Outline

Lisp

- 1958 von John McCarthy erfunden
- Funktionales Programmierparadigma
- Beeinflusst viele andere Sprachen
- Sprache (mit Macros) einfach zu verändern → viele Dialekte: Common-Lisp, Emacs-Lisp, Clojure,..
- Kein *gemeinsamer* Standard
- Übersetzung in Bytecode möglich aber auch nativer Maschinencode
- Datenstruktur Liste von zentraler Bedeutung
  - LISP ≡ LISt Processing[5]

[5] Manchmal auch: LISP ≡ Lots of Isolated Silly Parentheses

Lisp

- Lisp verarbeitet Listen von Listen
- Listen sind durch Klammern begrenzt
- Listen-Elemente sind durch Leerzeichen getrennt: `(1 (2 a)"aas")`
- Daten und Programme werden durch Listen repräsentiert
- Programmausgaben können selbst wieder auswertbare Programme sein
- Neben Listen auch *atoms*: kleinste, nicht weiter zerlegbare Bestandteile
- Für Nutzer angezeigte Repräsentation von Listen und Atomen heißen *symbolic expressions* (kurz: *s-expression*)

Emacs-Lisp Einführung

- In den Übungen verwenden wir Emacs-Lisp.
- Die folgenden Folien geben bloß eine sehr kurze Zusammenfassung!
- Eine detailiertere Einführung beispielweise unter: www.gnu.org/software/emacs/emacs-lisp-intro.

Emacs Lisp - Benutzung

- interaktiver Emacs Lisp Modus: `M-x ielm`
- Auswerten eines einzelnen Ausdrucks: `C-x C-e`
- Auswerten aller Ausdrücke im Buffer: `M-x eval-buffer`
- Laden und auswerten einer Datei durch folgenden Ausdruck:
  `(load "../Pfad_zur_Datei/datei.el")`

Emacs Lisp - Funktionsweise des Interpreters

- Daten und Programme in Listen organisiert
- Wenn Liste ausgeführt wird, ist erstes Element der Funktionsname[6]
- Einfaches Apostroph ', oder `(quote )` verhindert Ausführung, Liste wird daher unausgewertet zurückgegeben
- innerste Liste immer zuerst ausgewertet

```
ELISP> (1 2 3)
*** Eval error ***  Invalid function: 1
ELISP> '(1 2 3)
(1 2 3)
ELISP> (quote (1 2 3))
(1 2 3)
ELISP> (+ 1 2)
3
```

[6] Unterart der Präfixnotation

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**:
  - Beispiel: `(set 'blumen '(rose butterblume))`
  - Zweites Argument wird an das Symbol `blumen` gebunden
  - Beide Argumente mit *quote* versehen, da keine Auswertung erwünscht
- **setq**:
  - Wie `set` aber erstes Argument automatisch in `quote`
- **let**:
  - Beispiel: `(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

Variable Binding

- 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 (lex.) unterscheidet sich hier von Emacs List (dyn.)

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
  - Dokumentation der Funktion (bei `C-h f funktionsname` angezeigt)
  - Optionaler Ausdruck um interaktive Ausführung zu ermöglichen
  - Die Definition (tatsächlichen Befehle)

```
1 (defun funktionsname (Argumente)
2        "optionale Dokumentation"
3        (Argumente für interaktives parsen)
4        body
5 )
```

Beispiel Addition

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

Bedingungen: if

- `if`-Funktion braucht mindestens zwei Argumente

```
1 (if (wahr-falsch-Test)
2     (Ausdrücke-wenn-wahr)
3     (Ausdrücke-wenn-falsch) ;optional ("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 )
```

Bedingungen: cond

- Neben `if` gibt es auch die Funktion `cond` (*condition*).
- Beispiel:

```
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

Funktionen auf Listen

- `car`:[7] liefert das erste Listenelement
- `cdr`:[8] liefert alle Elemente nach dem ersten Listenelement
- `car` und `cdr` ändern Liste selbst nicht wohlaber `setcar` und `setcdr`
- `cons`: fügt das erste Element an Beginn des zweiten Arguments an

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

- Beispiele: siehe Abschnitt zur Rekursion

[7] content of the adress part of the register
[8] content of the decrement part of the register

While Schleife

- `(while wahr-falsch-Test body)`
- Interpreter wertet solang das zweite Argument body aus, bis der Test fehlschlägt

```
1 (defun nenne_blumen (blumen)
2         "Gib Listenelemente aus."
3         (while blumen
4                 (print (car blumen))
5                 (setq blumen (cdr blumen))
6         )
7 )
8 (nenne_blumen '(rose butterblume aster))
```

Rekursion

- Funktionsdefiniton in der die definierte Funktion selbst wieder vorkommt
- *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)
```

Emacs Lisp

- Sprachelemente

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