

Eine ganz andere Domäne: Verarbeitung arithmetischer Ausdrücke

- Nehmen wir an, wir wollen arithmetische Ausdrücke in „Maschinencode“ compilieren, also zum Beispiel:

```
"2+3*5"  ↦ "LIT 2; LIT 3; LIT 5; MUL; ADD; "  
"2*3+5"  ↦ "LIT 2; LIT 3; MUL; LIT 5; ADD; "
```

- Zunächst müssen wir erstmal die Struktur von (gültigen) Ausdrücken beschreiben.
- Zum Beispiel mittels einer formalen Grammatik:

```
expr    ::= term + expr | term  
term    ::= factor * term | factor  
factor  ::= nat | (expr)
```

- ... und jetzt könnten wir (in einer „konventionellen“ Programmiersprache) einen Algorithmus zum Parsen entsprechend einer/dieser Grammatik entwickeln/umsetzen.

Eine ganz andere Domäne: Verarbeitung arithmetischer Ausdrücke

- Attraktiver wäre, möglichst direkt die vorhandene Beschreibung

```
expr    ::= term + expr | term
term    ::= factor * term | factor
factor  ::= nat | (expr)
```

zu verwenden, und diese selbst als „Programm“ zu lesen.

- Immerhin nah dran:

```
expr = ( ADD <$> term <* char '+' <*> expr ) ||| term
term  = ( MUL <$> factor <* char '*' <*> term ) ||| factor
factor = ( LIT <$> nat ) ||| ( char '(' *> expr <* char ')' )
```

- Schonmal ausprobieren:

```
> parse expr "2*3+5"
ADD (MUL (LIT 2) (LIT 3)) (LIT 5)
```

Eine ganz andere Domäne: Verarbeitung arithmetischer Ausdrücke

- Um die eigentlich gewünschte Ausgabe zu erhalten:

```
data Expr = LIT Int | ADD Expr Expr | MUL Expr Expr

instance Show Expr where
  show (LIT n)      = "LIT " ++ show n ++ ";"
  show (ADD e1 e2) = show e1 ++ show e2 ++ "ADD;"
  show (MUL e1 e2) = show e1 ++ show e2 ++ "MUL;"
```

- Dann tatsächlich:

```
> parse expr "2*3+5"
LIT 2; LIT 3; MUL; LIT 5; ADD;
```

- Alternativ zum Beispiel auch direkte Berechnung des Ergebnisses möglich:

```
eval (LIT n)      = n
eval (ADD e1 e2) = eval e1 + eval e2
eval (MUL e1 e2) = eval e1 * eval e2
```

Eine ganz andere Domäne: Verarbeitung arithmetischer Ausdrücke

- Alternativ zum Beispiel auch direkte Berechnung des Ergebnisses möglich:

```
eval (LIT n)      = n
eval (ADD e1 e2)  = eval e1 + eval e2
eval (MUL e1 e2)  = eval e1 * eval e2
```

- Dann zum Beispiel:

```
> eval (parse expr "2*3+5")
11
```

- Oder sogar Auswertung direkt beim Parsen:

```
expr  = ( (+) <$> term <* char '+' <*> expr ) ||| term
term   = ( (*) <$> factor <* char '*' <*> term ) ||| factor
factor = nat ||| ( char '(' *> expr <* char ')' )
```

