Common Lisp - viel mehr als nur dämliche Klammern

Alexander Schreiber <als@thangorodrim.de>

http://www.thangorodrim.de

Chemnitzer Linux-Tage 2005

Greenspun's Tenth Rule of Programming:

"Any sufficiently-complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp."

Übersicht

Lisp? Wofür?

- NASA: Remote Agent (Deep Space 1), Planner (Mars Pathfinder),
- Viaweb, gekauft von Yahoo für 50 Millionen $,
- ITA Software: Orbitz engine (Flugticket Planung),
- Square USA: Production tracking für “Final Fantasy”,
- Naughty Dog Software: Crash Bandicoot auf Sony Playstation,
- AMD & AMI: Chip-Design & Verifizierung,
- typischerweise komplexe Probleme: Wissensverarbeitung, Expertensysteme, Planungssysteme

Lisp? Wofür?

- NASA: Remote Agent (Deep Space 1), Planner (Mars Pathfinder),
- Viaweb, gekauft von Yahoo für 50 Millionen $,
- ITA Software: Orbitz engine (Flugticket Planung),
- Square USA: Production tracking für “Final Fantasy”,
- Naughty Dog Software: Crash Bandicoot auf Sony Playstation,
- AMD & AMI: Chip-Design & Verifizierung,
- typischerweise komplexe Probleme: Wissensverarbeitung, Expertensysteme, Planungssysteme

Lisp? Wofür?

- NASA: Remote Agent (Deep Space 1), Planner (Mars Pathfinder),
- Viaweb, gekauft von Yahoo für 50 Millionen $,
- ITA Software: Orbitz engine (Flugticket Planung),
- Square USA: Production tracking für “Final Fantasy”,
- Naughty Dog Software: Crash Bandicoot auf Sony Playstation,
- AMD & AMI: Chip-Design & Verifizierung,
- typischerweise komplexe Probleme: Wissensverarbeitung, Expertensysteme, Planungssysteme

Lisp? Wofür?

- NASA: Remote Agent (Deep Space 1), Planner (Mars Pathfinder),
- Viaweb, gekauft von Yahoo für 50 Millionen $,
- ITA Software: Orbitz engine (Flugticket Planung),
- Square USA: Production tracking für “Final Fantasy”,
- Naughty Dog Software: Crash Bandicoot auf Sony Playstation,
- AMD & AMI: Chip-Design & Verifizierung,
- typischerweise komplexe Probleme: Wissensverarbeitung, Expertensysteme, Planungssysteme

Lisp? Wofür?

- NASA: Remote Agent (Deep Space 1), Planner (Mars Pathfinder),
- Viaweb, gekauft von Yahoo für 50 Millionen $,
- ITA Software: Orbitz engine (Flugticket Planung),
- Square USA: Production tracking für “Final Fantasy”,
- Naughty Dog Software: Crash Bandicoot auf Sony Playstation,
- AMD & AMI: Chip-Design & Verifizierung,
- typischerweise komplexe Probleme: Wissensverarbeitung, Expertensysteme, Planungssysteme

Lisp? Wofür?

- NASA: Remote Agent (Deep Space 1), Planner (Mars Pathfinder),
- Viaweb, gekauft von Yahoo für 50 Millionen $,
- ITA Software: Orbitz engine (Flugticket Planung),
- Square USA: Production tracking für “Final Fantasy”,
- Naughty Dog Software: Crash Bandicoot auf Sony Playstation,
- AMD & AMI: Chip-Design & Verifizierung,
- typischerweise komplexe Probleme: Wissensverarbeitung, Expertensysteme, Planungssysteme

Lisp? Wofür?

- NASA: Remote Agent (Deep Space 1), Planner (Mars Pathfinder),
- Viaweb, gekauft von Yahoo für 50 Millionen $,
- ITA Software: Orbitz engine (Flugticket Planung),
- Square USA: Production tracking für “Final Fantasy”,
- Naughty Dog Software: Crash Bandicoot auf Sony Playstation,
- AMD & AMI: Chip-Design & Verifizierung,
- typischerweise komplexe Probleme: Wissensverarbeitung, Expertensysteme, Planungssysteme

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Common Lisp in Schlagworten

- zweitälteste Programmiersprache (nach FORTRAN),
- funktional, imperativ, objekt-orientiert,
- strong typing, dynamic typing, late binding,
- Funktionen als first class objects,
- Symbolverarbeitung,
- REPL: interaktive Entwicklung,
- garbage collector,
- einfacher Sprachkern – mächtige Umgebung,
- einheitliche Darstellung von Programm und Daten,
- “the programmable programming language”,

Lisp als Innovationsmotor 1/2

- Conditionals (`(cond ...)`),
- rekursive Funktionsaufrufe,
- dynamische Speicherverwaltung,
- Garbage Collector,
- first class functions,
- lexical closures

Lisp als Innovationsmotor 1/2

- Conditionals (`(cond ...)`),
- rekursive Funktionsaufrufe,
- dynamische Speicherverwaltung,
- Garbage Collector,
- first class functions,
- lexical closures

Lisp als Innovationsmotor 1/2

- Conditionals (`(cond ...)`),
- rekursive Funktionsaufrufe,
- dynamische Speicherverwaltung,
- Garbage Collector,
- first class functions,
- lexical closures

Lisp als Innovationsmotor 1/2

- Conditionals (`(cond ...)`),
- rekursive Funktionsaufrufe,
- dynamische Speicherverwaltung,
- Garbage Collector,
- first class functions,
- lexical closures

Lisp als Innovationsmotor 1/2

- Conditionals (`(cond ...)`),
- rekursive Funktionsaufrufe,
- dynamische Speicherverwaltung,
- Garbage Collector,
- first class functions,
- lexical closures

Lisp als Innovationsmotor 1/2

- Conditionals (`(cond ...)`),
- rekursive Funktionsaufrufe,
- dynamische Speicherverwaltung,
- Garbage Collector,
- first class functions,
- lexical closures

Lisp als Innovationsmotor 2/2

- interaktive Programmierung
- Compilertechnologie,
- inkrementelles Compilieren,
- dynamic typing,
- erste standardisierte objektorientierte Sprache (CLOS)

Lisp als Innovationsmotor 2/2

- interaktive Programmierung
- Compilertechnologie,
- inkrementelles Compilieren,
- dynamic typing,
- erste standardisierte objektorientierte Sprache (CLOS)

Lisp als Innovationsmotor 2/2

- interaktive Programmierung
- Compilertechnologie,
- inkrementelles Compilieren,
- dynamic typing,
- erste standardisierte objektorientierte Sprache (CLOS)

Lisp als Innovationsmotor 2/2

- interaktive Programmierung
- Compilertechnologie,
- inkrementelles Compilieren,
- dynamic typing,
- erste standardisierte objektorientierte Sprache (CLOS)

Lisp als Innovationsmotor 2/2

- interaktive Programmierung
- Compilertechnologie,
- inkrementelles Compilieren,
- dynamic typing,
- erste standardisierte objektorientierte Sprache (CLOS)

