apnum.tex

version 1.7 Apr 2018

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, power to an integer and other calculation ($\sqrt{x}$, e^x, $\ln x$, $\sin x$, $\arctan x$, ...) with "large 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 decimal ⟨*digits*⟩ (but no 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 the `apnum.tex` implementation includes a lot of optimization 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 .3333333333333333333
```

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 in total 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=0` and `\apFRAC=20`.

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 (roughly speaking) the ⟨*expression*⟩ in the form "operand binary-operator operand binary-operator etc." without expansion. The spaces are not significant in the ⟨*expression*⟩. The operands are:

- numbers (in the format ⟨*sign*⟩⟨*digits*⟩`.`⟨*digits*⟩) or
- numbers in scientific notation (see the section 1.2) 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*⟩`}`. This case has two meanings:
  - numeric constant defined in a "literal macro" (`\def\A{42}`, `\evaldef\A{13/15}`) or
  - "function-like" macro which returns a value after processing.

The `apnum.tex` macro file provides the following "function-like" macros allowed to use them as an operand in the ⟨*expression*⟩:

- `\ABS {`⟨*value*⟩`}` for absolute value,
- `\SGN {`⟨*value*⟩`}` returns sign of the ⟨*value*⟩,
- `\iDIV {`⟨*dividend*⟩`}{`⟨*divisor*⟩`}` for integer division,
- `\iMOD {`⟨*dividend*⟩`}{`⟨*divisor*⟩`}` for integer remainder,
- `\iFLOOR {`⟨*value*⟩`}` for rounding the number to the integer,
- `\iFRAC {`⟨*value*⟩`}` for fraction part of the `\iFLOOR`,
- `\FAC {`⟨*integer value*⟩`}` for factorial,
- `\BINOM {`⟨*integer above*⟩`}{`⟨*integer below*⟩`}` for binomial coefficient,
- `\SQRT {`⟨*value*⟩`}` for square root of the ⟨*value*⟩,
- `\EXP {`⟨*value*⟩`}` applies exponential function to ⟨*value*⟩,
- `\LN {`⟨*value*⟩`}` for natural logarithm of the ⟨*value*⟩,
- `\SIN {`⟨*value*⟩`}`, `\COS {`⟨*value*⟩`}`, `\TAN {`⟨*value*⟩`}` for $\sin x$, $\cos x$ and $\tan x$ functions,
- `\ASIN {`⟨*value*⟩`}`, `\ACOS {`⟨*value*⟩`}`, `\ATAN {`⟨*value*⟩`}` for $\arcsin x$, $\arccos x$ and $\arctan x$ functions,
- `\PI`, `\PIhalf` for constants π and $\pi/2$.

The arguments of all these functions can be a nested ⟨*expressions*⟩ with the syntax like in the `\evaldef` macro. Example:

```
\def\A{20}
\evaldef\B{ 30*\SQRT{ 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 macros `\SQRT`, `\EXP`, `\LN`, `\SIN`, `\COS`, `\TAN`, `\ASIN`, `\ACOS`, `\ATAN` use `\apTOT` and `\apFRAC` registers similar like during division.

The `\PI` and `\PIhalf` are "function-like" macros without parameters. They returns the constant with `\apFRAC` digits after decimal point.

Users can define their own "function-like" macros, see the section 1.3.

The output of `\evaldef\foo{`⟨*expression*⟩`}` processing is stored, of course, to the "literal macro" `\foo`. But there are another outputs like side effect of the processing:

- The `\OUT` macro includes exactly the same result as `\foo`.
- The `\apSIGN` register includes the value `1` or `0` or `-1` respectively dependent on the fact that the output is positive, zero or negative.
- The `\apE` register is equal to the decimal exponent when scientific number format is used (see the next section 1.2).

For example, you can compare long numbers using `\apSIGN` register (where direct usage of `\ifnum` primitive may cause arithmetic overflow):

```
\TEST {1234567891234567 89} > {1234567891234567 88} \iftrue OK \else KO \fi
```

The `\TEST` macro is defined like:

```
\def\TEST#1#2#3#4{\evaldef\tmp{#1-(#3)}\ifnum\apSIGN #2 0 }
```

The `apnum.tex` macros do not provide the evaluation of the ⟨*expression*⟩ at the expansion level only. There are two reasons. First, the macros can be used in classical TEX only with Knuth's plain TEX macro. No eTEX is needed. And the expansion-only evaluation of any expression isn't possible in classical TEX. Second reason is the speed optimization (see the section 1.5). Anyway, users needn't expansion-only evaluation. They can write `\evaldef\a{`⟨*expression*⟩`} \edef\foo{...\a...}` instead of `\edef\foo{...`⟨*expression*⟩`...}`. There is only one case when this "pre-processing" trick cannot be used: while expansion of the parameters of asynchronous `\write` commands. But you can save the ⟨*expression*⟩ unexpanded into the file and you can read the file again in the second step and do `\evaldef` during reading the file.

1.2 Scientific Notation of Numbers

The macro `\evaldef` is 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 `\evaldef`⟨*sequence*⟩`{`⟨*expression*⟩`}` operates with mantissa and exponent separately if there are operands with scientific notation. It outputs the result in the scientific notation if the result have non-zero exponent.

The `\evalmdef`⟨*sequence*⟩`{`⟨*expression*⟩`}` does the same as `\evaldef` but only mantissa is saved in the output ⟨*sequence*⟩ and in the `\OUT` macro. The exponent is stored in the `\apE` register in such case. You can define the macro which shows the complete result after `\evalmdef` calculation, for example:

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

Suppose `\evalmdef\foo{`⟨*expression*⟩`}` is processed and the complete result is $R =$ `\foo*10^\apE`. There are two possibilities how to save such complete result R to the `\foo` macro: use `\apEadd\foo` or `\apEnum\foo`. Both macros do nothing if `\apE=0`. Else the `\apEadd`⟨*sequence*⟩ macro adds `E`⟨*exponent*⟩ to the ⟨*sequence*⟩ macro and `\apEnum`⟨*sequence*⟩ moves the decimal point to the new right position in

the ⟨*sequence*⟩ macro or appends zeros. The \apE register is set to zero after the macro \apEadd or \apEnum is finished. Example:

```
\evalmdef\foo{ 3 * 4E9 }    % \foo is 12, \apE=9
\apEadd\foo                 % \foo is 12E+9
\evalmdef\foo{ 7E9 + 5E9 }  % \foo is 12, \apE=9
\apEnum\foo                 % \foo is 12000000000
```

There are another usable macros for operations with scientific numbers.

- \apROLL ⟨*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 the \apEnum\A does \apROLL\A{\apE}.
- \apNORM ⟨*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 \apNORM\A{.0} defines \A as 1234 and \apE=-3.
- The \apROUND ⟨*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 decimal point is removed, if it is the right most character in the \OUT. The ignored part is saved to the \XOUT macro without trailing right zeros.

  Examples of \apROUND usage:

```
\def\A{12.3456}\apROUND\A{1}   % \A is "12.3",     \XOUT is "456"
\def\A{12.3456}\apROUND\A{9}   % \A is "12.3456",  \XOUT is empty
\def\A{12.3456}\apROUND\A{0}   % \A is "12",       \XOUT is "3456"
\def\A{12.0000}\apROUND\A{0}   % \A is "12",       \XOUT is empty
\def\A{12.0001}\apROUND\A{2}   % \A is "12",       \XOUT is "01"
\def\A{.000010}\apROUND\A{2}   % \A is "0",        \XOUT is "001"
\def\A{-12.3456}\apROUND\A{2}  % \A is "-12.34",   \XOUT is "56"
\def\A{12.3456}\apROUND\A{-1}  % \A is "10",       \XOUT is "23456"
\def\A{12.3456}\apROUND\A{-4}  % \A is "0",        \XOUT is "00123456"
```

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

```
\evalmdef\X{...}\apNORM\X{1}\apROUND\X{3}\apEadd\X
```

The macros \apEadd, \apEnum, \apROLL, \apNORM and \apROUND redefine the macro ⟨*sequence*⟩ given as their first argument. They are not "function-like" macros and they cannot be used in an ⟨*expression*⟩. The macro ⟨*sequence*⟩ must be 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 \evalmdef macro.

The build in function-like macros \SGN, \iDIV, ... \SIN, \COS, \ATAN etc. don't generate the result in scientific form regardless of its argument is in scientific form or not. But there are exceptions: \ABS and \SQRT returns scientific form if the argument is in this form. And \EXP returns scientific form if the result is greater than 10^{K+1} or less than 10^{-K-1} where $K =$ \apEX. The default value of this register is \apEX=10.

1.3 Notes for macro programmers

If you plan to create a "function-like" macro which can be used as an operand in the ⟨*expression*⟩ then observe that first token in the macro body must be \relax. This tells to the ⟨*expression*⟩ scanner

that the calculation follows. The result of this calculation must be saved into the \OUT macro and into the \apSIGN register.

Example. The \ABS macro for the absolute value is defined by:

apnum.tex

```
706: \def\ABS#1{\relax        % mandatory \relax for "function-like" macros
707:    \evalmdef\OUT{#1}%    % evaluation of the input parameter
708:    \ifnum\apSIGN<0       % if (input < 0)
709:       \apSIGN=1          %   sign = 1
710:       \apREMfirst\OUT    %   remove first "minus" from OUT
711:    \fi                   % fi
```

```
Usage: \evaldef\A{ 2 - \ABS{3-10} }%  \A includes -5.
```

Note, that \apSIGN register is corrected by final routine of the expression scanner according the \OUT value. But setting \apSIGN in your macro is recommended because user can use your macro directly outside of \evaldef.

If the result of the function-like macro needs to be expressed by scientific notation then you have two possibilities: use "E" notation in the \OUT macro and keep \apE register zero. Or save the matissa only to the \OUT macro and set the value of the exponent into the \apE register. The second possibility is preferred and used by build in function-like macros. Note the \ABS definition above: the \evalmdef in the line 707 keeps only mantissa in the \OUT macro and the \apE register is set by \evalmdef itself.

The \evaldef\foo{⟨*expression*⟩} is processed in two steps. The ⟨*expression*⟩ scanner converts the input to the macro call of the \apPLUS, \apMINUS, \apMUL, \apDIV or \apPOW macros with two parameters. They do addition, subtraction, multiplication, division and power to the integer. These macros are processed in the second step. For example:

```
\evaldef\A{ 2 - 3*8 }      converts the input to:
\apMINUS{2}{\apMUL{3}{8}}  and this is processed in the second step.
```

The macros \apPLUS, \apMINUS, \apMUL, \apDIV and \apPOW behave like normal "function-like" macros with one important exception: they don't accept general ⟨*expression*⟩ in their parameters, only single operand (see section 1.1) is allowed.

If your calculation is processed in the loop very intensively than it is better to save time of such calculation and to avoid the ⟨*expression*⟩ scanner processing (first step of the \evaldef). So, it is recommended to use directly the Polish notation of the expression as shown in the second line in the example above. See section 2.10 for more inspirations.

The output of the \apPLUS, \apMINUS, \apMUL, \apDIV and \apPOW macros is stored in \OUT macro and the registers \apSIGN and \apE are set accordingly.

The number of digits calculated by \apDIV macro is limited by the \apTOT and \apFRAC registers as described in the section 1.1. There is another result of \apDIV calculation stored in the \XOUT macro. It is the remainder of the division. Example:

```
\apTOT=0 \apFRAC=0 \apDIV{1234567892345}{2}\ifnum\XOUT=0 even \else odd\fi
```

You cannot apply \ifodd primitive on "large numbers" directly because the numbers may be too big.

