version 1.1 Jan 2015

Arbitrary Precision Numbers

Petr Olšák

ftp://math.feld.cvut.cz/olsak/makra/

Table Of Contents

1 User's Documentation

This macro file `apnum.tex` implements addition, subtraction, multiplication, division and power to an integer of numbers with arbitrary number of decimal digits. The numbers are in the form:

```
<sign><digits>.<digits>
```

where optional ⟨*sign*⟩ is the sequence of `+` and/or `-`. The nonzero number is treated as negative if and only if there is odd number of `-` signs. The first part or second part of ⟨*digits*⟩ (but not both) can be empty. The decimal point is optional if second part of ⟨*digits*⟩ is empty.

There can be unlimited number of digits in the operands. Only TeX main memory or your patience during calculation with very large numbers are your limits. Note, that this implementation includes many optimizations and it is above 100 times faster (on large numbers) than the implementation of the similar task in the package `fltpoint.sty`. And the `fp.sty` doesn't implements arbitrary number of digits. The extensive technical documentation can serve as an inspiration how to do TeX macro programming.

1.1 Evaluation of Expressions

After `\input␣apnum` in your document you can use the macro `\evaldef` ⟨*sequence*⟩`{`⟨*expression*⟩`}`. It gives the possibility for comfortable calculation. The ⟨*expression*⟩ can include numbers (in the form described above) combined by `+`, `-`, `*`, `/` and `^` operators and by possible brackets `()` in an usual way. The result is stored to the ⟨*sequence*⟩ as a literal macro. Examples:

```
\evaldef\A {2+4*(3+7)}
   % ... the macro \A includes 42
\evaldef\B {\the\pageno * \A}
   % ... the macro \B includes 84
\evaldef\C {123456789000123456789 * -123456789123456789123456789}
   % ... \C includes -15241578765447341344197531849955953099750190521
\evaldef\D {1.23456789 + 12345678.9 - \A}
   % ... the macro \D includes 12345596.13456789
\evaldef\X {1/3}
   % ... the macro \X includes .33333333333333333333
```

The limit of the number of digits of the division result can be set by `\apTOT` and `\apFRAC` registers. First one declares maximum calculated digits and second one declares maximum of digits after decimal point. The result is limited by both those registers. If the `\apTOT` is negative, then its absolute value is treated as a "soft limit": all digits before decimal point are calculated even if this limit is exceeded. The digits after decimal point are not calculated when this limit is reached. The special value `\apTOT=0` means that the calculation is limited only by `\apFRAC`. Default values are `\apTOT=-30` `\apFRAC=20`.

`\evaldef`: 3–6, 8–9, 11   `\apTOT`: 2–3, 6, 21, 30   `\apFRAC`: 2–3, 6, 21, 30

The operator ^ means the powering, i.e 2^8 is 256. The exponent have to be an integer (no decimal point is allowed) and a relatively small integer is assumed.

The scanner of the \evaldef macro reads something like "operand binary-operator operand binary-operator etc." without expansion. The spaces are not significant. The operands are:

- numbers (in the format ⟨*sign*⟩⟨*digits*⟩.⟨*digits*⟩) or
- numbers in scientific notation (see the section 1.3) or
- sequences ⟨*sign*⟩\the⟨*token*⟩ or ⟨*sign*⟩\number⟨*token*⟩ or
- any other single ⟨*token*⟩ optionally preceded by ⟨*sign*⟩ and optionally followed by a sequence of parameters enclosed in braces, for example \A or \B{⟨*text*⟩} or -\C{⟨*textA*⟩}{⟨*textB*⟩}.

It means that you can use numbers or macros without parameter or macros with one or more parameters enclosed in braces as operands.

The apnum.tex macro file provides the following "function-like" macros which can be used as an operand in the ⟨*expression*⟩: \ABS {⟨*value*⟩} for an absolute value, \iDIV {⟨*dividend*⟩}{⟨*divisor*⟩} for an integer division, \iMOD {⟨*dividend*⟩}{⟨*divisor*⟩} for an integer remainder, \iROUND {⟨*value*⟩} for rounding the number to the integer, \iFRAC {⟨*value*⟩} for fraction part of the \iROUND, \FAC {⟨*value*⟩} for a factorial. The arguments of these functions can be a nested ⟨*expressions*⟩ with the syntax like in the \evaldef macro. Example:

```
\def\A{20}
\evaldef\B{ 30*\ABS{ 100 - 1.12*\the\widowpenalty } / (4+\A) }
```

Note that the arguments of the "function-like" macros are enclosed by normal TEX braces {} but the round brackets () are used for re-arranging of the common priority of the +, -, *, / and ^ operators.

The macro used as an operand in the ⟨*expression*⟩ can be a "literal-macro" directly expandable to a number (like \A above) or it is a "function-like" macro with the following properties:

- It is protected by \relax as its first token after expansion.
- It calculates the result and saves it into the \OUT macro.

1.2 Basic Functions

The apnum.tex macro file provides the \PLUS, \MINUS, \MUL, \DIV and \POW macros (with two parameters). They are internally used for evaluation of the ⟨*expression*⟩ mentioned above. The parameters of these macros can be numbers or another \PLUS, \MINUS, \MUL, \DIV or \POW macro call or another "literal macro" with the number or "function-like" macro as described above. The result of calculation is stored in the macro \OUT. Examples:

```
\PLUS{123456789}{-123456789123456789}
   % ... \OUT is -123456789000000000
\PLUS{2}{\MUL{4}{\PLUS{3}{7}}}
   % ... \OUT is 42
\DIV{1}{3}
   % ... \OUT is .33333333333333333333
```

The number of digits calculated by \DIV macro is limited by the \apTOT and \apFRAC registers as described above. There is another result of \DIV calculation stored in the \XOUT macro. It is the remainder of the division. Example:

```
\apTOT=0 \apFRAC=0 \DIV{12345678912345}{2} \ifnum\XOUT=0 even \else odd\fi
```

You cannot use \ifodd primitive here because the number is too big.

The macro \POW{⟨*base*⟩}{⟨*exponent*⟩} calculates the power to the integer exponent. A slight optimization is implemented here so the usage of \POW is faster than repeated multiplication. The decimal non-integer exponents are not allowed because the implementation of exp, ln, etc. functions would be a future work.

The \SIGN is the TeX register with another output of the calculation of \evaldef, \PLUS, \MINUS, \MUL and \DIV macros. It is equal to 1 if the result is positive, it is equal to −1, if the result is negative and it is equal to 0, if the result is 0. You can implement the conditionals of the type

```
\TEST {123456789123456789} > {123456789123456788} \iftrue OK \else KO \fi
```

by the following definition:

```
\def\TEST#1#2#3#4{\MINUS{#1}{#3}\ifnum\SIGN #2 0 }
```

Note that the arguments of \PLUS, \MINUS, \MUL, \DIV and \POW macros accept their arguments as one single operand, no ⟨*expressions*⟩ (like in \evaldef) are allowed. There is no sense to combine the basic functions \PLUS, \MINUS etc. with binary operators +, -, *, / and ^.

The \ROUND ⟨*sequence*⟩{⟨*num*⟩} rounds the number, which is included in the macro ⟨*sequence*⟩ and redefines ⟨*sequence*⟩ as rounded number. The digits after decimal point at the position greater than ⟨*num*⟩ are ignored in the rounded number. The ignored part is saved to the \XOUT macro. Examples:

```
\def\A{12.3456}\ROUND\A{1}   % \A is "12.3",    \XOUT is "456"
\def\A{12.3456}\ROUND\A{9}   % \A is "12.3456", \XOUT is empty
\def\A{12.3456}\ROUND\A{0}   % \A is "12",      \XOUT is "3456"
\def\A{12.0001}\ROUND\A{2}   % \A is "12",      \XOUT is "01"
\def\A{.000001}\ROUND\A{2}   % \A is "0",       \XOUT is "0001"
\def\A{-12.3456}\ROUND\A{2}  % \A is "-12.34",  \XOUT is "56"
\def\A{12.3456}\ROUND\A{-1}  % \A is "10",      \XOUT is "23456"
\def\A{12.3456}\ROUND\A{-4}  % \A is "0",       \XOUT is "00123456"
```

1.3 Scientific Notation of Numbers

The macros \evaldef \PLUS, \MINUS, \MUL, \DIV and \POW are able to operate with the numbers written in the notation:

```
<sign><digits>.<digits>E<sign><digits>
```

For example 1.234E9 means $1.234 \cdot 10^9$, i.e. 1234000000 or the text 1.234E-3 means .001234. The decimal exponent (after the E letter) have to be in the range $\pm 2\,147\,483\,647$ because we store this value in normal TeX register.

The macros \evaldef \PLUS, \MINUS, \MUL, \DIV and \POW operate by "normal way" if there are no arguments with E syntax. But if an argument is expressed in scientific form, the macros provide the calculation with mantissa and exponent separately and the mantissa of the result is found in the \OUT macro (or in the macro defined by \evaldef) and the exponent is in stored the \apE register. Note, that \OUT is a macro but \apE is a register. You can define the macro which shows the result of the calculation, for example:

```
\def\showE#1{\message{#1\ifnum\apE=0 \else*10^\the\apE\fi}}
```

No macros mentioned above store the result back in the scientific notation, only mantissa is stored. You need to use \apE register to print the result similar as in the example above. Or you can use the macro \addE ⟨*sequence*⟩ macro which redefines the ⟨*sequence*⟩ macro in order to add the E⟨*exponent*⟩ to this macro. The ⟨*exponent*⟩ is read from the current value of the \apE register.

There are another usable functions for operations with scientific numbers.

- \ROLL ⟨*sequence*⟩{⟨*shift*⟩} ... the ⟨*sequence*⟩ is assumed to be a macro with the number. The decimal point of this number is shifted right by ⟨*shift*⟩ parameter, i.e. the result is multiplied by 10^⟨*shift*⟩. The ⟨*sequence*⟩ is redefined by this result. For example \ROLL\A{\apE} converts the number of the form ⟨*mantissa*⟩*10^\apE to the normal number.
- \NORM ⟨*sequence*⟩{⟨*num*⟩} ... the ⟨*sequence*⟩ is supposed to be a macro with ⟨*mantissa*⟩ and it will be redefined. The number ⟨*mantissa*⟩*10^\apE (with current value of the \apE register) is assumed. The new mantissa saved in the ⟨*sequence*⟩ is the "normalized mantissa" of the same number. The \apE register is corrected so the "normalized mantissa"*10^\apE gives the same number.

The ⟨*num*⟩ parameter is the number of non-zero digits before the decimal point in the outputted mantissa. If the parameter ⟨*num*⟩ starts by dot following by integer (for example `{.2}`), then the outputted mantissa has ⟨*num*⟩ digits after decimal point. For example `\def\A{1.234}\apE=0 \NORM\A{.0}` defines `\A` as `1234` and `\apE=-3`. The macros `\PLUS`, `\MUL` etc. don't use this macro, they operate with the mantissa without correcting the position of decimal point and adequate correcting of the exponent.

The following example saves the result of the `\evaldef` in scientific notation with the mantissa with maximal three digits after decimal point and one digit before.

```
\evaldef\X{...}\NORM\X{1}\ROUND\X{3}\addE\X
```

The macros `\ROUND`, `\addE`, `\ROLL` and `\NORM` redefine the macro ⟨*sequence*⟩ given as their first argument. The macro ⟨*sequence*⟩ must be directly the number in the format ⟨*simple-sign*⟩⟨*digits*⟩`.`⟨*digits*⟩ where ⟨*simple-sign*⟩ is one minus or none and the rest of number has the format described in the first paragraph of this documentation. The scientific notation isn't allowed here. This format of numbers is in accordance with the output of the macros `\evaldef`, `\PLUS`, `\MINUS` etc.

1.4 Experiments

The following table shows the time needed for calculation of randomly selected examples. The comparison with the package `fltpoint.sty` is shown. The symbol ∞ means that it is out of my patience.

input	# of digits in the result	time spent by `apnum.tex`	time spent by `fltpoint.sty`
200!	375	0.33 sec	173 sec
1000!	2568	29 sec	∞
5^{17^2}	203	0.1 sec	81 sec
5^{17^3}	3435	2.1 sec	∞
1/17	1000	0.13 sec	113 sec
1/17	100000	142 sec	∞

2 The Implementation

First, the greeting. The `\apnumversion` includes the version of this software.

apnum.tex

```
\def\apnumversion{1.1 <Jan. 2015>}
\message{The Arbitrary Precision Numbers, \apnumversion}
```

We declare auxiliary counters and one boolean variable.

apnum.tex

```
\newcount\apnumA \newcount\apnumB \newcount\apnumC \newcount\apnumD
\newcount\apnumE \newcount\apnumF \newcount\apnumG \newcount\apnumH
\newcount\apnumO \newcount\apnumL
\newcount\apnumX \newcount\apnumY \newcount\apnumZ
\newcount\apSIGNa \newcount\apSIGNb  \newcount\apEa \newcount\apEb
\newif\ifapX
```

Somebody sometimes sets the `@` character to the special catcode. But we need to be sure that there is normal catcode of the `@` character.

apnum.tex

```
\apnumZ=\catcode`\@ \catcode`\@=12
```

2.1 Public Macros

The definitions of the public macros follow. They are based on internal macros described below.

apnum.tex

```
\def\evaldef{\relax \apEVALa}
\def\PLUS{\relax \apPPab\apPLUSa}
\def\MINUS#1#2{\relax \apPPab\apPLUSa{#1}{-#2}}
\def\MUL{\relax \apPPab\apMULa}
\def\DIV{\relax \apPPab\apDIVa}
\def\POW{\relax \apPPab\apPOWa}
```

`\apnumversion`: 5

```
\def\ABS{\relax \apEVALone\apABSa}
\def\iDIV{\relax \apEVALtwo\apiDIVa}
\def\iMOD{\relax \apEVALtwo\apiMODa}
\def\iROUND#1{\relax \evaldef\OUT{#1}\apiROUNDa}
\def\iFRAC{\relax \apEVALone\apiFRACa}
\def\FAC{\relax \apEVALone\apFACa}
\def\ROUND{\apPPs\apROUNDa}
\def\ROLL{\apPPs\apROLLa}
\def\NORM{\apPPs\apNORMa}
\def\addE#1{\edef#1{#1\ifnum\apE=0 \else E\ifnum\apE>0+\fi\the\apE\fi}}
```

The `\apSIGN` is an internal representation of the public `\SIGN` register. Another public registers `\apE`, `\apTOT` and `\apFRAC` are used directly.

apnum.tex

```
\newcount\apSIGN  \let\SIGN=\apSIGN
\newcount\apE
\newcount\apTOT   \apTOT=-30
\newcount\apFRAC  \apFRAC=20
```

2.2 Evaluation of the Expression

Suppose the following expression `\A+\B*(\C+\D)+\E` as an example.

The main task of the `\evaldef\x{\A+\B*(\C+\D)+\E}` is to prepare the macro `\tmpb` with the content (in this example) `\PLUS{\PLUS{\A}{\MUL{\B}{\PLUS{\C}{\D}}}}{\E}` and to execute the `\tmpb` macro.

