# o++o on 8 pages

(18.07.2018)

Klaus Benecke

o++o, more precisely ottoPS (ottoProgrammingScript), is an end-user computer language, an improved calculator for mass data, that also allows queries to structured TABles and docuMENTs (tabments). o++o is therefore also suitable to allow database and other queries. o++o was designed so that the programs are as short as possible, with uniform and clear syntax. The following programs should be tried out under ottoPS.de, since you can hardly learn programming without testing yourself.

**Program 1:** Calculate the fourth root of 2.

```
2 sqrt sqrt
```

**Program 2:** Compute the sine of 1.

```
1 sin
```

**Program 3:** How many 10-digit binary numbers are there?

```
2 hoch 10
```

“hoch” is German and means high (to the power of).

**Program 4:** Calculate the edge length of a cube of volume 2 (the third root of 2).

```
2 hoch  0.3333
```

or more precise

```
2
hoch 1:3
```

or

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

**Program 5:** Make an average.

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

To save on writing costs, you can write a list of values without any visible delimiters and brackets. Then the averaging (++ :) operation follows. Instead of the average one can also form a sum (++), the product (**), the number (++1 = count), the maximum (max), the sine (sin), ln, … . Since only one input value is required, using the sine operation, this would result in a list of output values.

You could then apply ++: again to this list and then only receive one number. The above notation is more compact than the old spelling avg ([1 3 2 1 3 4])

In particular, if several operations are applied one behind the other, this saves many parentheses and causes of errors.

**Program 6**: Compute the value of the term “`sqrt(abs(sin(7.1))+abs(cos(8.1)))`”.

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

**Program 7:** Subtract 5 more numbers from the first number.

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

In o++o several operations are written according to the very old "forest principle". There is a word for the tree and wood means treetree. The above - - replaces 5 minus signs.

**Program 8:** Multiply each decimal number of a list by another number.

```
2.40 2.70 7.90 * 1.19
```

Each number in the input list is multiplied by 1.19. If the given numbers are net prices, the result is the associated gross prices, but if the given numbers are amounts of one currency and 1.19 is the conversion rate, the result is the values in the other currency, … .

**Program 9**: Make the sum of many positive and many negative numbers without using many minus signs and parentheses.

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

**Program 10:** Calculate the perimeters of several circles, given the radii.

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

**Program 11:** Compute the perimeters and areas of several circles, given the radii.

```
[R! 4 5 6 2 3.7 9.77]
PERIMETER:=R*pi*2
AREA:=R*R*pi
```

**Program 12:** Compute the areas and perimeters of few rectangles.

```
<TAB!
A,    B l
1.23  5.67
7.65  4.32
9.87  6.54
!TAB>
PERIMETER:=A+B*2
AREA:=A*B
```

The TAB-brackets (<TAB!, !TAB>) are required only in programs.
For flat given tables a second notation can be used.

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

**Program 13:** Determine the total price of a simple invoice.

```
<TAB!
ARTICLE,  PRICE l
Beer      0.61
Brause    0.23
Schnitzel 2.40
!TAB>
++
```

Here simply the sum over all numbers of the given table (list (l) of pairs) is formed. The article values are words and thus have no influence on the result. If you replace ++ with

```
* 1.10
```

then a table with 2 columns and 3 rows (records, tuples) is created again, where each number includes the tip. Then you can add again

```
++
```

to get the grand total.

**Program 14:** Determine the total price of a slightly more complicated bill, given by a simple table.

```
<TAB!
ARTICLE,  PRICE, NUMBER l
Beer      0.61   7
Brause    0.23   3
Schnitzel 2.40   4
!TAB>
POSPRICE:=PRICE*NUMBER
++ POSPRICE
```

The assignment extends the given table by a new column with the column name POSPRICE, multiplying the price three times by the corresponding number. To determine the total amount of the pub visit, you have to give the ++ operation a second input value. Otherwise, the total of all 9 numbers of the intermediate table would be formed. Both program lines can be easily replaced by

```
++ PRICE * NUMBER
```

. The first input value of an operation that is at the beginning of a program line is always the result of the previous program line.

