Letzte/Diese Vorlesung

Letzte Woche:
- Conszellen
- Assoc-Listen

Heute:
- `defstruct`
- Makro

Verbund-artige Strukturen – Motivation

Probleme:

- Zugriffsfunktionen müssen für jede Komponente explizit definiert werden.
```
(defun student-matrikelnummer (student)
 (second student))
```
- Änderungen der Struktur ziehen Änderungen im Code nach sich.
- Datenabstraktion wird nicht unterstützt (z.B. muß für jede(n) neue(n) StudentInn gleiche komplexe Datenstruktur explizit formuliert werden).

Erwünscht: Unterstützung bei der Definition komplexer Datenstrukturen durch das LISP-System.

Verbund-artige Strukturen – Die CommonLisp-Lösung: `defstruct`

- **defstruct**–Makro zur Definition verbund–ähnlicher Datenstrukturen (*structures*) mit benannten Komponenten (sog. *slots*).
- Dabei automatische Erzeugung von Instantiierungs–, Test–, Zugriffs–, und Zuweisungsfunktionen.
- Ermöglicht insbesondere die Definition eigener Datentypen.

```
(defstruct student    (defstruct student
  name                   (name "Noname" :type string)
  vorname                (vorname "Noname" :type string)
  alter                  (alter 25 :type integer)
  wohnort)               (wohnort *sbr* :type string ))
```


Verbund-artige Strukturen – `defstruct`

Auswirkungen eines defstruct-Aufrufs

- automatische Erzeugung einer *Konstruktorfunktion*
- automatische Erzeugung einer *Kopierfunktion*
- automatische Definition eines entsprechenden Datentypen und Erzeugung einer Testfunktion
- automatische Erzeugung von Zugriffsfunktionen für die spezifizierten Komponenten der Struktur
- automatische Erweiterung von **setf** zur Modifikation der Werte der einzelnen Komponenten

Verbund-artige Strukturen – `defstruct`

Konstruktorfunktionen Mit **defstruct** werden eigene Datenstrukturen definiert, aber noch keine konkreten Ausprägungen dieser Strukturen. Zur Definition aktueller Daten dient die Konstruktorfunktion.

- Bezogen auf das Beispiel der Struktur `student` wird eine Funktion **make-student** erzeugt, die bei Aufruf eine Datenstruktur mit den entsprechenden 4 Komponenten aufbaut.
- Beim Aufruf der Konstruktorfunktion ist es möglich Initialwerte für einzelne Komponenten anzugeben. Dazu werden die entsprechenden Komponenten und die Werte mittels Schlüsselwortparametern angegeben.

```
(make-student :name "Claudius") -->
#S(STUDENT :NAME "Claudius" :VORNAME "noname" :ALTER 25 :WOHNORT "Saarbrcken")
```


Verbund-artige Strukturen – `defstruct`

Kopierfunktion

- **copy-student** legt eine Kopie der als Argument übergebenen Struktur an. Die Werte sämtlicher Komponenten der Originalstruktur und der neu erzeugten Kopie sind identisch.

Testfunktionen

- **student** ist ein Symbol, das zum Namen eines Datentyps wird; das generelle Typprädikat **typep** ist damit auch auf **student** anwendbar.
  Beispiel: `(type-p (make-student) 'student) --> T`
- **student-p** ist ein spezielles Typprädikat, das testet ob ein Objekt vom Typ **student** ist.
  Beispiel: `(student-p (make-student)) --> T`

Verbund-artige Strukturen – `defstruct`

Zugriffsfunktionen

- Automatische Erzeugung einer Zugriffsfunktion für jede einzelne Komponente.
- **student-alter** wird als Funktion mit einem Argument definiert. Ist das Argument vom Typ **student**, so liefert die Funktion den Wert der Komponente `alter` zurück. Entsprechendes gilt für die übrigen Komponenten der Struktur `student`.

```
? (student-alter (first *students*)) -->
25
```


