Übung Informatik I

Übungsleiter: Tom Gelhausen

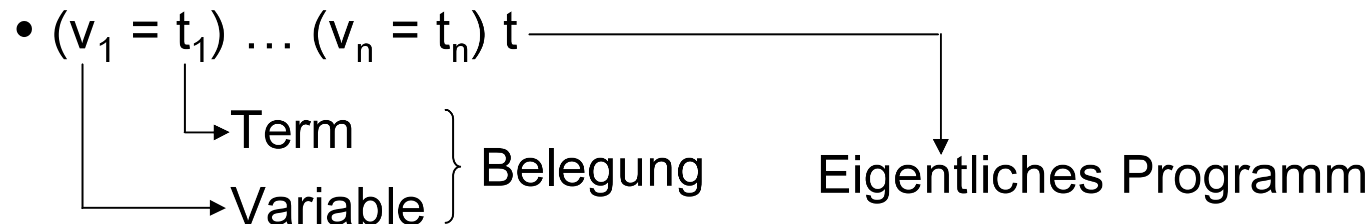
(mit freundlicher Unterstützung durch d&d)

1. λ - Kalkül
2. Haskell
3. Fehlerarten

1. Lambda-Kalkül

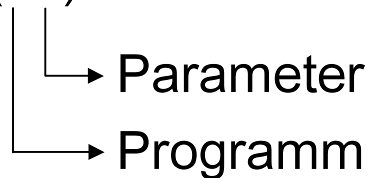
- Das Lambda-Kalkül ist ein Modell zur formalen Beschreibung von funktionalen Programmiersprachen.

- **Programm im λ-Kalkül:**
  - $(v_1 = t_1) \ldots (v_n = t_n)\ t$



- **Terme im λ-Kalkül:**
  - Variable: x
  - Lambda-Abstraktion: $f = (\lambda x.t)$
  - Im strengen λ-Kalkül gibt es keine Ziffern, +, -, …
  - Oft werden sie aber dazugefügt, um die Schreibweise zu erleichtern.

- **Ausführung eines Programms:**
  - (f a) → wird nach den α-, β-, η-Konversionsregeln umgeformt.



- **Konversionsregeln:**
  - α: λv.t ⇒ λw.t[w/v]
  - β: (λv.t) a ⇒ t[a/v]
  - η: (λv.(t v)) ⇒ t ,wenn v nicht in t vorkommt

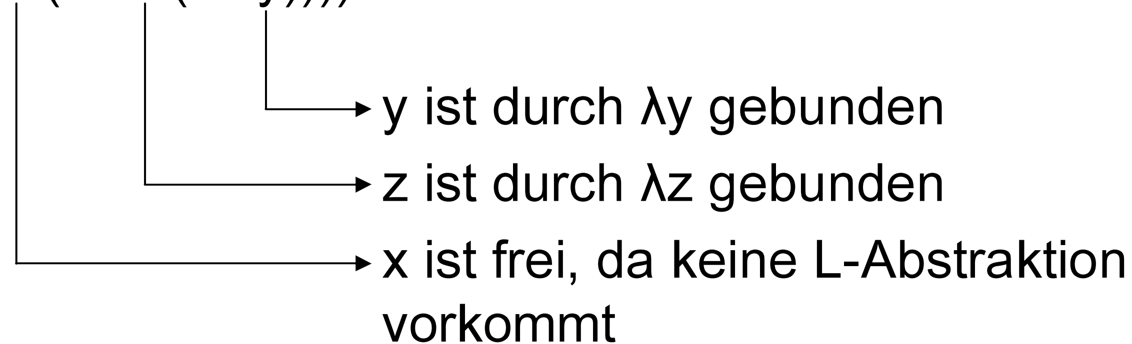
Freie Variablen bleiben auch nach dem Ersetzen frei und gebundende Variablen bleiben gebunden!

