Letzte/Diese Vorlesung

Debugging Letzte Woche

Makros

Mehr über Makros Diese Woche

Vielleicht: Mehr über Debugging

Makro – Wiederhohlung

Der Makro-Mechanismus in LISP erlaubt es, beliebige Prozeduren zu definieren, die bestimmte Anweisungen vor der Auswertung bzw. vor dem *Evaluieren* 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 **defmacro** 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**.

Für eine schöne (!) Abendstunde im Lieblingssessel: **defmacro** ist selbst ein Makro!!

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 – `macroexpand`

Um uns zu vergewissern, daß unsere Makrodefinitionen das tuhen, was wir uns vorgestellt haben, stellt Common-Lisp eine Funktion (diese Funktion wird auch von dem Compiler benutzt) die ein Form expandiert – **`macroexpand`** – zur Verfügung.

Beispeil:

```
(macroexpand '(my-if (eql 3 2) 14 42)) -->
(COND ((EQL 2 3) 14) (T 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

Beispeil

Ein ADT „`student`" besteht u.A. aus einem Konstruktor, `make-student`, und drei Selektoren, `student-name, student-matrikel` und `student-hauptfach`. 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, und entscheiden uns dewegen, die drei Selektoren als Makros umzuschreiben.

Makro – Weitere Verwendungszwecke

```
;; Konstruktor
(defun make-student (name matrikel hauptfach)
  (list hauptfach matrikel name))

;;      Funktion                                  Makro
;;;
```

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*)))
```

Kommentar: Wir nutzen also aus, daß ein Makro seine Argumente *nicht* evaluiert

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

Wir können jetzt die obigen Beispiele mithilfe von back-, un-quote und splice schreiben:

```
;;      Funktion                                  Makro
;;;
```

Makro – Beispiel

Schreibe ein eigenes `if`! (revisited)

```
(defmacro my-if (test then else)
  (list 'cond (list test then) (list 't else)))
```

vs

```
(defmacro my-if (test then else)
  `(cond (,test ,then) (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 – „Higher order Functions"

In CommonLisp sind Makros keine Funktionen. Insbesonderes können Makros nicht als Funktionale Argumente für z.B. `apply, map,...` verwendet werden. In solchen Situationen „existiert die Liste die das Original-Aufruf" nicht mehr (es kann sogar nicht mehr existieren, da ihre Argumente schon evaluiert sind) und wir müssen andere Techniken benutzen...

Makro – „Higher order Functions"

Mit der CommonLisp-Funktion **reduce** kann eine Liste mit hilfe einer Funktion reduziert werden. Z.B. kann man eine Liste mit Zahlen addieren:

```
(reduce #'+ '(1 2 3 4 5 6 7 8 42)) --> 78
```

Wir wollen jetzt feststellen ob alle Objekte in einer Liste nicht `NIL` sind. Eine Mglichkeit wäre **reduce** und **and** zu verwenden. Da aber **and** ein Makro ist können ist folgende Code nicht erlaubt:

```
(reduce #'and '(1 2 nil foo "a string" ...)) -->
#<Error: AND names a macro -- bad arg for function.>
```

Wie machen wir?

Lösung:

```
(reduce #'(lambda (x y) (and x y)) '(1 2 nil foo "a string" ...)) --> NIL
```