Praktische Algorithmen der Bioinformatik und Computerlinguistik mit Lisp

Teil 3: Kontrollstrukturen

Tilman Becker

13.5.2021

LISP-Intensivkurs

Übersicht:

- Kontrollstrukturen
- Funktionsdefinitionen
- Speichersparendes Programmieren
- Primitiver Entwicklungszyklus in Lisp
- Handhabung des Debuggers
- Compilierung
- Beispiele

Fallunterscheidung: `if`

In Common Lisp gibt es verschiedene Spezialformen zur Fallunterscheidung; die einfachste davon ist `if`:

```
(if condition-form
  then-form
  else-form)
```

Achtung: Hier wird von der üblichen Lisp-Auswertungsreihenfolge abgewichen!

Falls die Auswertung von `condition-form` einen von `nil` verschiedenen Wert ergibt, so wird `then-form` ausgewertet und als Resultat zurückgegeben, im anderen Falle wird `else-form` ausgewertet.

Fallunterscheidung: `if`

Beispiel:

```
(setf l '(1 2 3))

(if (null l) nil (append l l))

(if (member 3 l)
  (if (integerp (first l))
    (cons (+ (first l) (first l)) l)
    l)
  'nil)
```

Verallgemeinerte Fallunterscheidung: `cond`

```
(cond
  (test-form-1 then-form-1)
  (test-form-2 then-form-2)
  ...)
```

Bei `cond` wird `test-form-i` Formen von links nach rechts so lange ausgewertet, bis ein `test-form-i` einen von `nil` verschiedenen Wert ergibt. Dann wird die korrespondierende `then-form-i` ausgewertet und als Resultat zurückgegeben. Falls alle `test-form-i` zu `nil` auswerten, so ist das Ergebnis von `cond` gleich `nil`.

Verallgemeinerte Fallunterscheidung: `cond`

Beispiel:

```
(setf income 35000)

(cond
  ((< income 10000) 0.0)
  ((< income 30000) 0.2)
  ((< income 50000) 0.25)
  ((< income 70000) 0.30)
  (t                0.35))
==>   0.25
```

Funktionsdefinition

Die Spezialform `defun` steht für **de**fine **fun**ction.

```
(defun function-name (arg-name arg-name ... )
  "optional documentation"
  function-body)
```

Einfache **Beispiele:**

```
(defun square (x)
  (* x x))

(square 12)  ==>  144

(defun hyper-square (x)
  (* (square x) (square x)))

(hyper-square 3) ==> 81
```

Funktionsdefinition

Rekursive Funktionsdefinition:

```
(defun our-length (l)
  (if (null l)
    0
    (1+ (our-length (rest l)))))
```

Funktionsdefinition: Weitere Beispiele

Eine Funktion `flatten` berechnet aus einer beliebig geschachtelten Liste eine Liste ihrer Atome, z.B:

```
(flatten '((a (b c)) d (e) (f g) ((h) i)))
==> (a b c d e f g h i)
```

Implementierung in Lisp:

```
(defun flatten (l)
  (cond
    ((null l) 'nil)
    ((atom (first l))
      (cons (first l) (flatten (rest l))))
    (t (append (flatten (first l))
               (flatten (rest l))))))
```

Funktionsdefinition: Weitere Beispiele

Allgemeine Bemerkung: In Common Lisp ist bereits eine Unzahl von Funktionen vordefiniert (und recht effizient implementiert), so z.B. `append`. Siehe z.B. Buch von Steele.

Techniken: Speichersparendes Programmieren

Jeder Aufruf von `cons` (direkt oder indirekt durch `append` und ähnliche Fkt.) reklamiert neuen Speicherplatz (eine Cons-Zelle).

Ungünstiger Programmierstil kann zu großem Speicherbedarf und längerer Laufzeit führen (mehr Speicher muß durch den *Garbage Collector* zurückgefordert werden).

Die Funktion `our-append` hängt Liste `l1` an Liste `l2`:

```
(defun our-append (l1 l2)
  (if (null l1)
    l2
    (cons (first l1)
      (our-append (rest l1) l2))))
```

Beobachtung: Die Anzahl der Aufrufe von `cons` in `our-append` ist gleich der Länge des ersten Arguments.

