PSPP-Praktikum 11
Funktionale Programmierung (Teil 2)

1. Funktionale Problemlösung

Im Unterricht wurde in einem Beispiel gezeigt, wie eine im JSON-Format vorliegende Taskliste so verarbeitet werden kann, dass nur noch die Aufgaben einer bestimmten Person, die noch nicht abgeschlossen sind, reduziert auf bestimmte Angaben (ID, Priorität, Titel, Datum) und sortiert nach dem Fälligkeitsdatum ausgegeben werden. Die angegebene Lösung wurde in JavaScript sowohl imperativ und objektorientiert als auch funktional umgesetzt.

Das Vorgehen für eine funktionale Lösung soll nun in Common Lisp nachvollzogen werden. Die zu verarbeitenden Daten liegen in einer Datei *tasks.json* vor:

```
{ result: 'SUCCESS',
  interfaceVersion: '1.0.3',
  requested: '10/17/2013 15:31:20',
  lastUpdated: '10/16/2013 10:52:39',
  tasks:
   [ { id: 104,
       complete: false,
       priority: 'high',
       dueDate: '11/29/2013',
       member: 'Scott',
       title: 'Do something',
       created: '9/22/2013' },
     { id: 105,
       complete: false,
       priority: 'medium',
       dueDate: '11/22/2013',
       member: 'Lena',
       title: 'Do something else',
       created: '9/22/2013' },
    ... ]
}
```

Zum Einlesen und Verarbeiten der JSON-Datei wird ein JSON-Paket benötigt. Das kann mit dem Paketmanager Quicklisp installiert werden – wie in der fakultativen Aufgabe des vorletzten Praktikums beschrieben. Falls Sie die entsprechenden Installationen nicht gemacht haben (und nicht nachholen möchten, können Sie für das Einlesen auch von der bereits in Lisp-Strukturen konvertierten Datei *tasks.lisp* ausgehen und die folgenden Schritte entsprechend anpassen.

Aufgaben:

- Evaluieren Sie den folgenden Ausdruck (am besten öffnen Sie den Lisp-Interpreter im Verzeichnis, in dem die Datei *tasks.json* liegt, dann brauchen Sie sich nicht um die Pfadangaben zu kümmern):

```
> (with-input-from-string (s (file-to-string "tasks.json")) (json:decode-json s))
((:RESULT . "SUCCESS") (:INTERFACE-VERSION . "1.0.3")
 (:REQUESTED . "10/17/2013 15:31:20") (:LAST-UPDATED . "10/16/2013 10:52:39")
 (:TASKS
  ((:ID . 104) (:COMPLETE) (:PRIORITY . "high") (:DUE-DATE . "11/29/2013")
   (:MEMBER . "Scott") (:TITLE . "Do something") (:CREATED . "9/22/2013"))
 ...))
```

  Ohne das JSON-Paket laden Sie direkt die Lisp-Datei:

```
> (with-open-file (stream "tasks.lisp" :direction :input) (read stream))
```

- Untersuchen Sie den Aufbau des ausgegebenen Ausdrucks. Was macht das Makro *with-input-from-string* (*width-open-file*)? Sehen Sie bei Bedarf im CLTL2 nach.
- Vergleichen Sie die JSON-Daten mit der resultierenden Lisp-Struktur. Welche Datentypen werden verwendet? Welche Bezeichnungen wurden geändert?
- Erstellen Sie eine Funktion *read-json*, die nur einen Dateinamen als Parameter erhält und die zugehörige Lisp-Struktur liefert (wie gerade gezeigt).

Nun sind noch einige weitere Umformungen an der Listenstruktur nötig, um zum gewünschten Ergebnis zu kommen. Wir werden diese mit möglichst allgemein einsetzbaren Funktionen implementieren, die auch in anderen Programmen nützlich sein werden.

