Letzte/Diese Vorlesung

Letzte Woche:

- Was ist Programmieren? – Was ist ein Algorithmus?
- Grundelemente von Programmiersprachen; Datentypen
- (extended) Bachus-Nauer Form

Heute:

Lisp Syntax wie schreibt man ein Lisp-programm

Lisp Semantik wie schreibt man ein funktionierendes Lisp-Programm

ADT Abstrakte DatenTypen

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  3.14  -7.5  6.02E+23
```

Symbole Hier sind praktisch alle Zeichenfolgen erlaubt ausser z.B. Klammern, Anführungszeichen und weitere Satzzeichen.

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

Strings , d.h. Zeichenfolgen in hochgesetzten Anführungszeichen

```
"Hier ein String"  "&]$787?"  "setq"  "12.~a"
```

Lisp Syntax

Listen Listen dienen zur Beschreibung satzähnlicher Objekte. Eine Liste besteht aus einer öffnenden Klammer gefolgt von beliebig vielen Listenelementen, die jeweils durch ein Leerzeichen getrennt sind, und einer schließenden Klammer.

Ein 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 ist es, eingegebene symbolische Ausdrücke zu evaluieren (auszuwerten). Das Ergebnis, was auch ein Symbolisches Ausdrück ist, wird nach dem evaluieren 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 – 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`

Lisp Semantik

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

Beispiel:

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


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
```

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
? a -->
42
? (set 'user 'janal) -->
JANAL
user -->
JANAL
```

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

"Definition" ADT
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

Selektoren 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.

Beispiele:

```
(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

Traversieren von Listen 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*`)` $\equiv$ `(first` *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)
```