Letzte/Diese Vorlesung

Letzte Woche:

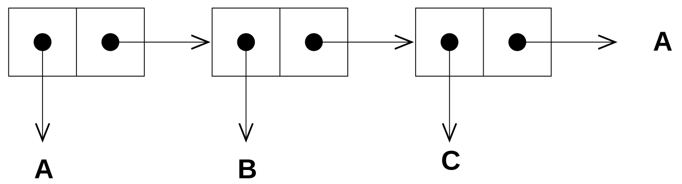
- Funktionsobjekte
- Lambdalistensyntax
- Conszellen

Heute:

- Conszellen – Unterschied `eq` `equal`
- Makro

Conszellen – Punktierte Paare

Wie repräsentiert man textuell folgende Liste?:



Die Lösung: `(a b c . a)`

Man kann aber die Punkte verwenden, um *jeden* Rest-Zeiger anzugeben:

`(a . (b . (c . a)))` ≡ `(a b c . d)`

Conszellen – Unterschied `eq` `equal`

Der Unterschied zwischen `eq` und `equal` ist, daß das Prädikat `eq` auf *Identität* testet und `equal` auf *Strukturgleichheit* (Isomorphie).

Beispiel:

```
(setq a '(1 2))
(setq b '(1 2))

(eq a b) --> NIL
(equal a b) --> T
```


Conszellen – Unterschied `eq` `equal`

Weitere Beispiele:

```
(setq a "Foo")
(setq b a)
(setq c (copy-seq a))

(eq a b) --> T
(equal a b) --> T

(eq a c) --> NIL
(equal a c) --> T
```

Conszellen – `eql`

Zum Test, ob zwei Zahlen oder Buchstaben gleich sind, gibt es zusätzlich das Prädikat `eql`. Zwei Objekte sind `eql`-gleich, wenn sie entweder `eq` sind, oder wenn sie Zahlen vom selben Typ mit gleichem Wert sind, oder wenn sie Buchstaben sind die das selbe Zeichen repräsentieren. **Wichtig:** `equal` benutzt `eql`.

```
(eq 3 3) --> T oder NIL je nach Implementation
(eql 3 3) --> T

(eq 3.0 3.0) --> T oder NIL je nach Implementation
(eql 3.0 3.0) --> T

(eq #\A #\A) --> T oder NIL je nach Implementation
(eql #\A #\A) --> T
```


Conszellen – `eql` – `=`

Für Zahlen gibt es natürlich immer noch das Prädikat `=()`, das das gleiche Ergebnis wie `eql()` liefert (wobei `=()` beliebig viele Argumente akzeptiert!).

```
(= 1.0 1.0) --> T
(eq 1.0 1.0) --> T oder NIL je nach Implementation
(eql 1.0 1.0) --> T
(equal 1.0 1.0) --> T
```

(Für Zahlen gibt es auch die Prädikate `/=()`, `<()`, `>()`, `<=()`, und `>=()`, die alle beliebig viele Argumente akzeptieren.)

Conszellen – `substitute-list`

Schreibe eine Funktion `substitute-list()`, die in einer beliebigen Liste alle Vorkommen eines Elementes gegen ein anderes austauscht. Die Elemente können sowohl Symbole als auch Zahlen sein.

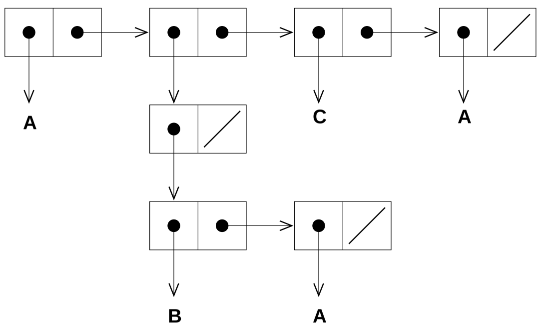
Lösung

```
(defun substitute-list (l x y)
  (cond ((null l) nil)
        ((atom l) (cond ((eql l x) y)
                        (t l)))
        (t (cons (substitute-list (first l) x y)
                 (substitute-list (rest l) x y)))))

(substitute-list '(a ((b a)) c a) 'c 'd)  --> (A ((B A)) D A)
```

