# Programmiersprachen – Konzepte und Realisationen

**Th. Letschert**

**TH Mittelhessen Gießen**

***University of Applied Sciences***

## Namens-Bindung und Lambda-Kalkül

- Namensbindung
- Gültigkeitsbereiche / freie und gebundene Namen
- Lambda-Kalkül

# Literale / Bezeichner

- **Literal** ***literal***

  **Lexikalische Einheit / Token mit – sprachweit – fixer Bedeutung**

  **z.B.: „12“ bedeutet in jedem Java-Progamm den gleichen int-Wert**

- **Bezeichner, Name,** ***identifier***

  **Lexikalische Einheit / Token mit einer fixen Bedeutung die auf einen bestimmten Programmabschnitt (*Scope*) beschränkt ist.**

  **Z.B.:**

```
public static void sort() {
  for (int i=0; i<z; i++) {
    for (int j=i+1; j<z; j++) {
      if (a[i] > a[j]) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
      }
    }
  }
}
```

***i, j, t haben eine lokale begrenzte Bedeutung mit unterschiedlichem Gültigkeitsbereich (Scope).***

# Namensbindung

- **Name (Bezeichner, *Identifier*)**

  **Bezeichner, dessen Bedeutung für ein Programm oder einen Programmabschnitt vom Programmierer festgelegt wird.**

  **Name und Bezeichner werden oft synonym verwendet. Bezeichner ist ein syntaktische Konstrukt. Beispiel:**

  ***java.util.List*  Ein Name, der aus drei Bezeichnern zusammengesetzt ist**

- **Namensbindung**

  **Abbildung: Name ~> Bedeutung des Namens**

- **Statische Namensbindung**

  **Die Bedeutung eines Namens ergibt sich komplett aus dem Text des Programms, der Name hat bei jeder Ausführung des Programms die gleiche Bedeutung.**

  **Der Name erhält seine Bedeutung durch die „textuell nächstliegende Definition des Namens“**

- **Dynamische Namensbindung**

  **Die Bedeutung eines Namens ergibt sich erst zur Laufzeit Programms, der Name kann bei der Ausführung des Programms wechselnde Bedeutungen haben.**

  **Der Name erhält seine Bedeutung durch die „zuletzt ausgeführt Definition des Namens“**

**Diskussion***: Welche Art der Namensbindung hat sich aus welchem Grund durchgesetzt? Was denken Sie: zu Recht?*

# Namensbindung

**Diskussion**: *Welche Namensbindung verwendet Java und wie kann man das an der Ausgabe dieser Klasse erkennen.*

```java
public class Binding {

  interface G { void call(); }

  static G f() {
    final int x = 42;
    return new G() {
      public void call() {
        System.out.println("x = " + x);
      }
    };
  }

  public static void main(String[] args) {
    G g = f();
    final int x = 47;
    g.call();
  }
}
```

Java

# Namensbindung

**Diskussion**: *Welche Namensbindung verwendet Scala und wie kann man das an der Ausgabe dieser Klasse erkennen.*

```scala
object NameBindung extends App {

  val x = 0;

  def f() {
    println(x);
  }

  def g() {
      val x = 1;
      f();
  }

  g();
}
```

Scala

# Namensbindung

**Diskussion**: *Welche Namensbindung verwendet JavaScript und wie kann man das an der Ausgabe dieser Beispiel erkennen.*

*Klick auf* button2
*Klick auf* button2
*Klick auf* button2

→

```
<body>
  <script type="text/javascript">
     var buttonId = new Array ("button0", "button1", "button2")
     for(var i in buttonId) {
        document.write("<button id=\""+buttonId[i]+"\" >");
        document.write("Knopf " + i);
        document.write("</button>");
     }
  </script>
</body>
  <script type="text/javascript">
     for (var i in buttonId) {
        document.getElementById(buttonId[i]).onclick =
            function () { alert("Klick auf "+ buttonId[i]); };
     }
 </script>
```

