

# FUNKTIONALE PROGRAMMIERKONZEPTE UND REKURSION

## Kurzbeschreibung

In dieser Kurseinheit werden die Funktionsapplikation und die Rekursion als die zentralen Elemente der funktionalen Programmierung vorgestellt und am Beispiel des rekursiven Java-Methodenaufrufs veranschaulicht.

## Schlüsselwörter

Funktionale Programmierung, applikative Programmierung, Funktionsanwendung, Funktionsapplikation, Funktionsabstraktion, Konversionen, Rekursion, rekursiver Java-Methodenaufruf, Nachklappern, Iteration

## Lernziele

1. Der Zusammenhang der funktionalen Programmierung zu den Rechenstrukturen wird verstanden.
2. Die Sprachelemente der funktionalen Programmierung sind bekannt und können innerhalb der Programmierung genutzt werden.
3. Das Prinzip der Rekursion wird verstanden und rekursive Java-Programme können erstellt werden.

## Hauptquellen

- Manfred Broy: Informatik – Eine grundlegende Einführung, Band 1: Programmierung und Rechenstrukturen, Springer Verlag 1998.
- Hanspeter Mössenböck: Sprechen Sie Java? – Eine Einführung in das systematische Programmieren, dpunkt.verlag 2003.

## Inhaltsverzeichnis

1	FUNKTIONALE PROGRAMMIERKONZEPTE .....	2
1.1	Syntax und Semantik eines funktionalen Programmen .....	4
1.2	Termersetzungskonzept .....	6
1.3	Identifikatoren .....	8
1.4	Bedingte Ausdrücke.....	10
1.5	Funktionsanwendung und Funktionsabstraktion .....	11
2	REKURSIVE FUNKTIONSDEKLARATION .....	15
2.1	Rekursiver Java-Methodenaufruf .....	15
2.2	Weitere Beispiele.....	16
	VERZEICHNISSE .....	20
	Abkürzungen und Glossar .....	20
	Index .....	20
	Informationen und Interaktionen .....	21
	Literatur .....	21

- FUNKTIONALE PROGRAMMIERKONZEPTE
  - Sprachelemente, Syntax und Semantik, Termersetzungskonzept, Bedingte Ausdrücke, Funktionsanwendung, Funktionsabstraktion
- REKURSIVE FUNKTIONSDEKLARATION
  - Rekursiver Java-Methodenaufruf, Nachklappern, Beispiele add(), mult(), mod(), fact()

### Information 1: FUNKTIONALE PROGRAMMIERKONZEPTE UND REKURSION

## 1 FUNKTIONALE PROGRAMMIERKONZEPTE

Funktionale Programmierkonzepte – die auch synonym als applikative Programmierkonzepte bezeichnet werden – stehen in einem engen Zusammenhang mit den in der Kurseinheit RECHENSTRUKTUREN [C&M-RS] beschriebenen Termersetzungssystemen. Wie in Interaktion 1 ausgeführt ist, sind funktionale Programme die Terme und die Ausführung des Programms stellt eine durch ein Rechensystem realisierbare Termersetzung dar [Br98].

- Ein Programm lässt sich als ein Term auffassen
  - kann auf einem Rechensystem ausgeführt werden, falls ein allgemeiner Auswertungsalgorithmus (“Interpreter”) verfügbar ist
  - im Term treten Funktionen auf
- Das zentrale in einem funktionalen Programm auftretende Sprachelement ist die Funktionsanwendung
- Beispiel: Fakultätsfunktion !
  - mathematische Spezifikation

$$\begin{aligned} !: \mathbb{N} &\rightarrow \mathbb{N} & 0! &= 1, \\ (n)! &= (n-1)! \cdot n \end{aligned}$$

$$n \in \mathbb{N}, n! = \left\{ \begin{array}{l} \text{_____} \\ \text{_____} \end{array} \right.$$

### Interaktion 1: FUNKTIONALE PROGRAMMIERKONZEPTE – Funktionales Programm

Der allgemeine Ausführungsalgorithmus zu einer Programmiersprache entspricht den Reduktionssystemen bei den Termersetzungssystemen. Sofern diese Form des Reduktionssystems, also der Interpreter, auf einem Rechensystem ausgeführt werden kann, sind die Algorithmen auf dem Rechensystem ablaufähig.

Das beherrschende Sprachelement der funktionalen Programmiersprachen ist die Funktionsanwendung, die auch als Funktionsapplikation bezeichnet wird. Dieses Sprachelement tritt in jeder höheren Programmiersprache auf. Funktionale Sprachen beschränken sich auf dieses Sprachelement und verzichten auf ablaforientierte Strukturen, wie z.B. Schleifen.

Als ein einfaches Beispiel wird in Interaktion 1 die Fakultätsfunktion eingeführt, die in Form durch eine Fallunterscheidung bezüglich des Wertes  $n$  eindeutig beschrieben ist. Ein funktionales Programm, das die Fakultätsfunktion berechnet, könnte z.B. als Java-Programm formuliert werden, wie im Folgenden gezeigt wird.

- Zahlreiche gegebene Funktionssymbole und Operationen werden vorausgesetzt
  - z.B. Addition (+), Vergleich (==)
- Die Fallunterscheidung (if-then-else) ist ein Element der funktionalen Sprache
  - wird nicht als eine (nicht-strikte) Funktion aufgefasst, sondern als fester Bestandteil der Sprache
- Fakultätsfunktion als funktionales Programm (Java-Syntax)

```
static long fact (int n) {
  if (n == 0) return 1;
  else return fact(n - 1) * n;
}
```

- Genutzte Funktionssymbole

---

- Genutzte Sprachelemente

---



---

### Interaktion 2: Elemente eines funktionalen Programms

Die als gegeben vorausgesetzten Operationen bilden die so genannte algorithmische Basis. Sie werden als elementare oder primitive Operationen bzw. primitive Funktionen bezeichnet. Hierzu zählt neben den arithmetischen Operationen auch die Fallunterscheidung, die als Element der funktionalen Sprache angesehen wird.

Am Beispiel des in Interaktion 2 angegebenen funktionalen Programms, das die zuvor eingeführte mathematische Beschreibung der Fakultätsfunktion umsetzt, lassen sich die genutzten Sprachelemente identifizieren.

Bei diesem Beispiel eines funktionalen Programms tritt eine Rekursion auf. Auf die Rekursion wird im nächsten Kapitel näher eingegangen, nachdem die elementaren Konzepte der funktionalen Programmierung eingeführt wurden.

