

Praktische Algorithmen der Bioinformatik und Computerlinguistik mit Lisp

Funktionales Programmieren

12.05.2021

Übersicht

- Das Lambda-Kalkül
- Was sind funktionale Programmiersprachen?
- Funktionen als Werte
- Funktionen höherer Ordnung
- Map-Funktionen
- Anonyme Funktionen
- Parameterlisten:
 - Optionale Parameter
 - Schlüsselwort-Argument

Das Lambda-Kalkül



- Lambda-Kalkül ist Basis für eine formale Beschreibung des Verhaltens von Computerprogrammen
- Lambda-Kalkül ist Notation, die *beliebige* berechenbare Funktion darstellen kann.

Das Lambda-Kalkül

- Menge der Ausdrücke (E) ist rekursiv definiert:

$$\bullet \forall v \in V : v \in E$$

Variablen

$$\bullet \forall e_1 \in E, e_2 \in E : (e_1, e_2) \in E$$

Applikation

(Funktion, Argument)

$$\bullet \forall v \in V, e_1 \in E : (\lambda v . e_i) \in E$$

Abstraktion

(Parameter, Rumpfterm)

- Variablen sind entweder **gebunden** oder **frei**

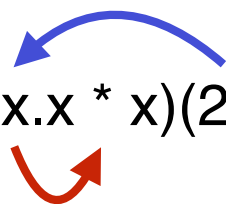
$(\lambda f . (f x))$

Das Lambda-Kalkül

- Beispiele : Identitätsfunktion: $\lambda x.x$

Quadrieren: $(\lambda x.(x * x))$

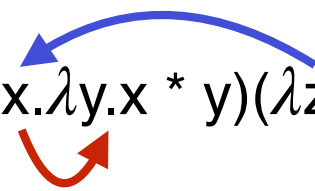
Bsp. : $(\lambda x.x * x)(2)$



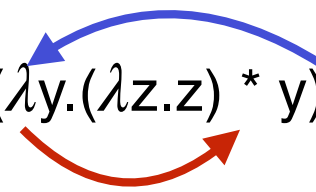
$\Rightarrow (2 * 2) = 4$

Multiplizieren: $(\lambda x.\lambda y.x * y)$

Bsp. : $(\lambda x.\lambda y.x * y)(\lambda z.z)(2)$



$\Rightarrow (\lambda y.(\lambda z.z) * y)(2)$



$\Rightarrow ((\lambda z.z) * 2)$

• Warum funktionale Programmiersprachen?

- Keine Seiteneffekte:

Gleiche Funktion auf gleiche Argumente liefert IMMER das gleiche Resultat (Referentielle Transparenz)

- „Mathematischer“ Aufbau von Programmen macht diese meist kürzer und einfach zu lesen
- Wiederverwendbarkeit durch hohen Abstraktionsgrad
- Einfaches Debuggen
- Einfaches Testen

• Was sind funktionale Programmiersprachen?

- Funktionale Programmierung entspringt dem Lambda-Kalkül (Church, 1932)
- Funktionale Elemente in immer mehr Sprachen vertreten
- Funktionen können an andere Funktionen übergeben werden
- Funktionen können als Resultat zurückgegeben werden

Funktionen, deren Argumente oder Resultate Funktionen sind, bezeichnet man als *Funktionen höherer Ordnung* (auch als *Funktionale*)

Funktionen als Werte

Beachte: Common Lisp hat **verschiedene Namensräume** für Werte und Funktionen; z.B:

```
cons
```

```
==> >>Error: The symbol CONS has no global value
```

```
(function cons)
```

```
==> #<Compiled-Function CONS 142B4E>
```

Mit **function** erhalten wir also den *funktionalen* Wert. **function** wird auch abgekürzt als **#'**, weshalb wir auch schreiben können:

```
#'cons ==> #<Compiled-Function CONS 142B4E>
```

Bemerkung: Die ganze Wahrheit über **function** werden wir erst später erfahren!

Die Funktionale `funcall` und `apply`

In COMMONLISP gibt es die Funktionen `funcall` und `apply`, mit denen man funktionale Werte auf ihre Argumente anwenden kann.