The result symbol `:=` of the assignment is to be distinguished from the equal sign `=`. The equals sign as well as `<,>, <=, in,` ... is needed to formulate conditions. Conditions are for selection. For example, add a condition

```
avec ARTICLE = Beer
```

or simply

```
avec Beer
```

, then the above program 14 results in the end result, only in the total price for the 7 beers. If you want to calculate the price for the rest, you can instead

```
sans ARTICLE = Beer
```

or simply

```
sans Beer
```

insert. Column names (meta data) must always be capitalized. The keywords (gib=(give me), sans =(without), avec = (with), ...) must always be written in lower case. If you write a word of the primary data always with uppercase and lowercase letters, the program becomes easier to read. avec and sans are French. It means with, respectively without.

**Program 15:** Determine the total price of a longer pub visit.

```
<TAB!
ARTICLE,  PRICE, NUMBERl l
Beer      0.61   7 6 5
                 3
Brause    0.23   3 4
Schnitzel 2.40   4 3 2
!TAB>
POSPRICE:=PRICE*NUMBER
++ POSPRICE
```

Here we are dealing with a structured table that contains multiple orders for each item. We do not have to write the NUMBER entries vertically so that you can present the pub visit compact. On the other hand, if you do not look at the final result but only at the result of the application of the assignment, then the (NUMBER, POSPRICE)- values must be organized vertically because otherwise the resulting structured table would become too confusing:

```
ARTICLE,  PRICE, (NUMBER, POSPRICE l)l
Beer      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
```

If you want to first add the NUMBER values and then multiply,

```
POSPRICE:= NUMBERl ++ * PRICE
```

, this results in a more compact intermediate table:

```
ARTICLE,  PREIS, POSPRICE,NUMBERl l
Beer      0.61   12.81    7 6 5 3
Brause    0.23    1.61    3 4
Schnitzel 2.4    21.6     4 3 2
```

Both program lines can also be replaced by

```
++ PRICE * NUMBER
```

. You can save the tables under a name, for example, with the extension `.tab` and then call them again. This allows tables or documents to be used in several programs.

**Program 15** could look like this:

```
pubvisit.tab
++ PRICE * NUMBER
```

This example can be applied to many applications. When bowling you could also build a table `bowling.tab` with repeating group: NAME, THROWl m. m truncates set and l list. For each bowler you can then determine the total by an assignment and then sort the data downhill. If you only want certain columns in the result and you still want to sort them according to these, you need a gib-clause.

**Program 16:** Determine the overall result of bowling for each person and then sort the data.

```
bowling.tab
TOTAL:=THROWl ++
gib TOTAL,NAME m-
```

It is also a little shorter:

**Program 17:** See program 16.

```
bowling.tab
gib TOTAL,NAME m-
    TOTAL! ++ THROW
```

Here the kind of the aggregation (++) results from the head of the desired table. TOTAL is one aggregation per NAME. For sets (m, m-), they are always sorted according to the column names specified first.

We assume that every river in `rivers.tab` has a LENGTH column. Then you can find all rivers that are longer than 500 km by the following query:

**Program 18:** Select in a given table.

```
rivers.tab
avec LENGTH>500
```

Now the rivers are still to be sorted according to the length and are issued only with tributaries:

**Program 19:** Selection with subsequent sorting.

```
rivers.tab
avec LENGTH > 500
gib LENGTH, RIVER, TRIBUTARYm m
```

If you want the longest rivers first, you just have to replace the outer m by m-.

Our file rivers.tab also contains a repeating group for each river `COUNTRY,BUNDESLANDm m`, which indicates by which countries and federal states the river flows. In rivers.tab, therefore, the RIVER is superior to the federal state. If you want to reverse this, it can be realized again simply by specifying the header of the desired table.