- Ein funktionales Programm besteht aus der Definition von Funktionen und deren Applikation innerhalb von Termen
- Beinhaltet der Term nur primitive Operationen, so heißt er primitiver Term, andernfalls (nichtprimitiver) Ausdruck
- Welche Art von Term liegt bei dem zuvor behandelten funktionalen Programm vor, das die Fakultät berechnet?

- 
- Was muss gegeben sein, damit sich lässt sich die Fakultätsfunktion funktional in Form eines primitiven Terms berechnen lässt?

---

### Interaktion 3: Funktion und Funktionsanwendung

Da das zentrale Sprachmittel der funktionalen Programmiersprache die Funktionsanwendung ist, besteht ein funktionales Programm aus der Definition von Funktionen und deren Anwendung in Termen. Die Sprachelemente führen unmittelbar zu einer Aufteilung der in einem funktionalen Programm auftretenden Terme. Es bestehen

- primitive Terme, die nur primitive Funktionen beinhalten und
- nichtprimitive Ausdrücke, die zuvor definierte Funktionen anwenden.

Am Beispiel der Fakultätsfunktion sind in Interaktion 3 die beiden Arten von Terme zu verdeutlichen.

## 1.1 Syntax und Semantik eines funktionalen Programmen

Nachfolgend werden der Aufbau (Syntax) eines funktionalen Programms und dessen Bedeutung (Semantik) behandelt.

- Die formale Sprache der Ausdrücke (Expressions Expr) in Backus Naur Form (BNF)

Expr	=	id   $\perp$   FunctionApplication   ConditionalExpression   (Expr)   MonadOp Expr   Expr DyadOp Expr   Block
MonadOp	=	"-"   "–"
DyadOp	=	"+"   "-"   "*"   "/"   "<"   "=="   ">"   ">="   "^"   "v"

- Forderung weiterer syntaktischer Restriktionen in Form von Kontextbedingungen
  - z.B. jedem Ausdruck kann genau eine Sorte zugeordnet werden

### Information 2: Syntax von Ausdrücken

Ein Ausdruck kann sein

- ein Identifikator, wobei auch das die undefinierte angegebene *Bottom*-Element  $\perp$  als spezieller Identifikator zugelassen wird,
- eine Funktionsanwendung oder ein bedingter Ausdruck (if),
- ein geklammerter oder ein in Infix- oder Präfix-Notation gebildeter Ausdruck,
- eine zur Einführung von lokalen Deklarationen dienende syntaktische Einheit Block, die (wie auch verschiedene andere Nichtterminale) später im Detail eingeführt werden.

Danach sind die monadischen (1-stelligen) bzw. dyadischen (2-stelligen) primitiven Operationen angegeben, die im Ausdruck auftreten dürfen. Dabei wird zwischen arithmetischen Operationen und booleschen Operationen unterschieden.

Die in Information 2 in Backus Naur Form (BNF, siehe Kurseinheit PROGRAMMIERGRUNDLAGEN [C&M-PG]) formulierten Regeln beschreiben den syntaktischen Aufbau von Ausdrücken. Es gibt noch weitere den syntaktischen Aufbau einschränkende Bedingungen (so genannte Kontextbedingungen), die neben den BNF-Regeln formuliert werden. Eine solche Kontextbedingung ist beispielsweise, dass jedem Ausdruck eine Sorte zugeordnet werden können muss. Diese Sorte wird durch das Ergebnis des Ausdrucks bestimmt und ist abhängig von der Zuordnung von Sorten zu den auftretenden Identifikatoren.

Programmiersprachen, für die diese Kontextbedingung gilt, heißen stark typisierte Sprachen. Eine starke Typisierung bietet einige Vorteile, wie z.B. die automatische Erkennung von Programmierfehlern aufgrund von Typkonflikten oder eine bessere Strukturierung und Lesbarkeit von Programmen.

Sehr viel schwieriger als die Syntax ist die Bedeutung der syntaktischen Beschreibung, also die Semantik des funktionalen Programms, festzulegen.

- Operationelle Semantik
  - Termersetzungsemantik
  - wird durch einen Termersetzungsalgorithmus dargestellt
  - Ausdrücke werden unter gewissen Nebenbedingungen in eine Normalform gebracht
  - nicht-wohldefinierte Terme (d.h. Terme  $t$ , für die  $I(t) = \perp$  gilt) besitzen keine terminierende Berechnung
  
- Funktionale Semantik
  - Definition einer Interpretationsfunktion  $I$
  - bei der Festlegung bildet eine Rechenstruktur die Grundlage
  - Ausdrücke der Programmiersprache werden auf mathematische Elemente (ihre Werte) abgebildet
  - Datenelemente für die semantische Interpretation von Ausdrücken
    - $D := \{a \in s^A : s \in S\}$
  - Menge der Funktionen, die in den Ausdrücken verwendet werden:
    - $FCT := \{f : s_1^A \times \dots \times s_n^A \rightarrow s_{n+1}^A : n \in \mathbb{N} \wedge f \text{ strikt} \\ \wedge \forall i, 1 \leq i \leq n+1: s_i \in S\}$

### Information 3: Semantik von Ausdrücken

Es werden grundsätzlich zwei verschiedene Semantik-Formen, die funktionale und die operationelle Semantik unterschieden. Funktionalen Programmiersprachen kann eine funktionale, aber auch eine operationelle Semantik zugeordnet werden. Ganz analog kann den imperativen Programmiersprachen neben einer operationellen auch eine funktionale Semantik zugewiesen werden.

Wird die operationelle Semantik auf funktionale Programme angewendet, so resultiert hieraus eine Termersetzungsemantik. Das Ziel der Termersetzung, die durch einen entsprechenden Algorithmus erfolgt, ist dabei die Erzielung einer gewissen Normalform.

Es ist zu beachten, dass es auch nicht wohldefinierte Terme gibt: Ein nicht-wohldefinierter Term  $t$  ist dadurch charakterisiert, dass für die Interpretationsfunktion  $I(t) = \perp$  gilt. Die hier zur Charakterisierung eines nicht-wohldefinierten Terms benutzte Interpretationsfunktion  $I$  führt direkt zur funktionalen Semantik. Diese stützt sich auf Abbildungen, also Funktionen von gewissen mathematischen Elementen ab, die sich in Form der Menge  $D$  fassen lassen. Die Grundlage bilden die Rechenstrukturen [C&M-RS].

