

Klausur
Programmierkurs Lisp
Wintersemester 2001/02

Fachrichtung 4.7 Allgemeine Linguistik
Universität des Saarlandes

21. Februar 2002

- Zeit: 11:00 - 13:00
- Verständlichkeit bringt auch Punkte - also kommentieren und erklären, was man vorhat!
- Hilfsmittel: Alles außer Rechner und Kommilitonen
- Gesamt: 100 Punkte

Aufgabe 1: *Wir spielen Lisp-Interpreter* (10 Punkte)

Zu was evaluieren die folgenden Ausdrücke?

1.

```
(let ((a 1) (b 2))  
  (let ((a b) (b a))  
    (cons a b)))
```
2.

```
(let* ((a 1) (b 2))  
  (let* ((a b) (b a))  
    (append a b)))
```
3.

```
(progn (defun a (a) (+ a 1)) (+ (a 2) (a 3)))
```
4.

```
(let ((y nil))  
  (mapcar (lambda (x) (setq y (cons x y)))  
    (quote (1 2 3))))
```
5.

```
(equal (eq '(1 2) '(1 2)) (eql 1 2))
```

Aufgabe 2: *Closures* (10 Punkte)

Schreibe eine Funktion (`close-enough-producer <diff>`), die mit einem Argument `<diff>` eine Funktion zurückliefert, so daß

```
(funcall (close-enough-producer <diff>) <a> <b>)
```

genau dann T ergibt, wenn die Zahlen `<a>` und `` höchstens um den Betrag `<diff>` unterschiedlich sind.

Zusatzhinweis: Die Common-Lisp-Funktion (`abs <num>`) liefert den Absolutbetrag einer Zahl: (`abs 1`) -> 1 und (`abs -3`) -> 3.

Aufgabe 3: Iteration und/oder Rekursion (12+10+3 Punkte)

1. Schreibe ein Prädikat (`korrekt-geklammert-p <arg>`) (iterativ oder rekursiv), das prüft, ob sein Argument ein korrekt geklammerter String ist, d.h.: zu jeder öffnenden Klammer „(“ gibt es eine passende nachfolgende Klammer „)“ und zu jeder schließenden Klammer „)“ eine passende vorhergehende Klammer „(“:

```
(korrekt-geklammert-p "(i (have (6)) parens)") → T
(korrekt-geklammert-p "((two), too much !)))" → NIL
(korrekt-geklammert-p ")tolly(( worng)") → NIL
```

Hinweis: Benutze z.B. `subseq`: `(subseq "Hallo" 1 3) -> "al"`

2. Das „*Pascalsche Dreieck*“ besteht aus (unendlich vielen) Zeilen, wobei jede Zeile aus der vorhergehenden berechnet wird. Die ersten sieben Zeilen sehen so aus:

```

          1
        1 1
      1 2 1
    1 3 3 1
  1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

- (a) schreibe eine Funktion (`pascal <n>`), welche die Zahlen der n -ten Zeile des Pascalschen Dreiecks in einer Liste zurückgibt.
- (b) Was bedeutet es, wenn eine Funktion restrekursiv ist? Gibt es eine restrekursive Lösung für Teil (a)?

Wenn ja, schreibe sie auf; wenn nein, begründe, warum nicht!
(Wenn dir Teil (a) fehlt, schreibe alternativ dazu ein restrekursives `reverse`)

Aufgabe 4: Makros (5+5+5 Punkte)

1. Angenommen, wir wollen eine Liste als Stack benutzen. Die destruktive Operation (`push <object> <stack>`) ersetzt eine an ein Symbol gebundene Liste durch eine anfangs um `<object>` erweiterte, wie im Beispiel gezeigt:

```
USER(8): a  
(HUND KATZE MAUS)  
USER(9): (push 'vogel a)  
(VOGEL HUND KATZE MAUS)  
USER(10): a  
(VOGEL HUND KATZE MAUS)
```

- (a) schreibe ein Makro `my-push-without`, ohne Backquote, Unquote oder Splice zu benutzen, (und natürlich auch, ohne das in Lisp eingebaute `push` zu benutzen),
- (b) schreibe eine Version von `my-push-with` mit Benutzung von Backquote, Unquote und/oder Splice,
- (c) Kann man `push` auch als Funktion schreiben? Wenn ja, schreibe eine solche Funktion `my-push-function`; wenn nein, begründe, warum es nicht geht.

Aufgabe 5: Multiples Mapping (10+10 Punkte)

1. Benutze (`mapcar <function> <list> &rest <more-lists>`), um die Funktion (`my-pairlis <keylist> <datalist> &optional <alist>`) zu schreiben. Zur Erinnerung:

```
(pairlis '(a b) '(1 2) '((c . 3)))  
-> ((B . 2) (A . 1) (C . 3))
```

2. Benutze `mapcar`, um eine Funktion (`unpairlis <alist>`) zu schreiben, die *zwei* Listen zurückgibt: die Liste der Keys und die Liste der Daten. Beispiel:

```
(unpairlis '((a . b) (c . d) (e . f)))  
-> (A C E) und (B D F)
```

Aufgabe 6: *Abstrakte Datentypen* (5+10+5 Punkte)

In der Vorlesung wurden Assoclisten benutzt, um ein Lexikon zu speichern (also eine Zuordnung von Wörtern (Strings) zu Kategorien (Symbolen)). Die benutzte Methode war aber nicht sehr effizient, weil eine lange Liste durchsucht werden mußte, um eine Zuordnung zu finden. Ein Lexikon enthält typischerweise viele Wörter, wobei jedes einigen (wenigen) Kategorien zugeordnet werden kann.

In dieser Aufgabe soll ein ADT „Hashtabellen-Lexikon“ entworfen werden, der eine Hashtabelle benutzt, um die Zuordnung Wort \rightarrow Kategorie(n) effizienter zugreifbar zu machen (ein Hashtabelleneintrag kann in konstanter Zeit gefunden werden).

1. Erläutere kurz, wie ein Hashtabellen-Lexikon funktionieren kann,
2. Schreibe dafür:

- (a) einen Konstruktor,
- (b) eine Funktion, die eine Eingabe der Form

```
(("andrew" N) ("and" CONJ) ("andrew" N) ("saw" VTRANS)
("the" DET) ("saw" N))
```

korrekt ins Hashtabellen-Lexikon einfügt,

- (c) eine Funktion, die alle Kategorien eines Wortes zurückliefert,
- (d) eine Funktion, die alle Wörter einer Kategorie zurückliefert,
- (e) eine Funktion, die alle Kategorien im Lexikon zurückliefert.

Zusatzhinweis: Manchmal kann man Geschwindigkeit gewinnen, wenn man Speicherplatz opfert.

3. Nenne noch zwei weitere mögliche Funktionen, die für diesen Datentyp sinnvoll wären.

Schmökeraufgabe: (5 Extrapunkte)

Die *Fibonacci-Zahlen* sind folgendermaßen definiert:

$$fib(1) = 1$$

$$fib(2) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2) \text{ für alle } n > 2$$

Die „naive“ Funktion `fib` ist zu einfach für eine Schmökeraufgabe. Aber:

- In der „naiven“ Version dauert die Berechnung für große Zahlen *unproportional* lange. Mit einem kleinen Trick kann man jedoch erreichen, daß `(fib 20)` nur (ungefähr) doppelt so lange dauert wie `(fib 10)`. Schreibe eine Funktion, die $fib(n)$ mit einem solchen Trick berechnet.