MDLUG-Themenabend:

Die Familie der LISP-Sprachen

Matthias Köppe

Otto-von-Guericke-Universität Magdeburg, FMA-IMO
Magdeburger Linux User Group e. V.

15. Oktober 2002

Überblick

- LISP (List Processing) ist eine der ältesten Programmiersprachen
- entwickelt von J. McCarthy 1956
- in den 1960er bis 1980er Jahren die Programmiersprache für künstliche Intelligenz
- heute relevante Dialekte: Emacs Lisp, Scheme, Common Lisp

Geschichte von (Common) Lisp

- 1960–1965: Lisp 1.5
- Anfang 1970er: zwei Hauptdialekte, MacLisp und InterLisp
- 1970–1985: verschiedene Lisp Machines (MIT, Xerox, LMI, Symbolics)
- 1975: Scheme (G. J. Sussman; G. L. Steele, Jr.) – lexikalischer Scope, Closures, Continuations
- Ende 1970er: OO-Programmiersysteme (Flavors, Common LOOPS)
- 1981–1986: Common Lisp
- 1986–1994: Standardisierung (ANSI Common Lisp) – erste standardisierte Programmiersprache mit OOP

„Lisp auf dem Desktop“

- GNU Emacs, XEmacs – Emacs Lisp (elisp) als Implementierungs- und Erweiterungsprache
- Sawfish (Window manager) – librep
- GNUcash, GNU lilypond, GNU TeXmacs, . . . – Guile (Scheme)
- GIMP Script-Fu – SIOD (Scheme)

„Große“ Lisp-Anwendungen

- Common Lisp im Weltraum: Remote Agent Experiment auf der Deep-Space-1-Mission (1998)
- Viaweb/Yahoo! Store: teilweise in Common Lisp geschrieben
- AutoLISP: Modellierungssprache von AutoCAD

Scheme-Implementierungen

Sprachstandard:

- Kleiner standardisierter Sprachkern: Revised^5 Report on the algorithmic language Scheme (R^{5}RS, 1992)
- Erweiterungen werden laufend im SRFI-Prozeß diskutiert

Viele unabhängige Implementierungen:

- Interpreter – MIT Scheme, SCM, Guile, Scheme48, . . .
- Compiler – Scheme2C, MzScheme, Static Language Implementation, . . .
- Kawa, Bigloo, SISC übersetzen Scheme in Java-Bytecode

Common-Lisp-Implementierungen

Freie Implementierungen

- CMUCL/SBCL: Native-Code-Compiler für x86, SPARC, Alpha, MIPS, HPPA unter UNIX (Public Domain)
- CLISP: Byte-Compiler, portabel (GPL)
- ECL: Compiliert via C (LGPL)
- OpenMCL: Native-Code-Compiler für PowerPC-Architektur unter Linux, Mac OS X (LGPL)

Proprietäre Implementierungen: Allegro Common Lisp (Franz), Corman Lisp, LispWorks (Xanalys)

Spracheigenschaften

- Einfache, reguläre Syntax, leistungsfähige Makros
- Lexikalischer Abschluß, Funktionen als Werte. . .
- Dynamische Typen: Typen sind mit Werten, nicht mit Variablen verbunden
- Inhomogene Container
- Automatische Speicherverwaltung: *Garbage collection*
- Mehrfache Rückgabewerte
- Interaktivität und Introspektion

Einfache Syntax

Programme bestehen aus **Ausdrücken**:

- **Atome**: `91`, `"Text"`, `write`
  - Zahlen, Strings, . . . stehen für sich selbst
  - **Symbole** stehen für Variablen oder Funktionen
- **Listen**: `(max 17 3 55 10)`
  - **Prozedur**aufrufe: `(write "Text")`
  - **Spezialformen**:
```
(if (even? x) (write "gerade")
              (write "ungerade"))
```
  Das Symbol (der Wert) am Anfang der Liste entscheidet (**Präfixnotation**).

