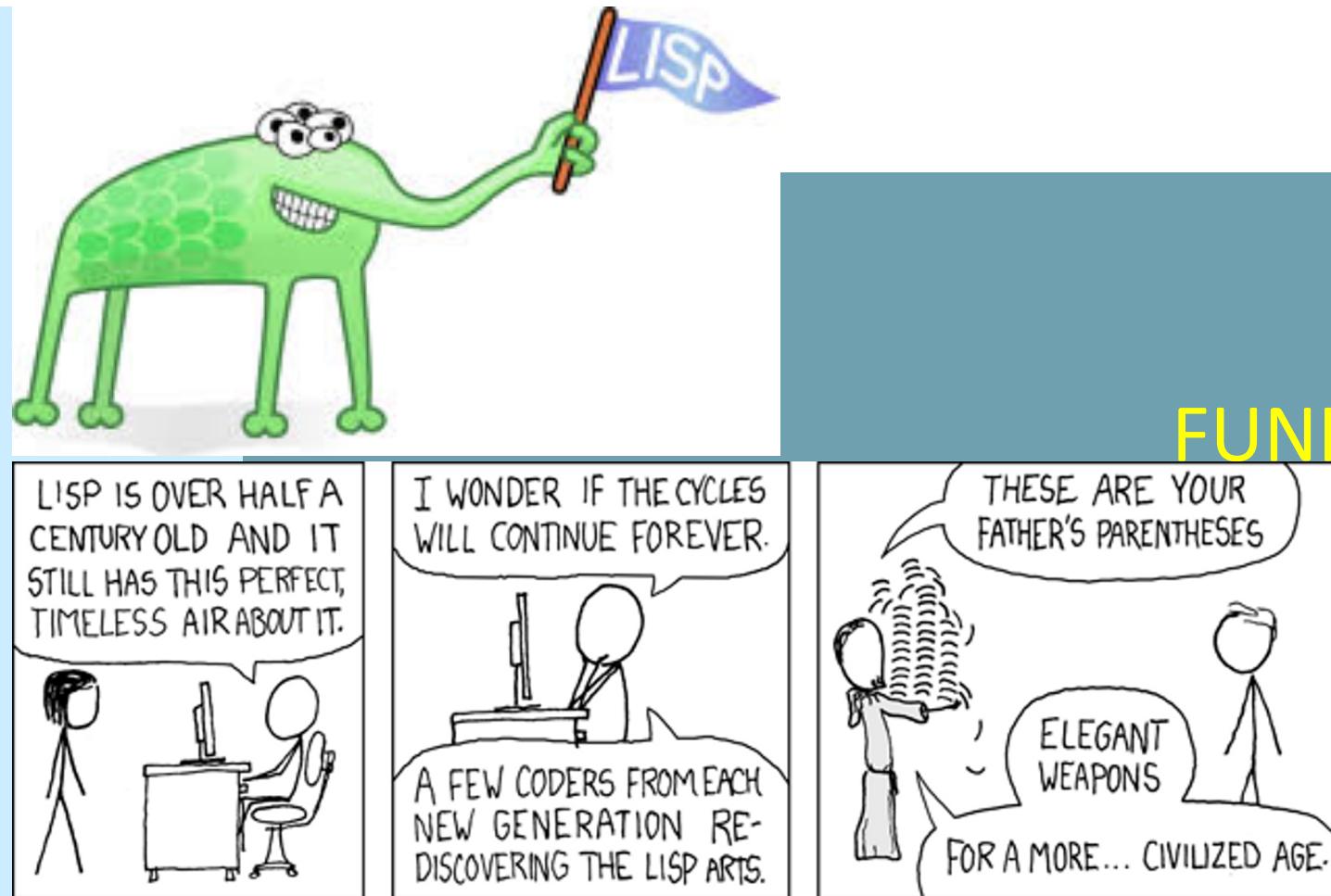


7. Kapitel

LISP

TEIL 3

FUNKTIONEN



Lernziele des Kapitels

2



- Unterschiede zwischen Special Forms, Funktionen und Makros verstehen.
- Benannte und unbenannte Funktionen (lambdas) definieren.
- Verschiedene Parameterarten kennenlernen.
- Lokale Funktionen definieren.
- Funktionen höherer Ordnung verstehen und anwenden können.
- Funktionen als Ergebnis eines Funktionsaufrufs definieren und anwenden können.

