(Common Lisp)

Ein Vortrag von Matthias Schütz und Andreas Klein

auf dem UUGRN FixMe Treffen (22.02.2008)

Aufbau des Vortrags

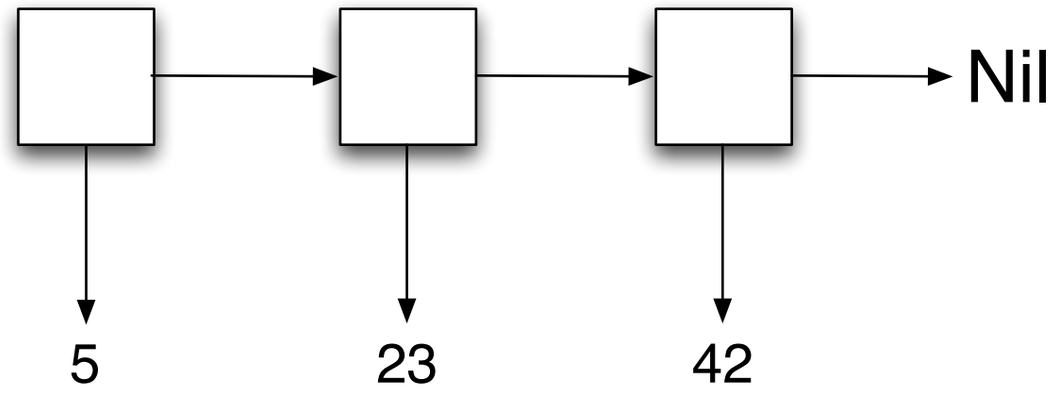
- Kurze Einführung in die Sprache
  - Listen
  - Funktionen
  - Macros
  - Objektorientierung mit CLOS (Common Lisp Object System)
- Vorurteile die wir zu zerstreuen versuchen:
  - Lisp ist altmodisch
  - Man kann ausschließlich funktional programmieren
  - Die Klammersyntax ist seltsam

Einführung

- Lisp wurde 1958 von John McCarthy (MIT) entwickelt und ist die zweitälteste Programmiersprache die noch benutzt wird.
- Nur Fortran ist älter.
- Common Lisp ist ein Lisp Dialekt

Listen

- Listen bestehen aus sog. Cons Cells



- `(list 5 23 42)`

- Cons Cells haben einen Car und einen Cdr was soviel heißt wie einen vorderen Teil und den Rest der Liste.

- Dies ist sehr praktisch für die funktionale Programmierung

- Lisp Code selbst ist eine Struktur von verschachtelten Listen

Codebeispiel (Listen)

```
(reduce #'+ (mapcar #'(lambda (x) (* x
x)) '(1 2 3)))
```

Funktionen

- Funktionen sind **First Class Objects**
- Man kann mit ihnen alles machen was man mit Variablen auch tun kann.
- Eine Funktion ist auch nur eine Liste
- (funktionsname parameterA parameterB parameterC)
- Beispiel:
  - (defun blupp () (print “abc”))

Codebeispiel (Funktionen)

```
(defun hello-world ()
  (format t “Hello World”))

(defun add (a b)
  (+ a b))

(defun print-list (list)
  (dolist (a list)
     (format t “~a~%” a)))

(defun print-list-2 (list)
  (mapcar #'print list))

(defun print-list-3 (list)
  (mapcar #'(lambda (e) (format t "~a~%" e)) list))
```

Macros

- Macros sind Funktionen welche Lisp Code als Ausgabe erzeugen.
- Sie greifen direkt in den Syntaxbaum der Sprache ein und verändern diesen.
- Macros sind **DER** Grund Lisp zu benutzen
- Man kann mit Macros eigene Subsprachen innerhalb von Lisp definieren. Z.B. das `loop` Konstrukt

Macros

- Dynamische Codeerzeugung
- Compilezeit (aber: Compilezeit ist immer)
- Ermöglicht, was sonst nur mit Lazy Evaluation möglich wäre
- völlig neue Sprachkonstrukte

Listquoting

```
CL-USER> ‘(1 2 3 (+ 3 1))
(1 2 3 (+ 3 1))
CL-USER> `(1 2 3 ,(+ 3 1))
(1 2 3 4)
```

Makrobeispiel: anaphorisches if

- Anapher in der Linguistik: Verweis
- Ziel: “if some file exists, read it”

```
(aif some-file
   (read it))
```

