Übung - Modellierung & Programmierung II

Mathias Goldau, Stefan Koch, Wieland Reich, Dirk Zeckzer, Stefan Philips, Sebastian Volke

math@informatik.uni-leipzig.de stefan.koch@informatik.uni-leipzig.de
reich@informatik.uni-leipzig.de zeckzer@informatik.uni-leipzig.de
philips@informatik.uni-leipzig.de volke@informatik.uni-leipzig.de

Outline

LISP

- 1958 von John McCarthy erfunden:
  - “Lisp seems to be a lucky discovery of a local maximum in the space of programming languages.” – John McCarthy
  - “Lisp is a programmable programming language” – John Foderaro
- Funktionales Programmierparadigma
- Beeinflusste und beeinflusst viele andere Sprachen: Perl, Smalltalk, Python, Ruby, Haskell
- Sprache (mit Macros[7]) einfach zu verändern → viele Dialekte: Common-Lisp, Scheme, Auto-Lisp, Emacs-Lisp, Clojure, ...

[7] Keinesfalls zu verwechseln mit C-Macros

LISP Dialekte

- Kein *gemeinsamer* Lisp Standard
- Interpretierte Sprache aber Übersetzung in Bytecode und nativen Maschinencode möglich
- Oft dynamische Typisierung
- Datenstruktur Liste von zentraler Bedeutung:
  - LISP ≡ LISt Processing
  - Daten und Programmcode als Listen
- In dieser Übung Common Lisp:
  - Standardisiert: ANSI/X3.226-1994[8]
  - Viele freie und proprietäre Implementierungen: CLISP, CUMCL, GCL
  - CLISP 2.49 empfohlen
  - Literatur: 2005, P. Seibel, *Practical Common Lisp*[9]

---

[8] Final Draft PDF, HTML

[9] http://www.gigamonkeys.com/book/

Common Lisp: Hello World

- Direkt im Interpreter:

```
(print "Hello World")
```

- Als Skript:

```
$ clisp hello.cl
```

- Noch viele andere Möglichkeiten:

```
(pprint "Hello World")
(format t "Hello World~%")
...
```

Grundlegendes

- Lisp nutzt Listen und Atome:
  - Listen sind durch runde Klammern begrenzt
  - Elemente in Listen sind durch Whitespace getrennt
  - Alles andere sind Atome
- Listen können beliebig verschachtelt werden
- Solche geklammerten Ausdrücke heißen *s-expressions*[10]
- Die leere Liste: `()` ≡ `NIL` ist hierbei Liste und Atom zugleich
- Kommentare beginnen mit einem Semikolon und gelten für den Rest der Zeile

[10] kurz für: *symbolic-expression*

Datentypen

- Variablen haben keinen Typ, aber ihr Inhalt (data-objects)
- Bspw. Literale, Atome
- (type-of "Hello World")

```
123 ; integer
2/8 ; ratio
1.0 ; float
1.0d ; double
#c(1 2) ; complex
#\a ; character
"str" ; string
'foobar ; symbol
```

- Aber auch: Array, Vector, Hashtable, Streams, Structures, . . .
- Listen können Elemente mit unterschiedliche Datentypen haben `(x 1 "drei")`

Listen als Funktionen und Daten

- Daten: `(1 2 3)`
- Funktionen: Das erste Element einer Liste ist der Name der Funktion und der Rest sind Argumente
- Beispiel: `(myfunction arg1 arg2)`
- Wie unterscheidet Lisp zwischen Listen `(1 2 3)` und Funktionen `(equal 1 2)`?
- Daten als Programm: `(eval (..))`
- Programm als Daten: `'(eval (..))`
- Standardmäßig werden Listen immer evaluiert was mit *quote* verhindert werden kann
- Der Operator ' heisst *quote*, alternativ: `(quote (1 2 3))`

Beispiel Listen Auswertung

```
[1]> '(1 2 3)
(1 2 3)
[2]> (1 2 3)
*** - EVAL: 1 is not a function name; try using a symbol instead
The following restarts are available:
USE-VALUE :R1 Input a value to be used instead.
ABORT :R2 Abort main loop
Break 1 [3]>
[4]> (equal 1 2)
NIL
[5]> '(equal 1 2)
(EQUAL 1 2)
[6]> (eval '(equal 1 2))
NIL
```