Funktionen, Special Forms, Makros

3

- Die dritte der im Kapitel 5 formulierten Regeln zur operationellen Semantik legt fest: bei der Evaluierung einer Liste (Funktionsaufruf) werden immer alle Elemente der Liste evaluiert (das Element zu der anzuwendenden Funktion, alle anderen zu den Argumenten der Funktion).

Das stimmt so nicht (ganz)!
- Ausnahme: Wenn QUOTE bzw. SETQ/ SETF als erstes Element einer Liste vorkommen \Rightarrow nicht alle Argumentbezeichner werden ausgewertet.
- QUOTE: wenn es notwendig ist, die Evaluierung des Argumentbezeichners zu verhindern.
- SETQ /SETF-Form: es wird immer das zweite Element eines Argument-Wert Paars ausgewertet, nicht aber das erste.

Funktionen, Special Forms, Makros

4

- Funktionen, die, wenn sie aufgerufen werden, nicht alle ihre Argumentbezeichner auswerten: ***special forms***.
 - ▣ Unterscheiden sich in vielen Fällen durch ihre Syntax von anderen Formen.
 - ▣ Es gibt in Common LISP etwa 30 special forms.
- Außer einer Funktion oder einer special form kann das erste Element einer Liste auch einen ***Makro*** bezeichnen.
- Makros werden nach besonderen Regeln evaluiert: Zunächst wird der Makro *expandiert*, und anschließend wird der aus der Expansion resultierende Ausdruck *evaluiert*.
- Es gibt in Lisp also
 - ▣ Funktionen
 - ▣ Special Forms
 - ▣ Makros.

Evaluation einer Liste – in 3 Stufen

5

3 Tests:

1. Kann das erste Element der Liste zu einer Special Form evaluiert werden?
2. Wenn nicht, bezeichnet es einen Makro?
3. Ist auch das nicht der Fall: handelt es sich um einen Funktionsbezeichner?
4. Ist keiner der Tests positiv, wird eine Fehlermeldung zurückgegeben.

Der Benutzer kann eigene Funktionen und Makros, nicht aber Special Forms definieren.

Arten von Funktionen

6

- Funktionen zur Listenverarbeitung (vordefiniert)
 - ▣ Konstruktoren. (Bsp. CONS, APPEND)
 - ▣ Selektoren. (Bsp. CAR, FIRST)
 - ▣ Modifikatoren (Bsp. SETF)
 - ▣ Sonstige (Bsp. LENGTH, MEMBER)
- Benutzerdefinierte Funktionen
 - ▣ Namenlose Funktionen
 - ▣ Funktionen mit Namen

Funktionen - benannt

7

- Können mittels **defun** neu definiert und an einen Namen gebunden werden. D.h. die Funktionsdefinition wird in der Funktionszelle des durch den Namen bezeichneten Symbols abgelegt.

Syntax:

(**defun** <Name der Funktion> (<Parameterliste>) <Funktionsrumpf>)

Name: beliebiges Symbol; bereits existierende Definition einer Funktion gleichen Namens wird überschrieben.

Parameterliste: formale Parameter

Funktionsrumpf: beliebige Folge symbolischer Ausdrücke

Wert der Funktion = Wert des zuletzt ausgewerteten symbolischen Ausdrucks.

Funktionen – Lambda-Ausdrücke

8

- Mittels **Lambda-Ausdruck** werden unbenannte (anonyme) Funktionen (kurz: lambdas) definiert.

Syntax:

Lambda-Ausdruck: (**lambda** lambda-list . Body)

oder

(**lambda** (<Parameterliste>) <Funktionsrumpf>) – Makro

Parameterliste: formale Parameter

Funktionsrumpf: beliebige Folge symbolischer Ausdrücke

Wert der Funktion = Wert des zuletzt ausgewerteten symbolischen Ausdrucks.

Es werden nicht alle Parameter des lambda-Ausdrucks ausgewertet.

