Praktische Algorithmen der Bioinformatik und Computerlinguistik mit Lisp

Variablen

12.05.2021

LISP

Übersicht:

- Variablenbindung durch `let`
- Lexikalische Sichtbarkeit
- `let` und globale Variablen
- Dynamische Bindungen

Bindung von Variablen durch `let`

Mit `let` läßt sich ein Programmblock mit lokalen Variablen bilden.

```
(let ((var-1 value-1)
      (var-2 value-2)
      ...)
  body ...)
```

Jede Variable wird an den jeweiligen Wert gebunden (keine spezifizierte Reihenfolge!), danach wird der Körper ausgewertet. Das Resultat berechnet sich aus dem Wert der letzten Form in `body`.

```
(let ((x 40)
      (y (+ 1 1)))
  (+ x y))
 ==> 42
```

Bindung von Variablen durch `let`

Bei der Spezialform `let*` werden die Variablen von links nach rechts gebunden; damit kann man also auch frisch eingeführte Variablen dazu benutzen, um den Wert einer neuen Variable zu berechnen.

```
(let* ((x 6)
       (y (* x x)))
  (+ x y))
  ==> 42
```

Vergleich `let` und `lambda`-Abstraktion

Von eher theoretischem Interesse sind folgende
Bemerkungen:

- `(let ((x val)) body)` ist äquivalent zu
  `((lambda (x) body) val)`.
- Das erste obige Beispiel ist z.B. äquivalent zu:

  ```
  ((lambda (x y) (+ x y)) 40 (+ 1 1))
  ```

- Bei Funktionen ohne Seiteneffekte(!) ist
  `(let ((x val)) body)` äquivalent zu einem *`body`*, in dem alle Vorkommen von `x` durch `val` ersetzt worden sind.
  So ist
  `(let ((x (+ 2 3))) (* x x))` äquivalent zu
  `(* (+ 2 3) (+ 2 3))`

Wesentlicher Sinn eines let-Konstrukt ist es, Werte von Zwischenrechungen zu speichern, ohne dabei mit globalen Variablen hantieren zu müssen.

Lexikalische Sichtbarkeit

Durch:

```
(defun name (arglist) body)
(lambda (arglist) body)
(let (arglist) body)
```

wird jeweils ein neuer lexikalischer Block eingeführt, in dem die Variablen aus `arglist` in `body` gebunden sind.

Lexikalische Sichtbarkeit

Variablen-Bindungen sind nur innerhalb ihres Blocks sichtbar:

```
> (defun bindings (x) (+ x y))
BINDINGS
> (bindings 3)
>>Error: The symbol Y has no global value
```

Bindungen in einem inneren Block überschatten Bindungen weiter außen, z.B. in:

```
(let ((x 2)) (let ((x 3)) x))   ==> 3
```

In Common Lisp existiert ein äußerer Bindungsblock, der die globalen Bindungen enthält.

```
> (setf y 4)
4
> (bindings 3)
7
```

Lexikalische Sichtbarkeit

Ein etwas ausführlicheres Beispiel:

```
(defun main (z)
   (contrived-scope-example z 3))

(defun contrived-scope-example (x y)
   (append (let ((x (first x)))
                (list x y))
                x
                z))
> (main '(a b))
>>Error: The symbol Z has no global value
```

Solche Fehler erkennt man rechtzeitig durch Kompilierung:

```
> (compile 'contrived-scope-example)
;;; Warning: Free variable Z assumed to be special
CONTRIVED-SCOPE-EXAMPLE
```

Lexikalische Sichtbarkeit

Folgendes klappt:

```
> (defun contrived-scope-example (x y)
      (append (let ((x (first x)))
                (list x y))
              x))
CONTRIVED-SCOPE-EXAMPLE
> (main '(a b))
(A 3 A B)
```

Techniken: `let` vs. globale Variablen

Häufig führt die Verwendung globaler Variablen zu ungewünschten Nebeneffekten.

Beispiel:

Durch Verwendung von `setf` wird `var` hier als globale Variable behandelt:

```
(defun rec-setf (n)
  (setf var n)
  (format T "Before recursive call: ~S~%" var)
  (if (eql n 0)
      nil
    (rec-setf (- n 1)))
  (format T "After recursive call:  ~S~%" var))
```

Techniken: `let` vs. globale Variablen

Problem: Die Veränderungen an der Variablen bei den rekursiven Aufrufen sind später noch sichtbar:

```
> (rec-setf 2)
Before recursive call: 2
Before recursive call: 1
Before recursive call: 0
After recursive call:  0
After recursive call:  0
After recursive call:  0
NIL
```

Techniken: `let` vs. globale Variablen

Durch Verwendung von `let` wird `var` hier lokal gebunden und bei jeder rekursiven Invokation eine neue Kopie angelegt:

```
(defun rec-let (n)
  (let ((var n))
    (format T "Before recursive call: ~S~%" var)
    (if (eql n 0)
        nil
      (rec-let (- n 1)))
    (format T "After recursive call:  ~S~%" var)))
```