**Program 20:** Restructuring of the rivers-file.

```
aus rivers.tab
gib BUNDESLAND,RIVERm m
```

Through this free reorganization, all rivers that flow through it are collected for each federal state. The federal states and the rivers within the federal states are sorted. A river can occur here in several

states. Each federal state appears only once, because we have chosen a set as outer collection. The beginning of a program can also be marked by `aus`.
In addition to the simple calculations (`ext`) (: =), which extend a table by a new column, there are the so-called recursive extensions (rec). Here are two formulas given. The first formula is for the first element of the collection and the second for the remaining elements. The formula for the remaining elements can refer to the predecessor of the new column. This is in the following program `AMOUNT pred`.

**Program 21:** What will become of 100 euros after 20 years at 9% growth (interest).

```
[YEAR! 0 .. 20]
rec AMOUNT:= first 100.
             next AMOUNT pred*1.09 at YEAR
```

**Program 22:** Convert a binary number (here 10111 = 23) to a decimal number.

```
[BIT! 1 0 1 1 1]
rec DECI:= first BIT next DECI pred*2+BIT at BIT
```

If not the whole development but only the last element of the list is desired, then one can add the condition `avec BIT pos- = 1`. If the last bit is not desired, you should follow `gib DECI`.

**Program 23.** Convert a decimal number (here 23) to a binary number.

```
rec (THROUGH,REST)l:= while THROUGH>0 | REST >0
                      first 23 divrest 2
                      next  THROUGH pred divrest 2
gib RESTl-
```

With a recursive extension, 2 new columns are introduced, THROUGH and REST. divrest is here the integer division with the integer remainder (a pair of numbers).

**Program 24:** Calculate the (German) weighted average for 3 students and the overall average.

```
<TAB!
NAME,  EXAMl,MARKl l
Ernst  1 2   1 2 3 1 3 1 1
Clara  1 1   3 1 2
Sophia 1 3   1 1 2
!TAB>
AVG:=EXAMl ++: *0.6 +(MARKl ++: *0.4)
TOTAL:=AVGl ++:
rnd 2
```

The result is rounded by `rnd`.

**Program 25:** Make a union (all StudentIDs included in one of the tables).

```
aus exams.tab, projects.tab
gib STIDm
```

Here are `exams.tab` and `projects.tab` of type `STID,COURSE,MARK m` resp. `STID,PROJ,HOURS m`.

**Program 26:** Make an intersection (all StudentIDs included in both tables).

```
aus  exams.tab, projects.tab
gib+ STIDm
```

With `gib+` symultaneously joins can be expressed and thus also queries to entire databases are formulated. `gib+` is yet not fully implemented. The following queries refer to a database `uni.tab` whose second table is unstructured. The queries do not need to be modified or only weakly modified if all `uni.tab` tables are unstructured (relational).

```
<STUDENTS!
STID, NAME,  LOC?,    STIP, FAC,  (COURSE,     MARK m),(PROJ, HOURS m) m
1234  Ernst  Oehna    500   Mathe  Algebra     1        Fritz  4
```

```
                                    History     1        Otto   2
                                    Logic       2
1245  Sophia Berlin   400   CS      Algebra     3        Deng   5
                                    Databases   1        Ming   4
                                    Otto        1        Otto   6
3456  Clara  Oehna    450   CS      Databases   1
                                    OCaml       2
4567  Ulrike          400   Art                          Monet 10
5678  Kaethe Gerwisch   0   Art     Apel        1        Monet 20
                                    Repin       1
!STUDENTS>
<FACS!
FAC,  DEAN,   BUDGET, STUDKAP m
CS    Reichel 10000   500
Art   Sitte   2000    600
Maths Gauss   1000    200
Philo Hegel   1000    10
!FACS>
```

The STUDENTS table is structured because 7 components are stored for each set (structured tuple) (students), and the last two components themselves are small tables. For example, the penultimate component of Clara contains two exam results and the last one the empty set of projects.