JavaScript

*Klick auf* button0
*Klick auf* button1
*Klick auf* button2

→

```
<body>
<script type="text/javascript">
     var buttonId = new Array ("button0", "button1", "button2")
     for(var i in buttonId) {
          document.write("<button id=\""+buttonId[i]+"\" >");
          document.write("Knopf " + i);
          document.write("</button>");
     }
</script>
</body>
  <script type="text/javascript">
       for (var i in buttonId) {
          document.getElementById(buttonId[i]).onclick
               = function(text){
                    return function(){
                        alert("Klick auf " + text);
                    };
                } (buttonId[i]);
       }
</script>
```

JavaScript

# Variable

- **Einstufige Bindung: Name ~> Wert**

  **Namen bedeuten direkt einen Wert**

- **Zweistufige Bindung: Name ~> Variable ~> Wert**

  **Namen verweisen indirekt auf Werte, ihre direkte Bedeutung ist ein (virtuelles?) Zwischenkonstrukt**

- **Variable: *t* ~> Wert**

  **Übliches Zwischenkonstrukt beim Weg vom Namen zum Wert: Zeitabhängiger Wert**


*x bedeutet / bezeichnet 5*


*x bedeutet / bezeichnet eine „Variable“, deren Wert ist (gerade) 5*

**Diskussion**: *Programmieranfängern muss (mehr oder weniger mühsam) beigebracht werden, dass es Variablen gibt, dass ein Name wie x sich nicht direkt, sondern nur indirekt auf einen Wert bezieht. Warum ist die Variable trotzdem eine nützliche Fiktion die von vielen Programmiersprachen vertreten wird?*

# Definition und Gültigkeitsbereich

– **Definition**

Namen erhalten bei ihrer Definition eine Bedeutung (Wert, Variable, ...)

– **Deklaration**

Die Begriffe „Definition“ und „Deklaration“ werden oft synonym verwendet, meist zu Recht.

Manche Sprachen unterscheiden jedoch Definitionen und Deklarationen.

Eine Deklaration ist oft eine „reduzierte Form“ der Definition, die aus technischen Gründen benötigt wird. Beispiel C/C++:

`int x;` Definition der Variablen x: *x bedeutet / bezeichnet eine bestimmte Variable*

`extern int x;` Deklaration der Variablen x: *x bedeutet / bezeichnet eine Variable die an anderer Stelle definiert wird*

– **Gültigkeitsbereich**

Der Gültigkeitsbereich einer Definition ist der textuelle Bereich in dem die von der Definition erzeugte Bindung gültig ist.

Natürlich ist das Konzept des Gültigkeitsbereichs nur bei Sprachen mit statischer Namensbindung sinnvoll.

**Diskussion:**
- *Warum macht es keinen Sinn bei Sprachen mit dynamischer Namensbindung von einem Güligkeits-bereich zu sprechen?*
- *Wann / Warum ist es vor allem bei Sprachen mit statischer Namensbindung sinnvoll, ein 2-stufiges Bindungs-Konzept zu verwenden?*

# Freie und gebundene Namen

- **Gebundener Name / Gebundene Variable**

  **Ein Name ist *in einem textuellen Kontext gebunden*, wenn er in diesem Kontext definiert wird. Dieser textuelle Bereich ist der Gültigkeitsbereich der Definition.**

  **Die Bedeutung des Namens wird also in dem betrachteten Kontext festgelegt.**

- **Freier Name / Freie Variable**

  **Ein Name ist *in einem textuellen Kontext frei*, wenn es in diesem Kontext keine Definition des Namens gibt.**

  **Der Name und seine Bedeutung kommt also von außen in dem betrachteten Kontext.**

```
new G() {
   public void call() {
      System.out.println("x = " + x);
   }
};
```

***x ist hier frei***

```
static G f() {
  final int x = 42;
  return new G() {
     public void call() {
        System.out.println("x = " + x);
     }
  };
}
```

