PSPP-Praktikum 6
Common Lisp (Teil 1)

Vorbereitung

Für diese und die nächsten Praktikumslektionen benötigen Sie eine *Common-Lisp*-Installation. Verschiedene Möglichkeiten wurden im Unterricht vorgestellt. Für die Zwecke von PSPP genügt bereits eine Kommandozeilen-Installation von *Common Lisp* und ein Code-Editor. Dieser sollte allerdings wenigstens zusammengehörende Klammern anzeigen. Wie in diesem Beispiel der *VSCode* mit installierter Erweiterung *Lisp* (Yasuhiro Matsumoto):

```
1  (defun flaeche2 (laenge &optional breite)
2    (list (if breite 'rechteck 'quadrat)
3          'mit 'flaeche
4          (* laenge (or breite laenge))))
```

Hier der *vim* (via SSH auf dem dublin.zhaw.ch):

```
(defun flaeche2 (laenge &optional breite)
  (list (if breite 'rechteck 'quadrat)
        'mit 'flaeche
        (* laenge (or breite laenge))))
```

In der Datei *lisp_implementierungen* sind einige mögliche Lisp-Systeme zusammengestellt. Zunächst ist eine Kommandozeilen-Installation zusammen mit einem Code-Editor gut ausreichend.

Entwicklungsumgebungen wie *Clozure CL* (http://ccl.clozure.com, Mac) oder *Corman Lisp* (https://github.com/sharplispers/cormanlisp, Windows) enthalten neben einem Editor zahlreiche weitere Tools wie Debugger oder Inspector. Auch Editoren wie der *Emacs* eignen sich gut als Entwicklungsumgebung für Lisp. Allerdings lohnt sich eine vertiefte Einarbeitung in solche Werkzeuge für den kurzen Einstieg in die Lisp-Programmierung nicht.

Common Lisp verfügt bereits über eine grosse Bibliothek von Funktionen und Makros. Die Schwierigkeit beim Start mit Common Lisp ist vor allem, die passenden Funktionen zu finden. Eine gute Zusammenstellung bietet das auch im Unterricht erwähnte *„Common Lisp the Language, 2nd Edition“*. Wenn Sie etwa wissen, wie eine Funktion heissen könnte, kann auch der Index in diesem Online-Buch hilfreich sein.

Common Lisp the Language, 2nd Edition
https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

Common Lisp Quick Reference
http://clqr.boundp.org/index.html

Common Lisp HyperSpec
http://www.lispworks.com/documentation/HyperSpec/Front/index.htm

1. Erste Schritte mit Common Lisp

Die ersten Beispiele dienen dazu, sich mit dem Lisp-Interpreter vertraut zu machen.

a. Zahlen und Operatoren

Formulieren Sie den folgenden Ausdruck als Lisp-Ausdruck und testen Sie diesen in der Konsole. Braucht es hier die Punkt-vor-Strich-Regel oder andere Festlegungen zur Bindungsstärke von Operatoren?

```
10 * (4 + 3 + 3 + 1) – 20
```

Überlegen Sie, zu welchem Resultat die folgenden Lisp-Ausdrücke evaluieren, und testen Sie anschliessend, ob Ihre Vermutung stimmt:

```
42
pi
(/ 4 3)
(/ 12 3)
(/ 4 3.0)
(+ 0.5 1/2)
(* 35 1.16)
(sqrt 2)
(expt 12 32)
(floor (/ 5 2))
(zerop 25)
(zerop 0)
```

b. Symbole und Listen

Überlegen Sie auch für die folgenden Beispiele, welches Resultat ausgegeben wird, bevor Sie es probieren:

```
'zitrone
(quote zitrone)
(list 'apfel 'banane 'zitrone)
'(apfel banane zitrone)
(car '(apfel banane zitrone))
(cdr '(apfel banane zitrone))
(caaddr '((apfel banane) zitrone (dattel erdbeere)))
(length '((apfel banane) zitrone (dattel erdbeere)))
(append '(apfel banane zitrone) '(dattel erdbeere))
(cons 'zitrone '(apfel banane))
(reverse '(apfel banane zitrone))
```

c. car und cdr

Schreiben Sie Sequenzen von *car* und *cdr*, die das Symbol y aus diesen Listen herauspicken:

```
((w (x y)))
((w u) y z)
((w (x) y) u)
((((y))) w)
```

Anstatt *car* und *cdr* können Sie auch die Funktionen *first* und *rest* verwenden. Weiterhin gibt es die Funktionen *second, third, fourth*, ..., sowie *last* und *butlast*. Sehen Sie in der Dokumentation nach, wie die Funktion *nth* benutzt wird.

