PSPP-Praktikum 10
Funktionale Programmierung (Teil 1)

1. Iteration

Um mehrfach einen bestimmten Wert in einer Liste anzulegen, könnte folgende Funktion verwendet werden:[1]

```
(defun repeat (times value)
    (mapcar (lambda (n) value)
            (range times)))
```

Dabei wird die Funktion *range* verwendet, die Sie bereits in einem Praktikum implementiert haben. Die Funktion *repeat* kann einfach eingesetzt werden:

```
> (repeat 5 'hello)
(HELLO HELLO HELLO HELLO HELLO)
```

Das ist soweit gut aber noch ziemlich eingeschränkt. Eine verallgemeinerte Version der Funktion könnte folgendermassen definiert werden:

```
(defun repeatedly (times fun)
    (mapcar fun (range times)))
```

Im Sinne der funktionalen Programmierung wird hier nicht ein Wert übergeben, sondern eine Funktion, welche die Werte bestimmt. Die neue Funktion ist flexibler als die alte.

Aufgaben

Wie können Sie mit *repeatedly* die folgenden Probleme lösen:

- Anlegen einer Liste, die fünfmal das Symbol HELLO enthält (also wie oben mit *repeat*).
- Eine Liste, die fünf zufällige Würfelergebnisse (1..6) enthält.
  Hinweis: Funktion *random*, Beschreibung im CLTL2.
- Eine Liste mit fünf IDs: `("id0" "id1" "id2" "id3" "id4")`
  Beachten Sie dazu die folgende Interaktion in der REPL:

```
> (concatenate 'string "Aufgabe " (write-to-string 12))
"Aufgabe 12"
```

[1] Dieses und weitere Beispiele in diesem Praktikum sind angelehnt an: Functional JavaScript, Introducing Functional Programming with Underscore.js (Michael Fogus, O'Reilly, 2013).

Für die erste dieser Aufgaben wäre es sinnvoll, eine Funktion zu haben, die immer denselben Wert liefert, unabhängig davon, mit welchen Werten die Funktion aufgerufen wird:

```
> (always 42)
#<FUNCTION :LAMBDA (&REST R) VAL>
> (setfun answer (always 42))
ANSWER
> (answer)
42
> (answer nil "?" :ok)
42
> (answer "was soll das?")
42
```

Hinweise

- Vielleicht ist es etwas ungewohnt, aber *always* liefert eine Funktion als Rückgabewert, eine Funktion, welche selbst immer einen bestimmten Wert liefert. Dadurch kann der Aufruf an Stellen stehen, an denen eine Funktion erwartet wird.
- Das Makro *setfun* kennen Sie aus dem Unterricht. Ähnlich wie das vordefinierte *defun* dient es dazu, eine Funktion einem Symbol zuzuordnen. Im Unterschied zu *defun* kann *setfun* aber beispielsweise eine Funktion zuweisen, die von einer anderen Funktion zurückgegeben wird.
- Das Makro *setfun* ist nicht Teil von Common Lisp. Sie finden die Definitionen in der Datei *pspp_basics.lisp*, die Sie mit *load* in der Lisp-REPL laden können. Das gilt auch für weitere in den folgenden Aufgaben benötigte Funktionen, zum Beispiel *partial*.

Aufgaben und Fragen

- Implementieren Sie die Funktion *always*.
- Wird hier eine *Closure* verwendet?
- Geben Sie an, wie Sie die Liste, welche fünfmal das Symbol HELLO enthält (erste der Aufgaben oben) nun mit *repeatedly* und *always* erzeugen können.
- Ist *always* eine Funktion höherer Ordnung?