## 1.2 Termersetzungskonzept

Bezüglich der Anwendungsmöglichkeit der Termersetzungregeln wird bei der operationellen Semantik für Programmiersprachen von den Termersetzungssystemen abgewichen, da Termersetzungregeln nicht an einer beliebigen Stelle eines Terms angewendet werden dürfen.

- Für Termersetzungssysteme gilt, dass Termersetzungsregeln in Termersetzungsalgorithmen an beliebigen Stellen in einem Term angewendet werden dürfen
  - gilt nicht für die operationelle Semantik von Programmiersprachen
- Effizienzsteigerung
  - für die Berechnung des Resultats nicht benötigte Auswertungen können unterdrückt werden
- Terminierungseigenschaft
  - nichtterminierende, aber nicht benötigte Teilausdrücke gefährden die Terminierung des Gesamtausdrucks nicht

```
if (true) { // Anweisungsfolge 1 }
else { // Anweisungsfolge 2 }
```

Effizienzsteigerung, weil

Terminierungsgefährdung, falls

### Interaktion 4: Termersetzungskonzept

Das eingeschränkte Termersetzungskonzept hat zwei Vorteile, die Effizienz und die Terminierungseigenschaft, wie anhand des in Interaktion 4 angegebenen Beispiels verdeutlicht wird.

Die Forderung nach einer Einschränkung der Anwendung von Termersetzungsregeln führt zu den bedingten Termersetzungsregeln.

- Aussehen von bedingten Termersetzungsregeln
  - $t_1 \rightarrow r_1 \wedge \dots \wedge t_n \rightarrow r_n \Rightarrow t_{n+1} \rightarrow r_{n+1}$
- Beispiel: Funktion cor (conditional or, bedingtes Oder)
  - fct cor = (bool, bool) bool
- Auswertungsregeln:
  1.  $x \rightarrow z \Rightarrow \text{cor}(x, y) \rightarrow \text{cor}(z, y)$ ,
  2.  $\text{cor}(\text{true}, y) \rightarrow \text{true}$ ,
  3.  $\text{cor}(\text{false}, y) \rightarrow y$ .
- Ergebnis: cor( $t_1, t_2$ ) terminiert auch dann sicher,
  - wenn Term  $t_1$  terminiert und true liefert
  - selbst für den Fall, dass Term  $t_2$  nicht terminiert

Wertetabelle zu cor

cor	I(t2)	L	O	⊥
I(t1)				
L				
O				
⊥				

### Interaktion 5: Bedingte Termersetzung

Eine bedingte Termersetzungsregel bedeutet: Die Ersetzung des Term  $t_{n+1}$  durch  $r_{n+1}$  ist nur unter der Bedingung zulässig, dass zuvor die Terme  $t_i$  durch  $r_i$  ( $i = 1, \dots, n$ ) ersetzt wurden.

Die bedingten Termersetzungsgesetze werden in Interaktion 5 am Beispiel des bedingten Oder, dem *conditional or* (cor) verdeutlicht.

Anhand der in Interaktion 5 aufzustellenden Wertetabelle wird ersichtlich, dass die cor-Funktion an genau einer Stelle einen zur (normalen) or-Funktion unterschiedlichen Wertverlauf aufweist.

### 1.3 Identifikatoren

Identifikatoren traten bereits im Zusammenhang mit dem Aufbau von Termen auf. In Programmiersprachen haben Identifikatoren eine große Bedeutung (siehe Information 4).

[www.cm-tm.uka.de/info1](http://www.cm-tm.uka.de/info1)  
Info1-Team (Prof. Abeck)

- Identifikatoren werden in Programmiersprachen für unterschiedliche Zwecke genutzt, wie z.B.
  - zur Bezeichnung von Zwischenergebnissen
  - als Parameter in Ausdrücken
  - zur Bezeichnung von Funktionen
- Syntaktischer Aufbau von Identifikatoren
  - Beispiel: erstes Zeichen muss ein lateinischer Buchstabe sein
    - hierdurch eine Unterscheidung von Zahlen möglich
- In funktionalen Programmiersprachen stehen Identifikatoren für
  - Elemente aus der Menge D (Elemente der Trägermengen)
  - Elemente aus der Menge FCT (n-stellige Funktionen)

#### **Information 4: Identifikatoren in Programmiersprachen**

An den Aufbau von Identifikatoren werden in Programmiersprachen üblicherweise gewisse syntaktische Anforderungen gestellt. Durch die Vorgabe, dass Identifikatoren mit einem Buchstaben beginnen müssen, lassen sie sich von Zahlen unterscheiden. Diese Anforderung lässt sich leicht durch eine entsprechende BNF-Regel formulieren.

Schwieriger zu behandeln ist die Frage der Semantik von Identifikatoren in funktionalen Programmen. Zum einen kann sich hinter einem Identifikator ein Datenelement, also ein Element aus den Trägermengen verbergen (diese Menge wurde mit D bezeichnet) oder es handelt sich um eine Funktion, bezeichnet mit der Menge FCT.

- Ausdrücken mit freien Identifikatoren wird eine Bedeutung dadurch zugeordnet, dass diese Identifikatoren belegt werden
  - $ENV := \{\beta: ID \rightarrow H\}$
  - $I: \langle exp \rangle \rightarrow (ENV \rightarrow H)$
- Interpretation einzelner Identifikatoren
  - $I_{\beta}[x] = \beta(x)$
  - Einführung einer speziellen Abbildung  $\Omega$ , die allen Identifikatoren das Element  $\perp$  zuordnet
 
$$\Omega : ID \rightarrow H \text{ mit } \Omega(x) = \perp$$
- Konstanten sind Zeichen und Zeichenfolgen mit fester, von der Belegung unabhängiger Interpretation
  - Deutung als 0-stellige Funktionen der Rechenstruktur möglich

### Information 5: Zuordnung von Bedeutungen zu Identifikatoren

Wie in Information 5 näher ausgeführt ist, wird die Menge  $H$  der semantischen Elemente benötigt, um den Identifikatoren eine Bedeutung, also eine Semantik zuzuordnen.

Eine Abbildung  $\beta$  weist jeder syntaktischen Beschreibung eines Identifikators (zusammengefasst in der Menge  $ID$ ) ein semantisches Element aus  $H$  zu. Die Abbildung  $\beta$  heißt eine Belegung.

Die Menge aller Belegungen wird mit  $ENV$  (*Environment*, Umgebung) bezeichnet. Mittels der Umgebung  $ENV$  lässt sich auch einem beliebigen Ausdruck, in dem Identifikatoren aus der Umgebung vorkommen, ein semantischer Wert aus  $H$  zuordnen. Die Abbildung, die das leistet, ist die Interpretationsvorschrift  $I$ .