The expression scanner adds the `\end` at the end of the expression and reads from left to right the couples "operand, operator". For our example: `\A+`, `\B*`, `\C+`, `\D+` and `\E\end`. The `\end` operator has the priority 0, plus, minus have priority 1, `*` and `/` have priority 2 and `^` has priority 3. The brackets are ignored, but each occurrence of the opening bracket `(` increases priority by 4 and each occurrence of closing bracket `)` decreases priority by 4. The scanner puts each couple including its current priority to the stack and does a test at the top of the stack. The top of the stack is executed if the priority of the top operator is less or equal the previous operator priority. For our example the stack is only pushed without execution until `\D+` occurs. Our example in the stack looks like:

```
    \D + 1  1<=5 exec:
    \C + 5         {\C+\D} + 1  1<=2 exec:
    \B * 2             \B  * 2    {\B*{\C+\D}} + 1  1<=1 exec:
    \A + 1             \A  + 1                \A  + 1   {\A+{\B*{\C+\D}}} + 1
bottom  0            bottom  0          bottom    0                bottom   0
```

Now, the priority on the top is greater, then scanner pushes next couple and does the test on the top of the stack again.

```
                 \E \end 0   0<=1   exec:
  {\A+{\B*{\C+\D}}}  +   1       {{\A+{\B*{\C+\D}}}+\E} \end 0   0<=0 exec:
               bottom    0                            bottom  0        RESULT
```

Let p_t, p_p are the priority on the top and the previous priority in the stack. Let v_t, v_p are operands on the top and in the previous line in the stack, and the same notation is used for operators o_t and o_p. If $p_t \le p_p$ then: pop the stack twice, create composed operand $v_n = v_p\, o_p\, v_t$ and push v_n, o_t, p_t. Else push new couple "operand, operator" from the expression scanner. In both cases try to execute the top of the stack again. If the bottom of the stack is reached then the last operand is the result.

The macro `\apEVALa` ⟨*sequence*⟩`{`⟨*expression*⟩`}` runs the evaluation of the expression in the group. The base priority is initialized by `\apnumA=0`, then `\apEVALb`⟨*expression*⟩`\end` scans the expression and saves the result in the form `\PLUS{\A}{\MUL{\B}{\C}}` (etc.) into the `\tmpb` macro. This macro is expanded after group and the content in `\tmpb` is executed. The new result of such execution is stored to the `\OUT` macro, which is finally set to the desired ⟨*sequence*⟩.

apnum.tex

```
\def\apEVALa#1#2{{\apnumA=0 \apnumE=1 \apEVALb#2\end\expandafter}\tmpb \let#1=\OUT}
```

`\apSIGN`: 5–6, 9–13, 15–16, 20–21, 26, 30, 33   `\apEVALa`: 5–6, 9

The scanner is in one of the two states: reading operand or reading operator. The first state is initialized by \apEVALb which follows to the \apEVALc. The \apEVALc reads one token and switches by its value. If the value is a + or - sign, it is assumed to be the part of the operand prefix. Plus sign is ignored (and \apEVALc is run again), minus signs are accumulated into \tmpa.

The auxiliary macro \apEVALd runs the following tokens to the \fi, but first it closes the conditional and skips the rest of the macro \apEVALc.

apnum.tex

```
48: \def\apEVALb{\def\tmpa{}\apEVALc}
49: \def\apEVALc#1{%
50:    \ifx+#1\apEVALd \apEVALc \fi
51:    \ifx-#1\edef\tmpa{\tmpa-}\apEVALd\apEVALc \fi
52:    \ifx(#1\apEVALd \apEVALe \fi
53:    \ifx\the#1\apEVALd \apEVALf\the\fi
54:    \ifx\number#1\apEVALd \apEVALf\number\fi
55:    \apTESTdigit#1\iftrue
56:       \ifx E#1\let\tmpb=\tmpa \expandafter\apEVALd\expandafter\apEVALk
57:       \else \edef\tmpb{\tmpa#1}\expandafter\apEVALd\expandafter\apEVALn\fi\fi
58:    \edef\tmpb{\tmpa\noexpand#1}\futurelet\apNext\apEVALg
59: }
60: \def\apEVALd#1\fi#2\apNext\apEVALg{\fi#1}
```

If the next token is opening bracket, then the global priority is increased by 4 using the macro \apEVALe. Moreover, if the sign before bracket generates the negative result, then the new multiplication (by -1) is added using \apEVALp to the operand stack.

apnum.tex

```
61: \def\apEVALe{%
62:    \ifx\tmpa\empty \else \ifnum\tmpa1<0 \def\tmpb{-1}\apEVALp \MUL 4\fi\fi
63:    \advance\apnumA by4
64:    \apEVALb
65: }
```

If the next token is \the or \number primitives (see lines 53 and 54), then one following token is assumed as TeX register and these two tokens are interpreted as an operand. This is done by \apEVALf. The operand is packed to the \tmpb macro.

apnum.tex

```
66: \def\apEVALf#1#2{\expandafter\def\expandafter\tmpb\expandafter{\tmpa#1#2}\apEVALo}
```

If the next token is not a number (the \apTESTdigit#1\iftrue results like \iffalse at line 55) then we save the sign plus this token to the \tmpb at line 58 and we do check of the following token by \futurelet. The \apEVALg is processed after that. The test is performed here if the following token is open brace (a macro with parameter). If this is true then this parameter is appended to \tmpb by \apEVALh and the test about the existence of second parameter in braces is repeated by next \futurelet. The result of this loop is stored into \tmpb macro which includes ⟨*sign*⟩ followed by ⟨*token*⟩ followed by all parameters in braces. This is considered as an operand.

apnum.tex

```
67: \def\apEVALg{\ifx\apNext \bgroup \expandafter\apEVALh \else \expandafter\apEVALo \fi}
68: \def\apEVALh#1{\expandafter\def\expandafter\tmpb\expandafter{\tmpb{#1}}\futurelet\apNext\apEVALg}
```

If the next token after the sign is a digit or a dot (tested in \apEVALc by \apTESTdigit at line 55), then there are two cases. The number includes the E symbol as a first symbol (this is allowed in scientific notation, mantissa is assumed to equal to one). The \apEVALk is executed in such case. Else the \apEVALn starts the reading the number.

The first case with E letter in the number is solved by macros \apEVALk and \apEVALm. The number after E is read by \apE= and this number is appended to the \tmpb and the expression scanner skips to \apEVALo.

apnum.tex

```
69: \def\apEVALk{\afterassignment\apEVALm\apE=}
70: \def\apEVALm{\edef\tmpb{\tmpb E\the\apE}\apEVALo}
```

The second case (there is normal number) is processed by the macro \apEVALn. This macro reads digits (token per token) and saves them to the \tmpb. If the next token isn't digit nor dot then the second state of the scanner (reading an operator) is invoked by running \apEVALo. If the E is found then the exponent is read to \apE and it is processed by \apEVALm.

apnum.tex
```
\def\apEVALn#1{\apTESTdigit#1%
   \iftrue \ifx E#1\afterassignment\apEVALm\expandafter\expandafter\expandafter\apE
           \else\edef\tmpb{\tmpb#1}\expandafter\expandafter\expandafter\apEVALn\fi
   \else \expandafter\apEVALo\expandafter#1\fi
}
```

The reading an operator is done by the \apEVALo macro. This is more simple because the operator is only one token. Depending on this token the macro \apEVALp ⟨*operator*⟩⟨*priority*⟩ pushes to the stack (by the macro \apEVALpush) the value from \tmpb, the ⟨*operator*⟩ and the priority increased by \apnumA (level of brackets).

If there is a problem (level of brackets less than zero, level of brackets not equal to zero at the end of the expression, unknown operator) we print an error using \apEVALerror macro.

The \apNext is set to \apEVALb, i.e. scanner returns back to the state of reading the operand. But exceptions exist: if the) is found then priority is decreased and the macro \apEVALo is executed again. If the end of the ⟨*expression*⟩ is found then the loop is ended by \let\apNext=\relax.

apnum.tex
```
\def\apEVALo#1{\let\apNext=\apEVALb
   \ifx+#1\apEVALp \apEPLUS 1\fi
   \ifx-#1\apEVALp \apEMINUS 1\fi
   \ifx*#1\apEVALp \apEMUL 2\fi
   \ifx/#1\apEVALp \apEDIV 2\fi
   \ifx^#1\apEVALp \apEPOW 3\fi
   \ifx)#1\advance\apnumA by-4 \let\apNext=\apEVALo \let\tmpa=\relax
      \ifnum\apnumA<0 \apEVALerror{many brackets ")"}\fi
   \fi
   \ifx\end#1%
      \ifnum\apnumA>0 \apEVALerror{missing bracket ")"}%
      \else \apEVALp\END 0\fi
      \let\apNext=\relax
   \fi
   \ifx\tmpa\relax \else \apEVALerror{unknown operator "\string#1"}\fi
   \apnumE=0 \apNext
}
\def\apEVALp#1#2{%
   \apnumB=#2 \advance\apnumB by\apnumA
   \toks0=\expandafter{\expandafter{\tmpb}{#1}}%
   \expandafter\apEVALpush\the\toks0\expandafter{\the\apnumB}% {value}{op}{priority}
   \let\tmpa=\relax
}
```

The public values of \PLUS, \MINUS etc. macros are saved to the \apEPLUS, \apEMINUS, \apEMUL, \apEDIV, \apEPOW and these sequences are used in \evaldef. The reason is that the public macros can be changed later by the user but we need be sure of usage the right macros.

apnum.tex
```
\let\apEPLUS=\PLUS \let\apEMINUS=\MINUS \let\apEMUL=\MUL \let\apEDIV=\DIV \let\apEPOW=\POW
```

The \apEVALstack macro includes the stack, three items {⟨*operand*⟩}{⟨*operator*⟩}{⟨*priority*⟩} per level. Left part of the macro contents is the top of the stack. The stack is initialized with empty operand and operator and with priority zero. The dot here is only the “total bottom” of the stack.

apnum.tex
```
\def\apEVALstack{{}{}{0}.}
```

The macro \apEVALpush {⟨*operand*⟩}{⟨*operator*⟩}{⟨*priority*⟩} pushes its parameters to the stack and runs \apEVALdo⟨*whole-stack*⟩@ to do the desired work on the top of the stack.

apnum.tex
```
\def\apEVALpush#1#2#3{% value, operator, priority
   \toks0={{#1}{#2}{#3}}%
   \expandafter\def\expandafter\apEVALstack\expandafter{\the\toks0\apEVALstack}%
   \expandafter\apEVALdo\apEVALstack@%
}
```

Finally, the macro \apEVALdo `{`⟨*vt*⟩`}{`⟨*ot*⟩`}{`⟨*pt*⟩`}{`⟨*vp*⟩`}{`⟨*op*⟩`}{`⟨*pp*⟩`}`⟨*rest-of-the-stack*⟩`@` performs the execution described at the beginning of this section. The new operand ⟨*vn*⟩ is created as ⟨*op*⟩`{vp}{vt}`, this means `\apEPLUS{`⟨*vp*⟩`}{`⟨*vt*⟩`}` for example. The operand is not executed now, only the result is composed by the normal TeX notation. If the bottom of the stack is reached then the result is saved to the `\tmpb` macro. This macro is executed after group by the `\apEVALa` macro.

apnum.tex
```
\def\apEVALdo#1#2#3#4#5#6#7@{%
   \apnumB=#3 \ifx#2\POW \advance\apnumB by1 \fi
   \ifnum\apnumB>#6\else
      \ifnum#6=0 \def\tmpb{#1}%\toks0={#1}\message{RESULT: \the\toks0}
                 \ifnum\apnumE=1 \def\tmpb{\apPPn{#1}}\fi
      \else \def\apEVALstack{#7}\apEVALpush{#5{#4}{#1}}{#2}{#3}%
   \fi\fi
}
```

The macro \apEVALerror ⟨*string*⟩ prints an error message. We decide to be better to print only `\message`, no `\errmessage`. The `\tmpb` is prepared to create `\OUT` as `??` and the `\apNext` macro is set in order to skip the rest of the scanned ⟨*expression*⟩.

apnum.tex
```
\def\apEVALerror#1{\message{\noexpand\evaldef ERROR: #1.}%
   \def\tmpb{\def\OUT{??}}\def\apNext##1\end{}%
}
```

The auxiliary macro \apTESTdigit ⟨*token*⟩`\iftrue` tests, if the given token is digit, dot or `E` letter.

apnum.tex
```
\def\apTESTdigit#1#2{%
   \ifx E#1\apXtrue \else
      \ifcat.\noexpand#1%
         \ifx.#1\apXtrue \else
            \ifnum`#1<`0 \apXfalse\else
               \ifnum`#1>`9 \apXfalse\else \apXtrue\fi
         \fi\fi
      \else \apXfalse
   \fi\fi
   \ifapX
}
```

2.3 Preparation of the Parameter

All operands of `\PLUS`, `\MINUS`, `\MUL`, `\DIV` and `\POW` macros are preprocessed by `\apPPa` macro. This macro solves (roughly speaking) the following tasks:

- It partially expands (by `\expandafter`) the parameter while ⟨*sign*⟩ is read.
- The ⟨*sign*⟩ is removed from parameter and the appropriate `\apSIGN` value is set.
- If the next token after ⟨*sign*⟩ is `\relax` then the rest of the parameter is executed in the group and the results `\OUT`, `\apSIGN` and `\apE` are used.
- Else the number is read and saved to the parameter.
- If the read number has the scientific notation ⟨*mantissa*⟩`E`⟨*exponent*⟩ then only ⟨*mantissa*⟩ is saved to the parameter and `\apE` is set as ⟨*exponent*⟩. Else `\apE` is zero.

The macro \apPPa ⟨*sequence*⟩⟨*parameter*⟩ calls \apPPb ⟨*parameter*⟩`@`⟨*sequence*⟩ and starts reading the ⟨*parameter*⟩. The result will be stored to the ⟨*sequence*⟩.

Each token from ⟨*sign*⟩ is processed by three `\expandafters` (because there could be `\csname...\endcsname`). It means that the parameter is partially expanded when ⟨*sign*⟩ is read. The `\apPPb` macro sets the initial value of `\tmpc` and `\apSIGN` and executes the macro \apPPc ⟨*parameter*⟩`@`⟨*sequence*⟩.

apnum.tex
```
\def\apPPa#1#2{\expandafter\apPPb#2@#1}
\def\apPPb{\def\tmpc{}\apSIGN=1 \apE=0 \apXfalse \expandafter\expandafter\expandafter\apPPc}
\def\apPPc#1{%
   \ifx+#1\apPPd \fi
```

```
   \ifx-#1\apSIGN=-\apSIGN \apPPd \fi
   \ifx\relax#1\apPPe \fi
   \apPPg#1%
}
\def\apPPd#1\apPPg#2{\fi\expandafter\expandafter\expandafter\apPPc}
```

The `\apPPc` reads one token from ⟨*sign*⟩ and it is called recursively while there are `+` or `-` signs. If the read token is `+` or `-` then the `\apPPd` closes conditionals and executes `\apPPc` again.

If `\relax` is found then the rest of parameter is executed by the `\apPPe`. The macro ends by `\apPPf` ⟨*result*⟩`@` and this macro reverses the sign if the result is negative and removes the minus sign from the front of the parameter.

apnum.tex

```
\def\apPPe#1\apPPg#2#3@{\fi\apXtrue{#3% execution of the parameter in the group
   \edef\tmpc{\apE=\the\apE\relax\noexpand\apPPf\OUT@}\expandafter}\tmpc
}
\def\apPPf#1{\ifx-#1\apSIGN=-\apSIGN \expandafter\apPPg\else\expandafter\apPPg\expandafter#1\fi}
```

The `\apPPg` ⟨*parameter*⟩`@` macro is called when the ⟨*sign*⟩ was processed and removed from the input stream. The main reason of this macro is to remove trailing zeros from the left and to check, if there is the zero value written for example in the form `0000.000`. When this macro is started then `\tmpc` is empty. This is a flag for removing trailing zeros. They are simply ignored before decimal point. The `\apPPg` is called again by `\apPPh` macro which removes the rest of `\apPPg` macro and closes the conditional. If the decimal point is found then next zeros are accumulated to the `\tmpc`. If the end of the parameter `@` is found and we are in the "removing zeros state" then the whole value is assumed to be zero and this is processed by `\apPPi␣@`. If another digit is found (say 2) then there are two situations: if the `\tmpc` is non-empty, then the digit is appended to the `\tmpc` and the `\apPPi`⟨*expanded-tmp*⟩ is processed (say `\apPPi␣.002`) followed by the rest of the parameter. Else the digit itself is stored to the `\tmpc` and it is returned back to the input stream (say `\apPPi 2`) followed by the rest of the parameter.

apnum.tex

```
\def\apPPg#1{%
   \ifx.#1\def\tmpc{.}\apPPh\fi
   \ifx\tmpc\empty\else\edef\tmpc{\tmpc#1}\fi
   \ifx0#1\apPPh\fi
   \ifx\tmpc\empty\edef\tmpc{#1}\fi
   \ifx@#1\def\tmpc{@}\fi
   \expandafter\apPPi\tmpc
}
\def\apPPh#1\apPPi\tmpc{\fi\apPPg}
```

The macro `\apPPi` ⟨*parameter-without-trailing-zeros*⟩`@`⟨*sequence*⟩ switches to two cases: if the execution of the parameter was processed then the `\OUT` doesn't include `E` notation and we can simply define ⟨*sequence*⟩ as the ⟨*parameter*⟩ by the `\apPPj` macro. This saves the copying of the (possible) long result to the input stream again.