- Dann nämlich:

```
> parse expr "2*3+5"
11
```

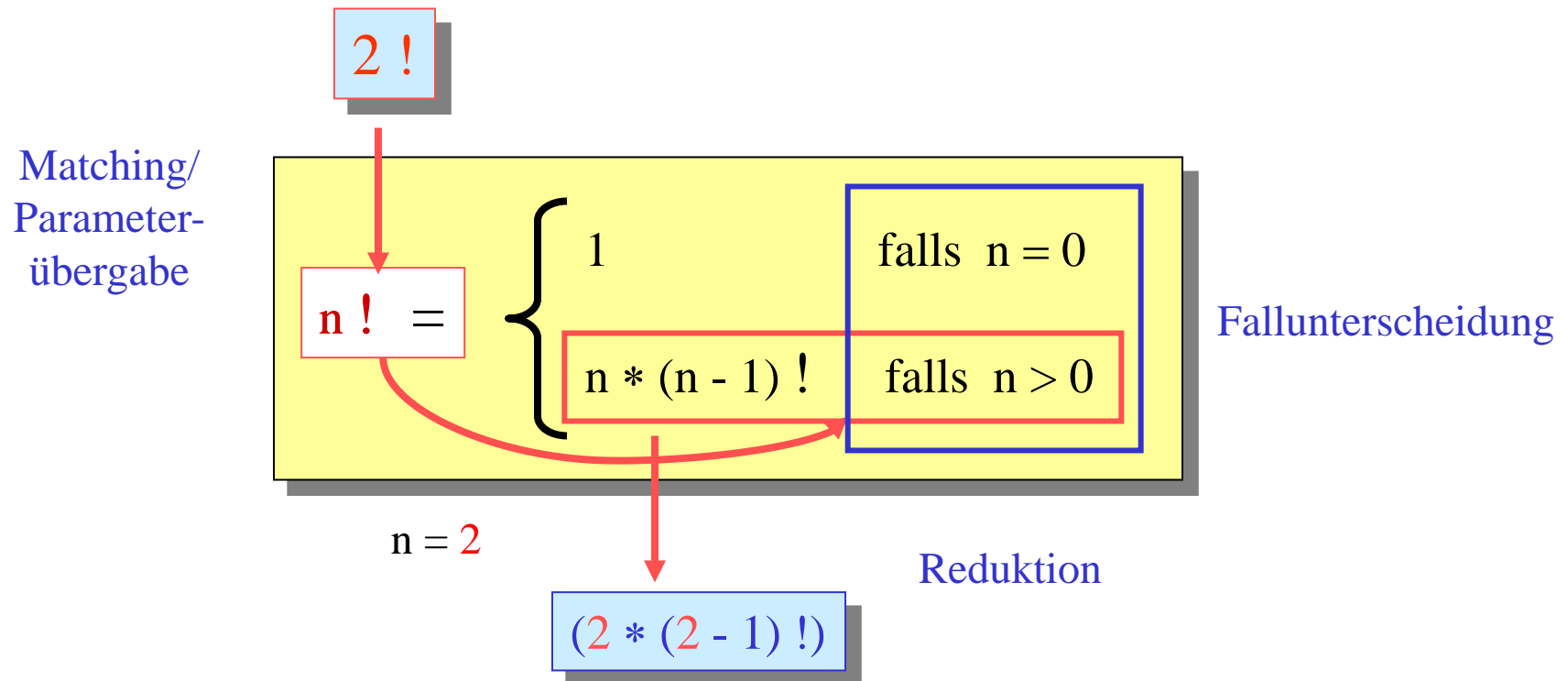
Deskriptive Programmierung

Haskell-Grundlagen/Syntax

Prinzip des funktionalen Programmierens

Spezifikationen: Funktionsdefinitionen

Operationalisierung: Auswertung von Ausdrücken (syntaktische Reduktion)



Prinzip des funktionalen Programmierens

„Let the symbols do the work.“

Leibniz/Dijkstra

Spezifikation („Programm“) \equiv
Funktionsdefinition(en)

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n - 1)! & \text{falls } n > 0 \end{cases}$$

vordefinierte Operatoren

2!
 $\Rightarrow (2 * (2 - 1) !)$
 $\Rightarrow (2 * 1!)$
 $\Rightarrow (2 * (1 * (1 - 1) !))$
 $\Rightarrow (2 * (1 * 0 !))$
 $\Rightarrow (2 * (1 * 1))$
 $\Rightarrow (2 * 1)$
 \Rightarrow **2**

