o++o on 42 Pages

(28.09.2021)

Klaus Benecke

Table of Contents

1 Vision 1
2 Our design criteria for an end-user language 2
3 Calculations with o++o: More than a pocket calculator 2
4 Schemes and TTDs of structured tables 11
5 Simple recursive assignments 14
6 Queries 16
7 Queries to multiple tables without joins (igib) 22
8 A BOM explosion with o++o number 25
9 Generation of images 27
10 Diagrams 30
11 "Hello otto" - gimmick 32
12 o++o for School 33
13 Closing words, a quote from Adam Ries 41
14 Literature 41

1 Vision

There are the 4 basic single value operations addition, subtraction, multiplication and division with almost standardized notations `+, -, × (*)` resp. `: (÷)`. Later many operations were added like `sin, cos, log,` … . Because of ongoing digitalization we need also powerful mass data operations with standards. The set of set-theoretic operations as intersection (∩), union (∪), set difference (\ or -) have been extended to include the Cartesian product (×), join (|×|), projection (π) and selection operations (σ). These operations are hidden in SQL – the standard database language for Relational databases. Most of today's data are stored with Relational Systems like ORACLE, MYSQL, HANA, DB2, MariaDB, … . The Relational Model was invented by the British mathematician E. F. Codd. Because the Relational model was not powerful enough to fulfill the requirements of the industrial applications the computer scientists extended the model by multi-sets, simple aggregate functions like sum, avg, … , by groupby, having, sorting operations … . Despite these extensions, you cannot process structured tables with SQL. But there are very simple structured tables as

```
SUBJECT,MARKl l
maths   1 2 2 3 1
physics 2 1 4 1 3 4
```

, which can be understood by children. Here, we have for each subject a list (l) of marks. The whole table consists of a list (the last l) of (SUBJECT,MARKl)-pairs. SQL and even EXCEL is not able to handle such schemes like `(SUBJECT,MARKl l)`. Therefore we believe that SQL will never achieve its original goal - to become an end-user language. So we need a new standard for databases and collections of documents. Our proposal: o++o

Our vision is that o++o will become the standard for querying Relational databases, big-data systems and even systems, which manage huge amounts of documents.

Facebook and Co. are working on preparing messages in such a way that they primarily address emotions, as if the social situation wasn't already sufficiently heated. Our approach is different. We are working on providing the end user with tools to strengthen his rational judgment with the help of the computer. To do this, he must be able to access extensive, trustworthy, publicly available information in the form of large tables and documents similar to Wikipedia. The individual should be able to evaluate these independently. Only in this way can he form his own judgment.

Even if this language is as simple as possible, it requires a certain amount of learning in contrast to the "Facebook approach".

Such a Programming language in combination with trustworthy data could be a big step towards a further democratization of society. A lot of "fake news" could easily be paralyzed by anyone, if he can write computer programs by himself to generate appropriate facts or statistical computations.

2 Our design criteria for an end-user language

1. A mathematical foundation is required.
2. Methodological, didactic and pragmatic questions have to be considered firstly and then efficiency problems.
3. The end-user programs should be short and easy to read, as well as the most important key words.
4. No loops (for, while). Loops often lead to programs that are difficult to read and difficult to modify. General recursion requires a relatively high level of abstraction and therefore a high learning curve; this should also be avoided. As a result, the programs should simply be written and processed from top to bottom and from left to right. That requires powerful operations.
5. The end-user language should be universally applicable. It must be usable not only for relations (flat simple tables), but also for structured tables and structured documents. It should not only be suitable for queries to a wide variety of systems, but also for a variety of calculations.
6. In order to be used in school lessons, it should support the various mathematical sub-areas as well as offer benefits for the other subjects.
7. The end-user language should be so powerful that it can replace other systems and languages such as spreadsheets, *XQuery*, *XPath* and *SQL*.
8. From the end-user perspective, there should be a uniform system with uniform syntax (notation) for processing mass data, just as the operations of single value data processing (+ - * : sqrt sin…) are standardized.

3 Calculations with o++o: More than a pocket calculator

o++o is designed to be user-friendly; that means so simple and memorable syntax as possible as well as short programs. His arithmetic operations follow the logic of mental arithmetic and therefore run like a simple pocket calculator.

The following programs should be tried out at ottops.eu, as you can hardly learn programming without testing it yourself.

o++o, more precisely *ottoPS* (otto programming language), is not just a much more advanced pocket calculator. *o++o* also allows queries to structured TABles and docuMENTS (TABMENTs). *o++o* is therefore also suitable for enabling database queries and later becoming the basis of certain search engines. *o++o* is an end-user programming language that can easily manage repeating groups. It allows queries to tables and documents to be formulated particularly easily.

Program 3.1: Calculate the fourth root of 16.

```
16 sqrt sqrt
```

Result:

```
2.
```

(Note: Results in the GUI are shown after clicking the button labeled "tab".)

You can see that unary operations are written after the input value (postfix). This saves us 4 parentheses compared to the well-known notation `sqrt (sqrt (16))`.

Program 3.2: Calculate the sine of "pi halve".

```
pi:2 sin
```

Result:

```
1.
```

Program 3.3: Calculate the sine of 30 degrees.

```
30:180*pi sin
```

Result:

```
0.5
```

Program 3.4: How many 10-digit binary numbers are there?

```
2 hoch 10
```

Result:

```
1024
```

"hoch" is German and shorter than "to the power of".

Program 3.5: Calculate the edge length of a cube of volume 2 (the third root of 2).

```
2 hoch 1/3
```

Result:

```
1.25992104989
```

or

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

Result:

```
1.25992104989
```

1/3 is a rational number. That means, "/" is not an operation in contrast to ":". Since we always calculate from left to right, 2 hoch 1: 3 = (2 hoch 1): 3 = 0.666666666667.

Program 3.6: Compute an average.

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

Result:

```
2.33333333333
```

In order to save writing effort, you can write without any visible separators and brackets a list of values (here the numbers in the line). The average operation (`++:`) is now applied to this list. In addition to the average, you can also form a sum (`++`), the product (`**`), count (`++1`), the maximum (`max`), the sine (`sin`), `ln`, ... Since `sin` only requires one input value, using the sine operation instead of the `++` operation results in a list of output values. You could then apply `++:` to this list again and then just get a number. The above notation may take some getting used to, but it is more compact than the old notation

```
avg ([1; 3; 2; 1; 3; 4]).
```

Especially when several operations are used one after the other, you save a lot of brackets and thus reduce the causes of errors. We consider `++:` and the other operations mentioned here as unary, just like `sin` or `sqrt`, and write them after the input value, since we regard the given numbers as **one** list (**one** tabment).

Program 3.7: Calculate the value of the term “sqrt(abs(sin(7.1))+abs(cos(8.1)))”.

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

Result:

```
0.986160835697
```

or in single line with a pair of parentheses

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

Result:

```
0.986160835697
```

In the three-line solution, as is usual in programming languages, the calculation is from top to bottom. You cannot calculate the value of the second line if you do not omit the "+" sign. Therefore the values of the first and second lines are simply added. From the overall result of the second line, the square root is then taken by the third line.

Program 3.8: Find the product of multiple numbers.

```
3 5 2 2 **
```

Result:

```
60
```

In *o++o*, several operations are written according to the very old "forest principle". There is a word for the tree and "tree tree" means forest. The above `**` therefore replaces 3 multiplication signs. Because of this operation, we can omit the factorial function.

`9!` can be expressed in *o++o* by `1 ..9 **`.

Program 3.9: Multiply each decimal number in a list by another number.

```
2.40 2.70 7.90 * 1.19
```

Result:

```
2.856 3.213 9.401
```

Each number in the input list is multiplied by `1.19`. If the given numbers are net prices, the result represents the associated gross prices (in Germany). If, on the other hand, the given numbers are amounts in one currency and if `1.19` is the exchange rate, then the result represents the values in the other currency. If the numbers in the list are lengths of rectangles, so there are 3 areas of rectangles. We see that decimal numbers are not represented with a comma but with a decimal point.

Program 3.10: Find the sum of many positive and many negative numbers without using many minus signs and brackets.

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

Result:

```
4
```

Program 3.11: Calculate the circumferences of several circles, given the radii. The results are to be rounded to 2 digits after the point.

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

Result:

```
25.13 31.42 37.7 12.57 23.25 61.39
```

The program can also be written in one line.

Program 3.12: Calculate the perimeters and areas of several circles, given the radii.

```
Rl:= 4 5 6 2 3.7 9.77
PERIMETER:=R*pi*2
AREA:=R*R*pi
rnd 1
```

Result:

```
R,  PERIMETER, AREA l
4.  25.1        50.3
```

```
5.  31.4          78.5
6.  37.7         113.1
2.  12.6          12.6
3.7 23.2          43.
9.8 61.4         299.9
```

By `Rl:=` each element of the given list is assigned the name (called “tag” in *o++o*) R. With an assignment (“`:=`”) the given table is extended by a new column. At the top, the columns PERIMETER and AREA are added one after the other, resulting in a table of type `R,PERIMETER,AREA l`. `l` stands for list. Unfortunately, this can easily be confused with the one.

Program 3.13: Calculate the areas and perimeters of several 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
```

Result:

```
A,   B,   PERIMETER, AREA l
1.23 5.67 13.8        6.9741
7.65 4.32 23.94      33.048
9.87 6.54 32.82      64.5498
```

The TAB brackets (“`<TAB!`”, “`!TAB>`”) are only required in the program part of the system. In a file the type is recognized by the system by the suffix “`.tab`”. In the TAB representation, the values must be aligned with the left-hand side of the associated column names.

Program 3.14: Find the total price of a simple calculation.

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

Result:

```
3.24
```

Here simply the sum over the numbers in the given table (a list of pairs) is formed. The article values are words and therefore have no influence on the result. Now, we replace `++` with `*` 1.1, this creates a table with 2 columns and three rows (records, tuples), each number including a tip of 10%:

Program 3.15: Multiplication of a table with a number.

