HANSER

Axel Strube-Zettler, Theodor Schönwald

Kochbuch AutoLISP

AutoCAD programmieren mit LISP

ISBN-10: 3-446-41217-4
ISBN-13: 978-3-446-41217-0

Leseprobe

3 AutoLisp Advanced

LISP has jokingly been called "the most intelligent way to misuse a computer". I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.

E.W. Dijkstra 1972

Willkommen auf diesen Seiten für AutoLisp-Programmierer! Ich möchte hier einige Kapitel zum Thema AutoLisp vorstellen, die aus einem Jahrzehnt der Beschäftigung mit dieser wunderbaren Sprache, die in AutoCAD eingebaut ist, entstanden sind.

Die ist kein Tutorial für Anfänger! Wer die hier vorgestellten Ideen nachvollziehen möchte, sollte über halbwegs solide Grundkenntnisse in der AutoLisp-Programmierung verfügen. Auch eine gewisse Vertrautheit mit dem kurzen und oft mit sehr tiefen Verschachtelungen einhergehenden Stil der Sprache ist von Vorteil. Wer sich da nicht sicher ist, sollte sich vielleicht erst einmal mein Einsteiger-Tutorial ansehen, das seit Mitte Mai ebenfalls online ist.

AutoLisp ist alles Andere als eine Makrosprache innerhalb von AutoCAD, die nur zur Automatisierung kleinerer Aufgaben geeignet ist. Im Gegenteil, auf der Skala von den in der Mächtigkeit der Funktionen wenig entwickelten Sprachen wie Assembler und C bis hin zu den High-Level-Sprachen ist sie weit oben angesiedelt, bleibt aber trotzdem recht universell.

Eigentlich ist AutoLisp eher ein Scheme-Dialekt als eine Teilmenge der gängigen Lisp-Implementierungen wie Common Lisp. Scheme ist nach meiner Ansicht die am saubersten strukturierte Sprache, die es bisher gibt. Die gesamte Syntax lässt sich in wenigen Sätzen zusammenfassen, es gibt keine Ausnahmen und Abweichungen.

Vielleicht liegt es daran, dass man auch die Syntax von AutoLisp in wenigen Minuten begriffen hat, dass viele AutoLisp-Programmierer über ein gewisses Niveau nicht hinauskommen, da sie nie wahrgenommen haben, was diese Sprache überhaupt leisten kann. Auch wenn nicht alle Sprachmerkmale eines ausgewachsenen Scheme oder Lisp in AutoCAD eingebaut wurden, braucht man nur auf weniges zu verzichten.

Manche Dinge lassen sich in AutoLisp ganz einfach realisieren, die in anderen Sprachen nur mit immensem Aufwand machbar sind. Ich denke da vor allem an Programme, die in gewissem Sinne lernfähig sind und sich selbst erweitern. Aber auch als Rapid-Prototyping-Sprache ist AutoLisp ganz hervorragend geeignet.

3.1 Zeichenketten-Verarbeitung AutoLisp

AutoLisp bietet zur Verarbeitung von Zeichenketten nur die Funktionen (substr ...), (strcat ...), (strcase ...) und (strlen ...) an. Das erscheint auf den ersten Blick sehr mager, erweist sich aber bei näherer Betrachtung als durchaus ausreichend. Nimmt man noch die Funktionen (chr ...) und (ascii ...) hinzu, lässt sich damit das gesamte Spektrum der anfallenden Aufgaben abdecken.

Allerdings ist das Arbeiten mit so wenigen Grundfunktionen in keiner Weise komfortabel. Damit man nicht das Rad immer wieder neu erfinden muss, sollte man über eine Erweiterungsbibliothek verfügen, die den Bedarf an Komfort abdeckt. Die Vorteile bei der Benutzung einer solchen Funktion liegen auf der Hand: Code kann schneller geschrieben werden, da er kürzer wird. Mit der Verkürzung des Codes entsteht natürlich auch eine größere Übersichtlichkeit. Zusätzlich wird der Code auch sicherer, wenn gewährleistet ist, dass die Bibliotheksfunktionen fehlerfrei sind.