2. Funktionen dekorieren

Für diese Aufgabe benötigen wir ein paar selbst definierte Funktionen:

- wrap-fn
- dispatch
- parse-int
- parse-float

Die Funktionen *wrap-fn* und *dispatch* wurden im Unterricht besprochen. Sie sind in den Folien sowie in der Datei *pspp_basics.lisp* zu finden. Auch *parse-int* und *parse-float* finden Sie in dieser Datei. Hier ein Beispiel, wie sie verwendet werden können:

```
> (parse-int "24")
24
> (parse-float "24.5")
24.5
```

Die Funktion *wrap-fn* kann man zum „Dekorieren“ von Funktionen verwenden. Dies soll nun anhand eines Beispiels ausprobiert werden. Hier ist eine Sequenz von Funktionsdefinitionen:

```
(setfun num (dispatch #'parse-int #'parse-float #'identity))

(defun num-args (&rest args)
    (mapcar #'num args))
```

Aufgaben und Fragen

- Was macht die Funktion *num*? Demonstrieren Sie dies an ein paar Beispielaufrufen. Welchen Zweck erfüllt die Funktion *identity* in der Definition von *num*?
- Schreiben Sie mit Hilfe von *wrap-fn* und *num-args* eine neue Funktion *add*, welche beliebige Zahlen addieren kann, egal ob sie als Zahlenliteral oder als String vorliegen.
- Demonstrieren Sie die Möglichkeiten der Funktion *add* an ein paar Beispielaufrufen.
- Welche Verbesserungen wären noch sinnvoll? Ergänzen Sie Ihre Implementierung entsprechend.

Die Funktion *add* packt die Addition auf eine bestimmte Art und Weise ein. Beachten Sie, dass die Low-level-Operationen *Verzweigung* und *Wiederholung* hier hinter Funktionsaufrufen versteckt sind.

3. Partielle Anwendung

Mit der Funktion *partial* können für eine Funktion bereits Argumente festgelegt werden. Zurückgegeben wird eine Funktion über die restlichen Argumente:

```
(defun partial (f &rest args)
  (lambda (&rest more-args)
    (apply f (append args more-args))))
```

Möglicher Aufruf:

```
> (setfun three-times (partial #'* 3))
THREE-TIMES
> (three-times 6)
18
> (setfun product (partial #'* 3 14))
PRODUCT
```

Im letzten Beispiel sind bereits alle Argumente vordefiniert. Das kann man als „verzögerte“ Funktionsausführung bezeichnen. Das Ergebnis ist eine Funktion, die nun ohne Parameter aufgerufen werden kann:

```
> (product)
42
```

Diese Möglichkeit soll nun auf die Fakultätsfunktion angewendet werden. Hier ist eine endrekursive Variante der Fakultätsfunktion:

```
(defun factorial (n &optional (fact 1))
  (if (<= n 1) fact
    (factorial (- n 1) (* n fact))))
```

Aufgaben und Fragen

- Welche Kriterien muss eine Funktion erfüllen, um *endrekursiv* (*tail recursive*) zu sein? Wozu dient hier das zweite Argument?
- Passen Sie die Funktion mit Hilfe von *partial* so an, dass der rekursive Aufruf mit allen Argumenten vorbereitet, aber noch nicht ausgeführt wird. Wie muss man die Funktion nun aufrufen, damit die Fakultät von 3 ausgerechnet wird?
- Sehen Sie in den Folien nach der Funktion *trampoline*. Mit ihrer Hilfe kann die „verzögerte“ Fakultätsfunktion nun iterativ aufgelöst werden, so dass kein Stack-Überlauf mehr auftritt. Die meisten Common-Lisp-Implementierungen optimieren endrekursive Funktionen aber sowieso, so dass nicht mit einem Stack-Überlauf gerechnet werden muss. Wie sieht der Aufruf mit *trampoline* aus?

4. Quicksort *(fakultativ)*

Die Funktion *qsort* sei wie folgt in Haskell implementiert. Sie sortiert eine Liste.

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

Haskell behandeln wir in PSPP zwar nicht, aber mit Ihrem Wissen über verschiedene Programmiersprachen (unter anderem: Prolog) können Sie diese Funktion vermutlich lesen.

Hinweise:

- In der Liste x:xs ist x das erste Element und xs die Restliste.
- `[y | y <- xs, y < x]` steht für die Liste aller y aus xs, die kleiner als x sind.
- Der Operator ++ verkettet zwei Listen.

