Gliederung

Vorwort

1. Funktionen

    1.1 Funktionen-Aufruf
    1.2 Special Forms

2. Typen

    2.1 Zeichen
    2.2 Symbole
    2.3 Listen
    2.4 Funktionen

3. Syntax

    3.1 „Scope“ von Bindungen
    3.2 let und letrec

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 wurde 1975 von Guy Steele und Gerald Sussman am MIT in Bosten entwickelt. Ziel war es, eine Programmiersprache zu entwickeln, welche Studenten eine völlig neue Sichtweise des Programmierens zugänglich machen soll. Diese Sprache sollte keinen Einschränkungen bezüglich der Möglichkeiten unterliegen, dennoch aber mathematisch genau sein und klar in ihrer Struktur.

Als Grundlagen wurden **Algol68** und **LisP** verwendet.

Algol, der Urvater der imperativen Programmierung, wurde wegen seiner Auswertbarkeit gewählt. Ein imperatives Programm ist immer ein wertfähiger Ausdruck. Auch die Möglichkeit eine Variable in Ihrer Gültigkeitsdauer zu definieren aus der imperativen Programmierung.

LisP hingegen bietet durch seinen Listen-Aufbau
die Freiheit, nicht mehr zwischen Anweisungen und Ausdrücken unterscheiden zu müssen. Aber die Überprüfbarkeit des Codes ist hierdurch sehr eingeschränkt, weshalb man nur partiell auswertet.

Weiterhin floss die Überlegung ein, möglichst wenig definierte Befehle vorzugeben, und dennoch eine maximal Funktionalität zu schaffen.

„Algol“ 1958

(imperative Programmierung)

- Blockstruktur (C, Pascal, Java)
- Lexical Scoping (Variablenwerte werden trotz eines neuen „scopes“ nicht überschrieben)

„LisP“ 1959

(funktionale Programmierung)

- Programme und Daten haben dieselbe Form (Listen)
- Programmausführung durch Auswertung (partielle Auswertung)

Man erhielt unweigerlich eine „Metasprache“.

Eine Metasprache ist eine Sprache, welche die kompletten Befehle einer zweiten Sprache, ohne diese zu verfälschen, in ihren eigenen Code überführt, darstellt und auswertet.

Somit ist es möglich, mit Scheme Compiler und Interpreter zu schreiben. Z.B. können alle Codes, wie ML (SML, XML), C, C++, und auch Java, in Scheme äquivalent ersetzt werden.

1. Funktionen

1.1 Funktionen-Aufruf:

Die allgemeine Form eines Funktionen-Aufrufs ist die Schreibweise:

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

Man kann sehen, dass der Aufruf der Funktion (Function-Call) mit den Argumenten zusammen in der Klammer steht. Es heißt auch, die Argumente sind an p gebunden.

Die Auswertung der Argumente erfolgt standardmäßig immer von links nach rechts, im Gegensatz zu Funktionsverknüpfungen, die immer von rechts nach links abgearbeitet werden (Rechtsklammerung).

Beispiele:

```
> (+ 2 3)
5

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

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

Higher-Order-Functions

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

