Letzte/Diese Vorlesung

Heute:

Bindungsumgebungen

Was ist ein Symbol

Lambda-funktionen

`defun` **II**

Letzte/Diese Vorlesung

Letzte Woche:

Blöcke/Seiteneffekte

Rekursion II

Programmierstil

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.

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.

Zum schluss...

- Bitte keine "Lösungen", die offensichtlich nicht funktionieren.
- Bitte keine doc/pdf/... schickt uns normaler text (ascii)!

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

Rekursion – Beispiel (Restrekursion)

```
 Linear                           Iterativ
? (! 3)                          ? (! 3)
(* 3 (! 2))                      (!-help 3 1)
(* 3 (* 2 (! 1)))                (!-help 2 3)
(* 3 (* 2 (* 1 (! 0))))          (!-help 1 6)
(* 3 (* 2 (* 1 1)))              (!-help 0 6)
(* 3 (* 2 1))                    6
(* 3 2)
6
```

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*`

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

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

Beispiel:

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

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 – `let` und `let*`

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

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

$\equiv$

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

Bindungsumgebungen – Beispiel

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


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)
```

Funktionsobjekte Die Repräsentation von Funktionsobjekten in LISP basiert auf der *Lambda-Notation* (vgl. Church41).

Lambda-Abstraktion – Ein *Lambda-Ausdruck*, wie z.B. `(lambda (x) (+ x 1))` steht für die Funktion `f` mit `f(x) = x + 1`. Wichtig hierbei ist, daß kein Name für die Funktion benötigt wird, d.h. Lambda-Ausdrücke beschreiben *anonyme* Funktionen.

Lambda-Applikation – Ein Lambda-Ausdruck beschreibt eine Funktion, ist aber selbst kein direkt auswertbarer Ausdruck. Erst durch die zusätzliche Angabe von Argumenten entsteht ein auswertbarer Funktionsaufruf. Bei einer solchen *Applikation* eines Lambda-Ausdrucks werden die formalen Parameter durch die konkreten Argumente, d.h. die Aufrufparameter, ersetzt und die angegebene Berechnungsvorschrift angewendet.

Funktionsobjekte

Die Funktion **defun** erzeugt eine Funktionsdefinition und legt diese in der Funktionszelle des durch den Funktionsnamen bezeichneten Symbols ab. Über den Namen des Symbols kann dann die durch die Definition spezifizierte Funktion angesprochen werden.

Naives Modell eines LISP-Symbols:

```
foo  ───┬─→ value:    NIL
        ├─→ function: ???
        ├─→ plist:    (:USED NIL :DOC "by JR")
        ├─→ name:     "FOO"
        └─→ package:  #:<"USER" package>
```

Frage: Wie sieht ein solches Funktionsobjekt aus ?

Funktionsobjekte – Funktionsaufruf

Um einen lambda-Ausdrück verwenden zu können, muß man ihn in ein *Funktionsobjekt* umwandeln. Das wird mit Hilfe der *special form* `function` gemacht

Beispiel:

```
(function (lambda (x) (* x x)))
-->  #<Interpreted Function (unnamed) @ #x1c8237a>
```

Für `function` gibt es (analog wie für `quote`) ein Readermakro `#'`. Eine äquivalente Schreibweise für das Beispiel oben ist also: `#'(lambda (x) (* x x))`

Funktionsobjekte – Syntax der Lambda-Ausdrücke

Die Syntax eines Lambda-Ausdruck entspricht weitgehend der schon bekannten Syntax von **defun**.

(lambda *Parameterliste Funktionskörper***)**

Die Parameterliste spezifiziert dabei die formalen Parameter, die dann im Funktionskörper auftreten können. Der Funktionskörper besteht wiederum aus einer Folge symbolischer Ausdrücken, wobei der Wert der Funktion durch den zuletzt ausgewerteten Ausdruck bestimmt wird.

Beispiel:

```
(lambda (x y) (+ x y))

(lambda (l) (sum l))
```

Funktionsobjekte – Anwendung anonymer Funktionen

Die direkte Verwendung von Lambda-Ausdrücken, d.h. der Gebrauch anonymer Funktionen, bietet sich oft in solchen Fällen an, in denen eine bestimmte Berechnungsvorschrift nur lokal an einer Stelle benötigt wird.

```
(let ((zahlen '(33 6 27 44 79 72 94 51 91 93)))
   (sort zahlen #'(lambda (x y) (cond ((oddp x) t)
                                      ((oddp y) nil)
                                      (t t)))))
--> (93 91 51 79 27 33 94 72 44 6)
```


Funktionsobjekte – Funktionsaufruf

Mit Hilfe der Funktionen **funcall** und **apply** können die durch Lambda-Ausdrücke definierten Funktionen auch explizit angewendet werden. Dabei wendet **funcall** die durch den Wert des ersten Argumentes bezeichnete Funktion auf die Werte der restlichen Argumente an. Im Gegensatz dazu erwartet **apply** als letztes Argument immer eine Liste mit den (restlichen) Funktionsargumenten.

Beispiel:

```
(setq fu #'(lambda (x y) (list x y x)))

(funcall fu 'a 'h)  -->  (A H A)
(apply fu (list 'a 'h))  -->  (A H A)
(apply fu 'a '(h))  -->  (A H A)
(apply fu 'a 'h '()))  -->  (A H A)
```

Schmökobjekte – Anwendung anonymer Funktionen

Gegeben die Folgende Funktionsdefinitionen:

```
(defun fie (x)
  (function (lambda (y)
              (print (+ x y)))))
(defun foo (f)
  (funcall f 17)
  'ok)
```

Was passiert wenn folgende Form evaluiert wird `(foo (fie 42))`

Funktionen – `defun`

Bei der Definition einer Funktion mittels **`defun`** wird ein Funktionsobjekt erzeugt, das in dem Slot `symbol-funtion` reingehängt wird

```
(defun foo (x)
  (+ x x))
```

≡

```
(progn
  (setf (symbol-function 'foo)
    (function
     (lambda (x)
         (+ x x))))
  'foo)
```