Kapitel 5: Applikative Programmierung

In der applikativen Programmierung wird ein Programm als eine mathematische Funktion von Eingabe-in Ausgabewerte betrachtet. Das Ausführen eines Programms besteht dann in der Berechnung des Funktionswertes zu einem gegebenen Eingabewert. Historisch erwachsen ist dieses Paradigma aus dem *Church'schen λ–Kalkü (lambda-Kalkül)l*.

5.1 λ-Kalkül

Gegeben sei ein Alphabet **A**={x_1,…,x_n}. Ein λ–Term ist definiert durch folgende Grammatik:

λ–Term ::= Variable | (λ–Term λ–Term) | λ Variable . λ–Term
Variable ::= x_1 | … | x_n

λ–Terme der Art (t_1 t_2) heißen *Applikation*, λ–Terme der Art λx.t heißen *λ-Abstraktion*.
In der λ-Abstraktion (λx.t) ist die Variable x innerhalb von t *gebunden*. Intuitiv wird der λ–Term (t_1 t_2) gelesen als „die Funktion t_1 angewendet auf Argument t_2" und der λ–Term λx.t als „diejenige Funktion, die x als Argument hat und t als Ergebnis liefert".

Die Auswertung von λ–Termen ist definiert durch die Regel der *β-Konversion*:
(λ x. t_1) t_2 = t_1 [x:=t_2]
Hierbei bezeichnet t_1 [x:=t_2] den Term t_1, wobei jedes freie (d.h. nicht durch ein λ gebundene) Vorkommnis von x durch t_2 ersetzt wird. Klammern werden, soweit eindeutig, weggelassen, wobei Linksassoziativität unterstellt wird: (x y z) = ((x y) z)

Eine weitere Regel ist die der *α-Konversion*, d.h. die Umbenennung von gebundenen Variablen:
λ x. t = λ y. t[x:=y]

Beispiel für eine Reduktion ist etwa die folgende:
((λx. x x) ((λz. z y) x)) →
((λz. z y) x) ((λz. z y) x) →
((λz. z y) x) (x y) →
(x y) (x y)

Eine andere Ableitung für denselben Term wäre z.B.:
((λx. x x) ((λz. z y) x)) →
(λx. x x) (x y) →
(x y) (x y)

Alonzo Church zeigte, dass man mit dem λ-Kalkül und einem kleinen „Trick" alle arithmetischen Funktionen berechnen kann:
Sei

0 ≡ λf.λx. x
1 ≡ λf.λx. f x
2 ≡ λf.λx. f (f x)
3 ≡ λf.λx. f (f (f x))
...
n ≡ λf.λx. f^n x

Dann lassen sich die arithmetischen Funktionen wie folgt definieren:
succ ≡ λn.λf.λx.n f (f x)

plus ≡ λm.λn.λf.λx. m f (n f x)
mult ≡ λm.λn.λf. n (m f)

Als Beispiel zeigen wir, dass 1+1=2:

```
(λm.λn.λf.λx. m f (n f x)) (λf.λx. f x)(λf.λx. f x)
= (λf.λx. (λf.λx. f x) f ((λf.λx. f x) f x))
= (λf.λx. (λx. f x) ((λx. f x) x))
= (λf.λx. (λx. f x) (f x))
= (λf.λx. f (f x))
```

Auch die Kodierung boolescher Ausdrücke ist möglich.
Anforderung an `True`, `False` und `If`:

`(If True M N = M)` und `(If False M N = N)`

Definitionen: `True = (λx.λy.x), False = (λx.λy.y), If = (λi.λt.λe.ite)`
Beweis:

```
If True M N
= (λi.λt.λe.ite) (λx.λy.x) M N
= (λt.λe.(λx.λy.x) te) M N
= (λe.(λx.λy.x) M e) N
= (λx.λy.x) M N
= (λy.M) N
= M

If False M N
= (λiλtλe.ite)(λx.λy.y) M N
= (λx.λy.y) M N
= (λy.y) N
= N
```

Auf diese Weise lässt sich jedes Programm einer beliebigen Programmiersprache auch im λ-Kalkül ausdrücken (sogenannte *Church'sche These* oder *Church-Turing-These*).