1. Gebundene und freie Variablen

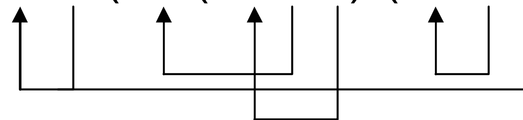
- **Beispiele:**
  - (λy. (x (λz.z (λz.y))))
    - y ist durch λy gebunden
    - z ist durch λz gebunden
    - x ist frei, da keine L-Abstraktion vorkommt
  - (λa.a (λb.(λc.b c) (λd.d a)) d)

1. Gebundene und freie Variablen

- **Beispiele:**
  - (λy. (x (λz.z (λz.y))))



  - (λa.a (λb.(λc.b c) (λd.d a)) d)



1. Auswertung von Lambda-Kalkülen

- (λx. (+ x x)) 2
  = (+ x x) [2/x] (β-Konversion)
  = (+ 2 2)
  = 4 (+ ausgewertet)
- Klammern durchnummerieren !!!
- (λx. (* 2 x)) ((λx.(λy.+ x y)) 2 3)
  = (λx. (* 2 x))[f/x] ((λx.(λy.+ x y)) 2 3) (α-Konversion)
  = (λf. (* 2 f)) ((λx.(λy.+ x y)) 2 3)
  = (* 2 f) [((λx.(λy.+ x y)) 2 3) / f] (β-Konversion)
  = (* 2 ((λx.(λy.+ x y)) 2 3))
  = (* 2 ((λy.+ x y) [2/x] 3)) (β-Konversion)
  = (* 2 ((λy.+ 2 y) 3))
  = (* 2 ((+ 2 y) [3/y])) (β-Konversion)
  = (* 2 (+ 2 3))
  = (* 2 5) (+ ausgewertet)
  = 10 (* ausgewertet)

1. Church-Boolean I

- **Church-Boolean:**
  ist eine Funktion, die eines ihrer Argumente zurückgibt
  True: erstes, **False**: zweites Argument

  true =(λx. (λy. (x)))

  false =(λx. (λy. (y)))

- **Beispiel**:
  - true angewendet auf λx.(λa.(λb.(x a b))) :

    λx.(λa.(λb.(x a b))) (λx.(λy.(x))) = $^{\alpha}$

    λx.(λa.(λb.(x a b))) (λz.(λy.(z))) = $^{\beta}$

    λa.(λb.((λz.(λy.(z))) a b)) = $^{\beta}$

    λa.(λb.((λy.(a)) b)) = $^{\beta}$

    λa.(λb.(a))

1. Church-Boolean II

- **Aufgabe:**

  Schreiben Sie die Funktion XOR ("entweder oder") in λ-Kalkül.

Wertetabelle:

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

true = (λx. (λy. (x)))

false = (λx. (λy. (y)))

not = (λx.(x false true))

XOR = ?

1. Church-Boolean II

- **Aufgabe:**
  Schreiben Sie die Funktion XOR ("entweder oder") in λ-Kalkül.

Wertetabelle:

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

true = (λx. (λy. (x)))

false = (λx. (λy. (y)))

not = (λx.(x false true))

XOR = (λx. (λy. (x (y false true) y)))

2. Warum funktionale Programmierung?

- **Konzepte**
  - Eingebaute Datentypen: Listen, Bäume
  - Rekursion als Ansatz zur Problemlösung
  - Currying
  - Funktionen höherer Ordnung

- **Didaktik**
  - Einstieg aus der Mathematik
  - Herausforderung auch für Könner imperativer Sprachen

- **Präsentation**
  - Kurze, knappe Programme
  - Fokus auf Problemlösung, nicht Artefakten der Sprache
  - Beispiel:
    - Schreibe eine Klasse, die eine Methode mit beliebiger Parameterzahl entgegennimmt, auf ein einzelnes Argument anwendet und die so gebundene Methode zurückliefert.

2. Vorlesung und Übung

- **In der Übung**
  - Einstieg in die Programmierung
  - Konzepte aus praktischer Sicht
  - Probleme in der Sprache lösen
  - Geläufigkeit erlangen
- **In der Vorlesung (Kap.5,6 im Buch)**
  - Einstieg in die Theorie
  - Grundlagen funktionaler Programmierung: λ-Kalkül
  - Probleme der Sprache betrachten
- **Im Folgenden also:**
  - Interessante Eigenschaften des Artefakts Programmiersprache
  - Wie gehe ich damit um?

2. Haskell

- **Was ist Haskell?**
  - typisierte funktionale Sprache
  - ohne freie Variablen
  - ohne wesentliche Sprachkonstrukte (for, while)
  - basiert auf Rekursion, fauler Auswertung und Termersetzung

Beispiele zur Termersetzung:

a. Programmtext: `hallo_Studis = "Hallo Tom"`
   Aufruf: > hallo_Studies [enter]
   Ausgabe: "Hallo Tom"

b. Programmtext: `inkrement x = x + 1`
   Aufruf: > inkrement 3 [enter]
   Intern: x+1 [3/x] = 3+1 = 4
   Ausgabe: 4

2. Datentypen und Operationen in Haskell

- **Datentypen:**
  - Integer, Int 1
  - Char 'a'
  - String "arg"
  - Bool True/False
  - Float, Double 1.2

- **Operationen auf Zahlen:**

  +, -, *, /, abs, div, mod...
- **Operationen auf Wahrheitswerten:**

  ==, /=, <, <=, >, >=, &&, ||, not...

