o++o auf 7 Seiten

(Stand 17.04.2018)

Klaus Benecke

o++o, genauer ottoPS (ottoProgrammierSprache), ist ein verbesserter Taschenrechner, der auch Anfragen an strukturierte TABellen und DokuMENTe (Tabmente) erlaubt. o++o ist damit auch geeignet Datenbankanfragen zu ermöglichen und später auch Basis einer Suchmaschine zu sein.
o++o wurde so entworfen, dass die Programme möglichst kurz, mit einheitlicher Syntax und übersichtlich sind. Die folgenden Programme sollte man unter ottoPS.de ausprobieren, da man Programmieren kaum ohne eigenes Testen erlernen kann.

Programm 1: Berechne die vierte Wurzel aus 2.

```
2 sqrt sqrt
```

Programm 2: Berechne den sinus von 1.

```
1 sin
```

Programm 3: Wie viele 10 stellige Dualzahlen gibt es?

```
2 hoch 10
```

Programm 4: Berechne die Kantenlänge eines Würfels des Volumens 2 (die dritte Wurzel aus 2).

```
2 hoch (1:3)
```

oder

```
2
hoch 1:3
```

Programm 5: Bilde einen Durchschnitt.

```
1 3 2 1 3 4
++:
```

Um Schreibaufwand zu sparen, kann man bei einer Liste von Werten (hier die Zahlen der ersten Zeile) auf jegliches sichtbare Trennzeichen und jegliche Klammerung verzichten. Auf diese Liste wird jetzt durch die nächste Zeile die Operation der Durchschnittsbildung (++:) angewandt. Man kann anstelle des Durchschnittes auch eine Summe (`++`), das Produkt (`**`), die Anzahl (`++1`), das Maximum (`max`), den Sinus (`sin`), ln, … bilden. Da `sin` nur einen Inputwert benötigt, ergibt sich bei Anwendung der Sinusoperation anstelle der ++:-Operation eine Liste von output-Werten. Auf diese Liste könnte man dann wieder ++: anwenden und erhält dann lediglich eine Zahl. Obige Schreibweise ist vielleicht etwas gewöhnungsbedürftig, aber sie ist kompakter als die alte Schreibweise

```
++:([1 3 2 1 3 4])
```

Insbesondere, wenn mehrere Operationen hintereinander angewandt werden, spart man sich hierbei viele Klammern und damit Fehlerursachen.

Programm 6: Berechne den Wert des Terms `sqrt(abs(sin(7.1))+abs(cos(8.1))).`

```
7.1 sin abs
+ 8.1 cos abs
sqrt
```

Programm 7: Ziehe von der ersten Zahl 5 weitere Zahlen ab.

```
444.8 77.9 45.6 33.5 23.74 25.88 --
```

In o++o werden mehrere Operationen nach dem sehr alten „WaldPrinzip“ geschrieben. Für den Baum gibt es ein Wort und BaumBaum heißt Wald. Obiges `- -` ersetzt daher 5 Minuszeichen

Programm 8: Multipliziere jede Dezimalzahl einer Liste mit einer weiteren Zahl.

```
2.40 2.70 7.90
* 1.19
```

Hierbei wird jede Zahl der Inputliste mit 1.19 multipliziert. Sind die gegebenen Zahlen Nettopreise, so stellt das Ergebnis die zugehörigen Bruttopreise dar, sind die gegebenen Zahlen dagegen Beträge einer Währung und ist 1.19 der Umrechnungskurs, so stellt das Ergebnis die Werte in der anderen Währung dar, … . Wir sehen, dass Dezimalzahlen nicht mit dem Komma sondern mit dem Punkt dargestellt werden.

