Scheme Grundlagen

(Version 1.1)

Zahlen

Scheme kennt die natürlichen Zahlen `0`, `1`, `2`, ... , und Zahlen mit Dezimalpunkt etwa `3.14` oder `7.21111`. Daneben gibt es noch auch rationale Zahlen wie `1/3` und `5/7` sowie komplexe Zahlen etwa `3+2i`.

Scheme unterscheidet zwischen exakten und inexakten Zahlen. Mit exakten Zahlen wird stets ohne Rundungsfehler gerechnet, mit inexakten Zahlen nicht. Zahlen mit Dezimalpunkt gelten als inexakt.

Es gibt eine Unzahl von eingebauten Funktionen auf Zahlen, unter anderem `+`, `-`, `*`, `/`, `abs` (Absolutwert), `max`, `min`, `quotient` (ganzzahlige Division), `remainder` (Rest), `numerator`, (Zähler), `denominator` (Nenner), `exp`, `log`, `sin`, `cos`, `tan`, und `sqrt` (Quadratwurzel).

Symbole

Man kann einem symbolischen Namen einen Wert zuordnen mittels

`(define` *Symbol Wert*`)`

Die Zuordnung eines symbolischen Namens zu einer Funktion bewerkstelligt man mit

`(define (`*Funktionsname Parameternamen*`)` *Werte*`)`

Beim Funktionsaufruf werden die Parameter wie mit einem `define` den aktuellen Werten im Funktionsaufruf zugeordnet. *Werte* steht für einen oder mehrere Schemeausdrücke, die die Parameternamen enthalten dürfen. Sie werden nun der Reihe nach ausgewertet und der letzte berechnete Wert wird dann der Rückgabewert der Funktion.

Neben den Buchstaben sind auch die meisten Sonderzeichen zur Bildung von Symbolen zugelassen. So ist etwa

```
(define (++ x) (+ x 1))
```

eine gültige Definition der Increment-Funktion `++`.

Auswertung und Quote

Scheme-Ausdrücke sind entweder einfache Daten, wie Zahlen, Buchstaben, Strings, Vektoren oder Wahrheitswerte, oder es sind Symbole oder aber sie haben die Form einer Liste.

Einfache Scheme Ausdrücke stellen in sich einen Wert dar. Einem Symbol muß, zum Beispiel durch ein vorausgehendes `define`, schon vorher ein Wert zugeordnet worden sein. Ist einem Symbol zur Laufzeit kein Wert zugeordnet, führt das in der Regel zu einer Fehlermeldung.

Bei Ausdrücken in Form einer Liste wird der Wert folgendermaßen berechnet: zunächst wird für jedes Element der Liste ein Wert berechnet. Der Wert des ersten Listenelementes muß dann eine Funktion sein mit genau so vielen Argumenten, wie noch weitere Elemente in der Liste vorhanden sind. Diese Funktion wird nun mit den Werten der weiteren Listenelemente als Parameter aufgerufen. Der Wert der Funktion ist dann der Wert des gesamten Ausdrucks.

Wenn man Symbole oder Ausdrücke als solche in Scheme verarbeiten will, muß man die Auswertung explizit unterbinden. Dies geschieht durch die Funktion `quote` (Anführungszeichen).

`(quote` *Schemeausdruck*`)` liefert als Ergebnis den Schemeausdruck selbst, nicht seinen Wert. Also `(quote (+ 1 1))` liefert als Ergebnis die dreielementige Liste `(+ 1 1)` und nicht die Zahl 2. `(quote hallo)` liefert als Ergebnis das Symbol `hallo` unabhängig davon, ob diesem Symbol schon ein Wert zugeordnet wurde.

Die Quote Funktion kann durch ein Anführungszeichen abgekürzt werden. Man kann also im obigen Beispiel auch kürzer `'(+ 1 1)` oder `'hallo` schreiben.

Listen

Listen werden in Scheme aus Paaren von Werten zusammengesetzt. Im ersten Teil eines Paares steht das Listenelement, im zweiten Teil des Paares der Rest der Liste. Für die leere Liste gibt es einen eigenen Wert, der einfach `()` geschrieben wird. Auf die beiden Teile eines Paares kann mit den Funktionen `car` (erster Teil) und `cdr` (zweiter Teil) zugegriffen werden. Zum Erzeugen von Paaren gibt es die Funktion `cons` und für ganze Listen die Funktion `list`.

So erzeugt also `(cons 1 2)` ein Paar mit `(car (cons 1 2))` gleich `1` und `(cdr (cons 1 2))` gleich `2`. Eine einelementige Liste erhält man mit `(cons 1 ())`, eine Liste, die man auch mit `(list 1)` hätte erzeugen können. `(list 1 2 3)` erzeugt eine Liste mit drei Elementen, an die man mit `(cons 0 (list 1 2 3))` vorne noch eine Null anfügen kann.