If you set something locally inside your "function-like" macro, then such data are accessible only when your macro is called outside of \evaldef. Each parameter and the whole \evaldef is processed inside a TEX group, so your locally set data are inaccessible when your macro is used inside another "function-like" parameter or inside \evaldef. The \XOUT output is set locally by \apDIV macro, so it serves as a good example of this feature:

```
{\apDIV{1}{3}             ... \XOUT is .00000000000000000001 }
                          ... \XOUT is undefined
\evaldef{1/3}             ... \XOUT is undefined
\apPLUS{1}{\apDIV{1}{3}} ... \XOUT is undefined
```

The macro \apPOW{⟨*base*⟩}{⟨*exponent*⟩} calculates the power to the integer exponent. A slight optimization is implemented here so the usage of \apPOW is faster than repeated multiplication. The decimal non-integer exponents are not allowed. Use \EXP and \LN macros if you need to calculate non-integer exponent:

```
\def\POWER#1#2{\relax \EXP{(#2)*\LN{#1}}}
```

Note that both parameters are excepted as an ⟨*expression*⟩. Thus the `#2` is surrounded in the rounded brackets.

Examples of another common "function-like" macros:

```
\evaldef\degcoef{PI/180}
\def\SINdeg#1{\relax \SIN{\degcoef*(#1)}}
\def\COSdeg#1{\relax \COS{\degcoef*(#1)}}
\def\SINH#1{\relax \evaldef\myE{\EXP{#1}}\evaldef\OUT{(\myE-1/\myE)/2}}
\def\ASINH#1{\relax \LN{#1+\SQRT{(#1)^2+1}}}
\def\LOG#1{\relax \apLNtenexec \apDIV{\LN{#1}}{\apLNten}}
```

In another example, we implement the field `\F{`⟨*index*⟩`}` as an "function-like" macro. User can set values by `\set\F{`⟨*index*⟩`}={`⟨*value*⟩`}` and then these values can be used in an ⟨*expression*⟩.

```
\def\set#1#2#3#4{\evaldef\index{#2}\evaldef\value{#4}%
   \expandafter\edef\csname \string#1[\index]\endcsname{\value}}
\def\F#1{\relax % function-like macro
   \evaldef\index{#1}%
   \expandafter\ifx\csname \string\F[\index]\endcsname\relax
      \def\OUT{0}% undefined value
   \else
      \edef\OUT{\csname \string\F[\index]\endcsname}%
   \fi
}
\set \F{12/2} = {28+13}
\set \F{2*4} = {144^2}
\evaldef\test { 1 + \F{6} } \message{result=\test}
```

As an exercise, you can implement linear interpolation of known values.

The final example shows, how to implement the macro `\usedimen{`⟨*dimen*⟩`}{`⟨*unit*⟩`}`. It is "function-like" macro, it can be used in the ⟨*expression*⟩ and it returns the ⟨*decimal number*⟩ with the property ⟨*dimen*⟩=⟨*decimal number*⟩⟨*unit*⟩.

```
\def\usedimen #1#2{\relax  % function-like macro
   \def\OUT{0}%            % default value, if the unit isn't known
   \csname dimenX#2\endcsname{#1}}
\def\dimenXpt #1{\apDIV{\number#1}{65536}}
\def\dimenXcm #1{\apDIV{\number#1}{1864682.7}}
\def\dimenXmm #1{\apDIV{\number#1}{186468.27}}
%... etc.
\evaldef\a{\usedimen{\hsize}{cm}} % \a includes 15.91997501773358008845
```

Note that user cannot write `\usedimen\hsize{cm}` without braces because this isn't the syntactically correct operand (see section 1.1) and the ⟨*expression*⟩ scanner is unable to read it.

1.4 Printing expressions

TEX was designed for printing. The `apnum.tex` provides common syntax of ⟨*expressions*⟩ (given in section 1.1) which can be used for both: for *evaluating* or for *printing*. Printing can be done using `\eprint{`⟨*expression*⟩`}{`⟨*declaration*⟩`}` macro. The ⟨*declaration*⟩ part declares locally what to do with "variables" or with your "function-like" macros. You can insert your local `\def`'s or `\let`'s here because the ⟨*declaration*⟩ is executed in the group before the ⟨*expression*⟩ is printed. The `\eprint` macro must be used in math mode only. Example:

```
\def\printresult#1{$$\displaylines{
  \eprint{#1}\vars = \cr = \eprint{#1}\nums = \cr
   = \apFRAC=8 \evaldef\OUT{#1}\OUT, \cr
  \nums x = \X, \quad y = \Y.
```

```
}$$}

\def\X{-.25} \def\Y{18.11}
\def\vars{\def\X{x}\def\Y{y}\let\apMULop=\relax}
\def\nums{\corrnum\X \corrnum\Y}
\printresult
{-(\X-\SQRT{\Y^2+1}) + -((\Y*\X+1)/2) + \SIN{\X+\PIhalf} + 2*\COS{\Y}}
```

generates the result:

$$-\Big(x-\sqrt{y^2+1}\Big)+\Big(-\frac{yx+1}{2}\Big)+\sin\Big(x+\frac{\pi}{2}\Big)+2\cos y=$$
$$=-\Big(-0.25-\sqrt{18.11^2+1}\Big)+\Big(-\frac{18.11\cdot(-0.25)+1}{2}\Big)+\sin\Big(-0.25+\frac{\pi}{2}\Big)+2\cdot\cos 18.11=$$
$$=22.5977863,$$
$$x=-0.25,\quad y=18.11$$

This example prints the given ⟨*expression*⟩ in two forms: with "variables as variables" first and with "variables as constants" second. The ⟨*declaration*⟩ is prepared in the `\vars` macro for the first form and in the `\nums` macro for the second.

Note that `\eprint` macro re-calculates the occurrences of round brackets but keeps the meaning of the ⟨*expression*⟩. For example `(a+b)/c` is printed as `{a+b\over c}` (without brackets) and `6*-(a+b)` is printed as `6\cdot(-(a+b))` (new brackets pair is added). Or `\SIN{x}` is printed as `\sin x` (without brackets) but `\SIN{x+1}` is printed as `\sin(x+1)` (with brackets). And `\SIN{x}^2` is printed as `\sin^2 x`.

You can do `\let\apMULop=\,` or `\let\apMULop=\relax` in the ⟨*declaration*⟩ part if you need not to print any operator for multiplying. The default setting is `\let\apMULop=\cdot`. Another possibility is to set `\let\apMULop=\times`.

The macro `\corrnum`⟨*token*⟩ corrects the number saved in the ⟨*token*⟩ macro if it is in the form `[`⟨*minus*⟩`].`⟨*digits*⟩ (i. e. without digits before decimal point). Then zero is added before decimal point. Else nothing is changed.

Warning. The first parameter of `\eprint` (i. e. the ⟨*expression*⟩), must be directly expression without any expansion steps. For example, you cannot define `\def\foo{`⟨*expression*⟩`}` and do `\eprint{\foo}{}` but you can do `\expandafter\eprint\expandafter{\foo}{}`.

The macro `\eprint` has its own intelligence about putting brackets. If you need to put or remove brackets somewhere where the intelligence of `\eprint` is different from your opinion, you can create your function-like macros `\BK{`⟨*expression*⟩`}` and `\noBK{`⟨*expression*⟩`}`. They evaluate the ⟨*expression*⟩ when using `\evaldef`. The `\BK` prints the ⟨*expression*⟩ with brackets and `\noBK` prints it without brackets when using `\eprint`.

```
\def\BK#1{\relax \evaldef\OUT{#1}}  \let\noBK=\BK
\def\apEPj{\def\BK##1{\left(\eprint{##1}{}\right)}%
           \def\noBK##1{\eprint{##1}{}}}
Now $\eprint{3+\BK{\SIN{1}}^2}{}$ prints $3+(\sin 1)^2$.
```

Note that `\apEPj` macro is an initial hook of `\eprint` (it is run inside group before processing of the second parameter of `\eprint`).

1.5 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

2.1 Name Convention, Version, Counters

The internal control sequence names typically used in `apnum.tex` have the form `\apNAMEsuffix`, but there are exceptions. The control sequences mentioned in the section 1.1 (user's documentation) have typically more natural names. And the internal counter registers have names `\apnumA`, `\apnumB`, `\apnumC` etc.

The code starts by the greeting. The `\apVERSION` includes the version of this software.

apnum.tex

```
\def\apVERSION{1.7 <Apr 2018>}
\message{The Arbitrary Precision Numbers, \apVERSION}
```

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\apnumP \newcount\apnumL
\newcount\apnumX \newcount\apnumY \newcount\apnumZ
\newcount\apSIGNa \newcount\apSIGNb  \newcount\apEa \newcount\apEb
\newif\ifapX
```

The counters `\apSIGN` , `\apE` , `\apTOT`, `\apFRAC` and `\apEX` are declared here:

apnum.tex

```
\newcount\apSIGN
\newcount\apE
\newcount\apTOT   \apTOT=0
\newcount\apFRAC  \apFRAC=20
\newcount\apEX    \apEX=10
```

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
```

2.2 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) `\apPLUS{\apPLUS{\A}{\apMUL{\B}{\apPLUS{\C}{\D}}}}{\E}` and to execute the `\tmpb` macro.

The expression scanner adds the `\limits` 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\limits`. The `\limits` 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 \limits 0  0<=1   exec:
{\A+{\B*{\C+\D}}}  +  1       {{\A+{\B*{\C+\D}}}+\E} \end 0   0<=0 exec:
             bottom    0                            bottom  0         RESULT
```

\apVERSION: 9 \apSIGN: 4, 6, 9–10, 12–16, 18–19, 23–24, 29, 33, 37–42, 45–47, 51
\apE: 4, 5–6, 9–14, 16, 18–19, 23–24, 29, 33, 37, 39–42, 50 \apTOT: 3, 4, 6, 9, 24, 37, 43–47
\apFRAC: 3, 4, 6, 9, 24, 37, 39–47 \apEX: 5, 9, 41

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 \evaldef and \evalmdef macros are protected by \relax. It means that it can be used inside an ⟨*expression*⟩ as a "function-like" macro, but I don't imagine any usual application of this. The \apEVALa is executed.

apnum.tex

```
29: \def\evaldef{\relax \apEVALa{\apEadd\OUT}}
30: \def\evalmdef{\relax \apEVALa{}}
```

The macro \apEVALa {⟨*final-step*⟩}⟨*sequence*⟩{⟨*expression*⟩} runs the evaluation of the expression in the group. The base priority is initialized by \apnumA=0, then \apEVALb⟨*expression*⟩\limits scans the expression and saves the result in the form \apPLUS{\A}{\apMUL{\B}{\C}} (etc.) into the \tmpb macro. This macro is executed. The group is finished by \apEND macro, which keeps the \OUT, \apSIGN and \apE values unchanged. Then ⟨*final-step*⟩ is executed and finally, the defined ⟨*sequence*⟩ is set equivalent to the \OUT macro.

apnum.tex

```
31: \def\apEVALa#1#2#3{\begingroup \apnumA=0 \apnumE=1 \apEVALb#3\limits \tmpb \apEND #1\let#2=\OUT}
```

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