Aufgabe:

Schreiben Sie eine Funktion *getprop-fn*, die für einen Schlüssel (erster Parameter) einer solchen Assoziationsliste (zweiter Parameter) den zugehörigen Wert liefert. Das ist noch relativ einfach.

```
> (getprop-fn :result '((:RESULT . "SUCCESS") (:INTERFACE-VERSION . "1.0.3")))
"SUCCESS"
```

In den Demos zum Funktionalen Programmieren haben wir eine Funktion *curry* verwendet. Diese erzeugt streng genommen keine curryfizierte Funktion, sondern eine, welche partiell angewendet werden kann. Erhält sie zu wenig Argumente, wird eine – ebenfalls mit *curry* angepasste – Funktion über die restlichen Parameter zurückgegeben. Sie finden die Funktion in der Funktionssammlung *more-functional.lisp*. Dort ist auch eine einfachere Variante *curry-n* enthalten:

```
(defun curry-n (f numargs)
  (lambda (&rest args)
    (if (>= (length args) numargs)
        (apply f args)
        (curry-n
          (lambda (&rest restargs) (apply f (append args restargs)))
          (- numargs (length args))))))
```

Aufgaben:

- Verwenden Sie *curry* (oder *curry-n*) um eine Funktion *getprop* zu erstellen, die *getprop-fn* entspricht, aber partiell angewendet werden kann.
- Binden Sie das Ergebnis eines Aufrufs mit dem Schlüssel *:TASKS* für Testzwecke an eine Variable **tasks**.[1]

```
> (getprop :tasks (read-json "tasks.json"))
??
> (getprop :tasks)
??
> (defvar *tasks* (getprop :tasks (read-json "tasks.json")))
...
```

Die beiden folgenden Funktionen sind eigentlich nur Aufrufe von bereits in Common Lisp vordefinierten Funktionen. Ausserdem soll eine weitere Funktion *prop-eq* implementiert werden.

```
(defun filter-fn (f seq)
  (remove-if-not f seq))

(defun reject-fn (f seq)
  (remove-if f seq))

(defun prop-eq (prop val)
  (pipeline (getprop prop) (partial #'equal val)))
```

Aufgaben:

- Machen Sie sich klar, wozu *filter-fn* und *reject-fn* gut sind und testen Sie die Funktionen an ein paar Beispielen.
- Verwenden Sie *curry*, um partiell anwendbare Funktionen *filter* und *reject* aus diesen Funktionen zu erstellen.

[1] Hinweis zu *defvar*: Es wird eine Spezialvariable mit dynamischem Gültigkeitsbereich. Sie wird nur beim ersten Aufruf initialisiert. Anschliessen kann der Wert mit *setq* geändert werden: *(setq *tasks* '(…))*

- Die Funktion *prop-eq* ist etwas speziell. Sie verwendet zwei bereits früher verwendete Funktionen *pipeline* und *partial*. Was liefert die Funktion *prop-eq* als Ergebnis? Beschreiben Sie, was die Funktion macht (am besten bevor Sie die Funktion testen).
- Testen Sie die folgenden Aufrufe:

```
> (filter (prop-eq :member "Scott") *tasks*)
??
> (filter (prop-eq :member "Scott"))
??
```

Jetzt werden noch weitere Funktionen implementiert:

```
(defun pick-fn (attrs obj)
  (remove-if-not #'(lambda (el) (member (car el) attrs)) obj))

(setfun pick (curry #'pick-fn 2))
(setfun forall (curry #'mapcar 2))
```

Aufgaben:

- Beschreiben Sie, was die Funktion *pick* macht.
- Erzeugen Sie mit Hilfe von *pick* und *forall* eine Liste aus **tasks**, welche nur die Attribute *:due-date* und *:title* enthält.