Standardfunktionen auf Listen sind `length` (Länge), `append` (zusammenhängen) und `reverse` (umkehren).

Zeichen und Strings

Einzelne Zeichen schreibt man als `#\`*Zeichen* oder `#\`*Zeichenname*. Zum Beispiel ist `#\a` ein kleines "a" und `#\newline` das Zeilenende Zeichen.

Strings (Zeichenketten) sind ein eigener Datentyp. Stringkonstanten schreibt man indem man die Zeichen in doppelte Anführungszeichen setzt. Zum Beispiel `"Hallo !"`

Strings kann man mit der Funktion `string` aus einzelnen Zeichen erzeugen, wie etwa `(string #\H #\i #\space #\!)`, oder auch mit `(make-string` *k Zeichen*`)`, wobei *k* die Länge des neuen Strings angibt der mit *Zeichen* aufgefüllt wird.

Auf einzelne Zeichen eines Strings greift man mit `(string-ref` *String k*`)` (das *k*-te Zeichen von *String*) zu.

Nützliche Funktionen sind auch `(substring` *String Anfang Ende*`)`, `string-append` (zusammenhängen) und `string-copy`.

Vektoren

Vektoren erlauben einen Zugriff auf die Elemente durch Indizierung. Die Elemente sind dabei von 0 an aufsteigend indiziert. Der Bereich der gültigen Indizes eines Vektors sind also alle exakten nicht negativen ganzen Zahlen kleiner als die Länge des Vektors. Vektoren erzeugt man mit `(make-vector` *Länge Wert*`)` oder mit `(vector` *Wert0* `...` *Wertn*`)`.

Mit `vector-length` bestimmt man die Länge eines Vektors und mit `(vector-ref` *Vektor Index*`)` greift man auf die einzelnen Elemente eines Vektors zu.

Zuweisungen

Man kann einer existierenden Variablen auch neue Werte zuweisen. Bei einer einfachen Variablen erfolgt das mit

`(set!` *Name Wert*`)`

Entspechendes gibt es auch für die Elemente eines Strings oder Vektors und die beiden Komponenten eines Paars. Die Funktionen sind dann:

`(set-string!` *Name Index Wert*`)`
`(set-vector!` *Name Index Wert*`)`
`(set-car!` *Name Wert*`)`
`(set-cdr!` *Name Wert*`)`

Scheme Grundlagen

(Version 1.1)

Bedingungen

Neben den arithmetischen Vergleichsfunktionen `=`, `<`, `>`, `<=`, und `>=` gibt es noch die verschiedensten Test-Funktionen, in der Regel mit einem Fragezeichen geschrieben. Zum Beispiel `zero?` (Null), `null?` (Leere Liste), `number?`, `pair?`, `list?`, `char?` (Zeichen), `symbol?`, `vector?`, `string?` (Zeichenkette) oder `char-upper-case?` (Großbuchstabe), um nur einige zu nennen.

Beliebige Ausdrücke können mit `equal?` auf Gleichheit getestet werden.

Es gibt auch die logischen Verknüpfungen `and` und `or` und die Werte `#t` (true) und `#f` (false).

Datenkonversion

Üblicherweise schreibt man Funktionen zur Datenkonversion mit einem Pfeil. Zum Beispiel

```
exact->inexact
number->string
string->list
list->vector
char->integer
```

Man kann davon ausgehen, daß alle sinnvollen Konversionen auch vorhanden sind.

Ein- und Ausgabe

Einfache Ausgaben macht man mit `(display` *Wert*`)`, oder `(write` *Wert*`)`. Dabei produziert `display` eine mehr lesbare Form und `write` eignet sich dazu Werte so auszugeben, daß sie wieder von Scheme eingelesen werden können.

Eine neue Zeile erhält man mit `(newline)`.

Das Einlesen eines Scheme-Ausdrucks erfolgt mit `(read)` ein einzelnes Zeichen bekommt man `(read-char)`.

if

Das if hat die Form

`(if` *Test Ja-Ausdruck Nein-Ausdruck*`)`

Das "if" ist keine normale Scheme Funktion, die immer zuerst alle Argumente auswertet. Das if wertet zuerst nur den Test aus. Ergibt dies den Wert `#f`, so wird dann der *Nein-Ausdruck* ausgewertet. Jeder anderer Wert des *Test*s ist so gut wie `#t` und führt zur Auswertung des *Ja-Ausdruck*s.

let

Mit "let" vereinbart man lokale Variable. Scheme erhält so eine Blockstruktur, vergleichbar mit dem "begin – end" von Pascal. Es hat die folgende Form:

```
(let ((Variable0  Wert0)
      (Variable1  Wert1)
         ...
      (VariableN  WertN))
   Scheme Ausdrücke)
```

Die genannten Variablen werden neu erzeugt und mit den gegebenen Werten initialisiert. In den folgenden Schemeausdrücken bis zur schließenden Klammer des "let" können die neuen Variablen dann wie gewöhnlich verwendet werden.