```
32: \def\apEVALb{\def\tmpa{}\apEVALc}
33: \def\apEVALc#1{%
34:    \ifx+#1\apEVALd \apEVALc \fi
35:    \ifx-#1\edef\tmpa{\tmpa-}\apEVALd\apEVALc \fi
36:    \ifx(#1\apEVALd \apEVALe \fi
37:    \ifx\the#1\apEVALd \apEVALf\the\fi
38:    \ifx\number#1\apEVALd \apEVALf\number\fi
39:    \apTESTdigit#1\iftrue
40:       \ifx E#1\let\tmpb=\tmpa \expandafter\apEVALd\expandafter\apEVALk
41:       \else \edef\tmpb{\tmpa#1}\expandafter\apEVALd\expandafter\apEVALn\fi\fi
42:    \edef\tmpb{\tmpa\noexpand#1}\expandafter
43:    \futurelet\expandafter\apNext\expandafter\apEVALg\romannumeral-`\.%
44: }
45: \def\apEVALd#1\fi#2-`\.{\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

```
46: \def\apEVALe{%
47:    \ifx\tmpa\empty \else \ifnum\tmpa1<0 \def\tmpb{-1}\apEVALp \apMUL 4\fi\fi
48:    \advance\apnumA by4
49:    \apEVALb
50: }
```

If the next token is \the or \number primitives (see lines 37 and 38), 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

```
51: \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 39) then we save the sign plus this token to the \tmpb at line 43 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

\evaldef: 3, 4, 6, 8–10, 12, 28, 36–37, 39, 41, 48 \evalmdef: 4, 5–6, 10, 37–40, 42, 45–47
\apEVALa: 10, 12 \OUT: 4, 5–6, 10, 12–14, 16–24, 26–27, 29–30, 33, 36–47 \apEVALb: 10–11, 48
\apEVALc: 10–11 \apEVALd: 10 \apEVALe: 10 \apEVALf: 10 \apEVALg: 10–11

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

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

If the next token after the sign is a digit or a dot (tested in `\apEVALc` by `\apTESTdigit` at line 39), 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

```
\def\apEVALk{\afterassignment\apEVALm\apE=}
\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 \apPLUS 1\fi
   \ifx-#1\apEVALp \apMINUS 1\fi
   \ifx*#1\apEVALp \apMUL 2\fi
   \ifx/#1\apEVALp \apDIV 2\fi
   \ifx^#1\apEVALp \apPOWx 3\fi
   \ifx)#1\advance\apnumA by-4 \let\apNext=\apEVALo \let\tmpa=\relax
      \ifnum\apnumA<0 \apEVALerror{many brackets ")"}\fi
   \fi
   \ifx\limits#1%
      \ifnum\apnumA>0 \apEVALerror{missing bracket ")"}\let\tmpa=\relax
      \else \apEVALp\END 0\let\apNext=\relax \fi
   \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
}
```

\apEVALh: 11 \apEVALk: 10–11 \apEVALm: 11 \apEVALn: 10–11 \apEVALo: 10–11
\apEVALp: 10–11

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 \apPLUS{⟨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\apPOWx \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\OUT{0}\apE=0\apSIGN=0\def\apNext##1\apEND{\apEND}%
}
```

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 \apPLUS, \apMINUS, \apMUL, \apDIV and \apPOW 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 `\expandafter`s (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 \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
   \begingroup\apE=0 #3% execution of the parameter in the group
   \edef\tmpb{\apE=\the\apE\relax\noexpand\apPPf\OUT@}\expandafter\endgroup\tmpb
}
\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 0`. 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{@}\apSIGN=0 \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{\ifnum\apE=0 \expandafter\apPPk \else \expandafter\apPPj \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
```

The \apPPab ⟨macro⟩{⟨paramA⟩}{⟨paramB⟩} is used for parameters of all macros \apPLUS, \apMUL 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 \apROLL, \apROUND and \apNORM 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 \apROLL, \apROUND or \apNORM 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
}
```

2.4 Addition and Subtraction

The significant part of the optimization in \apPLUS, \apMUL, \apDIV and \apPOW 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).

\apPPk: 14 \apPPl: 14 \apPPm: 14 \apPPn: 12, 14, 48, 50 \apPPab: 14–15, 18–19, 24, 28–29, 34
\apPPs: 14, 18, 31–33 \apPPt: 14 \apPPu: 14

```
  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 `\apPLUS` and `\apMINUS` macros prepare parameters using `\apPPab` and execute `\apPLUSa`:

apnum.tex

```
167: \def\apPLUS{\relax \apPPab\apPLUSa}
168: \def\apMINUS#1#2{\relax \apPPab\apPLUSa{#1}{-#2}}
```

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 170).
- 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 171 and 172).
- 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 173 and 174).
- The `\apnumC` includes the difference of Digits before the decimal point (multiplied by four) of given operands (line 175).
- 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 176 and 177).
- 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 178 to 180).
- 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 181 to 182).
- 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 184).

`\apPLUS`: 6, 9–12, 14–15, 37–39, 41–44, 46–51 `\apMINUS`: 6, 11–12, 15, 46, 48, 51 `\apPLUSa`: 15–16
`\apPLUSxA`: 15–17 `\apPLUSxB`: 15–17

- 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 185)
- The result of the first pass is expanded in the input stream and the \apNext (i. e. transmissions calculation) is activated at line 186.
- 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 187 to 193)

apnum.tex

```
169: \def\apPLUSa{%
170:    \ifnum\apEa=\apEb \apE=\apEa \else \apPLUSxE \fi
171:    \apDIG\tmpa\relax \apnumA=\apnumD % digits before decimal point
172:    \apDIG\tmpb\relax \apnumB=\apnumD
173:    \apIVmod \apnumA \apnumE \advance\apnumA by-\apnumE % digits in the first Digit
174:    \apIVmod \apnumB \apnumF \advance\apnumB by-\apnumF
175:    \apnumC=\apnumB \advance\apnumC by-\apnumA % difference between Digits
176:    \ifnum\apSIGNa<0 \def\apPLUSxA{-}\else \def\apPLUSxA{}\fi
177:    \ifnum\apSIGNb<0 \def\apPLUSxB{-}\else \def\apPLUSxB{}\fi
178:    \apSIGN=0 % \apSIGN=0 means that we are doing subtraction
179:    \ifx\apPLUSxA\empty \ifx\apPLUSxB\empty \apSIGN=1 \fi\fi
180:    \if\apPLUSxA-\relax \if\apPLUSxB-\relax \apSIGN=-1 \def\apPLUSxA{}\def\apPLUSxB{}\fi\fi
181:    \ifnum\apnumC>0 \apPLUSg  \apPLUSb \tmpb\apnumF \tmpa\apnumE \apnumB % first pass
182:    \else \apnumC=-\apnumC    \apPLUSb \tmpa\apnumE \tmpb\apnumF \apnumA
183:    \fi
184:    \ifnum\apnumG=0 \def\OUT{0}\apSIGN=0 \apE=0 \else
185:       \ifnum\apSIGN=0 \apSIGN=\apnumG \let\apNext=\apPLUSm \else \let\apNext=\apPLUSp \fi
186:       \apnumX=0 \edef\OUT{\expandafter}\expandafter \apNext \OUT@% second pass
187:       \ifnum\apnumD<1 % result in the form .000123
188:          \apnumZ=-\apnumD
189:          \def\tmpa{.}%
190:          \ifnum\apnumZ>0 \apADDzeros\tmpa \fi % adding dot and left zeros
191:          \edef\OUT{\ifnum\apSIGN<0-\fi\tmpa\OUT}%
192:       \else
193:          \edef\OUT{\expandafter}\expandafter\apPLUSy \OUT@% removing left zeros
194:    \fi\fi
195: }
```

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 $4-\langle$*second dig*$\rangle$ 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

```
196: \def\apPLUSb#1#2#3#4#5{%
197:    \edef\tmpd{\ifcase#4\or{}{}{}\or{}{}\or{}\fi#3}%
198:    \edef\tmpc{\ifcase#2\or{}{}{}\or{}{}\or{}\fi}%
199:    \let\apNext=\apPLUSc \apnumD=#5\advance\apnumD by4 \apnumG=0 \apnumZ=0 \def\OUT{}%
200:    \expandafter\expandafter\expandafter\apPLUSc\expandafter\tmpc#1\apNL\apNL\apNL\apNL@%
201: }
```

\apPLUSb: 15–16

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 214) 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

```
202: \def\apPLUSc#1#2#3#4{\apnumY=\apPLUSxA#1#2#3#4\relax
203:    \ifx\apNL#4\let\apNext=\apPLUSd\fi
204:    \ifx\apNL#1\relax \ifx\tmpd\empty \expandafter\expandafter\expandafter\apPLUSf \fi\fi
205:    \apPLUSe
206: }
207: \def\apPLUSd{\apnumY=0 \ifx\tmpd\empty \expandafter\apPLUSf \else\expandafter \apPLUSe\fi}
208: \def\apPLUSe{%
209:    \ifnum\apnumC>0 \advance\apnumC by-4
210:    \else \apIVread\tmpd \advance\apnumY by\apPLUSxB\apnumX \fi
211:    \ifnum\apnumZ=0 \apPLUSh \fi
212:    \edef\OUT{{\the\apnumY}\OUT}%
213:    \advance\apnumD by-4
214:    \apNext
215: }
216: \def\apPLUSf#1@{}
217: \def\apPLUSg{\let\tmpc=\apPLUSxA \let\apPLUSxA=\apPLUSxB \let\apPLUSxB=\tmpc}
218: \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

```
221: \def\apPLUSm#1{%
222:    \ifx@#1\else
223:       \apnumA=#1 \advance\apnumA by-\apnumX
224:       \ifnum\apnumA<0 \advance\apnumA by\apIVbase \apnumX=1 \else \apnumX=0 \fi
225:       \apPLUSw
226:       \expandafter\apPLUSm
227:    \fi
228: }
```

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.

\apPLUSc: 16–17 \apPLUSe: 17 \apPLUSh: 17 \apPLUSg: 15–17 \apPLUSd: 17 \apPLUSf: 17
\apPLUSm: 16–17 \apPLUSp: 16, 18

apnum.tex

```
\def\apPLUSp#1{%
   \ifx@#1\ifnum\apnumX>0 \apnumA=1 \apPLUSw \fi % .5+.5=.1 bug fixed
   \else
      \apnumA=\apnumX \advance\apnumA by#1
      \ifnum\apnumA<\apIVbase \apnumX=0 \else \apnumX=1 \advance\apnumA by-\apIVbase \fi
      \apPLUSw
      \expandafter\apPLUSp
   \fi
}
```

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 \apMUL prepares parameters using \apPPab and executes \apMULa

apnum.tex

```
\def\apMUL{\relax \apPPab\apMULa}
```

\apPLUSw: 17–18 \apPLUSy: 16, 18 \apPLUSz: 18 \apPLUSxE: 15–16, 18 \apMUL: 6, 9–12, 14, 18, 26, 36, 38, 41–42, 44, 46–49, 51

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 256).
- It calculates \apSIGN of the result (line 257).
- If \apSIGN=0 then the result is zero and we will do nothing more (line 258).
- 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 259 to 261).
- Let A is the number of total digits in the ⟨*param*⟩ 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⟨A⟩⟨F⟩ macro. All another Digits will have four digits. The \apMULb⟨*param*⟩@@@@ is able to read four digits, next four digits etc. We need to insert appropriate number of empty parameters before the ⟨*param*⟩. For example \apMULb{}{}{}⟨*param*⟩@@@@ reads first only one digit from ⟨*param*⟩, next four digits etc. The appropriate number of empty parameters are prepared in the \tmpc macro (lines 262 to 263).
- The \apMULb reads the ⟨*paramA*⟩ (all Digits) and prepares the \OUT macro in the special interleaved format (described below). The format is finished by *. in the line 265.
- Analogical work is done with the second parameter ⟨*paramB*⟩. But this parameter is processed by \apMULc, which reads Digits of the parameter and inserts them to the \tmpa in the reversed order (lines 266 to 268).
- The main calculation is done by \apMULd⟨*paramB*⟩@, which reads Digits from ⟨*paramB*⟩ (in reversed order) and does multiplication of the ⟨*paramA*⟩ (saved in the \OUT) by these Digits (line 269).
- The \apMULg macro converts the result \OUT to the human readable form (line 270).
- 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 271 to 273).