***x ist hier gebunden***

# Konzepte von Gültigkeitsbereichen

- **Funktionen / Methoden**

  **sind Gültigkeitsbereiche ihrer Parameter- und lokalen Definition.**

- **Blöcke**

  **In vielen Sprachen können Codeblöcke ( { … } ) als Güligkeitsbereiche verwendet werden**

- **Abdeckungsregel**

  **Praktisch alle Sprachen verwenden *Abdeckungsregeln* nach denen lokale Definitionen globale Definitionen abdecken.**

# Konzepte von Gültigkeitsbereichen

**Diskussion**:

- *Die Abdeckungsregeln einer Sprache definieren eine Suchreihenfolge nach der die richtige Bindung eines Namens bestimmt wird.*

  *„Erst Innen, dann Aussen“ ist eine korrekte Suchstrategie bei verschachtelten Definitionen.*

  *Abgeleiteter Typ / Basistyp definiert eine Suchreihenfolge bei Ableitungs-Hierarchien.*

  *Beide Regeln treten gelegentlich in Konflikt: Wie wird dieser in Scala gelöst? Erläutern Sie an diesem Beispiel.*

- *Übersetzen Sie das Beispiel nach Java. Wie sieht es hier aus?*

```
class Base {
  val x = 42;
}

class Outer {
  val x = 47;

  class Inner extends Base {
    def m() {
      println(x)
    }
  }
}

object NameBindung extends App {
  val outer = new Outer;
  val inner = new outer.Inner;

  inner.m();
}
```

Scala

# Der (untypisierte) λ-Kalkül

- Winzige dynamisch typisierte Programmiersprache
- von karger Eleganz
- eingeführt von A. Church (1936)
- als Formalismus zur Definition von Berechenbarkeit
- Es gibt nur Funktionen und Funktionsaufrufe => alles ist eine Funktion
- Erstaunlich / faszinierend: Der λ-Kalkül ist Turing-vollständig!

  Jede Berechenbare Funktion kann mit Hilfe von Funktionsdefinition und Anwendung definiert werden.
- Merkmale
  - Definition von Funktionen (höherer Ordnung)
  - Funktionsanwendung
  - Keine Schleifen, keine (!) Rekursion, keine Konstanten
  - *Namensbindungen / Gültigkeitsbereiche* sind die (fast) einzigen Features der Sprache

## Beispiele

- λ x . x — identische Funktion ( λx . x *entspricht* f *mit* f(x) = x )
- λ x . λ y . x — Paar auf erstes Element projizieren
- λ f . λ x . f (f (x)) — Funktion zweimal anwenden
- (λ x . x) y = y — Funktionsanwendung
- (λ x . λ y . x) a b — Projektion auf paar anwenden
  = ((λ x . λ y . x) a ) b
  = ( λ y . a) b
  = a
- (λ f . λ x . f (f (x))) (λ x . x)) a — Identität zweimal auf a anwenden
  = (λ x . (λ x . x) ((λ x . x) x) a
  = (λ y . (λ x . x) ((λ x . x) y) a — α-Reduktion: Umbenennung von gebundenen Variablen
  = (λ x . x) ((λ x . x) a)
  = (λ x . x) (a)
  = a

## Konkrete Syntax

| | |
|---|---|
| exp ::= x | Variable |
| \| $e_1$ $e_2$ | Funktionsanwendung |
| \| λ x . e | λ-Abstraktion / Funktionsdefinition |
| \| (e) | Klammerung |

Funktionsanwendung ist links-assoziativ:

$e_1\ e_2\ e_3 = (e_1\ e_2)\ e_3$

Aus Gewohnheit verwendet man oft „Aufrufklammern“:

$e_1\ e_2\ e_3 = (e_1\ e_2)\ e_3 = (e_1\ (e_2))\ (e_3)$

Der Punkt bindet soweit wie möglich

$\lambda\ x\ .\ e_1\ (e_2\ e_3) = (\ \lambda\ x\ .\ e_1\ (e_2\ e_3))$

# Konkrete Syntax

## Parser:

```
import scala.util.parsing.combinator.JavaTokenParsers
import scala.util.parsing.input.CharSequenceReader

// abstract syntax tree
sealed abstract class Exp
case class Variable(name: Symbol) extends Exp
case class Abstraction(v: Symbol, body: Exp) extends Exp
case class Application(fun: Exp, arg: Exp) extends Exp

object LambdaParser extends JavaTokenParsers {

  private def variable : Parser[Exp] = ident ^^ { x => Variable(Symbol(x)) }

  private def term : Parser[Exp]
      = variable |
        ("/"~>ident<~".") ~ exp ^^ {case a~b => Abstraction(Symbol(a),b)} |
        "("~>exp<~")"

  private def exp : Parser[Exp]
        = rep1(term) ^^ {
              (lst) => lst.reduceLeft{
                          (fun, arg) => Application(fun, arg)
                       }
            }

  def parse(s: String) : Exp =
    phrase(exp)(new CharSequenceReader(s)) match {
        case Success(t,_)     => t
        case NoSuccess(msg,_) => throw new IllegalArgumentException(
                                    "Parser Error  '" + s + "': " + msg)
  }

}
```

```
object Lambda_Parsing_App extends App {
  println(LambdaParser.parse("(/ x . x) y"))
}
```

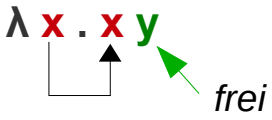
```
Application(Abstraction('x,Variable('x)),Variable('y))
```

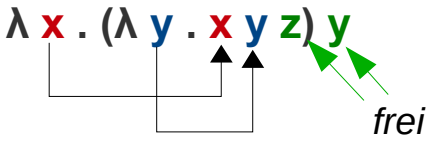
# Freie / gebundene Variablen / Kombinatoren

– **gebundene / freie Variable**

**gebunden**: Variable die durch eine λ-Abstraktion gebunden ist

**frei**: Variable die nicht durch eine λ-Abstraktion gebunden ist




– **offener / geschlossener Term**

**offen**: Term enthält freie Variablen

**geschlossen**: Term enthält keine freien Variablen

# Freie / gebundene Variablen / Kombinatoren

– **Kombinator = geschlossener Term**

**Beispiele:**

| | |
|---|---|
| **I** | **= λ x . x** |
| **K** | **= λ x . λ y . x** |
| **D** | **= λ x . x x** |
| **S** | **= λ f . λ g . λ x . f x (g x)** |
| **Y** | **= λ f . ( λ x . f (x x) ) ( λ x . f (x x) )** |

***Jeder geschlossene Term kann mit ausschließlicher Verwendung dieser Kombinatoren ausgedrückt werden*.**

## Der reine λ-Kalkül

- Im reinen λ-Kalkül muss alles mit Abstraktion und Applikation ausgedrückt werden
- Im reinen λ-Kalkül kann alles mit Abstraktion und Applikation ausgedrückt werden
- Zahlen als „Numerale“

0 ~ λf. λx. x
1 ~ λf. λx. f x
2 ~ λf. λx. f (f x)
3 ~ λf. λx. f (f (f x))
…
succ ~ λn. λf. λx. f (n f x)
succ(2) ~ (λn. λf. λx. f (n f x))  (λf. λx. f (f x))
= λf. λx. f ((λf. λx. f (f x)) f x)
= λf. λx. f (λx. f (f x) x)
= λf. λx. f ( f (f x))
~ 3

## Der reine λ-Kalkül

- Bedingte Ausdrücke / boolesche Werte

  if ~ λc. λi. λe. c i e

  true ~ λx. λy. x

  false ~ λx. λy. y

- Paare, Listen etc. sind codierbar

- Rekursion

  letrec f(x) = t [f,x]

  ~ Y λf (λ x t [f,x])

  = λ f . ( λ x . f (x x) ) ( λ x . f (x x) ) λf (λ x t [f,x])

*Selbst Rekursion kann im reinen Lambda-Kalkül ausgedrückt werden.*

# Der erweiterte λ-Kalkül

Der λ-Kalkül wird oft in Kombination mit Konstanten, Operationen und rekursiven Funktionsdefinitionen verwendet. Formal wird damit keine Erweiterung der Ausdrucksmächtigkeit erreicht. Die Lesbarkeit verbessert sich aber deutlich.