Wie alles begann

- Anfänge 1956 - Listenverarbeitung in *FORTRAN*,
- John McCarthy: mathematische Notation für rekursive Funktionen symbolischer Ausdrücke,
- Steve Russell: Notation einfach auf Computer ausführbar $\longrightarrow$ Programmiersprache,
- 1959: erster Lisp-Interpreter am MIT,
- System IBM 704, 36 Bit Maschine, `car` & `cdr`,
- ab 1962 verschiedene Implementationen,
- 1976: Lisp Maschinen

Wie alles begann

- Anfänge 1956 - Listenverarbeitung in *FORTRAN*,
- John McCarthy: mathematische Notation für rekursive Funktionen symbolischer Ausdrücke,
- Steve Russell: Notation einfach auf Computer ausführbar $\longrightarrow$ Programmiersprache,
- 1959: erster Lisp-Interpreter am MIT,
- System IBM 704, 36 Bit Maschine, `car` & `cdr`,
- ab 1962 verschiedene Implementationen,
- 1976: Lisp Maschinen

Wie alles begann

- Anfänge 1956 - Listenverarbeitung in *FORTRAN*,
- John McCarthy: mathematische Notation für rekursive Funktionen symbolischer Ausdrücke,
- Steve Russell: Notation einfach auf Computer ausführbar $\longrightarrow$ Programmiersprache,
- 1959: erster Lisp-Interpreter am MIT,
- System IBM 704, 36 Bit Maschine, `car` & `cdr`,
- ab 1962 verschiedene Implementationen,
- 1976: Lisp Maschinen

Wie alles begann

- Anfänge 1956 - Listenverarbeitung in *FORTRAN*,
- John McCarthy: mathematische Notation für rekursive Funktionen symbolischer Ausdrücke,
- Steve Russell: Notation einfach auf Computer ausführbar $\longrightarrow$ Programmiersprache,
- 1959: erster Lisp-Interpreter am MIT,
- System IBM 704, 36 Bit Maschine, `car` & `cdr`,
- ab 1962 verschiedene Implementationen,
- 1976: Lisp Maschinen

Wie alles begann

- Anfänge 1956 - Listenverarbeitung in *FORTRAN*,
- John McCarthy: mathematische Notation für rekursive Funktionen symbolischer Ausdrücke,
- Steve Russell: Notation einfach auf Computer ausführbar $\longrightarrow$ Programmiersprache,
- 1959: erster Lisp-Interpreter am MIT,
- System IBM 704, 36 Bit Maschine, `car` & `cdr`,
- ab 1962 verschiedene Implementationen,
- 1976: Lisp Maschinen

Wie alles begann

- Anfänge 1956 - Listenverarbeitung in *FORTRAN*,
- John McCarthy: mathematische Notation für rekursive Funktionen symbolischer Ausdrücke,
- Steve Russell: Notation einfach auf Computer ausführbar $\longrightarrow$ Programmiersprache,
- 1959: erster Lisp-Interpreter am MIT,
- System IBM 704, 36 Bit Maschine, `car` & `cdr`,
- ab 1962 verschiedene Implementationen,
- 1976: Lisp Maschinen

Wie alles begann

- Anfänge 1956 - Listenverarbeitung in *FORTRAN*,
- John McCarthy: mathematische Notation für rekursive Funktionen symbolischer Ausdrücke,
- Steve Russell: Notation einfach auf Computer ausführbar $\longrightarrow$ Programmiersprache,
- 1959: erster Lisp-Interpreter am MIT,
- System IBM 704, 36 Bit Maschine, `car` & `cdr`,
- ab 1962 verschiedene Implementationen,
- 1976: Lisp Maschinen

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Standardisierung

- Entwicklung verschiedener Lisp-Dialekte
- 1981 Beginn der Standardisierung
- 1984 Guy L. Steele: "Common Lisp the Language" (CLtL),
- 1986 erste Common Lisp Implementationen,
- 1986 Beginn der ANSI Standardisierung (X3J13),
- 1994 ANSI Standard X3.226-1994 bestätigt,
- aktuelle Lisp-Dialekte:
  - Common Lisp (= "industrial strength Lisp"),
  - elisp (Emacs Editor),
  - AutoLISP (AutoCAD),
  - Scheme (= "small, clean language"),
  - ...

Lisp Syntax

- basiert nur auf Ausdrücken,
- Ausdruck = entweder Liste oder Atom,
- Liste = `( ...)`, z.B. `(1 2 "boo!")`
- Atom = einfaches Element, z.B. `"boo!"`, `42`, `:symbol`,
- einfache, regelmäßige Syntax
- gleiche Darstellung für Code & Daten,

Lisp Syntax

- basiert nur auf Ausdrücken,
- Ausdruck = entweder Liste oder Atom,
- Liste = `( ...)`, z.B. `(1 2 "boo!")`
- Atom = einfaches Element, z.B. `"boo!"`, `42`, `:symbol`,
- einfache, regelmäßige Syntax
- gleiche Darstellung für Code & Daten,

Lisp Syntax

- basiert nur auf Ausdrücken,
- Ausdruck = entweder Liste oder Atom,
- Liste = `( ...)`, z.B. `(1 2 "boo!")`
- Atom = einfaches Element, z.B. `"boo!"`, `42`, `:symbol`,
- einfache, regelmäßige Syntax
- gleiche Darstellung für Code & Daten,

Lisp Syntax

- basiert nur auf Ausdrücken,
- Ausdruck = entweder Liste oder Atom,
- Liste = `( ...)`, z.B. `(1 2 "boo!")`
- Atom = einfaches Element, z.B. `"boo!"`, `42`, `:symbol`,
- einfache, regelmäßige Syntax
- gleiche Darstellung für Code & Daten,

Lisp Syntax

- basiert nur auf Ausdrücken,
- Ausdruck = entweder Liste oder Atom,
- Liste = `( ...)`, z.B. `(1 2 "boo!")`
- Atom = einfaches Element, z.B. `"boo!"`, `42`, `:symbol`,
- einfache, regelmäßige Syntax
- gleiche Darstellung für Code & Daten,

Lisp Syntax

- basiert nur auf Ausdrücken,
- Ausdruck = entweder Liste oder Atom,
- Liste = `( ...)`, z.B. `(1 2 "boo!")`
- Atom = einfaches Element, z.B. `"boo!"`, `42`, `:symbol`,
- einfache, regelmäßige Syntax
- gleiche Darstellung für Code & Daten,

Datentypen 1/2

- Listen,
- Hashtabellen,
- Zeichen (char),
- Strings (ggf. Unicode),
- Arrays,
- Fließkommazahlen
- structs (nutzerdefinierte strukturierte Datentypen)

Datentypen 1/2

- Listen,
- Hashtabellen,
- Zeichen (char),
- Strings (ggf. Unicode),
- Arrays,
- Fließkommazahlen
- structs (nutzerdefinierte strukturierte Datentypen)

Datentypen 1/2

- Listen,
- Hashtabellen,
- Zeichen (char),
- Strings (ggf. Unicode),
- Arrays,
- Fließkommazahlen
- structs (nutzerdefinierte strukturierte Datentypen)

Datentypen 1/2

- Listen,
- Hashtabellen,
- Zeichen (char),
- Strings (ggf. Unicode),
- Arrays,
- Fließkommazahlen
- structs (nutzerdefinierte strukturierte Datentypen)

Datentypen 1/2

- Listen,
- Hashtabellen,
- Zeichen (char),
- Strings (ggf. Unicode),
- Arrays,
- Fließkommazahlen
- structs (nutzerdefinierte strukturierte Datentypen)