```
<TAB!
ARTICLE, PRICE l
Bier      0.61
Brause    0.23
Schnitzel 2.40
!TAB>
* 1.10
```

Result:

```
ARITICLE, PRICE l
Bier      0.671
Brause    0.253
Schnitzel 2.64
```

Then you can add again

```
++
```

to get the grand total (3.564).

Program 3.16: Find the total price of a more complicated calculation given by a simple table.

```
<TAB!
ARITICLE, PRICE, COUNT l
Bier      0.61   7
Brause    0.23   3
Schnitzel 2.40   4
!TAB>
POSPRICE:=PRICE*COUNT
++ POSPRICE
```

Result:

```
14.56
```

As a result of the assignment, the given table is expanded by a new column with the column name POSPRICE, whereby each of the three PRICE value is multiplied by the associated COUNT value. In order to determine the total number of the pub visit, the ++ operation must be given a second input value. Otherwise the total of all nine numbers in the above table would be formed. Both lines of the program can also be replaced by

```
++ PRICE*COUNT
```

. 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 sign “:=” of the assignment must be distinguished from the equal sign =. The equal sign as well as <, >, <=, in etc. is required to formulate conditions. Conditions are used for selection (filtering of lines of structured tables).

For example, you add a condition

```
avec ARTICLE = Bier
```

or just

```
avec Bier
```

the final result is only the total price for the seven beers. If you want to calculate the price for the other articles instead, you use

```
sans ARTICLE = Bier
```

or simply

```
sans Bier
```

in its place.

Column names (metadata) must always be capitalized. The key words (`gib, sans, avec`, ...) must always be written in lower case. If you always write a word of the primary data with upper and lower case letters, the program becomes easier to read.

Program 3.17: Compute the total price of a longer pub visit.

```
<TABH!
ARITICLE, PRICE,COUNTl l
Bier      0.61  7 6 5
                3
Brause    0.23  3 4
Schnitzel 2.40  4 3 2
!TABH>
POSPRICE:=PRICE*COUNT
++ POSPRICE
```

Result:

```
36.02
```

Here we are dealing with a structured table that contains several orders for each position. Since we have chosen the horizontal version `TABH` of `TAB`, we do not have to write the `COUNT`-entries below each other, so that the pub visit table can be presented in a compact manner. If, on the other hand, one does not consider the final result but only the result of the application of the assignment, the `COUNT` and `POSPRICE` values must be below each other, since the resulting structured table would otherwise become too confusing:

```
ARITICLE, PRICE,(COUNT,  POSPRICE 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
```

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

```
POSPREIS:= COUNTl ++ *PRICE
```

, this results in a more compact intermediate table:

```
ARITICLE, PRICE, POSPREIS,COUNTl 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
```

Both program lines can also be replaced by `++ PRICE*COUNT`.

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

Program 3.16 could then look like this:

```
pub_visit.tabh
++ PRICE*COUNT
```

This example can be applied to many applications. When bowling you could also set up a table bowling.tabh with a repeating group: NAME, THROWl m.

The names and counts of the individual litters can be entered as you wish.

`m` abbreviates set and `l` list. For each bowler you can then determine the total amount by means of an assignment and then sort the data in descending order. If you only want certain columns in the result and you want to sort by these, you need a `gib`-clause.

Program 3.18: Determine the overall result of bowling for each person and sort the data accordingly.

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

It can also be a little shorter:

```
bowling.tabh
gib TOTAL,NAME m-
TOTAL:=THROW! ++
```

The reference to the aggregation (here `++`) results from the header of the desired table. `TOTAL` is one aggregation per `NAME`. Sets (`m`, `m-`) are always sorted according to the column names specified first.

Program 3.19: Find a weighted average for 3 students and the overall average.

```
<TABH!
NAME,  EXAl, MARKl l
Ernst  1 2   1 2 3 1 3 1 1
Clara  1 1   3
Sophia 1 3   1
!TABH>
TOT:=EXAl ++: *0.6 +(MARKl ++: *0.4)
```

```
TOTAL:=TOTl ++:
rnd 2
```

Result:

```
TOTAL,(NAME,  TOT, EXAl, MARKl  l)
1.66   Ernst  1.59 1 2   1 2 3 1 3 1 1
       Clara  1.8  1 1   3
       Sophia 1.6  1 3   1
```

Program 3.20: A bottle with cork costs one mark and ten. The bottle is a mark more expensive than the cork. How much does the cork cost?

```
BOTTLEl:= 0 .. 110
CORK:= BOTTLE - 100
avec CORK+BOTTLE = 110
```

Result:

```
BOTTLE, CORK l
105     5
```

The first assignment gives each of the numbers from 0 to 110 the tag BOTTLE. This is best seen if you look at the ment-representation:

```
<TABM>
   <BOTTLE> 0 </BOTTLE>
   <BOTTLE> 1 </BOTTLE>
   <BOTTLE> 2 </BOTTLE>
   <BOTTLE> 3 </BOTTLE>
   . . .
   <BOTTLE> 108 </BOTTLE>
   <BOTTLE> 109 </BOTTLE>
   <BOTTLE> 110 </BOTTLE>
</TABM>
```

If you had incorrectly written the assignment `BOTTLE:= 0 ..110`, the tag `BOTTLE` would only occur once, assigning the tag to the whole list of values rather than each value:

```
<BOTTLE>
0
1
2
3
. . .
108
109
110
</BOTTLE>
```

second solution:

```
BOTTLEl:= 0 ..110
CORK1 := BOTTLE – 100
CORK2 := 110 – BOTTLE
avec CORK1=CORK2
```

Result:

```
BOTTLE, CORK1, CORK2  l
105     5      5
```

This solution is advantageous from a methodological point of view, as the first 3 program lines can be illustrated by clicking on the image button. You can see that there are 2 straight lines, the intersection of which is determined by the condition.

Program 3.21: Write 20 times „I love you“.

```
"Je vous aime." mal 20
```

Result:

TEXTl

```
"Je vous aime." "Je vous aime." "Je vous aime." "Je vous aime."
"Je vous aime." "Je vous aime." "Je vous aime." "Je vous aime."
"Je vous aime." "Je vous aime." "Je vous aime." "Je vous aime."
"Je vous aime." "Je vous aime." "Je vous aime." "Je vous aime."
"Je vous aime." "Je vous aime." "Je vous aime." "Je vous aime."
```

“`mal`” is a binary operation that, like all other binary operations, is written infix (between the input values).

4 Schemes and TTDs of structured tables

We are adding this chapter at this point to clarify the difference between simple (flat) and structured tables.

First of all, a table is divided into n columns A1, A2,…, An. The column names in the o++o program must not contain any lowercase letters. If all A1, A2,..., An have an elementary type, then an `A1, A2,..., An` table contains exactly one simple line, e. g .:

```
NAME,LOC,  SALARY
Paul Oehna 1000
```

If we add a collection symbol, e.g. `l` for list, the table can contain 0, 1 or more rows. Example:

```
NAME,   LOC,     SALARY l
Paul    Oehna    1000
Sophia  Dallgow  900
Claudia Dallgow  2000
Clara     Oehna   900
```

tab-representation of a flat table (list of triples)

NAME	LOC	SALARY
Paul	Oehna	1000
Sophia	Dallgow	900
Claudia	Dallgow	2000
Clara	Oehna	900

(graphik-representation of the flat table)

If we replace the `l` with the set symbol `m`, the table is displayed in a sorted manner.

```
NAME,   LOC,    SALARY m
Clara   Oehna    900
Claudia Dallgow 2000
Paul    Oehna   1000
Sophia  Dallgow  900
```

If you want to sort this table by location, you just swap the columns in a corresponding statement

```
gib LOC,NAME,SALARY m
```

```
LOC,    NAME,   SALARY m
Dallgow Claudia 2000
Dallgow Sophia   900
Oehna   Clara    900
Oehna   Paul    1000
```

You can now see that the table contains some redundancy. This can be eliminated with a structured table `(gib LOC,(NAME, SALARY m)m)`:

```
LOC,   (NAME,   SALARY m) m
Dallgow Claudia 2000
        Sophia   900
Oehna   Clara    900
        Paul    1000
```

(`structured_table.tab`)

This table contains two struples (structured tuples = structured records = structured elements), i.e., e.g. that the number (`++1`) of the (now structured) elements of the table is no longer 4 but 2. The number of people remains of course 4. Nevertheless, we can process this table with o++o in the same way as the preceding ones. You can also create two or three tables from the output table with a `gib`-statement:

```
LOCm,   NAMEm,  SALARYm
Dallgow Clara    900
Oehna   Claudia 1000
        Paul    2000
        Sophia
```

(`three_collections.tab`)

This overall table contains all the elementary data of the output table, but the other information has been lost. `Dallgow` and `Clara` are in the same line in this tab-representation, but `Clara` does not live in `Dallgow`. You always have to look at the scheme. Strictly speaking, we are dealing with a triple of three sets: {Dallgow Oehna}, {Clara Claudia Paul Sophia}, {900 1000 2000}

Here the comma stands for "pair formation" and the space separates the elements of the sets.

The schema of `structured_table.tab` is:

```
LOC,(NAME,SALARY m)m
```

In many cases this metadata is sufficient. However, this does not yet show, for example, how one can calculate with which column values. The `TTD` (Tabment Type Definition) makes this clear:

```
TABMENT! (LOC,(NAME,SALARY m) m)
SALARY! ZAHL
NAME LOC! WORT
```

`TABMENT` is a made-up word from TABle and docuMENT. If the table is saved under the name above, the `TTD` is extended accordingly:

```
TABMENT!  STRUCTURED_TABLE
STRUCTURED_TABE! (LOC,(NAME,SALARY m) m)
SALARY! ZAHL
NAME LOC! WORT
```

The spelling of the collection symbols probably takes some getting used to. A collection symbol is written like the other unary operations according to the scheme. As I said, the `LOC` scheme stands for one location. This means that an o++o table with this scheme (with this header) can only contain one location. In contrast, relations or EXCEL-tables with the header `LOC` contain any number of locations. In o++o you have three options to represent these "relations":

```
LOC          LOCl        LOCb       LOCm
Dallgow      Dallgow     Dallgow    Dallgow
Oehna        Oehna       Dallgow    Oehna
Dallgow      Dallgow     Oehna
Oehna        Oehna       Oehna
EXCEL        o++o-list   o++o-bag   o++o-set
```

`Bag` refers to a multi-set. We recognize that bags are sorted and sets are sorted and, in addition, sets do not contain any duplicates. The schemes `LOCm` and `LOC m` express the same thing. The difference is only visible when there are several columns:

```
LOC,NAME m
```

contains several (`LOC, NAME`)-pairs, whereas

```
LOC,NAMEm
```

can only contain one location but with multiple names. This means that a collection symbol, which follows a column name without spaces, saves a pair of brackets.

