Common Lisp / Parallel Lisp

Dennis Bliefernicht
Nicolau Caseiro

Seminar Programmiersprachenkonzepte
SS 2004

Lisp

- entwickelt seit 1958 durch John McCarthy am Dartmouth College
- ANSI-Standard (ANSI X3.226-1994 (R1999))
- Verschiedene Implementationen
- Interaktive Umgebung: toplevel

Lisp

- Nicht notwendigerweise interpretiert
- Funktionale Programmiersprache
- Verschachtelte Klammerterme, Präfix-Notation: **`(- (/1500 50) (* 2 4))`**

```
(defmethod make-cursor-for ((collection list))
  #'(lambda (message)
      (case message
    (:next
     (let ((r (first collection)))
       (setf collection (cdr collection))
       (values r (null collection))))
    (:finished-p
     (null collection)))))
```

Ausdrücke und Auswertung

- Auswertung eines Lisp-Ausdrucks: erst Parameter von links nach rechts, dann Werte an Funktion übergeben
- Spezialoperatoren: eigene Auswertungsregeln!
- Die meisten Operatoren in Lisp sind in Lisp geschriebene Funktionen
- Beispiel:

```
> (- (/ 1500 50) (* 2 4))
22
```

Ausdrücke und Auswertung

- Spezialoperator **`quote`** / **`'`** : liefert das Argument unverändert zurück
- ```
  > (quote foo)
  FOO
  > (quote (x y (+ 2 3)))
  (X Y (+ 2 3))
  ```

- **`eval`**: wertet Liste als Programm aus
  ```
  > (eval (quote (+ 2 3)))
  5
  ```
- **`coerce`** / **`compile`**
  ```

Operationen auf Listen

- Funktion **`list`**: baut Liste aus Argumenten

```
> (list 'die (+ 10 2) 'apostel)
DIE 12 APOSTEL
```

- **`nil`** / **`()`** als leere Liste
- **`cons, car, cdr`**

```
> (cons 'a '(b c d))
(A B C D)
> (cons 'a (+ 2 3))
(A 5)
```

- Beispiel:

```
> (car (cdr (cdr '(a b c d))))
C
```

Definition neuer Funktionen

- **`(defun name (param-list) exp1 exp2 exp3...`**
  Ausdrücke werden sequentiell abgearbeitet

```
(defun third (x)
            (car (cdr (cdr x))))
> (third '(a b c d))
C
```

- Spezialoperator **`function`** / **`#'`** liefert eine Referenz auf die Funktion zurück

```
> (function +)
#<Compiled-Function + 17BA4E>
> (apply (function +) '(1 2 3))
6
```

Lambda-Operator

- Lambda: ohne explizite Namensgebung
  `(lambda param-list exp1 exp2 exp3 ...)`

- Beispiele:

```
> ((lambda (x) (* x 2)) 12)
24

> (funcall (lambda (x) (* x 2)) 12)
24
```

Ein- / Ausgabe

- format / read: Funktionen mit Seiteneffekten!
- format ähnlich aufgebaut wie printf:
- **Beispiel:**

```
> format t "~A plus ~A ergibt ~A.~%" 2 3
(+ 2 3))
2 plus 3 ergibt 5
NIL
```

- read als Standardeingabefunktion: liefert genau die Eingabe zurück

```
> (read)
42
42
```

Auswahl

- IF-Anweisung:
  ```
  (if test-p then-exp else-exp)
  ```
- Beispiel:
  ```
  (defun ist-liste (x)
                 (if (listp x)
                     "Ist eine Liste!"
                     "Ist keine Liste!"))
  > (ist-liste '(a b c))
  Ist eine Liste!
  ```
- Muß ein Spezialoperator sein!

Variablen und Zuweisungen

- **`let`**: setzt eine lokale Variable, sie ist nur innerhalb des 'let' sichtbar.
  `(let ((x 1) (y 2) (z (+ 2 3))) exp1 exp2 exp3 ...)`
- **`(defparameter *glob* 99)`**
  Globale Variable, überall sichtbar
- Meist verwendet: **`setf`**. Implizite Deklaration im Programmtext

Closures

- Mit let deklarierte Variable nur innerhalb des let sichtbar
- Falls Funktion innerhalb des let definiert wird, so kann sie auch außerhalb darauf zugreifen.
- Beispiel:

```
(setf fn (let ((i 3))
                #'(lambda (x) (+ x i))))
> (funcall fn 2)
5
```

Wiederholung

- **do**-Schleife

```
(do (var init update)
        (test result)
        (body))
```

- Spezialfall: durch Liste iterieren: **dolist**

```
(defun our-length (lst)
        (let ((len 0))
                (dolist (obj lst)
                        (setf len (+ len 1)))
                len))
```

Blockstrukturen

- **`(progn exp1 exp2 exp3 ...)`**
  Sequentielle Abarbeitung, letzter Rückgabe-wert zählt

- **`(block name exp1 exp2 exp3 ...)`**
  Rücksprung im Body möglich mit
  `(return-from name 'ergebnis)`

- **`(tagbody exp1 exp2 exp3 ...)`**
  Atome als Ausdrücke als Label, springen mit
  `(go label)`

Datentypen

- Nicht Variablen, sondern die Werte haben Typen ("manifest typing")
  Hierarchisches Typsystem, Abfrage mit typep
- jeder Bezeichner ist an alles bindbar
- Mehrarbeit für Funktionen, Abhilfe: optionale Deklaration mit

  `(declaim (type int *count*))`
  (globale Variable)

  `(declare (type int count))`
  (lokale Variable, dies als Ausdruck in Body)

Referenzen

- Referenz auf Objekte: Variablen sind Pointer auf Werte, erst Werte haben Typen

- Objekte auf Identität prüfen:

```
> (eql (cons 'a nil) (cons 'a nil))
NIL
> (equal (cons 'a nil) (cons 'a nil))
T
```

--> "If it prints the same."

Effizienz

- 'Consing' ist ein großer Teil üblicher Programme, viel Aufwand auf dem Heap
- automatische Garbage Collection: Objekte ohne Referenz werden gelöscht
- Abhilfe: Bessere Datenstrukturen nutzen nach Entwicklungsphase

Arrays und Vektoren

- Arrays: bis zu (min) 7 Dimensionen mit jeweils bis zu (min) 1023 Elementen

```
(setf arr (make-array '(2 3) :initial-element nil))
```

- Zugriff: `(aref arr 0 0)`
- Vektoren

Structures

- structure: Vektor, bei dem Funktionen impliz-it mitdeklariert werden

  `(defstruct punkt x y)` definiert auch
  - **`point-p,`**
  - **`copy-point,`**
  - **`point-x,`**
  - **`point-y`**

- Bei Deklaration füllen:
  (setf p (make-point :x 23 :y 42))

Hashtables

- **`> (setf ht (make-hash-table))`**
- Lesezugriff nach einem Schlüssel:

```
> (gethash 'farbe ht)
NIL
NIL
```

- Schreibzugriff nach einem Schlüssel:

```
> (setf (gethash 'farbe ht) 'rot)
ROT
> (gethash 'farbe ht)
ROT
T
```

OOP: CLOS

- CLOS ist das OOP-Subsystem in Common Lisp (Common Lisp Object System)
- Grundidee: nicht Methoden *in* Klassen schreiben, sondern Methoden *für* Klassen
  - Klassen stellen *nur die Struktur* der Objekte da, nicht die möglichen Operationen

OOP: Klassen 1

- *defclass* definiert eine Klasse
  - spezifiziert wird nur der Name und die enthaltenen Variablen

```
(defclass name (superklassen...) (variablen...))
```

  - Beispiel:

```
(defclass rectangle (geo-object) (height width))
```

- Erstellen von Instanzen mit Hilfe von *make-instance*
  Beispiel:

```
(setf rect (make-instance 'rectangle))
```

OOP: Klassen 2

- Zugriff auf Variablen über *slot-value*:

```
(setf (slot-value rect 'width) 10)
```

- Alternativ: Zugriffsfunktionen in der Klasse benennen
  - Alternativen: Vollzugriff (:accessor), nur Lesen (:reader), nur schreiben (:writer)

```
(defclass rectangle (geo-object)
   ((height :accessor rectangle-h)
    (width  :accessor rectangle-w)))

