Praktische Algorithmen der Bioinformatik und Computerlinguistik mit Lisp

Makros, Funktionen höherer Ordnung und Fehlerbehandlung

24.06.2021

Übersicht: Weiterführende Lisp-Themen.

Lisp syntaktisch:

- Makros

Lisp-Funktionen als Objekte

- Closures
- Continuations
- Memoisation

Fehlerbehandlung

- Erzeugung und Signalisieren von Fehlern
- Behandlung von Fehlern
- Das `Condition`-System

Makros: Definition

Mit Makros kann Lisp-Code *textuell* verändert werden. Makros eignen sich:

- Zur Definition neuer Kontrollstrukturen.
- Als Benutzerinterface für eingebettete Sprachen.
- Vorsicht: Makros nicht als Pseudo-Inline benutzen.
- viele eingebaute Makros: `defun`, `when`, `dolist`, `dotimes`, `with-open-file`, ...

Neue Makros werden mit `defmacro` definiert, das ähnlich wie defun aussieht:

- Liste von Parametern
- Makro-Body

Makros: Definition

Beispiel:

```
(defmacro our-when (test consequence)
  (list 'if test consequence nil))
```

Makros: Expansion

Auswertung von Makros:

Bei der Auswertung von Listen wird das erste Element betrachtet. Ist dieses ein Symbol, so muss es entweder eine Funktion oder ein Makro benennen. Falls das Symbol ein Makro benennt, so wird

1. die ganze Liste makroexpandiert; die Argumente werden dabei nicht evaluiert!
2. das expandierte Makro wird evaluiert

Makros: Expansion

Beispiel: explizite Makroexpansion mit `macroexpand`.

```
> (defmacro our-when (test consequence)
    (list 'if test consequence nil))
OUR-WHEN
> (macroexpand '(our-when t 'a))
(IF T (QUOTE A) NIL)
T
> (macroexpand '(our-when (atom 'x) (setf *flag* t)))
(IF (ATOM (QUOTE X)) (SETF *FLAG* T) NIL)
T
> (our-when (atom 'x) (setf *flag* t))
T
```

Backquote

Sowohl Quote ’ als auch Backquote ‘ schützen vor Ausführung. Bei Backquote kann man die Ausführung jedoch mit Komma wieder anschalten. Backquote ist nützlich zur Definition von Makros.
Hinter einem Backquote steht das Muster einer Datenstruktur (Form). Dieses Muster wird kopiert, bis auf die mit Komma gekennzeichneten Teile. Diese werden ausgewertet und das Ergebnis in das Muster eingesetzt.

- Backquote (`): Beginn eines Musters
- Komma (,): Evaluiere die nachfolgende Form und nimm das Ergebnis.
- Komma-at (,@): Die nachfolgende Form sollte bei der Auswertung eine Liste ergeben, die dann aufgespalten wird.

Backquote

Beispiele:

```
(defmacro our-when (test consequence)
  `(if ,test ,consequence nil))
```

```
(let ((x 'foo)
      (y '(a b c d)))
  `((1 ,x) (2 ,'x) (3 ,y) (4 ,@y)))
==> ((1 FOO) (2 X) (3 (A B C D)) (4 A B C D))
```

Das `let`-Makro

1. **Versuch:**

```
(defmacro our-let (bndngs &rest body)
  `((lambda ,(mapcar #'first bndngs) ,body)
    ,(mapcar #'second bndngs)))

(macroexpand '(our-let ((a 3) (b 4))
                       (list a b)))
==> ((LAMBDA (A B) ((LIST A B))) (3 4))
    T
```

In obigem Lambda-Ausdruck sind eindeutig zu viele Klammern!