Anmerkung: Natürlich lässt sich der λ-Kalkül direkt in Groovy ausdrücken. Beispiel:

```
nul = { f -> { x -> x}}
one = { f -> { x -> f (x) }}
two = { f -> { x -> f (f (x))}}
plus = { m -> { n -> { f -> { x -> (m (f)) ((n(f)) (x))}}}}
```

Leider schlägt aber der folgende Vergleich fehl:

```
assert ((plus (one)) (one)) == two
```

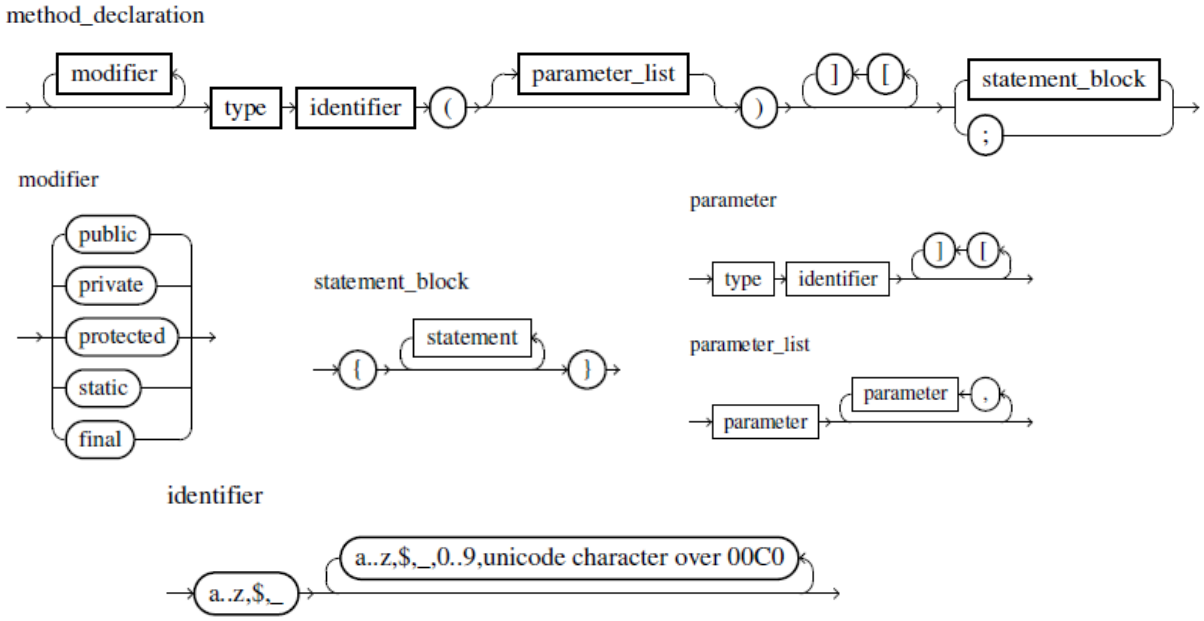
Das liegt daran, dass keine Programmiersprache die Gleichheit von Funktionen feststellen kann! Abhilfe kann dadurch geschaffen werden, dass wir für `f` etwa die Funktion `"I"+x` und für `x` die leere Zeichenreihe einsetzen:

```
f = {x -> "I"+x}
assert ((plus (one)) (one)) (f) ("") == two (f) ("")
(plus (one) (one)) (plus (one) (one)) (f) ("")
```

5.2 Rekursion, Aufruf, Terminierung

Eine *Funktion* oder *Methode* (auch: *Funktionsprozedur*) besteht aus einem *Funktionskopf* (auch *Signatur* genannt) und einem *Funktionsrumpf*. Der Kopf besteht im Wesentlichen aus dem *Namen*, den *Parametern* (oder *Argumenten*) und dem *Ergebnistyp* der Funktion. In Java

kann optional noch eine Anzahl von *Modifikatoren* (public, private, protected, static, final) davor stehen. Die folgenden Syntaxdiagramme sind aus http://www.infosun.fim.uni-passau.de/cl/passau/kuwi05/SyntaxDiagrams.pdf



In Groovy gibt es die Möglichkeit, durch Angabe von def den Ergebnistyp dynamisch bestimmen zu lassen. Der Rumpf der Funktion ist (in Java) ein *Anweisungsblock* bzw. (in Groovy) eine *Closure*. Ein Anweisungsblock ist ein in {…} durch Semikolon getrennte Liste von Anweisungen. In Groovy kann ein solcher Anweisungsblock auch einer Variablen zugewiesen werden, so dass er später aufgerufen werden kann (*closed block* oder *closure*). Ein solcher Anweisungsblock darf auch – genau wie im λ–Kalkül – formale Parameter enthalten.

```
def f(x) {3*(x**2) + 2*x + 5}
assert f(5) == 90
```

vollkommen gleichwertig dazu sind folgende Groovy-Anweisungen:

```
def g = {x -> 3*x**2+2*x+5}
assert g(5) == 90
```

