Gliederung

Die Scheme-Philosophie

G. Brados:

„Programming languages should be
designed not by piling feature on top of feature,
but by removing the weakness and
restrictions that make additional features appear
necessary.“

Scheme

Eine Einführung

Scheme ist aus den unterschiedlichsten Zutaten, hier die Wichtigsten:

„Algol“ 1958
(imperative Programmierung)

- Blockstruktur
- Lexical Scoping

„LisP“ 1959
(funktionale Programmierung)

- Programme und Daten haben die selbe Form
- Programmausführung durch Auswertung

1. Funktionen

1.1 Der Funktionen-Aufruf:

Die allgemeine Form eines Funktionen-Aufrufs

ist die Schreibweise:

$$(p\ arg_1\ arg_2\ ...\ arg_n)$$

Auswertung der Argumente erfolgt standardmäßig
immer von links nach rechts,

Funktionsverknüpfungen immer von rechts nach links
(Rechtsklammerung)

```
> (+ 2 3)
5

> (/ 2 (+ 2 3))
2/5

> (/ (+ 2 3) 2)
2 1/2
```

Higher-Order-Funktions

Da alle Funktionen in Scheme *First-Class* sind,
können sie auch als Argument verwendet,
oder als Ergebnis zurückgegeben werden.

```
> ((p 2) 4 3)   ;
```

sein nun der Aufruf (p 2) im Endwert die Addition (+),
so ergäbe der gesammte Ausdruck:

```
7
```

1.2 Special Forms

Gründe für die Notwendigkeit von Special Forms:

Nicht alle wünschenswerten Funktionen können mit
standart Syntax und standart Auswertung
ausreichend gut dargestellt werden

Es ergibt sich die Nötigkeit, dass Prozesse nicht immer
in der selben Auswertungsreíhenfolge berechnet
werden können.

Nun ein Beispiel anhand der IF-Funktion:

> (if (= 3 5) (print „then“) (print „else“)

eine Ersatz-Funktion nif wäre leider nicht in der Lage
zwischen den Ausgaben „then“ und „else“ zu entscheiden,
ohne wieder eine if-Funktion zu nutzen.

„then“ „else“

Wenn eine zu verwendende Befehlskette unverhältnismäßig
viel Code benötigt

Wenn eine Funktion sehr häufig verwendet wird.

Die sind die wichtigsten Befehle des SchemeText-Scheme,
der einfachsten aller Scheme-Implementationen.

Bei anderen Implementationen wurden für spezielle
Anwendungen noch zusätzliche Funktionen eingebettet.
z.B. PLT-Scheme kann grafische Oberflächen

Anwendung:

Diese Formen haben spezielle Syntax,
auf die bei Verwendung
explizit geachtet werden muss.

Beispiel: (define Variable Ausdruck)

> (define HW „Hello World!“)
> hw
„Hello World!“

> (if #t 1 2)
1

> (zero? 5)

> (if (zero? 2) 5 (* 3 2))

> (define true #t)
> (define false #f)
> (if (if true false true) (if false 1 2) 3)

2. Typen

2.1 Zeichen:

Buchstaben, oder Literale,
werden als solche mit #\ gekennzeichnet.

So zum Beispiel, #\a, #\?, oder #\space.

Man betrachte:

```
> (char? #\$)
```

```
> (char=? #\$ #\space)
```

```
> (char<? #\a #\b)
```

```
> (define L „Pro-Seminar“)
> (string-length L)
 11
```

```
> (string->list  L)
 (#\P #\r #\o #\- #\S #\e #\m #\i #\n #\a #\r)
```

2.2 Symbole:

Ein Symbol ist ein Name, der sich selbst als Wert hat.
Und nicht wie bei Variablen für einen anderen Wert steht.

(quote exp) oder kurz 'exp

```
> (define x 4)
> x
 4

> 'x
 x

> x
 4
```

```
> (define y 'Uni)
> (eq? y (quote Uni))
```

```
> (eq? y 'y)
```

2.3 Listen:

Scheme arbeitet mit Listen als grundlegende Struktur zur Darstellung von Daten und Programmen, ob eine Liste als Liste an sich oder als Programm verarbeitet wird, hängt von der Interpretation ab.

Verwendung:

(list arg_1 arg_2 ... arg_n)	;	erschafft neue Liste
(cons $list_1$ $list_2$)	;	fügt $list_1$ als 1. Element in $list_2$ ein

```
> (define a (list 'c 'd))
> a
 (c d)

> (cons 'a a)


> (cons '(ab) a)
```

(append $list_1$ $list_2$ ... $list_n$) ; setzt Listen zusammen

(car list) ; liefert erstes Element der Liste

(cdr list) ; liefert list ohne Ihr erstes Element = Rest

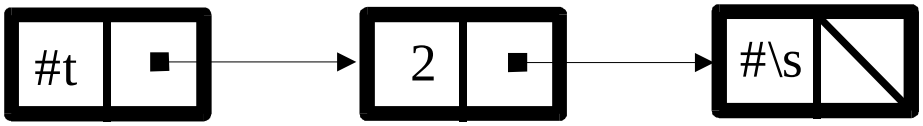
```
>(define l '((a (b)) (b) (c (b d))))
> l
 ((a (b)) (b) (c (b d)))

> (car l)


> (cdr l)


> (cadr l)
```

Dies bietet uns schon tiefe Einblicke in die Verarbeitung einer Liste in Scheme.