Eingabe: auszuwertender Term/Ausdruck

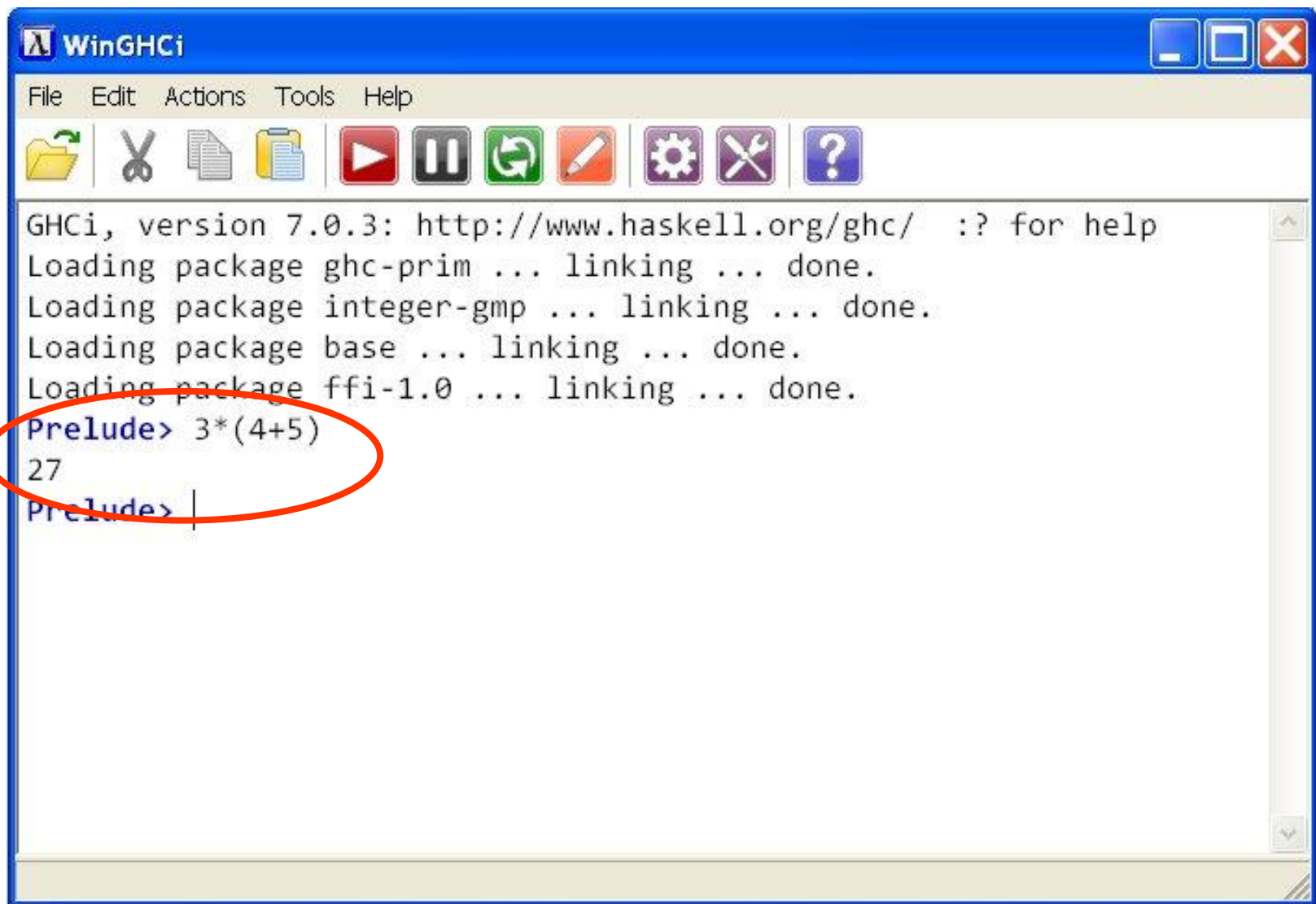


(wiederholte) Funktionsanwendung



Ausgabe: resultierender Funktionswert

GHCI als „Taschenrechner“



The screenshot shows a window titled "WinGHCi" with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations, execution, and settings. The main text area displays the following text:

```
GHCi, version 7.0.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> 3*(4+5)
27
Prelude> |
```

A red oval highlights the input `3*(4+5)` and the output `27`.

- **Int, Integer:**
 - ganze Zahlen (-12, 0, 42, ...)
 - Operatoren: +, -, *, ^
 - Funktionen: div, mod, min, max, ...
 - Vergleiche: ==, /=, <, <=, >, >=
- **Float, Double:**
 - Gleitkommazahlen (-3.7, pi, ...)
 - Operatoren: +, -, *, /
 - Funktionen: sqrt, log, sin, min, max, ...
 - Vergleiche: ...
- **Bool:**
 - Boolesche Werte (True, False)
 - Operatoren: &&, ||
 - Funktionen: not; Vergleiche: ...
- **Char:**
 - einzelne Zeichen ('a', 'b', '\n', ...)
 - Funktionen: succ, pred; Vergleiche: ...

Auswertung einfacher Ausdrücke

```
> 5+7  
12
```

```
> div 17 3  
5
```

nicht: `div(17,3)`

dafür:

```
> 17 `div` 3  
5
```

```
> pi/1.5  
2.0943951023932
```

```
> min (sqrt 4.5) (1.5^3)  
2.12132034355964
```

nicht: `min(sqrt(4.5),1.5^3)`

```
> 'a' <= 'c'  
True
```

```
> if 12<3 || 17.5/=sqrt 5 then 17-3 else 6  
14
```

nie ohne else-Zweig!