Ich werde hier auf drei grundlegende Aufgabenstellungen eingehen:

- Bereinigen von Zeichenketten: Entfernen oder Hinzufügen von Leerstellen, white space usw.
- Verändern von Zeichenketten: Suchen und Ersetzen von Teilen und Ähnliches
- Aufteilen und Zusammenfügen: Tokenizer (Teilzeichenketten als Listen verarbeiten) und wieder zusammenfügen

Hinzu kommen noch einige Prädikat-Funktionen (Testfunktionen für Zeichen), die von den eigentlichen Bearbeitungsfunktionen u.U. benötigt werden.

Fangen wir also mit der ersten Aufgabengruppe an: Sehr oft müssen Leerstellen oder white space am Anfang oder am Ende eines Strings entfernt werden. Um größte Flexibilität zu erreichen, definieren wir zunächst eine Funktion, die möglichst allgemein ist und mit allem fertig wird. Sie bekommt nach dem zu bearbeitenden String noch ein weiteres Argument, nämlich einen zweiten String, der alle Zeichen enthält, die entfernt werden sollen. Dieser zweite String wird also als set bzw. Liste von Einzelzeichen aufgefasst.

Um aber auf diese Einzelzeichen zugreifen zu können, benötigen wir zu allererst die Hilfsfunktion (str-list ...), die einen String in eine Liste von Einzelzeichen zerlegt. Wir benötigen diese Liste, um auf die Einzelzeichen des zweiten Arguments zugreifen zu können.

```
(defun str-list(str / ls i)
  (setq i 1)
  (repeat(strlen str)
    (setq ls(cons(substr str i 1)ls))
```

```
    (setq i(1+ i))
  )
  (reverse ls)
)
```

Anwendungsbeispiel:

```
(str-list "String") => ("S" "t" "r" "i" "n" "g")
```

Jetzt können wir unsere universell verwendbaren Trimm-Funktionen schreiben:

```
(defun str-ltrimset(str cset / i)
  (if(>(strlen str)0)
    (progn
      (setq cset(str-list cset))
      (setq i 1)
      (while(member(substr str i 1)cset)
        (setq i(1+ i))
      )
      (substr str i)
    )
  )
)
(defun str-rtrimset(str cset / len)
  (if(>(strlen str)0)
    (progn
      (setq cset(str-list cset))
      (setq len(strlen str))
      (while(member(substr str len 1)cset)
        (setq len(1- len))
      )
```

```
      (substr str 1 len)
    )
  )
)
```

Anwendungsbeispiele:

```
(str-ltrimset "  __String" " _") => "String"
```

Mit dieser Grundausstattung sind wir nun in der Lage, das Repertoire sehr schnell zu vervollständigen. Um eine Funktion zu implementieren, die white space entfernt, benötigen wir folgendes Set von Einzelzeichen: 9,10,13,32. Die Funktionen für den Anfang und das Ende könnten so aussehen:

```
(defun str-ltrim(str / )
  (str-ltrimset str (apply'strcat (mapcar'chr'(9 10 13 32))))
)
(defun str-rtrim(str / )
  (str-rtrimset str (apply'strcat (mapcar'chr'(9 10 13 32)))
)
```

Anwendungsbeispiele:

```
(str-ltrim "   (4 5 6)   ")      => "(4 5 6)   "
(str-rtrim "   (4 5 6)   ")      => "   (4 5 6)"
```

Ausgesprochen schlicht fällt dann unsere allgemeine Trimm-Funktion aus, die sowohl vorne als auch hinten abschneidet. Und wenn wir schon dabei sind, machen wir gleich noch eine Alternative, die (bei Zahlen) evtl. auftretende führende Nullen mit entfernt:

```
; entfernt white space an beiden Enden
(defun str-trim(str / )
  (str-ltrim(str-rtrim str))
)
; entfernt white space an beiden Enden
; sowie führende Nullen
(defun str-trimz(str / )
  (str-ltrimset
    (str-rtrim str)
```

```
      (apply'strcat
        (mapcar'chr'(9 10 13 32 48))
      )
    )
  )
)
```

