Vorlesung 8

beim letzten Mal:

- Mapping-Funktionen: `mapc`, `mapl`, ...
- Einführung in Iteration: `dotimes`, `dolist`
- Allgemeine Iteration: `do`, `do*`
- Monster-Iteration: `loop`

heute:

- noch Beispiel für `loop`
- `eq`, `eql` und `equal`
- ADT: Assoziationslisten II
- ADT: Hashtabellen
- Übersetzen von Programmen

`loop`: Beispiel

- **Destrukturierung:**

```
(loop for (tag monat jahr) in '((21 12 01) (14 08 71))
    with monthnames = '(jan feb mar apr may jun jul aug sep oct nov dec)
    do (let ((year (if (< jahr 10) (+ jahr 2000) (+ jahr 1900))))
         (format t "~s ~s, ~s ~%" (nth (1- monat) monthnames) tag year))))
```

Mit Destrukturierung kann man sehr einfach kompliziertere Strukturen beim Durchwandern in ihre Bestandteile auflösen.

gleich $\neq$ gleich $\neq$ gleich

In Lisp können Daten auf verschiedene Arten „gleich" sein: Es wird ein Unterschied gemacht, ob wirklich *dasselbe* Objekt verglichen wird oder zwei verschiedene Objekte, die möglicherweise den gleichen Wert haben.

- Die Funktion `equal` gibt `T` zurück, wenn ihre Argumente zum gleichen Wert evaluieren.
  - `(progn (setq a '(1 2)) (setq b '(1 2)) (equal b a))` → `T`
- Die (zweistellige) Funktion `eq` gibt `T` zurück, wenn seine Argumente dasselbe Lisp-Objekt sind:
  - `(progn (setq a '(1 2)) (setq b a) (eq b a))` → `T`, aber
  - `(progn (setq a '(1 2)) (setq b '(1 2)) (eq b a))` → `NIL` ! In diesem Fall zeigen `a` und `b` auf verschiedene `cons`-Zellen.

Symbole existieren nur einmal im Speicher, deshalb gilt immer z. B. `(eq 'foo 'foo)`. Bei *Zahlen* ist nicht verboten, daß es mehrere Kopien in einem Lisp-Interpreter gibt. Dafür gibt es noch eine weitere Funktion

- Die Funktion `eql` verhält sich wie `eq`, aber zusätzlich ist garantiert, daß sie auch für gleiche Zahlen `T` zurückliefert.
  - `(eq 5 5)` → `?`, aber `(eql 5 5)` → `T`.

Warum gleich *drei* „Gleichs"?

wir könnten `equal` auch definieren, wenn wir nur `eql` (oder auch nur `eq`) zur Verfügung hätten. Das würde ungefähr so aussehen (Strings etc. mal nicht berücksichtigt):

```
(defun my-equal (a b)
  (cond ((or (atom a) (atom b)) (eql a b))
        ((my-equal (car a) (car b))
         (my-equal (cdr a) (cdr b)))))
```

- `equal` muß im Allgemeinen einen rekursiven Test machen, um festzustellen, ob seine Argumente äquivalent sind.
- `eq` muß nur testen, ob seine Argumente *das gleiche Objekt* sind. Auf den meisten Rechnern erfordert das nur einen einzigen Befehl (Vergleich zweier Speicheradressen).

→ `eq` ist zwar primitiver als `equal`, aber wenn es ausreicht, viel effizienter. `eql` ist zwar aufwendiger als `eq`, aber immer noch viel effizienter als `equal`.

Man sollte also immer das „kürzestmögliche Gleich" verwenden, aber dabei aufpassen, daß das Programm auch korrekt bleibt.

`:test` und Keyword-Parameter

Bei Funktionen, die Vergleiche anstellen, kann man oft angeben, welche Vergleichsfunktion benutzt werden soll. Dies geschieht mit einem *Keyword-Parameter* `:test`

z. B. bei `member`:

- `(member '(1 2) '((2 3) (1 2) (4 3) :test #'eql)` → `NIL`
- `(member '(1 2) '((2 3) (1 2) (4 3) :test #'equal)` → `T`

wegen der höheren Effizienz benutzt z. B. `member` in Common Lisp standardmäßig zum Vergleichen `eql`, also Vorsicht!

Andere Funktionen, die `:test` benutzen: `remove`, `position`, `substitute`, ...

ADT: Assoziationslisten II

Es fehlen uns noch einige Funktionen fuer den Datentyp Assoziationsliste.

Zur Erinnerung:

Eine *Assoziationsliste* (kurz *Assoc-Liste* oder *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.

- Konstruktoren:
  - (`acons` <*key*> <*datum*> <*alist*>)
    Fügt ein neues Schlüssel-Wert-Paar in eine Assoc-Liste ein.
  - (`pairlis` <*keys*> <*data*> `&optional` <*alist*>)
    Aus einer Liste von Schlüsseln und einer Liste von Daten wird eine Assoc-Liste aufgebaut bzw. ergänzt.
- Zugriffsfunktionen:
  - (`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 Wert sind vertauscht.

