# MicroLisp Bedienungsanleitung

Max Rottenkolber

Sunday, 26 July 2015

## Inhaltsverzeichnis



## 1 Die *MicroLisp*-Sprache

Die *MicroLisp*-Sprache ist ein klassischer Lisp-Dialekt. MicroLisp-Quelltext besteht aus Ausdrücken und Kommentaren. Ein Kommentar beginnt mit einem Semikolon (`;`) und wird durch einen Zeilenumbruch beendet. Kommentare werden ignoriert. Ein Ausdruck ist eine so genannte *S-Expression* die speziell interpretiert wird.

Der Begriff *S-Expression* bezeichnet eine einfache Sprache zum ausdrücken von Datenliteralen. Sie wird in den meisten Lisp-Dialekten, so wie auch in *MicroLisp*, als Repräsentation für Daten und Quelltext benutzt.

Eine *S-Expression* ist entweder ein Atom oder eine Liste. Atome sind Literale für Symbole, Zahlen, Zeichen und Zeichenketten. Eine Liste ist eine Folge von *S-Expressions* umschlossen von Klammerzeichen, `(` und `)`.

```
;; Atome:
banane          ; Das Symbol banane
12              ; Die Zahl 12
1/2             ; Die Zahl 0.5
#\A             ; Das Zeichen 'A'
"Hallo, Welt!" ; Die Zeichenkette 'Hallo, Welt!'

;; Listen:
(1 2 3)              ; Eine Liste die die Zahlen 1, 2 und 3 beinhaltet
()                   ; Die leere Liste, äquivalent zum Symbol nil
(name ("Joe" "Peng") ; Eine Liste die das Symbol name, eine weitere
 alter 37)           ; Liste aus zwei Zeichenketten, noch ein Symbol
                     ; alter und die Zahl 37 enthält
```

7.4 Beispielhafte *S-Expressions* mit Kommentaren.

Quelltextbeispiel 7.4 veranschaulicht den Syntax von *S-Expressions* und verdeutlicht wie atomare Datentypen mithilfe von Listen kombiniert werden können um Datensätze darzustellen. *MicroLisp*-Ausdrücke sind Datensätze die nach folgenden Regeln interpretiert werden.

Jeder Ausdruck hat einen Wert. Alle atomaren Datentypen bis auf Symbole, also Zahlen, Zeichen und Zeichenketten, werden als Literale interpretiert. Sie haben also sich selbst als Wert. Symbole werden als Bezeichner interpretiert. Sie haben den Wert an den sie gebunden wurden. Listen werden als Prozeduraufrufe interpretiert. Der Wert des ersten Ausdrucks in der Liste ist die Prozedur und die Werte der restlichen Ausdrücke sind die Argumente, die der Prozedur übergeben werden. Prozeduraufrufe haben ihr Ergebnis als Rückgabewert.

Bezeichner können mithilfe von zwei Axiomen an Werte gebunden werden. Das `define`-Axiom bindet Bezeichner global an Werte. Ein *MicroLisp*-Programm kann beliebig viele `define`-Ausdrücke haben, diese müssen jedoch zusammen am Anfang des Programms stehen. Das `lambda`-Axiom ermöglicht die Konstruktion von anonymen Prozeduren mit Parametern und einem Rückgabewert. Innerhalb einer Prozedur werden die Bezeichner der Parameter an die übergebenen Werte gebunden.

```
;;; define-Ausdrücke am Anfang des Programms
;; Syntax von define: (define SYMBOL WERT)
(define globaler-bezeichner 1)
;; Syntax von lambda: (lambda (SYMBOLE...) AUSDRÜCKE...)
(define prozedur (lambda (x) (add x globaler-bezeichner)))

;;; Restliche Ausdrücke
3                              ; hat den Wert 3
globaler-bezeichner            ; hat den Wert 1
prozedur                       ; hat eine Prozedur als Wert
(prozedur globaler-bezeichner) ; hat den Wert 2
```

7.5 Beispielprogramm zu `define` und `lambda`.

`define` übernimmt ein Symbol, das nicht ausgewertet wird, und einen Wert als Parameter. `lambda` übernimmt eine Liste von Symbolen, die ebenfalls unausgewertet bleiben, und eine beliebige Anzahl von Ausdrücken, die zur Zeit des Prozeduraufrufs ausgewertet werden. Der Wert des letzten Ausdrucks ist der Rückgabewert der Prozedur.