Das bedeutet, der Term λ x. t wird in Groovy notiert als `{x -> t}`.
Natürlich kann eine Funktion mehrere Parameter unterschiedlichen Typs enthalten:

```
def gerade (int a,b, float x) {return a*x+b}
def parabel = { int a,b,c, float x -> a*x**2+b*x+c}
```

Rekursion

Der Wert einer Funktion ist das Ergebnis der letzen ausgeführten Anweisung. Falls innerhalb des Blocks eine Anweisung einen Ausdruck enthält, der den Namen der Funktion selbst enthält, sagt man, die Funktion ist *rekursiv*. Bei einer *einfachen Rekursion* gibt es nur einen einzigen rekursiven Aufruf (Beispiel: Fakultätsfunktion); falls dieser jeweils die letzte Aktion bei der Ausführung ist, sprechen wir von einer *Endrekursion* (*Tail-end-Rekursion*). Bei einer *Kaskadenrekursion* gibt es mehrere rekursive Aufrufe auf der gleichen Schachtelungstiefe

(Beispiel: Fibonacci). In einer *geschachtelten Rekursion* kommen rekursive Aufrufe innerhalb von rekursiven Aufrufen vor; Beispiel ist die *McCarthy'sche 91-er Funktion*:

```
f = {n -> (n>100)? (n-10) : f(f(n+11))}
g = {n -> (n>100)? (n-10) : 91}
assert (1..200).each {f(it) == g(it)}
```

Intuitiv lässt sich folgende Analogie herstellen:

Endrekursion – reguläre (Chomsky-3) Grammatik
einfache Rekursion – kontextfreie (Chomsky-2) Grammatik
Kaskadenrekursion – kontextsensitive (Chomsky-1) Grammatik
geschachtelte Rekursion – allgemeine (Chomsky-0) Grammatik

Diese Analogie gilt allerdings nur bedingt, weil sich bei Verfügbarkeit von Zuweisungen jede Funktion als Endrekursion darstellen lässt. Da sich Endrekursionen auf von-Neumann-Rechnern besonders effizient ausführen lassen (mit nur einer Schleife), war das Thema der „Entrekursivierung" lange Zeit von Bedeutung.

Die so genannte Ackermann-Péter-Funktion ist ein Beispiel für eine arithmetische Funktion, die sich nicht mit einfachen Mitteln entrekursivieren lässt:

ack(0,m) = m+1
ack(n+1,0) = ack(n,1)
ack(n+1,m+1) = ack(n, ack(n+1,m))

oder in Java/Groovy:

```
def ack(n,m) {(n==0)? m+1 : (m==0)? ack (n-1,1) : ack(n-1, ack(n,m-1))}
```

Hier sind die Werte der Funktion für verschiedene Eingabeparameter

	n=0	n=1	n=2	n=3
m=0	1	2	3	5
m=1	2	3	5	13
m=2	3	4	7	29
m=3	4	5	9	61
m=4	5	6	11	125
m=5	6	7	13	253
m=6	7	8	15	509
m=7	8	9	17	1021
m=8	9	10	19	2045
m=9	10	11	21	4093

Es ist eine interessante Übung, die Werte für $n \geq 4$ zu berechnen.

Aufrufmechanismen

Falls innerhalb eines λ-Terms ein Teilterm $(\ldots((\lambda\, x.\, t_1)\, t_2)\ldots)$ vorkommt, so kann auf diesen die β-Konversion angewendet werden. Im Falle mehrerer solcher Teilterme erlaubt es der λ-Kalkül, diese Regel an beliebiger Stelle anzuwenden. In einer Programmiersprache muss hierfür jedoch eine Reihenfolge festgelegt werden.

Beispiel: f= {x,y -> (x==y)? x : f(f(x,y),f(x,y))}

Wie erfolgt die Berechnung von f(0,1)?

Unter der *"leftmost-innermost"*-oder *normalen* Auswertungsregel versteht man die Regel, dass jeweils der am weitesten links stehende Ausdruck, der keinen weiteren rekursiven Aufruf mehr enthält, expandiert wird.

