Praktische Algorithmen der Bioinformatik und Computerlinguistik mit Lisp

Imperative Programmierung

12.05.2021

LISP-Kurs

Übersicht:

- Imperative Programmierung
- Ein- und Ausgabe
- “Destruktive” Operationen

Imperatives Programmieren

In einem imperativen Programm weisen Operationen den *Programmvariablen* Werte zu, die dann als Grundlage für weitere Berechnungen dienen.

In einem imperativen Programm gibt es einen *Programmzustand*, der durch Zuweisungen, Prozeduraufrufe etc. verändert wird.

Während in rein funktionalen Programmen (verschachtelter Aufruf von Funktionen) die Auswertung einer Form klar ist, muß in imperativen Programmen die Reihenfolge der Abarbeitung von Ausdrücken explizit (durch syntaktische Konstrukte) spezifiziert werden, da imperative Programme auf Nebeneffekten basieren.

Imperatives Programmieren

Konstrukte der imperativen Programmierung:

- Wertzuweisung
- Seiteneffekte; z.B. durch Ausgabefunktionen
- Programmkontrollstrukturen
  - Sequentielle Komposition
  - Schleifen

Wertzuweisung

Variablen (d.h. Symbolen) können Werte zugewiesen werden mit der Spezialform `setf`.

```
> p
>>Error: The symbol P has no global value
> (setf p '(Joe Sixpack))     ==> (JOE SIXPACK)
> p                           ==> (JOE SIXPACK)

> (setf x 10)                 ==> 10
> (+ x x)                     ==> 20
> (+ x (length p))            ==> 12
```

Wertzuweisung

Allgemeine Syntax von `setf`:

```
(setf acc-1 expr-1 ... acc-n expr-n)
```

Dies weist `acc-i` das Ergebnis nach Auswertung von `expr-i` zu. Die Zuweisungen werden sequentiell von links nach rechts abgearbeitet.

Bemerkung: Eine ältere (und eingeschränkte) Form der Wertzuweisung ist die mit setq.

Wertzuweisungen

Weitere **Beispiele**:

```
(setf w '(1 2 3) x 1 y "abc" z (first w))
      ==> 1
 w    ==> (1 2 3)
 x    ==> 1
 y    ==> "abc"
 z    ==> 1
```

`setf` gibt den zuletzt zugewiesenen Wert zurück und kann daher auch in Ausdrücken verwendet werden (... dubioser Programmierstil).

```
(* (setf y 6) (setf z (- y 3)))
     ==> 18
y    ==> 6
z    ==> 3
```

Wertzuweisungen

Beispiel: Aufbau von Assoziationslisten

```
(setf a-list (acons 'key1 'val1 nil))
   ==> ((KEY1 . VAL1))
(setf a-list (acons 'key2 'val2 a-list))
   ==> ((KEY2 . VAL2) (KEY1 . VAL1))
```

Wertzuweisungen

Häufig wünscht man sich, die Komponente eines strukturierten Datenobjektes zu modifizieren:

```
(setf l '(a b c d))    ==> (A B C D)
(setf (second l) 'e)   ==> E
l                      ==> (A E C D)
```

```
(setf (cdr (assoc 'key1 a-list)) 'newval1)
a-list          ==> ((KEY2 . VAL2) (KEY1 . NEWVAL1))
```

Anstatt eines einfachen Symbols kann man also auch die zu modifizierende Stelle mithilfe von Zugriffsfunktionen angeben; diese Stelle wird dann überschrieben. `setf` wirkt also destruktiv.

Wertzuweisungen

Mit `setf` lassen sich sogar Graphen und zyklische Strukturen realisieren; z.B:

```
(setf p '(1 . 2))
(setf (cdr p) p)  ==> 1 1 1 1 1 1 1 1 1 1 1.....
(setf *print-circle* t)        p     ==> #1=(1 . #1#)
```

Bemerkung: Es gibt definitiv bessere Methoden, um Graphen etc. zu realisieren (z.B. durch `struct` – siehe später).