Es gibt zwei weitere Axiome die ihre Parameter nicht wie Prozeduren auswerten. Das primitive Kontrollkonstrukt `cond` und `quote`. `cond` verhält sich wie das klassisch verkettete `IF-ELSE`-Statement. Es übernimmt eine beliebige Anzahl von Listen die Ausdrücke beinhalten als Parameter. Jede Liste stellt einen Fall dar. `cond` durchläuft die Fälle und wertet dabei die Bedingung, den Ersten Ausdruck des Falls, aus. Wenn der Wert der Bedingung nicht das Symbol `nil` ist werden die restlichen Ausdrücke des Falls ausgewertet und der Wert des letzten Ausdrucks ist der Rückgabewert von `cond`. Falls kein Fall zutrifft ist der Rückgabewert von `cond` das Symbol `nil`.

Das `quote`-Axiom verhält sich wie die Quotation in natürlicher Sprache. Es übernimmt einen Ausdruck und verhindert dessen Auswertung. Es ermöglicht die Darstellung von literalen Listen und Symbolen innerhalb Ausdrücken.

```
;; Beispiele zu cond
(cond (nil "Falsch")   ; gibt "Wahr zurück"
      (t "Wahr"))
(cond (12 "Nicht nil") ; gibt "Nicht nil" zurück
      (t "Wahr"))
(cond (nil "Falsch"))  ; gibt nil zurück

;; Beispiele zu quote
(quote banane)  ; gibt das Symbol banane zurück
(quote (1 2 3)) ; gibt eine Liste die die Zahlen 1, 2 und 3 beinhaltet
                ; zurück
```

7.6 Beispiele zu `cond` und `quote`. Das Symbol wird `t` konventionell als positiver Wahrheitswert benutzt. Tatsächlich ist aber nur ein Wert "falschünd zwar das Symbol `nil`, beziehungsweise die leere Liste.

Die `cond`-Ausdrücke in Beispielquelltext 7.6 funktionieren, weil die Bezeichner `nil` und `t` standardmäßig an die Symbole `nil` und `t` gebunden sind. Alternativ hätte auch `(quote nil)` und `(quote t)` geschrieben werden können.

Zur dynamischen Erzeugung von Listen werden drei Axiome `cell`, `first` und `rest` zur Verfügung gestellt. `cell` übernimmt einen beliebigen Wert und eine Liste als Parameter und gibt eine Liste zurück. Die resultierende Liste besteht aus dem übergebenen Wert und dem Inhalt der übergebenen Liste. `first` und `rest` übernehmen eine Liste als Parameter und geben jeweils das erste Element und die restlichen Elemente der Liste zurück.

```
(define liste1 (cell 1 nil))    ; liste1 bezeichnet die Liste (1)
(define liste2
  (cell 1 (quote (2 3)))) ; liste2 bezeichnet die Liste (1 2 3)
(first nil)                   ; gibt nil zurück
(rest nil)                    ; gibt nil zurück
(first liste1)                ; gibt die Zahl 1 zurück
(rest liste1)                 ; gibt das Symbol nil zurück
(rest liste2)                 ; gibt die Liste (2 3) zurück
(first (rest liste2))         ; gibt die Zahl 2 zurück
(first "Joe")                 ; gibt das Zeichen 'J' zurück
(rest "Joe")                  ; gibt die Zeichenkette "oe" zurück,
                              ; bzw. eine Liste die die Zeichen 'o'
                              ; und 'e' enthält
```

7.7 Beispiele zu `cell`, `first` und `rest`. Die letzten beiden Beispiele funktionieren weil Literale Zeichenkette tatsächlich nur eine Schreibweise für Listen von Zeichen ist. `"Joe"` ist also äquivalent zu `(quote (#J #o #e))` und `(cell #J (cell #o (cell #e nil)))`.

