```


Auswertungsprozess, linear rekursiv

bei applikativer Reihenfolge der Auswertung
(Nicht jeder Zwischenzustand ist angegeben)

```
(fakultaet 6)
(6 * (fakultaet (6-1)))
(6 * (5 * (fakultaet (5-1))))
(6 * (5 * (4 * (fakultaet (4-1)))))
(6 * (5 * (4 * (3 * (fakultaet (3-1))))))
(6 * (5 * (4 * (3 * (2 * (fakultaet (2-1)))))))
(6 * (5 * (4 * (3 * (2 * 1)))))
(6 * (5 * (4 * (3 * 2))))
(6 * (5 * (4 * 6)))
(6 * (5 * 24))
(6 * 120)
720
```


Auswertungsprozess, linear rekursiv

(`fakultaet 6`) Auswertungsprozess ist linear rekursiv

Charakterististisch:

- nur eine rekursive Funktionsanwendung in jedem Ausdruck der Reduktionsfolge
- Zwischenausdrücke sind nicht beschränkt.

Alternative Berechnungen der Fakultät

Iteriere folgende Regel:

Produkt	$\Rightarrow$	Produkt $*$ Zähler
Zähler	$\Rightarrow$	Zähler $+$ 1

```
fakt_iter  produkt  zaehler  max =
   if zaehler > max
   then produkt
   else fakt_iter (zaehler*produkt) (zaehler + 1) max

fakultaet_lin n =  fakt_iter 1 1 n
```


Endrekursion

Eine Endrekursion ist eine lineare Rekursion.
Zusätzlich muss gelten:

- In jedem Rekursionsaufruf: der rekursive Aufruf berechnet direkt den Rückgabewert
  ohne Nachverarbeitung

Auswertungsprozess, endrekursiv

Auswertung von `(fakultaet_lin 5)` bei verzögerter Reihenfolge der Auswertung

```
(fakultaet_lin 5)
(fakt_iter 1 1 5)
(fakt_iter (1*1) (1+1) 5)
(fakt_iter (2*(1*1)) (2+1) 5)
(fakt_iter (3*(2*(1*1))) (3+1) 5)
(fakt_iter (4*(3*(2*(1*1)))) (4+1) 5)
(fakt_iter (5*(4*(3*(2*(1*1))))) (5+1) 5)
(5*(4*(3*(2*(1*1)))))
120
```


Iterativer Auswertungsprozess

Iterativer Auswertungsprozess bei applikativer Auswertung:

```
(fakultaet_lin 6)
(fakt_iter 1 1 6)
(fakt_iter 1 2 6)
(fakt_iter 2 3 6)
(fakt_iter 6 4 6)
(fakt_iter 24 5 6)
(fakt_iter 120 6 6)
(fakt_iter 720 7 6)
720
```

Iterativer Prozess:

Charakteristisch: ist eine Endrekursion
Argumente sind Basiswerte
(bzw. Größe des Gesamtausdrucks bleibt beschränkt.)
optimierte Rückgabe des Wertes

Optimierung der Endrekursion

imperative Programmiersprachen	Endrekursion i.a. nicht optimiert. d.h. Wert wird durch alle Stufen der Rekursion zurückgegeben
Haskell	Endrekursion ist optimiert am Ende wird der Wert unmittelbar zurückgegeben.

Deshalb braucht man in imperativen Programmiersprachen:
Iterationskonstrukte
`for ...do`, `while`, `repeat ...until`.
Diese entsprechen iterativen Auswertungsprozessen

Iteration in Haskell

Bei verzögerter Auswertung:

> Eine rekursive Funktion f ist $\boxed{\text{iterativ}}$, wenn
> $f\ t_1 \ldots t_n$ für Basiswerte t_i
> bei applikativer Reihenfolge der Auswertung
> einen iterativen Prozess ergibt.

Viele (nicht alle) linear rekursive Funktionen
kann man zu iterativen umprogrammieren.
Zum Beispiel: `fakultaet` zu `fakultaet_lin`

Baumrekursion

Beispiel Berechnung der Fibonacci-Zahlen

1, 1, 2, 3, 5, 8, 13, 21, . . .

$$Fib(n) := \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{sonst} \end{cases}$$

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

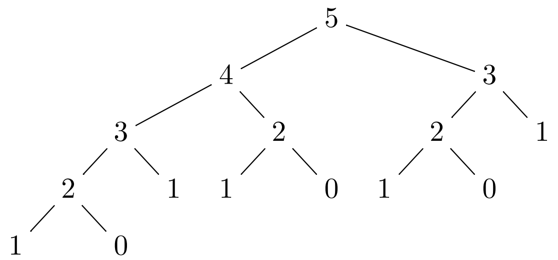

Auswertungsprozess zu fib

in applikativer Reihenfolge:
Der Auswertungs-Prozess ergibt folgende Zwischen-Ausdrücke:

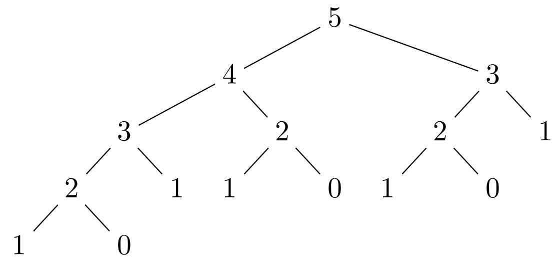
```
fib 5
fib 4 + fib 3
(fib 3 + fib 2)  + fib 3
((fib 2 + fib 1) + fib 2)  + fib 3
(((fib 1 + fib 0) + fib 1) + fib 2)  + fib 3
(((1+0) + fib 1) + fib 2)  + fib 3
((1 + fib 1) + fib 2)  + fib 3
((1+1) + fib 2)  + fib 3
(2 + fib 2)  + fib 3
(2 + (fib 1 + fib 0))  + fib 3
.......
```

Auswertungsprozess zu fib