Sequentielle Komposition

Die sequentielle Komposition dient dazu, Formen nacheinander auszuwerten. In C oder JAVA etwa erfolgt dies durch Aneinanderreihung von Operationen, die jeweils durch ‘;’ voneinander abgetrennt sind. In Lisp gibt es dafür mehrere Formen; die am häufigsten verwendeten sind `prog1` und `progn`.

Die `prog`-Formen werden aufgerufen mit dem syntaktischen Muster

```
(prog1 form-1 ... form-n)
(progn form-1 ... form-n)
(block block-label form-1 ... form-n)
```

Sequentielle Komposition

`form-1` bis `form-n` werden in der gegebenen Reihenfolge nacheinander ausgewertet. Der Wert der `prog1` Form ist das Resultat der Auswertung von `form-1`, während der Wert der progn Form der Wert von `form-n` ist.

Bei block wird wie bei `progn` von links nach rechts ausgewertet. Hier aber kann man mit return-from aus einem Block herausspringen.

Sequentielle Komposition

Beispiel:

```
>  (progn (setf x 1)
          (setf y (* 2 x))
          (setf z (+ x y)) )     ==> 3
    x  ==> 1   y  ==> 2   z  ==> 3
```

```
>  (prog1 (setf x 1)
          (setf y (* 2 x))
          (setf z (+ x y)) )     ==> 1
    x  ==> 1   y  ==> 2   z  ==> 3

>  (block yow (if (atom '(1))
                  (print 1)
                  (return-from yow 0))
                2)
      ==> 0
```

Sequentielle Komposition

Bemerkung: Viele Spezialformen und Makros benutzen

implizite `progn` Anweisungen: `defun`, `let`, `cond`, `when`, ...

```
> (let ((val 5))
    (format T "This form returns 1 + ~D" val)
    (1+ val))
This form returns 1 + 5
6
```

Schleifen

Zwei Arten von Schleifenkonstrukten werden häufig benötigt

- Schleifen über eine Datenstruktur–im wesentlichen FOR-Loop-artige Sprachkonstrukte.
- allgemeine Schleifen–im Prinzip WHILE-Loops.

`dolist` und `dotimes` sind Lisp Schleifen von der ersten Bauart. Sie haben das Aufrufmuster

```
(dolist (var list res) body)
(dotimes (var upper res) body)
```

`dolist` wird ausgewertet, indem var der Reihe nach an die Elemente der Liste gebunden wird, und mit dieser Bindung der Rumpf ausgewertet wird. Zuletzt wird, falls vorhanden, die optionale Form `res` ausgewertet; ihr Ergebnis ist das Ergebnis des `dolist`-Aufrufs.

Schleifen

```
(dolist (var list res) body)
(dotimes (var upper res) body)
```

`dotimes` iteriert den Rumpf `upper`-mal.

`var` wird dabei der Reihe nach an die Werte 0,1,. . . ,`upper`-1 gebunden (Die Auswertung von upper muß eine ganze Zahl sein).

Ansonsten ähnliche Abarbeitung wie bei `dolist`.

Schleifen

Beispiel:

```
(let ((x nil))
   (dotimes (y 5 x) (setf x (cons y x))))
==> (4 3 2 1 0)

(defun length-dolist (l)
   (let ((len 0))
      (dolist (el l len)
         (incf len))))

(defun product (l)
   (let ((res 1))
      (dolist (n l res)
         (if (= n 0)
            (return 0)
            (setf res (* n res))))))
```

Schleifen

Bemerkung: Neben diesen speziellen Schleifen gibt es noch allgemeine; z.B. das `loop` Makro, das wiederum eine eigene Sprache für sich darstellt; z.B:

```
(loop for x in '(1 2 3) do
  (if (= x 4)
    (return)
    (print x)))
```

Einfache Ausgabe

Es gibt in CommonLisp gut ein Dutzend Ausgabe-Funktionen. Die wichtigsten davon:

Die `print` Funktion hängt Zeichen an einen Ausgabestrom (hier: Terminal) an.

```
(print "hello,")
(terpri)
(princ "world")
```

Die Funktion `terpri` beginnt eine neue Zeile im Ausgabestrom. `princ` ist auf eine lesbare Ausgabe hin optimiert, während print auf die maschinelle Weiterverarbeitung hin ausgelegt ist.