- Aufruf eines Lambda-Ausdrucks (immer zusammen mit der Definition):

((**lambda** (<Parameterliste>) <Funktionsrumpf>) aktuelle Parameter)

Parameterliste / Lambdaliste

9

- LISP stellt einen sehr mächtigen Parameterübergabemechanismus zur Verfügung.
- Struktur der Parameterliste (auch Lambdaliste genannt):

```
({var}*  
[&optional {var | (var [initform [svar]])}*]  
[&rest var]  
[&key {var | ({var|{(keyword var)} [initform  
[svar])}* [&allow-other-keys]}  
[&aux {var | (var initform)}]*]  
)
```

- `var` und `svar` sind Symbole, `initform` ein beliebiger symbolischer Ausdruck.
- Jeder einzelne oder sogar alle Teile (= leere Parameterliste) dürfen leer sein.

Struktur der Parameterliste

10

- Elemente der Parameterliste:
 - entweder eine Parameterspezifikation oder
 - ein spezielles Schlüsselwort.
 - Schlüsselwörter: das erste Zeichen ist &.
- Funktionsaufruf:
 - Ersetzen der formalen Parameter der Reihe nach durch die entsprechenden Funktionsargumente.
 - Die Abarbeitung erfolgt dabei immer von links nach rechts.
 - Wichtig: Aufrufmuster und Parameterliste müssen miteinander vereinbar sein.

Obligatorische Parameter

11

- Alle formalen Parameter bis zum ersten Schlüsselwort bzw. bis zum Ende der Parameterliste sind notwendig und müssen beim Aufruf einer Funktion angegeben werden.
- Parameterspezifikation eines obligatorischen Parameters = Symbol, das als lexikalische Variable behandelt wird.

Optionale Parameter

12

```
[ &optional {var | (var [initform [svar]])}^ ]
```

- Optionale Parameter: können, müssen aber nicht beim Aufruf spezifiziert werden.
- Falls kein entsprechendes Argument vorhanden ist, wird `initform` ausgewertet und dieser Wert an den Parameter gebunden.
- Wurde `initform` nicht angegeben so ist in diesem Fall der Wert des Parameters gleich `nil`.
- Die Variable `svar` erhält den Wert `t` falls der optionale Parameter mit einem Argument gebunden wurde und `nil` falls `initform` ausgewertet werden musste.

[**&rest var**]

- Der Restparameter wird gebunden an eine (möglicherweise leere) Liste noch verbleibender Argumente.
- Damit lassen sich Funktionen definieren, die mit beliebig vielen Argumenten aufgerufen werden können.
- Wird der Restparameter mit Schlüsselwortparametern kombiniert, so werden dieselben verbleibenden Argumente bei der Verarbeitung der Schlüsselwortparameter noch einmal berücksichtigt .

Schlüsselwortparameter

14

```
[&key {var | ({var|{(keyword var)}} [initform  
[svar]])}* [&allow-other-keys]]
```

- Anderes Konzept zur Behandlung optionaler Argumente.
- Parameterzuweisung basiert dabei nur bedingt auf der Stellung der Argumente in der Aufrufstruktur.
- Dem entsprechenden Argument wird beim Aufruf ein Schlüsselwort vorangestellt.
- Dieses Schlüsselwort ergibt sich aus einem & unmittelbar gefolgt von dem Namen des formalen Parameters oder durch explizite Angabe eines Namens.

Key-Parameter: Beispiele

15

```
(lambda (&key vorname name (semester 1)) (list vorname name semester))  
  :name 'meier  :vorname 'klara)  
-> (KLARA MEIER 1)
```

```
(defun test (a b &key c d) (list a b c d))
```

```
(test 1 2)          --> (1 2 nil nil)
```

```
(test 1 2 :c 6)    --> (1 2 6 nil)
```

```
(test 1 2 :d 8)    --> (1 2 nil 8)
```

```
(test 1 2 :c 6 :d 8) --> (1 2 6 8)
```

```
(test 1 2 :d 8 :c 6) --> (1 2 6 8)
```

```
(test :a 1 :d 8 :c 6) --> (:a 1 6 8)
```

```
[&aux {var | (var initform)}*]
```

- Als letztes Argument einer Parameterliste kann, eingeleitet durch das Schlüsselwort auch eine Spezifikation lokaler Hilfsvariablen erfolgen.