Techniken: `let` vs. globale Variablen

Änderungen können jetzt nur noch lokal geschehen:

```
> (rec-let 2)
Before recursive call: 2
Before recursive call: 1
Before recursive call: 0
After recursive call:  0
After recursive call:  1
After recursive call:  2
NIL
```

`let` zum Verstecken “globaler” Variablen

Es kann erwünscht sein, eine quasi-globale Variable zu haben, auf die aber nur mit bestimmten Zugriffsfunktionen zugegriffen werden kann und die vor ungewollten Änderungen geschützt ist.

Dazu kann eine `let`-Bindung auch auf oberster Lisp-Ebene vorgenommen werden.

`let` zum Verstecken “globaler” Variablen

Beispiel: Auf einen Zähler soll nur zugegriffen werden durch die Funktionen `reset-counter`, `inc-counter`, `read-counter`:

```
(let ((counter 0))

  (defun reset-counter () (setf counter 0))

  (defun inc-counter () (setf counter (1+ counter)))

  (defun read-counter () counter))
```

Mit `setf` wird hier jeweils nur das lokal sichtbare Objekt modifiziert.

Eventuell auch existierende globale Objekte bleiben unberührt.

`let` zum Verstecken “globaler” Variablen

Verwendung des Zählers:

```
> (reset-counter)
0
> (inc-counter)
1
> counter                    ;;; globale Variable

>>Error: The symbol COUNTER has no global value
SYMBOL-VALUE:
   Required arg 0 (S): COUNTER
:C  0: Try evaluating COUNTER again
:A  1: Abort to Lisp Top Level

-> :a
Abort to Lisp Top Level
Back to Lisp Top Level
```

`let` zum Verstecken “globaler” Variablen

```
> (read-counter)
1
> (setf counter 3)     ;;; globale Variable
3
> (read-counter)
1
> counter              ;;; globale Variable
3
```

Lexikalische Closures

- Wird eine Funktion innerhalb einer `let`-Umgebung definiert, so erzeugt Lisp eine sogenannte *lexikalische Closure*. Eine Closure besteht aus einer Funktion zusammen mit der Umgebung von Variablenbindungen, in der sie kreiert wurde.
- Eine Closure wird auch erzeugt, wenn ein Lambda-Ausdruck innerhalb einer lokalen Umgebung erzeugt wird.
- lexikalische Closures sind eine spezielle Art von Funktionen mit Gedächtnis!

Lexikalische Closures

Der Ausdruck

```
(function fn)
#'fn
```

liefert die Closure der Funktion `fn`.

Beispiel:

```
> (defun make-generator ()
    (let ((value 0))
      #'(lambda () (setf value (1+ value)))))
MAKE-GENERATOR

> (setf *generator* (make-generator))
#<Interpreted-Function
    (LAMBDA NIL (SETF VALUE (1+ VALUE))) 126D51E>
```

Lexikalische Closures

```
> (funcall *generator*)
1
> (funcall *generator*)
2
> (funcall *generator*)
3
```

Beachte: `make-generator` gibt eine Closure zurück, d.h. eine Funktion, die über der Variablen `value` abgeschlossen ist.

Dynamische Bindungen

- In manchen Fällen ist es erwünscht, Variablen dynamisch zu binden; d.h. die freien Variablen erhalten ihren Wert aus der Umgebung, in der die Prozedur aufgerufen wurde und nicht aus der Umgebung, in der die Prozedur definiert ist.

- Spezielle Variablen werden dynamisch gebunden.

- Spezielle Variablen werden deklariert mit `defvar` oder aber mit `declare` und `special`.

