Assoc-Listen

Eine Assoc-Liste (kurz *alist*) ist eine Liste von eingebetteten Sublisten, deren jeweils erstes Element als *Schlüssel* dient. Üblicherweise sind die Sublisten Paare, bestehend aus Schlüssel und Datum, die als **cons** realisiert sind.

Der ADT Assoc-Liste beinhaltet einige vordefinierte Funktionen in Common-Lisp:

(acons key datum alist) Fügt ein neues Schlüssel-Datum-Paar in eine Assoc-Liste ein.

(pairlis keys data &optional alist) Aus einer Liste von Schlüsseln und einer separaten Liste korrespondierender Daten wird eine Assoc-Liste aufgebaut bzw. ergänzt.

(assoc key alist &key test test-not key) Schlüssel-orientierter Zugriff auf die entsprechende Subliste einer Assoc-Liste.

(rassoc key alist &key test test-not key) Wert-orientierter Zugriff, die Rolle von Schlüssel und Datum sind gerade vertauscht.

Letzte/Diese Vorlesung

Streams Letzte Woche

Hashtabellen Diese Woche

Assoc-listen

Übersetzen in Common-Lisp

:Keywords

Hashing

Hashing arbeitet mit einem Feld T[0... m-1], der Hashtabelle und einer Funktion h: U $\rightarrow$ [0... m-1], der Hashfunktion. U ist das Universium. Die grundlegende Idee beim Hashing ist, für ein gegebenes Element x einer Menge S einen Speicherplatz in der Hashtabelle zu finden. Konkret: $x \in S$ wird in T[h(x)] abgespeichert.
Dies bedeutet, daß mit Hilfe der Hashfunktion der Platz für ein konkretes Element bestimmt wird. Der Zugriff auf ein Element x kann dann einfach dadurch realisiert werden, daß h(x) berechnet wird und danach die Hashtabelle direkt mit diesem "Schlüssel" adressiert wird.

Beachte: Es können Objekte beliebiger Struktur als Schlüssel verwendet werden.

Problem: Kollision bei $h(x) = h(y)$, und $x \neq y$. Vorgehensweisen: Verkettung und offene Adressierung

Assoc-Listen – Beispiele:

```
[1]> (setq udo (pairlis '(age name) '(28 (udo p)))
          maria (pairlis '(age friend) '(18 udo))) ->
((FRIEND . UDO) (AGE . 18))

[2]> (assoc 'age udo)  -->  (AGE . 28)
[3]> (setq udo (acons 'age 29 udo))
  -->  ((AGE . 29) (AGE . 28) (NAME UDO P))
[4]> (assoc 'age udo)  -->  (AGE . 29)
[5]> (cdr (assoc 'name udo))  -->  (UDO P)
[6]> (rassoc 'udo maria)  -->  (FRIEND . UDO)
[7]> (cdr (assoc 'age maria))  -->  18
```

ADT Hashtabelle

(remhash &key hash-table) Löscht den Eintrag, der mit Hilfe von key in hash-table gefunden wurde.

(clrhash hash-table) Entfernt alle Einträge aus hash-tafel.

(maphash function hash-table) Appliziert eine Funktion an alle Objekte in der Hash-tabelle. Der Funktion wird den Schlüssel und das Objekt übergeben, und muss also zwestellig sein.

Hashing in Commonlisp

Commonlisp unterstützt Hashing unmittelbar. Es ist möglich Hashtabellen zu definieren für die automatisch entsprechende Hashingfunktionen zur Verfügung gestellt werden. Es können beliebige Lispobjekte als Index fungieren.

ADT Hashtabelle

(make-hash-table &key :test :size :rehash-size :rehash-threshold) Diese Funktion erzeugt und liefert eine Hashtabelle. Das :test Argument legt fest, wie Schlüssel verglichen werden sollen (nur: eq, eql oder equal). Mit :rehash-size und :rehash-threshold kann angebenen werden, wann und um welchen Faktor die Größe der Tafel erhöht werden soll.

(gethash &key hash-table &optional (default nil)) gehthash findet den Eintrag in hash-table dessen Schlüssel key ist und liefert den assoziierten Wert. gethash liefert tatsächlich zwei Werte, wobei der Zweite ein Prädikatswert ist, der anzeigt, ob ein Eintrag gefunden wurde oder nicht. Mit **setf** können Felder der Hashtabelle mit Werten besetzt werden.

Übersetzen – was ist das in Lisp?

Übersetzen (oder Kompilieren – Compile) ist der Prozeß, ein Programm (Quelltext) in andere Instruktionen zu überführen. Dies ist meistens notwendig, um das Programm überhaupt laufen lassen zu können, was der Fall ist für Programme, die in „kompilierenden Sprachen“ geschrieben sind (z.B. Cobol, Fortran, ...). In Lisp, das eine *interpretierende Sprache* ist, geht es nur um Laufzeitsteigerungen.