If the executing of the parameter was not performed, then we need to test the existence of the `E` notation of the number by the `\apPPk` macro. We need to put the ⟨*parameter*⟩ to the input stream and to use `\apPPl` to test these cases. We need to remove unwanted `E` letter by the `\apPPm` macro.

apnum.tex

```
\def\apPPi{\ifapX \expandafter\apPPj \else \expandafter\apPPk \fi}
\def\apPPj#1@#2{\def#2{#1}}
\def\apPPk#1@#2{\ifx@#1@\apSIGN=0 \def#2{0}\else \apPPl#1E@#2\fi}
\def\apPPl#1E#2@#3{%
   \ifx@#1@\def#3{1}\else\def#3{#1}\fi
   \ifx@#2@\else \afterassignment\apPPm \apE=#2\fi
}
\def\apPPm E{}
```

The `\apPPn` ⟨*param*⟩ macro does the same as `\apPPa\OUT{`⟨*param*⟩`}`, but the minus sign is returned back to the `\OUT` macro if the result is negative.

apnum.tex

```
\def\apPPn#1{\expandafter\apPPb#1@\OUT \edef\OUT{\ifnum\apSIGN<0-\fi\OUT}}
```

The \apPPab ⟨*macro*⟩{⟨*paramA*⟩}{⟨*paramB*⟩} is used for parameters of all macros \PLUS, \MUL etc. It prepares the ⟨*paramA*⟩ to \tmpa, ⟨*paramB*⟩ to \tmpb, the sign and ⟨*decimal-exponent*⟩ of ⟨*paramA*⟩ to the \apSIGNa and \apEa, the same of ⟨*paramB*⟩ to the \apSIGNa and \apEa. Finally, it executes the ⟨*macro*⟩.

apnum.tex

```
\def\apPPab#1#2#3{%
   \expandafter\apPPb#2@\tmpa \apSIGNa=\apSIGN \apEa=\apE
   \expandafter\apPPb#3@\tmpb \apSIGNb=\apSIGN \apEb=\apE
   #1%
}
```

The \apPPs ⟨*macro*⟩⟨*sequence*⟩{⟨*param*⟩} prepares parameters for \ROLL, \ROUND and \NORM macros. It saves the ⟨*param*⟩ to the \tmpc macro, expands the ⟨*sequence*⟩ and runs the macro \apPPt ⟨*macro*⟩⟨*expanded-sequence*⟩.@⟨*sequence*⟩. The macro \apPPt reads first token from the ⟨*expanded-sequence*⟩ to #2. If #2 is minus sign, then \apnumG=-1. Else \apnumG=1. Finally the ⟨*macro*⟩⟨*expanded-sequence*⟩.@⟨*sequence*⟩ is executed (but without the minus sign in the input stream). If #2 is zero then \apPPu ⟨*macro*⟩⟨*rest*⟩.@⟨*sequence*⟩ is executed. If the ⟨*rest*⟩ is empty, (i.e. the parameter is simply zero) then ⟨*macro*⟩ isn't executed because there in nothing to do with zero number as a parameter of \ROLL, \ROUND or \NORM macros.

apnum.tex

```
\def\apPPs#1#2#3{\def\tmpc{#3}\expandafter\apPPt\expandafter#1#2.@#2}
\def\apPPt#1#2{%
   \ifx-#2\apnumG=-1 \def\apNext{#1}%
   \else \ifx0#2\apnumG=0 \def\apNext{\apPPu#1}\else \apnumG=1 \def\apNext{#1#2}\fi\fi
   \apNext
}
\def\apPPu#1#2.@#3{\ifx@#2@\apnumG=0 \ifx#1\apROUNDa\def\XOUT{}\fi
   \else\def\apNext{\apPPt#1#2.@#3}\expandafter\apNext\fi
}
```

The macro \apEVALone ⟨*macro*⟩⟨*parameter*⟩ prepares one parameter for the function-like ⟨*macro*⟩. This parameter could be an ⟨*expression*⟩. The ⟨*macro*⟩ is executed after the parameter is evaluated and saved to the \OUT macro. The sign is removed from the parameter by the \apNOminus macro.

The macro \apEVALtwo ⟨*macro*⟩⟨*paramA*⟩⟨*paramB*⟩ evaluates the ⟨*paramA*⟩ and ⟨*paramB*⟩. They could be ⟨*expressions*⟩. They are saved to the \tmpa and \tmpb macros, the signs are saved to \apSIGNa and \apSIGNb, the exponents (if scientific notation were used) are saved to \apEa and \apEb registers. Finally the the function-like ⟨*macro*⟩ is executed.

apnum.tex

```
\def\apEVALone#1#2{\evaldef\OUT{#2}\ifnum\apSIGN<0 \expandafter\apNOminus\OUT@\OUT\fi #1}
\def\apEVALtwo#1#2#3{%
   {\evaldef\OUT{#2}\apOUTtmpb}\tmpb \let\tmpa=\OUT \apSIGNa=\apSIGN \apEa=\apE
   \ifnum\apSIGNa<0 \expandafter\apNOminus\tmpa@\tmpa\fi
   {\evaldef\OUT{#3}\apOUTtmpb}\tmpb \let\tmpb=\OUT \apSIGNb=\apSIGN \apEb=\apE
   \ifnum\apSIGNb<0 \expandafter\apNOminus\tmpb@\tmpb\fi
   #1%
}
\def\apNOminus-#1@#2{\def#2{#1}}
```

2.4 Addition and Subtraction

The significant part of the optimization in \PLUS, \MUL, \DIV and \POW macros is the fact, that we don't treat with single decimal digits but with their quartets. This means that we are using the numeral system with the base 10000 and we calculate four decimal digits in one elementary operation. The base was chosen 10^4 because the multiplication of such numbers gives results less than 10^8 and the maximal number in the TEX register is about $2 \cdot 10^9$. We'll use the word "Digit" (with capitalized D) in this documentation if this means the digit in the numeral system with base 10000, i.e. one Digit is four digits. Note that for addition we can use the numeral system with the base 10^8 but we don't do it, because the auxiliary macros \apIV* for numeral system of the base 10^4 are already prepared.

Suppose the following example (the spaces between Digits are here only for more clarity).

```
  123 4567 8901 9999         \apnumA=12 \apnumE=3 \apnumD=16
+                 22.423     \apnumB=0  \apnumF=2 \apnumC=12
--------------------------
sum in reversed order and without transmissions:
               {4230}{10021}{8901}{4567}{123}  \apnumD=-4
sum in normal order including transmissions:
  123 4567 8902 0021.423
```

In the first pass, we put the number with more or equal Digits before decimal point above the second number. There are three Digits more in the example. The `\apnumC` register saves this information (multiplied by 4). The first pass creates the sum in reversed order without transmissions between Digits. It simply copies the `\apnumC/4` Digits from the first number to the result in reversed order. Then it does the sums of Digits without transmissions. The `\apnumD` is a relative position of the decimal point to the edge of the calculated number.

The second pass reads the result of the first pass, calculates transmissions and saves the result in normal order.

The first Digit of the operands cannot include four digits. The number of digits in the first Digit is saved in `\apnumE` (for first operand) and in `\apnumF` (for second one). The rule is to have the decimal point between Digits in all circumstances.

The macro `\apPLUSa` does the following work:

- It gets the operands in `\tmpa` and `\tmpb` macros using the `\apPPab`.
- If the scientific notation is used and the decimal exponents `\apEa` and `\apEb` are not equal then the decimal point of one operand have to be shifted (by the macro `\apPLUSxE` at line 189).
- The digits before decimal point are calculated for both operands by the `\apDIG` macro. The first result is saved to `\apnumA` and the second result is saved to `\apnumB`. The `\apDIG` macro removes decimal point (if exists) from the parameters (lines 190 and 191).
- The number of digits in the first Digit is calculated by `\apIVmod` for both operands. This number is saved to `\apnumE` and `\apnumF`. This number is subtracted from `\apnumA` and `\apnumB`, so these registers now includes multiply of four (lines 192 and 193).
- The `\apnumC` includes the difference of Digits before the decimal point (multiplied by four) of given operands (line 194).
- If the first operand is negative then the minus sign is inserted to the `\apPLUSxA` macro else this macro is empty. The same for the second operand and for the macro `\apPLUSxB` is done (lines 195 and 196).
- If both operands are positive, then the sign of the result `\apSIGN` is set to one. If both operands are negative, then the sign is set to -1. But in both cases mentioned above we will do (internally) addition, so the macros `\apPLUSxA` and `\apPLUSxB` are set to empty. If one operand is negative and second positive then we will do subtraction. The `\apSIGN` register is set to zero and it will set to the right value later (lines 197 to 199).
- The macro `\apPLUSb`⟨*first-op*⟩⟨*first-dig*⟩⟨*second-op*⟩⟨*second-dig*⟩⟨*first-Dig*⟩ does the calculation of the first pass. The ⟨*first-op*⟩ has to have more or equal Digits before decimal point than ⟨*second-op*⟩. This is reason why this macro is called in two variants dependent on the value `\apnumC`. The macros `\apPLUSxA` and `\apPLUSxB` (with the sign of the operands) are exchanged (by the `\apPLUSg`) if the operands are exchanged (lines 200 to 201).
- The `\apnumG` is set by the macro `\apPLUSb` to the sign of the first nonzero Digit. It is equal to zero if there are only zero Digits after first pass. The result is zero in such case and we do nothing more (line 203).
- The transmission calculation is different for addition and subtraction. If the subtraction is processed then the sign of the result is set (using the value `\apnumG`) and the `\apPLUSm` for transmissions is prepared. Else the `\apPLUSp` for transmissions is prepared as the `\apNext` macro (line 204)
- The result of the first pass is expanded in the input stream and the `\apNext` (i.e. transmissions calculation) is activated at line 205.

- if the result is in the form `.000123`, then the decimal point and the trailing zeros have to be inserted. Else the trailing zeros from the left side of the result have to be removed by \apPLUSy. This macro adds the sign of the result too (lines 206 to 212)

apnum.tex
```
188: \def\apPLUSa{%
189:    \ifnum\apEa=\apEb \apE=\apEa \else \apPLUSxE \fi
190:    \apDIG\tmpa\relax \apnumA=\apnumD % digits before decimal point
191:    \apDIG\tmpb\relax \apnumB=\apnumD
192:    \apIVmod \apnumA \apnumE \advance\apnumA by-\apnumE % digits in the first Digit
193:    \apIVmod \apnumB \apnumF \advance\apnumB by-\apnumF
194:    \apnumC=\apnumB \advance\apnumC by-\apnumA % difference between Digits
195:    \ifnum\apSIGNa<0 \def\apPLUSxA{-}\else \def\apPLUSxA{}\fi
196:    \ifnum\apSIGNb<0 \def\apPLUSxB{-}\else \def\apPLUSxB{}\fi
197:    \apSIGN=0 % \apSIGN=0 means that we are doing subtraction
198:    \ifx\apPLUSxA\empty \ifx\apPLUSxB\empty \apSIGN=1 \fi\fi
199:    \if\apPLUSxA-\relax \if\apPLUSxB-\relax \apSIGN=-1 \def\apPLUSxA{}\def\apPLUSxB{}\fi\fi
200:    \ifnum\apnumC>0 \apPLUSg  \apPLUSb \tmpb\apnumF \tmpa\apnumE \apnumB % first pass
201:    \else \apnumC=-\apnumC    \apPLUSb \tmpa\apnumE \tmpb\apnumF \apnumA
202:    \fi
203:    \ifnum\apnumG=0 \def\OUT{0}\apSIGN=0 \apE=0 \else
204:       \ifnum\apSIGN=0 \apSIGN=\apnumG \let\apNext=\apPLUSm \else \let\apNext=\apPLUSp \fi
205:       \apnumX=0 \edef\OUT{\expandafter}\expandafter \apNext \OUT@% second pass
206:       \ifnum\apnumD<1 % result in the form .000123
207:          \apnumZ=-\apnumD
208:          \def\tmpa{.}%
209:          \ifnum\apnumZ>0 \apADDzeros\tmpa \fi % adding dot and left zeros
210:          \edef\OUT{\ifnum\apSIGN<0-\fi\tmpa\OUT}%
211:       \else
212:          \edef\OUT{\expandafter}\expandafter\apPLUSy \OUT@% removing left zeros
213:    \fi\fi
214: }
```

The macro \apPLUSb ⟨*first-op*⟩⟨*first-dig*⟩⟨*second-op*⟩⟨*second-dig*⟩⟨*first-Dig*⟩ starts the first pass. The ⟨*first-op*⟩ is the first operand (which have more or equal Digits before decimal point). The ⟨*first-dig*⟩ is the number of digits in the first Digit in the first operand. The ⟨*second-op*⟩ is the second operand and the ⟨*second-dig*⟩ is the number of digits in the first Digit of the second operand. The ⟨*first-Dig*⟩ is the number of Digits before decimal point of the first operand, but without the first Digit and multiplied by 4.

The macro\apPLUSb saves the second operand to `\tmpd` and appends the $4 -$ ⟨*second-dig*⟩ empty parameters before this operand in order to read desired number of digits to the first Digit of this oparand. The macro \apPLUSb saves the first operand to the input queue after \apPLUSc macro. It inserts the appropriate number of empty parameters (in `\tmpc`) before this operand in order to read the right number of digits in the first attempt. It appends the \apNL marks to the end in order to recognize the end of the input stream. These macros expands simply to zero but we can test the end of input stream by `\ifx`.

The macro \apPLUSb calculates the number of digits before decimal point (rounded up to multiply by 4) in `\apnumD` by advancing ⟨*first-DIG*⟩ by 4. It initializes `\apnumZ` to zero. If the first nonzero Digit will be found then `\apnumZ` will be set to this Digit in the \apPLUSc macro.

apnum.tex
```
\def\apPLUSb#1#2#3#4#5{%
   \edef\tmpd{\ifcase#4\or{}{}{}\or{}{}\or{}\fi#3}%
   \edef\tmpc{\ifcase#2\or{}{}{}\or{}{}\or{}\fi}%
   \let\apNext=\apPLUSc \apnumD=#5\advance\apnumD by4 \apnumG=0 \apnumZ=0 \def\OUT{}%
   \expandafter\expandafter\expandafter\apPLUSc\expandafter\tmpc#1\apNL\apNL\apNL\apNL@%
}
```

The macro \apPLUSc is called repeatedly. It reads one Digit from input stream and saves it to the `\apnumY`. Then it calls the \apPLUSe, which reads (if it is allowed, i.e. if `\apnumC<=0`) one digit from second operand `\tmpd` by the \apIVread macro. Then it does the addition of these digits and saves the result into the \OUT macro in reverse order.

Note, that the sign \apPLUSxA is used when `\apnumY` is read and the sign \apPLUSxB is used when advancing is performed. This means that we are doing addition or subtraction here.

If the first nonzero Digit is reached, then the macro \apPLUSh sets the sign of the result to the `\apnumG` and (maybe) exchanges the `\apPLUSxA` and `\apPLUSxB` macros (by the `\apPLUSg` macro) in order to the internal result of the subtraction will be always non-negative.