- `funcall` erwartet als erstes Argument eine Funktion `f`, danach eine bestimmte Anzahl von weiteren Werten (abhängig von der Anzahl der Argumente von `f`)

```
(funcall #'cons 1 2) ==> (1 . 2)
```

- `apply` erwartet als erstes Argument eine Funktion `f`, danach eine Liste von Werten (die Länge der Liste hängt ab von der Anzahl der Argumente von `f`)

```
(apply #'cons '(1 2)) ==> (1 . 2)
```

Diese beiden Formen sind synonym zu der Funktionsapplikation

```
(cons 1 2)
```

Die Funktionale `funcall` und `apply`

Weiteres **Beispiel:**

```
(funcall #'first '((a b))) ==> (A B)
```

```
(apply #'first '((a b))) ==> A
```

Die Funktion `mapcar`

Eine prominente Funktion höherer Ordnung ist `mapcar`. Diese Funktion bekommt als Argument eine Funktion und eine Liste, und berechnet eine neue Liste dadurch, daß die Argumentfunktion auf jedes Element der Argumentliste angewandt wird; z.B:

```
(mapcar #'- '(2 3 4))          ==> (-2 -3 -4)
```

```
(mapcar #'first '((a b) (c d))) ==> (A C)
```

```
(defun plus4 (x) (+ x 4))
```

```
(mapcar #'plus4 '(1 2 3))      ==> (5 6 7)
```

Die Funktion `mapcar`

Mit Hilfe von `funcall` läßt sich `mapcar` leicht selbst schreiben:

```
(defun our-mapcar (f l)
  (if (null l)
      nil
      (cons (funcall f (first l))
            (our-mapcar f (rest l)))))
```

Die Funktion `reduce`

Eine weitere prominente Funktion höherer Ordnung ist `reduce`. Diese Funktion bekommt als Argument eine Funktion eine Liste und einen initialen Wert, und berechnet ihr Ergebnis dadurch, dass die Argumentfunktion iterativ auf einen akkumulierten Wert und ein Listenelement anwendet.

```
(reduce #'* '(1 2 3 4) :initial-value 1) ==> 24
```

```
(reduce #'+ '(1 2 3 4)) ==> 10
```

```
(reduce (lambda (x y) (cons y x))  
      '(a b c d) :initial-value nil) ==> (D C B A)
```

Map-Reduce

Die Kombination von `map` und `reduce` ist ein mächtiges Werkzeug, um große Datenmengen verteilt zu verarbeiten (`map`) und die Ergebnisse danach wieder einzusammeln und zu kombinieren (`reduce`). Vereinfacht:

```
(reduce #' +  
  (mapcar #' *  
    '(1 2 3 4 5)  
    '(5 4 3 2 1)))  
==> 35
```

Map-Reduce-Dienste werden von verschiedenen Firmen angeboten (z.B. Google, Amazon, ...), lassen sich aber auch auf eigenen Rechnerclustern umsetzen (z.B. Apache Hadoop).

Funktionen höherer Ordnung

Weitere Funktionen höherer Ordnung:

`(find-if #'evenp '(1 2 3 2 1 0 -1))` ==> 2

`(remove-if #'oddp '(1 2 3 2 1 0 -1))` ==> (2 2 0)

`(every #'oddp '(1 3 4))` ==> NIL

`(some #'oddp '(1 3 4))` ==> T

Anonyme Funktionen

Manchmal will man eine Funktion als Argument einer anderen Funktion übergeben, ohne explizit einen neuen Namen für dieses Funktionsargument zu vergeben; diese sogenannten *anonymen Funktionen* werden durch **lambda**-Ausdrücke realisiert; z.B:

```
(mapcar (lambda (x) (+ x 4)) '(1 2 3))  
==> (5 6 7)
```

Der Aufbau einer anonymen Funktion ist ähnlich dem einer benannten Funktion:

- Benannte Funktion:

```
(defun name (arglist) body)
```

- Anonyme Funktion:

```
(lambda (arglist) body)
```


Anonyme Funktionen

Anonyme Funktionen können an den Stellen verwendet werden, an denen auch benannte Funktionen stehen.

```
(funcall (lambda (x y) (cons x y)) 1 2)
=> (1 . 2)
```

```
((lambda (x y) (cons x y)) 1 2)
=> (1 . 2)
```

Beispiel: Wende **f** auf jedes Element der Liste **l** an und hänge die Resultate aneinander.

```
(defun mappend (f l)
  (if (null l)
      nil
      (append (funcall f (first l))
                (mappend f (rest l)))))
```

```
(mappend #'(lambda (x) (list x (+ x x))) '(1 10 300))
=> (1 2 10 20 300 600)
```

Funktionen als Resultat von Funktionen

Natürlich gibt es in COMMONLISP nicht nur Funktionen als Argumente von Funktionen, sondern auch Funktionen als Resultat von Funktionen.