Das `let`-Makro

2. Versuch

```
(defmacro our-let (bndngs &rest body)
  `((lambda ,(mapcar #'first bndngs) ,@body)
    ,@(mapcar #'second bndngs)))
```

```
(macroexpand '(our-let ((a 3) (b 4))
                       (list a b)))
==> ((LAMBDA (A B) (LIST A B)) 3 4)
    T
```

Makros: Mögliche Fehlerquellen

Trotz ähnlichem Aussehen sind Makros keine Funktionen.
Insbesondere gibt es Schwierigkeiten, wenn

- Makro-Definitionen rekursiv sind (≈ unendliche Expansionen)
- Makro-Definitionen als Funktionsparameter übergeben
  werden in der Form: `(mapcar #'my-macro lst)`

Makros: Mögliche Fehlerquellen

Zu beachten ist, dass Argumente evtl. mehrmals ausgewertet werden:

```
> (defun foo (x)
    (format T "Calling foo~%")
    x)
FOO
> (defmacro square (arg)
    `(* ,arg ,arg))
SQUARE
> (macroexpand '(square (foo 3))) (* (FOO 3) (FOO 3))
T
> (square (foo 3))
Calling foo
Calling foo
9
```

Makros: Mögliche Fehlerquellen

Dies ist ineffizient und fehleranfällig bei Funktionen mit Nebeneffekten. Besser:

```
> (defmacro square (arg)
    `(let ((arg-val ,arg))
       (* arg-val arg-val)))
SQUARE
> (macroexpand-1 '(square (foo 3)))
(LET ((ARG-VAL (FOO 3))) (* ARG-VAL ARG-VAL))
> (square (foo 3))
Calling foo
9
```

Makros: Mögliche Fehlerquellen

Potenzielles Problem bei dieser Vorgehensweise: Variablenbindung

```
> (defmacro sum (arg1 arg2)
    `(let* ((x ,arg1) (y ,arg2)) (+ x y)))
SUM
> (macroexpand-1 '(sum (+ 1 2) (* 3 4)))
(LET* ((X (+ 1 2)) (Y (* 3 4))) (+ X Y))
```

Das ist in Ordnung - aber hier wird das äußere `x` überdeckt:

```
> (let ((x 3)) (sum 2 x))
4
```

Lösung: Generierung eines eindeutigen Namens mit `(gensym)` - siehe Lisp-Lehrbücher

Continuations

Im Folgenden haben Funktionen ein weiteres Argument, die sog. Continuation. Anstatt einfach Werte zurückzuliefern, wird die Continuation-Funktion mit den berechneten Werten aufgerufen.

```
(defun fact-cps (n cont)
  (if (= n 0)
    (funcall cont 1)
    (fact-cps (- n 1)
      #'(lambda (v) (funcall cont (* n v))))))

(trace fact-cps)

(fact-cps 4 #'identity) ==> 24
```

Continuations

Trace

```
1 Enter FACT-CPS 4 #<Compiled-Function IDENTITY 52C396>
|2 Enter FACT-CPS 3 #<Interpreted-Function (LAMBDA (V)
                     (FUNCALL CONT (* N V))) 125E00E>
| 3 Enter FACT-CPS 2 #<Interpreted-Function (LAMBDA (V)
                      (FUNCALL CONT (* N V))) 125E106>
| |4 Enter FACT-CPS 1 #<Interpreted-Function (LAMBDA (V)
                       (FUNCALL CONT (* N V))) 125E1DE>
| |  5 Enter FACT-CPS 0 #<Interpreted-Function (LAMBDA (V)
                        (FUNCALL CONT (* N V))) 125E2C6>
| |  5 Exit FACT-CPS 24
| |4 Exit FACT-CPS 24
| 3 Exit FACT-CPS 24
|2 Exit FACT-CPS 24
1 Exit FACT-CPS 24
```

Continuations

Bemerkung:

- Wir haben also den gleichen Effekt wie bei tail-rekursiven Programmen, dass der Aufrufstack nicht mehr für weitere Berechnungen benutzt wird.
- (rekursive) Funktionen lassen sich durch eine Menge von Transformationsregeln in eine äquivalente CPS-Form bringen:

1. Erweitere die Parameterliste um eine zusätzliche Variable cont, die das (Teil-) Ergebnis aufnimmt
2. Ersetze Wertrückgaben `val` durch `cont val`
3. Ersetze rekursive Aufrufe durch entsprechende Aufrufe der CPS-Variante der Funktion, die als letztes Argument eine mit Hilfe von cont gebildete Continuation enthalten.