If the end of input stream is reached, then `\apNext` (used at line 233) is reset from its original value `\apPLUSc` to the `\apPLUSd` where the `\apnumY` is simply set to zero. The reading from input stream is finished. This occurs when there are more Digits after decimal point in the second operand than in the first one. If the end of input stream is reached and the `\tmpd` macro is empty (all data from second operand was read too) then the `\apPLUSf` macro removes the rest of input stream and the first pass of the calculation is done.

apnum.tex
```
221: \def\apPLUSc#1#2#3#4{\apnumY=\apPLUSxA#1#2#3#4\relax
222:    \ifx\apNL#4\let\apNext=\apPLUSd\fi
223:    \ifx\apNL#1\relax \ifx\tmpd\empty \expandafter\expandafter\expandafter\apPLUSf \fi\fi
224:    \apPLUSe
225: }
226: \def\apPLUSd{\apnumY=0 \ifx\tmpd\empty \expandafter\apPLUSf \else\expandafter \apPLUSe\fi}
227: \def\apPLUSe{%
228:    \ifnum\apnumC>0 \advance\apnumC by-4
229:    \else \apIVread\tmpd \advance\apnumY by\apPLUSxB\apnumX \fi
230:    \ifnum\apnumZ=0 \apPLUSh \fi
231:    \edef\OUT{{\the\apnumY}\OUT}%
232:    \advance\apnumD by-4
233:    \apNext
234: }
235: \def\apPLUSf#1@{}
236: \def\apPLUSg{\let\tmpc=\apPLUSxA \let\apPLUSxA=\apPLUSxB \let\apPLUSxB=\tmpc}
237: \def\apPLUSh{\apnumZ=\apnumY
```

Why there is a complication about reading one parameter from input stream but second one from the macro `\tmpd`? This is more faster than to save both parameters to the macros and using `\apIVread` for both because the `\apIVread` must redefine its parameter. You can examine that this parameter is very long.

The `\apPLUSm` ⟨*data*⟩`@` macro does transmissions calculation when subtracting. The ⟨*data*⟩ from first pass is expanded in the input stream. The `\apPLUSm` macro reads repeatedly one Digit from the ⟨*data*⟩ until the stop mark is reached. The Digits are in the range −9999 to 9999. If the Digit is negative then we need to add 10000 and set the transmission value `\apnumX` to one, else `\apnumX` is zero. When the next Digit is processed then the calculated transmission value is subtracted. The macro `\apPLUSw` writes the result for each Digit `\apnumA` in the normal (human readable) order.

apnum.tex
```
240: \def\apPLUSm#1{%
241:    \ifx@#1\else
242:       \apnumA=#1 \advance\apnumA by-\apnumX
243:       \ifnum\apnumA<0 \advance\apnumA by\apIVbase \apnumX=1 \else \apnumX=0 \fi
244:       \apPLUSw
245:       \expandafter\apPLUSm
246:    \fi
247: }
```

The `\apPLUSp` ⟨*data*⟩`@` macro does transmissions calculation when addition is processed. It is very similar to`\apPLUSm`, but Digits are in the range 0 to 19998. If the Digit value is greater then 9999 then we need to subtract 10000 and set the transmission value `\apnumX` to one, else `\apnumX` is zero.

apnum.tex
```
248: \def\apPLUSp#1{%
249:    \ifx@#1\ifnum\apnumX>0 \edef\OUT{1\OUT}\fi
250:    \else
251:       \apnumA=\apnumX \advance\apnumA by#1
252:       \ifnum\apnumA<\apIVbase \apnumX=0 \else \apnumX=1 \advance\apnumA by-\apIVbase \fi
253:       \apPLUSw
254:       \expandafter\apPLUSp
255:    \fi
256: }
```

The \apPLUSw writes the result with one Digit (saved in \apnumA) to the \OUT macro. The \OUT is initialized as empty. If it is empty (it means we are after decimal point), then we need to write all four digits by \apIVwrite macro (including left zeros) but we need to remove right zeros by \apREMzerosR. If the decimal point is reached, then it is saved to the \OUT. But if the previous \OUT is empty (it means there are no digits after decimal point or all such digits are zero) then \def\OUT{\empty} ensures that the \OUT is non-empty and the ignoring of right zeros are disabled from now.

apnum.tex

```
\def\apPLUSw{%
   \ifnum\apnumD=0 \ifx\OUT\empty \def\OUT{\empty}\else \edef\OUT{.\OUT}\fi \fi
   \advance\apnumD by4
   \ifx\OUT\empty  \edef\tmpa{\apIVwrite\apnumA}\edef\OUT{\apREMzerosR\tmpa}%
   \else \edef\OUT{\apIVwrite\apnumA\OUT}\fi
}
```

The macro \apPLUSy ⟨*expanded-OUT*⟩@ removes left trailing zeros from the \OUT macro and saves the possible minus sign by the \apPLUSz macro.

apnum.tex

```
\def\apPLUSy#1{\ifx0#1\expandafter\apPLUSy\else \expandafter\apPLUSz\expandafter#1\fi}
\def\apPLUSz#1@{\edef\OUT{\ifnum\apSIGN<0-\fi#1}}
```

The macro \apPLUSxE uses the \apROLLa in order to shift the decimal point of the operand. We need to set the same decimal exponent in scientific notation before the addition or subtraction is processed.

apnum.tex

```
\def\apPLUSxE{%
   \apnumE=\apEa \advance\apnumE by-\apEb
   \ifnum\apEa>\apEb \apPPs\apROLLa\tmpb{-\apnumE}\apE=\apEa
   \else \apPPs\apROLLa\tmpa{\apnumE}\apE=\apEb \fi
}
```

2.5 Multiplication

Suppose the following multiplication example: 1234*567=699678.

```
      Normal format:          |       Mirrored format:
             1  2  3  4  *    |       4  3  2  1  *
                5  6  7       |       7  6  5
       ----------------       |     -----------------
*7:          7 14 21 28       | *7:  28 21 14  7
*6:       6 12 18 24          | *6:     24 18 12  6
*5:    5 10 15 20             | *5:        20 15 10  5
       ----------------       |     -----------------
       6  9  9  6  7  8       |       8  7  6  9  9  6
```

This example is in numeral system of base 10 only for simplification, the macros work really with base 10000. Because we have to do the transmissions between Digit positions from right to left in the normal format and because it is more natural for TEX to put the data into the input stream and read it sequentially from left to right, we use the mirrored format in our macros.

The macro \apMULa does the following:

- It gets the parameters in \tmpa and \tmpb preprocessed using the \apPPab macro.
- It evaluates the exponent of ten \apE which is usable when the scientific notation of numbers is used (line 274).
- It calculates \apSIGN of the result (line 275).
- If \apSIGN=0 then the result is zero and we will do nothing more (line 276).
- The decimal point is removed from the parameters by \apDIG⟨*param*⟩⟨*register*⟩. The \apnumD includes the number of digits before decimal point (after the \apDIG is used) and the ⟨*register*⟩ includes the number of digits in the rest. The \apnumA or \apnumB includes total number of digits in the parameters \tmpa or \tmpb respectively. The \apnumD is re-calculated: it saves the number of digits after decimal point in the result (lines 277 to 279).

- Let A is the number of total digits in the $\langle param\rangle$ and let $F = A \bmod 4$, but if $F = 0$ then reassign it to $F = 4$. Then F means the number of digits in the first Digit. This calculation is done by `\apIVmod`$\langle A\rangle\langle F\rangle$ macro. All another Digits will have four digits. The `\apMULb`$\langle param\rangle$`@@@@` is able to read four digits, next four digits etc. We need to insert appropriate number of empty parameters before the $\langle param\rangle$. For example `\apMULb{}{}{}`$\langle param\rangle$`@@@@` reads first only one digit from $\langle param\rangle$, next four digits etc. The appropriate number of empty parameters are prepared in the `\tmpc` macro (lines 280 to 281).
- The `\apMULb` reads the $\langle paramA\rangle$ (all Digits) and prepares the `\OUT` macro in the special interleaved format (described below). The format is finished by `*.` in the line 283.
- Analogical work is done with the second parameter $\langle paramB\rangle$. But this parameter is processed by `\apMULc`, which reads Digits of the parameter and inserts them to the `\tmpa` in the reversed order (lines 284 to 286).
- The main calculation is done by `\apMULd`$\langle paramB\rangle$`@`, which reads Digits from $\langle paramB\rangle$ (in reversed order) and does multiplication of the $\langle paramA\rangle$ (saved in the `\OUT`) by these Digits (line 287).
- The `\apMULg` macro converts the result `\OUT` to the human readable form (line 288).
- The possible minus sign and the trailing zeros of results of the type `.00123` is prepared by `\apADDzeros\tmpa` to the `\tmpa` macro. This macro is appended to the result in the `\OUT` macro (lines 289 to 291).

apnum.tex

```
273: \def\apMULa{%
274:    \apE=\apEa \advance\apE by\apEb
275:    \apSIGN=\apSIGNa \multiply\apSIGN by\apSIGNb
276:    \ifnum\apSIGN=0 \def\OUT{0}\apE=0 \else
277:       \apDIG\tmpa\apnumA \apnumX=\apnumA \advance\apnumA by\apnumD
278:       \apDIG\tmpb\apnumB \advance\apnumX by\apnumB \advance\apnumB by\apnumD
279:       \apnumD=\apnumX % \apnumD = the number of digits after decimal point in the result
280:       \apIVmod \apnumA \apnumF % \apnumF = digits in the first Digit of \tmpa
281:       \edef\tmpc{\ifcase\apnumF\or{}{}{}\or{}{}\or{}\fi}\def\OUT{}%
282:       \expandafter\expandafter\expandafter \apMULb \expandafter \tmpc \tmpa @@@@%
283:       \edef\OUT{*.\OUT}%
284:       \apIVmod \apnumB \apnumF % \apnumF = digits in the first Digit of \tmpb
285:       \edef\tmpc{\ifcase\apnumF\or{}{}{}\or{}{}\or{}\fi}\def\tmpa{}%
286:       \expandafter\expandafter\expandafter \apMULc \expandafter \tmpc \tmpb @@@@%
287:       \expandafter\apMULd \tmpa@%
288:       \expandafter\apMULg \OUT
289:       \edef\tmpa{\ifnum\apSIGN<0-\fi}%
290:       \ifnum\apnumD>0 \apnumZ=\apnumD \edef\tmpa{\tmpa.}\apADDzeros\tmpa \fi
291:       \ifx\tmpa\empty \else \edef\OUT{\tmpa\OUT}\fi
292:    \fi
293: }
```

We need to read the two data streams when the multiplication of the $\langle paramA\rangle$ by one Digit from $\langle paramB\rangle$ is performed and the partial sum is actualized. First: the digits of the $\langle paramA\rangle$ and second: the partial sum. We can save these streams to two macros and read one piece of information from such macros at each step, but this si not effective because the whole stream have to be read and redefined at each step. For TeX is more natural to put one data stream to the input queue and to read pieces of infromation thereof. Thus we interleave both data streams into one `\OUT` in such a way that one element of data from first stream is followed by one element from second stream and it is followed by second element from first stream etc. Suppose that we are at the end of $i-th$ line of the multiplication scheme where we have the partial sums $s_n, s_{n-1}, \ldots, s_0$ and the Digits of $\langle paramA\rangle$ are $d_k, d_{k-1}, \ldots, d_0$. The zero index belongs to the most right position in the mirrored format. The data will be prepared in the form:

```
. {s_n} {s_(n-1)}...{s_(k+1)} * {s_k} {d_(k-1)}...{s_1} {d_1} {s_0} {d_0} *
```

For our example (there is a simplification: numeral system of base 10 is used and no transmissions are processed), after second line (multiplication by 6 and calculation of partial sums) we have in `\OUT`:

```
. {28} * {45} {4} {32} {3} {19} {2} {6} {1} *
```

and we need to create the following line during calculation of next line of multiplication scheme:

```
. {28} {45} * {5*4+32} {4} {5*3+19} {3} {5*2+6} {2} {5*1} {1} *
```

This special format of data includes two parts. After the starting dot, there is a sequence of sums which are definitely calculated. This sequence is ended by first `*` mark. The last definitely calculated sum follows this mark. Then the partial sums with the Digits of ⟨*paramA*⟩ are interleaved and the data are finalized by second `*`. If the calculation processes the the second part of the data then the general task is to read two data elements (partial sum and the Digit) and to write two data elements (the new partial sum and the previous Digit). The line calculation starts by copying of the first part of data until the first `*` and appending the first data element after `*`. Then the `*` is written and the middle processing described above is started.

The macro `\apMULb` ⟨*paramA*⟩`@@@@` prepares the special format of the macro `\OUT` described above where the partial sums are zero. It means:

```
* . {d_k} 0 {d_(k-1)} 0 ... 0 {d_0} *
```

where d_i are Digits of ⟨*paramA*⟩ in reversed order.

The first "sum" is only dot. It will be moved before `*` during the first line processing. Why there is such special "pseudo-sum"? The `\apMULe` with the parameter delimited by the first `*` is used in the context `\apMULe.{`⟨*sum*⟩`}*` during the third line processing and the dot here protects from removing the braces around the first real sum.

apnum.tex
```
\def\apMULb#1#2#3#4{\ifx@#4\else
   \ifx\OUT\empty \edef\OUT{{#1#2#3#4}*}\else\edef\OUT{{#1#2#3#4}0\OUT}\fi
   \expandafter\apMULb\fi
}
```

The macro `\apMULc` ⟨*paramB*⟩`@@@@` reads Digits from ⟨*paramB*⟩ and saves them in reversed order into `\tmpa`. Each Digit is enclosed by TeX braces `{}`.

apnum.tex
```
\def\apMULc#1#2#3#4{\ifx@#4\else \edef\tmpa{{#1#2#3#4}\tmpa}\expandafter\apMULc\fi}
```

The macro `\apMULd` ⟨*paramB*⟩`@` reads the Digits from ⟨*paramB*⟩ (in reversed order), uses them as a coefficient for multiplication stored in `\tmpnumA` and processes the `\apMULe␣`⟨*special-data-format*⟩ for each such coefficient. This corresponds with one line in the multiplication scheme.

apnum.tex
```
\def\apMULd#1{\ifx@#1\else
     \apnumA=#1 \expandafter\apMULe \OUT
     \expandafter\apMULd
   \fi
}
```

The macro `\apMULe` ⟨*special-data-format*⟩ copies the first part of data format to the `\OUT`, copies the next element after first `*`, appends `*` and does the calculation by `\apMULf`. The `\apMULf` is recursively called. It reads the Digit to `#1` and the partial sum to the `#2` and writes `{\appnumA*#1+#2}{#1}` to the `\OUT` (lines 315 to 319). If we are at the end of data, then `#2` is `*` and we write the `{\apnumA*#1}{#1}` followed by ending `*` to the `\OUT` (lines 308 to 310).

apnum.tex
```
304: \def\apMULe#1*#2{\apnumX=0 \def\OUT{#1{#2}*}\def\apOUTl{}\apnumO=1 \apnumL=0 \apMULf}
305: \def\apMULf#1#2{%
306:   \advance\apnumO by-1 \ifnum\apnumO=0 \apOUTx \fi
307:   \apnumB=#1 \multiply\apnumB by\apnumA \advance\apnumB by\apnumX
308:   \ifx*#2%
309:      \ifnum\apnumB<\apIVbase
310:         \edef\OUT{\OUT\expandafter\apOUTs\apOUTl.,\ifnum\the\apnumB#1=0 \else{\the\apnumB}{#1}\fi*}%
311:      \else \apIVtrans
312:         \expandafter \edef\csname apOUT:\apOUTn\endcsname
313:                 {\csname apOUT:\apOUTn\endcsname{\the\apnumB}{#1}}%
314:         \apMULf0*\fi
315:   \else \advance\apnumB by#2
```

```
316:      \ifnum\apnumB<\apIVbase \apnumX=0 \else \apIVtrans \fi
317:      \expandafter
318:         \edef\csname apOUT:\apOUTn\endcsname{\csname apOUT:\apOUTn\endcsname{\the\apnumB}{#1}}%
319:      \expandafter\apMULf \fi
320: }
```

There are several complications in the algorithm described above.