Beispiel:

```
(defun select-glue (first)
  (if (listp first)
      #'(lambda (x y) (append x y))
      #'(lambda (x y) (cons x y)))))
```

```
(defun glue (x y)
  (funcall (select-glue x) x y))
```

```
(glue 1 '(2))    ==> (1 2)
```

```
(glue '(1) '(2)) ==> (1 2)
```

Funktionen als Resultat von Funktionen

Beispiel: Komposition von Funktionen

```
(defun compose (f1 f2)
  #'(lambda (a) (funcall f1 (funcall f2 a)))))
```

```
(defun our-cadr (a)
  (funcall (compose #'first #'rest) a))
```

```
(our-cadr '(1 2 3 4)) ==> 2
```

Argumentlisten von Funktionen

In Lisp gibt es Funktionen mit einer variablen Anzahl von Parametern oder aber mit optionalen Parametern.

<code>(+ 1 2)</code>	<code>==> 3</code>
<code>(+ 1 2 3 4)</code>	<code>==> 10</code>
<code>(append '(1 2) '(3 4 5) '(6 7))</code>	<code>==> (1 2 3 4 5 6 7)</code>
<code>(member 2.0 '(1 2 3))</code>	<code>==> NIL</code>
<code>(member 2.0 '(1 2 3) :test #'=)</code>	<code>==> (2 3)</code>

Das Schlüsselwort-Argument `:test #'=` bewirkt, daß für den Vergleich von `2.0` und den Elementen von `(1 2 3)` anstatt dem Default-Prädikat `eql` der Vergleich auf Zahlen = verwendet wird.

Argumentlisten von Funktionen

Im folgenden beschäftigen wir uns mit erweiterten Parameterlisten im Makro `defun`, die es uns erlauben, derart flexible Funktionen zu definieren:

- Beliebige viele Argumente
- Optionale Parameter
- Schlüsselwortparameter

Beliebig viele Argumente

Diese Flexibilität wird durch die Spezifikation eines Restparameters in der Argumentlist von `defun` erreicht:

```
(par-1 ... par-n &rest rest)
```

Alle Argumente beim Aufruf, die nicht den übrigen Parametern zugeordnet werden konnten, werden zu einer Liste zusammengefaßt, und diese Liste dem Parameter `rest` zugeordnet.

Beliebig viele Argumente

Beispiel:

```
(defun plus (a &rest l)
  (+ a (sum-up l)))
```

```
(defun sum-up (l)
  (if (null l)
      0
      (+ (first l) (sum-up (rest l)))))
```

```
(plus 1 2 3 4)    ==> 10
```

Optionale Parameter

(par-1 ... par-n &optional opt-1 ... opt-m)

`par-i` sind die *obligatorischen Parameter* der Funktion, während `opt-i` die optionalen Parameter spezifiziert, die beim Funktionsaufruf nicht unbedingt angegeben werden müssen. `opt-i` ist dabei von der Gestalt:

`name`

Default-Wert für Parameter `name`, falls dieser beim Funktionsaufruf nicht angegeben wird, ist `nil`.

`(name wert)`

Default-Wert für `name` ist `wert`.

`(name wert
sp)`

Default-Wert für `name` ist `wert`; zusätzlich wird `sp` an `t` oder `nil` gebunden, je nachdem, ob der Parameter beim Funktionsaufruf explizit angegeben wurde.

Optionale Parameter

Beispiele:

```
(defun iter-length-opt (list &optional (acc 0))  
  (if (null list)  
      acc  
      (iter-length-opt (rest list) (1+ acc))))
```

```
(defun inc (x &optional (y 1 sp))  
  (if sp (princ "Optional Parameter supplied")  
      (princ "Optional Parameter not supplied"))  
  (+ x y))
```

Techniken: Verwendung optionaler Parameter

Funktionen, die einen “Marker” brauchen, können diesen Marker als optionales Argument nehmen.

Beispiel: `occ` bestimmt die Positionen der Vorkommen von `elem` in `lst`.

Techniken: Verwendung optionaler Parameter

1. Realisierung durch Verwendung einer Hilfsfunktion:

```
(defun occ (elem lst)
  (occ-1 elem lst 0))
```

```
(defun occ-1 (elem lst pos)
  (cond ((null lst) nil)
        ((eql elem (first lst))
         (cons pos (occ-1 elem (rest lst) (1+ pos))))
        (T (occ-1 elem (rest lst) (1+ pos)))))
```

```
> (occ 'a '(a b a d))
(0 2)
```

Techniken: Verwendung optionaler Parameter

2. Der Positionsmarker wird durch ein optionales Argument realisiert:

```
(defun occ-opt (elem lst &optional (pos 0))
  (cond ((null lst)
        nil)
        ((eql elem (first lst))
         (cons pos (occ-opt elem (rest lst) (1+ pos))))
        (T (occ-opt elem (rest lst) (1+ pos)))))
```

Aufruf wie gewohnt; Problem: (Versehentliche Aufrufe)

Verwendung des optionalen Parameters:

```
> (occ-opt 'a '(a b a d))
(0 2)
> (occ-opt 'a '(a b a d) 3)
(3 5)
```

Schlüsselwort-Argumente:

```
(par-1 ... par-n &key opt-1 ... opt-m)
```

`opt-i` hat die gleiche Gestalt wie bei den `&optional`-Argumenten: `(name wert)`. Schlüsselwortargumente sind ebenfalls optional.