Einfache Syntax – Leistungsfähige Makros

- Parsen von Lisp-Programmen ist trivial (*reader*)
- Programme können in natürlicher Weise als **Daten** verarbeitet werden (Listenverarbeitung, keine Textverarbeitung)

$\longrightarrow$ Leistungsfähige Makros (Syntax-Transformationen)

$\longrightarrow$ Problemspezifische Sprachen (z. B. Datenbankabfragesprachen, Pattern Matching, . . .) können integriert werden

Außerdem: Leistungsfähige Editierkommandos möglich

Syntax – Vergleich mit C/C++/Java

- Parsen von C/C++/Java ist schwer
- Geparstes Programm steht nur in speziellen Datenstrukturen zur Verfügung

→ Für C/C++ ist nur ein simples Textersetzungssystem (Präprozessor) bzw. ein einfaches Pattern-Matching-System (C++-Templates) verfügbar

→ Java hat kein Makrosystem

→ *Cut-and-Paste*-Techniken beim Programmieren sind populär

→ Wiederkehrende syntaktische Muster werden auf den Programmierer abgewälzt (*Design patterns*)

Spracheigenschaften – Datentypen von Common Lisp

- Zahlen (insbesondere komplexe Zahlen, exakte Brüche, Bignums)
- Symbole
- inhomogene Listen
- Arrays (inkl. Strings, Bit-Vektoren und mehrdimensionaler Arrays)
- Hashtabellen
- Strukturen und Klassenobjekte
- Funktionen (insbesondere *Closures*)

Spracheigenschaften – Lexikalischer Abschluß

Sortieren in C mit Standardfunktion:

```
void qsort(void *, size_t, size_t,
           int (*cmp)(*, *));
```

Was, wenn die Vergleichsfunktion `cmp` von Parametern abhängen soll? Etwa Sortieren nach der i-ten Komponente eines Vektors:

```
void sort_vectors(int **vectors,
                  size_t count,
                  size_t i);
```

Der Parameter i muß in die Sortierfunktion hineingereicht werden!

Spracheigenschaften – Lexikalischer Abschluß II

Als globale Variable hineinreichen:

```
int the_i;  /* yuck! */

int vector_cmp(int **a, int **b) {
  return (*a)[the_i] - (*b)[the_i];
}

void sort_vectors(int **vectors, size_t count, int i)
{
  the_i = i;  /* yuck! */
  qsort(vectors, count, sizeof(int*), vector_cmp);
}
```

Oder neue Sortierfunktion schreiben!

Lösung in C++ mit Templates und `operator()`

```
template <class cmp>
void qsort(void *, size_t, size_t, cmp);

struct vector_cmp {
  int the_i;
  vector_cmp(int i) : the_i (i) {}
  int operator() (int **a, int **b)
    { return (*a)[the_i] - (*b)[the_i]; }
};

void sort_vectors(int **vectors, size_t count, int i)
{
  qsort(vectors, count, sizeof(int*), vector_cmp(i));
}
```

Spracheigenschaften – Lexikalischer Abschluß III

In Lisp:

```
(defun sort-vectors (vectors i)
  (sort vectors
        (lambda (a b)
          (< (aref a i)
             (aref b i)))))
```

Eine **anonyme Funktion** mit Argumenten `(a b)`, die über i **abschließt**, (eine *Closure*) wird zum Vergleichen benutzt.

Spracheigenschaften – Lexikalischer Abschluß IV

Funktionen höherer Ordnung (nehmen Funktionen als Argument):

- Auswahl aus Listen

```
(remove-if-not 'evenp '(12 13 17 18 21))   => (12 18)
```

- Iteration, z. B. über Listen:

```
(mapcar (lambda (x y) (* x y))
        '(1  2  3  4  5)
        '(2  3  4  5  6))
    =>   (2  6 12 20 30)
```

Auch hier ist es möglich, *Closures* zu übergeben.

Funktionen höherer Ordnung vs. C++-Iteratoren

C++-Iteratoren-Muster:

```
foo_iterator i;
for (i = bar.begin(); i != bar.end(); ++i) {
  body...
}
```

Problem: `foo_iterator::operator++()` ist nur für triviale Datenstrukturen leicht und effizient zu schreiben. (Kontrollfluß muß als Datenfluß simuliert werden!)