Techniken: Speichersparendes Programmieren

Die Funktion `our-reverse` invertiert die Liste `lst`:

```
(defun our-reverse (lst)
  (if (null lst)
    nil
    (our-append (our-reverse (rest lst))
                (list (first lst)))))
```

Beobachtung:

`our-reverse` hat einen quadratischen Speicheraufwand.

Techniken: Speichersparendes Programmieren

Eine Speicher-effiziente Variante dieser Funktion (die auch Laufzeit-effizient ist, siehe die Folien zum Thema Compilierung):

```
(defun our-eff-reverse (lst)
  (our-eff-reverse-1 lst nil))

(defun our-eff-reverse-1 (lst acc)
  (if (null lst)
    acc
    (our-eff-reverse-1 (rest lst)
                       (cons (first lst) acc))))
```

Beobachtung:

`our-eff-reverse` hat einen linearen Speicheraufwand.

Primitiver Entwicklungszyklus in Lisp

1. Entwickeln eines Programms in einer Datei (z.B. mit Emacs).
   Speichern des Resultats z.B. in program.lisp.
2. Laden, evtl. Compilieren des Programms im Lisp-Listener.

Laden und Compilieren von Dateien

- Laden: `(load "Dateiname")`
  - Beispiel: `(load "program.lisp")`
- Compilieren: `(compile-file "Dateiname")`
  - in EMACS/SLIME: `Ctrl-c Ctrl-k` (Compilieren und Laden)
- Einzelne Funktion neu laden:
  - in EMACS/SLIME: `Ctrl-Alt-x`

Primitiver Entwicklungszyklus in Lisp

Was tun, wenn ein Programm nicht nach Plan arbeitet?

- Schrittweise Ausführung eines Programmes mit dem Stepper
- Betrachten des Aufrufverhaltens (Funktionsparameter, Rückgabewerte) mit dem Tracer
- Setzen von Unterbrechungspunkten mit break
- Betrachten des Aufrufstacks mit dem Debugger
- Betrachten von Werten mit dem Inspector

Der Stepper

Mit dem Stepper läßt sich ein Programm schrittweise ausführen:

- Mit `:step t` wird Stepping für alle Ausdrücke eingeschaltet
- Mit `:step` $fun_1 \ldots fun_n$ wird Stepping für Funktionen $fun_1 \ldots fun_n$ ermöglicht
- Mit `:step` wird Stepping ausgeschaltet
- Mit *Return* wird der aktuelle Ausdruck im Stepping-Modus
- ausgeführt
- Mit `:so` (`:sover`) wird der aktuelle Ausdruck nicht im Stepping-Modus ausgeführt

Primitiver Entwicklungszyklus in Lisp

```
USER(37): :step t
USER(38): (+ (* 2 3) (* 3 4))
1: (+ (* 2 3) (* 3 4))
[step] USER(39): RETURN
2: (* 2 3)
[step] USER(39): :so
result 2: 6
2: (* 3 4)
[step] USER(40): RETURN
3: 3 => 3
3: 4 => 4
result 2: 12
result 1: 18
18
USER(41):
```

Der Tracer

Mit dem Tracer lassen sich Funktionsaufrufe verfolgen:

- Mit `(trace` fun_1`..`fun_n`)` wird der Trace für Funktionen fun_1…fun_n eingeschaltet
- Mit `(untrace` fun_1`...`fun_n`)` wird er für fun_1`...`fun_n ausgeschaltet
- Auch möglich `:Trace` in Abhängigkeit von Bedingungen (siehe Handbuch)

Der Tracer

```
USER(66): (defun our-length (l)
  (if (null l) 0
    (1+ (our-length (rest l)))))
OUR-LENGTH
USER(67): (trace our-length)
(OUR-LENGTH)
USER(68): (our-length '(a b c))
0: (OUR-LENGTH (A B C))
 1: (OUR-LENGTH (B C))
  2: (OUR-LENGTH (C))
   3: (OUR-LENGTH NIL)
   3: returned 0
  2: returned 1
 1: returned 2
0: returned 3
3
USER(69):
```