Continuations

Bemerkung (Fortsetzung):

- Transformation rekursiver Funktionen in CPS-Form kann als automatischer Optimierungsschritt in der ersten Phase einer Kompilation eingesetzt werden.
- Mit Hilfe von Continuations lassen sich grosse Teile der Lisp-Semantik auf einheitliche Weise erklären.

Multiple Values

In CommonLisp gibt es viele Funktionen (z.B. `get-decoded-time`, `floor`, `ceiling`, `truncate`, ...), die mehrere Resultate liefern:

```
(get-decoded-time) ==> 55 38 15 25 8 1994 3 T -1
(floor 10 3) ==> 3 1
```

Diese Resultate können z.B. mit Hilfe des Makros multiple-value-bind gebunden werden an lokale Variablen:

```
(multiple-value-bind (sec min hr day mon yr)
          (get-decoded-time)
  (list day mon yr))
==> (25 8 1994)
```

Multiple Values

Andere Möglichkeiten:

```
(multiple-value-list (get-decoded-time))
  ==> (43 48 15 25 8 1994 3 T -1)
```

Funktionen, die mehrere Resultate haben, können z.B. mit Hilfe von values und values-list geschrieben werden:

```
(values 1 2 3)           ==> 1 2 3
(values-list '(1 2 3)) ==> 1 2 3

(defun rd (x)
  (values (floor x) (- x (floor x))))

(rd 3.5)       ==> 3 0.5
```

Memoisierung

Caching von Resultaten früherer Berechnungen

```
(defun fib (n)
  (if (<= n 1) 1
    (+ (fib (- n 1)) (fib (- n 2)))))
```

Problem: dieselben Berechnungen werden immer und immer wieder durchgeführt.

```
(time (fib 30))          ==> ...608.43 seconds...
```

Ansatz: benutze Funktion `fib`, um eine Funktion zu bauen, die vorher berechnete Resultate speichert, und diese benutzt, anstatt sie noch einmal zu berechnen:

Memoisierung

```
(defun memo (fn)
  (let ((table (make-hash-table)))
    #'(lambda (x)
      (multiple-value-bind
        (val found-p) (gethash x table)
        (if found-p
        val
        (setf (gethash x table) (funcall fn x)))))))
```

Der Ausdruck `(memo #'fib)` produziert eine Funktion, die Resultate zwischen Berechnungen abspeichert.

Memoisierung

```
(setf *memo-fib* (memo #'fib))

(trace fib)

(funcall *memo-fib* 3) ==> 3 Seiteneffekt: ...
(funcall *memo-fib* 3) ==> 3
```

Beim zweiten Aufruf wird `fib` nicht mehr aufgerufen. Problem: rekursive Aufrufe von `fib` werden bisher nicht gespeichert:

```
(funcall *memo-fib* 2) ==> 2 Seiteneffekt: ...
```

Memoisierung

Lösung:

```
(defun memoize (name)
  (setf (symbol-function name)
    (memo (symbol-function name))))
```

memoize ändert den Wert einer Funktion zu einer Memo-Funktion.

```
(memoize 'fib)
(trace fib)
(time (fib 30))
  ==> ... User Run Time = 0.03 seconds ...
```

Bemerkung: Beim Memoisierungsprozess muss man immer abwägen zwischen Speicherplatzverbrauch und Laufzeit.