Datentypen 1/2

- Listen,
- Hashtabellen,
- Zeichen (char),
- Strings (ggf. Unicode),
- Arrays,
- Fließkommazahlen
- structs (nutzerdefinierte strukturierte Datentypen)

Datentypen 1/2

- Listen,
- Hashtabellen,
- Zeichen (char),
- Strings (ggf. Unicode),
- Arrays,
- Fließkommazahlen
- structs (nutzerdefinierte strukturierte Datentypen)

Datentypen 2/2

- Symbole,
- Integer & bignums (*beliebig* große ganze Zahlen, transparent),
- rationale Zahlen: `1/3`, `(+ 1/3 1/3 1/3)` $\Longrightarrow$ `1`,
- komplexe Zahlen,
- Funktionen als vollwertiger Datentyp,

Datentypen 2/2

- Symbole,
- Integer & bignums (*beliebig* große ganze Zahlen, transparent),
- rationale Zahlen: `1/3`, `(+ 1/3 1/3 1/3)` $\Longrightarrow$ `1`,
- komplexe Zahlen,
- Funktionen als vollwertiger Datentyp,

Datentypen 2/2

- Symbole,
- Integer & bignums (*beliebig* große ganze Zahlen, transparent),
- rationale Zahlen: `1/3`, `(+ 1/3 1/3 1/3)` $\Longrightarrow$ `1`,
- komplexe Zahlen,
- Funktionen als vollwertiger Datentyp,

Datentypen 2/2

- Symbole,
- Integer & bignums (*beliebig* große ganze Zahlen, transparent),
- rationale Zahlen: `1/3`, `(+ 1/3 1/3 1/3)` $\Longrightarrow$ `1`,
- komplexe Zahlen,
- Funktionen als vollwertiger Datentyp,

Datentypen 2/2

- Symbole,
- Integer & bignums (*beliebig* große ganze Zahlen, transparent),
- rationale Zahlen: `1/3`, `(+ 1/3 1/3 1/3)` $\Longrightarrow$ `1`,
- komplexe Zahlen,
- Funktionen als vollwertiger Datentyp,

S-Expressions

- symbolische Ausdrücke,
- Listenstruktur,
- Prefix-Notation
- externe Darstellung für Code *und* Daten,
- interne Darstellung: CONS-Zellen:
  - `("Lisp" LINUX-TAG 2005 :VORTRAG)`
  -

S-Expressions

- symbolische Ausdrücke,
- Listenstruktur,
- Prefix-Notation
- externe Darstellung für Code *und* Daten,
- interne Darstellung: CONS-Zellen:
  - `("Lisp" LINUX-TAG 2005 :VORTRAG)`

S-Expressions

- symbolische Ausdrücke,
- Listenstruktur,
- Prefix-Notation
- externe Darstellung für Code *und* Daten,
- interne Darstellung: CONS-Zellen:
  - `("Lisp" LINUX-TAG 2005 :VORTRAG)`

S-Expressions

- symbolische Ausdrücke,
- Listenstruktur,
- Prefix-Notation
- externe Darstellung für Code *und* Daten,
- interne Darstellung: CONS-Zellen:
  - `("Lisp" LINUX-TAG 2005 :VORTRAG)`

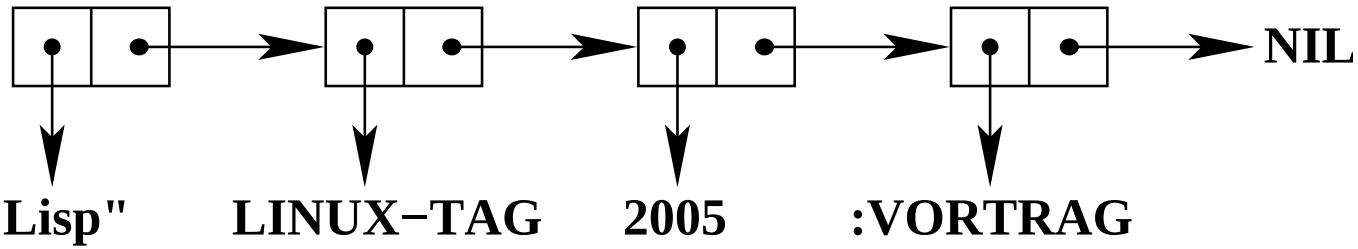
S-Expressions

- symbolische Ausdrücke,
- Listenstruktur,
- Prefix-Notation
- externe Darstellung für Code *und* Daten,
- interne Darstellung: CONS-Zellen:
  - `("Lisp" LINUX-TAG 2005 :VORTRAG)`

S-Expressions

- symbolische Ausdrücke,
- Listenstruktur,
- Prefix-Notation
- externe Darstellung für Code *und* Daten,
- interne Darstellung: CONS-Zellen:
  - `("Lisp" LINUX-TAG 2005 :VORTRAG)`

S-Expressions

- symbolische Ausdrücke,
- Listenstruktur,
- Prefix-Notation
- externe Darstellung für Code *und* Daten,
- interne Darstellung: CONS-Zellen:
  - `("Lisp" LINUX-TAG 2005 :VORTRAG)`
  - 

REPL

- REPL = **R**ead **E**val **P**rint **L**oop,
- interaktive Schnittstelle zu Lisp,
- * (+ 1 2 3 4 5 6)
  ⟹ 21
  * (print "hello, world")
  ⟹ "hello, world"
  "hello, world"
- `(loop (print (eval (read))))`

REPL

- REPL = **R**ead **E**val **P**rint **L**oop,
- interaktive Schnittstelle zu Lisp,
- * (+ 1 2 3 4 5 6)
  ⟹ 21
  * (print "hello, world")
  ⟹ "hello, world"
  "hello, world"
- `(loop (print (eval (read))))`

REPL

- REPL = **R**ead **E**val **P**rint **L**oop,
- interaktive Schnittstelle zu Lisp,
- * (+ 1 2 3 4 5 6)
  $\Longrightarrow$ 21
  * (print "hello, world")
  $\Longrightarrow$ "hello, world"
  "hello, world"
- `(loop (print (eval (read))))`

REPL

- REPL = **R**ead **E**val **P**rint **L**oop,
- interaktive Schnittstelle zu Lisp,
- * (+ 1 2 3 4 5 6)
  $\Longrightarrow$ 21
  * (print "hello, world")
  $\Longrightarrow$ "hello, world"
  "hello, world"
- `(loop (print (eval (read))))`

Lisp-Crashkurs: überall Klammern

- Darstellung von Code & Daten als S-Expressions,
- leere Liste: `()`,
- einfache Liste: `(1 2 3 4)`,
- strukturierte Liste: `(sum (1 2 3 4))`,
- Funktionsdefinition:

```
(defun !  (x)(if(eql x 0) 1 (* x(!(- x 1)))))
```

- Funktionsdefinition, mit Einrückung:

```
(defun !  (x)
  (if (eql x 0)
    1
    (* x (!  (- x 1)))))
```

Lisp-Crashkurs: überall Klammern

- Darstellung von Code & Daten als S-Expressions,
- leere Liste: `()`,
- einfache Liste: `(1 2 3 4)`,
- strukturierte Liste: `(sum (1 2 3 4))`,
- Funktionsdefinition:

```
(defun !  (x)(if(eql x 0) 1 (* x(!(- x 1)))))
```