Programm 9: Bilde die Summe von vielen positiven und vielen negativen Zahlen ohne viele Minuszeichen und Klammern zu benutzen.

```
4 5 3 2 1  8 9 ++
- 7 6 5 4 3 2 1 ++
```

Programm 10: Berechne die Umfänge mehrerer Kreise, wobei die Radien gegeben sind.

```
4 5 6 2 3.7 9.77
*pi*2
```

Programm 11: Berechne die Umfänge und Flächen mehrerer Kreise, wobei die Radien gegeben sind.

```
[R! 4 5 6 2 3.7 9.77]
UMFANG:=R*pi*2
FLAECHE:=R*R*pi
```

Programm 12: Berechne die Flächen und Umfänge mehrerer Rechtecke.

```
<TAB:
A,   B l
1.23 5.67
7.65 4.32
9.87 6.54
:TAB>
UMFANG:=A+B*2
FLAECHE:=A*B
```

Die TAB-Klammern (<TAB:, :TAB>) sind nur im Programmteil des Systems erforderlich. Bei kleinen flachen Inputtabellen kann auch eine einzeilige Schreibweise benutzt werden.

```
[A,B! 1.23 5.67 7.65 4.32 9.87 6.54]
UMFANG:=A+B*2
FLAECHE:=A*B
```

Programm 13: Bestimme den Gesamtpreis einer einfachen Rechnung.

```
<TAB:
ARTIKEL,  PREIS l
Bier      0.61
Brause    0.23
Schnitzel 2.40
:TAB>
++
```

Hier wird einfach die Summe über alle Zahlen der gegebenen Tabelle (Liste (l) von Paaren) gebildet. Die Artikelwerte sind Wörter und haben damit keinen Einfluss auf das Ergebnis. Ersetzt man ++ durch

```
*1.10
```

, so entsteht wieder eine Tabelle mit 2 Spalten und drei Zeilen (Sätzen,Tupeln), wobei jede Zahl das Trinkgeld einschließt. Anschließend kann man wieder

```
++
```

anfügen, um auf die Gesamtsumme zu kommen.

Programm 14: Bestimme den Gesamtpreis einer etwas komplizierteren Rechnung, die durch eine einfache Tabelle gegeben ist.

```
<TAB:
ARTIKEL,  PREIS, ANZAHL l
Bier      0.61   7
Brause    0.23   3
Schnitzel 2.40   4
:TAB>
POSPREIS:=PREIS*ANZAHL
++ POSPREIS
```

Durch die Zuweisung wird die gegebene Tabelle um eine neue Spalte mit dem Spaltennamen POSPREIS erweitert, wobei der Preis dreimal mit der zugehörigen Anzahl multipliziert wird. Um die Gesamtsumme des Kneipenbesuchs zu ermitteln, muss man der ++-Operation noch einen zweiten Inputwert geben. Ansonsten würde die Gesamtsumme aller 9 Zahlen der Zwischentabelle gebildet werden. Beide Programmzeilen können auch einfach durch

```
++ PREIS*ANZAHL
```

ersetzt werden. Der erste Inputwert einer Operation, die am Anfang einer Programmzeile steht, ist immer das Ergebnis der vorangehenden Programmzeile.

Das Ergibtzeichen `:=` der Zuweisung ist vom Gleichheitszeichen `=` zu unterscheiden. Das Gleichheitszeichen wie auch `<, >, <=`, in, … wird zur Formulierung von Bedingungen benötigt. Bedingungen dienen der Selektion. Fügt man beispielsweise eine Bedingung

```
avec ARTIKEL=Bier
```

oder einfach

```
avec Bier
```

ein, so ergibt sich im Endergebnis nur der Gesamtpreis für die 7 Bier. Will man den Preis für den Rest berechnen, so kann man stattdessen

```
sans ARTIKEL=Bier
```

oder einfach

```
sans Bier
```

einfügen. Spaltennamen (Metadaten) müssen stets groß geschrieben werden. Die Schlüsselworte (gib, sans, avec, …) müssen stets klein geschrieben werden. Schreibt man ein Wort der Primärdaten immer mit Groß- **und** Kleinbuchstaben, so wird das Programm leichter lesbar.

Programm 15: Bestimme den Gesamtpreis eines längeren Kneipenbesuchs.

```
<TAB:
ARTIKEL,  PREIS,ANZAHLl l
Bier      0.61  7 6 5
                3
Brause    0.23  3 4
Schnitzel 2.40  4 3 2
:TAB>
POSPREIS:=PREIS*ANZAHL
++ POSPREIS
```