apnum.tex

```
255: \def\apMULa{%
256:    \apE=\apEa \advance\apE by\apEb
257:    \apSIGN=\apSIGNa \multiply\apSIGN by\apSIGNb
258:    \ifnum\apSIGN=0 \def\OUT{0}\apE=0 \else
259:       \apDIG\tmpa\apnumA \apnumX=\apnumA \advance\apnumA by\apnumD
260:       \apDIG\tmpb\apnumB \advance\apnumX by\apnumB \advance\apnumB by\apnumD
261:       \apnumD=\apnumX % \apnumD = the number of digits after decimal point in the result
262:       \apIVmod \apnumA \apnumF % \apnumF = digits in the first Digit of \tmpa
263:       \edef\tmpc{\ifcase\apnumF\or{}{}{}\or{}{}\or{}\fi}\def\OUT{}%
264:       \expandafter\expandafter\expandafter \apMULb \expandafter \tmpc \tmpa @@@@%
265:       \edef\OUT{*.\OUT}%
266:       \apIVmod \apnumB \apnumF % \apnumF = digits in the first Digit of \tmpb
267:       \edef\tmpc{\ifcase\apnumF\or{}{}{}\or{}{}\or{}\fi}\def\tmpa{}%
268:       \expandafter\expandafter\expandafter \apMULc \expandafter \tmpc \tmpb @@@@%
269:       \expandafter\apMULd \tmpa@%
270:       \expandafter\apMULg \OUT
271:       \edef\tmpa{\ifnum\apSIGN<0-\fi}%
272:       \ifnum\apnumD>0 \apnumZ=\apnumD \edef\tmpa{\tmpa.}\apADDzeros\tmpa \fi
273:       \ifx\tmpa\empty \else \edef\OUT{\tmpa\OUT}\fi
274:    \fi
275: }
```

We need to read the two data streams when the multiplication of the ⟨*paramA*⟩ by one Digit from ⟨*paramB*⟩ is performed and the partial sum is actualized. First: the digits of the ⟨*paramA*⟩ 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

\apMULa: 18–19, 29

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 $\langle paramA \rangle$ 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 $\langle paramA \rangle$@@@@ 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 $\langle paramA \rangle$ 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.{$\langle sum \rangle$}* during the third line processing and the dot here protects from removing the braces around the first real sum.

apnum.tex

```
276: \def\apMULb#1#2#3#4{\ifx@#4\else
277:    \ifx\OUT\empty \edef\OUT{{#1#2#3#4}*}\else\edef\OUT{{#1#2#3#4}0\OUT}\fi
278:    \expandafter\apMULb\fi
279: }
```

The macro \apMULc $\langle paramB \rangle$@@@@ reads Digits from $\langle paramB \rangle$ and saves them in reversed order into \tmpa. Each Digit is enclosed by TEX braces {}.

apnum.tex

```
280: \def\apMULc#1#2#3#4{\ifx@#4\else \edef\tmpa{{#1#2#3#4}\tmpa}\expandafter\apMULc\fi}
```

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

apnum.tex

```
281: \def\apMULd#1{\ifx@#1\else
282:      \apnumA=#1 \expandafter\apMULe \OUT
283:      \expandafter\apMULd
284:   \fi
285: }
```

The macro \apMULe $\langle special\ data\ format \rangle$ 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 297 to 301). 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 290 to 292).

\apMULb: 19–20, 29 \apMULc: 19–20 \apMULd: 19–20, 29 \apMULe: 20–21, 30
\apMULf: 20–21, 30

apnum.tex

```
\def\apMULe#1*#2{\apnumX=0 \def\OUT{#1{#2}*}\def\apOUTl{}\apnumO=1 \apnumL=0 \apMULf}
\def\apMULf#1#2{%
  \advance\apnumO by-1 \ifnum\apnumO=0 \apOUTx \fi
  \apnumB=#1 \multiply\apnumB by\apnumA \advance\apnumB by\apnumX
  \ifx*#2%
     \ifnum\apnumB<\apIVbase
       \edef\OUT{\OUT\expandafter\apOUTs\apOUTl.,\ifnum\the\apnumB#1=0 \else{\the\apnumB}{#1}\fi*}%
     \else \apIVtrans
        \expandafter \edef\csname apOUT:\apOUTn\endcsname
                 {\csname apOUT:\apOUTn\endcsname{\the\apnumB}{#1}}%
        \apMULf0*\fi
  \else \advance\apnumB by#2
     \ifnum\apnumB<\apIVbase \apnumX=0 \else \apIVtrans \fi
     \expandafter
        \edef\csname apOUT:\apOUTn\endcsname{\csname apOUT:\apOUTn\endcsname{\the\apnumB}{#1}}%
     \expandafter\apMULf \fi
}
```

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 2.9 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 $\geq$ 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 296.
- 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 292.

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

```
\def\apMULg#1{\def\OUT{}\apMULh}
\def\apMULh#1{\ifx*#1\expandafter\apMULi
   \else \apnumA=#1 \apMULo4{\apIVwrite\apnumA}%
         \expandafter\apMULh
   \fi
}
\def\apMULi#1#2#3{\apnumA=#1
   \ifx*#3\apMULo{\apNUMdigits\tmpa}{\the\apnumA}\expandafter\apMULj
   \else  \apMULo4{\apIVwrite\apnumA}\expandafter\apMULi
   \fi{#3}%
}
\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

```
\def\apMULo#1#2{\edef\tmpa{#2}%
   \advance\apnumD by-#1
   \ifnum\apnumD<1 \ifnum\apnumD>-4 \apMULt\fi\fi
```

\apMULg: 19, 21 \apMULh: 21 \apMULi: 21 \apMULj: 21 \apMULo: 21 \apMULt: 21–22

```
318:    \edef\OUT{\tmpa\OUT}%
319: }
320: \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 789012345678901234567890123 4 : 1231231231231231231
 xxxxxxxxxxxxxxxxxx 7 <-  calculated remainder
  xxxxxxxxxxxxxxxxxx x8 <- new calculated remainder
   xxxxxxxxxxxxxxxxx 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 macro \apDIV runs \apDIVa macro which 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 326).
- The \apSIGN of the result is calculated (line 327).
- If the dividend \tmpa is equal to zero, then \OUT and \XOUT are zeros and do nothing more (line 328).
- Calculate the exponent of ten \apE when scientific notation is used (Line 328).
- 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 330 to 332).
- 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 333).
- 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 334).
- The number of desired digits in the result \apnumC is calculated (lines 335 to 341).
- If the number of desired digits is zero or less than zero then do nothing more (line 341).
- Finish the calculation of the position of decimal point in the remainder \apnumH (line 334).
- Calculate the number of digits in the first Digit \apnumF (line 345).
- 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 346).
- We need to read only \apnumF (or \apnumF − 1) digits from the \tmpa. This is done by the \apIVreadX macro at line 348. The second Digit of the "partial dividend" includes four digits and it is read by \apIVread macro at line 350.
- 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 351 to 352).
- The \XOUT macro for the currently computed remainder is initialized. The special interleaved data format of the remainder \XOUT is described below (line 353).
- 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 354 to 355).

\apDIV: 6, 11–12, 14, 24, 36–51 \apDIVa: 24, 29

- 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 355).
- The main calculation loop is processed by the `\apDIVg` macro (line 357).
- 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 358).
- 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 360 to 361).

apnum.tex

```
324: \def\apDIV{\relax \apPPab\apDIVa}
325: \def\apDIVa{%
326:    \ifnum\apSIGNb=0 \apERR{Dividing by zero}\else
327:       \apSIGN=\apSIGNa \multiply\apSIGN by\apSIGNb
328:       \ifnum\apSIGNa=0 \def\OUT{0}\def\XOUT{0}\apE=0 \apSIGN=0 \else
329:          \apE=\apEa \advance\apE by-\apEb
330:          \apDIG\tmpb\relax \apnumB=\apnumD
331:          \apDIG\tmpa\relax \apnumH=\apnumD
332:          \advance\apnumD by-\apnumB % \apnumD = num. of digits before decimal point in the result
333:          \apDIVcomp\tmpa\tmpb  % apXtrue <=>  A>=B, i.e 1 digit from A/B
334:          \ifapX \advance\apnumD by1 \advance\apnumH by1 \fi
335:          \apnumC=\apTOT
336:          \ifnum\apTOT<0 \apnumC=-\apnumC
337:             \ifnum\apnumD>\apnumC \apnumC=\apnumD \fi
338:          \fi
339:          \ifnum\apTOT=0 \apnumC=\apFRAC \advance\apnumC by\apnumD
340:          \else \apnumX=\apFRAC \advance\apnumX by\apnumD
341:                \ifnum\apnumC>\apnumX \apnumC=\apnumX \fi
342:          \fi
343:          \ifnum\apnumC>0  % \apnumC = the number of digits in the result
344:             \advance\apnumH by-\apnumC % \apnumH = the position of decimal point in the remainder
345:             \apIVmod \apnumC \apnumF   % \apnumF = the number of digits in the first Digit
346:             \apIVread\tmpb \apnumB=\apnumX % \apnumB = partial divisor
347:             \apnumX=\apnumF \ifapX \advance\apnumX by-1 \fi
348:             \apIVreadX\apnumX\tmpa
349:             \apnumA=\apnumX % \apnumA = first Digit of the partial dividend
350:             \apIVread\tmpa  % \apnumX = second Digit of the partial dividend
351:             \edef\apDIVxA{\the\apnumA\apIVwrite\apnumX}% first partial dividend
352:             \edef\apDIVxB{\the\apnumB}% partial divisor
353:             \edef\XOUT{{\apDIVxB}{\the\apnumX}@{\the\apnumA}}% the \XOUT is initialized
354:             \edef\OUT{\ifnum\apSIGN<0-\fi}%
355:             \ifnum\apnumD<0 \edef\OUT{\OUT.}\apnumZ=-\apnumD \apADDzeros\OUT \fi
356:             \apnumE=1 \apnumZ=0
357:             \let\apNext=\apDIVg \apNext % <--- the main calculation loop is here
358:             \ifnum\apnumD>0 \apnumZ=\apnumD \apADDzeros\OUT \fi
359:             \ifnum\apnumE=0 \def\XOUT{0}\else % extracting remainder from \XOUT
360:                \edef\XOUT{\expandafter}\expandafter\apDIVv\XOUT
361:                \def\tmpc{\apnumH}\apnumG=\apSIGNa \expandafter\apROLLa\XOUT.@\XOUT
362:             \fi
363:          \else
364:             \def\OUT{0}\def\XOUT{0}\apE=0 \apSIGN=0
365:    \fi\fi\fi
366: }
```

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

`\apDIVcomp`: 23–25

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 `\apNL`s 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@{%
   \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 `\apNL`s (which simply expands zero digit) followed by stop-mark. The `\apDIVh` reads one Digit from input stream. Else we

`\apDIVcompA`: 25 `\apDIVcompB`: 25 `\apDIVg`: 24, 26

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
}
```

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 \apNexti 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 \apMUL macro. This increases the speed of the calculation. The data \apnum0, \apnumL and \apOUT1 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".

\apDIVh: 25–27 \apDIVi: 26 \apDIVj: 26 \apDIVp: 26–27 \apDIVxA: 23–24, 26–27
\apDIVxB: 23–24, 26–27

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:$\langle num\rangle$ 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.

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:$\langle num\rangle$ 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
```

\apDIVq: 26–27 \apDIVr: 27

```
452:       \expandafter\apNext\expandafter0\expandafter\apnumZ\XOUT}%
453:       \expandafter\apNext
454:    \fi
455: }
```

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

```
456: \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*⟩.

apnum.tex

```
457: \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

```
458: \def\apDIVv#1#2{\apnumX=#2
459:    \ifx@#1\apDIVw{.\apIVwrite\apnumX}\else\apDIVw{\apIVwrite\apnumX}\expandafter\apDIVv\fi
460: }
461: \def\apDIVw#1{%
462:    \ifx\XOUT\empty \ifnum\apnumX=0
463:                    \else \edef\tmpa{#1}\edef\XOUT{\apREMzerosR\tmpa\XOUT}%
464:                    \fi
465:    \else \edef\XOUT{#1\XOUT}\fi
466: }
```

2.7 Power to the Integer

The \apPOW macro does the power to the integer exponent only. The \apPOWx is equivalent to \apPOW and it is used in \evaldef macro for the ^ operator. If you want to redefine the meaning of the ^ operator then redefine the \apPOWx sequence.

apnum.tex

```
470: \def\apPOW{\relax \apPPab\apPOWa}  \let\apPOWx=\apPOW % for usage as ^ operator
```

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.

