Letzte/Diese Vorlesung

Iteration – loop... Letzte Woche

Debugging Diese Woche

Makros

Debugger

Was ist ein **Debugger**?

Was ist ein **Debugger** in Lisp?

Interaktives Debugging

`break` Es ist in Lisp erlaubt sogenannte *breakpoints* zu setzen, bzw die Funktion `break` aufzurufen.

`error` Auch `error` kann explizit aufgerufen werden. Falls ein error signalisiert wird wird, abhängig von dem Programm, entweder der Debugger aufgerufen, oder den Fehler von Funktionen wie `ignore-error, handler-case,...` abgefangen und das Programm läuft weiter.

`trace` Man kan das Makro `trace` aufrufen um zu erfahren mit welchen Parameterwerte eine Funktion aufgerufen wurde, und welchen Wert sie zurückgibt.

`step` Durch ein Aufruf mit `step` zu wrappen (z.B. `(step (! 42))`) kann man Schritt für Schritt sein Programm durchgehen.

Merke: Das Verhalten des Debuggers ändert sich, wenn die Funktion kompiliert wird. Z.B. wird der Funktionskörper nicht mehr angezeigt.

Interaktives Debugging – `break`

Wenn man in den Debugger gerät, gibt es ein Reihe von Befehlen, die man benutzen kann um zu erfahren was schief gelaufen ist, und vor allem, wie man weiter kommen kann. Die wichtigsten sind:

COMMAND	ABBR	DESCRIPTION
:continue	:cont	continue from a continuable error
:dn / :up		move down/up the stack 'n' frames, default 1
:local	:loc	print the value of a local (interpreted or compiled) variable
:pop		pop up 'n' (default 1) break levels
:prt		:pop-and-retry the last expression which caused an error
:reset	:res	return to the top-most break level
:restart	:rest	restart the function in the current frame
:return	:ret	return values from the current frame
:set-local	:set-l	set the value of a local variable
:zoom	:zo	print the runtime stack

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_likely) (print 'should_work))

NOT_VERY_LIKELY   ;; 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 engeren 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 1. Schritt
(LIST 'COND (LIST (EQL 2 3) 14) (LIST 'T 42)) -->
;; Expand 2. Schritt
(COND ((EQL 2 3) 14) (T 42)) -->
;; Eval
42
```

Makro – Mehrere Verwendungszwecke

Man kann die Eigenschaft, daß ein Makro Anweisungen gegen andere Anweisungen austauscht, ausnutzen, um die Geschwindigkeit eines Programms zu beschleuningen – Ein guter Kompiler expandiert die Makros schon beim Übersetzen → Man „verliert“ also Funktionsaufrufe.

Makro – Weitere Verwendungszwecke

Beispiel

In einer fruüeheren Übungsaufgabe haben wir einen ADT “`student`” mit einem Konstruktor, `make-student()`, und drei Selektoren, `student-name, student-matrikel` und `student-hauptfach` geschrieben. Beim “profiling” des Programms sind wir dahintergekommen, daß das Programm sehr viel Zeit beim Aufrufen der drei Selektoren verbraucht – wir wollen aber nicht so lange auf die Ergebnisse 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* Studenten in unsere Datenbank einzutragen. Für jede(n) Student(In) muss der die Variable `*students*` umsetzen, z.B. mit:

```
(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 matrikel (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)
--> (l1 (1) ((3)) 2 3)
```


Makro – back-quote, un-quote und splice

Heimaufgabe (freiwillig): Wie schreibt man mit `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*)))
```