(setf rect (make-instance 'rectangle))
(setf (rectangle-h rect) 10)
```

OOP: Klassen 3

- Weitere Eigenschaften von Objektvariablen:
  - *:initarg* benennt den Parameter zur Initialisierung
  - *:initform* bestimmt den default-Wert
  - *:allocation :class* bestimmt Klassenvariablen

```
(defclass circle ()
  ((radius :accessor circle-radius
           :initarg :radius
           :initform 1)
   (center :accessor circle-center
           :initarg :center
           :initform (cons 0 0))))

(setf c (make-instance 'circle :radius 3))
```

OOP: Superklassen 1

- Vererbung bedeutet Vererbung von Variablen, sowie Aufbau einer Typhierarchie
- CLOS ermöglicht Mehrfachvererbung
  - Problem: wenn eine Variable in den Oberklassen mehrfach definiert ist, welche ist gültig?
- Bei mehrfach vorhandenen Variablen wird von links nach rechts durch die Superklassen gegangen

OOP: Superklassen 2

- Klassen, die von mehreren Superklassen Elternklasse sind, werden zuletzt betrachtet
- Beispiel:

```
(defclass subsub () (...))
(defclass sub1 (subsub) (...))
(defclass sub2 (subsub) (...))
(defclass class (sub1 sub2) (...))
```

- Suchreihenfolge: class, sub1, sub2, subsub

OOP: Methoden

- Methoden werden mit *defmethod* definiert (außerhalb der Klasse):

```
(defmethod name (parameter...) . body)
```

- Verschiedene Varianten

```
(defmethod foo (param1) ...)
(defmethod foo ((param1 rectangle)) ...)
(defmethod foo ((param1 (eq 'BAR)) ...)
```

- Es wird jeweils die spezialisierteste Variante aufgerufen (Spezialisierungshierarchie wie Variablenhierarchie bei Mehrfachvererbung)

OOP: Auxiliary Methods

- Auxiliary Methods ermöglichen zusätzliche Behandlung
  - :before-Methoden werden VOR dem eigentlichen Aufruf gestartet
  - :after-Methoden werden NACH dem eigentlichen Aufruf gestartet
  - :around-Methoden ersetzen den eigentlichen Aufruf, können diesen aber durch *call-next-method* wieder ausführen
    - Kettenbildung durch mehrere around-Methoden ist möglich, dann von der spezialisiertesten zur allgemeinsten Methode

OOP: Generics

- Generics kombinieren alle Definitionen einer Methode in der gesamten Typhierarchie

```
(defgeneric price (x) (:method-combination +))
(defclass jacket () ())
(defclass trousers () ())
(defclass suit (jacket traousers) ())

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)

> (price (make-instance 'suit))
550
```

- Erlaubt für +, and, append, list, max, min, nconc, or, progn

OOP: Kapselung

- Das CLOS-System enthält keine Kapselung in den Klassen (wie z.B. private/protected/ public in Java)
- Kapselung wird über Packages und Kapselnde Methoden realisiert:

```
(defpackage "FOO"(:use "COMMON_LISP")
                 (:export "incr"))
(in-package foo)

(defclass klasse () ((wert :initform 0)))
(defmethod incr ((k klasse))
   (incf (slot-value k 'wert)))
```

QLisp

- Parallelität und Lisp:
  - Parallelität leicht zu finden, aber schwer zu kontrollieren
- In den 80ern geschaffen, um Parallelität in Lisp möglichst einfach auszunutzen
- Später von der ARPA/DARPA unterstütztes Projekt zur Implementation eines QLisp-Systems

QLisp: Designziele

- *Gemeinsamer Speicher* zwischen Prozessen
- Grad der Parallelität zur *Laufzeit* bestimmbar
- Nur Erweiterungen schaffen, die nötig sind
  - Bisherigen Lisp-Konstrukten, wenn möglich, zusätzliche Bedeutung in Mehrprozessumgebungen geben, anstatt neue Konstrukte zu schaffen
- QLisp-Programme sollen auch in Einprozessumgebungen funktionieren

QLisp: Future-Konzept

- Prozesse geben direkt anstatt des Ergebnisses eine sog. *future* zurück
  - Eine *future* ist ein Versprechen, später das Ergebnis zur Verfügung zu stellen
- Solange der Wert einer *future* nicht benötigt wird, kann sie ganz normal verwendet werden, während der berechnende Prozess noch arbeitet
- Wird der Wert einer Future benötigt, so wird auf den berechnenden Prozess gewartet, sofern dieser nicht schon abgeschlossen ist

QLisp: Bedingungen/Queues

- **Q**Lisp: basiert auf Warteschlangen (Queues)
  - Wenn ein Prozessor nichts zu tun hat, holt er den nächsten Auftrag aus der Warteschlange
  - Dynamische Änderung der Prozessorzahl problemlos möglich
- Designziel: Parallelität zur Laufzeit bestimmen
  - Jeder Befehl der Prozesse erzeugt enthält einen Bedingungsparameter (prop), false = kein neuer Prozess, true = neuer Prozess
  - Hilfsfunktion: *qemptyp* (ist die Warteschlange für Prozesse zur Zeit leer)

QLisp: Spawn-Konstrukt

- Einfachste Primitive zum Erzeugen eines Prozesses:

```
(spawn prop expression)
```

- Gibt eine *future* zurück und startet einen neuen Prozess, der die Berechnung durchführt
- spawn behält die Umgebung bei, die bei der Prozesserstellung vorhanden war

QLisp: qlet-Konstrukt

- Äquivalent zu *let*, aber Parallel:

```
(qlet prop ((var1 expr1) (var2 expr2) ...) . body)
```

- Startet Subprozesse für alle Ausdrücke, wartet, bis alle Ausdrücke berechnet, dann wird body-Teil ausgewertet
- *eager*-Variante
  - Evaluiert *prop* zu *EAGER*, so werden nur *futures* an die Variablen gebunden, der body-Teil wird sofort gestartet

QLisp: Beispiel zu qlet

```
(defun parallel-fac (n tiefe)
  (labels
    ((prod (m n tiefe)
      (if (= m n)
        m
        (let ((h (floor (+ m n) 2)))
          (qlet (> tiefe 0)
                ((x (prod    m    h (- 1 tiefe)))
                 (y (prod (+ h 1) n (- 1 tiefe))))
            (* x y))))))
    (prod 1 n tiefe)))
```

QLisp: qprogn-Konstrukt

- Äquivalent zum progn-Konstrukt:

```
(qprogn prop (expr1) (expr2) ...)
```

- Wertet alle Subausdrücke in eigenen Prozessen aus

QLisp: qlambda-Konstrukt 1

- qlambda realisiert einen auf Nachrichten wartenden Prozess:

```
(qlambda prop (...) . body)
```

- erzeugt einen Prozess, der ab sofort auf Aufrufe wartet (Aufrufe werden als Nachrichten empfangen
- Ein Aufruf gibt sofort eine *future* zurück, der Prozess wird also *asynchron* angestoßen
- Varianten für lokale Prozesse: qflet/qlabels

QLisp: qlambda-Konstrukt 2

- Ein qlambda-Prozess arbeitet alle eingehenden Aufrufe in einer Warteschlange ab (einen Aufruf zur Zeit)
  - Synchronisationsmittel (auf zu schützende Elemente nur innerhalb eines qlambda-Prozesses zugreifen)

- *eager*-Variante
  - Evaluiert *prop* zu *EAGER, so* beginnt der Prozess sofort eine Berechnung und stoppt, sobald Parameter benötigt werden oder ein Ergebnis feststeht

QLisp: Kooperative Prozesse 1

- Mehrere Prozesse an einem Ergebnis rechnen lassen
- Subprozesse erzeugen Zwischenergebnisse, die am Ende mittels einer *combine*-Funktion miteinander verrechnet werden
- Realisierung: *future* als Zentralobjekt, Berechnende Prozesse werden an die *future* “angehängt”

QLisp: kooperative Prozesse 2

```
(defun parallel-minimum (data)
  (let ((fut-min (spawn t :combine
                          #'(lambda (wert1 wert2)
                              (if (< wert1 wert2)
                                wert1
                                wert2)))))
    (let ((h (floor (length data) 2)))
      (spawn t :future fut-min
               (apply #'min (subseq data 0 h)))
      (spawn t :future fut-min
               (apply #'min (subseq data h))))
    (realize-future fut-min))))
```

QLisp: kill-process

- kill-process bekommt eine *future*, dazugehörige Prozesse werden beendet

