Letzte/Diese Vorlesung

Letzte Woche:

- Abstrakte DatenTypen – **ADT** – Beispiel: Liste
  - Konstruktoren – **`cons, append, list, ...`**
  - Selektoren – **`first, rest, second, ...`**
  - Typprädikat – **`listp`**
- Eigene Funktionen definieren
  - **(defun (**<*param*>***)** <*form*>***)**
  - Symbole können gleichzeitig mit einem Wert und einer Funktion assoziert sein.
- Ausschnitt der Typhierarchie von Common-Lisp.

Letzte/Diese Vorlesung

Heute:

Listen als Mengen

Konstrollstrukturen

Rekursion

Bindungsumgebungen

Listen als Mengen

Eine Menge läßt sich sehr einfach als Liste darstellen. Zur Behandlung solcher Strukturen stellt LISP spezielle Funktionen bereit.

member Das Prädikat **member** testet, ob ein Objekt als Element in einer Menge auftritt. Das Ergebnis ist die Liste (Menge) ab dem Punkt an dem das Element sich befindet.

adjoin Der Konstruktor **adjoin** fügt ein Objekt neu in eine Menge ein, unter der Bedingung, daß das Objekt noch nicht Element der Menge ist.

Daneben stehen wichtige Mengenoperationen zur Verfügung: **subsetp**, **union**, **intersection**, **set-difference**, **set-exclusive-or**.

Listen als Mengen – Beispiele:

```
? (member 1 (list 2 3 4))
NIL
? (member 'b '(a b c))
(B C)
? (adjoin 'b '(a b c))
(A B C)
? (adjoin 'b '(a c))
(A B C)
? (union '(a b c) '(b c d))
(A B C D)
? (subsetp '(b d) '(a b c d))
T
```

Kontrollstrukturen

Elementare Kontrollstrukturen sind:

- Sequenzen
- Bedingungen
- Rekursion
- Iteration

Mit diesen Kontrollstrukturen können:

- Verzweigungspunkte im Kontrollfluß spezifiziert werden.
- Folgen von Anweisungen oder Funktionsaufrufe wiederholt durchgeführt werden.

Kontrollstrukturen – Sequenzen

Bei der Definition einer Funktion ist es erlaubt, eine *Sequenz* von symbolischen Ausdrücken (<*s-expr*> bzw <*form*>) zu schreiben

Beispiel:

```
(defun foo (x y)
  (bar x)
  (fum y)
  'ok)
```

Kontrollstrukturen – Bedingungen

Die fundamentale Funktion zur Beschreibung von bedingten Anweisungen in Lisp ist **cond**. Die allgemeine Form lautet:

(cond {(<*test*><*form*>*)}***)**

Die Klauseln werden nacheinander abgearbeitet, bis ein <*test*> einen Wert ungleich **nil** liefert. Erst danach werden die restlichen symbolischen Ausdrücke (<*form*>*) der betreffenden Klausel ausgewertet. Das Ergebnis eines **cond**-Aufrufs ist der Wert des zuletzt evaluierten Ausdrucks.

Beispiel:

```
(defun fie (x)
  (cond ((foo x) (bar x x))
        (t (fum x))))
```


Kontrollstrukturen – Beispiele

Schreibe eine Funktion, die eine Liste um ein Element erweitert, wenn das Element noch nicht vorhanden ist, und ansonsten die Liste unverändert zurückliefert. Das Element muß ein Symbol sein.

Lösung

```
? (defun mein-adjoin (element beutel)
     (cond ((symbolp element)
            (cond ((member element beutel) beutel)
                  (t (cons element beutel))))
           (t beutel)))
adjoin
? (mein-adjoin 'a '(a b c d)) -> (a b c d)
? (mein-adjoin 'f '(a b c d)) -> (f a b c d)
? (mein-adjoin 1 '(a b c d))  -> (a b c d)
```

Kontrollstrukturen – Beispiele

Schreibe eine Funktion `verbalisiere-ws`, die eine Wahrscheinlichkeit verbalisiert

```
? (verbalisiere-ws .4) -> unwahrscheinlich
? (verbalisiere-ws .2) -> sehr-unwahrscheinlich
? (verbalisiere-ws .8) -> sehr-wahrscheinlich
? (verbalisiere-ws .6) -> wahrscheinlich
```