Verbund-artige Strukturen – `defstruct`

Änderungsfunktionen

- Mit **setf** können Komponenten-Werte von Instanzen verändert bzw. gesetzt werden.

```
(setf (student-alter student1) 27)  -->  27
(setf (student-name student1)
      (concatenate 'string (student-name student1)
                   "-" (student-name student2)))
  --> "Mustermann-Schulze"
```

Verbund-artige Strukturen – `defstruct`

Syntax von defstruct

(defstruct $\{$*name* $|$ (*name* $\{$*option*$\}^+$)$\}$
  [*doc-string*]
  $\{$*slot-name* $|$ (*slot-name default-init* $\{$*slot-option*$\}^*$)$\}^+$)

Verbund-artige Strukturen – `defstruct`

Syntax von defstruct

name: Das Symbol wird zum Namen der Datenstruktur und (möglicherweise) des neuen Datentypen. *name* wird als Ergebnis des **defstruct**-Aufrufs geliefert.

option: Optionen zur feineren Kontrolle von **defstruct**

slot-name: Symbol, das als Bezeichner für die entsprechende Komponente dient.

default-init: Symbolischer Ausdruck, der jeweils ausgewertet und verwendet wird, falls die Komponente bei der Erzeugung einer Instanz nicht explizit initialisiert wurde.

slot-option: Die Option **:read-only** *boolean* (*boolean* ungleich **nil**) bedingt, daß die Komponente nach Aufbau der Struktur nicht modifiziert werden kann.
Mit **:type** *type-specifier* wird deklariert, daß Werte der Komponente stets den angebenen Datentyp haben. *type-specifier* wird nicht evaluiert.

Verbund-artige Strukturen – `defstruct`

Optionen für defstruct

(:constructor *name***)** Das Symbol *name* dient als Name für eine Konstruktorfunktion. Die **:constructor**-Option kann mehrfach angegeben werden. Ist *name* gleich **nil** so wird keine Konstruktorfunktion angelegt.

(:constructor *name lambda-list***)** Durch explizite Angabe einer Parameterliste lassen sich auch automatisch Konstrukturfunktionen erzeugen, die einer anderen Aufrufstruktur genügen. Die in *lambda-list* auftretenden Parameter werden dabei jeweils mit den Komponenten gleichen Namens identifiziert.

```
...  (:constructor create-person  (vorname name &optional (alter 30))) ...
...  (:constructor create-homburger (vorname name &optional alter &aux (wohnort *homburg*

(setq person2 (make-person :vorname "Udo" :name "Schulze"))
(setq person3 (create-person "Uli" "Schulze"))
(setq person4 (create-homburger "Matz" "Bauer"))
```


Verbund-artige Strukturen – Optionen für `defstruct`

(:conc-name *prefix***)** Statt des Strukturnamens (gefolgt von einem Bindestrich) wird das Symbol *prefix* als Präfix in den Namen der Zugriffsfunktionen verwendet. Ist *prefix* gleich **nil** so werden direkt die Namen der Komponenten als Bezeichner für die Zugriffsfunktionen benutzt.

(:copier *name***)** Das Symbol *name* dient als Name für die Kopierfunktion. Ist *name* gleich **nil** so wird keine Kopierfunktion angelegt.

(:predicate *name***)** Das Symbol *name* dient als Name für das Typprädikat. Ist *name* gleich **nil** so wird kein Typprädikat erzeugt.

```
(defstruct (person (:conc-name pers-) (:copier klone) (:predicate person?))
  name vorname alter wohnort)
(setq person1 (make-person :name "Hrmzz") person1b (klone person1))
(pers-name person1)  -->  "Hrmzz"
(person? person1b)  -->  T
```

Verbund-artige Strukturen – Optionen für `defstruct`

defstruct unterstützt auch einfache Vererbung. Diese wird benutzt, um eine neue Struktur als Erweiterung (*Spezialisierung*) einer bereits bekannten Struktur zu definieren.

