PSPP-Praktikum 9
Common Lisp (Teil 2)

Hinweis: Nach den eigentlichen Praktikumsaufgaben hat es noch fakultative Aufgaben, für den Fall dass Sie mit dem Beispiel der Produkteliste noch etwas weiter experimentieren möchten. Wenn etwas nicht auf Anhieb klappt (z.B. eine elegante Lösung der *range*-Funktion, Aufgabe 2): das ist normal beim Lernen einer neuen Programmiersprache, besonders wenn diese sich ziemlich von gängigen Sprachen unterscheidet. Lösen Sie die Aufgaben einfach soweit wie möglich.

1. Weitere Abstraktionen

Im letzten Praktikum haben Sie verschiedene, gleich aufgebaute Funktionen zu einer Funktion *map-list* verallgemeinert (welche in Common Lisp bereits unter dem Namen *mapcar* vorhanden ist). Ausserdem haben Sie Funktionen geschrieben, welche die Elemente einer Liste kombinieren, zum Beispiel addieren oder multiplizieren. Wahrscheinlich sehen Ihre Lösungen etwa so aus:

```
(defun list-sum (seq)
    (cond ((null seq) 0)
          (t (+ (car seq) (list-sum (cdr seq))))))

(defun list-mult (seq)
    (cond ((null seq) 1)
          (t (* (car seq) (list-mult (cdr seq))))))
```

Diese sollen nun ebenfalls verallgemeinert werden.

Aufgabe: Studieren Sie die beiden Funktionen und schreiben Sie eine neue Funktion *reduce-list*, eine verallgemeinerte Version von *list-sum*, *list-mult* und ähnlichen Funktionen. Sie soll drei Parameter haben: eine Funktion, einen Initialwert und die zu verarbeitende Liste.

Tatsächlich existiert auch diese Funktion in Common Lisp bereits. Sie heisst *reduce* und unterstützt diverse Schlüsselwort-Parameter.

Eine weitere gut für die Verarbeitung von Listen geeignete Funktion ist:

```
(defun func (f seq)
  (cond ((null seq) nil)
        ((funcall f (car seq))
            (cons (car seq) (func f (cdr seq))))
        (t (func f (cdr seq)))))
```

Aufgabe: Finden Sie heraus, was diese Funktion macht. Geben Sie ihr einen besseren Namen und testen Sie, ob die Funktion die erwarteten Ergebnisse liefert.

2. Listen erzeugen

In den bisherigen Beispielen haben wir vor allem Listen verarbeitet. Nun soll eine Liste von Zahlen gemäss einer in Form von einem bis drei Argumenten übergebenen Spezifikation erzeugt werden.

Schreiben Sie eine Funktion *range*, die eine Sequenz von Zahlen aus einem bestimmten Bereich erzeugt. Wenn der Funktion nur ein Argument übergeben wird, soll die Zahlenfolge von 0 bis zu dieser Zahl (exklusive) ausgegeben werden. Bei zwei Argumenten stellt das erste Argument die Ausgangszahl (von, inklusive) und das zweite das Ziel (bis, exklusive) dar. In einem dritten Argument kann die Schrittweite angegeben werden. Einige Beispiele sollen das verdeutlichen:

```
> (range 8)
(0 1 2 3 4 5 6 7)

> (range 5 10)
(5 6 7 8 9)

> (range 5 20 3)
(5 8 11 14 17)

> (range 10 2 -1)
(10 9 8 7 6 5 4 3)
```

```
> (range 5 5)
NIL

> (range 5 2)
NIL

> (range -5 0)
(-5 -4 -3 -2 -1)

> (range -5)
NIL
```

Das ist nicht ganz einfach. Überlegen Sie erst, wie Sie ein einfacheres Problem lösen könnten: eine Funktion *range-simple*, die immer zwei Parameter hat, *von* und *bis*. Ist eine rekursive Lösung möglich? Wenn Sie nicht weiterkommen, können Sie sich am Lösungshinweis in der Fussnote[1] orientieren.

Die Funktion *range-simple* können Sie dann zur flexibleren Variante verallgemeinern. Gelingt dies nicht, geben Sie *range-simple* ab.

3. Fakultät

Schreiben Sie die Fakultätsfunktion mit Hilfe von *range*.

```
(defun factorial (n)
    (apply #'* (...)))
```

[1] Rekursiver Ansatz: Falls die Abbruchbedingung noch nicht erfüllt ist, wird der Startwert mit *cons* in die Liste eingefügt, die der rekursive Aufruf erzeugt. Was ist der neue Startwert im rekursiven Aufruf?

4. CSV-Datei einlesen

In dieser und der folgenden Aufgabe sollen verschiedene Datenformate und ihre Umwandlung in andere Formate untersucht werden. Dreh- und Angelpunkt in Lisp sind dabei natürlich Listen.