Ist zum Beispiel `point` ein Vektor mit `x`- und `y`-Wert, so kann man schreiben:

```
(let ((x (vector-ref point 0))
      (y (vector-ref point 1)))
   (display "Polar: ")
   (display (sqrt (+ (* x x) (* y y))))
   (dispay ", ")
   (display (atan (/ x y)))
   (newline))
```

do

Einfache Schleifen schreibt man mit dem "do"-Befehl. Er hat die Form

```
(do ((Laufvariable  Anfangswert  Nachfolger))
    (Test)
  Scheme Ausdrücke)
```

Abgesehen von dem *Test* und den *Nachfolger* Ausdrücken erinnert das "do" sehr an das "let"; in der Tat sind die Unterschiede sehr gering. Zuerst werden die genannten Variablen erzeugt und mit den gegebenen Anfangswerten initialisiert (ähnlich wie das "let" kann das "do" natürlich auch mehrere Laufvariable verwalten).

Die Auswertung der Schleife beginnt mit dem Test. Ist dieser Test wahr, so wird die Schleife beendet. Andernfalls werden wie beim "let" die folgenden Schemeausdrücke ausgewertet. Neu ist, daß im Anschluß daran die *Nachfolger* Ausdrücke benutzt werden, um neue Werte für die Variablen zu berechnen, und danach die Schleife von vorne beginnt.

Das folgende Beispiel druckt die Quadratzahlen kleiner als 100.

```
(do ((i 0 (+ i 1)))
    ((> i 10))
  (display (* i i))
  (newline) )
```

cond

Für eine Folge von Tests ist `cond` das richtige Programmkonstrukt. Schreibt man

```
(cond (Test1  Ausdruck1)
      (Test2  Ausdruck2)
      ...
      (TestN  AusdruckN) )
```

so werden der Reihe nach die *Test*s ausgewertet, bis der erste erfüllt ist, und dann wird der zugehörige *Ausdruck* ausgewertet. Sein Wert bestimmt den Wert des `cond`-Ausdrucks. Als letzter Test kann auch `else` stehen.

Beispiel:

```
(define (relation x y)
  (cond ((= x y) 'gleich)
        ((< x y) 'kleiner)
        (else    'groesser)))
```

lambda

Scheme ist insbesondere deshalb eine funktionale Programmiersprache, weil in ihr die Funktionen selbst Datenobjekte sind und den anderen Datenobjekten vollkommen gleichgestellt sind. Man kann Funktionen speichern oder vergleichen und man kann neue Funktionen zur Laufzeit erzeugen. Dazu dient das Schlüsselwort `lambda`, das in seiner Namensgebung auf die Ursprünge von Scheme im mathematischen λ-Kalkül verweist.

Eine Funktion mit den Parametern *p1*, *p2*, ... *pN* erzeugt man mit

`(lambda (`*p1 p2* ... *pN*`)` *Scheme-Ausdrücke*`)`

Ganz genau so wie bei der Definition mit `define` werden zur Laufzeit beim Aufruf der Funktion zuerst die Parameter *p1*, *p2*, ... *pN* an die aktuellen Parameterwerte gebunden und dann werden die *Scheme-Ausdrücke* der Reihe nach ausgewertet. Der letzte Ausdruck liefert den Rückgabewert der Funktion.

Funktionen, die man mit `lambda` erzeugt, werden in der Regel als Argumente an andere Funktionen übergeben oder in einer Variablen gespeichert. Ein Beispiel folgt im nächsten Abschnitt.

map

Die Funktion `map` ist ein sogenannter Iterator. Sie erhält als Argumente eine Funktion und eine Liste und ruft dann die gegebene Funktion für jedes Element der Liste einmal auf wobei das jeweilige Listenelement als Parameter übergeben wird. So etwas nennt man einen Iterator. Als Ergebnis liefert `map` eine Liste der Einzelergebnisse.

Zum Beispiel liefert

```
(map (lambda (x) (* x x)) '(1 2 3 4 5 6 7 8 9 10))
```

eine Liste der Quadratzahlen von 1 bis 100.