Backquote-Parameter

17

- Backquote ` hat eine ähnliche Wirkung wie quote:
 - der nachfolgende Ausdruck wird nicht ausgewertet,
 - außer den Teilausdrücken, die auf ein Komma , folgen, diese werden ausgewertet.

```
(setq a 5)
```

```
(setq b 6)
```

```
(setq c `(a ,a b ,b))
```

```
-> (a 5 b 6)
```

```
`(a ist gleich ,a)
```

```
-> (a ist gleich 5)
```

```
(setq d `(a (b ,b) ,a))  
-> (a (b 6) 5)
```

```
(setq c `(a = ,a und b = ,b))  
-> (a = 5 und b = 6)
```

Lokale Funktionsdefinition / geschachtelte Funktionen

18

- Funktionen, die innerhalb einer anderen Funktion definiert werden und nur innerhalb dieser Funktion verwendet werden können.
- Stellt sicher, dass die so definierte Funktion von keiner anderen Funktion aufgerufen werden kann als der Funktion, innerhalb der sie definiert wurde.
- I.d.R. handelt es sich um eine Hilfsfunktion, die ausschließlich in diesem Kontext benötigt wird.
- Die einfachste Möglichkeit zur Generierung von lokalen Funktionen: Verwendung von Lambda-Ausdrücken.

Syntax:

```
FLET ( { ( Name Lambda-Liste {Form}* ) }*  
      {Form}*                                - Special Form
```

- Definition einer beliebigen Anzahl lokaler Funktionen – innerhalb einer defun-Anweisung.
- Definition besteht (vgl. DEFUN-Form) aus: Funktionsname, Lambda-Liste und dem den Funktionskörper bildenden Anweisungsblock.
- Geltungsbereich der Funktionen: FLET-Form, innerhalb der sie definiert wurden.

Lokale Funktionen - LABELS

20

Syntax:

```
LABELS( { ( Name Lambda-Liste {Form}* ) }*  
        {Form}* - Special Form
```

- Syntax = Syntax von FLET
- Geltungsbereich der Funktionen: LABELS-Form, innerhalb der sie definiert wurden + die in der Form enthaltenen Funktionsdefinitionen.
- Unterschied zwischen LABELS und FLET: Mit LABELS können rekursive Funktionen und sich wechselseitig aufrufende Funktionen definiert werden.

Funktionsparameter – Funktionen höherer Stufe

21

- Bisher betrachtet: Funktionen, die Zahlen, Symbole und Listen als Argumente nehmen.
- In LISP: Kaum Restriktionen bezüglich des Argument- und Wertebereichs von Funktionen → es ist z.B. möglich, Funktionen zu definieren, die als Argumente Funktionen nehmen.
- Solche Funktionen nennt man Funktionen höherer Stufe.
- Eine Funktion, die eine andere als Argument hat, ist eine Funktion zweiter Stufe.
- Die anzuwendende Funktion wird dynamisch zur Laufzeit bestimmt.

Funktionsparameter – Beispiel

22

□ Funktion mit zwei Parametern:

- Erster Parameter: Funktion
- Zweiter Parameter: Liste
- Die Funktion wird auf alle Top-Level Ausdrücke der Liste angewendet.
- Ergebnis: Liste der aus der Funktionsanwendung resultierenden Werte.
- `(anwenden 'first '((a) (b) (c)))` \Rightarrow (a b c)

Funktion

Liste, auf die Funktion
angewendet wird

Ergebnis

Funktionsparameter: Funktion aufrufen – Beispiel (Versuch 1)

23

```
(defun anwenden (funktion argumentliste)
  (if (endp argumentliste)
      ()
      (cons (funktion (first argumentliste))
            (anwenden funktion (rest argumentliste))))))
```

```
(anwenden 'first '((a) (b) (c)))
```

```
*** - EVAL: undefined function FUNKTION
```

- Ursache: Beim Aufruf einer Funktion gibt es bei der Parameterübergabe eine **Wertbindung** und **keine Funktionsbindung**; d.h. bei der Ausführung von (funktion (first argumentliste)) existiert für den Parameter funktion keine Funktionsbindung, d.h. es wird keine Funktion erwartet. funktion ist ein Symbol – keine Funktion !