Wir gehen von einer kleinen Liste von Produktbeschreibungen aus, die zunächst im CSV-Format *(Comma Separated Values)* in der Datei *products.csv* vorliegen. Dieses Textformat ist universell einsetzbar und zum Beispiel auch problemlos von Tabellenkalkulationsprogrammen wie Excel im- und exportierbar. Die Liste enthält zwar nur ein paar Beispieleinträge, mit tausenden von Einträgen würde es aber genauso gehen. In der ersten Zeile werden die Attribute der einzelnen Datensätze beschrieben, ab der zweiten Zeile folgen die Daten:

```
Id,Kategorie,Hersteller,Produkt,Menge,Preis
101,Teigwaren,Barilla,Penne Rigate - 73,1 kg,3.35
102,Teigwaren,Barilla,Spaghetti - 5,1 kg,3.35
...
404,Schokolade,Frey,Milch Extra Schokolade,3 x 100 g,3.90
501,Getraenke,Ice Tea,Ice Tea Classic,4 x 1 l,3.20
```

Hier ist die ursprüngliche Darstellung im Excel:

	A	B	C	D	E	F
1	Id	Kategorie	Hersteller	Produkt	Menge	Preis
2	101	Teigwaren	Barilla	Penne Rigate - 73	1 kg	3.35
3	102	Teigwaren	Barilla	Spaghetti - 5	1 kg	3.35
4						
5	402	Schokolade	Lindt	Excellence Extra cremige Milch-Chocolade	100 g	2.50
6	403	Schokolade	Cailler	Milchschokolade	400 g	7.30
7	404	Schokolade	Frey	Milch Extra Schokolade	3 x 100 g	3.90
8	501	Getraenke	Ice Tea	Ice Tea Classic	4 x 1 l	3.20
9						

Erstes Ziel ist, eine solche Datei einzulesen und in eine von Lisp gut zu verarbeitende Form zu bringen. Dazu lesen wir die Datei zunächst mit folgender Funktion in einen String ein:[2]

```
(defun file-to-string (path)
  (with-open-file (stream path)
    (let ((data (make-string (file-length stream))))
      (read-sequence data stream)
      data)))
```

Da es hier nicht vorrangig um Dateiverarbeitung geht, ist die Funktion vorgegeben. Bei Interesse können Sie die verwendeten Funktionen, die Sie noch nicht kennen, im CLTL2 nachschlagen. Noch ein Hinweis: Wenn mit grossen Dateien umgegangen werden soll, ist die Vorgehensweise,

[2] Für den Fall, dass Ihre Common-Lisp-Implementierung die *read-sequence*-Funktion nicht kennt, ist in *file-to-string.lisp* eine alternative Implementierung angegeben.

zunächst die ganze Datei zu lesen und dann erst zu verarbeiten, ungünstig. Bei CSV-Dateien bietet es sich an, diese zeilenweise vom Stream zu lesen und jede Zeile gleich zu verarbeiten.

Um die eingelesene Datei zu verarbeiten, benötigen wir eine weitere Hilfsfunktion: *string-split*. Diese soll einen String an einem vorgegebenen Zeichen auftrennen und eine Liste zurückgeben:

```
> (string-split #\Space "eins zwei drei")
("eins" "zwei" "drei")
> (string-split #\, "vier,fünf,sechs")
("vier" "fünf" "sechs")
```

Wir implementieren diese Funktion rekursiv. Auch hier werden einige String-Funktionen benötigt. Um zu verstehen, wie *string-split* arbeitet, sollten Sie die Funktionen *position* und *subseq* im CLTL2 nachschlagen. **Aufgabe:** Ergänzen Sie die fehlende Zeile der Funktion:

```
(defun string-split (c strg &optional (start 0))
  (let ((end (position c strg :start start)))
    (cond (end (cons (subseq strg start end)
                     ;; hier fehlt noch etwas
           (t (list (subseq strg start))))))
```

Nun zur eigentlichen **Aufgabe:** Schreiben Sie eine Funktion *simple-csv*, die einen CSV-String als Argument bekommt und eine Listenstruktur liefert:

```
> (simple-csv (file-to-string "products.csv"))
(("Id" "Kategorie" "Hersteller" "Produkt" "Menge" "Preis")
 ("101" "Teigwaren" "Barilla" "Penne Rigate - 73" "1 kg" "3.35")
 ("102" "Teigwaren" "Barilla" "Spaghetti - 5" "1 kg" "3.35")
 ...)
```

Diese erste Version eines CSV-Parsers muss gar nicht robust sein. Nehmen Sie an, dass sie nur korrekte CSV-Dateien zur Verarbeitung bekommt. Das Problem kann auf verschiedene Weise gelöst werden. Hilfreich sind die Funktionen *string-split* und die Common-Lisp-Funktion *mapcar*. Die Zeichen zum Zerlegen der Strings werden in Common Lisp als Zeichenliterale angegeben:

`#\Newline` und `#\,` (je nach Zeilenendezeichen auch: `#\Return`)

5. Assoziationsliste

Hinweis: Falls das Einlesen der CSV-Datei in Aufgabe 4 noch nicht funktioniert, können Sie in dieser Aufgabe mit *products.lisp* fortfahren.