```
fib 5
fib 4 + fib 3
(fib 3 + fib 2)  + fib 3
((fib 2 + fib 1) + fib 2)  + fib 3
(((fib 1 + fib 0) + fib 1) + fib 2)  + fib 3
(((1+0) + fib 1) + fib 2)  + fib 3
((1 + fib 1) + fib 2)  + fib 3
```



Auswertungsprozess zu fib: Baum der Aufrufe



Das ist Baumrekursion
Charakteristisch: Ausdrücke in der Reduktionsfolge

- können unbegrenzt wachsen
- können mehrere rekursive Aufrufe enthalten
- Aber: nicht geschachtelt
  (d.h. die Argumente eines rekursiven Aufrufs
  enthalten keine rekursiven Aufrufe)

Geschachtelte Rekursion

Ist der allgemeine Fall
wird normalerweise selten benötigt, da i.a. nicht effizient berechenbar. Beispiel: Die Ackermannfunktion

```
-----  Ackermanns Funktion  ----
ack 0 y                     = 1
ack 1 0                     = 2
ack x 0 | x >= 2            = x+2
ack x y | x > 0 && y > 0    = ack (ack (x-1) y) (y-1)
```

benutzt folgende Programmier-Technik in Haskell:
Reihenfolge: der Fallabarbeitung
von oben nach unten wird probiert, welche Definitionsgleichung passt:
1) Argumente anpassen
2) Bedingung rechts vom | prüfen

Optimierte Ackermannfunktion

```
-----  Ackermanns Funktion "optimiert" ----
ackopt 0 y = 1
ackopt 1 0 = 2
ackopt x 0 = x+2
ackopt x 1 = 2*x
ackopt x 2 = 2^x
ackopt x y | x > 0 && y > 0 =
         ackopt (ackopt (x-1) y) (y-1)
```

Optimierte Ackermannfunktion (2)