Makrobeispiel: anaphorisches if

```
(defmacro aif (&rest args)
  `(let ((it ,(first args)))
     (if it ,@(rest args))))
```

Makrobeispiel aif

```
CL-USER> (defparameter *foo* "hello")
*FOO*
CL-USER> (aif *foo* (format t "~a~%" it))
hello
CL-USER> (macroexpand-1 '(aif *foo*
(format t "~a~%" it)))
(LET ((IT *FOO*))
  (IF IT (FORMAT T "~a~%" IT)))
```

Makros - gensym

```
CL-USER> (gensym)
#:G2141

(defmacro foo ()
  (let ((var-name (gensym)))
    `(fun ,var-name)))

CL-USER> (macroexpand-1 '(foo))
(FUN #:G2147)
```

Makros - wir schreiben noch eins

- `(let ((var-name (gensym)` ist ein wiederkehrendes Muster

```
(defmacro with-gensyms ((&rest names) &body body)
  `(let ,(loop for n in names collect `(,n (gensym)))
     ,@body))

(defmacro bar () (with-gensyms (a b c) `(fun ,a ,b ,c)))

CL-USER> (macroexpand-1 '(bar))
(FUN #:G2159 #:G2160 #:G2161)
```

ASDF

- Make? Autoconf? ASDF!
- Another System Definition Facility
- Packaging
- Lisp Library Management
- Auch praktisch für eigenen Code

ASDF

```
(defsystem :foo
  (:files (“a” “b”)
  (:depends-on :cl-ppcre))
```

```
(asdf:operate ‘asdf-load-op :foo)
```

OO und Packages

- Wir wollen Namespaces

```
(defpackage :foo
  (:use :cl)
  (:export :bar))
(in-package :foo)
(defun bar (...) ...)
```

CLOS

- Ein Bankkonto:

```
(defclass basic-account ()
  ((balance :initarg :balance :accessor balance)
   (credit-type :accessor credit-type)))
```

Polymorphie

- Generische Funktionen

```
(defgeneric withdraw (account amount)
  (:documentation “withdraw money from account”))
```

```
(defmethod withdraw ((account basic-account)
amount)
  (with-slots (balance) account
    (decf balance amount)
    amount))
```

Polymorphie

```
(defmethod withdraw ((account president-account)) amount)
  amount)
```

Aspektorientierte Programmierung

```
(defmethod withdraw :before ((account basic-account) amount)

  (if (> amount (balance account))

      (error (make-instance 'savings-too-low :text "you don't
have any overdraft"))))
```

```
(defmethod withdraw :after ((account basic-account) amount)

   (log “~a was withdrawn from ~a at ~a~% ” amount (id account)
   (time)))
```

Exceptions aka Conditions

- Conditions mächtiger als normale Exceptions
- Stack un-unwinding

Exceptions aka Conditions

```
(define-condition savings-too-low (error)
  ((text :initarg :text :reader text)
  (amount :initarg :amount :reader amount)))


(error ‘savings-too-low :text text)

(handler-case (code)
  (savings-too-low () (handler-code))
```

Vor und Nachteile von Lisp

- **Vorteile**
- Extreme Hochsprache
- Beliebige Abstraktion
- Trotzdem gute Performance
- Kein Unterschied zwischen Code und Daten
- Dynamisch stark typisiert
- Beliebige Programmierstile sind möglich
- **MACROS!!!**

- **Nachteile**
- Standard ist 20 Jahre alt aus prae Internetzeiten
- Sockets sind implementierungsspezifisch
- Standard enthält Workarounds aus Symbolicszeiten
- Das Sprachdesign ist nicht immer ganz einheitlich
- Tendiert zu langen Funktionsnamen

Wie man tatsächlich Lisp programmiert

- Die **REPL**
- Die Sprache wächst dem Problem entgegen. Man findet sehr schöne und genaue Ausdrucksmöglichkeiten für sein Problem.
- Lisp fördert Exploratives Programmieren ...
- Lisp kann optimiert werden
    - z.B. durch hinzufügen von Typinformationen etc...
- Lisp kann interpretiert und kompiliert werden, die Compiler sind sehr fortschrittlich

Implementierungen von Common Lisp

- **SBCL**
  - Preferierte Open Source Implementierung
  - Generiert nativen Code
  - Arbeitet wunderbar mit Slime zusammen