Der Debugger

Durch Aufruf der Funktion `break` wird ein Unterbrechungspunkt gesetzt, man gelangt in den Debugger (`break` kann auch ohne Argument aufgerufen werden):

```
USER(34): (defun our-length (l)
  (break "Call with argument ~a" l)
  (if (null l)
    0
    (1+ (our-length (rest l)))))
OUR-LENGTH
USER(35): (our-length '(a b))
Break: Call with argument (A B)

Restart actions (select using :continue):
0: return from break.
[1c] USER(36):
```

Der Debugger

Gewöhnlich gelangt man durch einen Fehler in den Debugger:

```
USER(3): (+ 1 a)
Error: Attempt to take the value of the
unbound variable ‘A’.
[condition type: UNBOUND-VARIABLE]

Restart actions (select using :continue):
0: Try evaluating A again.
1: Use :A instead.
2: Set the symbol-value of A and use its value.
3: Use a value without setting A.
[1] USER(4):
```

Der Debugger

Mit dem Fehler kann man auf folgende Weise umgehen:

- Ausführung vollständig beenden mit `:reset`
- Eine der angebotenen Fehlerbehandlungen auswählen (mit `:continue` Optionsnummer):

```
[1] USER(5): :continue 3
enter an expression which will evaluate
to a new value: 4
5
```

Der Debugger

Es können auch im Debugger Lisp-Ausdrücke evaluiert werden. Bei Auftreten eines weiteren Fehlers gerät man in einen höheren Debugger-Level:

Der Debugger

```
[1] USER(11): (+ 1 2)
3
[1] USER(12): (+ 1 b)
Error: Attempt to take the value of the
unbound variable ‘B’.
[condition type: UNBOUND-VARIABLE]

Restart actions (select using :continue):
....
[2] USER(13): :pop
Previous error: Attempt to take
the value of the unbound variable ‘A’.
[1] USER(14):
```

Zurück zum ersten Level mit `:pop`

Weitere Optionen werden durch :help angezeigt

Compilierung

Funktionen und Methoden können *in Lisp* compiliert werden:

- Compilierung einer Funktion foo durch: `(compile 'foo)`
- Compilierung einer Datei “`program`" durch:

  `(compile-file "program")` Dabei Erzeugung einer Binär-Datei program.fasl.

Compilierung

Durch Compilierung erhält man:

- Warnungen bzw. Fehlermeldungen bei:
  - Ungebundenen Variablen:

```
USER(60): (defun foo1 (x) (+ x y))
FOO1
USER(61): (compile 'foo1)
; While compiling FOO1:
Warning: Free reference to undeclared variable Y ...
FOO1
```

  - Falsche Anzahl von Argumenten

Compilierung

- Warnungen bzw. Fehlermeldungen bei:
  - Ungebundene Variablen (~> declare ignore Statement)

```
USER(62): (defun foo3 (x y) (list x))
FOO3
USER(63): (compile 'foo3)
; While compiling FOO3:
Warning: Variable Y is never used.
FOO3
USER(64): (defun foo3 (x y)
  (declare (ignore y))
  (list x))
FOO3
```

- Effizienteren Code, z.B. für endrekursive Funktionen

Compilierung endrekursiver Funktionen

Eine Funktion wird als *endrekursiv* (tail recursive) bezeichnet, wenn es nach ihrem rekursiven Aufruf “nichts mehr zu tun gibt”. Nicht tail-rekursive Variante der Längen-Funktion für Listen:

```
(defun our-length (l)
  (if (null l)
    0
    (1+ (our-length (rest l)))))
```

Tail-rekursive Variante (mit einer Hilfsfunktion):

```
(defun tail-length (l)
  (tail-length-1 l 0))

(defun tail-length-1 (l acc)
  (if (null l)
    acc
     (tail-length-1 (rest l) (1+ acc))))
```