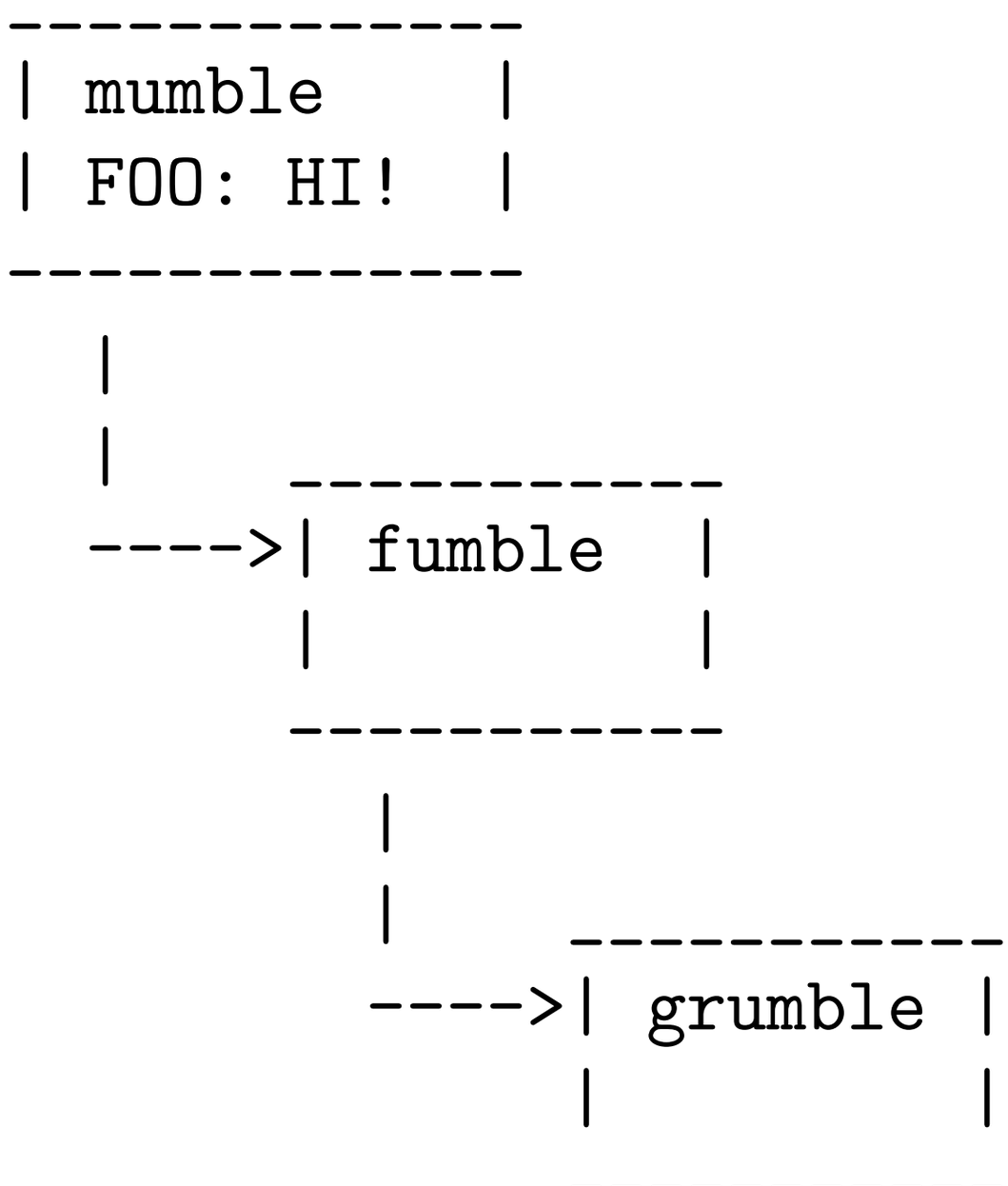
Dynamische Bindungen

Beispiel:

```
(defun mumble (foo)
   (declare (special foo))
   (fumble))
(defun fumble () (grumble))
(defun grumble () foo)
(mumble 'Hi!) ==> HI!
```

Dynamische Bindungen

Aktivierungsumgebung für dynamisch gebundene Variablen.



Dynamische Bindungen

Bemerkung:

- dynamische Bindungen brechen das Prinzip, daß eine Funktion/Prozedur als “black box” angesehen werden kann.
- falls möglich, werden deshalb lexikalisch gebundene Variablen den dynamisch gebundenen vorgezogen.
- dynamisch gebundene Variablen werden häufig dazu benutzt, Information zwischen Funktionen über globale Variablen auszutauschen.

Dynamische Bindungen

Beispiel:

Lexikalischer Bindung kann mit Hilfe dynamischer Bindung umgeschrieben werden zu:

```
(defun main (z)
   (declare (special z))
   (contrived-scope-example z 3))

(defun contrived-scope-example (x y)
  (append (let ((x (first x)))
            (list x y))
          x
          z))
```

Dynamische Bindungen

Beispiel:

Spezielle Variablen können temporär an andere Werte gebunden werden.

```
(defvar *warning-msg* "Problem:")

(defun print-warning (bad-form)
   (let ((response nil))
      (princ *warning-msg*)
      (print bad-form)
      (setf response (read))
      (or response bad-form)))

(defun atom-warning (bad-atom)
   (let ((*warning-msg* "Unrecognized atom:"))
      (print-warning bad-atom)))
```

Zusammenfassung

- Variablen können lokal durch das `let`-Konstrukt gebunden werden
- Globale Variablen sollten möglichst zugunsten von lokalen vermieden werden
- Lexikalische Bindungen wie die durch das `let`-Konstrukt erzeugten finden sich in zahlreichen Programmiersprachen
- Dynamische Bindungen sind eine Spezialität von Lisp.
- Dynamisch gebundene Variablen haben den Charakter von globalen Variablen mit zeitlichem Skopus.