Beispiel

```
letrec
  fak(x) = if (x=0) then 1 else f(x-1)*x
in
  fak(10)
```

*Dies ist auch im reinen Lambda-Kalkül ausdrückbar, das wollen wir aber nicht wirklich sehen. – Geschweige denn hinschreiben.*

Der reine λ-Kalkül eignet sich für theoretische Untersuchungen: Eine minimalistische aber vollständige Grundausstattung macht Beweisführungen einfach. Für eine wirkliche Verwendung ist der Minimalismus völlig ungeeignet – und auch nicht gedacht.

Der erweiterte λ-Kalkül ist eine sehr einfache Programmiersprache an der sich die Konzepte

- der Namensbindung und Gültigkeitsbereiche sowie
- Der Funktions-Definitionen und -Aufrufe

klar aufzeigen lassen. Es gibt ja sonst keine weiteren Sprachfeatures.

## Reduktionen

*Die Auswertung von Termen im λ-Kalkül wird über Reduktionen definiert. Reduktionen transformieren einen Term zu einer nicht weiter reduzierbaren Normalform.*

### Berechnung

Reduktion auf Normalform

### Normalform

Ein Term ist in Normalform, wenn er nicht weiter β-reduzierbar ist

### Reduktionen

- **Beta-Reduktion**
  $(\lambda x . M)\ a \rightsquigarrow_\beta M[x\backslash a]$
  **Funktionsanwendung**
  x wird in M durch a ersetzt
- **Alpha-Reduktion**
  $\lambda x. M \rightsquigarrow_\alpha \lambda y. M[x\backslash y]$
  **Umbenennung gebundener Variablen**
  x wird durch y in M ersetzt
- **Eta-Reduktion**
  $\lambda x. M\ x \rightsquigarrow_\eta M$
  **Elimination überflüssiger Abstraktionen**
  (x nicht frei in M, kann durch vorherige α-Reduktion erreicht werden)

## Auswertungsstrategien

***Die Auswertung von Termen durch Reduktion ist nicht eindeutig. Oft kann mehr als eine Reduktion ausgeführt werden.***

- **Redex**

  **ein reduzierbarer Teilterm wird Redex genannt**

- **Auswertungstrategien**
  - **Normalreihenfolge (Normal Order)**

    **der am weitesten links stehende äußere Redex wird zuerst reduziert**
  - **Applikative Reihenfolge (strikte Evaluation)**

    **Jeder Parameter wird vor dem Funktionsaufruf (β-Reduktion) reduziert: der am weitesten links stehende innerste Redex wird zuerst reduziert**

***Der Lambda-Kalkül schreibt keine Auswertungs- (Reduktions-) Strategie vor.***
***– Eine auf dem Lambda-Kalkül basierende Programmiersprache muss das schon tun.***

***Es gibt Terme ohne Normalform!***

***Es gibt Terme bei denen manche Reduktionen zur Normalform führen, andere aber nicht!***

## Auswertungstrategien

- **call-by-value**

  **Applikative Reihenfolge mit der Beschränkung: Keine Reduktion in einer Applikation wenn nicht das Argument vorher vollständig reduziert ist!**

- **call-by-name**

  **Normale Reihenfolge  mit der Beschränkung: Keine Reduktion in einer Abstraktion!**

- **call-by-need (lazy evaluation)**

  **Wie call-by-name, aber ein Term wird nicht mehrfach ausgewertet!**

***Programmiersprachen haben üblicherweise eine fixe Auswertungsstrategie: Parameter werden vor oder nach der Übergabe an eine Funktion ausgewertet.***

***Nur wenige erlauben dem Programmierer die Wahl.***

## Erstes Curch-Rosser-Theorem (Confluenz der Reduktion)

***Wenn ein Term M sich zu zwei Termen $M_1$ und $M_2$ reduzieren lässt,***
***dann gibt es einen Term L zu dem sich sowohl $M_1$ als auch $M_2$ reduzieren lassen.***