Assoziationslisten: Beispiele

```
USER(1): (setq udo (pairlis '(age name) '(28 (udo p)))
               maria (pairlis '(age friend) '(18 udo)))
((FRIEND . UDO) (AGE . 18))
USER(2): (assoc 'age udo)
(AGE . 28)
USER(3): (setq udo (acons 'age 29 udo))
((AGE . 29) (NAME UDO P) (AGE . 28))
USER(4): (assoc 'age udo)
(AGE . 29)
USER(5): (cdr (assoc 'name udo))
(UDO P)
USER(6): (rassoc 'udo maria)
(FRIEND . UDO)
USER(7): (cdr (assoc 'age maria))
18
```

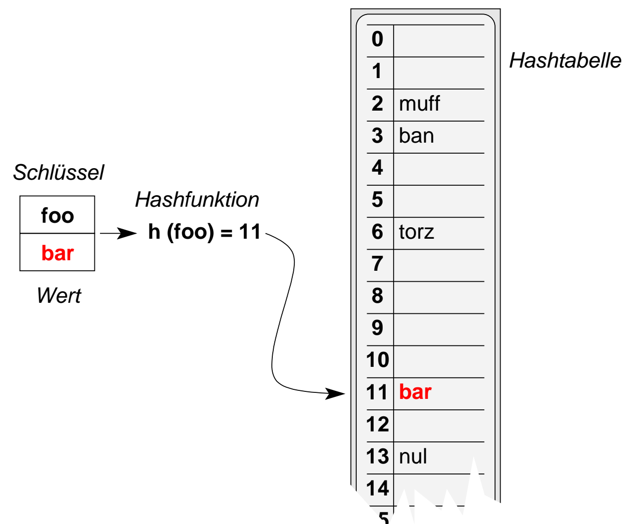

ADT: Hashtabellen (1)

Mit *Hashing* kann man ähnlich wie mit Assoziationslisten Schlüssel-Wert-Paare speichern. Der Vorteil ist, daß der Wert zu einem Schlüssel im Allgemeinen wesentlich schneller gefunden werden kann.

Hashing arbeitet mit einem Feld $T[0 \ldots n-1]$ (der *Hashtabelle*) und einer Funktion $h : U \to [0, n-1]$ (der *Hashfunktion*). U ist die Menge der möglichen Schlüssel. Die Idee beim Hashing ist, für jeden Schlüssel k der Menge U einen Speicherplatz in der Hashtabelle zu finden. Dieser Speicherplatz wird mittels der Hashfunktion ermittelt: Das Datum zu $k \in U$ wird in $T[h(k)]$ abgespeichert. Beim Auslesen läuft es genauso: um das Datum zu k zu finden, schaut man ebenfalls in $T[h(k)]$ nach.

Vorteil: Wenn das Berechnen von $h(k)$ schnell geht, weiß man schnell, wo das Datum zu finden ist, und muß nicht mühsam eine Liste der Länge nach durchsuchen.

Hashtabellen (2)



Hashtabellen (3)

Der Datentyp `hashtable` ist in Common-Lisp eingebaut. Die Hashfunktion wird automatisch zur Verfügung gestellt. Es können beliebige Lisp-Objekte als Schlüssel fungieren.

- Konstruktoren:
  - `(make-hash-table &key :test :size :rehash-size :rehash-threshold)`
    Erzeugt und liefert eine Hashtabelle. Die Bedeutung der (optionalen) Keyword-Parameter: `:test` - Welcher Test zum Schlüsselvergleich benutzt werden soll (`eq`, `eql` oder `equal`), `:size` - die anfängliche Größe der Hashtabelle, `:rehash-threshold` - wann ein *rehash* durchgeführt werden soll, `:rehash-size` - um wieviel bei einem *rehash* vergrößert werden soll (bei den beiden letzten Parametern bedeutet eine Ganzzahl einen absoluten Wert, eine Fließkommazahl einen prozentualen Anteil).

```
USER(1): (setq *hashtable* (make-hash-table))
#<EQL hash-table with 0 entries @ #x847b98a>
```

- Zugriffsfunktionen:
  - `(gethash` *<key>* *<hashtable>* `&optional` *<(default nil)>*`)`
    Liefert den Wert, der unter *<key>* in *<hashtable>* gespeichert ist; wenn keiner vorhanden ist, stattdessen *<default>*.
    Mit `setf` kann `gethash` auch benutzt werden, um neue Schlüssel/Wert-Paare in die Hashtabelle einzutragen.

```
USER(2): (setf (gethash 'zok *hashtable*) 'bouf)
BOUF
USER(3): (gethash 'zok *hashtable*)
BOUF
T       ;; <-- zweiter Rueckgabewert: Wert gefunden
```

- Manipulation und sonstige Funktionen:
  - `(remhash` <*key*> <*hashtable*>)
    Löscht den Eintrag, der unter <*key*> in <*hashtable*> gefunden wurde.
  - `(clrhash` <*hashtable*>)
    Löscht die gesamte Hashtabelle <*hashtable*> vollständig.
  - `(maphash` <*function*> <*hashtable*>)
    Wendet die Funktion <*function*> auf alle Schlüssel und Werte der Hashtabelle <*hashtable*> an und liefert eine Liste der Ergebnisse.
    Der Funktion <*function*> werden jeweils der Schlüssel und der Wert übergeben, sie muß also zweistellig sein.