Fehler

Fehlererzeugung:

- `checktype`
- `assert`
- `error`
- `cerror`

Fehlerbehandlung:

- Restarts
- Handler

Das Condition-System

Unfreiwillige Fehlererzeugung

```
(defun fak0 (n)
  (if (= n 0)
    1
    (if (= n 1)
      1
      (* n (fak0 (1- n))))))
```

Eingabe eines nicht-numerischen Werts führt zu einem Laufzeitfehler:

```
USER(182): (fak0 'a)
Error: EXCL::=_2OP:
  'A' is not of the expected type 'NUMBER'
  [condition type: TYPE-ERROR]
[1] USER(183): :bt
Evaluation stack:

= <-
  FAK0 <- EVAL <- TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP
       <- TPL:START-INTERACTIVE-TOP-LEVEL
```

Fehlererzeugung durch `checktype`

```
(checktype var typespec)
```

1. `checktype` prüft, ob in der Variablen var ein Datum mit dem Typ `typespec` steht.
2. Bei positivem Ausgang der Prüfung wird die nächste Anweisung ausgeführt.
3. Bei negativem Ausgang wird eine Fehlermeldung erzeugt.

```
(defun fak1 (n)
  (check-type n (integer 0 *))
  (if (= n 0) 1
    (if (= n 1) 1
      (* n (fak1 (1- n))))))
```

Beispiele für `checktype`

```
USER(186): (fak1 'a)
Error: the value of N is A,
  which is not of type (INTEGER 0 *).
  [condition type: TYPE-ERROR]

Restart actions (select using :continue):
 0: supply a new value for N.
[1] USER(187): :bt
Evaluation stack:

LET* <-
  [... EXCL::%EVAL ] <- PROGN <- FAK1 <- EVAL
  <- TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP
  <- TPL:START-INTERACTIVE-TOP-LEVEL

[1] USER(188): :cont 0
Type a form to be evaluated: 7
5040
USER(189):
```

Fehlererzeugung durch `assert`

```
(assert test-form (var1 ...))
```

1. `assert` berechnet `test-form`
2. Wenn `test-form` den Wert `Nil` hat, wird eine Fehlermeldung erzeugt.
3. Wenn `test-form` nicht `Nil` liefert, wird mit der nächsten Anweisung fortgefahren.

```
(defun fak2 (n)
  (assert (and (integerp n) (<= 0 n)) (n))
  (if (= n 0) 1
    (if (= n 1) 1
      (* n (fak2 (1- n))))))
```

Beispiel für `assert`

```
USER(190): (fak2 'a)
Error: the assertion (AND (INTEGERP N) (<= 0 N)) failed.
  [condition type: SIMPLE-ERROR]

Restart actions (select using :continue):
 0: retry assertion with new value for N.
[1] USER(191): :cont 0
the old value of N is B. do you want to supply a new
value? y
Type a form to be evaluated: 4
24
USER(192):
```

Fehlererzeugung durch `error` und `cerror`

Lisp unterscheidet zwischen “fatalen” Fehlern und solchen, bei denen eine Fortsetzung der Bearbeitung möglich ist:

- `(error format-string &rest args)` erzeugt einen fatalen Fehler
- `(cerror c-format-string e-format-string &rest args)` erzeugt einen Fehler, der eine Fortsetzung erlaubt.

Der Ausgabestring der Fehlermeldung wird wie durch `format` erzeugt.

Fehlererzeugung durch `error` und `cerror`

Beispiel:

```
(defun fak3 (n)
  (if (integerp n)
    (if (<= 0 n)
      (fak0 n)
      (progn
        (cerror "Compute (fak3 -(~D))"
                "Negative argument: ~D" n)
        (fak0 (- n))))
    (error "Non-integer argument: ~D" n)))
```

Beispiele für `error` und `cerror`

```
USER(194): (fak3 3)
6

USER(195): (fak3 -3)
Error: Negative argument: -3
Restart actions (select using :continue):
 0: Compute (fak3 -(-3))
[1c] USER(196): :cont 0
6

USER(197): (fak3 'a)
Error: Non-integer argument: A
[1] USER(198): :res
USER(199):
```