```
LOC,NAMEm
```

is equivalent to

```
LOC,(NAME m)
```

. Tables of the type

```
LOC,NAMEm m
```

can then have many different tables of the type

```
LOC,NAMEm
```

. In the context of the `gib` statement contains the result of

```
gib LOC,NAMEm m
```

each location only once. So that can

```
LOC,      NAMEm m
Dallgow  Claudia
Dallgow  Sophia
Oehna    Ernst
         Clara
```

not be the result of this give instruction, but:

```
LOC,      NAMEm m
Dallgow  Claudia
         Sophia
Oehna    Ernst
         Clara
```

The advantage of this structure only really comes into play if you save additional data for each location, such as number of inhabitants, mayor etc. The inner `m` says that there are a set of names for every location. This means again that no name can occur more than once within a location. Nevertheless, a second `Ernst` could of course also be live in `Dallgow`.

In *o++o*, every tabment has a scheme. This even applies to individual values to which the user has not assigned a scheme. `Gerwisch`, for example, has the elementary scheme `WORT` (=word) and `39175` the elementary scheme `ZAHL` (= big int). This makes `Gerwisch` a tabment. `Gerwisch` is not an object of the relational data model in contrast to "{Gerwisch}". That seems subtle, but it has a major impact on the architecture of the data model.

5 Simple recursive assignments

In addition to the simple calculations (`:=`) with an associated formula, which add a new column to a table, there are also so-called recursive assignments. Two formulas are given here. The first formula is intended for the first element of the new column of the corresponding collection and the second for the remaining elements. The formula for the remaining elements can refer to the predecessor of the column. That is in the following program `AMOUNT pred`.

Program 5.1: What happens to 100 euros after 10 years with 9% growth (interest).

```
YEARl:= 0 .. 10
AMOUNT:= first 100.  next AMOUNT pred*1.09  at YEAR
rnd 2
```

Result:

```
YEAR, AMOUNT  l
 0    100.
 1    109.
 2    118.81
 3    129.5
 4    141.16
 5    153.86
 6    167.71
 7    182.8
 8    199.26
 9    217.19
10    236.74
```

Program 5.2: Convert a binary number (here 10111 = 23) into a decimal number.

```
BITl:= 1 0 1 1 1
DECI:= first BIT  next DECI pred*2+BIT  at BIT
```

Result:

```
BIT, DECI l
1     1
0     2
1     5
1    11
1    23
```

If you don't want the whole development but only the last element of the list, you can add the condition `avec BIT pos- = 1` or simply `last`. If the last bit is not wanted either, `gib DECI` should follow.

Program 5.3. Convert a decimal number (here 23) into a binary number.

```
DIVRESTl := first 23 divrest 2
            next  DIVREST pred nth 1 divrest 2
            while DIVREST ++ > 0
gib RESTl-
```

Result (tabh-format):

```
1 0 1 1 1
```

With the above recursive extension, two new columns are introduced under one name, `DIV` and `REST`. `divrest` here is the integer division with the integer and the remainder (a pair of numbers with tags `DIV` and `REST`).

Program 5.4: Calculate the monthly development of a home construction loan of 110,000 euros at 2% annual interest if 900 euros are paid off monthly.

```
# Initial credit 110000 Euro
MONLOAD:=900            # monthly load
PROC:=1.02 hoch 1/12 -1 # 2 % interest per YEAR
CREDIT,INTEREST,REPAYMENT l:=
    first 110000.                    , 0.          , 0.
    next CREDIT pred-(REPAYMENT pred),(CREDIT*PROC),(MONLOAD-INTEREST,CREDIT min)
    while CREDIT > 0 at PROC
YEAR,MON:=CREDIT pos -1 divrest 12+1 leftat CREDIT
PROC::=PROC*100
rnd 2
```

Result:

```
MONLOAD, PROC, (YEAR, MON, CREDIT,   INTEREST,REPAYMENT l)
900      0.17    1     1   110000.     0.       0.
                 1     2   110000.   181.67   718.33
                 1     3   109281.67 180.49   719.51
                 1     4   108562.16 179.3    720.7
                 1     5   107841.46 178.11   721.89
                 1     6   107119.57 176.92   723.08
                 1     7   106396.49 175.72   724.28
                 1     8   105672.21 174.53   725.47
                 1     9   104946.74 173.33   726.67
                 1    10   104220.06 172.13   727.87
                 1    11   103492.19 170.93   729.07
                 1    12   102763.12 169.72   730.28
                 2     1   102032.84 168.52   731.48
                 2     2   101301.35 167.31   732.69
                 2     3   100568.66 166.1    733.9
                 2     4    99834.76 164.89   735.11
                 . . .
                11    12     5930.53   9.79   890.21
                12     1     5040.33   8.32   891.68
                12     2     4148.65   6.85   893.15
                12     3     3255.5    5.38   894.62
                12     4     2360.88   3.9    896.1
                12     5     1464.78   2.42   897.58
                12     6      567.2    0.94   567.2
```

Somewhat more sophisticated recursive assignments (with o++o numbers) can be found in chapter 8.

6 Queries

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

Program 6.1: Select in a given table.

```
rivers.tabh
avec LENGHT > 500
```

"`avec`" is French and means "with". Similarly, we use "`sans`" for "without".

Now the rivers are to be sorted according to their length and output with tributaries:

Program 6.2: Selection with subsequent sorting.

```
aus  rivers.tabh
avec LENGHT>520
gib  LENGHT,RIVER,TRIBUTARYm m
```

Result (tabh-format):

```
LENGHT, RIVER, TRIBUTARYm   m
 544    Main   "Fraenkische Saale" Gersprenz Kinzig
               Lohrbach Nidda Rechtenbach Regnitz
               "Roter Main" Tauber "Weisser Main"
 544    Mosel  Alf Dhron Elzbach Kyll Lieser Madon Meurthe
               Moselotte Orne Ruwer Saar Salm Sauer Seille
               Vologne
 544    Saar   Blies Nied Prims Rossel
 866    Oder   Bartsch Birawka Bober Eilang Faule Obra
               Glatzer Neisse Hotzenplotz Ihna Iseritz
               Kaltebach Katzbach Klodnitz Lohe Malapane
               Mietzel Neisse Ohle Olsa Oppa Ostrawitza
               Pleiske Raude Roehrike Stober Summina Thue
               Tiefenburgbach Warthe Weide Weistritz
               Weissfurth Welse Zinna
1165    Elbe   Adler Aland Alster Biela Bille Dahle
               Doellnitz Eger Elde Este Gottleuba Havel
               Ilmenau Iser Jahna Jeetze Kamnitz Kirnitzsch
               Lachsbach Lockwitz Loecknitz Model Moldau
               Mulde Mueglitz Ohre Oste Polzen Priessnitz
               Saale "Schwarze Elster" Stepenitz Stoer
               Tanger Weisseritz Wesenitz "Wilde Sau"
               Zidlina
1320    Rhein  Aare Ahr Alb Alb bei Albbruck Argen Birs
               "Bregenzer Ache" Duessel Elde Emscher Erft
               Glatt Hinterrhein Ill Kander Kinzig
               Kraichbach Lahn Lauter Leiblach Leimbach
               Leopoldskanal Lippe Main Mosel Murg Nahe
               Neckar Queich Ruhr Schussen Sieg Speyerbach
               Thur Toess Vorderrhein Weschnitz Wied Wiese
               Wupper Wutach
2845    Donau  Abens Altmuehl Blau Breg Brenz Brigach Cerna
               Donaudelta Donaukanal Drau Eipel Enns Fischa
               Friedberger Ach Grosse Laaber Grosse Muehl
               Guenz Hron Iller Ilz Inn Ipel Isar Jantra
               Jiu Kamp Kleine Paar Krems Lauchert Lech
               Leitha March Mindel Morava Naab Olt Paar
               Pruth Raab Regen Riss Save Schmutter
```

```
                    Schwechat Temesch Theiss Timok Traisen Traun
                    Vils Vah Woernitz Ybbs Zusam
```

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

Our `rivers.tabh` file also contains an additional repeating group for each river `LAND, BUNDESLANDm m`, which indicates through which states and federal states the river flows. In `rivers.tabh`, the `RIVER` is superordinate to the federal state. If you want to reverse this, you can do this again simply by specifying the header of the desired table.

Program 6.3: Restructuring of the rivers file.

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

Result (tabh)**:**

```
BUNDESLAND,            RIVERm  m
Baden Württemberg      Donau Iller Neckar Rhein
Bayern                 Abens Donau Guenz Iller Inn Isar
                       Lech Main Mindel Naab Regen Saale
                       Wertach Woernitz
Berlin                 Havel Spree
Brandenburg            Aland Elde Havel Neisse Oder Spree
                       Stepenitz Uecker
Bremen                 Weser
Hamburg                Alster Elbe Este Wandse
Hessen                 Fulda Main Neckar Rhein Werra Weser
Mecklenburg Vorpommern Aland Elde Havel Loecknitz Oder
                       Peene Stepenitz Uecker Warnow
Niedersachsen          Elbe Este Fulda Jeetze Loecknitz
                       Oste Werra Weser
NordrheinWestfalen     Rhein Ruhr Weser Wupper
RheinlandPfalz         Mosel Rhein Saar
Saarland               Saar
Sachsen                Elbe Neisse Spree
Sachsen Anhalt         Bode Elbe Havel Ilm Saale Unstrut
Schleswig Holstein     Alster Elbe Stoer Trave Wandse
Thüringen              Ilm Saale Unstrut Werra
```

As a result of this free restructuring, 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 federal states. Each state only appears once, as we have chosen a lot as the outer collection. The beginning of a part of the program can also be marked with `aus` (from).

Program 6.4: Form a union (all student IDs contained in one of the tables).

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

Result (tabh):