\apDIVt: 27–28 \apDIVu: 27–28 \XOUT: 6, 5, 14, 23–28, 32–33, 37–38, 46 \apDIVv: 24, 28
\apDIVw: 28 \apPOW: 6, 12, 14, 28, 36, 41–42, 46–47, 51 \apPOWx: 11–12, 28, 48
\apPOWa: 28–30

- 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 472 and 473).
- 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 475 to 478).
- The \apE is calculated from \apEa (line 479).
- The sign of the result is negative only if the \tmpb is odd and base is negative (line 481).
- 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 482).
- 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 484 to 485).
- 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 486 and 487).
- The initial value of $p = 1$ from our algorithm is set in interleaved format into \tmpc macro (line 488).
- The main loop described above is processed by \apPOWb macro. (line 489).
- The result in \tmpc is converted into human readable form by the \apPOWg macro and it is stored into the \OUT macro (line 490).
- 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 491 and 492).
- If the exponent is negative then do the $1/r$ calculation, where r is previous result (line 493).

apnum.tex

```
471: \def\apPOWa{%
472:    \ifnum\apSIGNa=0 \def\OUT{0}\apSIGN=0 \apE=0 \else
473:       \ifnum\apSIGNb=0 \def\OUT{1}\apSIGN=1 \apE=0 \else
474:          \apDIG\tmpb\apnumB
475:          \ifnum\apnumB>0 \apERR{POW: non-integer exponent is not implemented yet}\apPOWe\fi
476:          \ifnum\apEb=0 \else \apERR{POW: the E notation of exponent isn't allowed}\apPOWe\fi
477:          \ifnum\apnumD>8 \apERR{POW: too big exponent.
478:                        Do you really need about 10^\the\apnumD\space digits in output?}\apPOWe\fi
479:          \apE=\apEa \multiply\apE by\tmpb\relax
480:          \apSIGN=\apSIGNa
481:          \ifodd\tmpb \else \apSIGN=1 \fi
482:          \apDIG\tmpa\apnumA \apnumC=\apnumA \advance\apnumC by\apnumD
483:          \apnumD=\apnumA \multiply\apnumD by\tmpb
484:          \apIVmod \apnumC \apnumA
485:          \edef\tmpc{\ifcase\apnumA\or{}{}{}\or{}{}\or{}\fi}\def\OUT{}%
486:          \expandafter\expandafter\expandafter \apMULb \expandafter \tmpc \tmpa @@@@%
487:          \edef\OUT{*.\OUT}%    \OUT := \tmpa in interleaved format
488:          \def\tmpc{*.1*}%
489:          \apnumE=\tmpb\relax \apPOWb
490:          \expandafter\apPOWg \tmpc % \OUT := \tmpc in human raedable form
491:          \ifnum\apnumD=0 \ifnum \apSIGN<0 \edef\OUT{-\OUT}\fi
492:          \else \def\tmpc{-\apnumD}\apnumG=\apSIGN \expandafter\apROLLa\OUT.@\OUT\fi
493:          \ifnum\apSIGNb<0 \apPPab\apDIVa 1\OUT \fi
494:          \relax
495:    \fi\fi
496: }
```

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

```
497: \def\apPOWb{%
498:    \ifodd\apnumE   \def\tmpb{}\expandafter\apPOWd\OUT
499:                    \let\tmpd=\OUT \let\OUT=\tmpc
500:                    \expandafter\apMULd \tmpb@\expandafter\apPOWn\OUT@%
501:                    \let\tmpc=\OUT \let\OUT=\tmpd
```

\apPOWb: 29–30

```
502:    \fi
503:    \divide\apnumE by2
504:    \ifnum\apnumE>0 \expandafter\apPOWt\OUT \expandafter\apPOWn\OUT@%
505:                    \expandafter\apPOWb
506:    \fi
507: }
```

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

apnum.tex

```
508: \def\apPOWd#1#2{% \apPOWd <spec format> => \tmpb (in simple reverse format)
509:    \ifx*#1\expandafter\apPOWd \else
510:       \edef\tmpb{\tmpb{#1}}%
511:       \ifx*#2\else \expandafter\expandafter\expandafter\apPOWd\fi
512:    \fi
513: }
```

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

```
514: \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

```
515: \def\apPOWg#1#2{\def\OUT{}\apPOWh} % conversion to the human readable form
516: \def\apPOWh#1#2{\apnumA=#1
517:    \ifx*#2\edef\OUT{\the\apnumA\OUT}\else \edef\OUT{\apIVwrite\apnumA\OUT}\expandafter\apPOWh\fi
518: }
```

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

```
519: \def\apPOWn#1{\def\OUT{*}\apPOWna}
520: \def\apPOWna#1{\ifx*#1\expandafter\apPOWnn\else \edef\OUT{\OUT0{#1}}\expandafter\apPOWna\fi}
521: \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

```
522: \def\apPOWt#1#2{\apPOWu} % power to two
523: \def\apPOWu#1#2{\apnumA=#1
524:    \expandafter\apPOWv\OUT
525:    \ifx*#2\else \expandafter\apPOWu\fi
526: }
527: \def\apPOWv#1*#2#3#4{\def\apOUTl{}\apnumO=1 \apnumL=0
528:    \apnumB=\apnumA \multiply\apnumB by\apnumB \multiply\apnumA by2
529:    \ifx*#4\else\advance\apnumB by#4 \fi
530:    \ifx\apnumB<\apIVbase \apnumX=0 \else \apIVtrans \fi
531:    \edef\OUT{#1{#2}{\the\apnumB}*}%
532:    \ifx*#4\apMULf0*\else\expandafter\apMULf\fi
```

```
533: }
```

2.8 apROLL, apROUND and apNORM Macros

The macros \apROLL , \apROUND and \apNORM 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 \apROLL, \apROUND and \apNORM, 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 \apROLL, \apROUND and \apNORM 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

```
537: \def\apROLL{\apPPs\apROLLa}
538: \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

```
539: \def\apROLLc{\edef\tmpc{}\edef\tmpd{\ifnum\apnumG<0-\fi}\apnumB=0 \apROLLd}
540: \def\apROLLd#1{%
541:    \ifx.#1\expandafter\apROLLe
542:    \else \edef\tmpc{\tmpc#1}%
543:       \advance\apnumB by1
544:       \expandafter\apROLLd
545:    \fi
546: }
547: \def\apROLLe#1{\ifx@#1\edef\tmpc{\tmpc.@}\else\edef\tmpc{\tmpc#1}\fi
548:    \advance\apnumB by\apnumA
549:    \ifnum\apnumB<0
550:        \apnumZ=-\apnumB \edef\tmpd{\tmpd.}\apADDzeros\tmpd
551:        \expandafter\expandafter\expandafter\apROLLf\expandafter\tmpc
552:    \else
553:        \apnumA=\apnumB
554:        \expandafter\expandafter\expandafter\apROLLi\expandafter\tmpc
555:    \fi
556: }
557: \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

```
558: \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\apROUND{\apPPs\apROUNDa}
\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
```

\apROLLj: 31–32 \apROLLk: 31–32 \apROLLn: 32 \apROLLo: 32 \apROUNDa: 14, 31–33
\apROUNDb: 32 \apROUNDc: 32 \apROUNDd: 32–33

```
590:          \fi
591:    \fi \apNext
592: }
593: \def\apROUNDd#1.@#2{\def\XOUT{#1}\edef\XOUT{\apREMzerosR\XOUT}%
594:    \ifnum\apnumX=0 \def\tmpd{}\fi
595:    \ifx\tmpd\empty
596:       \ifx\tmpc\empty \def#2{0}%
597:       \else \edef#2{\ifnum\apnumG<0-\fi\tmpc}\fi
598:    \else\edef#2{\ifnum\apnumG<0-\fi\tmpc.\tmpd}\fi
599: }
```

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

```
600: \def\apROUNDe#1.@#2{\apnumC=\apnumD
601:    \apPPs\apROLLa#2{\apnumC}\apPPs\apROUNDa#2{0}\apPPs\apROLLa#2{-\apnumC}%
602: }
```

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

```
603: \def\apNORM{\apPPs\apNORMa}
604: \def\apNORMa#1.@#2{\ifnum\apnumG<0 \def#2{#1}\fi \expandafter\apNORMb\expandafter#2\tmpc@}
605: \def\apNORMb#1#2#3@{%
606:    \ifx.#2\apnumC=#3\relax  \apDIG#1\apnumA \apNORMc#1%
607:    \else \apnumC=#2#3\relax \apDIG#1\relax  \apNORMd#1%
608:    \fi
609: }
610: \def\apNORMc#1{\advance\apE by-\apnumA \advance\apE by\apnumC
611:    \def\tmpc{-\apnumC}\expandafter\apROLLa#1.@#1%
612: }
613: \def\apNORMd#1{\advance\apE by\apnumD \advance\apE by-\apnumC
614:    \def\tmpc{\apnumC}\expandafter\apROLLa\expandafter.#1.@#1%
615: }
```

The macro \apEadd ⟨*sequence*⟩ adds E in scientific format into ⟨*sequence*⟩ macro and \apEnum ⟨*sequence*⟩ normalizes the number in the ⟨*sequence*⟩. After processing these macros the \apE register is set to zero.

apnum.tex

```
616: \def\apEadd#1{\ifnum\apE=0 \else\edef#1{#1E\ifnum\apE>0+\fi\the\apE}\apE=0 \fi}
617: \def\apEnum#1{\ifnum\apE=0 \else\apROLL#1\apE \apE=0 \fi}
```

2.9 Miscelaneous Macros

The macro \apEND closes the \begingroup group, but keeps the values of \OUT macro and \apSIGN, \apE registers.

apnum.tex

```
621: \def\apEND{\global\let\apENDx=\OUT
622:    \edef\tmpb{\apSIGN=\the\apSIGN \apE=\the\apE}%
623:    \expandafter\endgroup \tmpb \let\OUT=\apENDx
624: }
```

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

\apROUNDe: 32–33 \apNORMa: 31, 33 \apNORMb: 33 \apNORMc: 33 \apNORMd: 33
\apEadd: 4, 5, 10, 33 \apEnum: 4, 5, 33, 37–38, 40, 45–47 \apEND: 10, 12, 33, 36–42, 45–47
\apDIG: 15–16, 19, 23–24, 29, 33–34, 39, 42, 47