Hierbei haben wir es mit einer strukturierten Tabelle zu tun, die für jede Position mehrere Bestellungen enthält. Wir müssen die `ANZAHL`-Einträge nicht untereinander schreiben, damit man den Kneipenbesuch kompakt darlegen kann. Betrachtet man dagegen nicht das Endergebnis sondern nur das Ergebnis der Anwendung der Zuweisung, so müssen die `ANZAHL`- und `POSPREIS`-Werte untereinander stehen, da die entstehende strukturierte Tabelle ansonsten zu unübersichtlich werden würde:

```
ARTIKEL,  PREIS, (ANZAHL, POSPREIS l)l
Bier      0.61    7       4.27
                  6       3.66
```

```
                  5       3.05
                  3       1.83
Brause    0.23    3       0.69
                  4       0.92
Schnitzel 2.4     4       9.6
                  3       7.2
                  2       4.8
```

Will man erst die ANZAHL-Werte addieren und dann multiplizieren,

```
POSPREIS:= ANZAHLl ++ *PREIS
```

, so ergibt sich wieder eine kompaktere Zwischentabelle:

```
ARTIKEL,  PREIS, POSPREIS,ANZAHLl m
Bier      0.61   12.81    7 6 5 3
Brause    0.23    1.61    3 4
Schnitzel 2.4    21.6     4 3 2
```

Beide Programmzeilen können auch wieder durch ++ `PREIS*ANZAHL` ersetzt werden.

Man kann die betrachteten Tabellen jeweils unter einem Namen beispielsweise mit der Endung `.tab` abspeichern und dann diese wieder aufrufen. Dadurch können Tabellen oder Dokumente in mehreren Programmen genutzt werden. Programm 15 könnte dann so aussehen:

```
kneipenbesuch.tab
++ PREIS*ANZAHL
```

Dieses Beispiel lässt sich auf viele Anwendungen übertragen. Beim Kegeln könnte man ebenfalls eine Tabelle `kegeln.tab` mit Wiederholgruppe aufbauen: `NAME,WURFl m. m` kürzt Menge und `l` Liste ab. Für jeden Kegler kann man die Gesamtsumme dann durch eine Zuweisung ermitteln und anschließend die Daten abfallend sortieren. Will man im Ergebnis nur noch bestimmte Spalten und will man nach diesen noch sortieren, so benötigt man eine gib-Klausel.

Programm 16: Bestimme für jede Person das Gesamtergebnis des Kegelns und sortiere die Daten danach.

```
kegeln.tab
GESAMT:=WURFl ++
gib GESAMT,NAME mv
```

Es geht auch noch etwas kürzer:

Programm 17: Siehe Programm 16.

```
kegeln.tab
gib GESAMT,NAME mv
    GESAMT! ++ WURF
```

Hierbei ergibt sich der Bezug der Aggregation (hier ++) aus dem Kopf der gewünschten Tabelle. GESAMT ist eine Aggregation pro NAME. Bei Mengen (m, mv) wird stets nach den zuerst angegebenen Spaltennamen sortiert.

Wir nehmen an, dass jeder Fluss in `fluesse.tab` eine `LAENGE`-Spalte hat. Dann findet man alle Flüsse, die länger als 500 km sind durch folgende Anfrage:

Programm 18: Selektiere in einer vorgegebenen Tabelle.

```
fluesse.tab
avec LAENGE > 500
```

„avec“ ist französisch und heißt „mit“. Analog benutzen wir für ohne „sans“.

Jetzt sollen die Flüsse noch nach der Länge sortiert werden und mit Nebenflüssen ausgegeben werden:

Programm 19: Selektion mit anschließender Sortierung.

```
aus  fluesse.tab
avec LAENGE>500
```

```
gib  LAENGE,FLUSS,NEBENFLUSSm m
```
Will man die längsten Flüsse zuerst, muss man lediglich, das äußere `m` durch `mv` ersetzen.
Unsere Datei fluesse.tab enthält für jeden Fluss auch eine Wiederholgruppe
`LAND,BUNDESLANDm m`, die angibt durch welche Länder und Bundesländer der Fluss fließt. In fluesse.tab ist also der FLUSS dem Bundesland übergeordnet. Will man das umkehren, kann das wieder einfach durch Angabe des Kopfes der gewünschten Tabelle realisiert werden.
Programm 20: Umstrukturierung der Fluesse-Datei.
```
aus fluesse.tab
gib BUNDESLAND,FLUSSm m
```
Durch diese freie Umstrukturierung werden zu jedem Bundesland alle Flüsse, die durch dieses fließen, gesammelt. Die Bundesländer und die Flüsse innerhalb der Bundesländer werden sortiert. Ein Fluss kann hier in mehreren Bundesländern vorkommen. Jedes Bundesland erscheint nur einmal, da wir als äußere Kollektion eine Menge gewählt haben. Den Beginn eines Programms kann man auch mit `aus` kennzeichnen.
Neben den einfachen Berechnungen (ext) (:=), durch die eine Tabelle um eine neue Spalte erweitert wird, gibt es noch die sogenannten rekursiven Erweiterungen (rec). Hierbei sind zwei Formeln gegeben. Die erste Formel ist für das erste Element der Spalte bestimmt und die zweite für die restlichen Elemente. Die Formel für die restlichen Elemente kann sich auf den Vorgänger der Spalte beziehen. Das ist im folgenden Programm `BETRAG pred`.
Programm 21: Was wird aus 100 Euro nach 20 Jahren bei 9 % Wachstum (Zinsen).
```
0 bis 20 tags BETRAG
rec BETRAG:= first 100. next BETRAG pred*1.09 at JAHR
```
Programm 22: Wandle eine Dualzahl (hier 10111=23) in eine Dezimalzahl um.
```
[BIT! 1 0 1 1 1]
rec DEZI:= first BIT next DEZI pred*2+BIT at BIT
```
Ist nicht die ganze Entwicklung sondern nur das letzte Element der Liste gewünscht, so kann man noch die Bedingung BIT `pos- =1` anhängen. Ist das letzte Bit auch nicht gewünscht, so sollte `gib DEZI` folgen.
Programm 23. Wandle eine Dezimalzahl (hier 23) in eine Dualzahl um.
```
rec (DURCH,REST)l:= while DURCH>0 | REST >0
                    first 23 divrest 2
                    next  DURCH pred divrest 2
gib RESTlv
```
Hierbei werden mit einer rekursiven Erweiterung gleich 2 neue Spalten eingeführt, DURCH und REST. `div` ist hier die ganzzahlige Division mit dem ganzzahligen Rest (ein Paar von Zahlen).
Programm 24: Berechne den gewichteten Durchschnitt für 3 Schüler und den Gesamtdurchschnitt.
```
<TAB:
NAME,  KLAl, NOTEl l
Ernst  1 2   1 2 3 1 3 1 1
Clara  1 1   3
Sophia 1 3   1
:TAB>
DUR:=KLAl ++: *0.6 +(NOTEl ++: *0.4)
GESAMT:=DURl ++: ; rnd 2
```
Durch `rnd` wird das Ergebnis gerundet. Das Semikolon trennt eine Zeile.
Programm 25: Bilde eine Vereinigung (alle StudentenIDs, die in einer der Tabellen enthalten sind).
```
aus examen.tab, projekte.tab
gib STIDm
```

Hierbei seien examen.tab und projekte.tab vom Typ `STID,KURS,NOTE m` bzw. `STID,PROJ,STUNDEN m`.

Programm 26: Bilde einen Durchschnitt (alle StudentenIDs, die in beiden Tabellen enthalten sind).

```
aus examen.tab, projekte.tab
gib+ STIDm
```

Mit gib+ können auch joins ausgedrückt werden und damit auch Anfragen an ganze Datenbanken formuliert werden. Die folgenden Anfragen beziehen sich auf eine Datenbank `uni.tab`, deren zweite Tabelle unstrukturiert ist. Die Anfragen müssen nicht oder nur schwach modifiziert werden, wenn alle Tabellen von uni.tab unstrukturiert (relational) sind.

```
<STUDENTEN:
STID, NAME,  ORT?,    STIP, FAK,  (KURS,       NOTE m), (PROJ, STUNDEN m) m
1234  Ernst  Oehna    500   Mathe  Algebra     1        Fritz  4
                                  Geschichte  1        Otto   2
                                  Logik       2
1245  Sophia Berlin   400   Inf    Algebra     3        Mao    5
                                  Datenbanken 1        Ming   4
                                  Otto        1        Otto   6
3456  Clara  Oehna    450   Inf    Datenbanken 1
                                  OCaml       2
4567  Ulrike          400   Kunst                       Monet 10
5678  Kaethe Gerwisch   0   Kunst  Apel        1        Monet 20
                                  Repin       1
:STUDENTEN>
<FAKS:
FAK,  DEKAN,  BUDGET, STUDKAP m
Inf   Reichel 10000   500
Kunst Sitte    2000   600
Mathe Dassow   1000   200
Philo Hegel    1000    10
:FAKS>
```

Die Tabelle `STUDENTEN` ist strukturiert, da zu jedem Satz (strukturiertem Tupel) (Studenten) 7 Komponenten gespeichert werden und die letzten beiden Komponenten selbst wieder kleine Tabellen sind. Die vorletzte Komponente von Clara enthält beispielsweise zwei Prüfungsergebnisse und die letzte die leere Menge von Projekten.

