Letzte/Diese Vorlesung

Letzte Woche

- Was ist Programmieren? – Was ist ein Algorithmus?
- Grundelemente von Programmiersprachen; Datentypen
- (Extended) Backus-Naur Form (nach *John Backus* und *Peter Naur*)

Diese Woche

Lisp Geschichte: *damals…*

Lisp Gegenwart: *…und heute*

Lisp Syntax: wie schreibt man ein Lisp-Programm

Lisp Semantik: wie schreibt man ein funktionierendes Lisp-Programm

ADT: Abstrakte DatenTypen

Geschichte von LISP

1956	IPL	erstmals verwendet zur Strukturierung symbolischer Daten (Assembler-artige Sprache)
1958	LISP	**LIS**t **P**rocessor, entwickelt von John McCarthy als Aufsatz auf Fortran
1960	LISP1	Programmiersystem mit Interpreter, Compiler und Speicherverwaltung
1961	LISP1.5	Erster Lisp-Standard
19..	Maclisp, FranzLisp, Interlisp, TLC-Lisp, UCI-Lisp, NIL, Zeta-Lisp, Scheme	

1984 Common Lisp

1989 Common Lisp ANSI Standard (X3J13)

Anwendungsgebiete für LISP

- Formelmanipulation ($\rightarrow$ Macsyma)
- Modellbildung, Simulation ($\rightarrow$ Nintendo)
- Computergrafik ($\rightarrow$ Auto-CAD)
- Editoren ($\rightarrow$ Emacs)
- Entwicklungsumgebungen ($\rightarrow$ Genera)
- Systemprogrammierung ($\rightarrow$ Lisp Maschine)
- **Künstliche Intelligenz (KI):**
  Expertensysteme, Planungssysteme, Maschinelles Sehen, Automatisches Beweisen, Maschinelles Lernen
- **Computerlinguistik:**
  Morphologie, Syntaxanalyse (Parsing), Semantik, Sprach-Generierung

Lisp Syntax

Daten und **Programme** sind aufgebaut aus **symbolischen Ausdrücken**, die entweder Atome, oder aber Listen sind.

Atome *sind „wortähnliche“ Objekte:*

Numerische Atome (eng: numbers) oder Zahlen

```
1  2  3  -4  3.14159265358979  -7.5  6.02E+23
```

Symbole Hier sind praktisch alle Zeichenfolgen erlaubt außer z.B. Klammern, Anführungszeichen, Leerzeichen und einigen weiteren Satzzeichen.

```
Symbol  Sym23  noch-eins  t  false  usw  NoMen  %NOMEN%  *nomen*
```

Strings , d.h. Zeichenfolgen in hochgesetzten Anführungszeichen

```
"Hier ein String"  "&]$8$77?"  "setq"  "12.~a"

"Zitat: \"Ich habe fertig!\"  -- Giovanni Trapattoni"
```

Lisp Syntax

Listen *sind „satzähnliche“ Objekte:*

Eine Liste besteht aus einer öffnenden Klammer `(` gefolgt von beliebig vielen Listenelementen, die jeweils durch ein Leerzeichen getrennt sind, und einer schließenden Klammer `)`.

Jedes Listenelement ist entweder ein Atom oder wiederum eine Liste.

Beispiel:

```
(Dies ist eine Liste)   ((das) ((auch)))   ()    (((((((((())))))))))

(s (np (det n))
   (vp (v (np (det n))
          (pp (prep (np (n)))))))

(defun print-quadratzahlen (x)
  (when (plusp x)
     (print (* x x)) (print-quadratzahlen (- x 1)))))
```

Lisp Syntax

Alle Symbolische Ausdrücke (wie oben beschrieben) sind *syntaktisch* korrekte LISP-Programme! `:-)`

Aber... die meisten sind aus *semantischen* Gründen nicht korrekt! `:-(`

Lisp Semantik

Der Kern eines Lisp-Systems ist der **Interpreter**, dessen Aufgabe es ist, eingegebene symbolische Ausdrücke zu evaluieren (auszuwerten). Das Ergebnis, das (i.a.) auch ein symbolischer Ausdrück ist, wird nach der Evaluation ausgegeben.