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

\apDIGa: 34 \apDIGb: 34 \apDIGc: 34 \apDIGd: 34 \apDIGe: 34 \apDIGf: 34
\apIVread: 17, 23–26, 34 \apIVreadA: 34–35 \apNL: 16–17, 25–26, 34–35 \apIVreadX: 23–24, 35

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 \apREMfirst ⟨*sequence*⟩ macro removes the first token from the ⟨*sequence*⟩ macro. It can be used for removing the "minus" sign from the "number-like" macros.

apnum.tex
```
\def\apREMfirst#1{\expandafter\apREMfirsta#1@#1}
\def\apREMfirsta#1#2@#3{\def#3{#2}}
```

The writing to the \OUT in the \apMUL, \apDIV and \apPOW 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 \apnum0 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{\apnum0=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}
```

If a "function-like" macro needs a local counters then it is recommended to enclose all calculation into a group \apINIT ... \apEND. The \apINIT opens the group and prepares a short name \do and the macro \localcounts⟨*counters*⟩;. The typical usage is:

```
\def\MACRO#1{\relax \apINIT % function-like macro, \apINIT
   \evaldef\foo{#1}%        % preparing the parameter
   \localounts \N \M \K ;%  % local \newcount\N \newcount\M \newcount\K
   ...                      % calculation
   \apEND                   % end of \apINIT group
}
```

Note that \localcounts is used after preparing the parameter using \evaldef in odrer to avoid name conflict of local declared "variables" and "variables" used in #1 by user.

The \apINIT sets locally \localcounts to be equivalent to \apCOUNTS. This macro increases the top index of allocated counters \count10 (used in plain TEX) locally and declares the counters locally. It means that if the group is closed then the counters are deallocated and top index of counters \count10 is returned to its original value.

apnum.tex
```
\def\apINIT{\begingroup \let\do=\apEVALxdo \let\localcounts=\apCOUNTS}
\def\apCOUNTS#1{\ifx;#1\else
   \advance\count10 by1 \countdef#1=\count10
   \expandafter\apCOUNTS\fi
}
```

The macro \do ⟨*sequence*⟩=⟨*calculation*⟩; allows to write the calculation of Polish expressions more synoptic:

\apREMfirst: 6, 36–37, 40, 45–47 \apOUTx: 21, 27, 36 \apOUTn: 21, 27, 36 \apOUTl: 21, 26–27, 30, 36 \apOUTs: 21, 27, 36 \apINIT: 36–42, 44–47 \localcounts: 36, 38–40, 42, 44, 46–47 \apCOUNTS: 36 \do: 36–38, 41–43, 46–47

```
\do \X=\apPLUS{2}{\the\N};%    % is equivalent to:
\apPLUS{2}{\the\N}\let\X=\OUT
```

The \do macro is locally set to be equivalent to \apEVALxdo .

apnum.tex

```
\def\apEVALxdo#1=#2;{#2\let#1=\OUT}
```

The \apRETURN macro must be followed by \fi. It skips the rest of the block \apINIT...\apEND typically used in "function-like" macros. The \apERR {⟨text⟩} macro writes ⟨text⟩ as error message and returns the processing of the block enclosed by \apINIT...\apEND. User can redefine it if the \errmessage isn't required.

apnum.tex

```
\def\apRETURN#1\apEND{\fi\apEND}
\def\apERR#1{\errmessage{#1}}
```

The \apNOPT macro removes the pt letters after expansion of ⟨dimen⟩ register. This is usable when we do a classical ⟨dimen⟩ calculation, see TBN page 80. Usage: \expandafter\apNOPT\the⟨dimen⟩.

apnum.tex

```
{\lccode`\?=`\p \lccode`\!=`\t  \lowercase{\gdef\apNOPT#1?!{#1}}}
```

The \loop macro from plain TEX is redefined here in more convenient way. It does the same as original \loop by D. Knuth but moreover, it allows the construction \if...\else...\repeat.

apnum.tex

```
\def\loop#1\repeat{\def\body{#1\relax\expandafter\body\fi}\body}
```

2.10 Function-like Macros

The implementation of function-like macros \ABS , \SGN , \iDIV , \iMOD , \iFLOOR , \iFRAC are simple.

apnum.tex

```
\def\ABS#1{\relax        % mandatory \relax for "function-like" macros
   \evalmdef\OUT{#1}%    % evaluation of the input parameter
   \ifnum\apSIGN<0       % if (input < 0)
      \apSIGN=1          %   sign = 1
      \apREMfirst\OUT    %   remove first "minus" from OUT
   \fi                   % fi
}
\def\SGN#1{\relax \evaldef\OUT{#1}\edef\OUT{\the\apSIGN}\apE=0 }
\def\iDIV#1#2{\relax \apINIT               % calculation in group
   \evalmdef\apAparam{#1}\apEnum\apAparam
   \evalmdef\apBparam{#2}\apEnum\apAparam      % evaluation of the parameters
   \apTOT=0 \apFRAC=0 \apDIV\apAparam\apBparam % integer division
   \apEND                                  % end of group
}
\def\iMOD#1#2{\relax \apINIT               % calculation in group
   \evalmdef\apAparam{#1}\apEnum\apAparam
   \evalmdef\apBparam{#2}\apEnum\apBparam  % evaluation of the parameters
   \apTOT=0 \apFRAC=0 \apDIV\apAparam\apBparam % integer division
   \let\OUT=\XOUT                          % remainder is the output
   \apEND                                  % end of group
}
\def\iFLOOR#1{\relax \evalmdef\OUT{#1}\apEnum\OUT \apROUND\OUT0%
   \ifnum\apSIGN<0 \ifx\XOUT\empty \else \apPLUS\OUT{-1}\fi\fi
   \def\tmp{0}\ifx\tmp\OUT \apSIGN=0 \fi
}
\def\iFRAC#1{\relax
   \evalmdef\OUT{#1}\apEnum\OUT \apROUND\OUT0% % preparing the parameter
   \ifx\XOUT\empty \def\OUT{0}\apSIGN=0        % empty fraction part means zero
   \else \ifnum\apSIGN<0
            \edef\XOUT{-.\XOUT}\apPLUS1\XOUT   % OUT = 1 - .\XOUT
         \else \edef\OUT{.\XOUT}\apSIGN=1      % else OUT = .\XOUT
   \fi   \fi
```

\apEVALxdo: 36–37 \apRETURN: 37–40, 42, 46–47 \apERR: 24, 29, 37–39, 42, 46–47
\apNOPT: 37, 40, 43 \loop: 37–42, 44, 46–47 \ABS: 3, 5–6, 37, 48 \SGN: 3, 5, 37, 48
\iDIV: 3, 5, 37, 48 \iMOD: 3, 37, 48 \iFLOOR: 3, 37, 48 \iFRAC: 3, 37, 48

```
738: }
```

The \FAC macro for **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 in my computer). But we want to calculate more factorial than only 255!.

apnum.tex

```
740: \def\FAC#1{\relax \apINIT             % "function-like" in the group, FAC = factorial
741:    \evalmdef\OUT{#1}\apEnum\OUT       % preparing the parameter
742:    \localcounts \N;%                  % local \newcount
743:    \ifnum\apSIGN<0 \apERR{\string\FAC: argument {\OUT} cannot be negative}\apRETURN\fi
744:    \let\tmp=\OUT \apROUND\tmp0%       % test, if parameter is integer
745:    \ifx\XOUT\empty \else \apERR{\string\FAC: argument {\OUT} must be integer}\apRETURN\fi
746:    \N=\OUT\relax                      % N = param (error here if it is an big integer)
747:    \ifnum\N=0\def\OUT{1}\apSIGN=1 \fi % special definition for factorial(0)
748:    \loop \ifnum \N>2 \advance\N by-1  % loop if (N>2) N--
749:          \apMUL{\OUT}{\the\N}\repeat  %               OUT = OUT * N , repeat
750:    \apEND                             % end of group
751: }
```

The \BINOM {a}{b} is **binomial coefficient** defined by

$$\binom{a}{b} = \frac{a!}{b!\,(a-b)!} = \frac{a\,(a-1)\cdots(a-b+1)}{b!} \quad \text{for integer } b>0, \qquad \binom{a}{0} = 1.$$

We use the formula where $(a-b)!$ is missing in numerator and denominator (second fraction) because of time optimization. Second advantage of such formula is that a need not to be integer. That is the reason why the \BINOM isn't defined simply as

```
\def\BINOM#1#2{\relax \evaldef{ \FAC{#1} / (\FAC{#2} * \FAC{(#1)-(#2)} }}
```

The macro \BINOM checks if a is integer. If it is true then we choose \C as minimum of b and $a-b$. Then we calculate factorial of \C in the denominator of the formula (second fraction). And nominator includes \C factors. If a is non-negative integer and $a<b$ then the result is zero because one zero occurs between the factors in the nominator. Thus we give the result zero and we skip the rest of calculation. If a is non-integer, then \C must be b. The \step macro (it generates the factors in the nominator) is prepared in two versions: for a integer we use \advance\A by-1 which is much faster than \apPLUS\paramA{-1} used for a non-integer.

apnum.tex

```
752: \def\BINOM#1#2{\relax \apINIT          % BINOM = {#1 \choose #2} ...
753:    \evalmdef\apAparam{#1}\apEnum\apAparam
754:    \evalmdef\apBparam{#2}\apEnum\apBparam % preparation of the parameters
755:    \localcounts \A \B \C ;%           % local \newcounts
756:    \let\OUT=\apBparam \apROUND\OUT0%  % test if B is integer
757:    \ifx\XOUT\empty\else\apERR{\string\BINOM: second arg. {\apBparam} must be integer}\apRETURN\fi
758:    \let\OUT=\apAparam \apROUND\OUT0%    % test if A is integer
759:    \ifx\XOUT\empty                    % A is integer:
760:       \A=\apAparam \B=\apBparam       %    A = #1, B = #2
761:       \C=\A \advance\C by-\B          %    C = A - B
762:       \ifnum\C>\B \C=\B \fi           %    if (C > B) C = B fi
763:       \ifnum\A<0  \C=\B               %    if (A < 0) C = B fi
764:       \else \ifnum\A<\B \def\OUT{0}\apSIGN=0  % if (0 <= A < B)  OUT = 0  return
765:                \expandafter\expandafter\expandafter \apRETURN \fi\fi
766:       \def\step{\advance\A by-1 \apMUL\OUT{\the\A}}%
767:    \else \C=\apBparam                 % A is not integer
768:       \def\step{\let\apBparam\OUT \do\apAparam=\apPLUS\apAparam{-1};%
769:                \let\OUT=\apBparam \apMUL\OUT\apAparam}%
770:    \fi
771:    \ifnum\C=0 \def\OUT{1}\apSIGN=1 \apRETURN\fi
772:    \do\D=\FAC{\the\C};%               %    D = C!
773:    \let\OUT=\apAparam                 %    OUT = #1
774:    \loop \advance\C by-1              %    loop C--
775:          \ifnum\C>0 \step \repeat     %         if (C > 0) A--, OUT = OUT * A, repeat
776:    \apDIV{\OUT}{\D}%                  %    OUT = OUT / D
777:    \apEND
```

\FAC: 3, 38, 48, 51 \BINOM: 3, 38, 48