Hashtabellen (4)

Ein Problem beim Hashing ist die Möglichkeit von *Kollisionen*. Normalerweise hat die Hashtabelle weniger Speicherplätze als U groß ist. Wenn zwei verschiedene Schlüssel den gleichen Hashwert ergeben, müßte man beide Daten an derselben Stelle speichern.

Lösung 1: Man tut genau das. Man speichert an jeder Speicherstelle in der Hashtabelle eine *Liste* von Werten. Nachteil: Das Suchen von Werten dauert wieder länger (im schlechtesten Fall sind alle Hashwerte gleich!), und die Abbildung ist nicht mehr eindeutig.

Lösung 2: Wenn die Hashtabelle „zu voll“ wird und Kollisionen wahrscheinlich, ordnet man die Werte in einer größeren Hashtabelle neu ein (*rehashing*). So macht es Lisp normalerweise. Nachteil: Das kann je nach Größe der Hashtabelle eine Weile dauern.

Lösung 3: Man benutzt mehrstufiges Hashing, um eine „perfekte Hashfunktion“ für U zu bauen, mit der es keine Kollisionen gibt. Nachteil: für diese Vorlesung zu kompliziert ;-)

Hashtabellen: Beispiele

```
USER(67): (setq *hashtable* (make-hash-table :test #'equal))
#<EQUAL hash-table with 0 entries @ #x847d67a>
USER(68): (setf (gethash 'blue *hashtable*) "sky")
"sky"
USER(69): (setf (gethash '(one two) *hashtable*) '(1 2))
(1 2)
USER(70): (setf (gethash '(one two) *hashtable*) 'other-value)
OTHER-VALUE
USER(71): (gethash 'not-in *hashtable* 'vanilla)
VANILLA
NIL
USER(72): (maphash (lambda (k v)
                     (format t "Schluessel ~s -> Wert ~s ~%" k v))
                   *hashtable*)
Schluessel (ONE TWO) -> Wert OTHER-VALUE
Schluessel BLUE -> Wert "sky"
NIL
```


Übersetzen von Lisp-Programmen (1)

Lisp-Programme werden normalerweise Schritt für Schritt vom Computer *interpretiert*. Man kann ein Programm auch vorher vollständig in Maschinenbefehle übersetzen (*kompilieren*). Ein kompiliertes Programm läuft schneller, weil der Übersetzungsaufwand wegfällt und auch andere Optimierungen vorgenommen werden können (z. B. Übersetzen von Endrekursion in Iteration).

Es gibt aber auch Nachteile, vor allem in Bezug auf das Debugging:

- Man kann nicht mit `step` schrittweise in kompiliertem Code vorangehen oder den Programmlauf unterbrechen, um sich Werte von Variablen anzusehen
- Das Programm ist nicht mehr direkt lesbar / änderbar (es sei den, man hat das Original – die *Quelldatei* – noch)
- Fehler können möglicherweise nicht mehr vom Lisp-Interpreter abgefangen werden.

Oft eine gute Idee:

- Bei der Programmentwicklung und beim Testen des Programms den Interpreter benutzen und sich damit das Debugging erleichtern,
- Wenn das Programm stabil läuft und bald eingesetzt werden soll, zu einer kompilierten Version übergehen (aber auch damit noch testen!)

Übersetzen von Lisp-Programmen (2)

Es ist auch möglich, nur einzelne Funktionen statt des gesamten Programms zu kompilieren. Diese können ohne Probleme mit interpretiertem Code zusammen benutzt werden.

Eine Funktion kann kompiliert werden mit `(compile` *<function-name>*`)`:

```
USER(10): (compile 'foo)
FOO
```

Mit `(compile-file` *<file-name>*`)` kann eine Lisp-Datei kompiliert werden. Wenn man die Datei später lädt, wird nur das Kompilat geladen.

```
USER(1): (compile-file "bla")
;;; Compiling file bla.cl
; Compiling FOO
Warning: Variable Y is never used.
; Compiling ZOK
Warning: Free reference to undeclared variable *LA* assumed special.
;;; Writing fasl file bla.fasl
;;; Fasl write complete
T
USER(2): (load "bla")
; Fast loading ./bla.fasl
T
```