- Haskell besitzt nur wenige eingebaute Funktionen, aber viele häufig benutze finden sich in der `prelude.hs` (z.B. even, gcd (=ggT)...)

2. Funktionen und Strukturen I

- **Signatur:**
  Die Signatur gibt die Typen der Eingabe und Ausgabe an.

```
f :: InputType1 -> InputType2 -> ... -> OutputType
```

Beispiel:

```
inkrement :: Int -> Int
inkrement x = x + 1
```

Hinweis: So funktioniert das Programm nur noch mit Int-Werten. Inkrement 1.2 würde einen Fehler verursachen.

2. Funktionen und Strukturen II

- **Funktionsrumpf:**
  Der Rumpf beinhaltet die eigentliche Funktion und kann noch Bedingungen enthalten.

- ```
  f x1 ... xn                 = Ausdruck
  ```

- ```
  f x1 ... xn  | Bedingung1 = Ausdruck1
               | Bedingung2 = Ausdruck2
               | otherwise  = Ausdruck3
  ```

- ```
  f x1 ... xn                 = if Bedingung then Ausdruck1
                                else Ausdruck2
  ```

- Bedingungen stellen Bool-Werte dar (z.B. `x == 3`),
  Ausdrücke bestehen aus Konstanten (z.B. `3`) oder weiteren Funktionen (z.B. `inkrement 2`)
  ```

2. Funktionen und Strukturen III

- Welchen Sinn haben folgende Sprachkonstrukte ?
  - ```
    f a = let  y   = a * 6
               g x = (x + y)/ y
          in g 3 + g 5
    ```
  - ```
    f x y  | y > z   = ...
           | y == z  = ...
           | y < z   = ...
      where z = x * x
    ```

- Beide Konstrukte dienen dazu, Funktionen und Variablen lokal zu definieren (also um die Arbeit zu erleichtern!)

- `where` : bindet über alles, auch über Guards (`| Bedingung =` …)
- `let` : bindet nur über den Term hinter `in` und kann auch verschachtelt benutzt werden.

2. Funktionen und Strukturen IV

- Schreibweisen für binäre Funktionen:

```
Infix:   2 `div` 4
Präfix:  div 2 4
```

- Layout ist wichtig!

```
 | if    | Bedingung  | then
 |       | Ausdruck1  |
 | else  |            |
 |       | Ausdruck2  |
 |       |            |
```

- Sollte ein Programm mal nicht funktionieren, kontrollieren Sie die Groß- und Kleinschreibung! Denn:
  - Funktionsnamen immer klein
  - Variablennamen immer klein
  - Typkonstruktoren („Datentypen“ wie `Int`, `Bool`,...) groß
  - Datenkonstruktoren (z.B. `True`, `False`,...) immer groß

2. Beispiele I

- **Größter gemeinsamer Teiler:**
  Schreiben Sie ein Haskell-Programm, dass den ggT von zwei Zahlen berechnet.

- **Lagerverwaltung:**
  Der Weihnachtsmann kalkuliert seine Kosten für das diesjährige Fest. Da auf seinem Schlitten nur begrenzt Platz für Geschenke ist, will er natürlich ein optimales Verhältnis von Wert der Geschenke und benötigtem Platzbedarf erreichen, um möglichst viele Kinder glücklich zu machen.
  Er beauftragt seine Helfer-Elfen ihm zunächst ein paar Funktionen in XMaskell (bei Karlsruher Studenten auch als Haskell bekannt) zu schreiben, die ihm zu jedem seiner drei Geschenksorten, die in seiner Geschenke-Datenbank nur mit 1 bis 3 durchnummeriert sind, den Preis, das Gewicht und den Namen liefern.

2. Beispiele I

- Größter gemeinsamer Teiler:

```
ggt :: Int -> Int -> Int
ggt p q       | mod p q == 0      = q
              | otherwise         = ggt q (mod p q)
```

- Lagerverwaltung:

```
preis :: Int -> Float
preis x       | x == 1 = 0.99
              | x == 2 = 17.99
              | x == 3 = 49.95

bezeichnung :: Int -> String
bezeichnung 1          = "Kaugummi"
bezeichnung 2          = "CD: Peter Mafay Greatest Hits"
bezeichnung 3          = "PC Spiel:  Return to Tetris"

volumen :: String -> Float
volumen "Kaugummi"                        = 1.0
volumen "CD: Peter Mafay Greatest Hits" = 6.5
volumen "PC Spiel:  Return to Tetris"   = 13.25
```

2. Listen I

- Listen sind (endliche) Folgen von Elementen.
- **Notation:**
  Liste = '[ ]' | Element ':' Liste
- **Beispiele**:
  - `liste1 = (1 : 2 : 3 : 4 : []) = [1, 2, 3, 4]`
  - `liste2 = (0 : liste1) = [0, 1, 2, 3, 4]`
  - `keine_liste = 1`
- **Typ einer Liste:**
  `[ElementTyp]`, z.B. `[Int]`, `[Char]`,...

Beachte: Eine Liste kann nur aus Elementen **eines** Typs bestehen!

2. Listen II

- **Beispiel** zur Verwendung von Listen:

```
erster (x:xs) = x
erstesumme (x:y:xs) = x + y
```

- **Einige Funktionen auf Listen:**

`length xs`: Liefert die Länge der Liste xs
`head xs`: Liefert das erste Element der Liste
`tail xs`: Liefert die Liste ohne das erste Element
`xs ++ ys`: Hängt zwei Listen aneinander

- **Beispiele**:

Sei `xs = [1, 2, 3, 4]` im folgenden:

```
length xs  = 4
head xs    = 1
tail xs    = [2, 3, 4]
xs ++ [5]  = [1, 2, 3, 4, 5]
```

- **Rekursion:**
  Man spricht von Rekursion, wenn sich eine Funktion selbst aufruft.

- **Idee:** wie vollständige Induktion
  - Löse Basisfall der Größe 1
  - Lösung für Fall Größe k gegeben, löse damit Fall Größe k+1

- **Beispiel**: Fakultätsberechnung

```
fac :: Int -> Int
fac 1 = 1
fac n = n * fac (n-1)
```

- **Beispiel**: Listenelemente aufsummieren

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs)= x + (sum xs)
```

- **Lagerverwaltung:**
  Nachdem die Elfen nun alles zur Zufriedenheit des Weihnachtsmannes erledigt haben, gibt er ihnen nun die Aufgabe geeignete Funktionen zu entwerfen, die das Gesamtvolumen und den Gesamtpreis seiner Fracht berechnen, damit man anschließend mit einer weiteren kleinen Funktion das Verhältnis von Wert der Geschenke und benötigtem Platzbedarf berechnen kann.

```
preis_pro_volumen xs   = (gesamtkosten xs)/(gesamtvolumen xs)

gesamtkosten xs        = kosten xs 1
gesamtvolumen xs       = menge xs 1

kosten [] _            = 0
kosten (x:xs) pos      = x*(preis pos)+(kosten xs (pos+1))

menge [] _             = 0
menge (x:xs) pos       = x*(volumen  (bezeichnung pos)) +
                         (menge xs (pos+1))
```

- **Ausführung:**

```
preis_pro_volumen [1,2,3] = 3.47572
```

3. verschiedene Fehlerarten I

Welche Fehlertypen treten in folgendem Beispiel auf?

isEmpty (Pop (Push (Pop (Push 1 CreateStack))(Top (Push CrateStack))))

3. verschiedene Fehlerarten I

Welche Fehlertypen treten in folgendem Beispiel auf?

isEmpty(Pop(Push(Pop(Push 1 CreateStack))(Top(Push CrateStack □))))

- top, pop wird kleingeschrieben
- Rechtschreibfehler
- Parameter vertauscht
- □ falsche Parameteranzahl

Wir haben also:

1. Tippfehler
2. Parametertypfehler
3. Parameteranzahlfehler

3. verschiedene Fehlerarten II

Was für eine Art von Fehler liegt hier vor?

```
fun x  5 = 35 + 27x
fun 3  x = 14 * 49x
fun 3  5 = 7
```

Programmaufruf: `fun 3 5`

Ausgabe? 116, nicht 7!!!

1. Tippfehler
2. Parametertypfehler
3. Parameteranzahlfehler
4. Logikfehler

4. Ende

"Morgen, Kinder, wird's was geben!"

(evtl. eine kleine Musterlösung zu Blatt 7…) ☺