- Funktionsdefinition, mit Einrückung:

```
(defun !  (x)
  (if (eql x 0)
    1
    (* x (!  (- x 1)))))
```

Lisp-Crashkurs: überall Klammern

- Darstellung von Code & Daten als S-Expressions,
- leere Liste: `()`,
- einfache Liste: `(1 2 3 4)`,
- strukturierte Liste: `(sum (1 2 3 4))`,
- Funktionsdefinition:

```
(defun !  (x)(if(eql x 0) 1 (* x(!(- x 1)))))
```

- Funktionsdefinition, mit Einrückung:

```
(defun !  (x)
  (if (eql x 0)
    1
    (* x (!  (- x 1)))))
```

Lisp-Crashkurs: überall Klammern

- Darstellung von Code & Daten als S-Expressions,
- leere Liste: `()`,
- einfache Liste: `(1 2 3 4)`,
- strukturierte Liste: `(sum (1 2 3 4))`,
- Funktionsdefinition:

```
(defun !  (x)(if(eql x 0) 1 (* x(!(- x 1)))))
```

- Funktionsdefinition, mit Einrückung:

```
(defun !  (x)
  (if (eql x 0)
    1
    (* x (!  (- x 1)))))
```

Lisp-Crashkurs: überall Klammern

- Darstellung von Code & Daten als S-Expressions,
- leere Liste: `()`,
- einfache Liste: `(1 2 3 4)`,
- strukturierte Liste: `(sum (1 2 3 4))`,
- Funktionsdefinition:

```
(defun !  (x)(if(eql x 0) 1 (* x(!(- x 1)))))
```

- Funktionsdefinition, mit Einrückung:

```
(defun !  (x)
  (if (eql x 0)
    1
    (* x (!  (- x 1)))))
```

Lisp-Crashkurs: überall Klammern

- Darstellung von Code & Daten als S-Expressions,
- leere Liste: `()`,
- einfache Liste: `(1 2 3 4)`,
- strukturierte Liste: `(sum (1 2 3 4))`,
- Funktionsdefinition:
  ```
  (defun !  (x)(if(eql x 0) 1 (* x(!(- x 1)))))
  ```
- Funktionsdefinition, mit Einrückung:
  ```
  (defun !  (x)
    (if (eql x 0)
      1
      (* x (!  (- x 1)))))
  ```

Lisp-Crashkurs: Auswertung von Ausdrücken

- Auswertung von Atomen $\Longrightarrow$ Wert des Atoms,
  - `* 23`
    $\Longrightarrow$ `23`
  - `* "This is a test"`
    $\Longrightarrow$ `"This is a test"`
- Auswertung von Ausdrücken = Funktionsaufruf,
  - `* (+ 5 23)`
    $\Longrightarrow$ `28`
  - `* (numberp 23)`
    $\Longrightarrow$ `T`
  - ```
    * (defun hello (name)
        (format t "Hello, Mr.  ~a!~%" name))
    ```
    $\Longrightarrow$ `HELLO`
    ```

Lisp-Crashkurs: Auswertung von Ausdrücken

- Auswertung von Atomen $\Longrightarrow$ Wert des Atoms,
  - `* 23`
    $\Longrightarrow$ 23
  - `* "This is a test"`
    $\Longrightarrow$ `"This is a test"`
- Auswertung von Ausdrücken = Funktionsaufruf,
  - `* (+ 5 23)`
    $\Longrightarrow$ 28
  - `* (numberp 23)`
    $\Longrightarrow$ T
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ HELLO

Lisp-Crashkurs: Auswertung von Ausdrücken

- Auswertung von Atomen $\Longrightarrow$ Wert des Atoms,
  - `* 23`
    $\Longrightarrow$ 23
  - `* "This is a test"`
    $\Longrightarrow$ `"This is a test"`
- Auswertung von Ausdrücken = Funktionsaufruf,
  - `* (+ 5 23)`
    $\Longrightarrow$ 28
  - `* (numberp 23)`
    $\Longrightarrow$ T
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ HELLO

Lisp-Crashkurs: Auswertung von Ausdrücken

- Auswertung von Atomen $\Longrightarrow$ Wert des Atoms,
  - `* 23`
    $\Longrightarrow$ 23
  - `* "This is a test"`
    $\Longrightarrow$ `"This is a test"`
- Auswertung von Ausdrücken = Funktionsaufruf,
  - `* (+ 5 23)`
    $\Longrightarrow$ 28
  - `* (numberp 23)`
    $\Longrightarrow$ T
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ HELLO

Lisp-Crashkurs: Auswertung von Ausdrücken

- Auswertung von Atomen $\Longrightarrow$ Wert des Atoms,
  - `* 23`
    $\Longrightarrow$ 23
  - `* "This is a test"`
    $\Longrightarrow$ `"This is a test"`
- Auswertung von Ausdrücken = Funktionsaufruf,
  - `* (+ 5 23)`
    $\Longrightarrow$ 28
  - `* (numberp 23)`
    $\Longrightarrow$ `T`
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ `HELLO`

Lisp-Crashkurs: Auswertung von Ausdrücken

- Auswertung von Atomen $\Longrightarrow$ Wert des Atoms,
  - `* 23`
    $\Longrightarrow$ `23`
  - `* "This is a test"`
    $\Longrightarrow$ `"This is a test"`
- Auswertung von Ausdrücken = Funktionsaufruf,
  - `* (+ 5 23)`
    $\Longrightarrow$ `28`
  - `* (numberp 23)`
    $\Longrightarrow$ `T`
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ `HELLO`

Lisp-Crashkurs: Auswertung von Ausdrücken

- Auswertung von Atomen $\Longrightarrow$ Wert des Atoms,
  - `* 23`
    $\Longrightarrow$ `23`
  - `* "This is a test"`
    $\Longrightarrow$ `"This is a test"`
- Auswertung von Ausdrücken = Funktionsaufruf,
  - `* (+ 5 23)`
    $\Longrightarrow$ `28`
  - `* (numberp 23)`
    $\Longrightarrow$ `T`
  - ```
    * (defun hello (name)
        (format t "Hello, Mr.  ~a!~%" name))
    ```
    $\Longrightarrow$ `HELLO`
    ```

Lisp-Crashkurs: Zusammenbauen & Zerlegen

- Liste erzeugen:
  `* (list 5 23 42)`
  $\Longrightarrow$ `(5 23 42)`
- Liste zerlegen:
  - `* (first (list 5 23 42))`
    $\Longrightarrow$ `5`
  - `* (rest (list 5 23 42))`
    $\Longrightarrow$ `(23 42)`

Lisp-Crashkurs: Zusammenbauen & Zerlegen

- Liste erzeugen:
  `* (list 5 23 42)`
  $\Longrightarrow$ `(5 23 42)`
- Liste zerlegen:
  - `* (first (list 5 23 42))`
    $\Longrightarrow$ `5`
  - `* (rest (list 5 23 42))`
    $\Longrightarrow$ `(23 42)`