Beispiel: f(0,1)
= (0==1)? 0: f(f(0,1), f(0,1))
= f(f(0,1), f(0,1))
= f((0==1)? 0: f(f(0,1), f(0,1)), f(0,1))
= f(f(f(0,1), f(0,1)), f(0,1))
= f(f(f(f(0,1), f(0,1)), f(0,1)), f(0,1))
= …

Bei der *„leftmost-outermost"* oder *verzögerten* Ausführung *(„lazy evaluation")* wird jeweils der äußerste linke Aufruf expandiert:

Beispiel: f(0,1)
= (0==1)? 0: f(f(0,1), f(0,1))
= f(f(0,1), f(0,1))
= (f(0,1) == f(0,1))? f(0,1) : f(f(f(0,1),f(0,1)),f(f(0,1),f(0,1)))
= f(0,1) = …

Bei der „full substitution"-Regel werden alle Aufrufe gleichzeitig expandiert. Für g= {x,y -> (x==y)? 0 : g(g(x,y),g(x,y))} terminiert die verzögerte, nicht jedoch die normale Auswertung.

Java/Groovy verwenden wie die meisten anderen Programmiersprachen die normale Auswertungsregel:

```
fun = {x,y -> println ("Aufruf mit x: "+x+", y: "+y);
      (y==0)? y : fun (1, y-1) + fun (2, y-1) }
fun(0,2)
```

ergibt

```
Aufruf mit x: 0, y: 2
Aufruf mit x: 1, y: 1
Aufruf mit x: 1, y: 0
Aufruf mit x: 2, y: 0
Aufruf mit x: 2, y: 1
Aufruf mit x: 1, y: 0
Aufruf mit x: 2, y: 0
Result: 0
```

Das Church-Rosser-Theorem im λ–Kalkül besagt, dass die Relation →, mit $t_1 \rightarrow t_2$ g.d.w. t_2 durch β-Reduktion aus t_1 entstanden ist, die *Rauteneigenschaft* („*diamond property*") besitzt:

Für alle Terme x, y z gilt: falls x→y und x→z,
so existiert ein Term w mit y→w und z→w.

Daraus folgt unmittelbar, dass die Rauteneigenschaft auch für Ableitungsfolgen gilt:

Falls $x \rightarrow^* y$ und $x \rightarrow^* z$, so existiert w mit $y \rightarrow^* w$ und $z \rightarrow^* w$.

Daraus lässt sich folgern, dass wenn ein Term nicht mehr weiter reduziert werden kann, das Ergebnis eindeutig sein muss:

Falls $x \rightarrow^* y$ und $x \rightarrow^* z$ und y ist irreduzibel, so gilt $z \rightarrow^* y$

Also ist für terminierende Berechnungen die Frage der Reihenfolge letztlich nur für die Effizienz der Berechnung, nicht aber für das richtige Ergebnis wichtig.

Terminierungsbeweise

Um für eine rekursive Funktion zu beweisen, dass sie terminiert, müssen wir eine Abbildung der Parameter in die natürlichen Zahlen (oder, allgemeiner, in eine fundierte Ordnung) finden, so dass für jeden Aufruf die aufgerufenen Parameterwerte echt kleiner sind als die aufrufenden.
Beispiel:

```
fun = { x -> (x>=10)? x**2 : x * fun(x+1) }
```

Um zu zeigen, dass die Funktion `fun` terminiert, betrachten wir die Abbildung

f: $\mathbb{Z} \rightarrow \mathbb{N}_0$, f(x) = max(10 – x, 0).

Dann ist für $x \leq 10$ auch f(x+1) = 10 – (x+1) = 9 – x < 10 – x = f(x).
Also können wir folgern, dass **`fun(x)`** für alle x aus $\mathbb{Z}$ terminiert.

Formal ist das ein Schluss nach dem Schema der transfiniten Induktion, siehe Kap. 1.2:
Aussage: Für alle i und alle x gilt: Wenn f(x) = i, so terminiert **`fun(x)`**

(a) *Induktionsanfang:* Wenn f(x) = 0, so terminiert **`fun(x)`**

Beweis: Wenn f(x)=0, so ist $x \geq 10$, in diesem Fall terminiert **`fun`**

Induktionsvoraussetzung: Für alle x mit f(x) < i gilt dass **`fun(x)`** terminiert, *zeige*: Wenn f(x)=i, so terminiert fun(x). *Beweis:* **`fun(x)`** ruft **`fun(x+1)`**; oben gezeigt: f(x+1)<f(x), also f(x+1)<i, also terminiert **`fun(x+1)`**, also auch **`fun(x)`**.