```
778: }
```

The square root is computed in the macro \SQRT {*a*} using Newton's approximation method. This method solves the equation $f(x) = 0$ (in this case $x^2 - a = 0$) by following way. Guess the initial value of the result x_0. Create tangent to the graph of f in the point $[x_0, f(x_0)]$ using the knowledge about $f'(x_0)$ value. The intersection of this line with the axis x is the new approximation of the result x_1. Do the same with x_1 and find x_2, etc. If you apply the general Newton method to the problem $x^2 - a = 0$ then you get the formula

$$\text{choose } x_0 \text{ as an initial guess,} \quad \text{iterate: } x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

If $|x_{n+1} - x_n|$ is sufficiently small we stop the processing. In practice, we stop the processing, if the \OUT representation of x_{n+1} rounded to the \apFRAC is the same as the previous representation of x_n, i. e. \ifx\Xn\OUT in TEX language. Amazingly, we need only about four iterations for 20-digits precision and about seven iterations for 50-digits precision, if the initial guess is good chosen.

The rest of the work in the \SQRT macro is about the right choose of the initial guess (using \apSQRTr macro) and about shifting the decimal point in order to set the a value into the interval $[1, 100)$. The decimal point is shifted by -\M value. After calculation is done, the decimal point is shifted by \M/2 value back. If user know good initial value then he/she can set it to \apSQRTxo macro. The calcualtion of initial value x_0 is skipped in such case.

apnum.tex

```
779: \def\SQRT#1{\relax \apINIT           % OUT = SQRT(#1) ...
780:    \evalmdef\A{#1}%                  % parameter preparation
781:    \localcounts \M \E ;%             % local counters
782:    \E=\apE \apE=0
783:    \ifnum\apSIGN=0 \apRETURN\fi      % SQRT(0) = 0 (OUT is set to 0 by previous \evaldef)
784:    \ifnum\apSIGN<0 \apERR{\string\SQRT: argument {\A} is out of range}\apRETURN\fi
785:    \ifodd\E \apROLL\A{-1}\advance\E by1 \fi % we need the E representation with even exponent
786:    \let\B=\A \let\C=\A
787:    \apDIG\C\relax \M=\apnumD         % M is the number of digits before decimal point
788:    \advance\M by-2 \ifodd\M \advance\M by1 \fi      % M = M - 2 , M must be even
789:    \ifx\apSQRTxo\undefined           % we need to calculate Xo
790:       \ifnum\M=0 \else \apROLL\B{-\M}\divide\M by2 \fi % shift decimal point by -M,  M = M / 2
791:       \apSQRTr\B \let\Xn=\OUT        % Xn = estimate of SQRT
792:       \ifnum\M<0 \let\A=\B \fi       % if (A < 1) calculate with B where decimal point is shifted
793:       \ifnum\M>0 \apROLL\Xn \M \fi   % if (A >= 100) shift the decial point of initial guess
794:    \else \let\Xn=\apSQRTxo \fi
795:    \loop                                      % loop    ... Newton's method
796:       \apDIV{\apPLUS{\Xn}{\apDIV{\A}{\Xn}}}{2}% %    OUT = (Xn + A/Xn) / 2
797:       \ifx\OUT\Xn \else                       %    if (OUT != Xn)
798:          \let\Xn=\OUT \repeat                 %       Xn = OUT, repeat
799:    \ifnum\M<0 \apROLL\OUT\M \fi      % shift the decimal point by M back
800:    \apE=\E \divide\apE by2           % correct the E exponent
801:    \apEND
802: }
```

Note that if the input $a < 1$, then we start the Newton's method with b. It is the value a with shifted decimal point, $b \in [1, 100)$. On the other hand, if $a \geq 1$ then we start the Newton's method directly with a, because the second derivative $(x^2)''$ is constant so the speed of Newton's method is independent on the value of x. And we need to calculate the \apFRAC digits after the decimal point.

The macro \apSQRTr ⟨*number*⟩ excepts ⟨*number*⟩ in the interval $[1, 100]$ and makes a roughly estimation of square root of the ⟨*number*⟩ in the \OUT macro. It uses only classical ⟨*dimen*⟩ calculation, it doesn't use any apnum.tex operations. The result is based on the linear approximation of the function $g(x) = \sqrt{x}$ with known exact points $[1, 1], [4, 2], [9, 3], \ldots, [100, 10]$. Note, that the differences between x_i values of exact points are $3, 5, 7, \ldots, 19$. The inverted values of these differences are pre-calculated and inserted after \apSQRTra macro call.

The \apSQRTra macro operates repeatedly for $i = 1, \ldots, 10$ until \dimen0 $= x < x_i$. Then the \apSQRTrb is executed. We are in the situation \dimen0 $= x \in [x_{i-1}, x_i)$, $g(x_i) = i$, $g(x_{i-1}) = i - 1$

\SQRT: 3, 4–5, 39, 44, 47–48 \apSQRTxo: 39, 44 \apSQRTr: 39–40 \apSQRTra: 39–40
\apSQRTrb: 40

and the calculation of \OUT $= g(x_{i-1}) + (x - x_{i-1})/(x_i - x_{i-1})$ is performed. If $x \in [1, 4)$ then the linear approximation is worse. So, we calculate additional linear correction in \dimen1 using the pre-calculated value $\sqrt{2} - 1.33333 \doteq 0.08088$ here.

apnum.tex

```
\def\apSQRTr#1{\dimen0=#1pt \apnumB=1 \apnumC=1 \apSQRTra}
\def\apSQRTra{\advance\apnumB by2 \advance\apnumC by\apnumB % B = difference, C = x_i
   \ifnum\apnumC>100 \def\OUT{10}\else
      \ifdim\dimen0<\apnumC pt \apSQRTrb \else
         \expandafter\expandafter\expandafter\apSQRTra\fi\fi
}
\def\apSQRTrb{% x = dimen0,  B = x_i - x_{i-1},  C = x_i = i
   \ifdim\dimen0<4pt
      \ifdim\dimen0>2pt \dimen1=4pt \advance\dimen1 by-\dimen0 \divide\dimen1 by2
      \else \dimen1=\dimen0 \advance\dimen1 by-1pt \fi
      \dimen1=.080884\dimen1     % dimen1 = additional linear correction
   \else \dimen1=0pt \fi
   \advance\apnumC by-\apnumB    % C = x_{i-1}
   \advance\dimen0 by-\apnumC pt % dimen0 = (x - x_{i-1})
   \divide\dimen0 by\apnumB      % dimen0 = (x - x_{i-1}) / difference
   \divide\apnumB by2            % B = i-1 = g(x_{i-1})
   \advance\dimen0 by\apnumB pt  % dimen0 = g(x_{i-1}) + (x - x_{i-1} / (x_i-x_{i-1})
   \advance\dimen0 by\dimen1     % dimen0 += additional linear correction
   \edef\OUT{\expandafter\apNOPT\the\dimen0}%  OUT = dimen0
}
```

The exponential function e^x is implemented in the \EXP macro using Taylor series at zero point:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

If $x \in (0, 1)$ then this series converges relatively quickly.

The macro \EXP takes its argument. If it is negative, remember this fact, remove minus sign and do \OUT=1/\OUT in final step. Now, the argument is positive always. If the argument is "big" (greater or equal than 4, tested by \testBig) then \apEXPb macro is used for evaluating. Else \apEXPa macro evaluates the exponential.

apnum.tex

```
\def\EXP#1{\relax\apINIT                          % OUT = EXP(#1) ...
   \evalmdef\OUT{#1}\apEnum\OUT                   % OUT = #1
   \localcounts \N \K ;%
   \ifnum\apSIGN=0 \def\OUT{1}\apSIGN=1 \apRETURN \fi
   \edef\digits{\the\apFRAC}\advance\apFRAC by4
   \edef\signX{\the\apSIGN}%
   \ifnum\apSIGN<0 \apSIGN=1 \apREMfirst\OUT \fi % remove "minus" sign
   \def\testBig ##1##2##3\relax##4{\ifx##1.\apXfalse \else
      \ifx##2.\ifnum##1<4 \apXfalse \else \apXtrue \fi \else \apXtrue
      \fi \fi \ifapX}%
   \expandafter\testBig \OUT.\relax
   \iftrue \apEXPb \else \apEXPa \fi   % OUT = e^OUT
   \ifnum\signX<0 \K=-\apE \apDIV 1\OUT \apE=\K \fi % if (signX < 0)  OUT = 1 / OUT
   \apSIGN=1                 % EXP is always positive
   \apEND
}
```

The \apEXPa macro supposes input argument (saved in \OUT macro) in the interval $[0, 4)$. If the argument is greater than 1, do argument = argument/2 and increase K register. Do this step in the loop until argument < 1. Then calculate e^x using Taylor series mentioned above. After \OUT is calculated then we do \OUT=\OUT^2 in the loop K times, because $e^{2x} = (e^x)^2$. Note that $K \leq 2$ in all cases.

The Taylor series is processed using the following variables: \S is total sum, \Sn is the new addition in the n-th step. If \Sn is zero (in accordance to the \apFRAC register) then we stop the calculation.

apnum.tex

```
\def\apEXPa{%
   \def\testDot ##1##2\relax##3{\ifx##1.}%
   \K=0 \N=0                                  % K = 0, N = 0
   \loop \expandafter \testDot\OUT \relax     % loop if (OUT >= 1)
```

\EXP: 3, 4–6, 40–43, 48, 50 \apEXPa: 40

```
843:       \iftrue \else                          %         OUT = OUT/2
844:          \apDIV\OUT{2}%                      %         K++
845:          \advance\K by1                      %         repeat
846:          \repeat                             % oriOUT = 2^K * OUT, OUT < 1
847:    \advance\apFRAC by\K
848:    \def\S{1}\def\Sn{1}\N=0 \let\X=\OUT       % S = 1, Sn = 1, N = 0, X = OUT
849:    \loop \advance\N by1                      % loop N++
850:       \do\Sn=\apDIV{\apMUL\Sn\X}{\the\N};%    %      Sn = Sn * X / N
851:       \apTAYLOR\iftrue \repeat               %      S = S + Sn (... Taylor)
852:    \N=0
853:    \loop \ifnum\N < \K                       % loop if (N < K)
854:       \apPOW\OUT{2}%                         %      OUT = OUT^2
855:       \advance\N by1 \repeat                 %      N++
856:    \apFRAC=\digits\relax  \apROUND\OUT\apFRAC
857: }
```

The macro \apTAYLOR is ready for general usage in the form:

```
\def\S{...}\def\Sn{...}\N=... % setting initial values for N=0
\loop
   ...                        % auxiliary calculation
   \do\Sn=\apDIV{...}{...};%  % calculation of new addition \Sn
                              % (division must be the last activity)
   \apTAYLOR \iftrue \repeat  % does S = S + Sn and finishes if Sn = 0
```

apnum.tex

```
858: \def\apTAYLOR#1{\ifnum\apSIGN=0 \let\OUT=\S \else \apPLUS\S\Sn \let\S=\OUT }
```

If the argument (saved i the \OUT macro) is greater or equal 4 then \apEXPb macro is executed. The $d = \lfloor x/\ln 10 \rfloor$ is calculated here. This is the number of decimal digits in the result before the decimal point. The result is in the form

$$e^x = e^{x - d\cdot\ln 10}\cdot 10^d.$$

The argument of the exponential function is less than $\ln 10 \doteq 2.3$ for this case, so we can call the \EXP macro recursively. And the result is returned in scientific form if $d \geq$ \apEX.

apnum.tex