Natürlicher Kontrollfluß, häufig effizienter:

```
bar.iterate(body_function);
```

. . . aber syntaktisch hoffnungslos in C++ (keine lokalen Funktionen/Closures)

Spracheigenschaften – Interaktivität

- *Read-eval-print loop*
- Reflexion (Introspektion)
  - Abfragen von Werten, Dokumentation, Signatur
  - Untersuchung der Klassenhierarchie, der anwendbaren Methoden usw.
- Code, Daten, Klassenhierarchie zur Laufzeit änderbar

$\longrightarrow$ wesentlich schnellere Entwicklung möglich als mit traditionellem Debugging-Zyklus

Wichtige Sprachelemente I – Definitionen

(defvar *Variable Initialwert*
  "Dokumentation")

(defun *Funktion* (*Argumente*. . .)
  "Dokumentation"
  $Ausdruck_1$
  ⋮
  $Ausdruck_n$)

(defmethod *Methode* ((Arg_1 Typ_1)
                      ⋮
                      (Arg_n Typ_n))
  Ausdrücke. . .)

(defstruct *Struktur*
  $Slot_1$
  ($Slot_2$ $Initform_2$:type Typ_2)
  ⋮
  $Slot_n$)

Wichtige Sprachelemente II – Kontrollstrukturen

(progn
  $Ausdruck_1$
  ⋮
  $Ausdruck_n$)

(if $Bedingung$
  $then\text{-}Ausdruck$
  $else\text{-}Ausdruck$)

(cond ($Bedingung_1$ $Ausdruck_1$
  ⋮
  $Ausdruck_n$)
  ⋮
  ($Bedingung_m$ $Ausdrücke \ldots$))

catch/throw/ block/return-from/
unwind-protect – für *non-local exits*

dotimes – Zählschleifen

dolist – Iteration über Listen

loop – sehr allgemeines Iterationsmakro

Wichtige Sprachelemente III – Binden/Setzen von Variablen

Paralleles/sequentielles Binden von Variablen an Werte

(let (($Variable_1$ $Ausdruck_1$)
⋮
($Variable_n$ $Ausdruck_n$))
$Ausdrücke...$)

(let* (($Variable_1$ $Ausdruck_1$)
⋮
($Variable_n$ $Ausdruck_n$))
$Ausdrücke...$)

Zuweisung an Variablen

(setq $Variable_1$ $Wert_1$
⋮
$Variable_n$ $Wert_n$)

Zuweisung an *generalized places* (Verallgemeinerung von *lvalues*)

(setf $Place_1$ $Wert_1$
⋮
$Place_n$ $Wert_n$)

Lisp-Makros

Makros sind zur Compile-Zeit ausgeführte Syntaxtransformationen:

(case $Ausdruck$
  ($Schlüssel_1$ $Ausdrücke_1 \ldots$)
  ($Schlüssel_2$ $Ausdrücke_2 \ldots$)
  ⋮
  ($Schlüssel_n$ $Ausdrücke_n \ldots$))

$\longrightarrow$

(let ((Wert $Ausdruck$))
  (if (eql Wert $Schlüssel_1$)
    (progn $Ausdrücke_1 \ldots$)
    (if (eql Wert $Schlüssel_2$)
      (progn $Ausdrücke_2 \ldots$)
      (if ...
        $\cdots$))))...)))

Realisiert durch eine gewöhnliche Lisp-Funktion, die aus Programm„text“ neuen Programm„text“ berechnet.

(Verarbeitung verschachtelter Listen)

Lisp-Makros – ganz einfaches Beispiel

$$\begin{array}{l}\text{(when } Bedingung \\ \quad Ausdruck_1 \\ \quad \vdots \\ \quad Ausdruck_n\text{)}\end{array} \longrightarrow \begin{array}{l}\text{(if } Bedingung \\ \quad \text{(progn } Ausdruck_1 \\ \qquad \vdots \\ \qquad Ausdruck_n\text{))}\end{array}$$

```
(defmacro (when Bedingung &rest Ausdruecke)
  (list 'if
        Bedingung
        (cons 'progn Ausdruecke)))
```

Oder mit Quasiquote:

```
(defmacro (when Bedingung &rest Ausdruecke)
  `(if ,Bedingung
       (progn ,@Ausdruecke)))
```

Literatur

Links finden sich auf

http://www.mdlug.de/index.php/links/Programmierung/Lisp

und

http://www.mdlug.de/index.php/links/Programmierung/Scheme