Einfache Ausgabe

Kontrolle über `print` durch globale Variablen:

```
(setf my-tree '(A (B (D) (E (F) (G))) (C)))
   ==> my-tree
my-tree ==> (A (B (D) (E (F) (G))) (C))
(setf *print-pretty* T)
my-tree ==> (A (B (D) (E (F) (G))) (C))
(setf *print-level* 2)
my-tree   ==> (A (B # #) (C))
(setf *print-length* 2)
my-tree   ==> (A (B # ...) ...)
```

Ausgabe durch `format`

Generelle Syntax:

```
(format stream "format-string" arg-1 ... arg-n)
```

Falls das `stream` Argument `nil` ist, gibt format einen formatierten String zurück. Im Falle, daß es `t` ist, gibt die Funktion nil zurück und als Strom wird das Terminal genommen. Weitere Streams können z.B. auf Dateien gelenkt werden.

Der Formatstring enthält Formatierungsanweisungen; (davon gibt es eine Unmenge, siehe z.B. CLtL2).

Ausgabe durch `format`

Beispiele: Verschiedene Stream-Parameter:

```
> (format nil "hello, world")
"hello, world"
```

Einfache *Rückgabe* des Format-Strings

```
> (format t "hello, world")
hello, world
NIL
```

Einfache *Ausgabe* des Format-Strings

```
> (format *standard-output* "hello, world")
hello, world
NIL
```

Ausgabe durch `format`

Beispiele: Verschiedene Formatierungsanweisungen: Ausgeben von Lisp-Ausdrücken mit `~A` (Ascii) und `~S` (S-Expressions)

```
> (format nil
    "No difference between ~A and ~S" '(a b) '(a b))
"No difference between (A B) and (A B)"
> (format nil
    "... but between ~A and ~S" "a b" "a b")
"... but between a b and \"a b\""
```

Ausgeben von Zahlen mit `~D` (Decimal), `~B` (Binary)...

```
> (format nil "Number ~D as binary: ~B" 3 3)
"Number 3 as binary: 11"
```

Ausgabe durch `format`

... Und vieles andere mehr:

```
> (let ((numbers '(1 2 3 4 5)))
     (format t "~&~{~r~^ plus ~} is ~@r"
                numbers (apply #'+ numbers)))
one plus two plus three plus four plus five is XV
NIL
```

Einfache Eingabe

Eingabe erfolgt typischerweise mit der Funktion

```
(read &optional input-stream)
```

`read` gibt den Wert des eingelesenen Ausdrucks zurück:

```
> (let ((val (read))) (first val))
(a b c)      ;;; dies hier ist die Eingabe !
A
```

Einfache Eingabe

Folgende Funktion summiert alle eingegebenen Zahlen:

```
> (defun sum-of-input ()
  (let ((sum 0)
        (input 0))
(format T "Enter numbers, then quit with NIL~%")
(loop while input do
      (format T "Next Number: ")
      (setf input (read))
      (when (numberp input)
            (setf sum (+ sum input))))
(format T "Sum is: ~D" sum)))
```

Einfache Eingabe

Ausgabe:

```
> (sum-of-input)
Enter numbers, then quit with NIL
Next Number: 3
Next Number: 4
Next Number: nil
Sum is: 7
NIL
```

Destruktive Listen-Operationen

Man unterscheidet zwischen *destruktiven* und *konservativen* Funktionen auf Listen.