Die Interpretationsvorschrift  $I$  lässt sich problemlos auf einen einzelnen Identifikator  $x$  anwenden (dieser bildet ja auch einen Ausdruck). Gemäß der Definition der Umgebung  $ENV$  ist die Interpretation eines Identifikators gerade die Belegungsabbildung

Das von  $\beta(x)$  einem Identifikator zugewiesene semantische Element kann auch das undefiniert-Element  $\perp$  sein, weil dieses zu  $H$  gehört. Es ist somit eine korrekte Belegung, jedem Identifikator  $\perp$  (Undefiniert-Element) zuzuordnen. Diese Belegung wird speziell gekennzeichnet und als  $\Omega$ -Abbildung bezeichnet.

Konstanten haben üblicherweise den gleichen syntaktischen Aufbau wie Identifikatoren, aber eine unterschiedliche Semantik. Der semantische Unterschied besteht darin, dass Konstanten unabhängig von der Belegung ein semantisches Element darstellen.

Mit den Identifikatoren und Konstanten sowie deren Syntax und Semantik wurde bereits ein Sprachelement einer funktionalen Programmiersprache vorgestellt. Dieses Sprachelement wird im Folgenden durch weitere ergänzt. Es wird dabei von der Annahme ausgegangen, dass die Rechenstrukturen  $BOOL$  und  $INT$  vorhanden sind. Das bedeutet konkret, dass die in den entsprechenden Signaturen vorhandenen Sorten und Funktionen in den Programmen verwendet werden.

## 1.4 Bedingte Ausdrücke

Die Nutzung der in der Kurseinheit RECHENSTRUKTUREN [C&M-RS] ausführlich beschriebenen Rechenstruktur BOOL und der darin auftretenden Sorte bool wird anhand der bedingten Ausdrücke und des darin verwendeten booleschen Ausdrucks B deutlich.

[www.cm-tm.uka.de/info1](http://www.cm-tm.uka.de/info1)  
Info1-Team (Prof. Abeck)

- Ein bedingter Ausdruck der Sorte s hat den folgenden Aufbau

if B then E else E'

- (1) B: Ausdruck der Sorte bool
- (2) E, E': beliebige Ausdrücke gleicher Sorte s

- Syntax:

ConditionalExpression = "if" "(" Expr ")" Expr ["else" Expr]

- Semantik:

$$I_{\beta}[\text{if (B) E else E'}] = \begin{cases} I_{\beta}[E] & \text{falls } I_{\beta}[B] = L \\ I_{\beta}[E'] & \text{falls } I_{\beta}[B] = O \\ \perp & \text{sonst} \end{cases}$$

### Information 6: Bedingte Ausdrücke

E und E' heißen Zweige. Die beiden Anforderungen an die Sorte von B, E und E' sind syntaktische Nebenbedingungen. Beide Nebenbedingungen machen eine Vorgabe an die erwarteten Typen bzw. Sorten. Zu beachten ist, dass eine stark typisierte Sprache zugrunde gelegt wird.

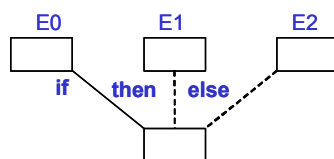
Die semantischen Interpretation  $I_{\beta}$  besagt, dass der bedingte Ausdruck einer nichtstrikten Abbildung entspricht: Auch wenn ein Ausdruck in einem der Zweige zu undefiniert ( $\perp$ ) interpretiert wird, kann der Wert des bedingten Ausdrucks verschieden von  $\perp$  sein.

[www.cm-tm.uka.de/info1](http://www.cm-tm.uka.de/info1)  
Info1-Team (Prof. Abeck)

- Beschreibung der Semantik durch folgende Axiome

1. if (true) E else E' = E
2. if (false) E else E' = E'
3. if (B) E else E' = if ( $\neg B$ ) E' else E
4. if ( $\perp$ ) E else E' =  $\perp$

- Beschreibung durch Formulare



Das Formular ist so auszufüllen, dass obiges Axiom 1 beschrieben wird

### Interaktion 6: Axiome und Formulare zu bedingten Ausdrücken

Die in Interaktion 6 angegebenen Axiome in der Gestalt von Gleichungen beschreiben die Semantik bedingter Ausdrücke. Das Gleichungssystem gestattet, zu jeder denkbaren

Wertebelegung einen Resultatwert abzuleiten. In diesem Zusammenhang wird von einer Wertverlaufssemantik gesprochen. Diese Art von Semantik kann z.B. auch zur Festlegung der Bedeutung von aussagenlogischen Termen genutzt werden.

Die Axiome stellen somit eine bestimmte Art der Semantikbeschreibung dar. Für die Syntax wurde bereits die Darstellungsform der Formulare eingeführt. Für bedingte Ausdrücke sieht das (Berechnungs-) Formular wie in Interaktion 6 angegeben aus.

Die gestrichelten Linien haben bei der Formuldarstellung die Bedeutung, dass die Ausdrücke (hier  $E_1$  und  $E_2$ ) nicht immer ausgewertet werden müssen.

## 1.5 Funktionsanwendung und Funktionsabstraktion

Das Sprachelement der Funktionsanwendung und die damit eng verbundene Funktionsabstraktion haben eine zentrale Bedeutung in den funktionalen Sprachen.

[www.cm-tm.uka.de/info1](http://www.cm-tm.uka.de/info1)  
Info1-Team (Prof. Abeck)



- $F(E_1, \dots, E_n)$  heißt Funktionsanwendung
  - äquivalente Bezeichnungen sind Funktionsapplikation oder Funktionsaufruf
- Nebenbedingungen
  - die durch  $F$  bezeichnete Funktion muss  $n$ -stellig sein
  - die Sorten der  $E_1, \dots, E_n$  müssen der Funktions-Funktionalität entsprechen
- $f(a_1, \dots, a_n)$  ist der Wert der Funktionsanwendung
  - $f$  ist die durch  $F$  bezeichnete Funktion
  - $a_1, \dots, a_n$  sind die Werte der Ausdrücke
- Syntax der Funktionsanwendung als BNF

als Syntaxdiagramm

FunctionApplication = Function "(" [Expr {"," Expr}] ")"



### Interaktion 7: Funktionsanwendung

Durch die Funktionsanwendung lässt sich eine Funktion  $F$  auf "geeignete" Ausdrücke anwenden. Die bei der Funktionsanwendung einzuhaltenden syntaktischen Nebenbedingungen betreffen

- (1) die Anzahl der Ausdrücke (Stelligkeit der Funktion),
- (2) die Sorten der Ausdrücke (Funktionalität der Funktion).

