Letzte/Diese Vorlesung

(Vorvor-)Letzte Woche:

- Rekursion
- Doppelrekursion
- `cond`

Letzte/Diese Vorlesung

Heute:

Lösungen

Gute vs. schlechte Programmierstile

Restrekursion

Bindungsumgebungen

Lösungen

Für die Lösungen möchten wir folgendes gemacht haben:

- Sinnvolle Namen (für Funktionen und Variablen)
- Die Lösungen soll ausreichend *getestet* sein. Guck z.B., daßalle Klauseln in einem `cond`-ausdrück getestet sind.
- Die Tests sollten beigefügt sein.
- Bitte keine “Lösungen”, die offensichtlich nicht funktionieren.

Gute vs. Schlechte Programmierstile – Indentierung

Indentierung ist ein tolles Konzept, um Programme lesbar(er) für uns Menchen zu machen.

- Was ist Indentierung?
  - Einrückung und Formatierung von Programmen, um die Lesbarkeit zu steigern.
- Es gibt für fast jede Form in Common-Lisp eine allgemein akzeptierte Art, den Ausdrücken ein “schönes” Aussehen zu geben.
- Es gibt nicht <u>die</u> Art ein Programm zu indentieren — es gibt aber ein paar allgemeine Regeln.

Gute vs. Schlechte Programmierstile – Beispiele:

- `defun` – `!` zweimal schlecht indentiert:

```
(defun !
(n) (cond (
(= 0 n) 1) (t (* n (! (
1- n))))))
```

```
(defun !(n)(cond((= 0 n)1)(t(* n(!(1- n))))))
```

- `defun` – `!` besser indentiert:

```
(defun ! (n)
  (cond ((= 0 n) 1)
        (t (* n
              (! (1- n))))))
```


Gute vs. Schlechte Programmierstile – Beispiele:

Ein paar Beispiele aus unserem Kurs ...

- `pair-up` (Lösung von NN)

```
(defun pair-up (L) (cond  ((or (null L)  (not (rest L)))  nil)
(T  (cons (list (first L) (second L))  (pair-up (rest (rest L))))))
```

- `pair-up` (Lösung von NN)

```
(defun pair-up (liste)
      (cond ((> (length liste) 1)
            (append (list(list (first liste) (second liste))) (pair-up
```

Gute vs. schlechte Programmierstile – Beispiele:

- `pair-up` besser?

```
(DEFUN PAIR-UP (L)
  (COND ((OR (NULL L) (NOT (REST L))) NIL)
        (T (CONS (LIST (FIRST L) (SECOND L)) (PAIR-UP (REST (REST L)))))))
```

- `pair-up` Noch besser?

```
(defun pair-up (liste)
  (cond ((or (null liste)
             (not (rest liste)))
          nil)
        (t
          (cons (list (first liste)
                      (second liste))
                (pair-up (rest (rest liste)))))))
```


Gute vs. schlechte Programmierstile – Beispiele:

Namen für formale Parameter – ein paar Tips:

- Eine Liste heißt: `l, list, liste` aber nie `x, y, ...`.
- Ein Integer heißt `i, j, k, n, m, ...` aber nie `list, x` (l?)
- Eine Fließkommazahl heißt z.B. `x, y, z` aber nie `n, liste, ...`
- Ein Atom heißt z.B. `atom`
- Eine beliebige Struktur heißt z.B. `obj, object, objekt, ding, thing, ...` aber nie `l, n, ...` (möglicherweise `x`).
- Eine temporäre Variable heißt z.B. `tmp, temp, ...`

Restrekursion: Vergleich

nicht restrekursiv

```
(defun ! (i)
  (cond ((= i 0) 1)
        (t (* i (! (1- i))))))
```

- das Ergebnis des rekursiven Aufrufs geht in weitere Berechnung(en) ein;

restrekursiv

```
(defun ! (i)
  (help i 1))
(defun !-help (i n)
  (cond ((= i 0) n)
        (t (help (1- i)
                 (* i n)))))
```

- der rekursive Aufruf ist gleichzeitig der Rückgabewert der Funktion.