- Listen:

- [Int] für [] oder [-12, 0, 42]
- [Bool] für [] oder [False, True, False]
- [[Int]] für [[3, 4], [], [6, -2]]
- ...
- Operatoren: :, ++, !!
- Funktionen: head, tail, last, null, ...

```
> 3 : [-12, 0, 42]  
[3, -12, 0, 42]
```

```
> [1.5, 3.7] ++ [4.5, 2.3]  
[1.5, 3.7, 4.5, 2.3]
```

```
> [False, True, False] !! 1  
True
```

- Zeichenketten:

- String = [Char]
- spezielle Notation: "" für [] und "abcd" für ['a', 'b', 'c', 'd']

- Tupel:

- (Int, Int) für (3, 5) und (0, -4)
- (Int, String, Bool) für (3, "abc", False)
- ((Int, Int), Bool, [Int]) für ((0, -4), True, [1, 2, 3])
- [(Bool, Int)] für [(False, 3), (True, -4), (True, 42)]
- ...
- Funktionen: fst und snd auf Paaren

```
> (3-4, snd (head [('a', 17), ('c', 3)]))  
(-1, 17)
```

Deklaration von Werten

- in Datei:

```
x = 7
y = 2 * x
z = (mod y (x + 2), tail [1 .. y])

a = b - c
b = fst z
c = head (snd z)

d = (a, e)
e = [fst d, f]
f = head e
```

← All dies sind
Deklarationen,
keine wert-
ändernden
Zuweisungen!

x = x + 1

ergibt keinen Sinn!

- nach dem Laden:

> z
(5, [2,3,4,5,6,7,8,9,10,11,12,13,14])

> a
3

> d
(3, [3, 3])

$x, y :: \text{Int}$

$x = 7$

$y = 2 * x$

$z :: (\text{Int}, [\text{Int}])$

$z = (\text{mod } y (x + 2), \text{tail } [1 .. y])$

$a, b, c :: \text{Int}$

$a = b - c$

$b = \text{fst } z$

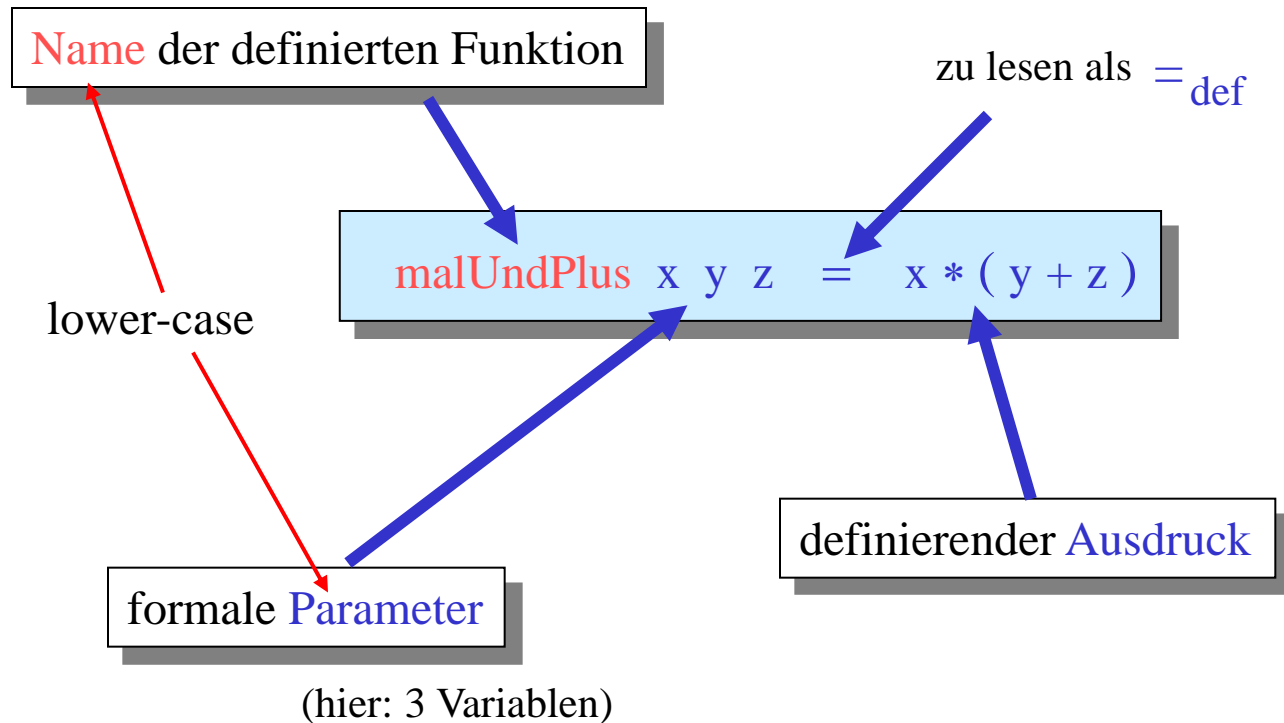
$c = \text{head } (\text{snd } z)$

$d :: (\text{Int}, [\text{Int}])$

$d = (a, e)$

...