```
*Main> logI10 (ackopt 5 3) ↵
19728.301029995662
```

19728 ist die Anzahl der Dezimalstellen
des Ergebnisses `(ackopt 5 3)` $= 2^{65536} = 2^{2^{2^{2^2}}}$

- sehr schnell wachsende Funktion
- man kann nachweisen: man braucht geschachtelte Baum-Rekursion um `ack` zu berechnen
- hat Anwendung in der Komplexitätstheorie

Tabelle der Rekursionsprozesse:

„impliziert“

geschachtelt baumrekursiv	mehrere rekursive Unterausdrücke auch in den Argumenten der rekursiven Unterausdrücke erlaubt
baumrekursiv	mehrere rekursive Unterausdrücke erlaubt, aber Argumente der rekursiven Unterausdrücke ohne weitere Rekursion
linear rekursiv	maximal ein rekursiver Unterausdruck
endrekursiv	linear rekursiv und Gesamtresultat ist Wert des rekursiven Unterausdrucks
iterativ	endrekursiv und Argumente des rekursiven Unterausdrucks sind Basiswerte

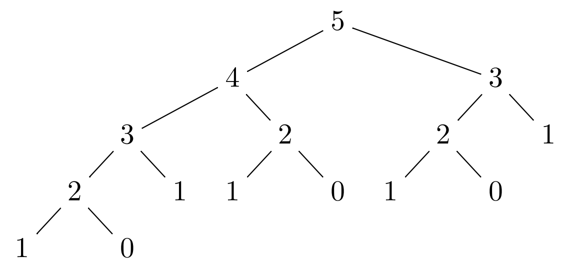
Optimierung und Analyse

Wir analysieren beispielhaft folgende Funktionen
auf Laufzeit und Speicherbedarf:

- `fakultaet`
- `fib`
- `ggt`

Optimierung: Iteration statt Rekursion

Beispiel
Berechnung von `(fib 5)`



`fib 3` wird 2 mal berechnet
`fib 2` wird 3 mal berechnet
`fib 1` wird 5 mal berechnet

Optimierung: Iteration statt Rekursion

Genauer und Allgemeiner:
Bei Berechnung von fib n für $n \geq 2$
wird fib(1) jeweils (fib n)-mal berechnet

fib $n \approx \frac{\Phi^n}{\sqrt{5}}$ wobei $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.6180$ (goldener Schnitt)

Fazit:

$\#$ Reduktionen für `fib`(n) ist exponentiell
d.h. die Laufzeit von `fib` ist exponentiell in n

Iterative Version von Fib

Beobachtung:	zur Berechnung von fib(n) benötigt man nur die Werte fib(i) für $1 \leq i \leq n$.
Idee:	Berechnung einer Wertetabelle für `fib`.
Verbesserte Variante:	aus fib_{n-1} und fib_{n-2} berechne fib_n Ohne Doppelberechnung von Funktionswerten $(\text{fib}_{n-1}, \text{fib}_{n-2}) \to (\underbrace{\text{fib}_{n-1} + \text{fib}_{n-2}}_{= \text{fib}_n}, \text{fib}_{n-1})$
Rechenvorschrift:	$(a, b) \to (a + b, a)$

Iterative Version von `fib`: Funktionen

```
fib_lin n             =  (fib_iter  1 0 n)
fib_iter a b zaehler =  if zaehler <=  0
                         then b
                         else fib_iter (a + b)
                                       a
                                       (zaehler - 1)
```


Prozess für `(fib_lin 5)`

```
(fib_lin 5)
(fib_iter 1 0 5)
(fib_iter 1 1 4)
(fib_iter 2 1 3)
(fib_iter 3 2 2)
(fib_iter 5 3 1)
5
```

Analyse der fib-Optimierung

Für (`fib_lin n`) gilt:

- ist operational äquivalent zu `fib`
- benötigt linear viele Reduktionen abhängig von n
- Größe der Ausdrücke ist beschränkt
- Platzbedarf ist konstant (d.h. unabhängig) von n.
  (unter Vernachlässigung der Darstellungsgröße der Zahlen)

erzeugt iterativen Auswertungsprozess (applikative R.)

Analyse von Programmen

Abschätzung und Messung des Ressourcenbedarfs
von Haskell-Programmen,
bei verzögerter Auswertung:

Zeit:	Anzahl der Transformationsschritte
Platz: (Gesamt-Speicher):	Maximale Größe der Ausdrücke
Arbeitsspeicher:	Maximale Größe der Ausdrücke (ohne die Eingabe)

arithmetische und Boolesche Operationen $= 1$
Transformationsschritt.

Analyse von Programmen (2)

Angabe des Ressourcenbedarf eines Algorithmus
in Abhängigkeit von der Größe der Eingabe.

Notation für Algorithmus `alg` bei Eingabe der Größe n:

$red_{\mathsf{alg}}(n)$ maximale Anzahl der Reduktionen bei verzögerter Auswertung für alle Eingaben der Größe n.

$Platz_{\mathsf{alg}}(n)$ Platzbedarf: maximale Größe der Ausdrücke (des gerichteten Graphen) bei verzögerter Auswertung für alle Eingaben der Größe n.
Die Eingaben nicht mitzählen

Beispiel: Fibonacci-Funktion fib

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

$red_{\mathtt{fib}}(n) \approx 1.6^n$ (einfach exponentiell)

Bezugsgröße: Zahl n

Achtung: Die Komplexitätstheorie verwendet
i.a. die Speicher-Größe der Eingabe

Beispiel `fakultaet n`

`fakultaet n` benötigt $5 * (n-1) + 3$ Reduktionsschritte.
bei verzögerter Reihenfolge der Auswertung

(Kann man durch Beispielauswertungen raten)
Z.B. `fakultaet 4` benötigt 18 Reduktionsschritte.

Nachweis mit vollständiger Induktion ....

Beispiel `fakultaet n`: Nachweis

Ind. Schritt:
Nachzuweisen ist: `fakultaet (n-1)` benötigt $5 * (n-1) + 3$ für $n > 2$.