Funktionsparameter: Funktion aufrufen – Beispiel (Versuch 2)

24

FUNCALL Funktion Argument^{*} - Funktion

- Bei Evaluierung von FUNCALL-Form wird Funktion auf die in der Form spezifizierten Argumente angewendet.
- Unterschied zu normalen Funktionsaufruf: in den Fällen, in denen die Funktion durch ein Symbol bezeichnet wird, wird hier auf die Wertbindung des Symbols zugegriffen.

```
(defun anwenden1 (funktion argumentliste)
  (if (endp argumentliste)
      ()
      (cons (funcall funktion (first argumentliste)) ; wendet die über den Wert des Symbols
            ; funktion gebundene Funktion an
            (anwenden1 funktion (rest argumentliste))))
```

1. Parameter wird als
Funktion betrachtet.

```
(anwenden1 'first '((a b c) (d e f)))
```

> (A D)

Argumente: Liste

Funktionsparameter: Funktion aufrufen – Beispiel (Versuch 3)

25

APPLY Funktion Argument⁺ - Funktion

- Erlaubt, eine andere Funktion auf eine beliebige Zahl von Argumenten anzuwenden:
 - ▣ Die Argumente werden als **Liste** übergeben.
 - ▣ Bei der Evaluierung einer APPLY-Form wird Funktion auf die in der Form spezifizierten Argumente angewendet.
 - ▣ Funktion = Lambda-Ausdruck oder
 - ▣ Funktion = Symbol.
 - Es wird die Funktionsbindung des Symbols verwendet.
- ▣ Achtung: Die Funktionsweise ist anders als bei `funcall`.

Apply - Beispiel

26

```
(defun anwenden2 (funktion argumentliste)
  (if (endp argumentliste)
      ()
      (cons (apply funktion (first argumentliste)) ; wendet die über den Wert des Sy
             ; funktion gebundene Funktion an
            (anwenden2 funktion (rest argumentliste)))))
```

```
(anwenden2 'first '((a b c) (d e f)))
```

> (A D)

Argumente: Liste von Listen

Beispiele zu funcall vs. apply

27

(funcall 'first '(a b c))

(apply 'first '((a b c)))

- Der Unterschied liegt in der Angabe der Parameterliste:
 - funcall: **Parameter** in der passenden Form für first – also als Liste
 - apply: **Parameterliste** in der passenden Form für first – also Parameterliste enthält Liste.
- Damit nicht der Benutzer der Funktion wissen muss, ob bei Verwendung eines Funktionsparameters mit funcall oder mit apply gearbeitet wird, muss das in der Funktion mit dem Funktionsparameter versteckt werden.

Beispiele zu funcall vs. apply

28

(funcall 'first '(a b c))

(apply 'first '((a b c)))

- Der Unterschied liegt in der Angabe der Parameterliste:
 - funcall: **Parameter** in der passenden Form für first – also als Liste
 - apply: **Parameterliste** in der passenden Form für first – also Parameterliste enthält Liste.
- Damit nicht der Benutzer der Funktion wissen muss, ob bei Verwendung eines Funktionsparameters mit funcall oder mit apply gearbeitet wird, muss das in der Funktion mit dem Funktionsparameter versteckt werden.

funcall vs. apply

29

```
(defun anwenden3 (funktion argumentliste)
  (if (endp argumentliste)
      ()
      (cons (apply funktion (list (first argumentliste)))
            ; wendet die über den Wert des Symbols
            ; funktion gebundene Funktion an
            (anwenden3 funktion (rest argumentliste))))))
```

```
(anwenden3 'first '((a b c) (d e f)))
```

Argumente für apply
werden hier in Liste gepackt

Wann nimmt man was?

- apply: die Parameter sind in Listen verpackt
- funcall: die Parameter kommen einzeln

- Noch ergänzen – insbesondere wenn man mit mapcar arbeitet

Funktionen als Ergebnisse von Funktionen

31

- Funktionen können als Wert eine Funktion liefern.
- Am Beispiel der Funktion **MAKE-POWER-OF-TWO-GENERATOR**, die als Wert eine Generator-Funktion (Generator) liefert.
 - ▣ Generator: Funktion, die bei jedem Aufruf ein neues Objekt aus einer bestimmten Folge von Objekten als Wert liefert.
- Die Funktion wird Generatoren erzeugen, die ausgehend von 2^1 bei jedem Aufruf die nächst höhere Potenz von 2 als Wert liefern.
- Wenn **POWER-OF-TWO** ein solcher Generator ist, dann soll er sich wie folgt verhalten:

(power-of-two) → 1

(power-of-two) → 2

(power-of-two) → 4

Funktionen als Ergebnisse von Funktionen – Beispiel Generatoren

32

; 1. Variante mit globaler Variablen + Funktion

```
(defvar *previous-power-of-two* 1)

(defun previous-power-of-two ()
  (setf *previous-power-of-two*
        (* *previous-power-of-two* 2)))
```

Ein Nachteil bei der Verwendung von globalen Variablen:

- ihr Wert kann durch andere Funktionen verändert werden
- die Definition mehrerer Generatoren eines Typs führt zu Konflikten bzw. erzwingt die Einführung weiterer globaler Variablen.