Ein Problem ist noch offen. Das Ergebnis soll nach Fälligkeitsdatum sortiert werden (ein Problem, das im JavaScript-Beispiel übrigens auch nicht komplett gelöst ist). Zum Sortieren wandeln wir das Datum in eine Zahl um – genau genommen die *Lisp Universal Time* (das ist die Anzahl Sekunden seit dem 1.1.1900):

```
> (date-to-universal "11/15/2013")  ; noch zu implementieren
3593458800
> (string-split #\/ "11/15/2013")
("11" "15" "2013")
```

Aufgaben:

- Implementieren Sie die Funktion *date-to-universal*. Hilfreich sind dabei die bereits früher implementierte Funktion *string-split* und die Common-Lisp-Funktionen *parse-integer* und *encode-universal-time* Die Datumsangaben sind übrigens im Format *mm/dd/yyyy*. Als Uhrzeit setzen Sie einfach 00:00:00 Uhr.
- Im ersten Anlauf werden Sie *date-to-universal* vermutlich so implementieren, dass Sie die Elemente der Liste einzeln auswählen und an *encode-universal-time* übergeben. In Common Lisp ist es auch möglich, dass eine Funktion mehrere Werte zurückgibt, die dann mit Hilfe von *multiple-value-bind* gebunden werden können. So ist recht elegant eine Änderung der Reihenfolge möglich:

```
> (multiple-value-bind (m d y)
      (values-list '(11 15 2021))
    (format t "~d.~d.~d" d m y))
15.11.2021
```

Mit der Umwandlung der Datumsangabe in eine ganze Zahl haben wir eine einfache Möglichkeit zur Sortierung nach Datumsangaben. Eine Hilfsfunktion *sort-by* soll das Sortieren erleichtern: Die übergebene Funktion f dient dazu, aus den zu sortierenden Elementen einen Wert zu bestimmen, der mit #'< verglichen werden kann.

```
(defun sort-by-fn (f seq)
  (sort (copy-list seq)
        (lambda (a b) (< (funcall f a) (funcall f b)))))

(setfun sort-by (curry #'sort-by-fn 2))
```

Jetzt haben wir alle Teile zusammen, um die zu Beginn gestellte Aufgabe einfach und elegant zu lösen: Liefere die Aufgaben einer bestimmten Person, die noch nicht abgeschlossen sind, reduziert auf bestimmte Angaben (ID, Priorität, Titel, Datum) und sortiert nach dem Fälligkeitsdatum. Zur Lösung setzen wir mehrere Funktionen in einer Pipeline zusammen:

```
(defun open-tasks (name)
  (pipeline
    (getprop :tasks)
    (filter (prop-eq :member name))
    (reject (prop-eq :complete t))
    (forall (pick '(:id :due-date :title :priority)))
    (sort-by (pipeline (getprop :due-date) #'date-to-universal))))
```

Aufgabe: Die resultierende Funktion muss nun nur noch mit den Daten der Tasklist aufgerufen werden. Wie kann man damit nun die offenen Tasks für *Scott* aus *tasks.json* extrahieren?

Ergebnisse

- Ausser diesen wenigen Zeilen ist kaum etwas an der Lösung problemspezifisch. Die oben definierten Funktionen *getprop*, *filter*, *reject*, *forall*, *pick* und *sort-by* sind so allgemein, dass sie in vielen anderen Programmen eingesetzt werden können und gut in einer allgemeinen Programmbibliothek untergebracht werden können. **Die Lösung unseres Tasklist-Problems schrumpft dann tatsächlich auf diese wenigen Zeilen.**

- Ein Merkmal funktionaler Programmierung: Soweit möglich allgemein einsetzbare Datenstrukturen mit möglichst allgemein einsetzbaren Funktionen bearbeiten, um Schritt für Schritt von der Ausgangsform in die gewünschte Form umzuwandeln. Vergleich: Fliessband.

- Die Pipeline bekommt Funktionen, die erst noch ausgeführt werden müssen. Das Filtern, Sortieren, etc. wurde dabei noch gar nicht durchgeführt, sondern auf einen späteren Zeitpunkt verschoben. Zurückgegeben wird ebenfalls wieder eine Funktion. Die Sortierfunktion wurde übrigens ebenfalls durch eine Pipeline zusammengesetzt: Extrahieren der Property und Umformen in die Universalzeit.

- Ausser dem Parameter werden in der Lösung keine Variablen verwendet. Somit spielen auch Zuweisungsoperationen keine Rolle. Auch Schleifen kommen nicht vor. Iterationen werden durch Funktionen wie *filter*, *reject*, *forall*, und *sort-by* ersetzt.