Prinzipieller Aufbau einer (sehr einfachen) Funktionsdefinition:



Deklaration von Funktionen (mit Typangaben)

zur Erinnerung: „if-then“ in Haskell immer mit explizitem „else“!

```
min3 :: Int → Int → Int → Int
min3 x y z = if x < y then (if x < z then x else z)
              else (if y < z then y else z)
```

```
> min3 5 4 6
4
```

```
min3' :: (Int, Int, Int) → Int
min3' (x, y, z) = if x < y then (if x < z then x else z)
                  else (if y < z then y else z)
```

```
> min3' (5, 4, 6)
4
```

```
min3" :: Int → Int → Int → Int
min3" x y z = min (min x y) z
```

```
> min3" 5 4 6
4
```

```
isEven :: Int → Bool
isEven n = (n `mod` 2) == 0
```

```
> isEven 12
True
```

Gleichheitstest!

Beispiele Syntax für Funktionsapplikationen

Mathe-üblich	Haskell-üblich
$f(x)$	$f\ x$
$f(x,y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x,g(y))$	$f\ x\ (g\ y)$
$f(x) + g(y)$	$f\ x + g\ y$
$f(a+b)$	$f\ (a + b)$
$f(a) + b$	$f\ a + b$

Deskriptive Programmierung

Haskell-Auswertungssemantik

Bedarfs-Auswerten von (rekursiven) Funktionen

```
fac :: Int → Int  
fac n = if n == 0 then 1 else n * fac (n - 1)
```

```
> fac 5  
120
```

```
sumsquare :: Int → Int  
sumsquare i = if i == 0 then 0 else i * i + sumsquare (i - 1)
```

```
> sumsquare 4  
30
```

Berechnung durch schrittweises Auswerten:

```
> sumsquare 2  
= if 2 == 0 then 0 else 2 * 2 + sumsquare (2 - 1)  
= 2 * 2 + sumsquare (2 - 1)  
= 4 + sumsquare (2 - 1)  
= 4 + if (2 - 1) == 0 then 0 else ...  
= 4 + (1 * 1 + sumsquare (1 - 1))  
= 4 + (1 + sumsquare (1 - 1))  
= 4 + (1 + if (1 - 1) == 0 then 0 else ...)  
= 4 + (1 + 0)  
= 5
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

$a = 3$
 $d = (a, e)$
 $e = [\text{fst } d, f]$
 $f = \text{head } e$



$> d$
 $= (a, e)$

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

$a = 3$

$d = (a, e)$

$e = [\text{fst } d, f]$

$f = \text{head } e$



$> d$

$= (a, e)$

$= (3, e)$

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

a = 3
d = (a, e)
e = [fst d, f]
f = head e



> d
= (a, e)
= (3, e)
= (3, [fst d, f])

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3  
d = (a, e)  
e = [fst d, f]  
f = head e
```



```
> d  
= (a, e)  
= (3, e)  
= (3, [fst d, f])  
= (3, [fst (3, [fst d, f]), f])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3  
d = (a, e)  
e = [fst d, f]  
f = head e
```



```
> d  
= (a, e)  
= (3, e)  
= (3, [fst d, f])  
= (3, [fst (3, [fst d, f]), f])  
= (3, [3, f])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3  
d = (a, e)  
e = [fst d, f]  
f = head e
```



```
> d  
= (a, e)  
= (3, e)  
= (3, [fst d, f])  
= (3, [fst (3, [fst d, f]), f])  
= (3, [3, f])  
= (3, [3, head e])
```


Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3  
d = (a, e)  
e = [fst d, f]  
f = head e
```



```
> d  
= (a, e)  
= (3, e)  
= (3, [fst d, f])  
= (3, [fst (3, [fst d, f]), f])  
= (3, [3, f])  
= (3, [3, head e])  
= (3, [3, head [3, head e]])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3  
d = (a, e)  
e = [fst d, f]  
f = head e
```



```
> d  
= (a, e)  
= (3, e)  
= (3, [fst d, f])  
= (3, [fst (3, [fst d, f]), f])  
= (3, [3, f])  
= (3, [3, head e])  
= (3, [3, head [3, head e]])  
= (3, [3, 3])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3  
d = (a, e)  
e = [fst d, f]  
f = head e
```