Assoziationslisten sind Listen von Schlüssel-Wert-Paaren, welche als *cons*-Zellen angelegt werden. Sie lassen sich mit der Funktion *assoc* durchsuchen:

```
> (defvar *people* '(("name" . "Hans") ("age" . "29") ("hair" . "braun")))
*PEOPLE*
> (assoc "age" *people* :test #'string=)
("age" . "29")
```

Als Schlüssel werden dabei anstelle von Strings oft Symbole aus dem Keyword-Package verwendet, also zum Beispiel *:name, :age, :hair.*

Die Produktliste soll nun in eine solche Assoziationsliste umgeformt werden. Dazu benötigen wir die Attributbezeichnungen, die in der CSV-Datei in der ersten Zeile standen. Es wird daher noch einmal die komplette Datei geladen:

```
> (defvar *products-csv* (simple-csv (file-to-string " products.csv")))
*PRODUCTS*
```

Zunächst implementieren wir eine Hilfsfunktion *zip-to-alist*. Beschreiben Sie, was diese Funktion macht und testen Sie die Funktion mit Beispieldaten:

```
(defun zip-to-alist (lst1 lst2)
  (cond ((or (null lst1) (null lst2)) nil)
        (t (cons (cons (car lst1) (car lst2))
                 (zip-to-alist (cdr lst1) (cdr lst2))))))
```

Implementieren Sie nun die Funktion *make-alist*:

```
> (make-alist *products-csv*)
((("Id" . "101") ("Kategorie" . "Teigwaren") ("Hersteller" . "Barilla")
  ("Produkt" . "Penne Rigate - 73") ("Menge" . "1 kg") ("Preis" . "3.35"))
 (("Id" . "102") ("Kategorie" . "Teigwaren") ("Hersteller" . "Barilla")
  ("Produkt" . "Spaghetti - 5") ("Menge" . "1 kg") ("Preis" . "3.35"))
 ...)
```

Basierend auf diesem Format könnte man nun zahlreiche weitere Funktionen schreiben, zum Beispiel zur Auswahl von Einträgen, zur Beschränkung der anzuzeigenden Attribute, usw.

Einige Erkenntnisse

- Listen sind flexible Datenstrukturen. Je nach Erfordernissen können verschiedene Formate zu oder von Listen konvertiert werden. Assoziationslisten repräsentieren Sammlungen von Schlüssel-Wert-Paaren. Bei Bedarf unterstützt Common Lisp aber auch Hashtabellen.
- Für viele Zwecke existieren in Common Lisp bereits vordefinierte Funktionen. Viele davon dienen zum Verarbeiten von Listen.
- Wenn es für einen bestimmten Zweck keine Funktion gibt, kann das gewünschte Ergebnis häufig durch Kombination bestehender Funktionen erzielt werden. Dies steht und fällt natürlich mit den Kombinationsmöglichkeiten. Funktionen höherer Ordnung (Funktionen als Parameter und/oder Rückgabewert) sind ein mächtiger Kombinationsmechanismus. (Vergleich: Unix-Shell, zahlreiche Befehle, Kombination durch Pipes, Scripts, etc.).
- Wenn auch das nicht möglich ist, können eigene Funktionen geschrieben werden. Oft werden auch diese sehr allgemein einsetzbar sein und können auch in anderen Programmen nützlich sein. Beispiele aus diesem Praktikum sind: *string-split, zip-to-alist, make-alist*. Auf diese Weise wird die Menge der verfügbaren Funktionen ständig erweitert.
- Der Umgang mit CSV-Daten oder JSON (Aufgabe 7) ist in Common-Lisp-Programmen problemlos möglich

Common Lisp Ökosystem

Nicht ganz neu, aber eine schöne Zusammenfassung der wichtigsten Entwicklungen rund um Common List Stand 2015 finden Sie in diesem Artikel:

http://borretti.me/article/common-lisp-sotu-2015

6. Arbeit mit der Produktliste *(fakultativ)*

Nach Abschluss von Aufgabe 4 haben wir die Produkteliste in einem für Lisp gut zu verarbeitenden Format. Am besten weisen Sie das Ergebnis einer globalen Variablen zu, um möglichst einfach verschiedene Experimente damit machen zu können:

```
> (defvar *products* (cdr (simple-csv (file-to-string "products.csv"))))
*PRODUCTS*
```

Achtung: Mit *defvar* definieren Sie eine dynamische Variable und weisen ihr einen Initialwert zu. Möchten Sie den Wert anschliessend ändern, geht das nicht mehr mit *defvar*, sondern mit *set*.