2. Funktionen und Listen

Listen sind die zentrale Datenstruktur in Lisp. Aus diesem Grund sollen nun ein paar Funktionen geschrieben werden, die Listen verarbeiten. Wir werden uns zwar erst noch mit funktionaler Programmierung beschäftigen, versuchen Sie aber bereits jetzt, einen möglichst funktionalen Stil zu verwenden, indem Sie die folgenden Punkte berücksichtigen:

- Listen sind rekursive Datenstrukturen: Eine Liste ist entweder leer (NIL) oder eine Liste, in die vorne ein Element eingefügt wird. Wir verarbeiten die Liste daher auch rekursiv.
- Es soll keine Zuweisungsoperation verwendet werden, also kein neuer Wert an ein Symbol gebunden werden (mit *set* o.ä.). Die verwendeten Variablen sind Parameter der Funktion. Sie werden einmalig beim Funktionsaufruf gebunden und dann nicht mehr verändert. Das heisst: Wir betrachten Variablen als unveränderlich (final).

a. list-double

Zunächst eine Funktion, die einer Übung aus dem Unterricht sehr ähnlich ist. Schreiben Sie eine Funktion *list-double*, die alle Elemente der als Parameter übergebenen Liste verdoppelt:

```
> (list-double '(1 2 3 4))
(2 4 6 8)
```

Für die rekursive Lösung benötigen Sie eine Abfrage (*if* oder *cond*), ob die Liste leer ist (*null*). Mit *cons* wird ein Element vorne in die Liste eingefügt:

```
> (cons 9 '(a b c d))
(9 a b c d)
```

b. sign

Schreiben Sie die Signum-Funktion *sign*, die angibt, ob ihr Argument positiv (Ergebnis: 1), negativ (-1) oder null (0) ist. **Achtung:** Präfix-Notation (> n 0) und nicht (n > 0).

```
> (sign -62)
-1
```

c. list-sign

Verwenden Sie das Ergebnis der letzten Teilaufgabe, um eine Funktion *list-sign* zu schreiben, die *sign* auf alle Elemente einer Liste anwendet:

```
> (list-sign '(5 2 -3 -1 0 3 -2))
(1 1 -1 -1 0 1 -1)
```

3. Funktionale Abstraktion: Mapping

Die bisher geschriebenen Funktionen *list-sqr* (Unterricht), *list-double* und *list-sign* haben etwas gemeinsam: Sie führen eine Funktion auf allen Elementen einer Liste aus und geben eine neue Liste mit den Ergebnissen zurück. Es zeigt sich, dass solche Operationen auf Listen von irgendwelchen Dingen und die Rückgabe der Ergebnisse in einer neuen Liste bei vielen Programmierproblemen immer wieder auftauchen.

Anstatt immer wieder ähnliche Lösungen für dieses Problem zu programmieren, kann man versuchen, den gemeinsamen Teil zu extrahieren und separat zu lösen. In diesem Fall ist das sehr einfach möglich, wenn die verwendete Programmiersprache Funktionen höherer Ordnung zulässt: Funktionen, denen Funktionen als Parameter übergeben werden können.

Fassen wir den gemeinsamen Aspekt von *list-sqr*, *list-double* und *list-sign* in einer Funktion *map-list* zusammen, deren erstes Argument eine Funktion und deren zweites Argument die Liste ist:

```
(defun map-list (f lst)
  (if (null lst) nil
      ;; else
      (cons (funcall f (car lst))
            (map-list f (cdr lst)))))
```

Wo ist der rekursive Aufruf? Wozu dient *funcall*? Schlagen Sie *funcall* in einer Common Lisp Dokumentation nach.