Der Interpreter ist in den meisten LISP-Systeme in einer sogenannten

Read-Eval-Print—Schleife

eingebettet.

Lisp Semantik

Auswertung

Der Interpreter arbeitet nach 3 Regeln:

1 – Identität Eine Zahl, eine Zeichenfolge, oder die Symbole `t` und `nil`, evaluiert zu sich selbst.

Beispiel:

```
11.5  -->  11.5   "String"  -->  "String"
```

2 – Symbole Die Auswertung eines Symbols liefert den mit dem Symbol *assoziierten* Wert zurück. Ausnahmen sind die Symbole `t` und `nil`.

Beispiel:

```
*teilnehmer*  -->  (KARL EGON MARIA PETER)
*user* --> NIL
```

Lisp Semantik

3 – Listen Jede Liste stellt einen Funktionsaufruf dar. Dabei bezeichnet das erste Element die anzuwendende Funktion und die möglicherweise vorhandenen weiteren Elemente der Liste stellen die Argumente dar (→ Präfix-Notation). „Normalerweise“ werden die Argumente vor Anwendung der bezeichneten Prozedur evaluiert.

Die leere Liste, `()`, hat den Wert `NIL`

Beispiel:

```
(max (min 3 5) 4) --> 4
```

Mit jedem Symbol können also *zwei* Informationen „assoziiert“ sein:
Eine **Funktion** und ein **Wert**.

Man sagt auch: *Der Wert/Die Funktion ist an das Symbol* **gebunden**.

Lisp Semantik

Problem: Wie können Listen und Symbole als Daten behandelt werden?

Beispiel:

```
? (Peter geht nach Hause)
Fehler: nicht bekannte Funktion PETER
? Haus
Fehler: kein Wert fuer HAUS bekannt
```

Lösung: `quote` verhindert, daß ihre Argumente ausgewertet werden.

Beispiel:

```
? (quote (peter geht nach hause))
(PETER GEHT NACH HAUSE)
? (quote haus)
HAUS
```

Achtung: Bei Symbolen übersetzt LISP alles in GROSSBUCHSTABEN:

```
(eq 'aBcDE 'ABCDE)
```

Zitieren, „Quotierung“

Beispiele:

```
? (quote (max 5 4))
(max 5 4)
```

Aber:

```
? (max 5 4)
5
```

Abkürzung von `quote`: Der Aufruf **(quote pi)** kann durch **’pi** abgekürzt werden.

```
? '(peter geht nach hause)
(PETER GEHT NACH HAUSE)
? 'haus
HAUS
? '10
10
```

Lisp Semantik

Zuweisung und Auswertung

Die Funktion **set** weist Symbolen einen Wert zu. **set** liefert als Ergebnis den Wert des zweiten Arguments zurück, wird jedoch in erster Linie wegen des *Seiteneffekts* angewendet, d.h. der Veränderung des Interpreterzustandes.

Beispiel:

```
? (set 'foo 42)
42
? foo
42
? (set 'user 'tbecker)
TBECKER
? user
TBECKER
```

Lisp Semantik

Zuweisung und Auswertung

Der symbolische Ausdruck **(setq symbol expr)** ist eine Abkürzung für **(set 'symbol expr)** (Das erste Argument, der Name eines Symbols, wird dabei *nicht* evaluiert.).

Beispiele:

```
? (setq farbe 'rot)
rot
? farbe
rot
? (setq haarfarbe farbe)
????
? (setq max (max 3 2.5 -1))
3
```

Abstrakte DatenTypen – ADT

<u>"Definition" ADT</u>
Eine Sammlung von Datenstrukturen und verschiedenen Arten von Funktionen, die es ermöglichen, die Datenstrukturen zu verarbeiten, **ohne** daß man deren interne Struktur kennt.