Es ist zwischen der Funktionsanwendung  $F(E_1, \dots, E_n)$  und seinem Wert  $f(a_1, \dots, a_n)$  zu unterscheiden. Der Wert der Funktionsanwendung, also  $f$ , wird auch einfach als „Funktion“ bezeichnet.

Die Syntax einer Funktionsanwendung lässt sich durch eine einfache BNF-Regel beschreiben, die in Interaktion 7 in ein äquivalentes Syntaxdiagramm umzuwandeln ist. Üblicherweise wird eine Funktionsanwendung in Präfix-Schreibweise mit geklammerter Folge von aktuellen Argumenten formuliert.

- $I_{\beta}[F(E_1, \dots, E_n)] = \begin{cases} f(I_{\beta}[E_1], \dots, I_{\beta}[E_n]) & \text{falls } \forall i, 1 \leq i \leq n : I_{\beta}[E_i] \neq \perp \\ \perp & \text{sonst} \end{cases}$
- $f$  ist dabei die Interpretation des Funktionsausdrucks  $F$  unter der Belegung  $\beta$ , d.h.  $f = I_{\beta}[F]$
- Auswertung eines Funktionsaufrufs
  - Call-by-Value (Wertaufruf)
    - zuerst werden in beliebiger Reihenfolge die Argumentausdrücke ausgewertet
  - Alternative: Call-by-Name (Namensaufruf)
    - es wird mit der Auswertung des Funktionsaufrufs begonnen, bevor die Parameterausdrücke ausgewertet sind

### Information 7: Interpretation und Auswertung der Funktionsanwendung

Nach der Beschreibung der Syntax ist auch bei diesem Sprachelement wieder die Frage nach der Semantik zu stellen. Es ist also die Interpretationsabbildung  $I$  zu diesem Sprachelement festzulegen. Die Interpretation der Funktionsanwendung bedeutet die Anwendung der durch  $f$  beschriebenen Funktion auf die Werte der aktuellen Parameterausdrücke.  $f$  ist dabei die Interpretation des Funktionsausdrucks  $F$  unter der Belegung  $\beta$ , d.h.  $f = I_{\beta}[F]$ .

Bezüglich des Vorgehens, wie der Funktionsaufruf ausgewertet wird, lassen sich zwei grundsätzlich verschiedene Strategien unterscheiden:

- (1) Alle Argumentausdrücke müssen ausgewertet sein, bevor mit der Auswertung des Funktionsaufrufs begonnen wird (*Call-by-Value*, d.h. Funktionsaufruf mit dem Wert der Parameter).
- (2) Die Argumentausdrücke müssen noch nicht ausgewertet sein (*Call-by-Name*).


Im Weiteren wird ausschließlich von der *Call-by-Value*-Auswertungsstrategie ausgegangen.

- Aufgabe eines Identifikators
  - hat die Funktion eines Platzhalters
    - steht für einen Wert, der später eingesetzt wird
  - bezeichnet ein bestimmtes, unter Umständen erst später genauer anzugebendes Element
    - Vergleich mit der Mathematik: „Sei  $x$  ein Element der Menge  $M$  mit dem Wert ...“
- Beispiel:  $x \cdot (x+1)$ 
  - Funktion, die eine natürliche Zahl auf eine andere abbildet  
 $f: \mathbb{N} \rightarrow \mathbb{N}$
  - wird definiert durch die Gleichung  
 $f(x) = x \cdot (x+1)$  für alle  $x \in \mathbb{N}$
  - Notation in Programmiersprachen: Funktionsdeklaration

### Information 8: Identifikatoren in Funktionen

Die Platzhalterfunktion von Identifikatoren drückt aus, dass diese Identifikatoren den Platz für einen später einzusetzenden Wert einnehmen. Identifikatoren weisen auch die Eigenschaft auf, dass sie im Allgemeinen konsistent gegen einen Identifikator mit anderer Bezeichnung ersetzt werden können. Das gilt genau dann, wenn durch die neue Bezeichnung keine Beziehungskonflikte auftreten. Im obigen Beispiel  $x \cdot (x+1)$  führt die konsistente Ersetzung von  $x$  gegen  $y$  zu  $y \cdot (y+1)$ . Zwischen der Interpretation von  $x \cdot (x+1)$  und  $y \cdot (y+1)$  wird nicht weiter unterschieden.

Die nachfolgenden Ausführungen dienen als Vorbereitung zur Einführung des Sprachelements der Funktionsdefinition.

[www.cm-tm.uka.de/info1](http://www.cm-tm.uka.de/info1)  
Info1-Team (Prof. Abeck) 

- Notation, bei der auf die Einführung einer Bezeichnung  $f$  für die Funktion verzichtet wird:  $(\text{nat } x) \text{ nat: } x \cdot (x+1)$ 
  - $x$  wird im Funktionsbereich gebunden genannt (im Gegensatz zu frei)
  - $x$  ist syntaktisch durch die Kennzeichnung  $\text{nat } x$  als Platzhalter (formaler Parameter) ausgezeichnet
- Gebundene Identifikatoren können durch beliebige andere Identifikatoren konsistent ersetzt werden
  - Voraussetzung: keine Konflikte zu freiem Auftreten der betreffenden Bezeichnung im betrachteten Ausdruck
  - im Beispiel kann  $x$  gegen  $y$  ersetzt werden:  $(\text{nat } y) \text{ nat: } y \cdot (y+1)$
  - Kann  $x$  gegen  $y$  in  $(\text{nat } x): x \cdot (x+y)$  ersetzt werden?

- 
- Diese Umbenennung gebundener Bezeichnungen heißt  $\alpha$ -Konversion

### **Interaktion 8: Gebundene und freie Identifikatoren**

Zunächst wird eine notationelle Änderung an der Funktionsschreibweise vorgenommen, indem auf die Funktionsbezeichnung  $f$  verzichtet wird, wie in Interaktion 8 am Beispiel  $(\text{nat } x) \text{ nat: } x \cdot (x+1)$  gezeigt wird.

Durch die Kennzeichnung  $\text{nat } x$  als Platzhalter wird  $x$  im Funktionsausdruck gebunden. Freie Identifikatoren sind in Funktionsausdrücken genau solche Identifikatoren, die keine formalen Parameter sind.