- The result isn't saved directly to the `\OUT` macro, but partially into the macros `\apOUT:`⟨*num*⟩, as described in the section about auxiliary macros where the `\apOUTx` macro is defined.
- The transmissions between Digit positions are calculated. First, the transmission value `\apnumX` is set to zero in the `\apMULe`. Then this value is subtracted from the calculated value `\apnumB` and the new transmission is calculated using the `\apIVtrans` macro if `\apnumB` ≥ 10000. This macro modifies `\apnumB` in order it is right Digit in our numeral system.
- If the last digit has nonzero transmission, then the calculation isn't finished, but the new pair `{`⟨*transmission*⟩`}{0}` is added to the `\OUT`. This is done by recursively call of `\apMULf` at line 314.
- The another situation can be occurred: the last pair has both values zeros. Then we needn't to write this zero to the output. This is solved by the test `\ifnum\the\apnumB#1=0` at line 310.

The macro `\apMULg` ⟨*special-data-format*⟩`@` removes the first dot (it is the `#1` parameter) and prepares the `\OUT` to writing the result in reverse order, i.e. in human readable form. The next work is done by `\apMULh` and `\apMULi` macros. The `\apMULh` repeatedly reads the first part of the special data format (Digits of the result are here) until the first `*` is found. The output is stored by `\apMULo`⟨*digits*⟩`{`⟨*data*⟩`}` macro. If the first `*` is found then the `\apMULi` macro repeatedly reads the triple `{`⟨*Digit-of-result*⟩`}{`⟨*Digit-of-A*⟩`}{`⟨*next-Digit-of-result*⟩`}` and saves the first element in full (four-digits) form by the `\apIVwrite` if the third element isn't the stop-mark `*`. Else the last Digit (first Digit in the human readable form) is saved by `\the`, because we needn't the trailing zeros here. The third element is put back to the input stream but it is ignored by `\apMULj` macro when the process is finished.

apnum.tex

```
321: \def\apMULg#1{\def\OUT{}\apMULh}
322: \def\apMULh#1{\ifx*#1\expandafter\apMULi
323:    \else \apnumA=#1 \apMULo4{\apIVwrite\apnumA}%
324:          \expandafter\apMULh
325:    \fi
326: }
327: \def\apMULi#1#2#3{\apnumA=#1
328:    \ifx*#3\apMULo{\apNUMdigits\tmpa}{\the\apnumA}\expandafter\apMULj
329:    \else  \apMULo4{\apIVwrite\apnumA}\expandafter\apMULi
330:    \fi{#3}%
331: }
332: \def\apMULj#1{}
```

The `\apMULo` ⟨*digits*⟩`{`⟨*data*⟩`}` appends ⟨*data*⟩ to the `\OUT` macro. The number of digits after decimal point `\apnumD` is decreased by the number of actually printed digits ⟨*digits*⟩. If the decimal point is to be printed into ⟨*data*⟩ then it is performed by the `\apMULt` macro.

apnum.tex

```
333: \def\apMULo#1#2{\edef\tmpa{#2}%
334:    \advance\apnumD by-#1
335:    \ifnum\apnumD<1 \ifnum\apnumD>-4 \apMULt\fi\fi
336:    \edef\OUT{\tmpa\OUT}%
337: }
338: \def\apMULt{\edef\tmpa{\apIVdot{-\apnumD}\tmpa}\edef\tmpa{\tmpa}}
```

2.6 Division

Suppose the following example:

```
    <paramA> : <paramB>        <output>
        12345:678 = [12:6=2]   2   (2->1)
2*678  -1356
        -1215 <0 correction!   1
```

```
        12345
1*678   -678
         5565       [55:6=8]    9  (9->8)
9*678   -6102
         -537 <0 correction!    8
         5565
8*678   -5424
          1410      [14:6=2]     2
2*678    -1356
            0540    [05:6=0]      0
0*678         -0
             5400    [54:6=8]       9 (2x correction: 9->8, 8->7)
              ...
                     12345:678 = 182079...
```

We implement the division similar like pupils do it in the school (only the numeral system with base 10000 instead 10 is actually used, but we keep with base 10 in our illustrations). At each step of the operation, we get first two Digits from the dividend or remainder (called partial dividend or remainder) and do divide it by the first nonzero Digit of the divisor (called partial divisor). Unfortunately, the resulted Digit cannot be the definitive value of the result. We are able to find out this after the whole divisor is multiplied by resulted Digit and compared with the whole remainder. We cannot do this test immediately but only after a lot of following operations (imagine that the remainder and divisor have a huge number of Digits).

We need to subtract the remainder by the multiple of the divisor at each step. This means that we need to calculate the transmissions from the Digit position to the next Digit position from right to left (in the scheme illustrated above). Thus we need to reverse the order of Digits in the remainder and divisor. We do this reversion only once at the preparation state of the division and we interleave the data from the divisor and the dividend (the dividend will be replaced by the remainder, next by next remainder etc.).

The number of Digits of the dividend can be much greater than the number of Digits of the divisor. We need to calculate only with the first part of dividend/remainder in such case. We need to get only one new Digit from the rest of dividend at each calculation step. The illustration follows:

```
...used dividend.. | ... rest of dividend ...    | .... divisor ....
1234567890123456789 7890123456789012345678901234 : 1231231231231231231
 xxxxxxxxxxxxxxxxxx 7 <-  calculated remainder
  xxxxxxxxxxxxxxxxx x8 <- new calculated remainder
   xxxxxxxxxxxxxxxx xx9 <- new calculated remainder   etc.
```

We'll interleave only the "used dividend" part with the divisor at the preparation state. We'll put the "rest of dividend" to the input stream in the normal order. The macros do the iteration over calculation steps and they can read only one new Digit from this input stream if they need it. This approach needs no manipulation with the (potentially long) "rest of the dividend" at each step. If the divisor has only one Digit (or comparable small Digits) then the algorithm has only linear complexity with respect to the number of Digits in the dividend.

The numeral system with the base 10000 brings a little problem: we are simply able to calculate the number of digits which are multiple of four. But user typically wishes another number of calculated decimal digits. We cannot simply strip the trailing digits after calculation because the user needs to read the right remainder. This is a reason why we calculate the number of digits for the first Digit of the result. All another calculated Digits will have four digits. We need to prepare the first "partial dividend" in order to the F digits will be calculated first. How to do it? Suppose the following illustration of the first two Digits in the "partial remainder" and "partial divisor":

```
0000 7777 : 1111 = 7    .. one digit in the result
0007 7778 : 1111 = 70   .. two digits in the result
0077 7788 : 1111 = 700  .. three digits in the result
0777 7888 : 1111 = 7000 .. four digits in the result
7777 8888 : 1111 = ???? .. not possible in the numeral system of base 10000
```

We need to read $F-1$ digits to the first Digit and four digits to the second Digit of the "partial dividend". But this is true only if the dividend is "comparably greater or equal to" divisor. The word "comparably greater" means that we ignore signs and the decimal point in compared numbers and we assume the decimal points in the front of both numbers just before the first nonzero digit. It is obvious that if the dividend is "comparably less" than divisor then we need to read F digits to the first Digit.

The `\apDIVa` macro uses the `\tmpa` (dividend) and `\tmpb` (divisor) macros and does the following work:

- If the divisor `\tmpb` is equal to zero, print error and do nothing more (line 343).
- The `\apSIGN` of the result is calculated (line 344).
- If the dividend `\tmpa` is equal to zero, then `\OUT` and `\XOUT` are zeros and do nothing more (line 345).
- Calculate the exponent of ten `\apE` when scientific notation is used (Line 345).
- The number of digits before point are counted by `\apDIG` macro for both parameters. The difference is saved to `\apnumD` and this is the number of digits before decimal point in the result (the exception is mentioned later). The `\apDIG` macro removes the decimal point and (possible) left zeros from its parameter and saves the result to the `\apnumD` register (lines 347 to 349).
- The macro `\apDIVcomp`⟨*paramA*⟩⟨*paramB*⟩ determines if the ⟨*paramA*⟩ is "comparably greater or equal" to ⟨*paramB*⟩. The result is stored in the boolean value `apX`. We can ask to this by the `\ifapX`⟨*true*⟩`\else`⟨*false*⟩`\fi` construction (line 350).
- If the dividend is "comparably greater or equal" to the divisor, then the position of decimal point in the result `\apnumD` has to be shifted by one to the right. The same is completed with `\apnumH` where the position of decimal point of the remainder will be stored (line 351).
- The number of desired digits in the result `\apnumC` is calculated (lines 352 to 358).
- If the number of desired digits is zero or less than zero then do nothing more (line 358).
- Finish the calculation of the position of decimal point in the remainder `\apnumH` (line 351).
- Calculate the number of digits in the first Digit `\apnumF` (line 362).
- Read first four digits of the divisor by the macro `\apIVread`⟨*sequence*⟩. Note that this macro puts trailing zeros to the right if the data stream ⟨*param*⟩ is shorter than four digits. If it is empty then the macro returns zero. The returned value is saved in `\apnumX` and the ⟨*sequence*⟩ is redefined by new value of the ⟨*param*⟩ where the read digits are removed (line 363).
- We need to read only `\apnumF` (or `\apnumF` $-$ 1) digits from the `\tmpa`. This is done by the `\apIVreadX` macro at line 365. The second Digit of the "partial dividend" includes four digits and it is read by `\apIVread` macro at line 367.
- The "partial dividend" is saved to the `\apDIVxA` macro and the "partial divisor" is stored to the `\apDIVxB` macro. Note, that the second Digit of the "partial dividend" isn't expanded by simply `\the`, because when `\apnumX=11` and `\apnumA=2222` (for example), then we need to save `22220011`. These trailing zeros from left are written by the `\apIVwrite` macro (lines 368 to 369).
- The `\XOUT` macro for the currently computed remainder is initialized. The special interleaved data format of the remainder `\XOUT` is described below (line 370).
- The `\OUT` macro is initialized. The `\OUT` is generated as literal macro. First possible ⟨*sign*⟩, then digits. If the number of effective digits before decimal point `\apnumD` is negative, the result will be in the form `.000123` and we need to add the zeros by the `\apADDzeros` macro (lines 371 to 372).
- The registers for main loop are initialized. The `\apnumE` signalizes that the remainder of the partial step is zero and we can stop the calculation. The `\apnumZ` will include the Digit from the input stream where the "rest of dividend" will be stored (line 372).
- The main calculation loop is processed by the `\apDIVg` macro (line 374).
- If the division process stops before the position of the decimal point in the result (because there is zero remainder, for example) then we need to add the rest of zeros by `\apADDzeros` macro. This is actual for the results of the type `1230000` (line 375).
- If the remainder isn't equal to zero, we need to extract the digits of the remainder from the special data formal to the human readable form. This is done by the `\apDIVv` macro. The decimal point is inserted to the remainder by the `\apROLLa` macro (lines 377 to 378).

`\apDIVa`: 5, 21, 26, 30

apnum.tex

```
\def\apDIVa{%
   \ifnum\apSIGNb=0 \errmessage{Dividing by zero}\else
      \apSIGN=\apSIGNa \multiply\apSIGN by\apSIGNb
      \ifnum\apSIGNa=0 \def\OUT{0}\def\XOUT{0}\apE=0 \apSIGN=0 \else
         \apE=\apEa \advance\apE by-\apEb
         \apDIG\tmpb\relax \apnumB=\apnumD
         \apDIG\tmpa\relax \apnumH=\apnumD
         \advance\apnumD by-\apnumB % \apnumD = num. of digits before decimal point in the result
         \apDIVcomp\tmpa\tmpb  % apXtrue <=>  A>=B, i.e 1 digit from A/B
         \ifapX \advance\apnumD by1 \advance\apnumH by1 \fi
         \apnumC=\apTOT
         \ifnum\apTOT<0 \apnumC=-\apnumC
            \ifnum\apnumD>\apnumC \apnumC=\apnumD \fi
         \fi
         \ifnum\apTOT=0 \apnumC=\apFRAC \advance\apnumC by\apnumD
         \else \apnumX=\apFRAC \advance\apnumX by\apnumD
               \ifnum\apnumC>\apnumX \apnumC=\apnumX \fi
         \fi
         \ifnum\apnumC>0  % \apnumC = the number of digits in the result
            \advance\apnumH by-\apnumC % \apnumH = the position of decimal point in the remainder
            \apIVmod \apnumC \apnumF   % \apnumF = the number of digits in the first Digit
            \apIVread\tmpb \apnumB=\apnumX % \apnumB = partial divisor
            \apnumX=\apnumF \ifapX \advance\apnumX by-1 \fi
            \apIVreadX\apnumX\tmpa
            \apnumA=\apnumX % \apnumA = first Digit of the partial dividend
            \apIVread\tmpa  % \apnumX = second Digit of the partial dividend
            \edef\apDIVxA{\the\apnumA\apIVwrite\apnumX}% first partial dividend
            \edef\apDIVxB{\the\apnumB}% partial divisor
            \edef\XOUT{{\apDIVxB}{\the\apnumX}@{\the\apnumA}}% the \XOUT is initialized
            \edef\OUT{\ifnum\apSIGN<0-\fi}%
            \ifnum\apnumD<0 \edef\OUT{\OUT.}\apnumZ=-\apnumD \apADDzeros\OUT \fi
            \apnumE=1 \apnumZ=0
            \let\apNext=\apDIVg \apNext % <--- the main calculation loop is here
            \ifnum\apnumD>0 \apnumZ=\apnumD \apADDzeros\OUT \fi
            \ifnum\apnumE=0 \def\XOUT{0}\else % extracting remainder from \XOUT
               \edef\XOUT{\expandafter}\expandafter\apDIVv\XOUT
               \def\tmpc{\apnumH}\apnumG=\apSIGNa \expandafter\apROLLa\XOUT.@\XOUT
           \fi
         \else \def\OUT{0}\def\XOUT{0}\apE=0 \apSIGN=0
   \fi\fi\fi
}
```

The macro \apDIVcomp ⟨*paramA*⟩⟨*paramB*⟩ provides the test if the ⟨*paramA*⟩ is "comparably greater or equal" to ⟨*paramB*⟩. Imagine the following examples:

```
123456789 : 123456789 = 1
123456788 : 123456789 = .99999999189999992628
```

The example shows that the last digit in the operands can be important for the first digit in the result. This means that we need to compare whole operands but we can stop the comparison when the first difference in the digits is found. This is lexicographic ordering. Because we don't assume the existence of eTEX (or another extensions), we need to do this comparison by macros. We set the ⟨*paramA*⟩ and ⟨*paramB*⟩ to the `\tmpc` and `\tmpd` respectively. The trailing \apNLs are appended. The macro \apDIVcompA reads first 8 digits from first parameter and the macros \apDIVcompB reads first 8 digits from second parameter and does the comparison. If the numbers are equal then the loop is processed again.

apnum.tex

```
\def\apDIVcomp#1#2{%
   \expandafter\def\expandafter\tmpc\expandafter{#1\apNL\apNL\apNL\apNL\apNL\apNL\apNL\apNL@}%
   \expandafter\def\expandafter\tmpd\expandafter{#2\apNL\apNL\apNL\apNL\apNL\apNL\apNL\apNL@}%
   \def\apNext{\expandafter\expandafter\expandafter\apDIVcompA\expandafter\tmpc\tmpd}%
   \apXtrue \apNext
}
\def\apDIVcompA#1#2#3#4#5#6#7#8#9@{%
```

\apDIVcomp: 20–21 \apDIVcompA: 21 \apDIVcompB: 22

```
   \ifx#8\apNL \def\tmpc{0000000\apNL@}\else\def\tmpc{#9@}\fi
   \apnumX=#1#2#3#4#5#6#7#8\relax
   \apDIVcompB
}
\def\apDIVcompB#1#2#3#4#5#6#7#8#9@{%
   \ifnum\apnumX<#1#2#3#4#5#6#7#8 \let\apNext=\relax \apXfalse \else
   \ifnum\apnumX>#1#2#3#4#5#6#7#8 \let\apNext=\relax \apXtrue
   \fi\fi
   \ifx\apNext\relax\else
      \ifx#8\apNL \def\tmpd{0000000\apNL@}\ifx\tmpc\tmpd\let\apNext=\relax\fi
      \else\def\tmpd{#9@}\fi
   \fi
   \apNext
}
```

The format of interleaved data with divisor and remainder is described here. Suppose this partial step of the division process:

```
  R0    R1    R2    R3   ...    Rn        :   d1 d2 d3 ... dn   = ...A...
   @ -A*d1 -A*d2 -A*d3   ... -A*dn                [ R0 R1 : d1 = A ]
   0    N0    N1    N2   ...    N(n-1) Nn
```

The R_k are Digits of the remainder, d_k are Digits of the divisor. The A is calculated Digit in this step. The calculation of the Digits of the new remainder is hinted here. We need to do this from right to left because of the transmissions. This implies, that the interleaved format of \XOUT is in the reverse order and looks like

```
 dn  Rn  ...  d3  R3  d2  R2  d1  R1  @  R0
```

for example for ⟨*paramA*⟩=`1234567893`, ⟨*paramB*⟩=`454502` (in the human readable form) the \XOUT should be `{200}{9300}{4545}{5678}@{1234}` (in the special format). The Digits are separated by TeX braces `{}`. The resulted digit for this step is $A = 12345678/1415 = 2716$.