Statt *list-sign* können wir nun die allgemeinere Funktion aufrufen und die Funktion *sign* als Argument übergeben. Mit *#'sign* ist gemeint: Die Funktion, die an das Symbol *sign* gebunden ist. Es ist eine Abkürzung für *(function sign)*.

```
> (map-list #'sign '(5 2 -3 -1 0 3 -2))
(1 1 -1 -1 0 1 -1)
```

Diese Abstraktion wird häufig benötigt, so dass sie in vielen Sprachen bereits vordefiniert ist, meist unter dem Namen *map*. In Common Lisp heisst die Funktion *mapcar*:

```
> (mapcar #'sign '(5 2 -3 -1 0 3 -2))
(1 1 -1 -1 0 1 -1)
```

Aufgabe:

Schreiben Sie die Funktionen *list-double*, *list-sqr* (Beispiel aus dem Unterricht) und *list-sign* neu mit Hilfe von *map-list*. Vermutlich benötigen Sie noch Hilfsfunktionen *sqr* und *dbl* (für *double*, manche Lisp-Implementierungen erlauben es nicht, eine Funktion *double* zu nennen).

4. Listen reduzieren

Schreiben Sie eine Funktion *list-sum*, die die Summe der Elemente einer Liste bestimmt. Schreiben Sie ausserdem eine Funktion *list-mult*, die das Produkt der Elemente einer Liste bestimmt.

```
> (list-sum '(10 20 7))
37

> (list-mult '(10 20 7))
1400
```

Wichtig ist, dass Sie auch diese Aufgabe wieder mit einer rekursiven Funktion lösen. Es wird keine Variablenzuweisung benötigt. Alle Zustandsänderungen geschehen durch Funktionsaufrufe mit neuen Argumentwerten.

Es geht hier um den Aufbau der Funktion. Daher ignorieren wir, dass derselbe Effekt mit `(apply #'+ '(10 20 7))` bzw. `(apply #'* '(10 20 7))` erreicht werden könnte.

An welchen Stellen unterscheiden sich die beiden Funktionen *list-sum* und *list-mult*?

Probieren wir, noch eine weitere ähnlich aufgebaute Funktion zu schreiben: Schreiben Sie eine Funktion *all-true*, die *true* (T) liefert, wenn alle Elemente der Liste *true* (nicht NIL) sind, sonst NIL. Eine Und-Verknüpfung ist mit *and* möglich.

```
> (all-true '())
T

> (all-true '(34 hallo (7)))
T

> (all-true '(34 hallo ()))
NIL
```

Wir werden diese Funktionen in Zukunft ebenfalls noch verallgemeinern.

Programmdatei und Abgabe

Stellen Sie die implementierten Funktionen aus den Aufgaben 3 und 4 mit Kommentaren in einer Datei (Erweiterung: *.lisp*) zusammen. Nennen Sie die Funktionen wie angegeben.

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

Praktikum: `PS8` (im normalen Semesterablauf ist dies das Praktikum 8)
Name und Kurznamen angeben und *Upload* wählen.

5. Debugging *(fakultativ)*

Eine einfache Möglichkeit, die Aufrufe einer Funktion zu überwachen, bietet das Makro *trace*. Probieren Sie es aus:

```
> (trace list-sum)
(LIST-SUM)
> (list-sum '(1 2 3))
...
> (untrace)
(LIST-SUM)
```

6. Verallgemeinertes Mapping *(fakultativ)*

Das ist etwas schwieriger. Schreiben Sie eine neue Funktion *map-list-rec*, die ähnlich wie *map-list* eine Funktion auf alle Listenelemente anwendet. Die Liste soll nun aber auch verschachtelt sein können und das Ergebnis soll die gleiche Struktur wie die Ausgangsliste aufweisen:

```
> (map-list-rec #'sign '(4 -6 (-1 0 ((((9)))) 12) -9))
(1 -1 (-1 0 ((((1)))) 1) -1)
```

Tipp:

```
> (listp '(1 2 3))
T
> (listp 1)
NIL
```