- Um so programmieren zu können, müssen die Funktionen „curryfiziert“ sein. Rein funktionale Sprachen machen dies automatisch. In Common Lisp haben wir es mit Hilfe der *curry*-Funktion erreicht. Lisp basiert zwar auf den Ideen der funktionalen Programmierung, Common Lisp ist aber eine Multiparadigmensprache.

- Die objektorientierte Lösung der Aufgabe in JavaScript (Folien) ist erheblich umfangreicher. Sicher könnte diese auch kompakter umgesetzt werden. Wichtig: Der funktionale Ansatz ist nicht grundsätzlich besser, das hängt von einer Reihe weiterer Kriterien ab. **Bestimmte Probleme können aber sehr elegant funktional gelöst werden.**

Programmdatei und Abgabe

Stellen Sie die Funktion *open-tasks* und allenfalls nötige Hilfsfunktionen aus Aufgabe 1 mit Kommentaren in einer Datei (Erweiterung: *.lisp*) zusammen.

Wichtig: Entfernen Sie den Quicklist-Aufruf zum Laden des JSON-Moduls *(ql:quickload "cl-json")* und alle Aufrufe von *read-json*, da dies auf dem Testserver nicht funktioniert. Zum Test wird mit einer vordefinierten Datenstruktur gearbeitet.

Geben Sie diese Datei bis zum nächsten PSPP ab:
https://radar.zhaw.ch/python/UploadAndCheck.html

Praktikum: `PS11`
Name und Kurznamen angeben und *Upload* wählen.

2. Beispiele in JavaScript *(fakultativ)*

(fakultative Übungsaufgabe)

In dieser Aufgabe werden einige Techniken, die im Unterricht anhand von Common Lisp gezeigt wurden, in JavaScript umgesetzt, welches ebenso wie Common Lisp zwar eine Multiparadigmensprache ist, das aber ebenfalls viele Aspekte der funktionalen Programmierung gut unterstützt. Es soll gezeigt werden, dass die gezeigten Techniken nicht auf Lisp beschränkt sind, sondern sich auch in anderen Sprachen nutzen lassen.

Im Code-Archiv zu diesem Praktikum finden Sie einige Beispiele mit JavaScript, die Sie direkt in einem aktuellen Browser testen können.

Die Ausgabe erfolgt direkt auf der Seite. Alternativ können Sie die Ausgabe durch Entfernen der Kommentarzeichen vor *log("to", "console")* auch auf die Konsole schicken. Eine weitere Möglichkeit ist, interaktiv in der Node.js-Konsole zu arbeiten.

Zunächst zur den bekannten Funktionen *reduce*, *map* und *filter*. Diese sind in JavaScript als Methoden von *Array.prototype* definiert:

```
> [0, 1, 2, 3, 4].reduce( (prev, curr) => prev + curr )
10
> [0, 1, 2, 3, 4].filter( (curr) => curr%2==0 )
0,2,4
> [0, 1, 2, 3, 4].map( (curr) => curr*curr )
0,1,4,9,16
```

Als Argumente dieser Methoden werden Funktionen übergeben. Weitere Informationen:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/prototype

Wir haben bereits gesehen, dass *reduce* flexibel genug ist, auch *filter* oder *map* zu ersetzen. Hier ist ein Beispiel:

```
> [0, 1, 2, 3, 4].reduce((prev,curr) => curr%2==0 ? prev.concat(curr) : prev, [])
0,2,4
```

Aufgabe: Versuchen Sie, auch das weiter oben angegebene Beispiel mit *map* durch *reduce* zu ersetzen, so dass das gleiche Ergebnis resultiert:

```
> [0, 1, 2, 3, 4].reduce(...)
0,1,4,9,16
```

Das sind Methoden von Objekten. Um „funktionaler“ damit arbeiten zu können, müssen wir die Methoden von den Objekten „befreien“. Mit Hilfe der Funktion *curry* (ebenfalls in der JavaScript-