```
STIDm
1234 1245 3456 4567 5678
```

Here `exams.tab` and `projects.tab` are the following tables:

```
STID, COURSE,     MARK  m,(STID, PROJ, HOURS  m)
1234  Algebra     1        1234  Fritz  4
1234  Geschichte  1        1234  Otto   2
1234  Logik       2        1245  Konfuz 5
1245  Algebra     3        1245  Ming   4
1245  Datenbanken 1        1245  Otto   6
1245  Otto        1        4567  Monet 10
3456  Datenbanken 1        5678  Monet 20
3456  OCaml       2
5678  Apel        1
5678  Repin       1
```

Program 6.5: Take an intersection (all student IDs contained in both tables).

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

Result (tabh):

```
STIDm
1234 1245 5678
```

With `igib` (see chapter 7) "joins" can also be expressed and thus queries to entire databases can be formulated. The following queries refer to a database `uni.tab`, the second table of which is unstructured. The queries do not have to be modified or only slightly modified if all tables from `uni.tab` are unstructured (relational).

```
FAC,  DEAN,   BUDGET, STUDCAP m
Inf   Reichel 10000   500
Kunst Sitte    2000   600
Mathe Dassow   1000   200
Philo Hegel    1000    10
STID, NAME,  LOC?,    STIP, FAC,  (COURSE,     MARK m),(PROJ, HOURS m) m
1234  Ernst  Oehna    500   Mathe  Algebra     1        Fritz  4
                                   Geschichte  1        Otto   2
                                   Logik       2
1245  Sophia Berlin   400   Inf    Algebra     3        Konfuz 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
```

The database consists of 2 tables. The table `STUDENTS` is structured, since 7 components are stored for each record (record = structured tuple = student) and the last two components are themselves again small tables. The penultimate component of `Clara`, for example, contains two exam-results and the last the empty set of projects. The `FACS` table is flat (1. normal form = relational).

Program 6.6: Calculate the university's total budget.

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

Result:

```
14000
```

Program 6.7: Calculate the overall budget and the average budget of the university.

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

Result:

```
SUMBUD, AVGBUD
14000   3500.
```

Since the last two lines start with more than 3 spaces, they still belong to the previous line from a logical point of view.

Program 6.8: How many faculties and students does the university have.

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

Result (tab)**:**

```
4     5
```

Program 6.9: Give the corresponding students with grades for each course.

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

Result:

```
COURSE,    (NAME,  MARK  b) m
Algebra     Ernst  1
            Sophia 3
Apel        Kaethe 1
Datenbanken Clara  1
            Sophia 1
Geschichte  Ernst  1
Logik       Ernst  2
OCaml       Clara  2
```

```
Otto         Sophia 1
Repin        Kaethe 1
```

Program 6.10: Give all the records (struples) that contain 1234.

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

Result (hsq=hierarchical sequential)**:**

```
STID NAME LOC STIP FAC
 COURSE, MARK
  PROJ HOURS
   FAC DEAN BUDGET STUDCAP
1234 Ernst Oehna 500 Mathe
 Algebra 1
 Geschichte 1
 Logik 2
  Fritz 4
  Otto 2
```

Program 6.11: Select in all tables that contain the student identifier for the student with the number 1234.

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

Result (hsq):

```
STID NAME LOC STIP FAC
 COURSE MARK
  PROJ HOURS
   FAC DEAN BUDGET STUDCAP
1234 Ernst Oehna 500 Mathe
 Algebra 1
 Geschichte 1
 Logik 2
  Fritz 4
  Otto 2
   Inf Reichel 10000 500
   Kunst Sitte 2000 600
   Mathe Dassow 1000 200
   Philo Hegel 1000 10
```

Program 6.11 only differs from Program 6.10 with regard to the FACS table. In the result of program 6.10 this is empty and in program 6.11 it remains completely. In other applications, article numbers or part numbers, ... could also be selected instead of the student identifier (STID).

7 Queries to multiple tables without joins (igib)

Program 7.1: Give the dean and his exams to the student with the number 1234.

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

Result:

```
NAME, DEAN,  (COURSE,    MARK  m) m
Ernst Dassow  Algebra    1
              Geschichte 1
              Logik      2
```

Here a "join" is implemented without a join condition. If you were to use `gib` instead of `igib`, the result set would remain empty, since `gib` does not establish a connection between `NAME` and `DEAN` via `FAC`.

Program 7.2: Calculate the number of ones for each faculty.

```
aus  uni.tab
avec MARK=1
gib  FAC,CNT m
     CNT:=MARK! ++1
```

Result (tab)**:**

```
FAC,  CNT m
Inf   3
Kunst 2
Mathe 2
Philo 0
```

Program 7.3: Try the following program.

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

Result:

```
FAC,  DEAN,  (NAME, STIP m)m
Inf   Reichel
Mathe Dassow
```

Program 7.4: Add all the students belonging to the Dassow and Reichel faculties.

```
aus  uni.tab
avec DEAN in [Reichel Dassow]
igib FAC,DEAN,(NAME,STIP m)m
```

Result:

```
FAC,  DEAN,  (NAME,  STIP  m) m
Inf   Reichel Clara  450
              Sophia 400
Mathe Dassow  Ernst  500
```

Program 7.5: We are looking for all faculties that have students with at least two ones and one three.

```
aus  uni.tab
avec {{1 1 3}} inmath MARKb
igib FAC,DEAN m
```

Result:

```
FAC, DEAN  m
Inf  Reichel
"b" stands for bag (multi-set).
```

Program 7.6: We are looking for all students who have exactly 2 ones.

```
aus  uni.tab
avec MARKl = [1 1]
gib  STUDENTS
```

Result (hsq):

```
STID NAME LOC STIP FAC
 COURSE, MARK
  PROJ HOURS
5678 Kaethe Gerwisch 0 Kunst
 Apel 1
 Repin 1
  Monet 20
```

Program 7.7: We are looking for all of Ernst's data (including his faculty data).

```
aus  uni.tab
avec NAME=Ernst
igib
```

Result (hsq)**:**

```
STID NAME LOC STIP FAC
 COURSE MARK
  PROJ HOURS
   FAC DEAN BUDGET STUDCAP
1234 Ernst Oehna 500 Mathe
 Algebra 1
 Geschichte 1
 Logik 2
  Fritz 4
  Otto 2
   Mathe Dassow 1000 200
```

The program does not have to be changed if all tables from `uni.tab` are in the first normal form (unstructured).

Program 7.8: Determine the total price of a longer visit to the pub using a separate price table.

```
<TABH!
ARITICLE, COUNTl m
Bier      2 4 5
Brause    3 2
Schnitzel 4 3
!TABH>
,articles.tab
igib TOT,(ARITICLE,TOT m)  TOT:=COUNT*PRICE! ++
```

Result:

```
TOT,  (ARITICLE, TOT2 m)
79.3   Bier      29.7
       Brause    16.
       Schnitzel 33.6
```

articles.tab:

```
ARITICLE, PRICE m
Bier      2.7
Brause    3.2
Schnitzel 4.8
Steak     4.8
```

Program 7.9: We are looking for the dean of Ernst. (query to a relational database.)

```
unirelational.tab
avec NAME=Ernst
igib DEAN
```

Result:

```
DEAN
Dassow
unirelational.tab has the following TTD:
TABMENT! UNIRELATIONAL
UNIRELATIONAL! STUDENTS,EXAMS,PROJECTS,FACS
PROJECTS! (STID,PROJ,HOURS m)
STUDENTS! (STID,NAME,LOC?,STIP,FAC m)
EXAMS! (STID,COURSE,,MARK m)
FACS! (FAC,DEAN,BUDGET,STUDCAP m)
```

Program 7.10: Give me all the data from Ernst.

```
unirelational.tab
avec NAME=Ernst
igib
```

Result (hsq):

```
STID NAME LOC STIP FAC
 STID COURSE MARK
  STID PROJ HOURS
   FAC DEAN BUDGET STUDCAP
1234 Ernst Oehna 500 Mathe
 1234 Algebra 1
 1234 Geschichte 1
 1234 Logik 2
  1234 Fritz 4
  1234 Otto 2
   Mathe Dassow 1000 200
```

It should be noted that all of Ernst's data appear in the result without using a cross product. Otherwise the projects "Fritz" and "Otto" would appear three times each.

Program 7.11: We are looking for the exam results of the student with the number 1234.

```
unirelational.tab
avec STID=1234
igib NAME,DEAN,(COURSE,MARK m)m
```

Result:

```
NAME, DEAN,  (COURSE,    MARK m) m
Ernst Dassow  Algebra    1
              Geschichte 1
              Logik      2
```

It should be noted that the result contains information from three tables without the need for a join condition.

8 A BOM explosion with o++o number

Program 8.1: We are looking for all assemblies and individual parts of the Polo and the motor.