Der Übersetzer in Lisp überführt das Lispprogramm in Codevektoren, welche entweder Sequenzen von Assemblerinstruktionen, Maschinenbefehle, oder sogar Instuktionen für eine abstrakte Maschine sein können.

Ein guter Übersetzer kann Makros während des Übersetzens expandieren, was im Endeffekt heißt, daß sie überhaupt keine Laufzeit kosten.

Hashtabelle – Beispiel

```
[1] > (setf ht (make-hash-table))
#<EQL hash-table with 0 entries @ #x865f12>

[2] > (setf (gethash 'al ht) 'Alabama) => ALABAMA
[3] > (setf (gethash 'ak ht) 'Alaska) => ALASKA
[4] > (setf (gethash 'ak ht) (cons 'Arkansas (gethash 'ak ht))) => ARKANSAS

[5] > (gethash 'AK) ht) => ALASKA
[6] > (gethash 'TX) ht) => NIL

[7] > (maphash #'(lambda (k v)
                       (format t "{k: ~a v: ~a}" k v))
                   ht)
{k: AK v: (ARKANSAS . ALASKA)}{k: AL v: ALABAMA}
=> NIL
```

Übersetzen – was ist das in Lisp?

Aber:

```
DIA(72): (compile 'get-all-student-names)
GET-ALL-STUDENT-NAMES
DIA(73): (compile 'student-name)
STUDENT-NAME
DIA(74): (time (get-all-student-names students))
; cpu time (non-gc) 10 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  10 msec user, 0 msec system
; real time  5 msec
; space allocation:
;  10,001 cons cells, 0 symbols, 32 other bytes
(ERIK KLAUS MARIA ...)
```


Übersetzen – was ist das in Lisp?

Beispiel:

```
(defun student-name (student)
  (first student))

(defun get-all-student-names (list-of-students)
  (let ((res nil))
    (loop for s in list-of-students do
      (push (student-name s) res))
    res))

USER(70): (length students)
10000
USER(71): (time (get-all-student-names students))
; cpu time (non-gc) 800 msec user, 0 msec system
; cpu time (gc)     70 msec user, 0 msec system
; cpu time (total)  870 msec user, 0 msec system
; real time  878 msec
; space allocation:
;  460,157 cons cells, 1 symbol, 2,080,176 other bytes
(ERIK KLAUS MARIA ...)
```

Übersetzen – was ist das in Lisp?

In dem obigen Beispiel wird beim Übersetzen die Funktion `get-all-student-names` zuerst expandiert (Die Expansion von loop betrachten wir „diesmal" nicht...) zu:

```
(defun get-all-student-names (list-of-students)
  (let ((res nil))
    (loop for s in list-of-students do
      (push (student-name s) res))
    res)) ->
(defun get-all-student-names (list-of-students)
  (let ((res nil))
    (loop for s in list-of-students do
      (push (first s) res))
    res))
```


Übersetzen – was ist das in Lisp?

```
DIA(77): (defmacro student-name (student)
           `(first ,student))
Warning: STUDENT-NAME was a function and is being redefined as a macro
STUDENT-NAME
DIA(83): (defun get-all-student-names (list-of-students)
         ...)
GET-ALL-STUDENT-NAMES
DIA(84): (time (get-all-student-names students))
; cpu time (non-gc) 1,810 msec user, 0 msec system
; cpu time (gc)     130 msec user, 0 msec system
; cpu time (total)  1,940 msec user, 0 msec system
; real time  1,953 msec
; space allocation:
;  1,010,157 cons cells, 1 symbol, 2,080,176 other bytes
(ERIK KLAUS MARIA ...)
DIA(81): (progn (compile 'student-name) (compile 'get-all-student-names))
GET-ALL-STUDENT-NAMES
DIA(82): (time (get-all-student-names students))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  3 msec
; space allocation:
;  10,001 cons cells, 0 symbols, 32 other bytes
(ERIK KLAUS MARIA ...)
```

:Keywords

Es gibt eine reihe verwendungzwecke für diese Symbole.
Sie werden verwendet als

- „Key” in einem Funktions-aufruf:
- Kommandosymbole für den Debugger
- ...

```
[43] > (defun foo (x &key (y nil))
         (print y))
FOO
[44] > (foo :y 42)
42
42
[45] > :cur
(TOP-LEVEL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```


:Keywords

Wir haben manchmal in den Vorlesungen Symbole gezeigt, die mit einem Doppelpunkt anfangen (z.B `:rules`). Diese Symbole, so genannte „keywords” sind ein bischen speziell da die nicht gequoted werden müssen. Die können auch kein Wert zugewiesen werden, sondern haben einen festen Wert – sich selbst. Ein solches Symbol kann mittels `keywordp` erkannt werden.

```
[41] > (print :foo)
:FOO ;; Print
:FOO ;; Ergebniss

[42] > (keywordp :foo)
T
```