**Program 27:** Calculate the total budget of the university.

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

**Program 28:** Calculate the total budget and the average budget of the university.

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

**Program 29:** How many students and faculties does the university have.

```
uni.tab
++1
```

**Program 30:** For each course, give the associated students a grade.

```
aus uni.tab
gib COURSE,(NAME,MARK b)m
```

**Program 31:** Give all records (tuples) that contain 1234.

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

**Program 32:** In all tables containing the student identifier STID, select the student data with the number 1234.

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

Program 22 differs only with respect to the FAKS table of program 21. In the result of program 21 this is empty and in program 22 it is completely preserved. In other applications, instead of the student identifier (STID), you could also select part numbers or part numbers.

**Program 33:** Give the Dean and his exams to the student with the number 1234.

```
aus  uni.tab
avec STID=1234
gib+ NAME,DEAN,(COURSE,MARK m)m
```

Here a "join" without join condition is realized. If you were to use `gib` instead of `gib+`, the result set would be left empty because there is no connection between NAME and DEAN.

**Program 34:** Calculate the number of ones for each faculty.

```
aus  uni.tab
```

```
avec MARK=1
gib FAC,COUNT m  COUNT! ++1 MARK
```
**Program 35:** Test the following program.
```
aus uni.tab
avec DEAN in [Reichel Dassow]
gib FAC,DEAN,(NAME,STIP m)m
```
**Program 36:** Give all the students to the faculties of Gauss and Reichel.
```
uni.tab
avec DEAN in [Reichel Gauss]
gib+ FAC,DEAN,(NAME,STIP m)m
```
**Program 37:** Wanted are all faculties that have students with the best and the worst (in this example it is the 3) mark.
```
aus  uni.tab
avec {1 3} in MARKm
gib+ FACS
```
**Program 38:** We are looking for all students who have exactly 2 ones and only these.
```
aus  uni.tab
avec MARKl = [1 1]
gib STUDENTS
```
**Program 39:** We are looking for all data from Ernst (including his faculty data).
```
aus  uni.tab
avec NAME=Ernst
gib+ UNI
```
The program does not need to be changed if all tables of `uni.tab` are in the first normal form (unstructured).
**Program 40:** A bottle and a cork costs one mark and ten. The bottle is a mark more expensive than the cork. How much is the cork?
```
[BOTTLE! 0 .. 110]
CORK:=BOTTLE - 100
avec CORK+BOTTLE = 110
```
**Program 41:** Write 1000 times *I love you.*
1000 mal "I love you!"
**Program 42:** Compute the total number of occurrences of all assemblies and all individual parts of the Hyundai I40 with aluminum rim and without air conditioning.
```
<TAB!
SUP,          PROPERTY,  (SUB,           OCC l)m
bushing       cylindrical
rim                      aluminum rim    1
                         steel rim       1
i30           modern     wheel           4
                         engine          1
                         body            1
i40           fast       wheel           4
                         spare           1
                         body            1
                         conditioning    1
                         engine          1
body          blue
conditioning  clean
```

```
piston ring  round
piston       light       piston ring    2
                         bushing        1
engine       heavy       piston         6
                         screw          8
wheel        round       screw          5
                         tire           1
                         rim            1
tire         black
screw        stable
!TAB>
avec SUB! SUP= rim -> SUB="aluminum rim"
sans SUB! SUP = i40 & SUB=conditioning     # Optional condition
wege i40      # all paths starting in i40 are generated
rec AUX:=first OCC next AUX pred*OCC at OCC
avec SUB! SUB pos- =1
gib SUB,TOTAL m
    TOTAL! ++ AUX
```

result:

```
SUB,          TOTAL  m
aluminum rim   4
body           1
bushing        6
engine         1
piston         6
piston ring   12
rim            4
screw         28
spare          1
tire           4
wheel          4
```