```
> ((p 2) 4 3)   ;
        sei nun der Aufruf (p 2) im Endwert die Addition (+):
  7
```

1.2 Special Forms:

Gründe für die Notwendigkeit von Special Forms:

1. Nicht alle gewünschen Funktionen können mit Standard-Syntax und Standard-Auswertungsreihenfolge ausreichend gut dargestellt werden.

2. Es ergibt sich die Notwendigkeit, dass Prozesse nicht immer in derselben Auswertungsreíhenfolge berechnet werden können. (z. B. If-Funktion)

   Nun ein Beispiel anhand der IF-Funktion:

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

   eine Ersatz-Funktion nif wäre nicht in der Lage, zwischen den Ausgaben „then“ und „else“ zu unterscheiden, ohne wieder eine if-Funktion nutzen zu müssen, Also gibt sie beide Möglichkeiten zurück und hat somit nicht „entschieden“.

   „then“ „else“

3. Wenn eine zu verwendende Befehlskette unverhältnismäßig viel Code benötigt, wird zur Übersichtlichkeit und Reduktion des Programmcodes eine „Special Form“ eingeführt. Dies dient der Übersichtlichkeit des Programmcodes und der Spezialisierung der Sprache auf ihr Einsatzgebiet.

4. Wenn eine Funktion sehr häufig verwendet wird, kann es sein, dass ein neuer Befehl (Befehlssatz) extra für diese Anwendung nachträglich hinzugefügt wird Dies ist einer schnelleren Programmierung und einem kurzeren Programmcode zuträglich.

Folgende Liste zeigt die wichtigsten Befehle des SchemeText-Scheme, der einfachsten aller Scheme-Implementationen.

Die Anzahl der Befehle ist sehr übersichtlich und die „Special Forms“ sind logisch, d.h. zu „char?“ gibt es natürlich auch „number?“, oder „procedure?“.
Auch Vergleiche wurden sehr eingängig formuliert; z.B. „number>?“

Bei anderen Implementationen wurden für spezielle Anwendungen noch zusätzliche Funktionen eingebettet.
Zum Beispiel kann PLT-Scheme grafische Oberflächen erstellen.

Anwendung:

Diese Formen haben spezielle Syntax, auf die bei Verwendung explizit geachtet werden muss. Meist erschließt sich die Auswertungsreihenfolge logisch aus der Verwendung des Ausdrucks. Die Anzahl der benötigten Argumente in der Funktion sind entweder frei, oder bestimmt (Zum Beispiel benötigt „if“ immer drei Argumente: Das Erste wird zum Testen verwendet. Aufgrund des Testergebnisses wird dann eines der beiden anderen Argumente ausgewählt, welches zurückgegeben wird.)
Argumente, Variablen und Ausdrücke werden immer mit Leerzeichen getrennt!

Beispiel:

(define Variable Ausdruck)

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


> (if #t 1 2)
 1


> (zero? 5)
  #f


> (if (zero? 2) 5 (* 3 2))
  6


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

2. Typen

2.1 Zeichen:

Buchstaben, oder Literale, werden als solche mit #\ gekennzeichnet.
So zum Beispiel, #\a, #\?, oder #\space.

Man betrachte:

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

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

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

> (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, im Gegensatz zu Variablen, die für einen anderen Wert stehen. Oft wird eine Funktion für eine spätere Berechnung als Symbol verarbeitet und erst am Schluss evaluiert.

(quote exp) oder kurz 'exp

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

> 'x
 x

> x
 4
```

```
> (define y 'Uni)
> (eq? y (quote Uni))
   #t

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

2.3 Listen:

Scheme arbeitet mit Listen als grundlegende Struktur zur Darstellung von Daten und Programmen. Ob eine Liste als Liste (z.B. Symbol) an sich, oder als Programm verarbeitet wird, hängt von der Interpretation ab.

Verwendung:

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

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

> (cons 'a a)
 (a c d)

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

(append $list_1$ $list_2$ ... $list_n$) ; setzt Listen in neuer Liste zusammen
(car list) ; liefert erstes Element der Liste zurück

(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)
  (a (b))

> (cdr l)
  ((b) (c (b d)))

> (cadr l)
  (b)
```

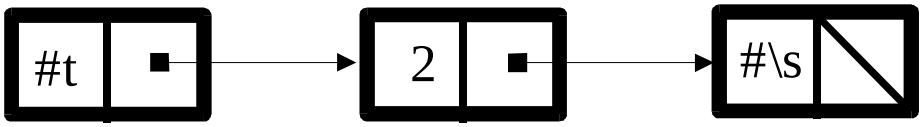
Dies bietet uns schon tiefe Einblicke in die Verarbeitung einer Liste in Scheme.
So zum Beispiel bildet „cons“ immer eine „geschachtelte“ Liste.

Dies lässt Rückschlüsse auf die Datenstruktur zu.
Der allgemeine Aufbau von Listen erfolgt in Paaren. Diese werden so weit geschachtelt bis das Paar mit der „leeren Liste“ erreicht ist.

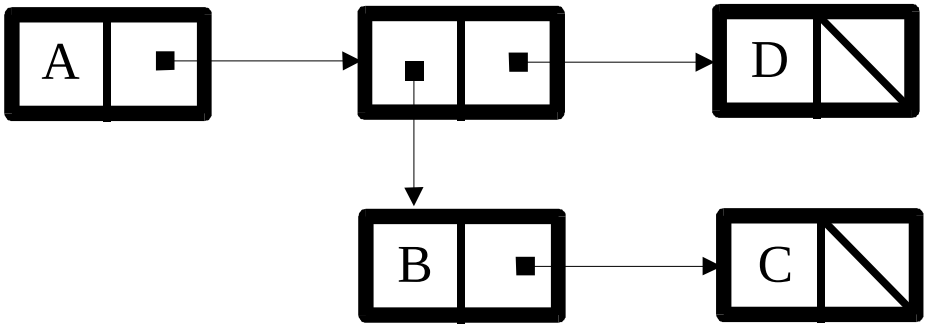
Diese Paare enthalten entweder:

- Werte,
- Zeiger auf ein neues Paar,
- die „leere Liste“ ().

(#t 2 #\s)



(A (B C) D)



Mit dieser Struktur der Datenablage (aus LisP entliehen) zeigt sich, warum kein Unterschied in der Ablage, und somit der Abarbeitung zwischen Listen und Funktionenaufrufen in imperativen Programmiersprachen, besteht.

2.4 Funktionen:

Lambda

Die „Special Form“

(lambda *parameter body*)

ermöglicht dem Programmierer, neue Funktionen selbst zu erstellen.
Das hiermit verwirklichte Lambda-Kalkül bietet eine annähernd unbeschränkte Menge an funktionalen Möglichkeiten zur Datenverarbeitung.