```
(kill-process future)
```

  - Ist die *future* eine “normale” *future* (z.B. durch einen spawn- oder qlet-Aufruf, so wird der Prozess beendet
  - Ist die *future* Zentralpunkt von kooperativen Prozessen, so werden alle beteiligten Prozesse beendet
  - Handelt es sich um einen *qlambda*-Aufruf, so wird der Aufruf aus der dazugehörigen Warteliste entfernt

QLisp: Abbruch durch throw

- Erfolg innerhalb eines *catch*-Blockes ein *throw*, so werden alle durch *qlet/qprogn* gestarteten Subprozesse beendet
- Wird ein *catch*-Block normal verlassen, so werden ebenfalls alle eventuell noch laufenden Subprozesse beendet
- Alternative *qcatch*:
  - bei einem *throw* ebenso Prozessabbruch
  - Bei normalem Verlassen des Blockes wird auf alle Subprozesse gewartet

QLisp: sonstige Primitiven

- *qwait* wartet auf alle Subprozesse des enthaltenen Ausdruckes

```
(qwait expression)
```

- *suspend-process/resume-process* halten einen Prozess an und setzen ihn fort

```
(suspend-process future)
(resume-process future)
```

QLisp: PMI-Funktionen 1

- Problemstellung: 4 parallele Prozesse, 3 (A,B,C) davon versorgen den vierten (D) mit Daten

- *qlambda* ermöglicht die Konstruktion der Prozesse, aber:
  - Wie synchronisieren, dass Prozess D nur dann anläuft, wenn A, B und C ihre Daten weiter-gegeben haben?
  - Wie mehrere Instanzen von D modellieren?

QLisp: PMI-Funktionen 2

- Lösung: PMI(partially, multiply invoked)-Funktionen
- PMI-Funktionen bekommen ausschließlich benannte Parameter
- Ein Aufruf einer PMI-Funktion kann alle oder einen Teil der Parameter übergeben
- Solange nicht alle Parameter übergeben wurden, gibt die Funktion eine unrealisierte *future* zurück
- Sobald alle Parameter vorhanden sind, beginnt die Auswertung

QLisp: PMI-Funktionen 3

- Konstrukte um PMI-Funktionen zu verwenden: *pmi-defun/pmi-qflet*

- Beispiel:

```
> (pmi-defun add (x y) (:summand :summand) (+ x y))
ADD
> (add-up :summand 1 :summand 2)
3
> (add-up :summand 4)
<FUTURE#1>
> (add-up :summand 3)
7
```

QLisp: PMI-Funktionen 4

- Gefärbte Funktionsaufrufe ermöglichen mehrere Instanzen einer Funktion:

```
> (add-up :color 1 :summand 1)
<FUTURE#1>
> (add-up :color 2 :summand 2)
<FUTURE#2>
> (add-up :color 2 :summand 3)
5
> (add-up :color 1 :summand 5)
6
```

QLisp: Zusammenfassung

- QLisp ermöglicht dynamische Parallelität (zur Laufzeit bestimmbar, durch Warteschlangen variable Parallelität)
- Einfache Grundkonstrukte ermöglichen verschiedene Formen der Parallelität
  - dennoch mächtig genug um alle Aufgaben zu erledigen
- Synchronisation auch gemeinsame Daten durch *qlambda*-Konstrukte ermöglicht

Literatur

- Paul Graham, ANSI Common Lisp, Prentice-Hall 1996
- T. Ito, R.H. Halstead, Jr. (Eds.), Parallel Lisp: Languages and Systems, Springer-Verlag 1990
- http://www.apl.jhu.edu/~hall/lisp.html
- http://www.dreamsongs.com/Qlisp.html
  - Ron Goldman, Richard P. Gabriel, Qlisp: Parallel Processing in Lisp
  - Richard P. Gabriel, John McCarthy: Qlisp
- http://www.cs.indiana.edu/~tanaka/GEB/fmvParLisp/
- http://www.apl.jhu.edu/~hall/Lisp-Notes/Good-Lisp-Style.ps