Gebundene Identifikatoren haben die Eigenschaft, dass sie konsistent ersetzt werden können. Allerdings muss man darauf achten, dass durch die Umbenennung keine Konflikte zu nicht gebundenen Auftreten der Bezeichnung entstehen, wie anhand des abgeänderten Beispiels in Interaktion 8 verdeutlicht wird.

- Schreibweise für eine Funktionsabstraktion  $(s_1 x_1, \dots, s_n x_n) s : E$ 
  - $E$  ist ein Ausdruck der Sorte  $s$
  - $x_1, \dots, x_n$  heißen formale Parameter
  - $(s_1 x_1, \dots, s_n x_n) s$  heißt Kopfzeile der Funktionsabstraktion
    - die Kopfzeile gibt die Funktionalität der Funktionsabstraktion wieder
    - im Beispiel:  $M_1 \times \dots \times M_n \rightarrow M$ 
      - $M_i$  bezeichnen dabei die Mengen der Elemente der Sorten  $s_i$
  - der Ausdruck  $E$  heißt Rumpf der Funktionsabstraktion
- Eine Funktionsabstraktion wird in Verbindung mit einer Funktionsanwendung genutzt
  - $((s_1 x_1, \dots, s_n x_n) s : E) (E_1, \dots, E_n)$

### Information 9: Funktionsabstraktion

Die Bestandteile der Funktionsabstraktion sind:

- ein Ausdruck  $E$ , der den so genannten Rumpf (*Body*) der Funktionsabstraktion darstellt.
- eine Liste von formalen Parametern einschließlich deren Sorten und der Sorte des von  $E$  gelieferten Ergebnisses, die gemeinsam den Kopf (*Head*) oder die Kopfzeile ausmachen. Mit den Sorten sind entsprechende (Träger-) Mengen  $M_i$  verknüpft, die den zulässigen Wertebereich der Parameter und des Funktionsergebnisses festlegen.

Der Kopf gibt die Funktionalität der beschriebenen Funktion an. Da die Sorten gewisse (Träger-) Mengen darstellen, könnte die Funktionalität auch in der Mengenschreibweise angegeben werden.

- Termersetzungsregel:  
 $((s_1 x_1, \dots, s_n x_n) s : E)(E_1, \dots, E_n) \rightarrow E[E_1/x_1, \dots, E_n/x_n]$   
 falls  $E_i$  in Normalform für  $1 \leq i \leq n$ 
  - besagt: vollständig ausgewertete Ausdrücke auf der Argumentposition werden in den Rumpfausdruck der Funktionsabstraktion eingesetzt
  - Auswertung gemäß der Call-by-Value-Auffassung
  - erfüllt die Striktheitsregel
- Unbedingte Termersetzungsregel:  
 $((s_1 x_1, \dots, s_n x_n) s : E)(E_1, \dots, E_n) \rightarrow E[E_1/x_1, \dots, E_n/x_n]$ 
  - Auswertung erfolgt durch Einsetzen der noch nicht auf Normalform gebrachten aktuellen Parameter
  - Auswertung gemäß der Call-by-Name-Auffassung
  - erfüllt nicht die Striktheitsregel

### Information 10: Auswertung der Funktionsanwendung

Die Grundlage zur Auswertung der Funktionsanwendung liefern die Termersetzungsregeln.

Die in Information 10 angegebene Termersetzungsregel besagt, dass die auftretenden  $x_i$  durch die  $E_i$  im Ausdruck  $E$  (der Rumpf der Funktion) zu ersetzen sind. Hierzu muss allerdings sichergestellt sein, dass in  $E_i$  nur primitive Funktionen auftreten. Diese Bedingung entspricht der Forderung, dass die  $E_i$  in Normalform sind. In der Bedingung und damit in der gesamten Termersetzungsregel ist enthalten, dass die Auswertung der Ausdrücke auf der Argumentposition vollständig ausgewertet sind, bevor sie in den Rumpf eingesetzt und dieser dann berechnet wird. Das entspricht genau der Auffassung *Call-by-Value*. Insbesondere wird durch diese Termersetzungsregel und somit durch die *Call-by-Value*-Regel die Striktheitsregel erfüllt.

Es ist aber auch vorstellbar, dass die Auswertung eines Ausdrucks  $E_i$  erst dann vorgenommen wird, wenn im Rumpf das  $x_i$  auftritt. Das führt zur *Call-by-Name*-Auswertung. Durch die Möglichkeit, dass *Call-by-Name* nicht-strikte Funktionen zulässt, entsteht auch der folgende gravierende Unterschied, dass die Auswertung nach der *Call-by-Name*-Regel häufiger terminiert als gemäß der *Call-by-Value*-Regel.

## 2 REKURSIVE FUNKTIONSDEKLARATION

Bislang eingeführte Sprachelemente lassen keine nichtterminierenden (also nicht endlich-beschränkte bzw. unendliche) Berechnungen zu. Nichtterminierung würde nur dann entstehen, wenn die Berechnung irgendeiner Funktion der Basissignatur nicht terminieren würde. Diese Annahme ist allerdings vor dem Hintergrund der tatsächlich angenommenen Basisoperationen, wie die arithmetischen Operationen oder auch die *if*-Funktion nicht aufrechtzuerhalten. Was lediglich passieren kann, ist die Rückgabe eines undefiniert-Wertes  $\perp$ , wie z.B. bei der Division durch 0.

Durch das Sprachelement der Rekursion wird diese Beschränkung der statisch beschränkten Länge der Berechnungssequenz aufgehoben, wodurch das Problem der Nichtterminierung auftreten kann [Mö03].

### 2.1 Rekursiver Java-Methodenaufruf

Das in Information 11 gezeigte Beispiel einer rekursiven Funktion berechnet die Addition natürlicher Zahlen und führt (unter der Voraussetzung, dass die Forderung im Kopf der Funktion erfüllt wird) zu terminierenden Berechnungsfolgen.