```
fakultaet (n-1)
if (n-1) <= 1 then ...
if  n1 <= 1 then ...    -- n1 ist Basiswert > 1
if False then ...
n1*fakultaet (n1-1)   -- Wert n2 als Ind hypothese zaehlt nicht
n1*n2                 --  Produkt-Berechnung zaehlt noch dazu
n3
```

Das sind $5 + 5 * (n1 - 1) + 3 = 5 * (n-1) + 3$ Reduktionsschritte

Komplexitäten von Algorithmen

Beachte: breite Streuung des Ressourcenbedarfs ist möglich für die Menge aller Eingaben einer bestimmten Größe.

Komplexitäten von Platz und Zeit:

Ressourcenbedarf verschiedene Varianten:

im *schlimmsten Fall* (worst-case)
im *besten Fall* (best-case)
Minimum von # Reduktionen
bzw. Minimum der Größe der Ausdrücke.
im *Mittel* (average case)
Welche Verteilung?

Abschätzung von Komplexitäten mit O(.)

Als Beispiel hatten wir das Ergebnis:

`fakultaet (n-1)` benötigt $5 * (n-1) + 3$ Reduktionen für $n > 2$.

Abschätzung von $5 * (n-1) + 3$ nach oben (als Funktion von n):

$$5 * (n-1) + 3 \quad \leq \quad 6 \cdot n$$

Geschrieben als $\lambda n.(5 * (n-1) + 3) \quad \in \quad O(n)$.
! Multiplikative und additive Konstanten werden wegabstrahiert.

Einige Komplexitäten

$O(1)$	konstant
$O(log(n))$	logarithmisch
$O(n)$	linear
$O(n * log(n))$	fastlinear (oder auch n-log-n)
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(n^k)$	polynomiell
$O(2^n)$	exponentiell

n ist die Größe der Eingabe (i.a Bit-Anzahl)

Analyse zum größten gemeinsamen Teiler

ggT(a, b) (Euklids Algorithmus)

Teile a durch b ergibt Rest r,
wenn $r = 0$, dann ggT$(a, b) := b$
wenn $r \neq 0$, dann berechne ggT(b, r).

Beispiel ggT$(30, 12)$ = ggT$(12, 6) = 6$

```
ggt a b =     if  b == 0
              then a
              else ggt b (rem a b)
```


Komplexität des Euklidischen Algorithmus

SATZ (Lamé, 1845):
Wenn der Euklidische ggt-Algorithmus k Schritte benötigt,
dann ist die kleinere Zahl der Eingabe $\geq fib(k)$.

Platz- und Zeitbedarf von ggt: $O(log(n))$

Begründung:
Wenn n die kleinere Zahl ist und der Algorithmus k Schritte benötigt,
dann ist $n \geq fib(k) \approx 1.6180^k$
Also ist Anzahl der Rechenschritte $k = O(log(n))$

Vergleichstabelle (Zusammenfassung)

Aufruf	Zeitaufwand - Abschätzung	
Arithmetische Operationen als $O(1)$ angenommen		
fakultaet n	$O(n)$	
fib n	$O(1,62^n)$	
fib_lin n	$O(n)$	
ggt m n	$O(\log(\mathtt{max}(m, n)))$	
$m + n$	$O(1)$	m, n :: Int
$m * n$	$O(1)$	m, n :: Int
quadratsumme m n	$O(1)$	m, n :: Int
Arithmetische Operationen auf großen Zahlen		
$m + n$	$O(\log(m + n))$	m, n :: Integer
$m * n$	$O(\log(m + n))$	m, n :: Integer
quadratsumme m n	$O(\log(m + n))$	m, n :: Integer