Fehlerbehandlung durch `Restarts`

`Restarts` sind Aufsetzpunkte, die nach Auftreten eines Fehlers aktiviert werden.

```
(defun fak4 (n)
  (typecase n
    ((integer 0 *) (fak0 n))
    (float
   (restart-case
       (error "Factorial of ~D is undefined." n)
     (nil ()
       :report "Truncate and compute factorial"
       (fak4 (truncate n)))
     (nil ()
       :report "Round and compute factorial"
       (fak4 (round n)))))
(T (error "Non-integer argument: ~D" n))))
```

Fehlerbehandlung durch `Restarts`

(Ungefähres) Format: `restart-case` *`expression cases`*

Tritt in *expressions* ein Fehler auf, wird der Debugger aktiviert, wobei die *cases* als Aufsetzpunkte angeboten werden. Der mit dem gewählten Aufsetzpunkt verbundene Ausdruck wird ausgewertet und bildet das Ergebnis von restart-case.

Restarts in `action`

```
USER(200): (fak4 2.6)
Error: Factorial of 2.6 is undefined.
  [condition type: SIMPLE-ERROR]

Restart actions (select using :continue):
 0: Truncate and compute factorial
 1: Round and compute factorial
[1] USER(201): :cont 0
2

USER(202): (fak4 2.6)
Error: Factorial of 2.6 is undefined.
  [condition type: SIMPLE-ERROR]

Restart actions (select using :continue):
 0: Truncate and compute factorial
 1: Round and compute factorial
[1] USER(203): :cont 1
6
```

Das `Condition`-System

Das `Condition`-System bietet einen objekt-orientierten Ansatz zur Fehlerbehandlung, der in CLOS integriert ist:

- Fehlerarten können als Klassen definiert werden
- Beziehungen zwischen Fehlerarten können durch Vererbung zwischen Klassen ausgedrückt werden.
- Bei Auftreten von konkreten Fehlersituationen können Instanzen von Fehlerklassen erzeugt werden.

Das `Condition`-System

Ein kleiner Überblick über einige vordefinierte Fehlerklassen:

- `condition`
  - `simple-condition`
  - `serious-condition`
    - `error`
      - `simple-error`
      - `arithmetic-error` (division-by-zero oder floating-point-overflow)
    - `storage-condition`
  - `warning`

Das `Condition`-System

Neben den vordefinierten Klassen können vom Benutzer eigene Klassen definiert werden. Dies geschieht durch das Makro `define-condition`.

Beispiele:

```
(define-condition my-condition (simple-error) ())

(define-condition my-list-condition (my-condition)
  (list)
  (:report (lambda (condition stream)
    (format stream
      "List ~A has length not equal 2"
      (my-list-condition-list condition)))))
```

Fehlerbehandlung durch `handler-case`

```
(handler-case expression
  (type-1 (var-1) form-1)
  (type-2 (var-2) form-2)
  ...)
```

Mit `handler-case` lassen sich Fehler behandeln, die bei der Auswertung von expression auftreten. In Abhängigkeit von der dabei erzeugten Fehlerklasse wird in einen der folgenden Ausdrücke verzweigt und `form-i` ausgewertet, wobei `var-i` an die *condition* gebunden wird, die signalisiert wurde.

Fehlerbehandlung durch `handler-case`

Resultat von `handler-case` ist:

- Das Ergebnis von `form-i`, falls die Ausnahme behandelt wird.
- Die (nicht behandelte) Ausnahme, falls kein zu der Ausnahme passender Typ in der Fallunterscheidung vorkommt.

Kann eine Ausnahme nicht lokal behandelt werden, so kann sie evtl. durch einen anderen Handler tiefer im Aufruf-Stack abgefangen werden.