Funktionen auf Listen

- `car`:[11] liefert das erste Listenelement
- `cdr`:[12] liefert alle Elemente nach dem ersten Listenelement
- `car` und `cdr` ändern die Liste nicht
  - `setcar` und `setcdr` ändern die Liste
- `cons`: fügt das erste Element an Beginn des zweiten Arguments an

```
(cons 'nelke (cons 'rose (cons 'butterblume nil)))
```

- `append`: vereint zwei Listen

```
(append '(1 2) '(3 4)) ; ergibt (1 2 3 4)
```

- Beispiele: siehe Abschnitt zur Rekursion

[11] content of the adress part of the register

[12] content of the decrement part of the register

Wo ist mein printf?

- Synopsis: `format` *destination control-string args => result*
- Beispiel: `(format t "Hello World.")`
- `t` ist hierbei stellvertretend für *standard-output*
- Beispiel:

```
(format t "idx: ~d~%val: ~a~%" 12 "-34.4")
```

```
idx: 12
val: -34.4
```

Variablen

- Variablen oder auch Symbole können verschiedene Werte annehmen: Symbol, Nummer, Liste, String, . . .
- Insbesondere aber auch *Funktionsdefinitionen*

```
1 (set 'foo 'bar)
2 (set 'foo 1)
3 (set 'foo '(1 2 3 (4 5) ))
4 (set 'foo "Hello World")
```

Variablenzuweisung

set: `(set 'blumen '(rose butterblume))`

- Zweites Argument wird an das Symbol `blumen` gebunden
- Beide Argumente mit *quote* versehen, da keine Auswertung erwünscht

setq: `(setq blumen '(rose butterblume))`

- Wie `set` aber erstes Argument automatisch in `quote`

setf: `(setf (car list) 'nelke)`

- Wie `setq` kann darüber hinaus auch auf Speicherstellen zugreifen

let: `(let ((Variable Wert)..(Variable Wert)) body)`

- Bindet wie `set` den Wert aus der Variablenliste an jeweilige Variable
- `body` beinhaltet anschließend auszuwertende Ausdrücke
- Bildet Scope und lokale Variablen überdecken gleichnamige außerhalb
- Wert nach der Auswertung des `let`-Ausdrucks wieder zurückgesetzt
- Alle Wert-Variablen Paare parallel gebunden

let*:

- Wie `let` aber sequentielle Bindung `(let* ((i 1)(j i)..)..)`

Variablenbindung

- Binding: Wie Variablename zum Speicherplatz korrespondiert
- Lexical Binding:
  - Bindung eines Werts an eine Variable bei der Definition
- Dynamic Bining:
  - Bindung eines Werts an eine Variable bei der Benutzung
  - Jede Variable hat einen Stack auf dem Bindungen hinterlegt sind
  - Neue Bindungen während der Laufzeit verdecken vorherige Bindungen
- Common Lisp verwendet nur lexikalische Bindung
- Emacs Lisp verwendet darüber hinaus auch dynamische Bindung

Variablenbindung Beispiel

```
1                                ; dyn. binding   || lex. binding
2                                ;----------------++--------------
3 (setq x 7)                     ;       7        ||       7
4 (defun blablub () x)           ;   blablub      ||   blablub
5 (blablub)                      ;       7        ||       7
6 (let ((x 42)) (blablub))       ;       42       ||       7
7 (blablub)                      ;       7        ||       7
```

Funktionsdefinition

- Es gibt viele vordefinierte Grundfunktionen
- Funktionsdefinitionen beginnen mit dem Symbol `defun`
  - Symbolname an den Funktionsdefinition gebunden werden soll
  - Liste der benötigten Argumente
  - Die Definition (tatsächlichen Befehle)
  - Vor Funktionsaufruf werden alle Argumente evaluiert
  - Nicht alles was aussieht wie eine Funktion ist auch eine:
    `(if (eq x 0) (..) (..))`

```
1 (defun funktionsname (Argumente)
2       "Dokumentation der Funktion, vgl. (describe 'fktname)"
3       body
4 )
```

Beispiel Addition

```
1 (defun add (a b)
2       "Addiere a und b."
3       (+ a b)
4 )
5
6 (add 3 2) ;Beispiel für Funktionsaufruf
```

Kontrollstrukturen: if

- `if` braucht mindestens zwei Argumente

```
1 (if (wahr-falsch-Test)
2     (Ausdrücke-wenn-wahr)    ; then
3     (Ausdrücke-wenn-falsch) ; else
4 )
```