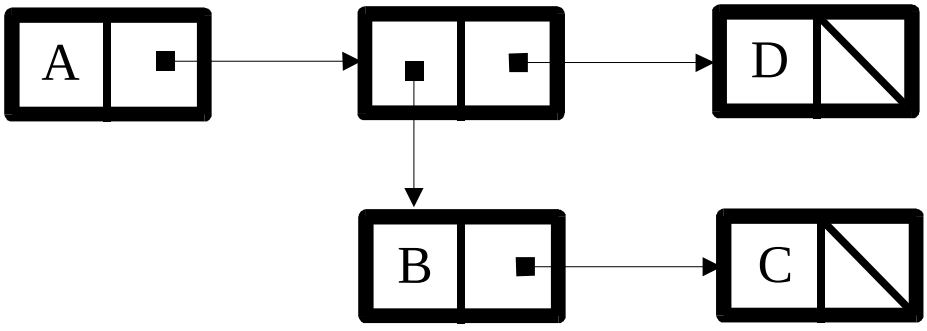
Der allgemeine Aufbau der Datenstruktur erfolgt in Paaren, diese Paare enthalten entweder:

Werte,
Zeiger auf ein neues Paar,
oder die „Leere Liste“ ().

(#t 2 #\s)



(A (B C) D)



2.4 Funktionen:

Lambda

Die „Special Form“

(lambda *parameter body*)

ermöglicht dem Programmierer neue Funktionen
selbst zu erstellen.

- „*parameter*“ ist eine möglicher Weise auch
leere Liste von Variablen

- „*body*“ kann jede Art von Ausdruck sein

Die im Lambda-Ausdruck verwendeten Variablen
sind an die Funktion „gebundene“ Variablen.

Das heißt sie sind von außerhalb der Funktion
nicht zugreifbar, sie gelten nur in ihrem „scope“.

Da in Scheme kein Unterschied zwischen
einer Funktion und einer Variablen besteht,
schafft man durch
Schachtelung „anonyme Funktionen“.

Verwendung:

anonymer Funktionsaufruf:

```
> ((lambda (n) (+ n 3)) 6)
 9
```

als Argument verwendet:

```
> (list (lambda (n) (+ n 3)) 6)
 (#<procedure:53:9> 6)
```

benannter Funktionsaufruf:

```
> (define add3 (lambda (n) (+ n 3)))
> (add3 9)
 12
```

map und andmap

(map *funktion list*)

Erschafft aus den Funktionswerten der ursprünglichen *list*-Einträge eine neue Liste.

Die Länge der Liste ist hierbei egal, nur die Funktion muss einargumentig sein.

```
> (define map (lambda (Fkt Liste)
        (if (null? Liste)
            '()
           (cons (Fkt (car Liste))
               (map Fkt (cdr Liste))))))
```

```
> (map list '(5 4 3))
 ((5) (4) (3))
```

> > (map + '(5 4 3))

(andmap *funktion list*)

Prüft alle Funktionswerte der *list*-Einträge
auf Ihre Wahrheit.

Liefert bei Richtigkeit **aller** Funktionswerte #t zurück,
sonst #f.

```
>(map null? '((a) () () (3)))
 (#f #t #t #f)

>(andmap null? '((a) () () (3)))

> (map (lambda (f) (f '(a b c d)))
       (list car cdr cadr cddr caddr))
 (a (b c d) b (c d) c)
```

Beides map und andmap
sind Paradebeispiele einer Higher-Order-Funktion!

3. Syntax

3.1 Der „scope“ von Bindungen:

Definitionen (z.B. define...) bilden *Top-Level* Bindungen,
diese besitzen ihre Gültigkeit über das gesammte Programm.

Der Speicher wird wärend des gesammten
Programms benötigt.

Lambda-Ausdrücke die Funktionen darstellen binden nur *lokal*,
die Bindung erstreckt sich nur über den „body“.

Der Speicher wird nach der Berechnung
sofort wieder frei gegeben.

3.2 let und letrec:

(let (var_1 exp_1)
(var_2 exp_2)
...
(var_n exp_n)
body)

In dieser Form werden erst alle Ausdrücke als Werte
an die Variablen gebunden und schließlich
diese bezüglich dem *body* ausgewertet.

Die Bindung der Variablen var_1 bis var_n
sind hier bezüglich *body*!

(letrec (var_1 exp_1)

(var_2 exp_2)

...

(var_n exp_n)

body)

Die Bindung der Variablen var_1 bis var_n sind hier bezüglich dem kompletten *letrec*-Ausdruck*!*

Beide Bindungen sind dennoch lokal!

Vielen Dank für Ihre Aufmerksamkeit.