Wie kann ein Generator ohne Verwendung globaler Variablen definiert werden?

Funktionen als Ergebnisse von Funktionen – Beispiel Generatoren

33

Erster Schritt: Bindung einer (anonymen) Funktion an den Generator

; 2. Variante mit globaler Variablen
; + Bindung der anonymen Funktion an neuen Funktionsname

```
(setf (symbol-function 'power-of-two)
      #'(lambda ()
          (setf *previous-power-of-two*
                (* *previous-power-of-two* 2))))
```

Durch die Verwendung von SYMBOL-FUNCTION wird die anonyme Funktion in das Funktionswert-Feld von POWER-OF-TWO eingetragen.

Der Generator kann wie bei Verwendung einer DEFUN-Form ohne FUNCALL aufgerufen werden.

Ein Nachteil durch Verwendung der globalen Variablen bleibt erhalten.

Funktionen als Ergebnisse von Funktionen – Beispiel Generatoren

34

Dritter Schritt: Ersetzen der globalen Variablen PREVIOUS-POWER-OF-TWO durch eine lokale Variable gleichen Namens.

; 3. Variante nur mit an Funktionsnamen gebundene anonyme Funktion
; die Variable previous-power-of-two ist nur lokal sichtbar!

```
(setf (symbol-function 'power-of-two)
  (let* ((previous-power-of-two 1))
    #'(lambda ()
      (setf res previous-power-of-two)
      (setf previous-power-of-two
            (* previous-power-of-two 2))
      res)))
```

Aufruf des Generators: Zugriff auf die innerhalb der durch die LET-Form erzeugten Umgebung geltende Bindung der lokalen Variable PREVIOUS-POWER-OF-TWO.

Andere Funktionen können nicht mehr auf sie zugreifen.

→ Es können verschiedene Generatoren dieses Typs generiert werden, ohne dass es beim Variablenzugriff zu Konflikten kommt.

Funktionen als Ergebnisse von Funktionen – Beispiel Generatoren

35

2 Generatoren

```
(setf (symbol-function 'gen1)
  (let* ((previous-power-of-two 1))
    #'(lambda ()
      (setf res previous-power-of-two)
      (setf previous-power-of-two
        (* previous-power-of-two 2))
      res)))
```

```
(setf (symbol-function 'gen2)
  (let* ((previous-power-of-two 1))
    #'(lambda ()
      (setf res previous-power-of-two)
      (setf previous-power-of-two
        (* previous-power-of-two 2))
      res)))
```

Liefern unabhängig voneinander die korrekten Ergebnisse.

Nachteil: Code-Verdopplung.

Funktionen als Ergebnisse von Funktionen – Beispiel Generatoren

36

Generator mit reset

```
; generator mit reset

(setf generators
  (let ((previous-power-of-two 1))
    (list
      #'(lambda ()
          (setf previous-power-of-two 1))
      #'(lambda ()
          (setf res previous-power-of-two)
          (setf previous-power-of-two
                (* previous-power-of-two 2))
          res)))))

(setf (symbol-function 'power-of-two-reset) (first generators))
(setf (symbol-function 'power-of-two-value) (second generators)))
```

Funktionen als Ergebnisse von Funktionen – Beispiel Generatoren

37

Auswahl der Generator-Funktion über Parameter

```
(setf (symbol-function 'generator-with-dispatch-procedure)
  (let* ((previous-power-of-two 1)
    (reset-procedure
      #'(lambda () (setf previous-power-of-two 1)))
    (value-procedure
      #'(lambda ()
        (setf previous-power-of-two
          (* previous-power-of-two 2))))
  )
  #'(lambda (accessor)
    (if (eq 'reset accessor)
      (funcall reset-procedure)
      (funcall value-procedure)))))

(generator-with-dispatch-procedure 'reset) ; reset generator
(generator-with-dispatch-procedure 'next)  ; next power of 2
```