```
<TAB!
SUPPART,   PROPERTY,   (SUBPART,   CNT m) m
Buchse     zylindrisch
Felge      glatt
Polo       modern       Rad        4
                        Motor      1
                        Karosse    1
Golf       schnell      Rad        4
                        Reserverad 1
                        Karosse    1
                        Klimaanl   1
                        Motor      1
Karosse    blau
```

```
Klimaanl   robust
KolbRing   rund
Kolben     leicht       KolbRing   2
                        Buchse     1
Motor      schwer       Kolben     6
                        Schraube   8
Rad        rund         Schraube   5
                        Reifen     1
                        Felge      1
Reserverad rund         Schraube   4
                        Reifen     1
                        Felge      1
Reifen     schwarz
Schraube   stabil
!TAB>
onrs OTTONR ! [Motor Polo] # introduction of o++o-numbers (ONR)
RCNT:=firstonr CNT nextonr RCNT pred * CNT at CNT # ONR-Recursion
gib SUPPART,(SUBPART,TOT m) m TOT:=RCNT!++
```

Intermediate result after the first statement:

```
SUPPART,PROPERTY,  (OTTONR, SUBPART, CNT  m) l
Motor   schwer      1        Schraube 8
                    2        Kolben   6
                    2.1      KolbRing 2
                    2.2      Buchse   1
Polo    modern      1        Rad      4
                    1.1      Schraube 5
                    1.2      Reifen   1
                    1.3      Felge    1
                    2        Motor    1
                    2.1      Schraube 8
                    2.2      Kolben   6
                    2.2.1    KolbRing 2
                    2.2.2    Buchse   1
                    3        Karosse  1
```

Intermediate result after applying the recursive o++o number assignment:

```
SUPPART,PROPERTY,  (OTTONR, SUBPART, CNT, RCNT  m) l
Motor   schwer      1        Schraube 8      8
                    2        Kolben   6      6
                    2.1      KolbRing 2     12
                    2.2      Buchse   1      6
Polo    modern      1        Rad      4      4
                    1.1      Schraube 5     20
                    1.2      Reifen   1      4
                    1.3      Felge    1      4
                    2        Motor    1      1
```

```
2.1     Schraube 8     8
2.2     Kolben   6     6
2.2.1   KolbRing 2    12
2.2.2   Buchse   1     6
3       Karosse  1     1
```

final result:

```
SUPPART,(SUBPART, TOT  m) m
Motor    Buchse    6
         Kolben    6
         KolbRing 12
         Schraube  8
Polo     Buchse    6
         Felge     4
         Karosse   1
         Kolben    6
         KolbRing 12
         Motor     1
         Rad       4
         Reifen    4
         Schraube 28
```

9 Generation of images

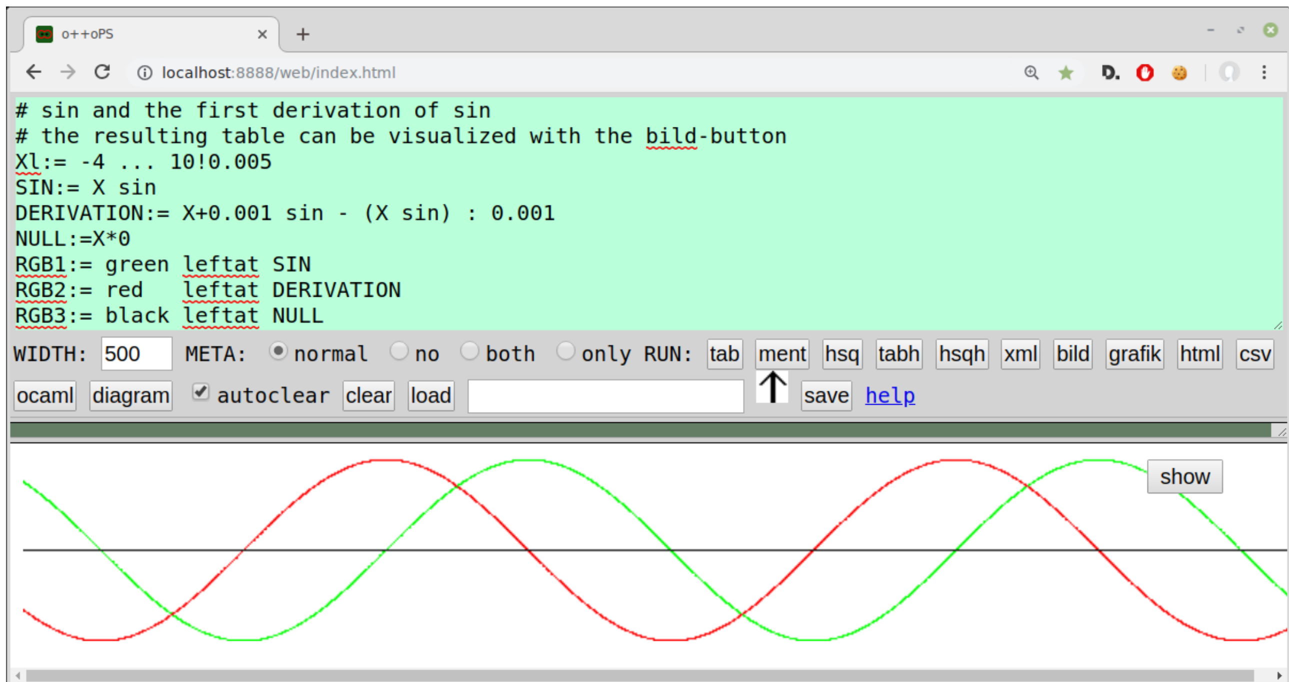
Program 9.1: Plot the points of three functions in multiple colors.

```
# sin and the first derivation of sin
# the resulting table can be visualized with the bild-button
Xl:= -4 ... 10!0.005
SIN:= X sin
DERIVATION:= X+0.001 sin - (X sin) : 0.001
NULL:=X*0
RGB1:= green leftat SIN
RGB2:= red   leftat DERIVATION
RGB3:= black leftat NULL
```

Extract from the result table (without color values) of 2800 points:

```
X,                   SIN,                 DERIVATION,         NULL  l
-4.                   0.756802495308     -0.654021913139     -0.
-3.995                0.75352483081      -0.657796099859     -0.
-3.99                 0.75022832823      -0.661553841711     -0.
-3.985                0.746913069981     -0.665295044752     -0.
-3.98                 0.743579138944     -0.66901961545      -0.
-3.975                0.740226618468     -0.672727460694     -0.
-3.97                 0.736855592364     -0.676418487786     -0.
-3.965                0.73346614491      -0.680092604451     -0.
. . .
```

```
9.96      -0.510032040244   -0.859900243999   0.
9.965     -0.514326423954   -0.857337195738   0.
9.97      -0.518607949529   -0.854752714092   0.
9.975     -0.522876509934   -0.852146863673   0.
9.98      -0.527131998452   -0.849519709627   0.
9.985     -0.531374308698   -0.846871317632   0.
9.99      -0.535603334614   -0.844201753898   0.
9.995     -0.539818970475   -0.841511085165   0.
```



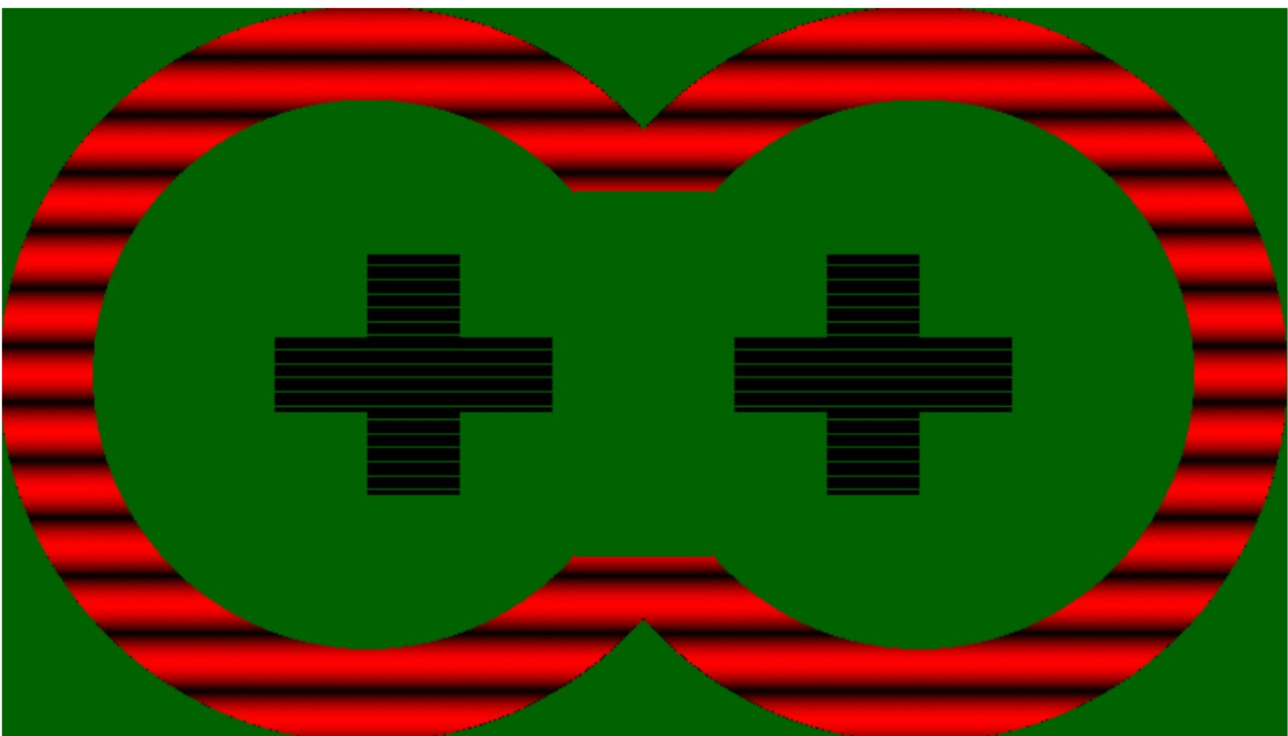
Except for the color values, nothing else is done here than setting a point three times for each of the 2800 elements. Every student learns this procedure – to set up a table of function values. Here this simple method already leads to a visualization. Although only individual points are given here, one has the impression of function course. With suitable o++o programs, not only individual functions but also entire images, such as flags and random color gradients, can be generated with the image button.

Program 9.2: Generation of an o++o logo:

```
RGB:=darkgreen
X3l:= -2 ... 5!0.02
Y3l:= -2 ... 2!0.02 at X3
=: $HINTERGRUND
aus empty_t
Xl:= -2. ... 1.5!0.02
Y1:= 4 - (X*X) sqrt
Y2:=if X< -1.5 then 0 else if X >1.13 then 1. else  2.25 - (X*X) sqrt
Zl:= Y2 ... Y1!0.02 at Y2
RGB:=10*Z cos abs,,1,,0 sin abs leftat Z
STUFE:=if -0.5 <X & X< 0. | (0.5<X & X<1.) then 0.2 else (if -0. <X & X<
0.5 then 0.65)
```

```
Wl := 0 ... STUFE!0.01 at STUFE
RGB:=black leftat W #1,,10*W*X sin abs,0  leftat W
=: $OBENLINKS
$OBENRECHTS:= $OBENLINKS *(-
1,1,1,1,1,1,1,1,1,1,1,1)+(3,0,0,0,0,0,0,0,0,0,0,0)
$UNTENLINKS:=$OBENLINKS * (1,-1,-1,-1,1,1,1,-1,1,1,1,-1)
$UNTENRECHTS:=$OBENRECHTS*(1,-1,-1,-1,1,1,1,-1,1,1,1,-1)
aus $HINTERGRUND,$OBENLINKS,$OBENRECHTS,$UNTENRECHTS,$UNTENLINKS
,<TAB!
X,  Y l
-30 -30
!TAB>
```

The logo can be generated again with the image-button:



The program currently needs a relatively long time (more than a second), so you should only do such a calculation with a local server. Refining the step sizes leads to longer processing time. By making small modifications in the program, you get a variety of changes in the picture. In section 12 (o++o for school) another example (12.16) for the generation of images is given.

10 Diagrams

Program 10.1: Calculate the average BMI per age and the BMI per person and age for all people over 20 as well as the overall average.

```
<TAB!
NAME,      LENGHT,  (AGE,   WEIGHT  l) l
Klaus      1.68      18     61
                     30     65
                     56     80
Rolf       1.78      40     72
Kathi      1.70      18     55
                     40     70
Walleri    1.00       3     16
Viktoria   1.61      13     51
Bert       1.72      18     66
                     30     70
!TAB>
avec NAME! 20<AGE
gib BMI,(AGE,BMI,(NAME,BMI m) m) BMI:=WEIGHT:LENGHT:LENGHT!++:
rnd 1
```

The TAB brackets indicate that the enclosed data correspond to the TAB representation. The above condition selects person records, i.e. `NAME,LENGTH,(AGE,WEIGHT l)`-tuples (structured tuples or struples). Since a person has several `AGE` information, it must be quantified.

```
NAME! 20 <AGE
```

therefore selects all persons who have a corresponding age entry. That is, the existential quantifier is not written, but belongs to every condition. In this small example, you could of course also make the selection by hand.

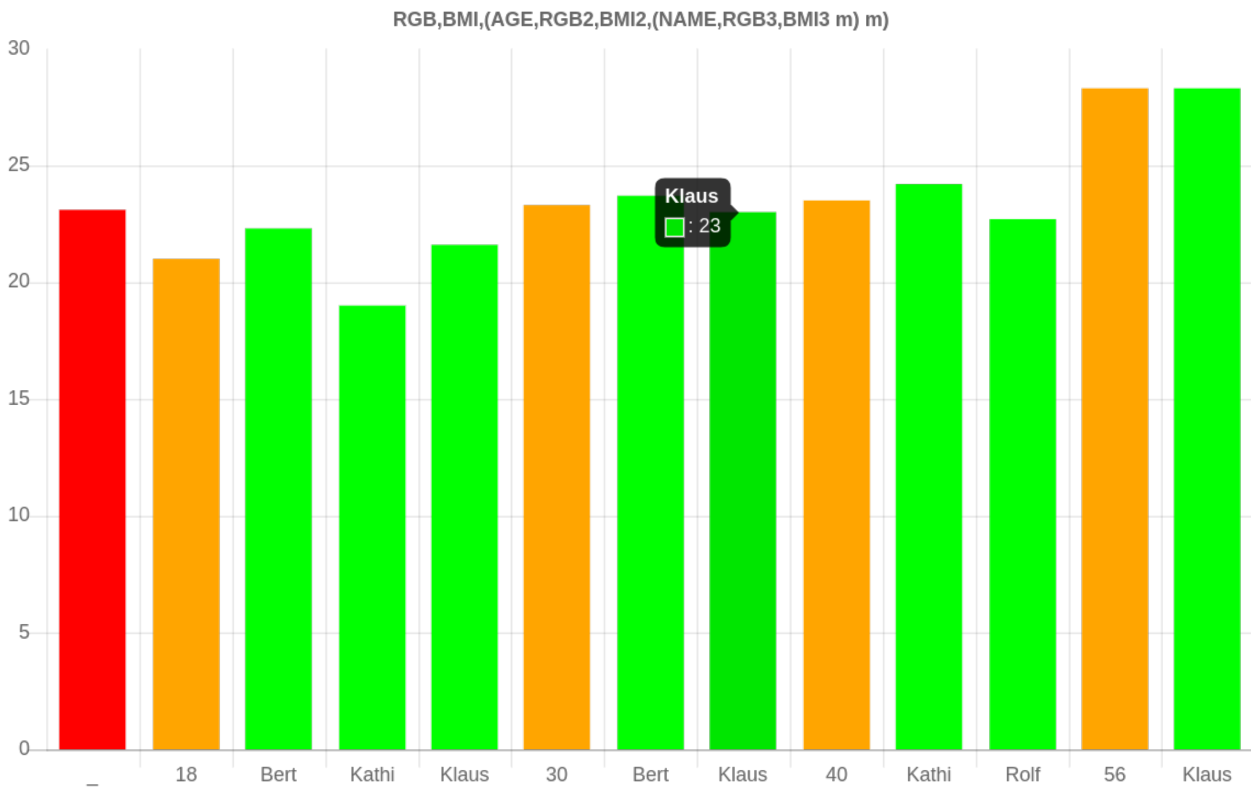
Result (tab)**:**

```
BMI, (AGE,   BMI2, (NAME, BMI3  m) m)
23.1  18     21.    Bert  22.3
                    Kathi 19.
                    Klaus 21.6
      30     23.3   Bert  23.7
                    Klaus 23.
      40     23.5   Kathi 24.2
                    Rolf  22.7
      56     28.3   Klaus 28.3
```

The example shows that you can reverse a hierarchy simply by specifying the desired scheme. As a result, the name is subordinate to the age.

The result can be displayed as a diagram chart with two clicks, for example. To do this, we assign a certain color to each level. The last line is required so that the age is not shown as a column but as a signature.

```
RGB:=red leftat BMI
RGB2:=orange at AGE
RGB3:=green at NAME
AGE::=AGE wort
```

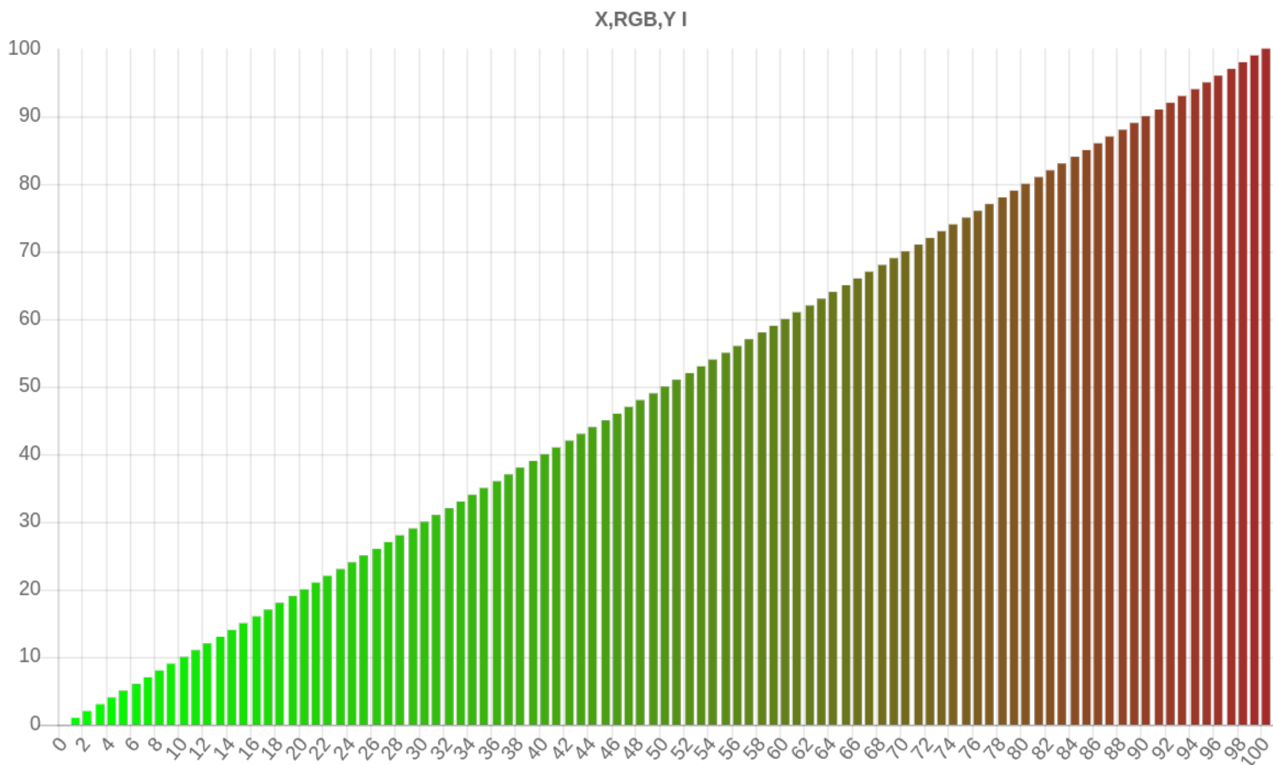


Here you can see the advantages of diagrams based on structured tables.

1. The superordinate columns can be highlighted (zero-th level red; super-ordinate first level orange, second level green)
2. The column names are shorter and clearer, since the values of the super-ordinate columns do not have to be repeated (instead of Bert30 and Klaus30, for example, only Bert and Klaus have to be output)

Program 10.2: A graph with a gradient from green to brown.