Lisp-Crashkurs: Zusammenbauen & Zerlegen

- Liste erzeugen:
  `* (list 5 23 42)`
  $\Longrightarrow$ `(5 23 42)`
- Liste zerlegen:
  - `* (first (list 5 23 42))`
    $\Longrightarrow$ `5`
  - `* (rest (list 5 23 42))`
    $\Longrightarrow$ `(23 42)`

Lisp-Crashkurs: Zusammenbauen & Zerlegen

- Liste erzeugen:
  `* (list 5 23 42)`
  $\Longrightarrow$ `(5 23 42)`
- Liste zerlegen:
  - `* (first (list 5 23 42))`
    $\Longrightarrow$ `5`
  - `* (rest (list 5 23 42))`
    $\Longrightarrow$ `(23 42)`

Lisp-Crashkurs: Symbole 1/2

- Liste von Symbolen: `(red green blue)`
- Symbole: eindeutige Namen,
- Symbolname nicht case-sensitive,
- Symbole vergleichen:
  ```
  * (eq 'red 'RED)
  ⟹ T
  ```
- Symbol-Werte:
  - ```
    * (setf color "red")
    ⟹ "red"
    ```
  - ```
    * (setf magic-numbers (list 5 7 23 42 666))
    ⟹ (5 7 23 42 666)
    ```

Lisp-Crashkurs: Symbole 1/2

- Liste von Symbolen: `(red green blue)`
- Symbole: eindeutige Namen,
- Symbolname nicht case-sensitive,
- Symbole vergleichen:
  ```
  * (eq 'red 'RED)
  ⟹ T
  ```
- Symbol-Werte:
  - ```
    * (setf color "red")
    ⟹ "red"
    ```
  - ```
    * (setf magic-numbers (list 5 7 23 42 666))
    ⟹ (5 7 23 42 666)
    ```

Lisp-Crashkurs: Symbole 1/2

- Liste von Symbolen: `(red green blue)`
- Symbole: eindeutige Namen,
- Symbolname nicht case-sensitive,
- Symbole vergleichen:
  ```
  * (eq 'red 'RED)
  ⟹ T
  ```
- Symbol-Werte:
  - ```
    * (setf color "red")
    ⟹ "red"
    ```
  - ```
    * (setf magic-numbers (list 5 7 23 42 666))
    ⟹ (5 7 23 42 666)
    ```

Lisp-Crashkurs: Symbole 1/2

- Liste von Symbolen: `(red green blue)`
- Symbole: eindeutige Namen,
- Symbolname nicht case-sensitive,
- Symbole vergleichen:
  `* (eq 'red 'RED)`
  $\Longrightarrow$ `T`
- Symbol-Werte:
  - `* (setf color "red")`
    $\Longrightarrow$ `"red"`
  - `* (setf magic-numbers (list 5 7 23 42 666))`
    $\Longrightarrow$ `(5 7 23 42 666)`

Lisp-Crashkurs: Symbole 1/2

- Liste von Symbolen: `(red green blue)`
- Symbole: eindeutige Namen,
- Symbolname nicht case-sensitive,
- Symbole vergleichen:
  `* (eq 'red 'RED)`
  $\Longrightarrow$ `T`
- Symbol-Werte:
  - `* (setf color "red")`
    $\Longrightarrow$ `"red"`
  - `* (setf magic-numbers (list 5 7 23 42 666))`
    $\Longrightarrow$ `(5 7 23 42 666)`

Lisp-Crashkurs: Symbole 1/2

- Liste von Symbolen: `(red green blue)`
- Symbole: eindeutige Namen,
- Symbolname nicht case-sensitive,
- Symbole vergleichen:
  `* (eq 'red 'RED)`
  $\Longrightarrow$ `T`
- Symbol-Werte:
  - `* (setf color "red")`
    $\Longrightarrow$ `"red"`
  - `* (setf magic-numbers (list 5 7 23 42 666))`
    $\Longrightarrow$ `(5 7 23 42 666)`

Lisp-Crashkurs: Symbole 1/2

- Liste von Symbolen: `(red green blue)`
- Symbole: eindeutige Namen,
- Symbolname nicht case-sensitive,
- Symbole vergleichen:
  `* (eq 'red 'RED)`
  $\Longrightarrow$ `T`
- Symbol-Werte:
  - `* (setf color "red")`
    $\Longrightarrow$ `"red"`
  - `* (setf magic-numbers (list 5 7 23 42 666))`
    $\Longrightarrow$ `(5 7 23 42 666)`

Lisp-Crashkurs: Symbole 2/2

- Symbol mit Funktion *und* Wert:
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    `⟹ HELLO`
  - `* (setf hello "hi!")`
    `⟹ "Hi!"`
  - `* (hello "Joe")`
    `⟹ Hello, Mr. Joe!`
  - `* hello`
    `⟹ "Hi!"`

Lisp-Crashkurs: Symbole 2/2

- Symbol mit Funktion *und* Wert:
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ `HELLO`
  - `* (setf hello "hi!")`
    $\Longrightarrow$ `"Hi!"`
  - `* (hello "Joe")`
    $\Longrightarrow$ `Hello, Mr. Joe!`
  - `* hello`
    $\Longrightarrow$ `"Hi!"`

Lisp-Crashkurs: Symbole 2/2

- Symbol mit Funktion *und* Wert:
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ `HELLO`
  - `* (setf hello "hi!")`
    $\Longrightarrow$ `"Hi!"`
  - `* (hello "Joe")`
    $\Longrightarrow$ `Hello, Mr. Joe!`
  - `* hello`
    $\Longrightarrow$ `"Hi!"`

Lisp-Crashkurs: Symbole 2/2

- Symbol mit Funktion *und* Wert:
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ `HELLO`
  - `* (setf hello "hi!")`
    $\Longrightarrow$ `"Hi!"`
  - `* (hello "Joe")`
    $\Longrightarrow$ `Hello, Mr. Joe!`
  - `* hello`
    $\Longrightarrow$ `"Hi!"`

Lisp-Crashkurs: Symbole 2/2

- Symbol mit Funktion *und* Wert:
  - `* (defun hello (name)`
    `(format t "Hello, Mr. ~a!~%" name))`
    $\Longrightarrow$ `HELLO`
  - `* (setf hello "hi!")`
    $\Longrightarrow$ `"Hi!"`
  - `* (hello "Joe")`
    $\Longrightarrow$ `Hello, Mr. Joe!`
  - `* hello`
    $\Longrightarrow$ `"Hi!"`

Lisp-Crashkurs: Funktionsdefinition

- ```
  * (defun warm-enoughp (temperature)
      (let ((nice-and-cozy 25))
        (cond ((= temperature nice-and-cozy) t)
              ((> temperature nice-and-cozy) :hot)
              ((< temperature nice-and-cozy) :cold))))
  ```
  $\Longrightarrow$ WARM-ENOUGHP
- (warm-enoughp 23)
  $\Longrightarrow$:COLD)
- (warm-enoughp 30)
  $\Longrightarrow$:HOT)
- (warm-enoughp 25)
  $\Longrightarrow$ T)
  ```

Lisp-Crashkurs: Funktionsdefinition

- * (defun warm-enoughp (temperature)
    (let ((nice-and-cozy 25))
      (cond ((= temperature nice-and-cozy) t)
            ((> temperature nice-and-cozy) :hot)
            ((< temperature nice-and-cozy) :cold))))
  $\Longrightarrow$ WARM-ENOUGHP
- (warm-enoughp 23)
  $\Longrightarrow$:COLD)
- (warm-enoughp 30)
  $\Longrightarrow$:HOT)
- (warm-enoughp 25)
  $\Longrightarrow$ T)

Lisp-Crashkurs: Funktionsdefinition

- * (defun warm-enoughp (temperature)
    (let ((nice-and-cozy 25))
      (cond ((= temperature nice-and-cozy) t)
            ((> temperature nice-and-cozy) :hot)
            ((< temperature nice-and-cozy) :cold))))
  ⟹ WARM-ENOUGHP
- (warm-enoughp 23)
  ⟹ :COLD)
- (warm-enoughp 30)
  ⟹ :HOT)
- (warm-enoughp 25)
  ⟹ T)

Lisp-Crashkurs: Funktionsdefinition