(:include *structure***)** Das Symbol *structure* muß Name einer bereits definierten Struktur sein. Alle Komponenten dieser Struktur werden automatisch in die neu zu definierende Struktur integriert; die Namen der neuen Komponenten müssen daher eindeutig sein. Sämtliche Zugriffsoperationen die für *structure* erzeugt wurden sind auch auf alle Ausprägungen der neuen Struktur anwendbar.

(:include *structure* {*slot-description*}***)** Zusätzlich kann jede ererbte Komponentenbeschreibung explizit überschrieben werden, um z.B. andere Initialwerte festzulegen.

Verbund-artige Strukturen – Optionen für `defstruct`

```
(defstruct (astronaut
             (:include student (alter 45)))
  (raumschiff *orion*) helmgroesse raumhafen)

(setq *spock*
      (make-astronaut :name "Spock" :alter 427 :raumschiff *enterprise*))
(setq student (make-student :name "O'Mally"))

(student-name *spock*)  -->  "Spock"
(astronaut-name *spock*)  -->  "Spock"
(typep *spock* 'student)  -->  T
(typep student 'student)  -->  T
(typep student 'astronaut)  -->  NIL
(astronaut-name student)  -->  ??undefiniert??
```

Verbund-artige Strukturen – Optionen für `defstruct`

Die tatsächliche interne Realisierung, d.h. die zugrundeliegende Repräsentation einer Struktur ist gemäß dem Prinzip der Datenabstraktion für den Benutzer nicht sichtbar und außerdem von der jeweiligen Implementation abhängig.
Bei manchen Anwendungen kann jedoch das interne Format eines Strukturtyps bereits fest vorgegeben sein. Um auch in solchen Fällen eine konzeptuelle Beschreibung mittels des **defstruct**-Mechanismus zu ermöglichen, kann bei der Definition der Struktur angegeben werden, welcher Datentyp zur internen Repräsentation gewählt werden soll.

(:type *type***)** Der angegebene *type* (mögliche Werte sind **list**, **vector**, **(vector** *element-type***)**) wird zur internen Repräsentation der Struktur verwendet. Die Angabe dieser Option verhindert die Integration in die COMMON LISP Typhierarchie; d.h. der Name der neuen Struktur ist kein zulässiger Typbezeichner für die Funktion **typep**. Instanzen der Struktur sind vielmehr vom Typ *type*. In diesem Sinne wird eine solche Struktur als *konzeptueller* Datentyp bezeichnet.

Makro – Motivation

Wir haben schon (hoffentlich) erfahren, daß wir mit unserem bisherigen Wissen z.B. ein eigenes `push()` *nicht* schreiben können.

```
;;
? (defparameter *hacker-stack* '(steele ww))
? (defun my-push (x stack)
    (setq stack (cons x stack)))

? (my-push 'oe *hacker-stack*) -->
(OE STEELE WW)
;; Wobei
? *hacker-stack* -->
(STEELE WW)
```

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 `push`!

```
? (defparameter *hacker-stack* '(steele ww))
? (defmacro my-push (x stack)  (list 'setq stack (list 'cons x stack)))

? (my-push 'oe *hacker-stack*) -->
;; Expand
  (LIST 'SETQ STACK (LIST 'CONS X STACK)) -->
  (LIST SETQ *HACKER-STACK* (LIST CONS 'OE *HACKER-STACK*)) -->
  (SETQ *HACKER-STACK* (CONS OE *HACKER-STACK*))
;; Eval
(OE STEELE WW)
;; Ausserdem:
? *HACKER-STACK* -->
(OE STEELE WW)
```


Makro – Weitere 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

Beispiel

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 Tippse hat die Aufgabe, *alle* StudentInnen in unsere Datenbank einzutragen. Für jede(n) StudentInn muss er den Variable `*students*` umsetzen mit z.B.:

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

Nach schon 13 StudentInnen geht seine `quote`-Taste kaputt!!! Können wir ihr 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 hauptfach))
              '*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 l2)
--> (l1 (1) ((3)) 2 3 l2)
```

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 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*)))
```