Die Liste ist nun an das Symbol **products** gebunden. Wozu wurde *cdr* verwendet?

Aufgaben:

a. Schreiben Sie einen Lisp-Ausdruck, mit dem sie die Kategorien der Produkte erhalten:

```
("Teigwaren" "Teigwaren" ... "Getraenke")
```

b. Schreiben Sie eine Funktion *unique* mit einem Keyword-Argument *:test*, welche Duplikate aus einer Liste entfernt (die Funktion *string=* dient zum Vergleichen von Strings):

```
> (unique '("Teigwaren" "Teigwaren" "Getraenke") :test #'string=)
("Teigwaren" "Getraenke")
```

c. Schreiben Sie eine Funktion *prod-categories*, die eine Liste der Produktkategorien erstellt (ohne Duplikate):

```
> (prod-categories *products*)
("Teigwaren" "Reis" "Salz&Zucker" "Schokolade" "Getraenke")
```

d. Schreiben Sie eine Funktion *prod-avg-price*, die den Durchschnittspreis aller Produkte aus der Liste bestimmt:

```
> (prod-avg-price *products*)
3.1764705
```

Ein kleines Problem dabei ist das Verarbeiten der Einzelpreise, die noch als Strings vorliegen. Es gibt in Common Lisp zwar eine vordefinierte Funktion *parse-integer* aber keine Funktion *parse-float*. Eine schnelle Implementierung könnte so aussehen (wir lesen von einem String und prüfen, ob das Ergebnis ein Float ist):

```
(defun parse-float (strg)
  (with-input-from-string (s strg)
    (let ((res (read s)))
      (if (eq (type-of res) 'single-float) res nil))))
```

7. Quicklisp und JSON *(fakultativ)*

Von Assoziationslisten ist es kein weiter Weg mehr zu JSON. Als Common Lisp standardisiert wurde, gab es noch kein JavaScript und folglich auch kein JSON. Import- und Exportmöglichkeiten für JSON lassen sich aber leicht nachrüsten. Dazu können wir mit dem Paketmanager *Quicklisp* das Paket *cl-json* installieren.

Zunächst zum Paketmanager: Quicklisp ist eine Paketverwaltung für die gängigen Common-Lisp-Installationen. Es liegt bisher erst in einer Betaversion vor. Zur Installation wird die Datei *quicklisp.lisp* von der Website *http://beta.quicklisp.org* geladen und ausgeführt. Ablauf am Beispiel SBCL:

```
$ curl -O http://beta.quicklisp.org/quicklisp.lisp
$ curl -O http://beta.quicklisp.org/quicklisp.lisp.asc
```

An dieser Stelle wäre die Signatur der geladenen Datei zu überprüfen. Dann geht es weiter:

```
$ sbcl --load quicklisp.lisp
... diverse Ausgaben ...
* (quicklisp-quickstart:install)
... diverse Ausgaben ...
* (ql:add-to-init-file)
... damit Quicklisp immer zur Verfügung steht ...
```

Damit ist Quicklisp installiert. Hier werden alle Pakete mit „json“ im Namen aufgelistet und anschliessend wird das Paket *cl-json* geladen:

```
* (ql:system-apropos "json")
#<SYSTEM cl-json / cl-json-20141217-git / quicklisp 2015-01-13>
#<SYSTEM cl-json.test / cl-json-20141217-git / quicklisp 2015-01-13>
...
* (ql:quickload "cl-json")
... ; Loading "cl-json" ...
```

Nun können wir die Produktliste nach JSON exportieren.

```
* (json:encode-json (make-alist *products*))
[{"Id":"101","Kategorie":"Teigwaren","Hersteller":"Barilla","Produkt":"Penne Rigate -
73","Menge":"1 kg","Preis":"3.35"},{"Id":"102","Kategorie":"Teigwaren", "Hersteller":
"Barilla","Produkt":"Spaghetti - 5","Menge":"1 kg","Preis":"3.35"},{"Id":"103",
"Kategorie":"Teigwaren","Hersteller":"Barilla", ...}]
```

Importieren von JSON-Code ist ebenso einfach. Die Möglichkeiten sind auf der Website *https://common-lisp.net/project/cl-json/* beschrieben.