Compilierung tail-rekursiver Funktionen

Trace der Ausführung:

```
USER(75): (trace tail-length tail-length-1)
(TAIL-LENGTH-1 TAIL-LENGTH)
USER(76): (tail-length '(a b))
0: (TAIL-LENGTH (A B))
 1: (TAIL-LENGTH-1 (A B) 0)
  2: (TAIL-LENGTH-1 (B) 1)
   3: (TAIL-LENGTH-1 NIL 2)
   3: returned 2
  2: returned 2
 1: returned 2
0: returned 2
2
```

Compilierung tail-rekursiver Funktionen

Compilierung der Funktion (mit optimierender Compiler-Option)

```
USER(78): (proclaim '(optimize (speed 3)))
T
USER(79): (excl:explain-compiler-settings)
;; Compiler optimization quality settings:
;;   safety   1
;;   space    1
;;   speed    3
;;   debug    2
;; ....
USER(80): (compile 'tail-length-1)
TAIL-LENGTH-1
```

Compilierung tail-rekursiver Funktionen

Ausführung jetzt:

```
USER(81): (tail-length '(a b))
0: (TAIL-LENGTH (A B))
1: (TAIL-LENGTH-1 (A B) 0)
1: returned 2
0: returned 2
2
```

Compilierung tail-rekursiver Funktionen

Tail-rekursive Funktionen lassen sich leicht in Schleifen umwandeln:

```
function tail-lenght-1 (l acc)
  acc' := acc ;
  l' := l ;
  while (not (null l')) do
    l'   := (rest l') ;
    acc' := (1+ acc') ;
  end;
  return(acc')
```

Mit Standard-Verfahren lassen sich sogar linear rekursive Funktionen (wie z.B. `our-length`) in Iterationen überführen.

Compilierung tail-rekursiver Funktionen

Moral der Geschichte:

1. Rekursive Funktionen können oft so effizient wie iterative ausgeführt werden.
2. Aufzeichnungen der Programmausführung (z.B. in Stepper, Tracer, Debugger) sehen manchmal anders aus als erwartet.

Verwendung von Konstruktor- und Selektor-Funktionen

Es macht ein Programm-Design wesentlich übersichtlicher und änderungsfreundlicher, wenn für bestimmte “abstrakte Datentypen” Konstruktor- und Selektorfunktionen eingeführt werden.

Beispiel: Binärbaum:

- Konstruktoren: `empty-tree`, `cons-tree`
- Selektoren: `root`, `left-subtree`, `right-subtree`
- ‘Erkennungs-Prädikate’: `empty-tree?`, `cons-tree?`

Verwendung von Konstruktor- und Selektor-Funktionen

Implementierung z.B.:

```
(defun empty-tree ()
NIL)
(defun cons-tree (root left right)
  (list root left right))
(defun root (tree)
  (first tree))
(defun left-subtree (tree)
  (second tree))
(defun right-subtree (tree)
  (third tree))
(defun empty-tree? (tree)
  (null tree))
(defun cons-tree? (tree)
  (consp tree))
```

Verwendung von Konstruktor- und Selektor-Funktionen

Anwendung:

```
> (cons-tree 'a
    (cons-tree 'b
      (empty-tree)
      (empty-tree))
    (empty-tree))
(A (B NIL NIL) NIL)
```

Aufgabe: Schreiben Sie eine Funktion swap-tree, die (rekursiv) den linken und rechten Teilbaum eines Binärbaums vertauscht.

Bemerkung: Durch eine inline Compiler-Deklaration oder ein Makro (siehe später) wird der Code einer Funktion direkt in andere Funktionen “hineinkopiert”. Daher keinerlei Effizienzverlust!

Verwendung von Konstruktor- und Selektor-Funktionen

Aufgabe: Die obigen Konstruktorfunktionen sind nicht "speicheroptimal". Besser wäre z.B.:

```
(defun cons-tree (root left right)
  (cons root (cons left right)))
```

Schreiben Sie die anderen Funktionen entsprechend um.

(Funktioniert jetzt die Funktion `swap-tree` noch?!)

Weitere Beispiele zu rekursiven Funktionen

Es folgen weitere Beispiele zum (funktionalen)

Programmieren in Lisp.