The calculation of the new remainder takes d_k, R_k, d_{k-1} for each k from n to 0 and creates the Digit of the new remainder $N_{k-1} = R_k - A \cdot d_k$ (roughly speaking, actually it calculates transmissions too) and adds the new couple d_{k-1} N_{k-1} to the new version of \XOUT macro. The zero for N_{-1} should be reached. If it is not completed then a correction of the type $A := A - 1$ have to be done and the calculation of this step is processed again.

The result in the new \XOUT should be (after one step is done):

```
 dn  Nn  ...  d3  N3  d2  N2  d1  N1  @  N0
```

where N_n is taken from the "rest of the dividend" from the input stream.

The initialization for the main loop is done by \apDIVg macro. It reads the Digits from `\tmpa` (dividend) and `\tmpb` macros (using \apIVread) and appends them to the \XOUT in described data format. This initialization is finished when the `\tmpb` is empty. If the `\tmpa` is not empty in such case, we put it to the input stream using `\expandafter\apDIVh\tmpa` followed by four \apNLs (which simply expands zero digit) followed by stop-mark. The \apDIVh reads one Digit from input stream. Else we put only the stop-mark to the input stream and run the \apDIVi. The `\apNexti` is set to the \apDIVi, so the macro \apDIVh will be skipped forever and no new Digit is read from input stream.

apnum.tex

```
\def\apDIVg{%
   \ifx\tmpb\empty
      \ifx\tmpa\empty  \def\apNext{\apDIVi!}\let\apNexti=\apDIVi
      \else \def\apNext{\expandafter\apDIVh\tmpa\apNL\apNL\apNL\apNL!}\let\apNexti=\apDIVh
   \fi\fi
   \ifx\apNext\apDIVg
      \apIVread\tmpa \apnumA=\apnumX
      \apIVread\tmpb
      \edef\XOUT{{\the\apnumX}{\the\apnumA}\XOUT}%
   \fi
   \apNext
}
```

\apDIVg: 20–22

The macro `\apDIVh` reads one Digit from data stream (from the rest of the dividend) and saves it to the `\apnumZ` register. If the stop-mark is reached (this is recognized that the last digit is the `\apNL`), then `\apNexti` is set to `\apDIVi`, so the `\apDIVh` is never processed again.

apnum.tex

```
\def\apDIVh#1#2#3#4{\apnumZ=#1#2#3#4
   \ifx\apNL#4\let\apNexti=\apDIVi\fi
   \apDIVi
}
```

The macro `\apDIVi` contains the main loop for division calculation. The core of this loop is the macro call `\apDIVp`⟨*data*⟩ which adds next digit to the `\OUT` and recalculates the remainder.

The macro `\apDIVp` decreases the `\apnumC` register (the desired digits in the output) by four, because four digits will be calculated in the next step. The loop is processed while `\apnumC` is positive. The `\apnumZ` (new Digit from the input stream) is initialized as zero and the `\nexti` runs the next step of this loop. This step starts from `\apDIVh` (reading one digit from input stream) or directly the `\apDIVi` is repeated. If the remainder from the previous step is calculated as zero (`\apnumE=0`), then we stop prematurely. The `\apDIVj` macro is called at the end of the loop because we need to remove the “rest of the dividend” from the input stream.

apnum.tex

```
\def\apDIVi{%
   \ifnum\apnumE=0 \apnumC=0 \fi
   \ifnum\apnumC>0
      \expandafter\apDIVp\XOUT
      \advance\apnumC by-4
      \apnumZ=0
      \expandafter\apNexti
   \else
      \expandafter\apDIVj
   \fi
}
\def\apDIVj#1!{}
```

The macro `\apDIVp` ⟨*interleaved-data*⟩`@` does the basic setting before the calculation through the expanded `\XOUT` is processed. The `\apDIVxA` includes the “partial dividend” and the `\apDIVxB` includes the “partial divisor”. We need to do `\apDIVxA` over `\apDIVxB` in order to obtain the next digit in the output. This digit is stored in `\apnumA`. The `\apnumX` is the transmission value, the `\apnumB`, `\apnumY` will be the memory of the last two calculated Digits in the remainder. The `\apnumE` will include the maximum of all digits of the new remainder. If it is equal to zero, we can finish the calculation.

The new interleaved data will be stored to the `\apOUT:`⟨*num*⟩ macros in similar way as in the `\MUL` macro. This increases the speed of the calculation. The data `\apnumO`, `\apnumL` and `\apOUTl` for this purpose are initialized.

The `\apDIVq` is started and the tokens `0\apnumZ` are appended to the input stream (i.e to the expanded `\XOUT`. This zero will be ignored and the `\apnumZ` will be used as a new N_n, i.e. the Digit from the “rest of the dividend”.

apnum.tex

```
\def\apDIVp{%
   \apnumA=\apDIVxA \divide\apnumA by\apDIVxB
   \def\apOUTl{}\apnumO=1 \apnumL=0
   \apnumX=0 \apnumB=0 \apnumE=0
   \let\apNext=\apDIVq \apNext 0\apnumZ
}
```

The macro `\apDIVq` $\langle d_k\rangle\,\langle R_k\rangle\,\langle d_{k-1}\rangle$ calculates the Digit of the new remainder N_{k-1} by the formula $N_{k-1} = -A\cdot d_k + R_k - X$ where X is the transmission from the previous Digit. If the result is negative, we need to add minimal number of the form $X\cdot 10000$ in order the result is non-negative. Then the X is new transmission value. The digit N_k is stored in the `\apnumB` register and then it is added to `\apOUT:`⟨*num*⟩ in the order $d_{k-1}\,N_{k-1}$. The `\apnumY` remembers the value of the previous `\apnumB`. The d_{k-1} is put to the input stream back in order it would be read by the next `\apDIVq` call.

If $d_{k-1} =$ `@` then we are at the end of the remainder calculation and the `\apDIVr` is invoked.

\apDIVh: 22–24 \apDIVi: 22–23 \nexti \apDIVj: 23 \apDIVp: 23 \apDIVxA: 20–21, 23–24 \apDIVxB: 20–21, 23 \apDIVq: 23–24

apnum.tex

```
\def\apDIVq#1#2#3{% B A B
   \advance\apnumO by-1 \ifnum\apnumO=0 \apOUTx \fi
   \apnumY=\apnumB
   \apnumB=#1\multiply\apnumB by-\apnumA
   \advance\apnumB by#2\advance\apnumB by-\apnumX
   \ifnum\apnumB<0 \apnumX=\apnumB \advance\apnumX by1
      \divide\apnumX by-\apIVbase \advance\apnumX by1
      \advance\apnumB by\the\apnumX 0000
   \else \apnumX=0 \fi
   \expandafter
      \edef\csname apOUT:\apOUTn\endcsname{\csname apOUT:\apOUTn\endcsname{#3}{\the\apnumB}}%
   \ifnum\apnumE<\apnumB \apnumE=\apnumB \fi
   \ifx@#3\let\apNext=\apDIVr \fi
   \apNext{#3}%
}
```

The \apDIVr macro does the final work after the calculation of new remainder is done. It tests if the remainder is OK, i.e. the transmission from the R_1 calculation is equal to R_0. If it is true then new Digit `\apnumA` is added to the `\OUT` macro else the `\apnumA` is decreased (the correction) and the calculation of the remainder is run again.

If the calculated Digit and the remainder are OK, then we do following:

- The new `\XOUT` is created from `\apOUT:`⟨*num*⟩ macros using `\apOUTs` macro.
- The `\apnumA` is saved to the `\OUT`. This is done with care. If the `\apnumD` (where the decimal point is measured from the actual point in the `\OUT`) is in the interval $[0, 4)$ then the decimal point have to be inserted between digits into the next Digit. This is done by `\apDIVt` macro. If the remainder is zero (`\apnumE=0`), then the right trailing zeros are removed from the Digit by the `\apDIVu` and the shift of the `\apnumD` register is calculated from the actual digits. All this calculation is done in `\tmpa` macro. The last step is adding the contents of `\tmpa` to the `\OUT`.
- The `\apnumD` is increased by the number of added digits.
- The new "partial dividend" is created from `\apnumB` and `\apnumY`.

apnum.tex

```
\def\apDIVr#1#2{%
   \ifnum\apnumX=#2  % the calculated Digit is OK, we save it
      \edef\XOUT{\expandafter\apOUTs\apOUTl.,}%
      \edef\tmpa{\ifnum\apnumF=4 \expandafter\apIVwrite\else \expandafter\the\fi\apnumA}%
      \ifnum\apnumD<\apnumF \ifnum\apnumD>-1 \apDIVt \fi\fi %adding dot
      \ifx\apNexti\apDIVh \apnumE=1 \fi
      \ifnum\apnumE=0 \apDIVu % removing zeros
            \advance\apnumD by-\apNUMdigits\tmpa \relax
      \else \advance\apnumD by-\apnumF \apnumF=4 \fi
      \edef\OUT{\OUT\tmpa}% save the Digit
      \edef\apDIVxA{\the\apnumB\apIVwrite\apnumY}% next partial dvividend
   \else  % we need do correction and run the remainder calculation again
      \advance\apnumA by-1 \apnumX=0 \apnumB=0 \apnumE=0
      \def\apOUTl{}\apnumO=1 \apnumL=0
      \def\apNext{\let\apNext=\apDIVq
      \expandafter\apNext\expandafter0\expandafter\apnumZ\XOUT}%
      \expandafter\apNext
   \fi
}
```

The \apDIVt macro inserts the dot into digits quartet (less than four digits are allowed too) by the `\apnumD` value. This value is assumed in the interval $[0, 4)$. The expandable macro `\apIVdot`⟨*shift*⟩⟨*data*⟩ is used for this purpose. The result from this macro has to be expanded twice.

apnum.tex

```
\def\apDIVt{\edef\tmpa{\apIVdot\apnumD\tmpa}\edef\tmpa{\tmpa}}
```

The \apDIVu macro removes trailing zeros from the right and removes the dot, if it is the last token of the `\tmpa` after removing zeros. It uses expandable macros `\apREMzerosR`⟨*data*⟩ and `\apREMdotR`⟨*data*⟩.

\apDIVr: 23–24 \apDIVt: 24 \apDIVu: 24–25

apnum.tex

```
473: \def\apDIVu{\edef\tmpa{\apREMzerosR\tmpa}\edef\tmpa{\apREMdotR\tmpa}}
```

The rest of the code concerned with the division does an extraction of the last remainder from the data and this value is saved to the `\XOUT` macro in human readable form. The `\apDIVv` macro is called repeatedly on the special format of the `\XOUT` macro and the new `\XOUT` is created. The trailing zeros from right are ignored by the `\apDIVw`.

apnum.tex

```
474: \def\apDIVv#1#2{\apnumX=#2
475:    \ifx@#1\apDIVw{.\apIVwrite\apnumX}\else\apDIVw{\apIVwrite\apnumX}\expandafter\apDIVv\fi
476: }
477: \def\apDIVw#1{%
478:    \ifx\XOUT\empty \ifnum\apnumX=0
479:                     \else \edef\tmpa{#1}\edef\XOUT{\apREMzerosR\tmpa\XOUT}%
480:                     \fi
481:    \else \edef\XOUT{#1\XOUT}\fi
482: }
```

2.7 Power to the Integer

The power to the decimal number (non integer) is not implemented yet because the implementation of exp, ln, etc. is a future work.

We can implement the power to the integer as repeated multiplications. This is simple but slow. The goal of this section is to present the power to the integer with some optimizations.

Let a is the base of the powering computation and $d_1, d_2, d_3, \ldots, d_n$ are binary digits of the exponent (in reverse order). Then

$$p = a^{1\,d_1 + 2\,d_2 + 2^2\,d_3 + \cdots + 2^{n-1}\,d_n} = (a^1)^{d_1} \cdot (a^2)^{d_2} \cdot (a^{2^2})^{d_3} \cdot (a^{2^{n-1}})^{d_n}.$$

If $d_i = 0$ then z^{d_i} is one and this can be omitted from the queue of multiplications. If $d_i = 1$ then we keep z^{d_i} as z in the queue. We can see from this that the p can be computed by the following algorithm:

```
(* "a" is initialized as the base, "e" as the exponent *)
p := 1;
while (e>0) {
   if (e%2) p := p*a;
   e := e/2;
   if (e>0) a := a*a;
}
(* "p" includes the result *)
```

The macro `\apPOWa` does the following work.

- After using `\apPPab` the base parameter is saved in `\tmpa` and the exponent is saved in `\tmpb`.
- In trivial cases, the result is set without any computing (lines 487 and 488).
- If the exponent is non-integer or it is too big then the error message is printed and the rest of the macro is skipped by the `\apPOWe` macro (lines 490 to 493).
- The `\apE` is calculated from `\apEa` (line 494).
- The sign of the result is negative only if the `\tmpb` is odd and base is negative (line 496).
- The number of digits after decimal point for the result is calculated and saved to `\apnumD`. The total number of digits of the base is saved to `\apnumC`. (line 497).
- The first Digit of the base needn't to include all four digits, but other Digits do it. The similar trick as in `\apMULa` is used here (lines 499 to 500).
- The base is saved in interleaved reversed format (like in `\apMULa`) into the `\OUT` macro by the `\apMULb` macro. Let it be the a value from our algorithm described above (lines 501 and 502).
- The initial value of $p = 1$ from our algorithm is set in interleaved format into `\tmpc` macro (line 503).
- The main loop described above is processed by `\apPOWb` macro. (line 504).

