frames und Umgebungen

Neben den für SCHEME wichtigen zentralen Begriffen bzw. Strukturen, *funktionale Programmierung* , *rekursive Programmierung* und *lambda - Ausdrücke* gibt es einen weiteren, der zum Verständnis vieler Anwendungen unverzichtbar ist. Er bildet, wenn man es genauer betrachtet, sogar die Grundlage einiger wesentlicher Elemente und Möglichkeiten dieser Sprache: *frames und Umgebungen .*
Dabei will ich Ihnen zeigen, dass davon ein natürlicher Weg, ohne Brüche und ohne große Verständnissprünge, zur Objekt - orientierten Programmierung führt.

Der Begriff ***frame*** läßt sich schwer Wort-zu-Wort ins Deutsche übersetzen. Wörtlich heißt er Rahmen., das gibt aber im deutschen Sprachgebrauch nicht den Sinn an. Wir werden also bei diesem Wort bleiben. Seine Bedeutung läßt sich selbstverständlich beschreiben :

Ein frame legt den Bereich fest, der unmittelbar zur Auswertungsstufe einer Funktion gehört. Das geschieht im einfachsten Fall durch die Bindung mit einem lambda-Ausdruck.

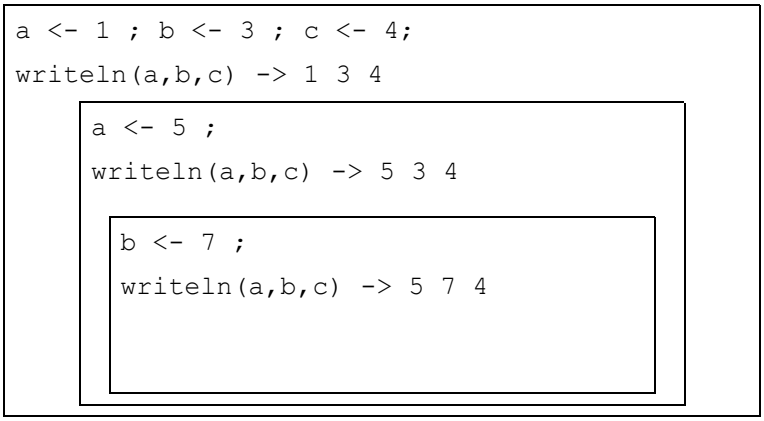
Die eigentliche sprachlich festgelegte Definition einer einfachen Funktion hätte nämlich die Form :

```
(define
     Funktionsname
(lambda
          (Parameterliste)
          Auswertungsteil)
```

Bei der Schachtelung von Funktionsaufrufen, also auch von Selbstaufrufen bei der Rekursion, entstehen jeweils neue frames. Diese frames stehen aber nicht beziehungslos nebeneinander. Zu jedem frame gehört genau eine ***Umgebung***, in die er eingebettet ist.

Grundsätzlich ist das Problem auch in anderen Programmiersprachen bekannt. Bei der Unterscheidung von globalen Variablen, lokalen Variablen und Parametern muss man sich Gedanken darüber machen, welche Deklaration in welcher Prozedur Gültigkeit hat. Dabei ist auch das Überdecken von Bindungen bekannt.

Das läßt sich schematisch so für viele Sprachen darstellen :



Unsere im Bild dargestellten Rahmen kennzeichnen jeweils den frame. Er bildet mit seinen Vorgängern in der Aufrufschachtelung eine *Umgebung*, in der die Variablen im innersten frame schließlich a zu 5, b zu 7 und c zu 4 auswerten. Eine spätere Bindung

überdeckt also eine frühere[1].