Konstruktoren (z.B. `make-ding`)

Selektoren (z.B. `ding-first`)

Typprädikate (z.B. `ding-p`)

Andere Funktionen wie z.B `print`, `traverse`, `map`,...

ADT – Liste

Typprädikate

Zur Erkennung einer Liste gibt es eine funktion **listp**, die `T` oder `NIL` zurückgibt abhängig von typ der Parameter.

```
(listp '(1 2 3)) --> T
(listp '()) --> T
(listp 3) --> NIL
```

ADT – Liste

<u>Selektoren</u> Primitive Zugriffsfunktionen, die die interne Darstellung von Listen berücksichtigen sind **first** und **rest**. Die Funktion **first** angewandt auf eine Liste liefert deren erstes Element zurück, bzw. **nil** im Falle der leeren Liste. Die Funktion **rest** angewandt auf eine Liste gibt als Ergebnis die Liste ohne das erste Element zurück, bzw. **nil** im Falle der leeren Liste.

<u>Beispiele:</u>

```
(first '(a b c))       -->  A              (first '())  -->  NIL
(rest '(a))            -->  NIL            (rest '())   -->  NIL
(first '((a b) c))     -->  (A B)
(first '(nil b c))     -->  NIL
(rest '(a b c))        -->  (B C)
(rest '((a b) c))      -->  (C)
```

ADT – Liste

<u>Traversieren von Listen</u> Mit Sequenzen von **first**s und **rest**s können Listenstrukturen durchwandert (**traversiert**) werden.

```
(first (rest '(beachtet das quote)))  -->  das
(first (rest (rest '(beachtet das quote)))) --> quote
(first (rest (rest (rest '(beachtet das quote))))) --> NIL
(first (first '(beachtet das quote))) --> ???
```

Aus historischen Gründen gibt es zwei zusätzliche Namen für **first** und **rest** – **car** (contents of address register) bzw **cdr** (contents of decrement register); es gilt:

`(car` *liste*`)` ≡ `(first` *liste*`)` `(cdr` *liste*`)` ≡ `(rest` *liste*`)`

ADT – Liste

Selektoren

Neben **first** und **rest** stehen noch zahlreiche weitere Funktionen für den Listenzugriff zur Verfügung.

second, ... , tenth Selektiert das zweite, ... , zehnte Element aus einer Liste.

(nth *n liste***)** Selektiert das n+1-ste Element aus *liste*.

(nthcdr *n liste***)** Führt die **cdr**-Operation n-mal auf der *liste* aus.

last Selektiert den letzten **cdr** aus einer Liste, d.h. das letzte Element wird als Liste zurückgegeben.

Beispiele:

```
(second '(1 2 3)) --> 2
(nth 2 '(1 2 3)) --> 3
(nthcdr 2 '(1 2 3)) --> (3)
(last '(1 2 3)) --> (3)
```

ADT – Liste

Konstruktoren

In Analogie zu **first** und **rest** dient die Funktion **cons** dazu, durch Aufbau einer neuen Verkettung ein Element zu einer Liste hinzuzufügen.

```
(cons 'a '(b c))  -->  (A B C)
(cons '(a) '(b c))  -->  ((A) B C)
(cons (first '(1 2 3)) (rest '(1 2 3)))
     -->  (1 2 3)
```

Zwischen **first**, **rest** und **cons** steht folgende Beziehung:

$$\texttt{(cons (first <liste>) (rest <liste>))} \equiv \texttt{<liste>}$$

ADT – Liste

Konstruktoren

Neben **cons** stehen noch weitere Funktionen für den Listenkonstruktion zur Verfügung.

(list *arg-1 arg-2* **...** *arg-n***)** Erzeugt eine liste, bestehend aus den Argumenten *arg-i*.
`EBNF: (list` <*arg*>*`)`

(append *liste-1* **...** *liste-n***)** Erzeugt durch Konkatenation seine Argumente (die Listen sein müssen), eine neue Liste.