- The result in `\tmpc` is converted into human readable form by the `\apPOWg` macro and it is stored into the `\OUT` macro (line 505).
- If the result is negative or decimal point is needed to print then use simple conversion of the `\OUT` macro (adding minus sign) or using `\apROLLa` macro (lines 506 and 507).
- If the exponent is negative then do the $1/r$ calculation, where r is previous result (line 508).

apnum.tex

```
486: \def\apPOWa{%
487:    \ifnum\apSIGNa=0 \def\OUT{0}\apSIGN=0 \apE=0 \else
488:       \ifnum\apSIGNb=0 \def\OUT{1}\apSIGN=1 \apE=0 \else
489:          \apDIG\tmpb\apnumB
490:          \ifnum\apnumB>0 \errmessage{POW: non-integer exponent is not implemented yet}\apPOWe\fi
491:          \ifnum\apEb=0 \else \errmessage{POW: the E notation of exponent isn't allowed}\apPOWe\fi
492:          \ifnum\apnumD>8 \errmessage{POW: too big exponent.
493:                        Do you really need about 10^\the\apnumD\space digits in output?}\apPOWe\fi
494:          \apE=\apEa \multiply\apE by\tmpb\relax
495:          \apSIGN=\apSIGNa
496:          \ifodd\tmpb \else \apSIGN=1 \fi
497:          \apDIG\tmpa\apnumA \apnumC=\apnumA \advance\apnumC by\apnumD
498:          \apnumD=\apnumA \multiply\apnumD by\tmpb
499:          \apIVmod \apnumC \apnumA
500:          \edef\tmpc{\ifcase\apnumA\or{}{}{}\or{}{}\or{}\fi}\def\OUT{}%
501:          \expandafter\expandafter\expandafter \apMULb \expandafter \tmpc \tmpa @@@@%
502:          \edef\OUT{*.\OUT}%    \OUT := \tmpa in interleaved format
503:          \def\tmpc{*.1*}%
504:          \apnumE=\tmpb\relax \apPOWb
505:          \expandafter\apPOWg \tmpc % \OUT := \tmpc in human raedable form
506:          \ifnum\apnumD=0 \ifnum \apSIGN<0 \edef\OUT{-\OUT}\fi
507:          \else \def\tmpc{-\apnumD}\apnumG=\apSIGN \expandafter\apROLLa\OUT.@\OUT\fi
508:          \ifnum\apSIGNb<0 \apPPab\apDIVa 1\OUT \fi
509:          \relax
510:    \fi\fi
511: }
```

The macro `\apPOWb` is the body of the loop in the algorithm described above. The code part after `\ifodd\apnumE` does `p␣:=␣p*a`. In order to do this, we need to convert `\OUT` (where `a` is stored) into normal format using `\apPOWd`. The result is saved in `\tmpb`. Then the multiplication is done by `\apMULd` and the result is normalized by the `\apPOWn` macro. Because `\apMULd` works with `\OUT` macro, we temporary set `\tmpc` to `\OUT`.

The code part after `\ifnum\apnumE<0` does `a␣:=␣a*a` using the `\apPOWt` macro. The result is normalized by the `\apPOWn` macro.

apnum.tex

```
\def\apPOWb{%
   \ifodd\apnumE   \def\tmpb{}\expandafter\apPOWd\OUT
                   \let\tmpd=\OUT \let\OUT=\tmpc
                   \expandafter\apMULd \tmpb@\expandafter\apPOWn\OUT@%
                   \let\tmpc=\OUT \let\OUT=\tmpd
   \fi
   \divide\apnumE by2
   \ifnum\apnumE>0 \expandafter\apPOWt\OUT \expandafter\apPOWn\OUT@%
                   \expandafter\apPOWb
   \fi
}
```

The macro `\apPOWd` ⟨*initialized-interleaved-reversed-format*⟩ extracts the Digits from its argument and saves them to the `\tmpb` macro.

apnum.tex

```
\def\apPOWd#1#2{% \apPOWd <spec format> => \tmpb (in simple reverse format)
   \ifx*#1\expandafter\apPOWd \else
      \edef\tmpb{\tmpb{#1}}%
      \ifx*#2\else \expandafter\expandafter\expandafter\apPOWd\fi
   \fi
}
```

The `\apPOWe` macro skips the rest of the body of the `\apPOWa` macro to the `\relax`. It is used when `\errmessage` is printed.

apnum.tex
```
\def\apPOWe#1\relax{\fi}
```

The `\apPOWg` macro provides the conversion from interleaved reversed format to the human readable form and save the result to the `\OUT` macro. It ignores the first two elements from the format and runs `\apPOWh`.

apnum.tex
```
\def\apPOWg#1#2{\def\OUT{}\apPOWh} % conversion to the human readable form
\def\apPOWh#1#2{\apnumA=#1
   \ifx*#2\edef\OUT{\the\apnumA\OUT}\else \edef\OUT{\apIVwrite\apnumA\OUT}\expandafter\apPOWh\fi
}
```

The normalization to the initialized interleaved format of the `\OUT` is done by the `\apPOWn` ⟨*data*⟩`@` macro. The `\apPOWna` reads the first part of the ⟨*data*⟩ (to the first `*`, where the Digits are non-interleaved. The `\apPOWnn` reads the second part of ⟨*data*⟩ where the Digits of the result are interleaved with the digits of the old coefficients. We need to set the result as a new coefficients and prepare zeros between them for the new calculation. The dot after the first `*` is not printed (the zero is printed instead it) but it does not matter because this token is simply ignored during the calculation.

apnum.tex
```
\def\apPOWn#1{\def\OUT{*}\apPOWna}
\def\apPOWna#1{\ifx*#1\expandafter\apPOWnn\else \edef\OUT{\OUT0{#1}}\expandafter\apPOWna\fi}
\def\apPOWnn#1#2{\ifx*#1\edef\OUT{\OUT*}\else\edef\OUT{\OUT0{#1}}\expandafter\apPOWnn\fi}
```

The powering to two (`\OUT:=\OUT^2`) is provided by the `\apPOWt` ⟨*data*⟩ macro. The macro `\apPOWu` is called repeatedly for each `\apnumA`=Digit from the ⟨*data*⟩. One line of the multiplication scheme is processed by the `\apPOWv` ⟨*data*⟩ macro. We can call the `\apMULe` macro here but we don't do it because a slight optimization is used here. You can try to multiply the number with digits `abcd` by itself in the mirrored multiplication scheme. You'll see that first line includes `a^2␣2ab␣2ac␣2ad`, second line is intended by two columns and includes `b^2␣2bc␣2bd`, next line is indented by next two columns and includes `c^2␣2cd` and the last line is intended by next two columns and includes only `d^2`. Such calculation is slightly shorter than normal multiplication and it is implemented in the `\apPOWv` macro.

apnum.tex
```
\def\apPOWt#1#2{\apPOWu} % power to two
\def\apPOWu#1#2{\apnumA=#1
   \expandafter\apPOWv\OUT
   \ifx*#2\else \expandafter\apPOWu\fi
}
\def\apPOWv#1*#2#3#4{\def\apOUTl{}\apnumO=1 \apnumL=0
   \apnumB=\apnumA \multiply\apnumB by\apnumB \multiply\apnumA by2
   \ifx*#4\else\advance\apnumB by#4 \fi
   \ifx\apnumB<\apIVbase \apnumX=0 \else \apIVtrans \fi
   \edef\OUT{#1{#2}{\the\apnumB}*}%
   \ifx*#4\apMULf0*\else\expandafter\apMULf\fi
}
```

2.8 ROLL, ROUND and NORM Macros

The public macros `\ROLL`, `\ROUND` and `\NORM` are implemented by `\apROLLa`, `\apROUNDa` and `\apNORMa` macros with common format of the parameter text: ⟨*expanded-sequence*⟩`.@`⟨*sequence*⟩ where ⟨*expanded-sequence*⟩ is the expansion of the macro ⟨*sequence*⟩ (given as first parameter of `\ROLL`, `\ROUND` and `\NORM`, but without optionally minus sign. If there was the minus sign then `\apnumG=-1` else `\apnumG=1`. This preparation of the parameter ⟨*sequence*⟩ is done by the `\apPPs` macro. The second parameter of the macros `\ROLL`, `\ROUND` and `\NORM` is saved to the `\tmpc` macro.

`\apROLLa` ⟨*param*⟩`.@`⟨*sequence*⟩ shifts the decimal point of the ⟨*param*⟩ by `\tmpc` positions to the right (or to the left, if `\tmpc` is negative) and saves the result to the ⟨*sequence*⟩ macro. The `\tmpc` value is saved to the `\apnumA` register and the `\apROLLc` is executed if we need to shift the decimal point to left. Else `\apROLLg` is executed.

apnum.tex
```
\def\apROLLa{\apnumA=\tmpc\relax \ifnum\apnumA<0 \expandafter\apROLLc\else \expandafter\apROLLg\fi}
```

The `\apROLLc` ⟨*param*⟩`.@`⟨*sequence*⟩ shifts the decimal point to left by the `-\apnumA` decimal digits. It reads the tokens from the input stream until the dot is found using `\apROLLd` macro. The number of such tokens is set to the `\apnumB` register and tokens are saved to the `\tmpc` macro. If the dot is found then `\apROLLe` does the following: if the number of read tokens is greater then the absolute value of the ⟨*shift*⟩, then the number of positions from the most left digit of the number to the desired place of the dot is set to the `\apnumA` register a the dot is saved to this place by `\apROLLi`⟨*parameter*⟩`.@`⟨*sequence*⟩. Else the new number looks like `.000123` and the right number of zeros are saved to the ⟨*sequence*⟩ using the `\apADDzeros` macro and the rest of the input stream (including expanded `\tmpc` returned back) is appended to the macro ⟨*sequence*⟩ by the `\apROLLf` ⟨*param*⟩`.@` macro.

apnum.tex
```
\def\apROLLc{\edef\tmpc{}\edef\tmpd{\ifnum\apnumG<0-\fi}\apnumB=0 \apROLLd}
\def\apROLLd#1{%
   \ifx.#1\expandafter\apROLLe
   \else \edef\tmpc{\tmpc#1}%
      \advance\apnumB by1
      \expandafter\apROLLd
   \fi
}
\def\apROLLe#1{\ifx@#1\edef\tmpc{\tmpc.@}\else\edef\tmpc{\tmpc#1}\fi
   \advance\apnumB by\apnumA
   \ifnum\apnumB<0
       \apnumZ=-\apnumB \edef\tmpd{\tmpd.}\apADDzeros\tmpd
       \expandafter\expandafter\expandafter\apROLLf\expandafter\tmpc
   \else
       \apnumA=\apnumB
       \expandafter\expandafter\expandafter\apROLLi\expandafter\tmpc
   \fi
}
\def\apROLLf#1.@#2{\edef#2{\tmpd#1}}
```

The `\apROLLg` ⟨*param*⟩`.@`⟨*sequence*⟩ shifts the decimal point to the right by `\apnumA` digits starting from actual position of the input stream. It reads tokens from the input stream by the `\apROLLh` and saves them to the `\tmpd` macro where the result will be built. When dot is found the `\apROLLi` is processed. It reads next tokens and decreases the `\apnumA` by one for each token. It ends (using `\apROLLj\apROLLk`) when `\apnumA` is equal to zero. If the end of the input stream is reached (the `@` character) then the zero is inserted before this character (using `\apROLLj\apROLLi0@`). This solves the situations like `123`, ⟨*shift*⟩`=2`, → `12300`.

apnum.tex
```
\def\apROLLg#1{\edef\tmpd{\ifnum\apnumG<0-\fi}\ifx.#1\apnumB=0 \else\apnumB=1 \fi \apROLLh#1}
\def\apROLLh#1{\ifx.#1\expandafter\apROLLi\else \edef\tmpd{\tmpd#1}\expandafter\apROLLh\fi}
\def\apROLLi#1{\ifx.#1\expandafter\apROLLi\else
   \ifnum\apnumA>0 \else \apROLLj \apROLLk#1\fi
   \ifx@#1\apROLLj \apROLLi0@\fi
   \advance\apnumA by-1
   \ifx0#1\else \apnumB=1 \fi
   \ifnum\apnumB>0 \edef\tmpd{\tmpd#1}\fi
   \expandafter\apROLLi\fi
}
```

The `\apROLLg` macro initializes `\apnumB=1` if the ⟨*param*⟩ doesn't begin by dot. This is a flag that all digits read by `\apROLLi` have to be saved. If the dot begins, then the number can look like `.000123` (before moving the dot to the right) and we need to ignore the trailing zeros. The `\apnumB` is equal to zero in such case and this is set to `1` if here is first non-zero digit.

The `\apROLLj` macro closes the conditionals and runs its parameter separated by `\fi`. It skips the rest of the `\apROLLi` macro too.

apnum.tex
```
\def\apROLLj#1\fi#2\apROLLi\fi{\fi\fi#1}
```

The macro \apROLLk puts the decimal point to the \tmpd at current position (using \apROLLn) if the input stream is not fully read. Else it ends the processing. The result is an integer without decimal digit in such case.

apnum.tex
```
\def\apROLLk#1{\ifx@#1\expandafter\apROLLo\expandafter@\else
   \def\tmpc{}\apnumB=0 \expandafter\apROLLn\expandafter#1\fi
}
```

The macro \apROLLn reads the input stream until the dot is found. Because we read now the digits after a new position of the decimal point we need to check situations of the type 123.000 which is needed to be written as 123 without decimal point. This is a reason of a little complication. We save all digits to the \tmpc macro and calculate the sum of such digits in \apnumB register. If this sum is equal to zero then we don't append the .\tmpc to the \tmpd. The macro \apROLLn is finished by the \apROLLo @⟨*sequence*⟩ macro, which removes the last token from the input stream and defines ⟨*sequence*⟩ as \tmpd.

apnum.tex
```
\def\apROLLn#1{%
   \ifx.#1\ifnum\apnumB>0 \edef\tmpd{\tmpd.\tmpc}\fi \expandafter\apROLLo
   \else \edef\tmpc{\tmpc#1}\advance\apnumB by#1 \expandafter\apROLLn
   \fi
}
\def\apROLLo@#1{\let#1=\tmpd}
```

The macro \apROUNDa ⟨*param*⟩.@⟨*sequence*⟩ rounds the number given in the ⟨*param*⟩. The number of digits after decimal point \tmpc is saved to \apnumD. If this number is negative then \apROUNDe is processed else the \apROUNDb reads the ⟨*param*⟩ to the decimal point and saves this part to the \tmpc macro. The \tmpd macro (where the rest after decimal point of the number will be stored) is initialized to empty and the \apROUNDc is started. This macro reads one token from input stream repeatedly until the number of read tokens is equal to \apnumD or the stop mark @ is reached. All tokens are saved to \tmpd. Then the \apROUNDd macro reads the rest of the ⟨*param*⟩, saves it to the \XOUT macro and defines ⟨*sequence*⟩ (i.e. #2) as the rounded number.

apnum.tex
```
\def\apROUNDa{\apnumD=\tmpc\relax
   \ifnum\apnumD<0 \expandafter\apROUNDe
   \else \expandafter\apROUNDb
   \fi
}
\def\apROUNDb#1.{\edef\tmpc{#1}\apnumX=0 \def\tmpd{}\let\apNext=\apROUNDc \apNext}
\def\apROUNDc#1{\ifx@#1\def\apNext{\apROUNDd.@}%
   \else \advance\apnumD by-1
         \ifnum\apnumD<0 \def\apNext{\apROUNDd#1}%
         \else \ifx.#1\else \advance\apnumX by#1 \edef\tmpd{\tmpd#1}\fi
         \fi
   \fi \apNext
}
\def\apROUNDd#1.@#2{\def\XOUT{#1}%
   \ifnum\apnumX=0 \def\tmpd{}\fi
   \ifx\tmpd\empty
      \ifx\tmpc\empty \def#2{0}%
      \else \edef#2{\ifnum\apnumG<0-\fi\tmpc}\fi
   \else\edef#2{\ifnum\apnumG<0-\fi\tmpc.\tmpd}\fi
}
```

The macro \apROUNDe solves the "less standard" problem when rounding to the negative digits after decimal point \apnumD, i.e. we need to set -\apnumD digits before decimal point to zero. The solution is to remove the rest of the input stream, use \apROLLa to shift the decimal point left by -\apnumD positions, use \apROUNDa to remove all digits after decimal point and shift the decimal point back to its previous place.

apnum.tex
```
\def\apROUNDe#1.@#2{\apnumC=\apnumD
   \apPPs\apROLLa#2{\apnumC}\apPPs\apROUNDa#2{0}\apPPs\apROLLa#2{-\apnumC}%
}
```