Die Art der Variablenbindung ist in SCHEME statisch (*lexical binding*) . Das hat zur Folge, dass man die Art der Bindung direkt aus dem Programmtext ablesen kann. Die oberste Stufe der Umgebungen in PCSCHEME heißt user - global - environment, in der alle Systemfunktionen gebunden sind, die darunter ist das user - initial -environment, in dem der Benutzer die Variablen (bei Scheme genau so wie die Funktionen) erzeugt, Mit (define a 5) wird z.B. die Variable a angelegt und an den Wert 5 gebunden.
Die Formen zur Bindung sind in PCSCHEME außerdem: DO , LAMBDA , LET , LET* , LETREC und REC.
DrScheme schafft zu jedem Programm nicht nur ein neues Fenster, sondern auch eine neue, eigene Umgebung. Die Definitionen eines Fensters beeinflussen also nicht Umgebungen eines anderen Fensters. Außerdem wird nach jedem „Ausführen“, also Auswerten der Definitionen im Editorfenster die Benutzerumgebung neu angelegt.
Ein paar einfache...

Beispiele zu Bindungen :

Aufruf (f-1 Zahl-Parameter Zahl-Parameter)

```
(define
   (f-1 a b)
(display "Berechnet die Summe der Parameter a + b = ")
   (+ a b))
```

Dies zweite einfache Beispiel zeigt die Zugehörigkeit der Variablen zu ihren Frames . a und b sind zwar im UIE gebunden (also global) , beim Aufruf von f-1a wird aber in der dazu gehörenden Umgebung die Bindung von b überdeckt , während die von a erhalten bleibt .
Aufruf (f-2 Zahl-Parameter)

```
(define a 5)
(define b 19)

(define
   (f-2 b)
   (+ a b))
```

Nächste Variante : mit let und read :
Aufruf (f-3) , also ohne Parameter.

kommentierte Eingabe

```
(define
   (f-3)
   (let
      ((a (begin (display "Gib a ein ") (read)))       ;
       (b (begin (display "Gib a ein ") (read))))
      (+ a b)))
```

Noch eine Variante : als Parameter in zwei Stufen ! (Aufruf ???)

```
(define
   (f-4 a)
   (lambda
      (b)
      (+ a b)))
```

1 Ein vergleichbares Beispiel in JAVA ist das Überdecken einer Objektvariablen durch eine lokal in einer Methode neu definierte Variable.

oder auch mit demselben Ergebnis :

```
(define
   f-5
   (lambda
      (a)
     (lambda
        (b)
        (+ a b)))
```

(Erläuterung s.u.)

Letzte Variante : Natürlich kann man bei den lokalen Bindungen auf globale Variablen zugreifen. Machen Sie das besser nicht so !
Aufruf : (f-6)

```
(define
   (f-6)
   (let
      ((a x)
       (b y))
      (+ a b)))

(define x 3)
(define y 4)
```

Das, was auf den ersten Blick bei der vierten und fünften Variante identisch mit der ersten Variante zu sein scheint, stellt sich bei genauerer Betrachtung als ein qualitativ neuer Schritt heraus. Im ersten Schritt wird hier nämlich eine Funktion mit einem Parameter definiert. Der Wert der Auswertung ist hier also keine Zahl, sondern eine Funktion ! Diese kann nun im zweiten Schritt selbst ausgewertet werden, wobei ihr dabei dann ein weiterer Parameter übergeben werden muss und sie liefert nun erst in dieser zweiten Stufe eine Zahl als Wert.

Funktionsgeneratoren

Die mehrschichtige Auswertung von lambda - Ausdrücken macht es in SCHEME möglich, Funktionsgeneratoren zu schaffen.
Das angegebene Beispiel eignet sich also nicht nur zur Betrachtung der erzeugten Frames, sondern es zeigt auch, wie hier ein Funktionsgenerator entsteht, mit dessen Hilfe eine Funktionenschaar erzeugt werden kann. Wir wollen uns diese Auswertung einmal genauer ansehen.

```
(define
   erzeugt-n-Addierer
   (lambda
      (n)   ; n ist freier Parameter des ersten Frames
       (bkpt "1. Aufruf" ())                    nur in PCSCHEME !
      (lambda
         (m)  ; m ...des 0. Frames
           (bkpt "2. Aufruf" ())
         (+ m n))))

(define 2+
   (erzeugt-n-Addierer 2))
```

