Handling of the arithmetic expressions

The text structure of one formula or calculation is the following.

```
GRAPHIK:
ID=>3
NAME=>"Demo"
DATEI=>"Grphexmpl2.tmp"
PARAMETER=>3
RESOLUTIONS=>1 0.5 0.75 0
FORMEL=>
if (p_0 neq 3) { Fehler } else { p_1 + p_2 - p_3 }
```

First comes the identification number of the formula. The user interface enables the user to select such an id within the existing set of values, and will go through the list and execute each code with the same ID. Therefore several calculations
Then a name is given (free value), and eventually a file name (both are unused in our case) that was thought to hand over a description of the page in which the values are presented.
The next parameter states the number of parameters (used under the name p_ in the code) used by the code. This value is passed by the user interface into parameter p_0.
The resolution field states the step values with which the values of the parameter p_x are incremented or decremented using the + or - keys. The values for all 4 parameters must be present. Those values are only used if the macro SPRACHE is not defined. Otherwise, another scheme is used to set the values of the parameters. First the parameter has to be select with the values 1 to 4. Then the integer part is set with the keys 0-9 and -. The type the . key and set the fractional part with keys 0-9.
The formal field specifies the code compute the value. If one graphic identification number is assigned to more than one formula or calculation, the result values are calculated and displayed on the console in the order in with they are listed in the text file.

At the start of the input file, several values have to be specified for the size of stack and buffers.
SPEICHER_KODE:1000
SPEICHER_STACK:200

Those are in the order:
temporary buffer in which the code of the formula/calculation are stored when read in from the file. This code is then copied into an appropriate memory space.
The second value is not used yet. In future I will implement a size which will be the maximum number of variables in a calculation. When parsing the code, the variables are stored into a buffer that is later copied into a appropriately sized memory area.
Now a buffer that can containt 50 variables is allocated.
Per default a variable named Fehler is created that has a value of NAN.

For instance, 20 graphic formulas can be stored. This is defined by the MAXGRAPHS compiler define in file Parser.h. Later versions may take it from the text input file.

For the formula / calculations, different control structures are provided

if (...) { .... } else { ... }
<Variable Name> = ... ;
When parsing the formula, the variables are stored in a table. This table will be called variable table in the following document. When the code is executed, the value calculated between the following the = is assigned to this variable. When a variable is defined the first time a value has to be assigned.
Per default a variable named "Fehler" is defined, and its value set to NAN (Not A Number). If an operation returns this value, the execution is stopped and a message is displayed in the console.
while (…) { … }
do { … } while (…)
for (<index name>= ..; <index name> <><=>= .. ; ++/--<index name>--/++) { … }

The last ; before the } character ending a control structure path is optional as the parser will end the control path anyway when finding the } character.

Input parameters for the calculation

In total 5 input parameters are handled by the user interface (the underlying software loading and executing the calculations can handle more). The first one p_0 is set to the number of parameters given in the field PARAMETER.
The 4 other input parameters name p_1 to p_4 can be set by the user to any floating point value using the two input schemes described above.

Storage of the calculation formulas

The code is stored into a tree structure of objects that are all derived from a common parent class. They all provide a calculate routine that returns the result of the execution of the corresponding code.
For each type of argument (constant, input parameter or variable) and operation (unary/binary operator function call or variable operation).

First we start with a simple example:

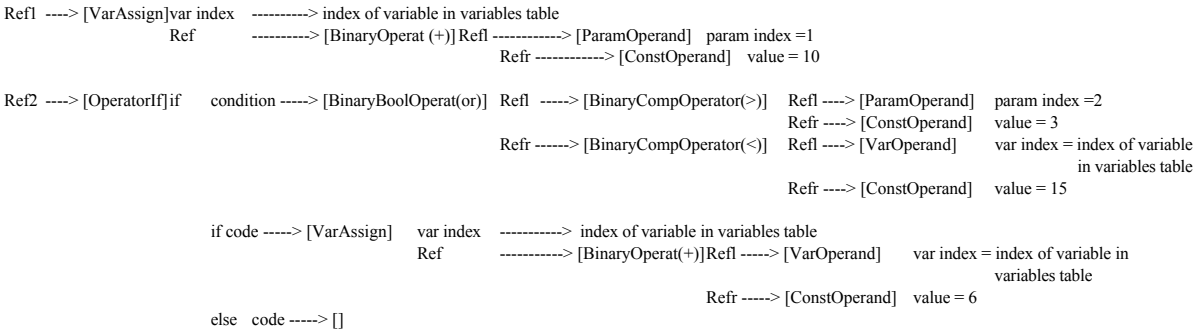
```
variable = p_1 + 10;
if (p_2 > 3 or variable < 15) {
        variable = variable + 6
} else {

}
```

For this calculation the variable table will contain the following data:

index	variable name	value
0	Fehler	NAN
1	variable	p_1+10 (+6)

The main code will contain 2 references (one per line of code / control structure):

per convention we use the following notation: Refx-------> [class of object] attribute 1 -------> explanation / ref to another class with attributes
…
attribute n -------> "



For arithmetic expressions:

The global parent class is:
GeneralOperand

The derived classes are:
ConstOperand : holds an attribute that contains the value of the constant. The calculate method simply returns this value.
ParamOperand : holds an attribute that contains the index of the input parameter. Its calculate routine gets the value at this index in the array of input parameters and returns it.
UnaryOperat (-) : holds an attribute that references the structure of objects that returns the value of the argument, The calculate routine calls the calculate routine of this reference, that applies its operator to the value before returning the result.
FunctOperat : holds an attribute that references the structure of objects that returns the value of the argument, and a reference to the appropriate library function. The calculate routine calls the calculate routine of this reference, that applies its function to the value before returning the result.
BinaryOperat (- + * /) : holds two attribute that references the structure of objects that return the value of the left and right argument, and a reference to the corresponding operator. The calculate routine calls the calculate routine of this reference, that applies its
VarAssign : holds an attribute that is the index of the variable in the variables table (containing variable name and its value), and a reference to the structure of objects that returns the value of the code whose value is to be assigned to the variable.
VarOperand : holds an attribute that is the index of the variable in the variables table. The calculate routine returns the value stored at this index in the variables table.

OperatorIf : holds one reference to the structure that implements the code of the condition and two arrays with counter of items in each array. Those array contain the references to the implementation of the code of the if and else branch. Every "line of code" (terminated by a ; or the last one before the closing } (in this case the; is not necessary as the parser ends the line when reaching the } character). A control structure as if else, for or while loops also leads to one reference.
The size of the arrays is initially 5 (given by #define OPERAND_BLOCK in OperandClasses.cpp) and increased by the same size every time it is necessary.
OperatorFor: holds 3 references and one array with counter of items in this array. This array contain references to the implementation of the code of the loop. Each reference points to the implementation of one line of code as in the if operator. The size is also initially 5 and extended as necessary as in the if operator.
The 3 references are:
- once reference to the code setting the loop initial condition,
- one reference to the structure that implements the code of the condition,
- one reference to the code additionally performed on each loop iteration. This mainly contains variable incrementations or decrementations.
OperatorIncDec : incrementas or decrements a variable's value. Used in for loops.
OperatorWhile : holds one reference to the structure that implements the code of the condition and one array with counter of items in this array that implements the code in the loop.
OperatorDo : same as the while operator. Only the execution calls the code of the condition at the end of the loop.

The branches

For boolean expressions: the same concept has been implemented:

Parent class
GeneralBoolOperand

In general, all values in the calculations are of type double. If the value is 0, it is considered false otherwise true.

The derived classes are:
UnaryBoolOperat (not) : holds a reference to the code implementation that calculates the value from which the boolean complement must be returned.
BinaryBoolOperat (or and xor) : holds two attribute that references the structure of objects that return the value of the left and right argument, and a reference to the corresponding operator.
BinaryCompOperator (< > >= >= !=) : holds two attribute that references the structure of objects that return the value of the left and right argument, and a reference to the corresponding operator.