Für die Datentypen der Sprache sind primitive Prädikate vordefiniert. `procedure?`, `cell?`, `symbol?`, `number?` und `character?` übernehmen jeweils einen Wert und geben das Symbol `nil` zurück wenn der entsprechende Typ nicht mit dem Typ des Wertes übereinstimmt. `symbolic=`, `numeric=` und `character=` übernehmen jeweils zwei Werte des entsprechenden Typs und geben das Symbol `nil` zurück wenn die übergebenen Werte nicht äquivalent sind.

Für Zahlen gibt es zusätzlich das Vergleichsprädikat `numeric>` welches zwei Zahlen übernimmt und das Symbol `nil` zurück gibt wenn die erste übergebene Zahl nicht größer ist als die zweite übergebene Zahl. Außerdem können zwei Zahlen mit den Axiomen `add`, `subtract`, `multiply` und `divide` jeweils addiert, subtrahiert, multipliziert und dividiert werden. `modulo` dividiert zwei Zahlen mit Rest und gibt den Rest zurück.

Für Ein- und Ausgabe werden die Axiome `read`, `write` und `delete` zur Verfügung gestellt. `read` übernimmt eine Quelle und gibt einen Wert zurück. Eine Quelle ist eine Liste die Zeichenketten beinhaltet und designiert einen Dateipfad oder die Standardein- oder Ausgabe im Fall der leeren Liste (`nil`). `read` liest einen Wert aus der angegebenen Quelle und gibt es zurück. Im Falle eines Fehlers wird das Symbol `nil` zurückgegeben. `write` übernimmt einen Wert und eine Quelle und schreibt den Wert in die angegebene Quelle. Wenn ein Fehler beim Schreiben auftritt gibt `write` das Symbol `nil` zurück. `delete` übernimmt eine Quelle

und löscht den dort persistierten Wert. Wenn der Löschvorgang nicht erfolgreich war gibt `delete` das Symbol `nil` zurück.

Damit sind alle Regeln und Axiome der *MicroLisp*-Sprache definiert. Alles andere wird durch Kombination der Erkärten Axiome definiert.

# 2 Die *MicroLisp*-Entwicklungsumgebung

Die *MicroLisp*-Entwicklungsumgebung stellt zwei Befehle für die interaktive Auswertung von Ausdrücken zur Verfügung. `evaluate` übernimmt einen Ausdruck als Parameter und gibt seinen Wert zurück und `include` übernimmt eine Quelltextdatei und wertet sie aus.

Für die Kompilation von Quelltextdateien nach *ANSI C* gibt es den `compile-files`-Befehl. Er übernimmt eine beliebige Anzahl von Quelltextdateien und generiert ein entsprechendes *C*-Programm. Das resultierende *C*-Programm hat den Namen der letzten Quelltextdatei mit der Endung `.c`.

Der `define-macro` Befehl dient zur Definition von Makros. Makros müssen in *Common Lisp* Kodiert werden. `define-macro` übernimmt ein Symbol, eine Lambda-Liste und einen Prozedurkörper. Das Symbol dient als Name unter dem die Makroprozedur, die durch die Lambda-Liste und den Prozedurkörper definiert wird, registriert wird. Bevor Ausdrücke Ausgewertet oder Kompiliert werden, werden gegebenenfalls Makroausdrücke mit Ausdrücken ersetzt, die von den entsprechenden Makroprozeduren generiert wurden.

```
;; if:  Wenn CONDITION nicht nil ist wird THEN ausgewertet, ansonsten
;; wird ELSE ausgewertet.
(define-macro if (condition then &optional else)
  `(cond (,condition ,then)
         (t ,else)))

;; Wenn das if-Makro definiert ist würde beispielsweise der Ausdruck
;; (if (numeric> 2 1) 3 4) zu (cond ((numeric> 2 1) 3) (t 4))
;; ausgedehnt werden.
```

7.7 Beispielhafte Makrodefinition von `if`.