Machen Sie sich zunächst klar, wie *qsort* funktioniert. Versuchen Sie dann, eine ähnlich aufgebaute Sortierfunktion in Common Lisp zu schreiben. Am besten verwenden Sie *filter-list*.

```
> (sort-list '(1 8 3 7 3 6 8 3 7 9 9 4 6))
(1 3 3 3 4 6 6 7 7 8 8 9 9)
```

5. Lisp-Übung: HTML generieren *(fakultativ)*

(ehemalige Prüfungsaufgabe)

Es soll ein Lisp-Programm geschrieben werden, das aus einer internen Darstellung HTML-Code generiert. Dies könnte zum Beispiel in einem in Lisp geschriebenen Content Management System benötigt werden. HTML-Dokumente können als Baumstruktur dargestellt werden, und diese wiederum kann leicht auf eine Lisp-Datenstruktur abgebildet werden. Wir verwenden für diese Aufgabe folgende Abbildung von HTML nach Lisp:

- Elemente (Tags) ohne Inhalt und ohne Attribute sind einfach Lisp-Symbole.
- Sonst werden Elemente als Liste dargestellt, deren *car* der Elementname als Symbol ist. Der Rest der Liste beschreibt Attribute des Elements sowie den Inhalt (untergeordnete Elemente und/ oder Textinhalt).
- Attribute werden als *cons*-Zellen dargestellt, deren *car* der Attributname als Symbol und deren *cdr* der Attributwert als String ist.
- Textinhalt wird als String dargestellt.

Hier ist ein Beispiel:

HTML-Code	Lisp-Repräsentation

`<html>`
`  <head>`
`  <title>Hello Beispiel</title>`
`  </head>`
`  <body>`
`  <div id="nav">Navi</div>`
`  <div id="main">Hello</div>`
`  </body>`
`</html>`
	`(html`
	`  (head`
	`  (title "Hello Beispiel")`
	`  )`
	`  (body`
	`  (div (id . "nav") "Navi")`
	`  (div (id . "main") "Hello"))`
	`  )`
	`)`

a. Zunächst wird eine Funktion benötigt, die HTML-Attribute in dieser Darstellung erkennt. Beispiel-Aufrufe:

```
> (html-is-attr '(id . "nav"))
T
> (html-is-attr '(id  "nav"))
NIL
> (html-is-attr "nav")
NIL
> (html-is-attr 'id)
NIL
```

Geben Sie die Implementierung der Funktion *html-is-attr* an.

b. Nun soll der Rest des Programms geschrieben werden, das aus der Lisp-Darstellung die HTML-Darstellung erzeugt. Beispiel-Aufrufe:

```
> (html-node 'br)
<br />

> (html-node '(p "Hello World"))
<p>Hello World</p>

> (html-node '(img (src . "logo.png")))
<img src="logo.png"></img>

> (html-node '(p "hallo " (em "dies " (strong "ist") " ein") " Test"))
<p>hallo <em>dies <strong>ist</strong> ein</em> Test</p>
```

Aufgabe: Ergänzen Sie das Programm auf der nächsten Seite.

Hinweise:

- Der erzeugte HTML-Code wird direkt mit *format* ausgegeben.
- Die Kontrollsequenz `"~(~A~)"` in einem format-String bewirkt, dass ein Symbol in Kleinbuchstaben ausgegeben wird.
- Zur Ausgabe eines Strings ohne Anführungszeichen wird `"~A"` verwendet, `"~S"` gibt den String mit Anführungszeichen aus.
- `(unless (`*bedingung*`) `*ausdruck1 ausdruck2...*`)`
  entspricht:
  `(if (not (`*bedingung*`)) (progn `*ausdruck1 ausdruck2 ...*`))`

```
(defun html-node (node)
   (cond ((symbolp node)
           (format t "<~(~A~) />" node node))
         ((stringp node)
           (format t "~A" node))
         ((listp node)
           (format t "<~(~A~)" (car node))
```

.....

.....

.....

.....

```
(defun html-subnodes (nodes)
    (unless (null nodes)
        (unless (html-is-attr (car nodes))
```

.....

```
        (html-subnodes (cdr nodes))))

(defun html-nodeattrs (nodes)
    (unless (null nodes)
        (if (html-is-attr (car nodes))
```

.....

```
        (html-nodeattrs (cdr nodes))))
```