- Bislang lassen sich nur Ausdrücke aufschreiben, die stets auf terminierende Berechnungen führen
  - Länge der Berechnungssequenzen damit stets statisch beschränkt
- Durch das Konzept der rekursiven Deklaration von Funktionen wird die Formulierung von Rechenvorschriften mit unbeschränkt langen Berechnungen möglich
- Beispiel: Addition zweier ganzer Zahlen
  - Basis: Rechenstruktur der natürlichen Zahlen mit den Operationssymbolen zero (abgek. 0), succ, pred und Prädikat ==

```
static int add (int x, int y /* x, y >= 0 */) {  
    if (x==0) return y;  
    else return succ(add(pred(x), y));  
}
```

### Information 11: REKURSIVE FUNKTIONSDEKLARATION – Einführung

Die korrekte Arbeitsweise der rekursiven Funktionsdeklaration, durch die die Addition durchgeführt wird, kann man sich geeignet anhand eines Aufrufs mit konkreten Parameterwerten klarmachen.

Die rekursive Funktionsdeklaration ist in Java-Syntax angegeben. Da in Java die natürlichen Zahlen nicht als Zahlentyp zur Verfügung stehen, wird der ganzzahlige Typ `int` verwendet. Die Einschränkung auf die natürlichen Zahlen erfolgt im Beispielprogramm durch einen Kommentar in der Kopfzeile der Funktionsdeklaration.

Bereits an diesem einfachen Beispiel einer rekursiven Funktion lässt sich ein wichtiges Phänomen der Rekursion beobachten: Mit der Berechnung (Ausführung) der Nachfolgerfunktion `succ` kann erst dann begonnen werden, wenn man an das Ende der rekursiven Aufrufkette von `add` gelangt ist. Die Ausführung erfolgt dann in entgegen gesetzter Richtung zur Erzeugung der `succ`-Funktionen. Man nennt dieses Phänomen auch Nachklappern. Als Konsequenz muss die nachklappernde `succ`-Funktionen in bestimmter Form, und zwar gemäß dem Kellerprinzip, zwischengespeichert werden.

## 2.2 Weitere Beispiele

Ähnlich wie im vorhergehenden Beispiel die Nachfolger-Funktion zur rekursiven Berechnung der Addition verwendet wurde, kann die Multiplikation auf der Addition aufbauen. Die Arbeitsweise dieser in Interaktion 9 zu vervollständigenden, rekursiven Funktionsdeklaration kann man sich wiederum anhand eines Aufrufs mit konkreten Werten verdeutlichen.



- Multiplikation
  - kann durch Rekursion auf die Addition zurückgeführt werden
- Division mit Rest
  - Annahme: Additions- und Subtraktionsoperation stehen zur Verfügung
  - Unterverbinden einer Division durch 0, indem eine Zusicherung (assertion) getroffen wird
- Rest der Division: mod
  - analog zu div

```
static int mult (int x, int y /* x, y >= 0 */) {
  if (x==0) return 0;
  else return _____;
}
```

```
static int div (int x, int y /* x >= 0, y > 0 */) {
  if (x < 0) return 0;
  else return _____;
}
```

```
static int mod (int x, int y /* x >= 0, y > 0 */) {
  if (x < y) return ____;
  else return _____;
}
```

### Interaktion 9: Multiplikation, Division mit Rest, Rest der Division

In diesem Fall ist nun die Addition die nachklappernde Funktion. Man kann sich leicht vorstellen, wie aufwendig diese Form der Realisierung der Multiplikation ist, da jede nachklappernde Addition eine Anzahl nachklappernder succ-Funktionen nach sich zieht.

Beim nächsten Beispiel, der Division mit Rest (auch ganzzahlige Division genannt, Infix-Schreibweise  $x \div y$ ) muss eine Division durch 0 verhindert werden, weshalb im Kommentar  $y > 0$  gefordert wird. Ein Verstoß gegen diese Bedingung führt zu einer nichtterminierenden Rekursion.

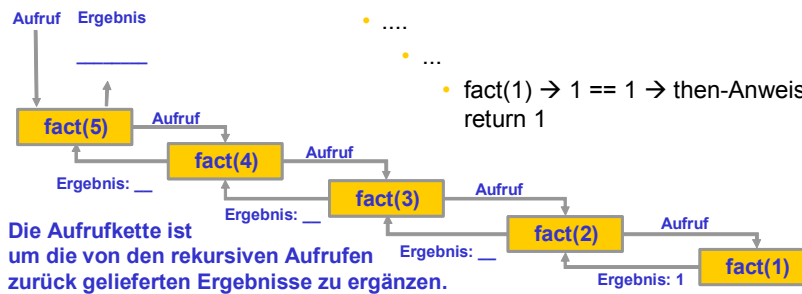
Das Gegenstück zur Division mit Rest (div) ist die Funktion, die den Rest der ganzzahligen Division berechnet, die modulo-Funktion (mod). Für diese Funktion mod gilt die gleiche Restriktion wie für div, dass der Nenner nicht 0 sein darf. Das Berechnungsprinzip ist einfach: Solange  $y$  noch in  $x$  hineingeht ( $x \geq y$ ), zieht man  $y$  von  $x$  ab und probiert es von neuem (rekursiver Funktionsaufruf). Im anderen Fall, also  $x < y$  ist der Wert  $x$  das Ergebnis.

Interaktion 10 greift das am Anfang bereits kennen gelernte Beispiel der Fakultätsfunktion wieder auf und verdeutlicht den Programmablauf, der beim Aufruf von `fact(5)` entsteht.



```
static long fact (int n) {
  if (n==0) return 1;
  else return fact(n-1)*n;
}
```

- Aufruf der Methode fact() mit dem aktuellen Parameterwert 5, also fact(5)
  - Übergabe des aktuellen Parameterwertes 5 an den formalen Parameter n
  - if-Anweisung: Auswertung von  $5 == 1$  liefert false
  - else-Anweisung:  $\text{fact}(n-1) * n$ , d.h. rekursiver Aufruf fact(4)
    - Übergabe des aktuellen Parameterwertes 4 an den formalen Parameter n
    - ....
    - ....
    - fact(1)  $\rightarrow 1 == 1 \rightarrow$  then-Anweisung  $\rightarrow$  return 1



### Interaktion 10: Ablauf einer rekursiven Berechnung am Beispiel der Fakultät

Die Berechnung eines Aufrufs der Funktion fact() führt solange zu einem rekursiven Aufruf, bis der aktuelle Parameter, mit dem die Funktion aufgerufen wird, den Wert 1 erreicht hat. Da sich der Wert des aktuellen Parameters beim rekursiven Aufruf jeweils um 1 verringert, wird dieser Fall beim Aufruf von fact() mit einem Wert  $n > 0$  nach  $n - 1$  rekursiven Aufrufen erreicht.