The macro \apNORMa redefines the ⟨*sequence*⟩ in order to remove minus sign because the \apDIG macro uses its parameter without this sign. Then the \apNORMb ⟨*sequence*⟩⟨*parameter*⟩@ is executed where the dot in the front of the parameter is tested. If the dot is here then the \apDIG macro measures the digits after decimal point too and the \apNORMc is executed (where the \apROLLa shifts the decimal point from the right edge of the number). Else the \apDIG macro doesn't measure the digits after decimal point and the \apNORMd is executed (where the \apROLLa shifts the decimal point from the left edge of the number).

apnum.tex

```
\def\apNORMa#1.@#2{\ifnum\apnumG<0 \def#2{#1}\fi \expandafter\apNORMb\expandafter#2\tmpc@}
\def\apNORMb#1#2#3@{%
   \ifx.#2\apnumC=#3\relax  \apDIG#1\apnumA \apNORMc#1%
   \else \apnumC=#2#3\relax \apDIG#1\relax  \apNORMd#1%
   \fi
}
\def\apNORMc#1{\advance\apE by-\apnumA \advance\apE by\apnumC
   \def\tmpc{-\apnumC}\expandafter\apROLLa#1.@#1%
}
\def\apNORMd#1{\advance\apE by\apnumD \advance\apE by-\apnumC
   \def\tmpc{\apnumC}\expandafter\apROLLa\expandafter.#1.@#1%
}
```

2.9 Function-like Macros

The internal implementation of function-like macros \ABS, \iDIV etc. are simple. The \apFACa macro (factorial) doesn't use recursive call because the TEX group is opened in such case and the number of levels of TEX group is limited (to 255 at my computer). But we want to calculate more factorial than only 255!.

apnum.tex

```
\def\apABSa{\ifnum\apSIGN<0 \apSIGN=1 \fi}
\def\apiDIVa{{\apFRAC=0 \apTOT=0 \apDIVa \apOUTtmpb}\tmpb}
\def\apiMODa{{\apFRAC=0 \apTOT=0 \apDIVa \let\OUT=\XOUT \apOUTtmpb}\tmpb}
\def\apiROUNDa{\apROUNDa\OUT0}
\def\apiFRACa{\apROUNDa\OUT0\ifx\XOUT\empty\def\OUT{0}\else\edef\OUT{.\XOUT}\fi}
\def\apFACa{{\apnumC=\OUT\relax
   \loop \ifnum \apnumC>2 \advance\apnumC by-1
         \MUL{\OUT}{\the\apnumC}\repeat
   \global\let\OUT=\OUT}%
}
```

2.10 Auxiliary Macros

The macro \apREV {⟨*tokens*⟩} reverses the order of the ⟨*tokens*⟩. For example \apREV{revers} expands to srever. The macro uses \apREVa and works at expansion level only.

apnum.tex

```
\def\apREV#1{\expandafter\apREVa#1@!}
\def\apREVa#1#2!{\ifx@#1\else\apREVa#2!#1\fi}
```

The macro \apDIG ⟨*sequence*⟩⟨*register-or-relax*⟩ reads the content of the macro ⟨*sequence*⟩ and counts the number of digits in this macro before decimal point and saves it to \apnumD register. If the macro ⟨*sequence*⟩ includes decimal point then it is redefined with the same content but without decimal point. The numbers in the form .00123 are replaced by 123 without zeros, but \apnumD=-2 in this example. If the second parameter of the \apDIG macro is \relax then the number of digits after decimal point isn't counted. Else the number of these digits is stored to the given ⟨*register*⟩.

The macro \apDIG is developed in order to do minimal operations over a potentially long parameters. It assumes that ⟨*sequence*⟩ includes a number without ⟨*sign*⟩ and without left trailing zeros. This is true after parameter preparation by the \apPPab macro.

The macro \apDIG prepares an incrementation in \tmpc if the second parameter ⟨*register*⟩ isn't \relax. It initializes \apnumD and ⟨*register*⟩. It runs \apDIGa ⟨*data*⟩..@⟨*sequence*⟩ which increments the `\apnumD` until the dot is found. Then the `\apDIGb` is executed (if there are no digits before dot) or the `\apDIGc` is called (if there is at least one digit before dot). The `\apDIGb` ignores zeros immediately after dot. The `\apDIGc` reads the rest of the ⟨*data*⟩ to the `#1` and saves it to the `\tmpd` macro. It runs the counter over this ⟨*data*⟩ `\apDIGd` ⟨*data*⟩`@` only if it is desired (`\tmpc` is non-empty). Else the `\apDIGe` is executed. The `\apDIGe` ⟨*dot-or-nothing*⟩`@`⟨*sequence*⟩ redefines ⟨*sequence*⟩ if it is needed. Note, that `#1` is empty if and only if the ⟨*data*⟩ include no dot (first dot was reached as the first dot from `\apDIG`, the second dot from `\apDIG` was a separator of `#1` in `\apDIGc` and there is nothing between the second dot and the `@` mark. The ⟨*sequence*⟩ isn't redefined if it doesn't include a dot. Else the sequence is set to the `\tmpd` (the rest after dot) if there are no digits before dot. Else the sequence is redefined using expandable macro `\apDIGf`.

apnum.tex

```
\def\apDIG#1#2{\ifx\relax#2\def\tmpc{}\else #2=0 \def\tmpc{\advance#2 by1 }\fi
   \apnumD=0 \expandafter\apDIGa#1..@#1%
}
\def\apDIGa#1{\ifx.#1\csname apDIG\ifnum\apnumD>0 c\else b\fi\expandafter\endcsname
   \else \advance\apnumD by1 \expandafter\apDIGa\fi}
\def\apDIGb#1{%
   \ifx0#1\advance\apnumD by-1 \tmpc \expandafter\apDIGb
   \else \expandafter\apDIGc \expandafter#1\fi
}
\def\apDIGc#1.{\def\tmpd{#1}%
   \ifx\tmpc\empty \let\apNext=\apDIGe
   \else \def\apNext{\expandafter\apDIGd\tmpd@}%
   \fi \apNext
}
\def\apDIGd#1{\ifx@#1\expandafter\apDIGe \else \tmpc \expandafter\apDIGd \fi}
\def\apDIGe#1@#2{%
   \ifx@#1@\else  % #1=empty <=> the param has no dot, we need to do nothing
      \ifnum\apnumD>0 \edef#2{\expandafter\apDIGf#2@}% the dot plus digits before dot
      \else \let#2=\tmpd  % there are only digits after dot, use \tmpd
   \fi\fi
}
\def\apDIGf#1.#2@{#1#2}
```

The macro `\apIVread` ⟨*sequence*⟩ reads four digits from the macro ⟨*sequence*⟩, sets `\apnumX` as the Digit consisting from read digits and removes the read digits from ⟨*sequence*⟩. It internally expands ⟨*sequence*⟩, adds the `\apNL` marks and runs `\apIVreadA` macro which sets the `\apnumX` and redefines ⟨*sequence*⟩.

The usage of the `\apNL` as a stop-marks has the advantage: they act as simply zero digits in the comparison but we can ask by `\ifx` if this stop mark is reached. The `#5` parameter of `\apIVreadA` is separated by first occurrence of `\apNL`, i.e. the rest of the macro ⟨*sequence*⟩ is here.

apnum.tex

```
\def\apNL{0}
\def\apIVread#1{\expandafter\apIVreadA#1\apNL\apNL\apNL\apNL\apNL@#1}
\def\apIVreadA#1#2#3#4#5\apNL#6@#7{\apnumX=#1#2#3#4\relax \def#7{#5}}
```

The macro `\apIVreadX` ⟨*num*⟩⟨*sequence*⟩ acts similar as `\apIVread`⟨*sequence*⟩, but only ⟨*num*⟩ digits are read. The ⟨*num*⟩ is expected in the range 0 to 4. The macro prepares the appropriate number of empty parameters in `\tmpc` and runs `\apIVreadA` with these empty parameters inserted before the real body of the ⟨*sequence*⟩.

apnum.tex

```
\def\apIVreadX#1#2{\edef\tmpc{\ifcase#1{}{}{}0\or{}{}{}\or{}{}\or{}\fi}%
   \expandafter\expandafter\expandafter\apIVreadA\expandafter\tmpc#2\apNL\apNL\apNL\apNL\apNL@#2%
}
```

The macro `\apIVwrite` ⟨*num*⟩ expands the digits from ⟨*num*⟩ register. The number of digits are four. If the ⟨*num*⟩ is less than `1000` then left zeros are added.

apnum.tex

```
\def\apIVwrite#1{\ifnum#1<1000 0\ifnum#1<100 0\ifnum#1<10 0\fi\fi\fi\the#1}
```

The macro `\apIVtrans` calculates the transmission for the next Digit. The value (greater or equal 10000) is assumed to be in `\apnumB`. The new value less than 10000 is stored to `\apnumB` and the transmission value is stored in `\apnumX`. The constant `\apIVbase` is used instead of literal 10000 because it is quicker.

apnum.tex
```
\mathchardef\apIVbase=10000
\def\apIVtrans{\apnumX=\apnumB \divide\apnumB by\apIVbase \multiply\apnumB by-\apIVbase
   \advance\apnumB by\apnumX \divide\apnumX by\apIVbase
}
```

The macro `\apIVmod` ⟨*length*⟩⟨*register*⟩ sets ⟨*register*⟩ to the number of digits to be read to the first Digit, if the number has ⟨*length*⟩ digits in total. We need to read all Digits with four digits, only first Digit can be shorter.

apnum.tex
```
\def\apIVmod#1#2{#2=#1\divide#2by4 \multiply#2by-4 \advance#2by#1\relax
```

The macro `\apIVdot` ⟨*num*⟩⟨*param*⟩ adds the dot into ⟨*param*⟩. Let $K = \langle num\rangle$ and F is the number of digits in the ⟨*param*⟩. The macro expects that $K \in [0, 4)$ and $F \in (0, 4]$. The macro inserts the dot after K-th digit if $K < F$. Else no dot is inserted. It is expandable macro, but two full expansions are needed. After first expansion the result looks like `\apIVdotA`⟨*dots*⟩⟨*param*⟩`....@` where ⟨*dots*⟩ are the appropriate number of dots. Then the `\apIVdotA` reads the four tokens (maybe the generated dots), ignores the dots while printing and appends the dot after these four tokens, if the rest `#5` is non-empty.

apnum.tex
```
\def\apIVdot#1#2{\noexpand\apIVdotA\ifcase#1....\or...\or..\or.\fi #2....@}
\def\apIVdotA#1#2#3#4#5.#6@{\ifx.#1\else#1\fi
   \ifx.#2\else#2\fi \ifx.#3\else#3\fi \ifx.#4\else#4\fi\ifx.#5.\else.#5\fi
}
```

The expandable macro `\apNUMdigits` `{`⟨*param*⟩`}` expands (using the `\apNUMdigitsA` macro) to the number of digits in the ⟨*param*⟩. We assume that maximal number of digits will be four.

apnum.tex
```
\def\apNUMdigits#1{\expandafter\apNUMdigitsA#1@@@@!}
\def\apNUMdigitsA#1#2#3#4#5!{\ifx@#4\ifx@#3\ifx@#2\ifx@#10\else1\fi \else2\fi \else3\fi \else4\fi}
```

The macro `\apADDzeros` ⟨*sequence*⟩ adds `\apnumZ` zeros to the macro ⟨*sequence*⟩.

apnum.tex
```
\def\apADDzeros#1{\edef#1{#10}\advance\apnumZ by-1
   \ifnum\apnumZ>0 \expandafter\apADDzeros\expandafter#1\fi
}
```

The expandable macro `\apREMzerosR` `{`⟨*param*⟩`}` removes right trailing zeros from the ⟨*param*⟩. It expands to `\apREMzerosRa`⟨*param*⟩`@0@!`. The macro `\apREMzerosRa` reads all text terminated by `0@` to `#1`. This termination zero can be the most right zero of the ⟨*param*⟩ (then `#2` is non-empty) or ⟨*param*⟩ hasn't such zero digit (then `#2` is empty). If `#2` is non-empty then the `\apREMzerosRa` is expanded again in the recursion. Else `\apREMzerosRb` removes the stop-mark `@` and the expansion is finished.

apnum.tex
```
\def\apREMzerosR#1{\expandafter\apREMzerosRa#1@0@!}
\def\apREMzerosRa#10@#2!{\ifx!#2!\apREMzerosRb#1\else\apREMzerosRa#1@0@!\fi}
\def\apREMzerosRb#1@{#1}
```

The expandable macro `\apREMdotR` `{`⟨*param*⟩`}` removes right trailing dot from the ⟨*param*⟩ if exists. It expands to `\apREMdotRa` and works similarly as the `\apREMzerosR` macro.

apnum.tex
```
\def\apREMdotR#1{\expandafter\apREMdotRa#1@.@!}
\def\apREMdotRa#1.@#2!{\ifx!#2!\apREMzerosRb#1\else#1\fi}
```

The writing to the `\OUT` in the `\MUL`, `\DIV` and `\POW` macros is optimized, which decreases the computation time with very large numbers ten times and more. We can do simply `\edef\OUT{\OUT`⟨*something*⟩`}` instead of

```
\expandafter\edef\csname apOUT:\apOUTn\endcsname
                 {\csname apOUT:\apOUTn\endcsname<something>}%
```

but \edef\OUT{\OUT⟨*something*⟩} is typically processed very often over possibly very long macro (many thousands of tokens). It is better to do \edef over more short macros \apOUT:0, \apOUT:1, etc. Each such macro includes only 7 Digits pairs of the whole \OUT. The macro \apOUTx is invoked each 7 digit (the \apnumO register is decreased). It uses \apnumL value which is the ⟨*num*⟩ part of the next \apOUT:⟨*num*⟩ control sequence. The \apOUTx defines this ⟨*num*⟩ as \apOUTn and initializes \apOUT:⟨*num*⟩ as empty and adds the ⟨*num*⟩ to the list \apOUTl. When the creating of the next \OUT macro is definitely finished, the \OUT macro is assembled from the parts \apOUT:0, \apOUT:1 etc. by the macro \apOUTs ⟨*list-of-numbers*⟩⟨*dot*⟩⟨*comma*⟩.

apnum.tex
```
\def\apOUTx{\apnumO=7
   \edef\apOUTn{\the\apnumL}\edef\apOUTl{\apOUTl\apOUTn,}%
   \expandafter\def\csname apOUT:\apOUTn\endcsname{}%
   \advance\apnumL by1
}
\def\apOUTs#1,{\ifx.#1\else\csname apOUT:#1\expandafter\endcsname\expandafter\apOUTs\fi}
```

The macro \apOUTtmpb is used in the context {...\apOUTtmpb}\tmpb. It saves the results \OUT, \apE and \apSIGN calculated in the TeX group in the \tmpb macro, expands the \tmpb, ends the TeX group and executes the \tmpb in order to make possible to use these results outside this group.

apnum.tex
```
\def\apOUTtmpb{\edef\tmpb{\apSIGN=\the\apSIGN \apE=\the\apE \edef\noexpand\OUT{\OUT}}\expandafter}
```

2.11 Conclusion

Here is my little joke. Of course, this macro file works in LaTeX without problems because only TeX primitives (from classic TeX) and the \newcount macro are used here. But I wish to print my opinion about LaTeX. I hope that this doesn't matter and LaTeX users can use my macro because a typical LaTeX user doesn't read a terminal nor .log file.

apnum.tex
```
\ifx\documentclass\undefined \else % please, don't remove this message
\message{SORRY, you are using LaTeX. I don't recommend this. Petr Olsak}\fi
\catcode`\@=\apnumZ
\endinput
```

3 Index

The bold number is the number of the page where the item is documented. Other numbers are pagenumbers of the occurrences of such item.

\apOUTx: 17–18, 24, 33 \apOUTn: 17–18, 24, 33 \apOUTl: 17, 23–24, 27, 33 \apOUTs: 17, 24, 33
\apOUTtmpb: 11, 30, 33