Beispiel:

```
(defun our-member-key (x l &key (test #'eql))  
  (if (or (null l) (funcall test x (first l)))  
      l  
      (our-member-key x (rest l) :test test)))
```

Schlüsselwort-Argumente:

Beispiel:

```
(defun bsp (a b &key c (d a))  
  (list a b c d))
```

```
(bsp 1 2) ==> (1 2 NIL 1)
```

```
(bsp 1 2 :d 3 :c 4) ==> (1 2 4 3)
```

```
(bsp :a :b :c :d) ==> (:A :B :D :A)
```

```
(bsp :a :b :d :c) ==> (:A :B NIL :C)
```

Schlüsselwort-Argumente:

Das Zusammenwirken der unterschiedlichen Parameterspezifikationen verdeutlicht das folgende Beispiel:

```
(defun bsp2 (a &optional b &rest c &key d)
  (list a b c d))
```

```
(bsp2 1 2 :d 3) ==> (1 2 (:d 3) 3)
```

Wenn `&rest`- und `&key`-Parameter spezifiziert sind, werden, nach Bindung der obligatorischen und der optionalen Parameter, die übrigen in einer Liste an den `&rest`-Parameter gebunden. Diese Liste muß eine gerade Anzahl von Elementen enthalten, die dann als Paare von Schlüsselwort und Wert zusammengefaßt werden. Diese werden dann entsprechend den `&key`-Parametern zugeordnet.

Funktionen mit Schlüsselwort-Argumenten

Viele vordefinierten Lisp-Funktionen machen ausgiebig Gebrauch von Schlüsselwort-Argumenten, so z.B. die Funktion:

```
(remove-if predicate sequence &key :start :end :count)
```

Es wird eine Liste zurückgegeben, die alle Elemente von *sequence* enthält bis auf diejenigen, die *predicate* erfüllen:

```
(remove-if #'(lambda (x) (eq (first x) 'a))  
           '((b 3) (a 5) (c 7) (a 9) (a 4))  
           :start 2 :count 1)  
==> ((B 3) (A 5) (C 7) (A 4))
```


Funktionen mit Schlüsselwort-Argumenten

Mit dem `:start` und `:end` Schlüsselwort können Positionen angegeben werden:

```
(remove-if #'(lambda (x) (eq (first x) 'a))  
           '((b 3) (a 5) (c 7) (a 9) (a 4))  
           :start 2)  
==> ((B 3) (A 5) (C 7))
```

`:count` gibt die maximale Anzahl der entfernten Elemente an:

```
(remove-if #'(lambda (x) (eq (first x) 'a))  
           '((b 3) (a 5) (c 7) (a 9) (a 4))  
           :start 2 :count 1)  
==> ((B 3) (A 5) (C 7) (A 4))
```

Funktionen mit Schlüsselwort-Argumenten

```
(substitute newitem olditem sequence &key :start :end :test)
```

(Bem.: `substitute` erlaubt noch mehr Key-Argumente, siehe Steele)

```
(substitute 0 7 '(1 6 7 2 3 7 8))  
==> (1 6 0 2 3 0 8)
```

Mit `:test` kann ein zweistelliges Prädikat angegeben werden.
Ist dieses erfüllt, so wird die Substitution durchgeführt:

```
(substitute 0 7 '(1 6 7 2 3 7 8) :test #'>)  
==> (0 0 7 0 0 7 8)
```

... ersetzt alle Elemente kleiner als 7 durch 0

In den **Übungsaufgaben** werden einige andere Funktionen mit Key-Argumenten behandelt

Zusammenfassung

- In Lisp können Funktionen an Funktionen als Werte übergeben werden.
- In Lisp können Funktionen von anderen Funktionen zurückgegeben werden (im Unterschied zu den meisten imperativen Sprachen)
- In Lisp gibt es anonyme Funktionen (im Unterschied zu vielen imperativen Sprachen, jedoch auch hier immer häufiger)
- In Lisp sind viele Funktionen (höherer Ordnung) vordefiniert, die auf Listen arbeiten.
- Funktionen können:
 - beliebig viele Argumente annehmen
 - optionale Parameter haben
 - durch Schlüsselworte modifiziert werden