Wie die in Interaktion 10 abgebildete Aufrufkette für den Aufruf von fact(5) skizziert, erfolgt nach Erreichen des Endes der Aufrufkette bei fact(1) die eigentliche Ergebnisberechnung, indem das zurück gelieferte Ergebnis des rekursiven Aufrufs in den Ausdruck  $\text{fact}(n-1) * n$  eingesetzt wird. D.h., fact(1) liefert 1 zurück, das von fact(2) zur Berechnung des Ergebnisses von  $1 * 2 = 2$  nutzt, womit fact(3) seinerseits das Ergebnis  $2 * 3 = 6$  zurückliefern kann, usw.

Im Falle der Methode fact() handelt es sich um eine so genannte direkte Rekursion, weil sich die Methode direkt selbst aufruft. Liegen Methodenaufrufe zwischen dem rekursiven Aufruf, spricht man von indirekter Rekursion.

- Jedes rekursive Problem kann auch iterativ gelöst werden und umgekehrt
  - es gibt Problemfälle, deren rekursive Lösung kürzer ist und einfacher in der Aufschreibung
  - iterative Lösungen sind üblicherweise kostengünstiger, weil Methodenaufrufe lauffzeit- und speicherintensiv sind
- Rekursionen treten an vielen Stellen in der Informatik auf, wie z.B.
  - bei der Beschreibung von formalen Sprachen  
Deklaration von nichtterminalen Hilfsbezeichnungen für BNF-Ausdrücke
  - rekursive Sortendeklarationen

### Information 12: Abschließende Bemerkungen zur Rekursion

Das behandelte Fakultätsberechnungsproblem könnte auch iterativ, d.h. in Form einer Schleife gelöst werden. Dieser Sachverhalt gilt grundsätzlich für beliebige rekursive Lösungen, was eine wichtige und beweisbare Aussage der Berechenbarkeitstheorie darstellt.

Die Rekursion gehört zu den Kernelementen der Informatik. Wie die in Information 12 angegebenen Beispiele zeigen, tritt die Rekursion nicht nur in Programmiersprachen bei der rekursiven Deklaration von Rechenvorschriften auf, sondern auch im Zusammenhang mit formalen Sprachen und (rekursiven) Datenstruktur-Definitionen, wie in der Kurseinheit OBJEKTORIENTIERTE PROGRAMMIERUNG [C&M-OP]}] verdeutlicht wird.

## VERZEICHNISSE

### Abkürzungen und Glossar

Abkürzung oder Begriff	Langbezeichnung und/oder Begriffserklärung
$\perp$	<i>Bottom-Element</i> Symbolisiert im Zusammenhang mit Zuständen den gedachten „Endzustand“ nichtterminierender Programme. Synonymer Begriff: undefiniert-Element
BNF	Backus Naur Form Schreibweise für Regeln einer Grammatik.
<i>Call-by-Value</i>	Strategie beim Aufruf einer Funktion, die vorschreibt, dass alle Argumentausdrücke ausgewertet sein müssen, bevor mit der Auswertung des Funktionsaufrufs begonnen wird. Deutscher Begriff: Wertaufruf
<i>Call-by-Name</i>	Strategie beim Aufruf einer Funktion, bei der nicht alle Argumentausdrücke ausgewertet sein müssen, bevor mit der Auswertung der Funktion begonnen wird. Deutscher Begriff: Namensaufruf
dyadisch	2-stellig Beispiel: Addition (+) ist ein dyadischer Operator
ENV	<i>Environment</i> Bezeichnet im Zusammenhang mit der Semantik-Festlegung von Identifikatoren die Menge aller Belegungen von Identifikatoren.
Funktionsanwendung	Das beherrschende Sprachelement der funktionalen Programmiersprachen. Synonymer Begriff: Funktionsapplikation
monadisch	1-stellig Beispiel: Negation (-) ist ein monadischer Operator
Nachklappern	Bezeichnung eines Phänomens, das im Zusammenhang mit der verzögerten Berechnung von rekursiven Aufrufen entsteht.
Rekursion	Zurückführen der Lösung eines Problems auf das gleiche Problem.

### Index

$\perp$ 5	Funktionsanwendung 2
Backus Naur Form 5	Nachklappern 16
Call-by-Name 12	Nichtterminierung 15
Call-by-Value 12	Rekursion 3, 15
Environment 9	

## Informationen und Interaktionen

Information 1: FUNKTIONALE PROGRAMMIERKONZEPTE UND REKURSION.....	2
Information 2: Syntax von Ausdrücken .....	5
Information 3: Semantik von Ausdrücken .....	6
Information 4: Identifikatoren in Programmiersprachen.....	8
Information 5: Zuordnung von Bedeutungen zu Identifikatoren .....	9
Information 6: Bedingte Ausdrücke .....	10
Information 7: Interpretation und Auswertung der Funktionsanwendung .....	12
Information 8: Identifikatoren in Funktionen.....	12
Information 9: Funktionsabstraktion .....	14
Information 10: Auswertung der Funktionsanwendung .....	14
Information 11: REKURSIVE FUNKTIONSDEKLARATION – Einführung.....	16
Information 12: Abschließende Bemerkungen zur Rekursion .....	18
Interaktion 1: FUNKTIONALE PROGRAMMIERKONZEPTE – Funktionales Programm.....	2
Interaktion 2: Elemente eines funktionalen Programms .....	3
Interaktion 3: Funktion und Funktionsanwendung .....	4
Interaktion 4: Termersetzungskonzept.....	7
Interaktion 5: Bedingte Termersetzung.....	7
Interaktion 6: Axiome und Formulare zu bedingten Ausdrücken.....	10
Interaktion 7: Funktionsanwendung.....	11
Interaktion 8: Gebundene und freie Identifikatoren.....	13
Interaktion 9: Multiplikation, Division mit Rest, Rest der Division .....	17
Interaktion 10: Ablauf einer rekursiven Berechnung am Beispiel der Fakultät.....	18

## Literatur

- [Br98]            Manfred Broy: Informatik – Eine grundlegende Einführung, Band 1: Programmierung und Rechstrukturen, Springer Verlag 1998.
- [C&M-IP]        Cooperation&Management: IMPERATIVE PROGRAMMIERUNG, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).
- [C&M-OP]        Cooperation&Management: OBJEKTORIENTIERTE PROGRAMMIERUNG, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).
- [C&M-RS]        Cooperation&Management: RECHENSTRUKTUREN, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).
- [Mö03]           Hanspeter Mössenböck: Sprechen Sie Java? – Eine Einführung in das systematische Programmieren, dpunkt.verlag 2003.