**Konsequenz:** **Die Normalform eines Terms ist eindeutig**

## Zweites Curch-Rosser-Theorem (Primat der Normal-Order Reduktion)

***Wenn ein Term M mit eine Normalform hat (die mit irgendeiner Strategie erreicht werden kann),***
***dann führt auch eine Reduktion mit der Normalreihenfolge zu dieser Normalform.***

**Konsequenz:** **Die Normalform eines Terms kann immer (nur !) mit einer Reduktion in Normalreihenfolge erreicht werden.**

*Church-Rosser-Theorem und Programmiersprachen:*

- *Normal-Order-Reduction ~ Call-by-need.*
- *Wenn ein Funktionsaufruf mit einer anderen Strategie zu keinem Ergebnis führt (unendliche Reduktion / unendliche Rekursion), dann führt Call-By-Need eventuell doch zu einem Ergebnis.*

## Kalkül

Ein Kalkül hat zwei Bestandteile:

### Ausdrücke

Ein Kalkül umfasst die Definition von Ausdrücken (Termen)

### Ein Deduktionssystem

Ein Kalkül definiert ein Deduktionssystem mit dem zutreffende Aussagen getroffen werden.
Das Deduktionssystem besteht aus Regeln die aufgeteilt werden können in

- **Axiome** Bedingungslose Aussagen (ohne Begründung wahr)
- **Ableitungsregeln** Schlussfolgerungen

## Reduktionen im λ-Kalkül als Deduktion der Äquivalenz

$$\frac{}{e \rightsquigarrow e}$$

$$\frac{e_1 \rightsquigarrow e_2 \quad e_2 \rightsquigarrow e_3}{e_1 \rightsquigarrow e_3}$$

$$\frac{e_1 \rightsquigarrow e_1' \quad e_2 \rightsquigarrow e_2'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2'}$$

$$\frac{}{\lambda x.\ e \rightsquigarrow \lambda y.e[y/x]}$$

$$\frac{}{(\lambda x.e_1)\ e_2 \rightsquigarrow e_1[x/e_2]}$$

$$\frac{e \rightsquigarrow e'}{\lambda x.\ e \rightsquigarrow \lambda x.\ e'}$$

# Implementierung der Reduktion

```
object Lambda_App extends App {
  println(ReducingEvaluator.eval(LambdaParser.parse("(/ x . x) y")))
}
```

Variable('y)

```
object ReducingEvaluator {

  // replace x by e2 in e1
  private def substitute (x : Symbol, e2: Exp, e1 : Exp) : Exp =
    e1 match {
        case Variable(y) if x != y => Variable(y)

        case Variable(x) => e2

        case Abstraction (y, body) =>
            if (x == y) e1 else Abstraction (y, substitute (x, e2, body))

        case Application (fun, arg) =>
            Application(substitute (x, e2, fun), substitute (x, e2, arg))
    }

  // reduce one step
  private def reduce(e: Exp) : Exp =
    e match {
        case Application (Abstraction(v, body) ,arg)  => substitute(v, arg, body)
        case Application (e1, e2) => Application (reduce(e1), reduce(e2))
        case Abstraction(v, body)  => Abstraction(v, reduce(body))
        case _ => e
    }

  // reduce until nothing happens
  def eval(e: Exp) : Exp = {
    var er = reduce(e)
    if (e == er) e else reduce(er)
  }
}
```

# Bedeutung

## Mathematik / theoretische Informatik:

Untersuchungen zur Berechenbarkeit, Term-Systeme, etc.

## Praktische Informatik

Minimalistische Programmiersprache mit den Kernkonzepten der Programmiersprachen

## Historie

| | |
|---|---|
| McCarthy: | 1960er Wiederentdeckung des λ-Kalküls |
| | Lisp ~> Funktionale Sprachen |
| | in stetigem Strom wird alle Jahre der λ-Kalkül in einer neuen Variation erfunden – aktuell *Clojure* |
| Strachey / Landin: | Semantik von Programmiersprachen, (Sequenzielle) Programme ~ λ-Ausdrücke |
| Strachey / Scott | mathematisch fundierte Grundlagen der Semantik Programmiersprachen basierend auf λ-Kalkül |