Rekursion – Beispiel (Restrekursion)

```
Nicht Restrekursiv
? (! 3)
(* 3 (! 2))
(* 3 (* 2 (! 1)))
(* 3 (* 2 (* 1 (! 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

```
Restrekursiv
? (! 3)
(!-help 3 1)
(!-help 2 3)
(!-help 1 6)
(!-help 0 6)
6
```

Restrekursion

- Was ist Restrekursion?
  - *Iterative Probleme rekursiv zu behandeln.*

    oder
  - Der rekursive Aufruf ist die *letzte* Anweisung, die im Funktionsrumpf ausgeführt wird.
- Können alle rekursiven Funktionen restrekursiv umgeschrieben werden?
- Nein, aber alle “iterativen Probleme” können (rest)rekursiv beschrieben werden.

Bindungsumgebungen

- Beim Aufruf einer Funktion werden die *formalen Parametern*, FP, der Funktion F mit den *aktuellen Parametern*, AP, ihres Aufrufmusters gebunden (Bindungsumgebung BU_f von f) .
- BU_f gilt nur für die Dauer der Auswertung von f.
- Wird in f selbst eine Funktion g aufgerufen, so wird eine neue Bindungsumgebung, BU_g, erzeugt.
- BU_g ist in BU_f eingebettet.
- Ist g abgearbeitet, wird BU_g gelöscht

Bindungsumgebungen – `let` und `let*`

CommonLisp enthält zwei Konstruktionen für *lokale* Bindungsumgebungen – `let` bzw `let*`. Deren formalen Syntax:

- **(let** $(\{(\langle \mathit{var}\rangle\ \langle \mathit{value}\rangle)\}^*)\ \langle \mathit{form}\rangle^*$**)**
- **(let*** $(\{(\langle \mathit{var}\rangle\ \langle \mathit{value}\rangle)\}^*)\ \langle \mathit{form}\rangle^*$**)**

Beispiel:

```
(defun where (obj list)
  (cond ((null list) nil)
        ((equal obj (first list) 0))
        (t (let ((tmp (where obj (rest list))))
             (cond ((numberp obj) (1+ obj))
                   (t nil))))))
```


Bindungsumgebungen – `let` und `let*`

`let` und `let*` unterscheiden sich dadurch, daßdie Bindungen in einem `let` *parallel* abgerarbeitet werden und in einem `let*` *sequentiell.*

```
(defun sum-pair (liste-mit-zwei-zahlen)
  (let ((erste-zahl (first liste-mit-zwei-zahlen))
        (zweite-zahl (second liste-mit-zwei-zahlen)))
    (+ erste-zahl zweite-zahl)))

(defun sum-tripple (liste-mit-drei-zahlen)
  (let* ((erste-zahl (first liste-mit-drei-zahlen))
         (zwischen-summe (+ erste-zahl (second liste-mit-drei-zahlen)))
         (dritte-zahl (third liste-mit-drei-zahlen)))
    (+ zwischen-summe dritte-zahl)))
```

Bindungsumgebungen – `let` und `let*`

`let*` ist syntaktischer Zucker für eine Kombination von `let`-Ausdrücken:

```
(let* ((a 3)
       (b 4))
  (+ a b))
```

≡

```
(let ((a 3))
  (let ((b 4))
    (+ a b)))
```


Bindungsumgebungen – `let` und `let*`

Die Variablen in einem `let`-/`let*`-Ausdruck *verdecken* eventuelle globale Variablen

```
(defun komische-flaeche (radius)
  (let ((pi 42))
    (* radius radius pi)))
```

Bindungsumgebungen – `defparameter`

Globale Werte oder Parametern werden mit `defparameter` definiert.

- **(defparameter** <*name*> <*initial value*>**)**

Globale Variablen markiert man gerne mit Sternchen (*) auf beiden Seiten.

Beispiel:

```
(defparameter *the-answer* 42)
```


Bindungsumgebungen – Beispiel

```
? (defparameter foo 42)
42
? (defun foo (foo)
    (set 'foo (+ 42 foo)))
foo
? (foo 42)
84
? (foo foo)
84
? foo
42
```