- Konservative Funktionen legen eine Kopie der Liste an und verändern dann diese Kopie
- Destruktive Funktionen verändern die Struktur der Liste selbst.

Bemerkung: Destruktive Funktionen sind oft effizienter, da kein neuer Speicher allokiert werden muß. Bei konservativen Funktionen entsteht auch viel Speichermüll, d.h. belegter Speicher, auf den aber nicht mehr zugegriffen werden kann.

Destruktive Listen-Operationen

Beispiel: Die Funktionen `append` (konservativ) und `nconc` (destruktiv).

```
(setf l1 '(1 2))              ==> (1 2)
(setf l2 '(3 4))              ==> (3 4)
(setf l3 (append l1 l2)) ==> (1 2 3 4)
(setf l4 (nconc l1 l2))  ==> (1 2 3 4)
(eq l1 l3)                    ==> NIL
(eq l1 l4)                    ==> T
```

Destruktive Listen-Operationen haben oft unerwünschte Seiteneffekte. Daher sollten sie möglichst vermieden werden:

```
l1   ==> (1 2 3 4)
```

Destruktive Listen-Operationen

Die destruktive Funktion `nconc` etwa könnte wie folgt definiert werden:

```
(defun our-nconc (l1 l2)
   (setf (rest (last l1)) l2) l1)
```

Beispiel: Destruktives Umdrehen eine Liste

```
(defun our-nreverse (l)
   (our-nreverse-1 nil l))

(defun our-nreverse-1 (hd tl)
   (let ((rest (cdr tl)))
      (setf (cdr tl) hd)
      (if (null rest)
         tl
         (our-nreverse-1 tl rest))))
```

Destruktive Listen-Operationen

Beispiel: Zirkuläre Strukturen

```
(defun circularize! (l)
   (if (null l)
      l
      (setf (cdr (last l)) l)))
```

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

(circularize! l)   ==> (1 2 3 1 2 3 1 2 3 1 ...)
```

Destruktive Listen-Operationen

Wenn destruktive Operationen nicht bedachtsam angewandt werden, können vollkommen undurchschaubare Effekte auftreten:

```
> (defun constr-palindrome (lst)
    (append lst (reverse lst)))
CONSTR-PALINDROME

> (constr-palindrome '(a b c))
(A B C C B A)

> (defun destr-palindrome-1 (lst)
    (append lst (nreverse lst)))
DESTR-PALINDROME-1

> (destr-palindrome-1 '(a b c))
(A C B A)
```

Destruktive Listen-Operationen

Aufgabe: Erklären Sie diese Effekte. Versuchen Sie den Aufruf auch mit folgender Funktion:

```
> (defun destr-palindrome-2 (lst)
    (nconc lst (nreverse lst)))
DESTR-PALINDROME-2
```

(Bemerkung: Top-level interrupt in EMACS mit C-c C-c)

Moral der Geschichte: Möglichst auf destruktive Funktionen verzichten.

Zusammenfassung

Imperative Konstrukte:

- Zuweisungen durch `setf`
- Sequentielle Komposition durch Konstrukte wie `prog1`, `progn` oder `block`
- Iterationskonstrukte wie `dolist`, `dotimes` und `loop`

Ein-/ Ausgabe:

- Bequemste Form der Ausgabe durch `format`
- Eingabe durch `read`

Nachtrag: Es gibt vordefinierte Lisp-Pakete zum Pretty-Printing (z.B. für Einrückungen bei programmiersprachlichen Konstrukten) und passable Parser-Generatoren.

Zusammenfassung

Destruktive Funktionen:

- Durch destruktive Funktionen wird nicht eine neue Liste mit entsprechenden Werten angelegt, sondern die Struktur einer gegebenen Liste verändert
- Destruktive Funktionen können abenteuerliche Effekte zur Folge haben und sollten möglichst vermieden werden.