Anwendungsbeispiele:

```
(str-trim "   (4 5 6)   ")      => "(4 5 6)"
(str-trimz "  00004.75\n")      => "4.75"
```

Nun aber zum umgekehrten Fall: Zeichenketten müssen manchmal auch mit einem Zeichen aufgefüllt werden. Hier haben wir es immer nur mit einem einzigen Füllzeichen zu tun, Sets sind ausgeschlossen. Allerdings muss jetzt eine Länge angegeben werden, auf welche die Zeichenkette gebracht werden soll. Die erste Funktion füllt links auf, die zweite rechts, und die dritte zentriert den String zwischen den Füllzeichen. Alle drei Funktionen schneiden den String ab, wenn die gewünschte Länge kürzer ist als die tatsächlich vorhandene.

```
(defun str-lpadchar(str nl ch / )
  (cond
    ( (=(strlen str)nl)str)
    ( (>(strlen str)nl)(substr str 1 nl))
    (1(str-lpadchar(strcat ch str)nl ch))
  )
)
(defun str-rpadchar(str nl ch / )
  (cond
    ( (=(strlen str)nl)str)
    ( (>(strlen str)nl)(substr str 1 nl))
    (1(str-rpadchar(strcat str ch)nl ch))
  )
)
(defun str-cpadchar(str nl ch / len)
  (setq len(strlen str))
```

```
  (cond
    ( (= len nl)str)
    ( (> len nl)(substr str (/(- len nl)2) nl))
    (1
      (str-rpadchar
        (str-lpadchar str(+ len(/(- len nl)2))ch)
        nl
        ch
      )
    )
  )
)
```

Anwendungsbeispiele:

```
(str-lpadchar "String" 10 "_") => "____String"
(str-rpadchar "String" 10 "_") => "String____"
(str-cpadchar "String" 10 "_") => "__String__"
```

Ähnlich wie bei den Trimm-Funktionen können wir nun Funktionen definieren, bei denen das Füllzeichen fest verdrahtet ist. Als Beispiele: Leerstellen und Nullen (zum bündigen Darstellen von Zahlen):

```
(defun str-lpad(str nl / )(str-lpadchar str nl " "))
(defun str-rpad(str nl / )(str-rpadchar str nl " "))
(defun str-cpad(str nl / )(str-cpadchar str nl " "))
(defun str-lpadz(str nl / )(str-lpadchar str nl "0"))
(defun str-rpadz(str nl / )(str-rpadchar str nl "0"))
```

Und nun noch ein kurzer Absatz zu den bereits erwähnten Prädikatfunktionen (Testfunktionen). Ich stelle hier einige vor, ohne im Einzelnen darauf einzugehen, sie sind einfach zu verstehen. Bei der Benennung halte ich mich an die Scheme-Konvention, Prädikatfunktionen am Ende des Namens mit einem Fragezeichen zu versehen (in AutoLisp wird eher der Buchstabe p verwendet, aber das auch nicht durchgängig). Argument kann ein Zeichen (als String) oder aber die ASCII-Nummer eines Zeichens sein.

```
(defun isalnum? (c / )
  (if(=(type c)'STR)(setq c(ascii c)))
  (or(> 58  c 47)(> 91  c 64)(> 123 c 96))
)
(defun isalpha?(c / )
  (if(=(type c)'STR)(setq c(ascii c)))
  (or(> 91 c 64)(> 123 c 96))
)
(defun isascii?(c / )
  (if(=(type c)'STR)(setq c(ascii c)))
  (> 128 c 0)
)
(defun iscntrl?(c / )
  (if(=(type c)'STR)(setq c(ascii c)))
  (> 31 c 0)
)
(defun isdigit?(c / )
  (if(=(type c)'STR)(setq c(ascii c)))
  (> 58 c 47)
)
(defun islower?(c / )
  (if(=(type c)'STR)(setq c(ascii c)))
  (> 123 c 96)
)
(defun isprint?(c / )
  (if(=(type c)'STR)(setq c(ascii c)))
  (> 127 c 31)
)
(defun ispunct?(c / )
  (if(=(type c)'STR)(setq c(ascii c)))
```