Beispiele:

```
(list 1 2 3) --> (1 2 3)
(list (list 1) 2 (list 1 2 3)) --> ((1) 2 (1 2 3))

(append '(1) (list 2)) --> (1 2)
(append (cons 1 (list 2)) nil (list 3)) --> (1 2 3)
```

ADT – Liste

Beispiele:

```
? (setq imperativ-satz '(kommt nach hause))
(kommt nach hause)
? (setq aussage-satz (cons 'ihr imperativ-satz))
(ihr kommt nach hause)
? (setq aussage-satz '(peter faehrt mit dem bus))
(peter faehrt mit dem bus)
? (setq relativ-satz '(der sich beeilen muss))
(der sich beeilen muss)
? (setq aussage-satz
            (cons (first aussage-satz)
                  (cons relativ-satz
                       (rest aussage-satz))))
(peter (der sich beeilen muss) faehrt mit dem bus)
```

ADT – Liste

Aufbau von Listen aus Atomen

Im Prinzip genügt **cons** zum Aufbau beliebig komplexer Listen aus Atomen und der leeren Liste:

Beispiele:

```
? (cons 'a (cons 'b (cons 'c nil)))
(a b c)
? (cons 'a (cons (cons 'c (cons 'b nil))
                 (cons 'd nil)))
(a (b c) d)
```

ADT – Liste

Noch zwei nützliche Funktionen:

(reverse <*liste*>**)** Erzeugt eine Liste mit den gleichen Elementen wie <*liste*>, aber in umgekehrter Reihenfolge.

(length <*liste*>**)** Liefert die Anzahl der Elemente von <*liste*>.

```
? (reverse '(a b c))
(c b a)
? (first (rest (rest (rest '(a b c d)))))
d
? (first (reverse '(a b c d)))
d
? (length '())
0
? (length '(((a b)) c)) --> ??
```

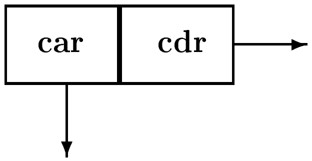
ADT – Liste

Schmök-aufgabe: Wie fügt man ein neues Element vor das letzte in einer Liste?

```
? ; vor das letzte Element einer Liste
? ; ein neues Element einfuegen
? (setq liste
        (reverse
          (cons (first (reverse liste))
                (cons 'neu
                      (rest (reverse liste))))))
```

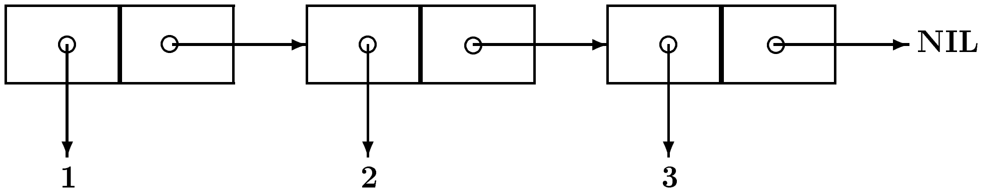
Interne Struktur – Liste

- Wird vom LISP-Reader bei Eingabe einer Liste automatisch erzeugt.
- variable Länge von Listen
  $\longrightarrow$ Realisiert durch Paare von Verweisen

  Cons-Element: 

- Die letzte Restliste ist **NIL**.
- `car`-Komponenten der Paare heißen Elemente der Liste.

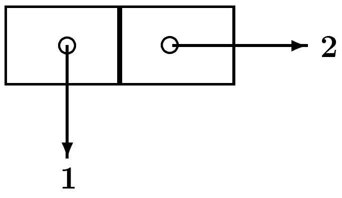
Beispiel: (`(1 2 3)`)



Interne Struktur – Liste

Punktierte Paare

Auch der cdr-pointer kann auf einem Atom zeigen. Das wird mit der schreibweise `(1 . 2)` – *ein punktiertes Paar*.



NIL

> Schmök-aufgabe: Versuch die interne Struktur als Lispausdruck zu schreiben (max 2 Minuten).