- Beispiele:

```
1 (if (atom '(rose butterblume))
2    'ist_Atom
3    'ist_Liste
4 )
5 (if (equal 1 2)
6   "1==2"
7   "1!=2"
8 )
```

Kontrollstrukturen: cond

- Ähnlich zu `switch-case` gibt es auch die Funktion `cond` (von *condition*)

```
1 (cond
2       (wahr-falsch-Test Auswirkung)
3       (wahr-falsch-Test Auswirkung)
4       ...
5       (wahr-falsch-Test Auswirkung)
6 )
```

- `cond` testet Ausdrücke auf wahr/falsch
- Ergibt Test “wahr”: “Auswirkung” wird ausgewertet; sonst: nächster Ausdruck wird getestet

Kontrollstrukturen: Schleifen

- Schleifen und Rekursion sind prinzipiell gleichmächtig
- Manchmal ist eine Schleife einer Rekursion vorzuziehen und andersherum
- Beispiel Rekursion: Ackermann-Funktion
- Beispiel Schleife: Iteration über Liste

Schleifen: loop

```
1 (loop (pprint "Mir ist langweilig.")) ;endless loop
```

- Das loop Macro ist sehr sehr sehr mächtig, vgl. www.lispworks.com/documentation/HyperSpec/Body/m_loop.htm
- Es birgt Erweiterungen um alle gängigen Schleifen nachzubauen
- Es folgen Beispiele für `for`, `while` und `dolist`

Schleifen: dolist

```
1 (dolist (e '(1 2 3)) (print e)) ; print: 1,2,3
2
3 (dolist (e '(1 2 3))
4   (print e)
5   (if (evenp e) (return))
6 )                                 ; print: 1,2
7
8 (dolist (e (reverse '(1 2 3))) (print e)) ; print: 3,2,1
```

- Iteriert über Liste

Schleifen: for

```
1 (loop for i in '(1 2 3) do (print i)) ;print: 1, 2, 3
2 (loop for i from 1 to 3 do (print i)) ;print: 1, 2, 3
3 (loop for i from 3 downto 1 do (print i)) ;print: 3, 2, 1
4 (loop for i on '(1 2 3) do (print i)) ;print cdr: (1 2 3),(2 3),(3)
5 (loop for i across "foobar" do (pprint i)) ;print: #\f,#\o,#\o,..
6 (loop for i from 1 to 3 do (print i) collect (* i i) ) ;print:
      1,2,3
7 (1 4 9)
```

- Sehr flexible Schrittweitensteuerung

Schleifen: while-do und do-while

```
1 (setf i 0)
2 (loop while (< i 3) do (incf i 1)(print i)) ; print: 1, 2, 3
3 (setq i 0)
4 (loop do (incf i)(print i) while (/= 0 (mod i 3))) ; print: 1,2,3
```

Rekursion

- An die Abbruchbedingung denken!

```
1 (defun funktionsname (Argumente)
2       "Dokumentation"
3       (if Test-der-Abbruchbedinung
4           body
5           (funktionsname neue-Argumente)
6       )
7 )
```

Beispiel Rekursion: Summe

```
1  ; Summe aller Zahlen einer Liste
2  ; rekursiv durch Abtrennen des jeweils ersten Elements
3  ; (summe nil) -> 0
4  ; (summe '(3 5 2)) -> (+ 3 (summe '(5 2))) -> ... -> 10
5  (defun summe (L)
6      (if (eq L nil) ; leere Liste?
7          0 ; summe(nil) -> 0
8          (+ (car L) (summe (cdr L))) ; head+summe(tail)
9      )
10 )
11
12 (summe '(2 6 3 9))
```

Beispiel Rekursion: Liste generieren

```
1  ; Liste der Zahlen von n bis 1, rekursiv
2  ; durch Voranstellen von n den Zahlen von n-1 bis 1
3  (defun countdown (n)
4      (if (eq n 0) ; keine Zahlen?
5          nil ; leere Liste
6          (cons n (countdown (- n 1))) ; n voranstellen
7      )
8  )
9
10 (countdown 10)
```

- Sprachelemente

```
quote '
car cdr cons nil
cond if atom eq not and or
defun lambda
+ - * / mod < > >= <= /=
```