Eine Betrachtung mit breakpoint zeigt : m und n gehören zu verschiedenen frames von der durch 2+ definierten Umgebung.
Die Definition von **`erzeugt-n-Addierer`** wird beim compile ausgewertet. Ihr Wert ist das, was hinter dem ersten lambda kommt, also eine Reihung von
`(bkpt "1. Aufruf" ())` das System unterbricht die Auswertung, lassen wir es weiterarbeiten, dann wird der Wert des zweiten Ausdrucks bestimmt. Dieser Wert besteht -abgesehen vom weiteren bkpt- aus

```
(lambda (m)
         (+ m n))
```

und ***ist daher eine Funktion*** !
Die Definition von 2+ wird ebenfalls beim compile ausgewertet und ruft ihrerseits zunächst die gerade vorher definierte Funktion **`erzeugt-n-Addierer`** mit dem Parameter 2 auf.
Sie erzeugt daher eine Funktion, nämlich die Funktion

```
(lambda (m)
         (+ m 2))
```

bei welcher der ursprüngliche Parameter n nun fest an 2 gebunden ist und deren einziger eigener Parameter nun m ist.
Im vorigen Beispielprogramm wurde also zusätzlich gezeigt, wie sich durch Definition von Variablen vor der Funktionsdefinition mit lambda Variable schaffen lassen, die nicht dem UIE, aber auch nicht dem frame der Auswertung zugeordnet sind, sondern an die Funktion selbst gebunden sind.
Ruft man nun diese Funktion 2+ auf, berechnet erst sie einen konkreten Zahlenwert, nämlich zu jedem eingegebenen Wert den um 2 größeren.
Da sich nun aber ebenso 3+ oder 7+ oder meinetwegen auch pi+ erzeugen läßt, ist die erste Funktion **`erzeugt-n-Addierer`** also ein Funktionsgenerator !

Weitere Beispiele zu Bindungen :

Hier werden die Arten der Bindung von Variablen bei geschachtelten Aufrufen demonstriert. Als Beispiel wird die Fakultät einer natürlichen Zahl gewählt.

---

1. Beispiel Fakultät rekursiv, also typisch LISP

```
(define
   (fac-1 n)
   (if
      (zero? n)
      1
      (* n (fac-1 (sub1 n)))))
```

---

2. Beispiel Fakultät endrekursiv, (also wie iterativ) mit Hilfsfunktion :

```
(define fac-2
  (lambda (n)
    (fac-hilf 1 n)))

(define fac-hilf
  (lambda
     (akku aktuell)
     (if
       (zero? aktuell)
        akku
       (fac-hilf (* aktuell akku) (sub1 aktuell)))))
```

Es ist typisch SCHEME, dass bei endrekursiven Versionen kein Müll auf dem Stack erzeugt wird.
Es gibt noch zwei Beispiele Fakultät mit set! , einmal mit Zuweisung zu Variablen. Beide folgen noch !

Hier werden die Arten der Bindung von Variablen bei Aufrufen von verschiedenen Funktionen demonstriert.

```
(define
   (ist-gerade n)
   (if
      (zero? n)
      #t
      (if
         (>? n 0)
         (ist-ungerade (sub1 n))
         #f)))
```

```
(define
   (ist-ungerade n)
   (if
      (=? n 1)
      #t
      (if
         (>? n 1)
         (ist-gerade (sub1 n))
         #f)))
```

Ergebnis :Die Bindungen erfolgen völlig separat beim Aufruf. Eine Veränderung in der einen Prozedur wirkt sich nur durch die Übergabe auf die andere aus.