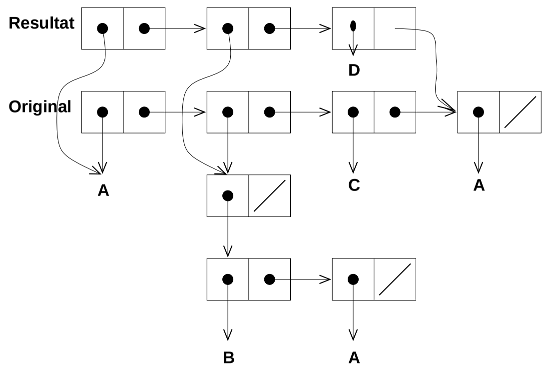

Conszellen – `substitute-list`

Graphische Darstellung `(a ((b a)) c a)`:



Conszellen – `substitute-list`

Wir sparen Conszellen: Schreibe die Funktion so, daß in den Substrukturen, die das Element nicht enthalten, keine neue Conszelle(n) benutzt wird. Für z.B. `(substitute-list '(a ((b a)) c a) 'c 'd)`:



Conszellen – `substitute-list`

```
(defun substitute-list (l x y)
  (cond ((null l) nil)
        ((atom l) (cond ((eql l x) y)
                        (t l)))
        (t (let ((s-first (substitute-list (first l) x y))
                 (s-rest (substitute-list (rest l) x y))))
             (cond ((or (not (eq s-first (first l)))
                        (not (eq s-rest (rest l))))
                    (cons s-first s-rest))
                   (t l))))))

(substitute-list '(a ((b a)) c a) 'c 'd)  --> (A ((B A)) D A)
```

Makro – Motivation

Wir haben schon gelernt, daß wir mit unserem bisherigen Wissen ein eigenes `if()` *nicht* schreiben können.

```
;;
(defun my-if (test then else)
  (if test then else))

? (my-if (= 2 3) (print 'not_very_probable) (print 'should_work))

NOT_VERY_PROBABLE ;; then
SHOULD_WORK       ;; else
SHOULD_WORK       ;; Ergebnis
```


Makro

Der Makro-Mechanismus in LISP erlaubt es, beliebige Prozeduren zu definieren, die bestimmte Anweisungen vor der Auswertung bzw. vor dem Compilieren zunächst in andere Anweisungen umwandeln. Die geschieht auf der Ebene der symbolischen Ausdrücke und nicht auf der Ebene der Zeichenketten, wie etwa in vielen anderen Programmiersprachen.

Findet der Interpreter bei der Evaluation eines Aufrufs einen Prozedurbezeichner, der als Makro definiert ist, so wird der Makroaufruf, gemäß der Makrodefinition, zuerst expandiert und dann anstatt der ursprünglichen Anweisung ausgewertet.

Im Gegensatz zu normalen Funktionen evaluieren Makros ihre Parameter also nicht.

Makro

Ein Symbol kann entweder als Makro oder aber als Funktion definiert sein, jedoch nicht beides gleichzeitig. Wichtig ist auch, daß Makros im engen Sinne keine Funktionen sind. Sie können daher nicht als funktionale Argumente mit Funktionen wie **apply**, **funcall** oder **map** verwendet werden.

Makros werden mit dem **defmacro**-Makro definiert, dessen Syntax weitgehend der von **defun** entspricht.

(defmacro *name lambda-list* {**form**}* **)**

Als zusätzliches Lambdalisten-Schlüsselwort ist beispielsweise **&body** erlaubt. Dieses Schlüsselwort entspricht ganz genau **&rest**.

Makro – Beispiel

Schreibe ein eigenes `if`!

```
(defmacro my-if (test then else)
  (list 'cond (list test then) (list 't else)))

(my-if (eql 3 2) 14 42) -->
;; Expand
(LIST 'COND (LIST (EQL 2 3) 14) (LIST 'T 42)) -->
;; Eval
(COND ((EQL 2 3) 14) (T 42)) -->
42
```

Makro – Mehrere Verwendungszwecke

Man kann die Eigenschaft, daß ein Makro Anweisungen gegen andere Anweisungen austauscht, ausnutzen, die Geschwindigkeit von einem Programm zu beschleuningen – Ein guter Kompiler expandiert die Makros schon beim Übersetzen → Man "verliert" also Funktionsaufrufe.