```
Xl:= 0 ..100
Y:= X at X
RGB:= (green*(100-X) :100)+ (brown *X :100) leftat Y
X::=X wort
```



11 "Hello otto" - gimmick

Program 11.1: Output two words.

```
Hallo otto
```

The result is a list of two words (WORTl).

Program 11.2: Output a pair of two words.

```
Hallo, otto
```

The result is a pair of 2 words (`WORT,WORT`). The comma separates components of tuples. A tuple (the pair is a 2-tuple) is usually output horizontally and elements of simple collections (lists, sets, multisets) are output vertically in `tabh-` format. In order to save space, simple collections are also given out horizontally with tabh-button.

Program 11.3: Output a text with spaces.

```
"Hallo otto"
```

The result is of the type (scheme) TEXT.

Program 11.4: Connect two words.

```
Hallo + otto
```

The result is a word. `+` also serves as a connection operation (concatenation) of texts or words if both input values are words or texts.

Program 11.5: Give out a greeting with a list of words.

```
GREETING:=[Hallo otto]
```

The result is a list of two words.

Program 11.6: Output two words with two column names.

```
DEAR:=Hallo
GREETING:=otto
```

The result is a pair (2-tuple) of single-column tables. The scheme is: DEAR, GREETING

Program 11.7: Output text with a tag (column name).

```
GREETING:=“Hallo otto“
```

Result:

```
GREETING
Hallo otto
```

Program 11.8: Sort some words.

```
GREETING:={otto Hallo}
```

Result in ment-format:

```
<GREETING>
Hallo
otto
</GREETING>
```

Program 11.9: Sort some words.

```
GREETINGm:= otto Hallo
```

Result in ment-Format:

```
<TABM>
  <GREETING>Hallo</GREETING>
  <GREETING>otto</GREETING>
</TABM>
```

Sets and bags are always sorted; the order of the elements remains the same for lists.

12 o++o for School

Federal Chancellor of Germany A. Merkel has repeatedly stated: “In addition to reading, computing and writing, every student should also learn to program.” Programming skills are an essential aspect of the requirements that the modern information society places on students for participation in society and in future professional life. We are of the opinion that o++o as an easy-to-use table-oriented programming language is the right key for this.

o++o dispenses with loops. Nevertheless, o++o is very expressive. You can use it to do not only compact queries but also a variety of calculations for structured tables and structured documents.

o++o uses and teaches many math concepts, so we see the main benefits of teaching in math class, just as the essential skills for using the calculator are covered in math class. o++o uses the following concepts in particular: set, multi-set, list, equality and inclusion relationships of these; tuple; powerful operations for selecting; calculations; restructuring, sorting and aggregating (sum; average; …) etc.

Spreadsheet programs such as EXCEL and the database standard query language SQL, on the other hand, do not have any structured table schemes. In this difference lies the decisive higher benefit of *o++o*.

Initial tests with preschool children suggest that computations with structured tables are easier than with decimal numbers. We want to add more o++o sample programs. The first two could be interesting for the lower grades.

Program 12.1: Calculate 4 times 3 with unary numbers (|).

```
CHILDl := Ernst Clara Ulrike Sophia
APPLEl := | mal 3 at CHILD
++1 APPLE
```

Instead of `| mal 3` you can also write `[| | |].` That might be better at first, but you would have problems with larger numbers. The result is 12. What is particularly interesting, however, is the intermediate result in `tabh`-format after the first two program lines:

```
CHILD, APPLEl l
Ernst  | | |
Clara  | | |
Ulrike | | |
Sophia | | |
```

You can see here that the multiplication represents the area of a rectangle, which is no longer the case with the decimal number multiplication.

Program 12.2: Calculate (3 + 4) * 5 using unary numbers.

```
Xl:= | mal 3
Xl:= | mal 4
Yl:= | mal 5 at X
++1 Y
```

Result:

```
35
```

Program 12.3: Find X if X times X is 25.

```
Xl:= 1 .. 10
XMALX:= X*X
```

Result:

```
X, XMALX  l
 1   1
 2   4
```

```
 3   9
 4  16
 5  25
 6  36
 7  49
 8  64
 9  81
10 100
```

Analogous to the logarithm table, the pupil can now read (select) the result. The selection can then be implemented later by o++o:

```
avec XMALX = 25
```

or

```
avec XMALX <= 25
last
```

This approach can be applied to a wide variety of problems.

Program 12.4: Find the value of a simple term.

```
2*3+4
```

Result:

```
10
```

* and + each have 2 input values. First, 2 * 3 (6) is calculated. The 6 is the first input value of +, so that a total of 10 results. So here it is simply calculated from left to right.

Program 12.5: Write the term $\cos^3(\sin^2(3.14159))$ in o++o.

```
pi sin hoch 2 cos hoch 3
```

Result:

```
1.
```

In our opinion, the starting term is difficult to read for the average consumer. You start with `pi` and go left to `sin`; then right to the power of 2; now you move back to the left to `cos` and finally to the right to power 3. This notation was probably introduced to save brackets. In order to be unambiguous, the initial term should actually look like this:

$$(\cos((\sin(3.14159))^2))^3$$

That is certainly even more difficult to read and you move even more from left to right and vice versa.

Program 12.6: Write the term $\sin^2(x)+\cos^3(y)$ in *o++o*.

```
X sin hoch 2 + (Y cos hoch 3)
```

Result:

```
Empty_t
```

You could write all terms in o++o without brackets, but then certain terms would have to be written on multiple lines and assignments would have to be used.

No result appears because X and Y have no value. This can be changed, for example, in the following way:

```
X:=2
Y:=3
Z:=X sin hoch 2 + (Y cos hoch 3)
```

Result:

```
X, Y, Z
2  3  -0.14345512749
```

Program 12.7: How to calculate the term 2+3:4*5?

```
2+(3:(4*5))=2 3/20
2+((3:4)*5)=5 3/4
o++o: ((2+3):4)*5=2+3:4*5=6 1/4
```

One recognizes in particular that one does not get along with the school wisdom “point calculation goes before line calculation” yet. You need the rule “from left to right” in addition.

Program 12.8: Calculate the average of several marks.

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

Result:

```
1.8
```

From a methodological point of view, this program can be improved by adding the brackets for lists:

```
[1 2 3 1 2] ++:
```

You can now see that the averaging operation `++:` has an input value, namely a list, and that `++:` follows the input value. Since users usually do not want to type much, we assume that the first notation will be used more often in practice.

Program 12.9: Calculate the averages of the structured table `marks.tabh` for each subject.

```
marks.tabh
TOT:=MARKl ++:
```

`marks.tabh` could look like this:

```
SUBJECT,MARKl l
Ma      1 2 1 3 1 2
Phy     4 3 2 2 1
```

Here `l` abbreviates list. That is, `marks.tabh` is a structured table (list) that contains a list of MARKs for each subject.

The result of the request again in "tabh-format":

```
SUBJECT, TOT,          MARKl  l
Ma       1.66666666667 1 2 1 3 1 2
Phy      2.4           4 3 2 2 1
```

Program 12.10: Form the sum of the numbers from 1 to 100 (task from Gauss class 5).

```
1 .. 100 ++
```

Like addition and multiplication, ".." has two input values (1 and 100). The following list is created as an intermediate result:

```
ZAHLl
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100
```

These numbers are then added up, so that `5050` results.

Program 12.11: Approximately calculate the maximum of the sine function in the interval [1,2].

```
1 ... 2!0.001 sin max
```

"..." requires 3 input values: the start value 1, the end value 2, and the increment 0.001. This results in the numbers `1 1.001 1.002 1.003`… `1.999 2`.

The sine function is applied to each of the numbers, resulting in 1001 numbers again. The function `max` then finds the maximum from this list.

Although this is an approximation method, the exact value 1 comes out as the step size is further refined. `sin` and `max` each have an input value (here a list) but the output value of `sin` is again a list and `max` only generates one number, since this is an aggregation function. The second and third input values of a ternary operation ("…" in the example) are each separated by a "!". This is necessary in o++o, because the comma is already used for pair-operation and the empty-space already separates list elements.

Program 12.12: Approximately compute the minimum of the polynomial „$X^3 + 4X^2 - 3X+2$“ in the interval [0, 2]. Output also the associated X value.

```
Xl:= 0 ... 2!0.001
Y:= X poly [1 4 -3 2]
MINI:= Yl min
avec Y = MINI
```

`avec` is French and describes a selection. A concrete polynomial of a variable X always has only one input value that is used for X. `poly` is more general and has 2 input values:

1. The input value for X, which here accepts all numbers that were generated in the first line.
2. A list of numbers that correspond to the coefficients of the particular polynomial.

The first line creates a list of numbers, all of which have been given the name X. This is best seen in the xml- or ment-representation:

```
<X>0.</X>
<X>0.001</X>
<X>0.002</X>
…
```

Result (tab)**:**

```
MINI,      (X,    Y        l)
1.481482037 0.333 1.481482037
```

Program 12.13: Approximately calculate a null of the cosine function in the interval [1, 2].

```
Xl:= 1 ... 2!0.0001
avec X cos < 0
avec X pos = 1
```

Result:

```
Xl
1.5708
```

After the first selection, only the `X` values with a function value less than `0` remain. From these, the first value is selected in the second step. Since we know that `cos` has only one null in the considered interval, this is approximated by the result.

Program 12.14: Given 5 annual growth numbers, calculate total growth. Round the result to one digit after the decimal point.