Lösung

```
(defun verbalisiere-ws (ws)
   (cond ((> ws .75) 'sehr-wahrscheinlich)
         ((> ws .5) 'wahrscheinlich)
         ((> ws .25) 'unwahrscheinlich)
         (T 'sehr-unwahrscheinlich)))
```


Logische Operatoren

Die logischen Operatoren **(and** <*form*>***)**, **(or** <*form*>***)** und **(not** <*form*>**)**

```
? (not t)
NIL
? (not nil)
T
? (and 1 2 3)
3
? (or (equal 3 (+ 1 2)) nil 3)
T
```

Logische Operatoren

Die logischen Operatoren **and**, **or** und **not** können mit hilfe von **cond** ausgedrückt werden.

- `(not <form>)` ≡

```
(cond (<form> nil)
      (t t))
```

- `(and x y ... z)` ≡

```
(cond ((not x) nil) ((not y) nil) ...
       (t z)))
```

- `(or x y ... z)` ≡

```
(cond (x) (y) ...
      (t z))
```


Kontrollstrukturen – Beispiele

Schreibe eine Funktion, die, wenn die beide Argumente Zahlen sind, die Summe sine beiden Argumente zurückgibt, sonst nil

Lösung

```
(defun sum-wenn-beide-zahl (x y)
  (cond ((and (numberp x)
              (numberp y))
         (+ x y))
        (t nil)))
```

Rekursion

Was bedeutet rekursiv? Funktionen, die (zum Teil) durch sich selbst definiert sind, werden *rekursiv* genannt.

Vorgehen:

- Identifiziere und formuliere entsprechende Test(s) für *Basis–Situation(en)*, die ohne weitere Zerlegung gelöst werden können.
- Zerlege das Problem in *kleinere* Teile, so daß die gleiche Aufgabe für einfachere Problemgrößen angewendet werden kann.
- Gib an, wie die *Summe* der Einzelteile die Gesamtlösung bildet.

Rekursion – Beispiel (Fakultät)

Die Funktion *fakultät* ist definiert durch:

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{für } n \geq 1 \\ \textit{undefiniert} & \text{sonst} \end{cases}$$

Lösung

```
(defun ! (n)
  (cond ((= n 0) 1)
        (t (* n (! (1- n))))))
```

Rekursion – Beispiel (Fakultät)

```
? (! 3)
(cond ((= n 0) 1) (t (* n (! (1- n)))))
(cond ((= 3 0) 1) (t (* n (! (1- n)))))
(* 3 (! (1- 3)))))
(* 3 (! 2))
(* 3 (* 2 (! 1)))
(* 3 (* 2 (* 1 (! 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```


Rekursion – Beispiel

Schreibe eine eigene Implementation von `length`

```
(defun mein-length (liste)
  (cond ((null liste) 0)
        (t (1+ (mein-length (rest liste))))))
```

Schreibe eine eigene Implementation von `member`

```
(defun mein-member (ding liste)
  (cond ((endp liste) nil)
        ((equal ding (first liste)) liste)
        (t (mein-member ding (rest liste)))))
```

Rekursion – Beispiel

Was macht die folgende Funktion?

```
(defun spegel (list)
  (cond ((null list) nil)
        (t (append (spegel (rest list))
                   (list (first list))))))
```

Was macht die folgende Funktion?

```
(defun kul (n l)
  (cond ((equal n 0) (first l))
        (t (kul (1- n) (rest l)))))
```


Bindungsumgebungen

- Beim Aufruf einer Funktion werden die *formalen Parametern*, FP, der Funktion F mit den *aktuellen Parametern*, AP, ihres Aufrufmusters gebunden (Bindungsumgebung BU_f von f) .
- BU_f gilt nur für die Dauer der Auswertung von f.
- Wird in f selbst eine Funktion g aufgerufen, so wird eine neue Bindungsumgebung, BU_g, erzeugt.
- BU_g ist in BU_f eingebettet.
- Ist g abgearbeitet, wird BU_g gelöscht

Bindungsumgebungen – Beispiel

```
? (set 'foo 42)
42
? (defun foo (foo)
     (set 'foo (+ 42 foo)))
foo
? (foo 42)
84
? (foo foo)
84
? foo
42
```