Programm 27: Berechne das Gesamtbudget der Uni.

```
aus uni.tab
++  BUDGET
```

Programm 28: Berechne das Gesamtbudget und das Durchschnittsbudget der Uni.

```
aus uni.tab
gib SUMBUD,DURBUD  SUMBUD! ++ BUDGET
                   DURBUD! ++:BUDGET
```

Programm 29: Wie viele Fakultäten hat die Uni.

```
uni.tab
gib FAKS
++1
```

Programm 30: Gib zu jedem Kurs die zugehörigen Studenten mit Note.

```
aus uni.tab
gib KURS,(NAME,NOTE b)m
```

Programm 31: Gib alle Sätze (Tupel) die 1234 enthalten.

```
aus  uni.tab
avec 1234
```

Programm 32: Selektiere in allen Tabellen, die den Studentenidentifikator enthalten, nach dem Studenten mit der Nummer 1234.

```
uni.tab
avec STID=1234
```

Programm 22 unterscheidet sich nur bzgl. der FAKS-Tabelle von Programm 21. Im Ergebnis von Programm 21 ist diese leer und in Programm 22 bleibt sie vollständig erhalten. In anderen Anwendungen könnte man statt des Studentenidentifikators (STID) auch Artikelnummern oder Teilenummern, … wählen.

Programm 33: Gib zum Studenten mit der Nummer 1234 den Dekan und seine Examen.

```
aus  uni.tab
avec STID=1234
gib+ NAME,DEKAN,(KURS,NOTE m)m
```

Hier wird ein „Join“ ohne Joinbedingung realisiert. Würde man anstelle von `gib+ gib` verwenden, so würde die Ergebnismenge leer bleiben, da keine Verbindung zwischen NAME und DEKAN gegeben ist.

Programm 34: Berechne die Anzahl der Einsen für die einzelnen Fakultäten.

```
aus  uni.tab
avec NOTE=1
gib  FAK,ANZ m  ANZ! cnt NOTE
```

Programm 35: Teste das folgende Programm.

```
aus uni.tab
avec DEKAN in [Reichel Dassow]
gib FAK,DEKAN,(NAME,STIP m)m
```

Programm 36: Gib zu den Fakultäten von Dassow und Reichel alle zugehörigen Studenten.

```
uni.tab
avec DEKAN in [Reichel Dassow]
gib+ FAK,DEKAN,(NAME,STIP m)m # besser ?
```

Programm 37: Gesucht sind alle Fakultäten, die Studenten mit der besten und der schlechtesten (hier im Beispiel ist es die 3) Note haben.

```
aus  uni.tab
avec {1 3} in NOTEm
gib+ FAKS
```

Programm 38: Gesucht sind alle Studenten, die genau 2 Einsen haben.

```
aus  uni.tab
avec NOTEl = [1 1]
gib+ STUDENTEN
```

Ersetzt man hierbei gib+ durch gib, so erhält man das gleiche Ergebnis.

Programm 39: Gesucht sind alle Daten von Ernst (einschließlich seiner Fakultätsdaten).

```
aus  uni.tab
avec NAME=Ernst
gib+ UNI
```

Das Programm muss nicht verändert werden, wenn alle Tabellen von uni.tab in erster Normalform (unstrukturiert) sind.

Programm 40: Eine Flasche mit Kork kostet eine Mark und zehn. Die Flasche ist eine Mark teurer als der Korken. Wie viel kostet der Korken?

```
0 bis 110 tags FLASCHE
ext  KORK := FLASCHE - 100
avec KORK+FLASCHE = 110
```

Programm 41: Schreibe 1000 mal ich liebe dich: `1000 mal “Ich liebe Dich!“`