```
Wl:= 0 1.5 2.1 1.3 0.4 1.2
ACCU:= first 100.  next ACCU pred *(W:100+1)  at W
rnd 1
```

Result (tab)**:**

```
W,  ACCU l
0.  100.
1.5 101.5
2.1 103.6
1.3 105.
0.4 105.4
1.2 106.7
```

The first `ACCU` value results from the expression after `first` (100.). For the second value, the value `100` is used for `ACCU pred` and the term after `next` is evaluated. It results in `101.5`. This number is used again in `ACCU pred` and the next term is calculated again (around `103.6`), ... until the last `W` value is reached. `pred` is the predecessor.

Program 12.15: Calculate the area under the sine curve in the interval [0, pi] approximately.

```
0 ... pi!0.0001 sin *0.0001 ++
```

Result:

```
1.99999999867
```

All numbers between `0` and `pi` are generated one after the other, then the sine of each number is calculated and then each number is multiplied by `0.0001`. 10000 rectangular areas are created, which are then added up.

Program 12.16: Pascal's triangle:

```
Xl:=1 ..9
Y:=first 1  next Y pred,0 + (0,Y pred)  at X
```

Result (hsq):

```
X,Y l
X Y
1 1
2 1 1
3 1 2 1
4 1 3 3 1
5 1 4 6 4 1
6 1 5 10 10 5 1
7 1 6 15 20 15 6 1
8 1 7 21 35 35 21 7 1
9 1 8 28 56 70 56 28 8 1
```

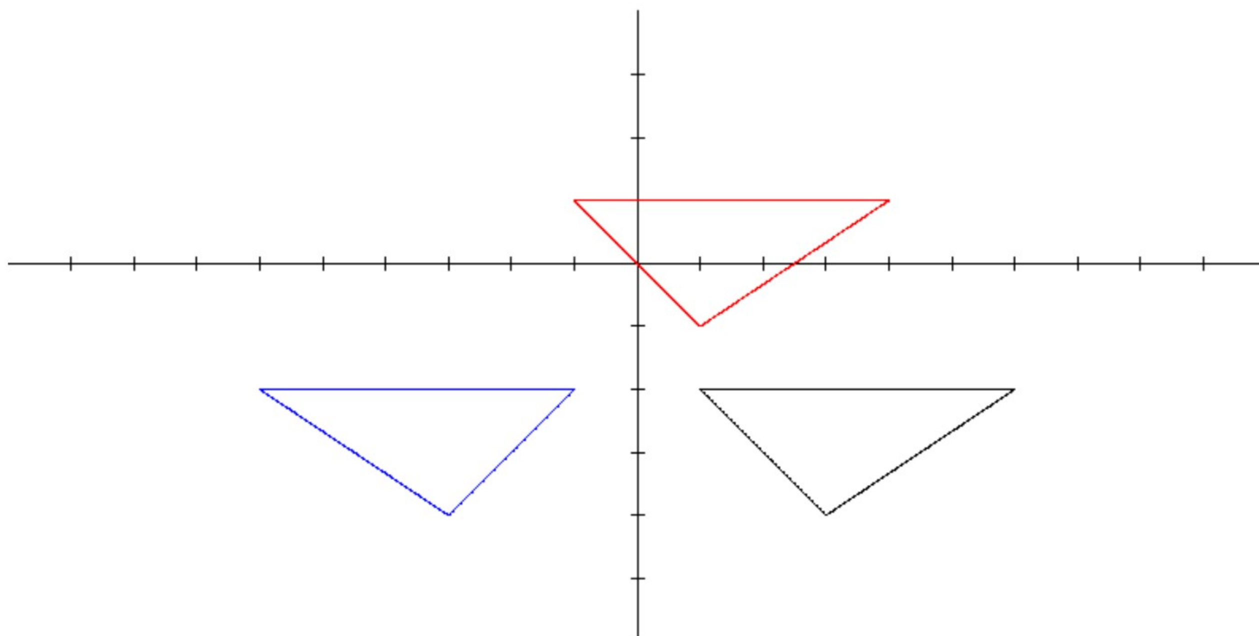
Here Y represents a tuple of values. The result is therefore no longer a normal table. It can therefore not be output in tab-format.

Program 12.17: Move and mirror a triangle

```
# Class 7 p.14 No. 5 (red) secondary school Saxony-Anhalt
# Math textbook
# please click bild
Xl:= -10 ... 10!0.02
Y:=0*X
X0l:= 0
Y0l:= -6 ...4!0.02 at X0
X1l:= -9 .. 9
Y1l:= -0.1 ... 0.1!0.01 at X1
X2l:= -0.1 ... 0.1!0.01
Y2l:= -5 .. 3 at X2
=: $KOORDINATEN
aus <TAB!
X3,Y3 l
1  -2
3  -4
6  -2
1  -2
```

```
!TAB>
route
=: $DREIECK
aus $KOORDINATEN ,$DREIECK
RGBROT:=red
,$DREIECK +(-2,3) # move
RGBBLAU:=blue
,$DREIECK *(-1,1) # mirror on Y-axis
```

Result (bild-button)**:**



The above examples make it particularly clear that the tasks can be solved without knowledge of differential and integral calculus. With *o++o*, math lessons can be supported in a variety of ways. This ranges from grade 7 or lower to grade 12. It concerns: Calculating with natural numbers, decimal numbers, rational numbers of any size, approximate calculation of nulls of any functions, derivations, areas under curves, extreme values (can already be taught in secondary school), probability calculation etc. With *o++o* things can be calculated in a simple way that are otherwise only dealt with theoretically. As a result, the understanding of the concepts can be significantly improved, expanded and deepened. Further information on *o++o* can be found at ottops.de.

We believe that *o++o* offers special advantages for mathematics and computer science lessons but can also be used meaningfully in the other subjects (queries to *Wikipedia*).

13 Closing words, a quote from Adam Ries

Ein mensch dem zahl (tabment)[1)] verborgen ist
Leichtlich der verführt wird mit list
Das nimm zu hertzen bitt ich sehr
Und jeder sein Kind rechnen (programmieren)[1)] lehr ...

Adam Ries
[1)] Added by the author

14 Literature

[AB84] S. Abiteboul, N. Bidot, „Non First Normal Form Relations: An Algebra Allowing Data Restructuring“ Rapports de Recherche No347, Institute de Recherce en Informatique et en Automatique, Rocquencourt, France, Nov. 1984

[AC75] M. M. Astrahan, D. D. Chamberlain, „Implementation of a Structured English Query Language“, Communications of the ACM 18 10, Oct. 1975 pages 580-587

[BCFFRS10] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu,Jonathan Robie, Jérôme Siméon, XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation 14 December 2010 (Link errors corrected 3 January 2011), http://www.w3.org/TR/2010/REC-xquery-20101214/

[Ben80] K. Benecke, „Signaturketten und Operations- und Mengenformen über Signaturketten“, Dissertation A, TH Magdeburg 1980

[Ben82] K. Benecke. “AFUL- Eine Anfragesprache für Datenbanken”. In Weiterbildungszentrum für mathematische Kybernetik und Rechentechnik/Informationsverarbeitung, Studientexte Datenbanken TU Dresden Heft 59, pages 100 – 107, 1982.

[Ben88] K. Benecke, "Hierarchische Datenstrukturen", Habilitation, Technische Hochschule

Magdeburg, 1988

[Ben91] K. Benecke, "A powerful Tool for Object-Oriented Manipulation", in "Object-Oriented Databases: Analysis, Design & Construction (DS-4) R.A. Meersmann, W. Kent, S. Khosla (editors), Elsevier Science Publisher B.V. (North Holland) 1991, S. 95-121

[Ben98] K. Benecke, “Strukturierte Tabellen – Ein neues Paradigma für Datenbank- und Programiersprachen”, Deutscher Universitätsverlag, ISBN 3-8244-2099-6 Wiesbaden 1998

[Ben16] K. Benecke, „o++oPS The simplest Programing Language“, BoD, 2016, ISBN 978-3-7412-4281-6

[BH11] K. Benecke, A. Hauptmann, “Does the School Need a Tabular Computer Language?” http://www.infonomics-society.org/IJDS/Does%20the%20School%20Need%20a%20Tabular%20Computer%20Language.pdf, pages 520-527, 2011.

[Cod70] E.F.Codd, „A Relational Model of Data for Large Shared Data Banks“, Communications of the ACM, Vol. 13, No. 6 June 1970, S. 377-387

[Hau10] A. Hauptmann, “OttoQL: Probleme der Implementation nichtrelationaler Datenbanksprachen (mit besonderer Berücksichtigung der logischen Optimierung)”. Studienarbeit, Uni Magdeburg, benecke-systeme, 2010.

[KR71] H. Kaphengst, H. Reichel, „Algebraische Algorithmentheorie“, VEB Robotron, Wiss. Informationen und Berichte, Nr. 1/71 Reihe A, Sommer 1971

[Rei87] Reichel. H., Initial Computability, Algebraic Specifications, and Partial Algebras, Oxford

UK, Claredon Press, 1987

[Rei90] W. Reichstein. “Implementation der Strichlistenoperation Operation stroke in C”, Praktikumsbeleg, VE CS Magdeburg, 1990.

[Sch96] D. Schamschurko, „Implementation des Rumpfes des GIB-AUS-MIT-Konstrukts in CAML-Light“, Praktikumsbeleg, DeTeCSM Magdeburg, Betreuer: K. Benecke, März 1996, 123 Seiten

[SHL75] N. C. Shu, B. C. Housel, V. Y. Lum, „CONVERT- A High Level Translation Definition Language for Data Conversion“, Communications of the ACM, Vol.18 Nr.10,Oct., 1975, S. 557-567

[Ull82] J. D. Ullman, „Principles of Database Systems“, Computer Science Press, Rockville, Maryland 1982

[Zem85] H. Zemanek, „Formal Definition the Hard Way“, Proc. IFIP TC2 Working Conference, Wien 1985; North Holland, S. 411-417