Funktionen als Ergebnisse von Funktionen – Beispiel Generatoren

38

Generierung von Generatoren

```
(defun make-power-of-two-generator ()
  (let* ((previous-power-of-two 1)
         (reset-procedure
          #'(lambda () (setf previous-power-of-two 1)))
         (value-procedure
          #'(lambda () (setf previous-power-of-two
                               (* previous-power-of-two 2))))
        )
    #'(lambda (accessor)
        (if (eq 'reset accessor)
            (funcall reset-procedure)
            (funcall value-procedure))))
```

Generatoren erzeugen

```
(setf (symbol-function 'generator1) (make-power-of-two-generator)
      (symbol-function 'generator2) (make-power-of-two-generator))
```

Generatoren aufrufen

(generator1 'reset)	; > 1
(generator1 'next)	; > 2
(generator2 'reset)	; > 1
(generator1 'next)	; > 4
(generator2 'next)	; > 2

Lambda-Ausdrücke (nochmal): Motivation

39

LAMBDA-Ausdruck: Definition einer lokalen Funktion ohne Namen.

Beispiel:

```
(defun add-2 (x) (+ x 2))
```

```
(mapcar 'add-2 '(4 6 2 8)) -> (6 8 4 10)
```

Wenn eine solche spezielle Funktion, wie add-2 nur an einer Stelle benötigt wird, lohnt sich eine separate Definition, die global gilt, eigentlich nicht.

Lambda-Ausdrücke: Motivation

40

Alternative:

```
(mapcar (defun add-2 (x) (+ x 2)) '(4 6 2 8))
-> 6 8 4 10)
```

Auch in diesem Fall gilt die Definition von add-2 global, deshalb so nicht sinnvoll.

Alternative 2:

```
(mapcar (function (lambda (x) (+ x 2))) '(4 6 2 8))
-> (6 8 4 10)
```

lambda (variable*) {form}* - *Lambda-Form*

- Eine Liste mit erstem Element LAMBDA heißt LAMBDA-Ausdruck.
- Ein LAMBDA-Ausdruck entspricht einer Funktionsdefinition mit defun, aber ohne Namen.
- Wenn die durch den Lambda-Ausdruck bezeichnete Funktion ausgeführt wird, werden - nach Bindung der Parameter an die beim Funktionsaufruf verwendeten Argumente - die Formen (Funktionsrumpf) sequentiell evaluiert und der Wert der letzten Form als Funktionswert zurückgegeben.
- Ein Lambda-Ausdruck ist keine Form; d.h. er kann nicht sinnvoll evaluiert werden. Er bezeichnet eine Funktion.

Lambdas - Eigenschaften

42

- Die spezielle Funktion "FUNCTION" muss in manchen Situationen verwendet werden, wenn der entsprechende Ausdruck nicht ausgewertet werden soll (analog Quote).
- Ohne vorherige Funktionsdefinition sind jederzeit Funktionsaufrufe durch Lambda-Ausdrücke möglich:

((lambda (x) (+ x 3)) 23) → 26

- Der Ausdruck (function (lambda ...)) kann durch
#'(lambda ...) bzw. einfach
(lambda ...) abgekürzt werden.
- D.h. der function-Aufruf kann in den meisten Fällen entfallen.

Funktion bekannt machen - FUNCTION

43

FUNCTION Funktionsbezeichner - [Special Form]

- Rückgabewert: das durch Funktionsbezeichner bezeichnete Funktionsobjekt.
- Ist der Funktionsbezeichner ein Symbol, evaluiert es zu seiner Funktionsbindung.
- Ist es ein Lambda-Ausdruck, wird eine Lexical-Closure erzeugt, und die durch sie repräsentierte Funktion wird unter Beachtung der für lexikalische Bindung geltenden Regeln ausgeführt.
- Stellt sicher, dass auch dann auf die Funktionsbindung des Symbols zugegriffen wird, wenn das Symbol nicht in Funktionsposition – als erstes Element einer Liste – vorkommt.
- Statt FUNCTION lässt sich auch das Makro-Zeichen #' verwenden:
`(function symbol) = #' symbol`