Datei – wie die genau arbeitet, soll hier nicht interessieren) erstellen wir eine „curryfizierte“[2] Version von *reduce* und testen diese:

```
> var reduce = curry((fun,init,arr) => arr.reduce(fun, init));
> var sumlist = reduce(add)(0);
> sumlist([1,2,3,4,5])
15
```

Als Nächstes wird *map* mit Hilfe von *reduce* definiert:

```
> var map = (fn) => reduce((prev, curr) => prev.concat(fn(curr)), []);
```

Aufgabe: Können Sie nachvollziehen, wie *map* arbeitet? Geben Sie einen Aufruf an, mit dem wie oben alle Elemente einer Liste quadriert werden;

```
> map ...
0,1,4,9,16
```

Nun wird eine endrekursive Version der Fakultätsfunktion definiert. Der rekursive Aufruf wird aber nicht direkt ausgeführt, sondern mit *partial* (weiter unten in der Datei definiert) erst einmal nur vorbereitet:

```
function factorial (n, res) {
  res = res || 1;
  if (n < 2) return res;
  else return partial(factorial, n-1, n*res);
}
```

Die Funktion verhält sich speziell:

```
> factorial(4)
[Function]
> factorial(4)()
[Function]
> factorial(4)()()
[Function]
> factorial(4)()()()
24
```

[2] Kennt jemand ein eleganteres Wort?

Aufgabe: Ergänzen Sie die Funktion *trampoline*, so dass folgender Aufruf möglich ist:

```
> trampoline(factorial, 4)
24
```

Diese Technik kann verwendet werden, um den Stack bei endrekursiven Funktionen zu entlasten, wenn der Compiler oder die Laufzeitumgebung solche Funktionen nicht optimiert. ES6 sieht eine solche Optimierung zwar vor, dies ist aber noch nicht überall implementiert. Die meisten Common-Lisp-Implementierungen unterstützen übrigens eine solche *Tail Call Optimization* (Ausnahme: clisp).

Aufgabe: Testen Sie die Funktionen *iseven* bzw. *isodd* mit grösseren Zahlen. Können Sie einen Stack-Überlauf provozieren? Falls nicht, könnte die JavaScript-Umgebung des verwendeten Browsers bereits *Tail Call Optimization* implementiert haben. Ansonsten: Probieren Sie das Problem mit partieller Anwendung und der Funktion *trampoline* zu beheben.

3. Listen ohne Listen in JavaScript *(fakultativ)*

(fakultative Übungsaufgabe)

Im Unterricht wurde eine Möglichkeit erwähnt, in JavaScript Listen und natürliche Zahlen mit Hilfe von Funktionen zu definieren, ohne auf Datenstrukturen wie Arrays zurückzugreifen. Beschrieben ist dies in den Folien *funktional_teil3_listen_ohne_listen*.

Sie finden die nötigen Funktionen in der Datei *list-out-of-lambda.js* (Geringfügig angepasste Version von https://gist.github.com/sjl/5277681, leider ohne Kommentare).

Machen Sie ein paar Versuche damit. Wenn Sie über kein JavaScript in der Kommando-Shell verfügen, können Sie auch eine Online-Shell wie http://www.squarefree.com/shell/shell.html verwenden.

Aufgaben:

- Legen Sie eine Liste an.
- Schreiben Sie eine Funktion, die eine String-Repräsentation einer solchen Liste erzeugt. Angewendet auf `cons(1, cons(2, cons(3, nil)))` würde die Funktion zum Beispiel `"1 2 3 "` liefern.
- Testen Sie auch die Funktionen *filter* und *map*.
- Ergänzen Sie die noch fehlende Funktion *reduce*.
- Schreiben Sie auch eine Funktion, die eine String-Repräsentation einer Zahl erzeugt. Testen Sie damit auch die numerischen Funktionen wie *mul* und *pow.*

4. Wieder einmal: Fibonacci *(fakultativ)*

(fakultative Übungsaufgabe, Challenge...)

Wir haben schon verschiedene Möglichkeiten gesehen, eine Funktion zu implementieren, welche die n-te Fibonacci-Zahl bestimmt. Können Sie diese Funktion auch mit Hilfe von *reduce* und *range* schreiben?