- „*parameter*“ ist eine möglicherweise auch leere Liste von Variablen
- „*body*“ kann jede Art von Ausdruck sein

Die im Lambda-Ausdruck verwendeten Variablen sind an die Funktion „gebundene“ Variablen. Ihnen können innerhalb der Funktion neue Werte zugewiesen werden. Diese sind aber immer „lokal“. Das heißt, man kann ihre vorübergehenden Werte außerhalb der Funktion nicht abrufen.
Sie gelten nur innerhalb ihres „scope“.

Da in Scheme kein Unterschied zwischen einer Funktion, einem Listenelement und einer Variablen besteht, schafft man durch Schachtelung unter Umständen „anonyme Funktionen“. Diese verändern ihre Varablen bezüglich der Werte nur lokal und sind nichtmehr im Rest-Code aufrufbar, was bei mehreren Programmierern an einem Projekt zu einer Vereinfachung führt.

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*)

Der Befehl „map“ erschafft aus den Funktionswerten der ursprünglichen *list*-Einträge eine neue Liste und gibt diese zurück. Die Länge der Liste ist hierbei egal, nur die Funktion muss einargumentig sein.

Zur Demonstration definieren wir uns unsere map-Funktion selbst, indem wir die Möglichkeiten des Lambda-Kalküls zuhilfe nehmen:

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

Hier zeigt sich schön, das Scheme auch als Interpreter für sich selbst eingesetzt werden kann. Dies ist im Einsatz nur bedingt sinnvoll, aber dennoch möglich.

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

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

(andmap *funktion list*)

Der Befehl „andmap“ prüft alle Funktionswerte der *list*-Einträge auf ihre Wahrheit und liefert bei Richtigkeit **aller** Funktionswerte #t zurück, sonst #f. Wird als Funktionswert eine Liste oder Funktion ermittelt, liefert andmap #f.

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


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

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

Sowohl „map“ als auch „andmap“ sind Paradebeispiele einer Higher-Order-Funktion, sie im Fall (map null? '((a) () () (3)))
(#f #t #t #f)
Ist „map“ eine Funktion höherer Ordnung und „(null? '((a) () () (3)))“ nicht.

3. Syntax

3.1 „Scope“ von Bindungen:

Definitionen (z.B. define...) bilden ***Top-Level***-Bindungen. Diese behalten ihre Gültigkeit über das gesamte Programm.
Es kann jederzeit auf diese Definitionen zugegriffen werden und sie können an einer anderen Stelle des Codes wieder Verwendung finden.

Das ist aber mit Nebeneffekten verbunden:

- Der Speicher wird während des gesamten Programms benötigt.
- Es ist schwer, definierte Variablen eines Codes, der von verschiedenen Programmierern erstellt wurde, nicht aus Versehen zu verwenden.

Eine ausschließlich globale (top-level)-Bindung aller Variablen eines Codes, bringt schnell einen Bezeichnungsnotstand und eine immense Menge an benötigtem Speicher mit sich. Die Informationen werden in einem „stack“ abgelegt und liegen dort immer vor. Das heißt, der „stack“ wächst bei einem komplizierten Code ins Gigantische.

Lambda-Ausdrücke, und die meisten „special forms“, binden nur ***lokal;***
die Bindung erstreckt sich nur über den „body“ und ein fehlerhafter Zugriff auf die Variable ist ausgeschlossen.
Der Speicher kann nach der Berechnung sofort wieder frei gegeben werden.
Dies löst auch das Problem der Fehlreferenzierung. Da der „Stack“ nur kurzzeitig belegt wird und nach der Berechnung sofort wieder freigegeben wird, ist kein weiterer Zugriff auf diese Information mehr möglich.
Wird sie später erneut verwendet, also auch wieder lokal gebunden, wird sie neu im „stack“ angelegt.

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 optimiert, das heißt so weit wie möglich ausgewertet und vereinfacht und dann als Wert an die Variablen gebunden. Diese werden schließlich bezüglich des „body“ ausgewertet.

Die Bindung der Variablen var_1 bis var_n ist hier bezüglich „body“, das heißt var_1 bis var_n werden erst bei der Ausführung der Funktion „body“ mit den optimierten Ausdrücken exp_1 bis exp_n belegt!

Diese Funktion ist für manche Anwendungen nicht ausreichend, da eine rekursive Funktion exp_n , die Variable var_n nicht aufrufen kann.

Es wurde „letrec“ geschaffen.

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

Die Bindung der Variablen var_1 bis var_n ist hier bezüglich dem kompletten „letrec“-Ausdruck!
Die Variablen werden vor der ersten Optimierung eines der Ausdrücke angelegt.
Dies ermöglicht uns, alle Variablen in der Funktion exp_n zu verwenden,
was zum Beispiel bei rekursiven Funktionen benötigt wird.

Beide Bindungen sind dennoch lokal!
Der „stack“ ist nach der Abarbeitung von letrec wieder von var_1 ... var_n geleert.