- Listenfunktionen und -prädikate
- Implementierung von Assoziationslisten
- gegenseitige Rekursion

Weitere Beispiele zu rekursiven Funktionen

Beispiel: Gleichheit; `our-equal` implementiert eine Annäherung an das Systemprädikat `equal` mit Hilfe der Identität `eql`:

```
(defun our-equal (x y)
  (if (or (atom x) (atom y))
      (eql x y)
      (and (our-equal (car x) (car y))
           (our-equal (cdr x) (cdr y)))))

(our-equal '(a b c) '(a b c))               ==> T

(our-equal '(a . (b . c)) '(a . (b . c))) ==> T
```

Weitere Beispiele zu rekursiven Funktionen

Bemerkung: `our-equal` ist jedoch nur eine Annäherung an `equal`; **z.B:**

```
(our-equal "abc" "abc")    ==> NIL
(equal "abc" "abc")        ==> T
```

Weitere Beispiele zu rekursiven Funktionen

Beispiel: Eine Funktion `deepmember` soll aus einer beliebig geschachtelten Liste testen, ob irgendwo ein bestimmtes Element enthalten ist; z.B:

```
(deepmember 'c '((a (b c)) d (e) (f g) ((h) i)))
==> T
```

Implementierung in Lisp:

```
(defun deepmember (a l)
  (cond
    ((null l) nil)
    ((atom (first l))
      (or (eql a (first l))
          (deepmember a (rest l))))
    (t (or (deepmember a (first l))
           (deepmember a (rest l))))))
```

Weitere Beispiele zu rekursiven Funktionen

Beispiel: Implementierung von Assoziationslisten; dies sind Listen der Form:

```
((key1 . val1) (key2 . val2) (key3 . val3))
```

Auf dieser Struktur gibt es die Systemfunktionen `acons`, `pairlis` und `assoc`, die im folgenden nachgebaut werden. `our-acons` ist unsere Implementierung der Systemfunktion `acons`.

```
(defun our-acons (key val alist)
  (cons (cons key val) alist))
```

Weitere Beispiele zu rekursiven Funktionen

`our-pairlis` baut aus einer Variablenliste und einer Werteliste eine Assoziationsliste und hängt sie an das dritte Argument an:

```
(defun our-pairlis (keys vals alist)
  (if (null keys)
    alist
    (our-acons (first keys) (first vals)
      (our-pairlis (rest keys) (rest vals)
        alist))))

(setf *a-list*
(our-pairlis '(x y z) '(1 2 3) nil))
==> ((X . 1) (Y . 2) (Z . 3))
```

Weitere Beispiele zu rekursiven Funktionen

`our-assoc` ist unsere Implementierung der Systemfunktion `assoc`.

```
(defun our-assoc (e a)
  (cond ((null a) nil)
    ((eq e (first (first a))) (first a))
    (t (our-assoc e (rest a))) ))

(our-assoc 'y *a-list*)   ==> (Y . 2)
(our-assoc 'w *a-list*)   ==> NIL
```

Weitere Beispiele zu rekursiven Funktionen

Beispiel: Gegenseitige Rekursion, d.h zwei oder mehr Funktionen rufen sich gegenseitig rekursiv auf; z.B: Funktionen `odd-elems` und `even-elems` berechnen aus einer Argumentliste die Liste der Elemente an ungeraden respektive an geraden Positionen; z.B:

```
(odd-elems '(a b c d e))      ==> (A C E)
(even-elems '(a b c d e))     ==> (B D)
```

Weitere Beispiele zu rekursiven Funktionen

Implementierung in Lisp:

```
(defun odd-elems (l)
  (if (null l)
    'nil
    (cons (first l) (even-elems (rest l)))))

(defun even-elems (l)
  (if (null l)
    'nil
    (odd-elems (rest l))))
```

Zusammenfassung

- Mit `if` und `cond` können Fallunterscheidungen durchgeführt werden
- Funktionen werden durch `defun` definiert
- Es gibt verschiedene Techniken,um Speicher- und Laufzeit-effiziente Programme zu schreiben
- Rekursive Konstrukte können u.U. vom Kompilieren effiziente iterative Konstrukte umgeformt werden.
- Mit `step`, `trace` und `break` kann die Programmausführung beobachtet werden
- Compilierung erfolgt innerhalb von Lisp