Makro – Weitere Verwendungszwecke

Beispeil

In Übungsaufgabe 7 haben wir einen ADT "`student`" mit einem Konstruktor, `make-student()`, und drei Selektoren, `student-name, student-matrikel` und `student-hauptfach` geschrieben. Beim "profiling" von dem Programm sind wir dahintergekommen, daß das Programm sehr viel Zeit beim Aufrufen von den drei Selektoren verbraucht – wir wollen aber nicht auf die Ergebnisse so lange warten. Wir entscheiden uns, die drei Selektoren als Makros umzuschreiben.

Makro – Weitere Verwendungszwecke

```
;; Konstruktor
(defun make-student (name matrikel hauptfach)
  (list hauptfach matrikel name))

;;      Funktion                                        Makro
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun student-name (student)      (defmacro student-name (student)
  (third student))                     (list 'third student))

(defun student-matrikel (student)  (defmacro student-name (student)
  (second student))                    (list 'second student))

(defun student-hauptfach (student) (defmacro student-hauptfach (student)
  (first student))                     (list 'first student))
```


Makro – Weitere Verwendungszwecke

Herr Tippser hat die Aufgabe, *alle* Studierenden in unsere Datenbank einzutragen. Für jede(n) Student(in) muss er die Variable `*students*` umsetzen mit z.B.:

```
(setq *students*
  (cons (make-student "Hannibal" 4243 'computerlinguistique)
        *students*))
```

Nach schon 13 Studenten geht seine `quote`-Taste kaputt!!! Können wir ihm helfen? Wir fragen nach, ob er mit folgender Schreibweise zufrieden ist (was er gleich akzeptiert):

```
(add-student "Hannibal" 4243 computerlinguistique)
```

Makro – Weitere Verwendungszwecke

```
(defmacro add-student (name matrikel hauptfach)
  (list 'setq '*students*
        (list 'cons
              (list 'make-student name (list 'quote fach))
              '*students*))
```


Makro – back-quote, un-quote und splice

Die bisher betrachteten Konstruktoren evaluieren ihre Argumente entweder gar nicht (**quote**) oder immer (**cons**, **list**, **append**). Will man nun eine mehrfach geschachtelte Liste unter Rückgriff auf bestimmte Variablen aufbauen, so sind hierzu überaus komplexe Ausdrücke nötig.

In Lisp steht daher neben dem **Quote-Makro** (`'`) noch das **Backquote-Makro** (`` ` ``) zur Verfügung, das es erlaubt, innerhalb eines quotierten Ausdrucks einzelne Teilausdrücke explizit zu evaluieren.

Die zu evaluierenden Teile werden durch ein vorangestelltes Komma (un-quote) markiert. Ist der Wert eines zu evaluierenden Teilausdrucks eine Liste, so können, statt der Liste selbst, auch nur die Elemente eingebaut werden (*splicing*). Dies wird durch `,@` markiert.

Makro – back-quote, un-quote und splice

```
(defparameter *computer-typ* 'macintosh)

? (list 'ein 'neuer *computer-typ* 'ist 'da)
--> (ein neuer macintosh ist da)
;; Einfacher?
? `(ein neuer ,computer-typ ist da)
--> (ein neuer macintosh ist da)
;; Weiter...
(defparameter l1 '(1))
(defparameter l2 '(2 3))

? `(l1 ,l1 (,(rest l2)) ,@l2)
--> (l1 (1) ((3)) 2 3)
```


Makro – back-quote, un-quote und splice

Heimaufgabe (freiwillig): Wie schreibt man mit hilfe von `append`, `cons` und `list` einen equivalenten Ausdruck für `` `(l1 ,l1 (,(rest l2)) ,@l2) ``?

Makro – back-quote, un-quote und splice

Wir können jetzt die obigen Beispiele mithilfe von back-, un-quote und splice schreiben:

```
;;      Funktion                                        Makro
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun student-name (student)      (defmacro student-name (student)
  (third student))                     `(third ,student))

;;; ... usw

(defmacro add-student (name matrikel hauptfach)
  `(setq *students*
      (cons (make-student ,name ,matrikel ',hauptfach)
            *students*)))
```