- ```
  * (defun warm-enoughp (temperature)
      (let ((nice-and-cozy 25))
        (cond ((= temperature nice-and-cozy) t)
              ((> temperature nice-and-cozy) :hot)
              ((< temperature nice-and-cozy) :cold))))
  ```
  $\Longrightarrow$ WARM-ENOUGHP
- (warm-enoughp 23)
  $\Longrightarrow$:COLD)
- (warm-enoughp 30)
  $\Longrightarrow$:HOT)
- (warm-enoughp 25)
  $\Longrightarrow$ T)
  ```

Lisp-Crashkurs: anonyme Funktionen 1/2

- Funktionen in Lisp: first class objects,
  $\longrightarrow$ Verwendung als Daten,
- anonyme Funktionen definiert mit `(lambda ...)`,
- Beispiel:

```
(setf stuff '(#\y "John Doe" 23 :try-again
              startup 1/3 #C(1 2) 1234567890))
(format t "~{~S~%~}"
        (mapcar #'(lambda (item)
                    (format nil "(~a:  ~a)"
                            item
                            (type-of item)))
                stuff))
```

Lisp-Crashkurs: anonyme Funktionen 1/2

- Funktionen in Lisp: first class objects,
  $\longrightarrow$ Verwendung als Daten,
- anonyme Funktionen definiert mit `(lambda ...)`,
- Beispiel:

```
(setf stuff '(#\y "John Doe" 23 :try-again
              startup 1/3 #C(1 2) 1234567890))
(format t "~{~S~%~}"
        (mapcar #'(lambda (item)
                    (format nil "(~a:  ~a)"
                            item
                            (type-of item)))
                stuff))
```

Lisp-Crashkurs: anonyme Funktionen 1/2

- Funktionen in Lisp: first class objects,
  $\longrightarrow$ Verwendung als Daten,
- anonyme Funktionen definiert mit `(lambda ...)`,
- Beispiel:

```
(setf stuff '(#\y "John Doe" 23 :try-again
              startup 1/3 #C(1 2) 1234567890))
(format t "~{~S~%~}"
        (mapcar #'(lambda (item)
                    (format nil "(~a:  ~a)"
                            item
                            (type-of item)))
                stuff))
```

Lisp-Crashkurs: anonyme Funktionen 1/2

- Funktionen in Lisp: first class objects,
  $\longrightarrow$ Verwendung als Daten,
- anonyme Funktionen definiert mit `(lambda ...)`,
- Beispiel:

```
(setf stuff '(#\y "John Doe" 23 :try-again
              startup 1/3 #C(1 2) 1234567890))
(format t "~{~S~%~}"
        (mapcar #'(lambda (item)
                    (format nil "(~a:  ~a)"
                            item
                            (type-of item)))
                stuff))
```

Lisp-Crashkurs: anonyme Funktionen 1/2

```
⟹ "(y:  BASE-CHAR)"
"(John Doe:  (SIMPLE-BASE-STRING 8))"
"(23:  FIXNUM)"
"(TRY-AGAIN: SYMBOL)"
"(STARTUP: SYMBOL)"
"(1/3:  RATIO)"
"(#C(1 2):  COMPLEX)"
"(1234567890:  BIGNUM)"
NIL
```

Macros

- ```
  (defmacro with-db-connection ((con &rest open-args)
                                &body body)
    `(let ((,con (db-connect ,@open-args)))
       (unwind-protect
            (progn ,@body)
         (when ,con (db-disconnect ,con)))))
  ```
- ```
  (with-db-connection (db-link host db user pass)
    (update-tables db-link user-info)
    (show-stats (get-user-stats db-link)))
  ```
- ```
  (let ((db-link (db-connect host db user pass)))
    (unwind-protect
         (progn (update-tables db-link user-info)
                (show-stats (get-user-stats db-link)))
      (when db-link (db-disconnect db-link))))
  ```
  ```

Macros

- ```
  (defmacro with-db-connection ((con &rest open-args)
                                &body body)
    `(let ((,con (db-connect ,@open-args)))
       (unwind-protect
            (progn ,@body)
         (when ,con (db-disconnect ,con)))))
  ```
- ```
  (with-db-connection (db-link host db user pass)
    (update-tables db-link user-info)
    (show-stats (get-user-stats db-link)))
  ```
- ```
  (let ((db-link (db-connect host db user pass)))
    (unwind-protect
         (progn (update-tables db-link user-info)
                (show-stats (get-user-stats db-link)))
      (when db-link (db-disconnect db-link))))
  ```
  ```

Macros

- ```
  (defmacro with-db-connection ((con &rest open-args)
                                &body body)
    `(let ((,con (db-connect ,@open-args)))
      (unwind-protect
           (progn ,@body)
        (when ,con (db-disconnect ,con)))))
  ```
- ```
  (with-db-connection (db-link host db user pass)
    (update-tables db-link user-info)
    (show-stats (get-user-stats db-link)))
  ```
- ```
  (let ((db-link (db-connect host db user pass)))
    (unwind-protect
         (progn (update-tables db-link user-info)
                (show-stats (get-user-stats db-link)))
      (when db-link (db-disconnect db-link))))
  ```
  ```

Lisp-Crashkurs: Text-I/O

- einfache Textausgabe:

```
* (princ "Lisp rocks")
⟹ Lisp rocks
"Lisp rocks"
```

- formatierte Textausgabe:

```
* (format t "~a is cool~%" "Lisp")
⟹ Lisp is cool
NIL
```

- einfache Texteingabe:

```
* (read-line)
Common Lisp
⟹ "Common Lisp" ; NIL
```

Lisp-Crashkurs: Text-I/O

- einfache Textausgabe:

```
* (princ "Lisp rocks")
⟹ Lisp rocks
"Lisp rocks"
```

- formatierte Textausgabe:

```
* (format t "~a is cool~%" "Lisp")
⟹ Lisp is cool
NIL
```

- einfache Texteingabe:

```
* (read-line)
Common Lisp
⟹ "Common Lisp" ; NIL
```

Lisp-Crashkurs: Text-I/O

- einfache Textausgabe:

```
* (princ "Lisp rocks")
⟹ Lisp rocks
"Lisp rocks"
```

- formatierte Textausgabe:

```
* (format t "~a is cool~%" "Lisp")
⟹ Lisp is cool
NIL
```

- einfache Texteingabe:

```
* (read-line)
Common Lisp
⟹ "Common Lisp" ; NIL
```

Lisp-Crashkurs: Hilfsmittel

- tracing:

```
* (trace !)
⟹ ;; Tracing function !.
* (!  2)
⟹ 1.  Trace:  (!  '2)
2.  Trace:  (!  '1)
3.  Trace:  (!  '0)
3.  Trace:  !  ==> 1
2.  Trace:  !  ==> 1
1.  Trace:  !  ==> 2
2
```

- timing:

```
* (time (!  1000))
⟹ Real time:  0.013661 sec.
Run time:  0.01 sec.
Space:  498844 Bytes
402387260077093773543702433 9...
```

Lisp-Crashkurs: Hilfsmittel

- tracing:

```
* (trace !)
⟹ ;; Tracing function !.
* (!  2)
⟹ 1.  Trace:  (!  '2)
2.  Trace:  (!  '1)
3.  Trace:  (!  '0)
3.  Trace:  !  ==> 1
2.  Trace:  !  ==> 1
1.  Trace:  !  ==> 2
2
```

- timing:

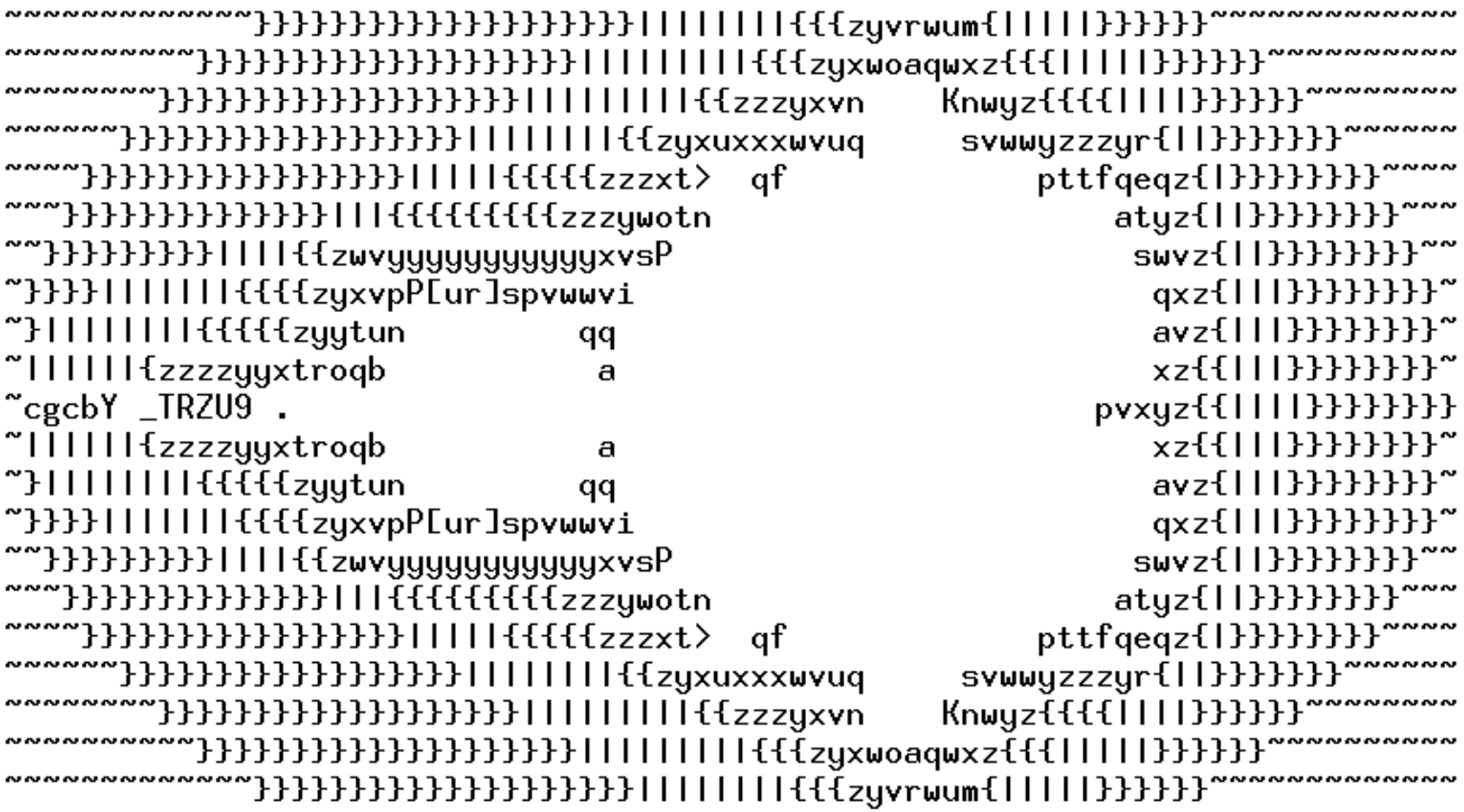
```
* (time (!  1000))
⟹ Real time:  0.013661 sec.
Run time:  0.01 sec.
Space:  498844 Bytes
402387260077093773543702433 9...
```

Lisp-Schnipsel