Dazu im folgenden Block verschiedene Benennungen und die Erweiterung um globale Variable mit gleicher Benennung :

```
(define u 17)
(define g 12)

(define
   (ist-g-2 g)
   (if
      (zero? g)
      #t
      (if
         (>? g 0)
         (ist-u-2 (sub1 g))
         #f)))
```

```
(define
   (ist-u-2 u)
   (if
      (=? u 1)
      #t
      (ist-g-2 (sub1 u))))
```

Im folgenden Block wird nun auf ***eine*** globale Variable zugegriffen :

```
(define die-Zahl 0)

(define
   (gerade-oder-ungerade)
   (set! die-Zahl (read))
   (writeln "Die Zahl " die-Zahl " ist " (if (ist-g-3) "gerade" "ungerade")))

(define
   (ist-g-3)
   (if
      (zero? die-Zahl)
      #t
      (if
         (>? die-Zahl 0)
         (begin
            (set! die-zahl (sub1 die-Zahl))
            (ist-u-3))
         #f)))

(define
   (ist-u-3)
   (if
      (=? die-Zahl 1)
      #t
      (if
         (>? die-Zahl 1)
         (begin
            (set! die-zahl (sub1 die-Zahl))
            (ist-g-3))
```

```
         #f)))
```

Im folgenden Block wird mit Bindung an die Funktionen gearbeitet :

```
(define
   (gerade-oder-ungerade-2)
   (let
      ((die-Zahl (read)))
      (writeln
         "Die Zahl "
         die-Zahl
         " ist "
         (if (ist-g-3) "gerade" "ungerade"))))
```

Dies geht z.T. scheinbar gut, wenn vorher schon mit den anderen Varianten gearbeitet wurde. Dann ist die-Zahl nämlich noch definiert. Ohne diese "Hilfe" geht es nicht, zudem sind die Ergebnisse nur z.T. richtig.
(Welcher Teil ?)

Nun noch die fehlenden Varianten von oben
3. Beispiel Fakultaet mit set!

Aufruf (fac-3 Zahl-Parameter)

```
(define
   fac-3
   (let
      ((Hilfs-Variable 1))
      (define                      eine lokal definierte Hilfsfunktion
         (Hilfs-fac n)
         (if
            (< n 1)
            Hilfs-Variable
            (begin
               (bkpt "in Hilfs-fac" ())                    nur in PCSCHEME
               (set! hilfs-Variable (* n Hilfs-Variable))
               (Hilfs-fac (sub1 n)))))
      (lambda
         (m)
         (Hilfs-fac m))))
```

Aber, ach wie böse : Beim zweiten Aufruf geht es schief ! Das Problem ist : Hilfs-Variable hat den ursprünglichen Wert vergessen und sich den aus der letzten Berechnung gemerkt.

Also besser anders herum !
Beispiel Fakultaet mit set!
Aufruf (fac-4 Zahl-Parameter)

```
(define
   fac-4
   (let
      ((Hilfs-Variable "sinnlos"))
      (define
         (Hilfs-fac n)
         (if
            (< n 1)
            Hilfs-Variable
            (begin
               (bkpt "in Hilfs-fac" ())
               (set! hilfs-Variable (* n Hilfs-Variable))
               (Hilfs-fac (sub1 n)))))
```

```
      (lambda
         (m)
         (set! Hilfs-Variable 1)
         (Hilfs-fac m))))

(writeln "Warum dann ueberhaupt so umstaendlich ?")
```

Funktionen mit Zustand

Der Kern der Betrachtungen im letzten Teil ist, dass man auch bei Scheme Funktionen mit Zustand schaffen kann, indem man im Kopf des Definitionsteils der Funktion vor dem lambda (und dies geht nur mit lambda!) Variablen bindet. Ihre Werte stehen im Funktionskörper zur Verfügung und ihre Werte behalten so lange Gültigkeit wie die Funktion selbst. Dieser Wert ist daher eine Eigenschaft der Funktion, im Sinne der Objekt – Orientierung also ihr Attribut.