```
(loop for y from -1 to 1.1 by 0.1 do
      (loop for x from -2 to 1 by 0.04 do
            (let* ((c 126)
                   (z (complex x y))
                   (a z))
              (loop while (< (abs
                              (setq z (+ (* z z) a)))
                             2)
                    while (> (decf c) 32))
              (princ (code-char c))))
      (format t "~%"))
```

Autor: Frank Buß <fb@frank-buss.de>

Apfelmännchen



Weiterführendes - Lispsysteme

- Freie Lispsysteme (Auszug):
  - CLISP (UNIX, Win, Mac): `http://clisp.cons.org/`
  - CMUCL (UNIX): `http://www.cons.org/cmucl/`
  - SBCL (UNIX, Mac): `http://www.sbcl.org/`
- Kommerzielle Lispsysteme (Auszug):
  - Corman Lisp (Win): `http://www.cormanlisp.com/`,
  - Allegro CL (Unix, Win, Mac): `http://www.franz.com/`
  - LispWorks (Unix, Win, Mac): `http://www.lispworks.com`

Weiterführendes - Lispsysteme

- Freie Lispsysteme (Auszug):
  - CLISP (UNIX, Win, Mac): `http://clisp.cons.org/`
  - CMUCL (UNIX): `http://www.cons.org/cmucl/`
  - SBCL (UNIX, Mac): `http://www.sbcl.org/`
- Kommerzielle Lispsysteme (Auszug):
  - Corman Lisp (Win): `http://www.cormanlisp.com/`,
  - Allegro CL (Unix, Win, Mac): `http://www.franz.com/`
  - LispWorks (Unix, Win, Mac): `http://www.lispworks.com`

Weiterführendes - Lispsysteme

- Freie Lispsysteme (Auszug):
  - CLISP (UNIX, Win, Mac): `http://clisp.cons.org/`
  - CMUCL (UNIX): `http://www.cons.org/cmucl/`
  - SBCL (UNIX, Mac): `http://www.sbcl.org/`
- Kommerzielle Lispsysteme (Auszug):
  - Corman Lisp (Win): `http://www.cormanlisp.com/`,
  - Allegro CL (Unix, Win, Mac): `http://www.franz.com/`
  - LispWorks (Unix, Win, Mac): `http://www.lispworks.com`

Weiterführendes - Lispsysteme

- Freie Lispsysteme (Auszug):
  - CLISP (UNIX, Win, Mac): `http://clisp.cons.org/`
  - CMUCL (UNIX): `http://www.cons.org/cmucl/`
  - SBCL (UNIX, Mac): `http://www.sbcl.org/`
- Kommerzielle Lispsysteme (Auszug):
  - Corman Lisp (Win): `http://www.cormanlisp.com/`,
  - Allegro CL (Unix, Win, Mac): `http://www.franz.com/`
  - LispWorks (Unix, Win, Mac): `http://www.lispworks.com`

Weiterführendes - Lispsysteme

- Freie Lispsysteme (Auszug):
  - CLISP (UNIX, Win, Mac): `http://clisp.cons.org/`
  - CMUCL (UNIX): `http://www.cons.org/cmucl/`
  - SBCL (UNIX, Mac): `http://www.sbcl.org/`
- Kommerzielle Lispsysteme (Auszug):
  - Corman Lisp (Win): `http://www.cormanlisp.com/`,
  - Allegro CL (Unix, Win, Mac): `http://www.franz.com/`
  - LispWorks (Unix, Win, Mac): `http://www.lispworks.com`

Weiterführendes - Lispsysteme

- Freie Lispsysteme (Auszug):
  - CLISP (UNIX, Win, Mac): `http://clisp.cons.org/`
  - CMUCL (UNIX): `http://www.cons.org/cmucl/`
  - SBCL (UNIX, Mac): `http://www.sbcl.org/`
- Kommerzielle Lispsysteme (Auszug):
  - Corman Lisp (Win): `http://www.cormanlisp.com/`,
  - Allegro CL (Unix, Win, Mac): `http://www.franz.com/`
  - LispWorks (Unix, Win, Mac): `http://www.lispworks.com`

Weiterführendes - Lispsysteme

- Freie Lispsysteme (Auszug):
  - CLISP (UNIX, Win, Mac): `http://clisp.cons.org/`
  - CMUCL (UNIX): `http://www.cons.org/cmucl/`
  - SBCL (UNIX, Mac): `http://www.sbcl.org/`
- Kommerzielle Lispsysteme (Auszug):
  - Corman Lisp (Win): `http://www.cormanlisp.com/`,
  - Allegro CL (Unix, Win, Mac): `http://www.franz.com/`
  - LispWorks (Unix, Win, Mac): `http://www.lispworks.com`

Weiterführendes - Lispsysteme

- Freie Lispsysteme (Auszug):
  - CLISP (UNIX, Win, Mac): `http://clisp.cons.org/`
  - CMUCL (UNIX): `http://www.cons.org/cmucl/`
  - SBCL (UNIX, Mac): `http://www.sbcl.org/`
- Kommerzielle Lispsysteme (Auszug):
  - Corman Lisp (Win): `http://www.cormanlisp.com/`,
  - Allegro CL (Unix, Win, Mac): `http://www.franz.com/`
  - LispWorks (Unix, Win, Mac): `http://www.lispworks.com`

Weiterführendes - Literatur

- Peter Seibel: “Practical Common Lisp”
  `http://www.gigamonkeys.com/book/`
- David S. Touretzky: “Common Lisp: A Gentle Introduction to Symbolic Computation”
- David B. Lamkins: “Successful Lisp”
- Lisperati: `http://www.lisperati.com/`
- Guy L. Steele: “Common Lisp the Language, 2nd ed.” (CLtL2)
- Common Lisp HyperSpec:
  `http://www.lispworks.com/documentation/HyperSpec/`
- Common Lisp Wiki: `http://www.cliki.net/index`

Weiterführendes - Literatur

- Peter Seibel: “Practical Common Lisp”
  `http://www.gigamonkeys.com/book/`
- David S. Touretzky: “Common Lisp: A Gentle Introduction to Symbolic Computation”
- David B. Lamkins: “Successful Lisp”
- Lisperati: `http://www.lisperati.com/`
- Guy L. Steele: “Common Lisp the Language, 2nd ed.” (CLtL2)
- Common Lisp HyperSpec:
  `http://www.lispworks.com/documentation/HyperSpec/`
- Common Lisp Wiki: `http://www.cliki.net/index`

Weiterführendes - Literatur

- Peter Seibel: “Practical Common Lisp”
  `http://www.gigamonkeys.com/book/`
- David S. Touretzky: “Common Lisp: A Gentle Introduction to Symbolic Computation”
- David B. Lamkins: “Successful Lisp”
- Lisperati: `http://www.lisperati.com/`
- Guy L. Steele: “Common Lisp the Language, 2nd ed.” (CLtL2)
- Common Lisp HyperSpec:
  `http://www.lispworks.com/documentation/HyperSpec/`
- Common Lisp Wiki: `http://www.cliki.net/index`

Weiterführendes - Literatur

- Peter Seibel: “Practical Common Lisp”
  `http://www.gigamonkeys.com/book/`
- David S. Touretzky: “Common Lisp: A Gentle Introduction to Symbolic Computation”
- David B. Lamkins: “Successful Lisp”
- Lisperati: `http://www.lisperati.com/`
- Guy L. Steele: “Common Lisp the Language, 2nd ed.” (CLtL2)
- Common Lisp HyperSpec:
  `http://www.lispworks.com/documentation/HyperSpec/`
- Common Lisp Wiki: `http://www.cliki.net/index`

Weiterführendes - Literatur

- Peter Seibel: “Practical Common Lisp” `http://www.gigamonkeys.com/book/`
- David S. Touretzky: “Common Lisp: A Gentle Introduction to Symbolic Computation”
- David B. Lamkins: “Successful Lisp”
- Lisperati: `http://www.lisperati.com/`
- Guy L. Steele: “Common Lisp the Language, 2nd ed.” (CLtL2)
- Common Lisp HyperSpec: `http://www.lispworks.com/documentation/HyperSpec/`
- Common Lisp Wiki: `http://www.cliki.net/index`

Weiterführendes - Literatur

- Peter Seibel: “Practical Common Lisp”
  `http://www.gigamonkeys.com/book/`
- David S. Touretzky: “Common Lisp: A Gentle Introduction to Symbolic Computation”
- David B. Lamkins: “Successful Lisp”
- Lisperati: `http://www.lisperati.com/`
- Guy L. Steele: “Common Lisp the Language, 2nd ed.” (CLtL2)
- Common Lisp HyperSpec:
  `http://www.lispworks.com/documentation/HyperSpec/`
- Common Lisp Wiki: `http://www.cliki.net/index`

Weiterführendes - Literatur

- Peter Seibel: “Practical Common Lisp”
  `http://www.gigamonkeys.com/book/`
- David S. Touretzky: “Common Lisp: A Gentle Introduction to Symbolic Computation”
- David B. Lamkins: “Successful Lisp”
- Lisperati: `http://www.lisperati.com/`
- Guy L. Steele: “Common Lisp the Language, 2nd ed.” (CLtL2)
- Common Lisp HyperSpec:
  `http://www.lispworks.com/documentation/HyperSpec/`
- Common Lisp Wiki: `http://www.cliki.net/index`

Fragen & Antworten

Fragen?

Ende

Vielen Dank für Ihr Interesse!