```
859: \def\apEXPb{%
860:    \let\X=\OUT \apLNtenexec  \apDIV\X\apLNten  \let\D=\OUT
861:    \apROUND\D{0}%             % D = floor( X/ln(X) )
862:    \ifnum\D<\apEX  \advance\apFRAC by\D \relax \apLNtenexec \fi
863:    \EXP{\X-\D*\apLNten}%      % mantissa = EXP(X-D*LN(10))
864:    \ifnum\D<\apEX  \apROLL\OUT\D \apE=0 \else \apE=\D \relax \fi
865:    \apFRAC=\digits  \apROUND\OUT\apFRAC  % OUT = mantissa * 10^D
866: }
```

The logarithm function $\ln x$ (inverse to e^x) is implemented in \LN macro by Taylor series in the point zero of the arg tanh function:

$$\ln x = 2\ \operatorname{arg\,tanh}\frac{x-1}{x+1} = 2\left(\frac{x-1}{x+1} + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \cdots\right).$$

This series converges quickly when x is approximately equal to one. The idea of the macro \LN includes the following steps:

- Whole calculation is in the group \apINIT...\apEND. Enlarge the \apFRAC numeric precision by three digits in this group.
- Read the argument \X using \evaldef.
- If the argument is non positive, print error and skip the next processing.
- If the argument is in the interval $(0, 1)$, set new argument as 1/argument and remember the "minus" sign for the calculated \OUT, else the \OUT remains to be positive. This uses the identity $\ln(1/x) = -\ln x$.

\apTAYLOR: 41–42, 44, 46–47 \apEXPb: 40–41 \LN: 3, 4, 6, 41–42, 48

- shift the decimal point of the argument by M positions left in order to the new argument is in the interval $[1, 10)$.
- Let $x \in [1, 10)$ be the argument calculated as mentioned before. Calculate roughly estimated $\widetilde{\ln x}$ using \apLNr macro. This macro uses linear interpolation of the function $\ln x$ in eleven points in the interval $[1, 10]$.
- Calculate $A = x/\exp(\widetilde{\ln x})$. The result is approximately equal to one, because $\exp(\ln x) = x$.
- Calculate $\ln A$ using the Taylor series above.
- The result of $\ln x$ is equal to $\ln A + \widetilde{\ln x}$, because $x = A \cdot \exp(\widetilde{\ln x})$ and $\ln(ab) = \ln a + \ln b$.
- The real argument is in the form $x \cdot 10^M$, so \OUT is equal to $\ln x + M \cdot \ln(10)$ because $\ln(ab) = \ln a + \ln b$ and $\ln(10^M) = M \ln(10)$. The $\ln(10)$ value with desired precision is calculated by \apLNtenexec macro. This macro saves its result globally when firstly calculated and use the calculated result when the \apLNtenexec is called again.
- Round the \OUT to the \apFRAC digits.
- Append "minus" to the \OUT if the input argument was in the interval $(0, 1)$.

apnum.tex

```
867: \def\LN#1{\relax \apINIT              % OUT = LN(#1) ...
868:    \evalmdef\X{#1}%                   % X = #1
869:    \localcounts \M \N \E;%
870:    \E=\apE
871:    \edef\digits{\the\apFRAC}\advance\apFRAC by4
872:    \ifnum\apSIGN>0 \else \apERR{\string\LN: argument {\X} is out of range}\apRETURN\fi
873:    \apDIG\OUT\relax \M=\apnumD        % find M: X = mantissa * 10^M
874:    \ifnum\M>-\E \def\sgnout{1}\else   % if X in (0,1):
875:       \def\sgnout{-1}%                %    sgnout = -1
876:       \do\X=\apDIV 1\X;\E=-\E         %    X = 1/X
877:       \apDIG\OUT\relax \M=\apnumD     %    find M: X = mantissa * 10^M
878:    \fi                                % else sgnout = 1
879:    \advance\M by-1                    % M = M - 1
880:    \ifnum\M=0 \else\apROLL\X{-\M}\fi % X = X * 10^(-M), now X in (1,10)
881:    \advance\M by\E                    % M = M + E (sientific format of numbers)
882:    \do\lnX=\apLNr\X;%                 % lnX = LN(X) ... roughly estimate
883:    \do\A=\apDIV\X{\EXP\lnX};%         % A = X / EXP(lnX)  ... A =approx= 1
884:    \apLNtaylor                        % OUT = LN(A)
885:    \do\LNOUT=\apPLUS\OUT\lnX;%        % LNOUT = OUT + LNrOUT
886:    \ifnum\M>0                         % if M > 0
887:       \apLNtenexec                    % LNtenOUT = ln(10)
888:       \apPLUS\LNOUT{\apMUL{\the\M}{\apLNten}}% OUT = LNOUT + M * LNten
889:    \fi
890:    \ifnum\apSIGN=0 \else \apSIGN=\sgnout \fi % if (OUT != 0) apSIGN = saved sign
891:    \apROUND\OUT\digits                % round result to desired precision
892:    \ifnum\apSIGN<0 \xdef\OUT{-\OUT}\else \global\let\OUT=\OUT \fi
893:    \apEND
894: }
```

The macro \apLNtaylor calculates $\ln A$ for $A \approx 1$ using Taylor series mentioned above.

apnum.tex

```
895: \def\apLNtaylor{%
896:    \apDIV{\apPLUS{\A}{-1}}{\apPLUS{\A}{1}}%  % OUT = (A-1) / (A+1)
897:    \ifnum\apSIGN=0 \def\OUT{0}\else          % ln 1 = 0 else:
898:       \let\Sn=\OUT \let\Kn=\OUT \let\S=\OUT  %    Sn = OUT, Kn = OUT, S = OUT
899:       \apPOW\OUT{2}\apROUND\OUT\apFRAC \let\XX=\OUT  %    XX = OUT^2
900:       \N=1                                   %    N = 1
901:       \loop \advance\N by2                   %    loop N = N + 2
902:             \do\Kn=\apMUL\Kn\XX\apROUND\OUT\apFRAC;%   Kn = Kn * XX
903:             \do\Sn=\apDIV\Kn{\the\N};%        %         Sn = Kn / N
904:             \apTAYLOR\iftrue \repeat         %         S = S + Sn (Taylor)
905:       \apMUL\S{2}%                           %    OUT = 2 * OUT
906:    \fi
907: }
```

The macro \apLNr finds an estimation $\widetilde{\ln x}$ for $x \in [1, 10)$ using linear approximation of $\ln x$ function. Only direct ⟨*dimen*⟩ and ⟨*count*⟩ calculation with TEX registers is used, no long numbers

\apLNtaylor: 42–43 \apLNr: 42–43

apnum.tex calculation. The $\ln x_i$ is pre-calculated for $x_i = i$, $i \in \{1,2,3,4,5,6,7,8,9,10\}$ and the values are inserted after the \apLNra macro call. The input value x is set as \dimen0.

The \apLNra {⟨*valueA*⟩}{⟨*valueB*⟩} macro reads the pre-calculated values repeatedly in the loop. The loop ends if \apnumC (i. e. x_i) is greater than x. Then we know that $x \in [x_{i-1}, x_i)$. The linear interpolation is

$$\widetilde{\ln x} = f(x_{i-1}) + \big(f(x_i) - f(x_{i-1})\big)\,(x - x_{i-1}),$$

where $f(x_{i-1}) = \langle valueA \rangle$, $f(x_i) = \langle valueB \rangle$ and $x =$ \dimen0. The rest of the pre-calculated values is skipped by processing \next to \relax.

The pre-calculated approximation of $\ln 10$ is saved in the macro \apLNrten because we use it at more places in the code.

apnum.tex

```
\def\apLNr#1{\dimen0=#1pt \apnumC=1
   \apLNra {0}{.69}{1.098}{1.386}{1.609}{1.791}{1.9459}{2.079}{2.197}{\apLNrten}{}\relax
}
\def\apLNra #1#2{\advance\apnumC by1
   \ifx\relax#2\relax \let\OUT=\apLNrten \let\apNext=\relax
   \else
      \ifdim\dimen0<\apnumC pt  % linear interpolation:
         \advance\dimen0 by-\apnumC pt \advance\dimen0 by1pt % dimen0 = x - x_{i-1}
         \dimen1=#2pt \advance\dimen1 by-#1pt           % dimen1 = f(x_i) - f(x_{i-1})
         \dimen1=\expandafter\apNOPT\the\dimen0 \dimen1 % dimen1 = (x - x_{i-1}) * dimen1
         \advance\dimen1 by#1pt                         % dimen1 = f(x_{i-1}) + dimen1
         \edef\OUT{\expandafter\apNOPT\the\dimen1}%     % OUT = dimen1
         \def\apNext##1\relax{}%
      \else \def\apNext{\apLNra{#2}}%
   \fi\fi \apNext
}
\def\apLNrten{2.302585}   % apLNrten = ln 10 (roughly)
```

The \apLNtenexec macro calculates the $\ln 10$ value with the precision given by \apFRAC. The output is prepared to the \apLNten macro. The \apLNtenexec saves globally the result to the macro \LNten:⟨*apFRAC*⟩ in order to use it if the value is needed again. This saves time.

apnum.tex

```
\def\apLNtenexec{%                          % OUT = ln 10 ...
   \expandafter\ifx\csname LNten:\the\apFRAC\endcsname \relax
      \begingroup \apTOT=0
         \do\A=\apDIV{10}{\EXP\apLNrten};%    % A = 10 / exp(LNrten)
         \apLNtaylor                         % OUT = ln A
         \apPLUS\OUT\apLNrten                % OUT = OUT + LNrten
         \global\expandafter\let\csname LNten:\the\apFRAC\endcsname=\OUT
      \endgroup
   \fi
   \expandafter\let\expandafter \apLNten \csname LNten:\the\apFRAC\endcsname
}
```

The constant π is saved in the \apPIvalue macro initially with 30 digits. If user needs more digits (using \apFRAC > 30) then the \apPIvalue is recalculated and the \apPIdigits is changed appropriately.

apnum.tex

```
\def\apPIvalue{3.14159265358979323846264338327 9}
\def\apPIdigits{30}
```

The macro \apPIexec prepares the π constant with \apFRAC digits and saves it to the \apPI macro. And $\pi/2$ constant with \apFRAC digits is saved to the \apPIhalf macro. The \apPIexec uses macros \apPI:⟨*apFRAC*⟩ and \apPIh:⟨*apFRAC*⟩ where desired values are usually stored. If the values are not prepared here then the macro \apPIexecA calculates them.

apnum.tex

```
\def\apPIexec{%
   \expandafter\ifx\csname apPI:\the\apFRAC\endcsname \relax \apPIexecA \else
      \expandafter\let\expandafter\apPI\csname apPI:\the\apFRAC\endcsname
      \expandafter\let\expandafter\apPIhalf\csname apPIh:\the\apFRAC\endcsname
```

\apLNra: 43 \apLNrten: 43 \apLNtenexec: 41–43 \apLNten: 41–43 \apPIvalue: 43–44
\apPIdigits: 43–45 \apPIexec: 43, 45, 47–48 \apPI: 43–46 \apPIhalf: 43–48

```
942:    \fi
943: }
```

The macro \apPIexecA creates the π value with \apFRAC digits using the \apPIvalue, which is rounded if \apFRAC < \apPIdigits. The \apPIhalf is calculated from \apPI. Finally the macros \apPI:⟨*apFRAC*⟩ and \apPIh:⟨*apFRAC*⟩ are saved globally for saving time when we need such values again.

apnum.tex

```
944: \def\apPIexecA{%
945:    \ifnum\apPIdigits<\apFRAC \apPIexecB \fi
946:    \let\apPI=\apPIvalue
947:    \ifnum\apPIdigits>\apFRAC \apROUND\apPI\apFRAC \fi
948:    \apnumP=\apTOT \apTOT=0 \apDIV\apPI2\let\apPIhalf=\OUT \apTOT=\apnumP
949:    \global\expandafter\let\csname apPI:\the\apFRAC\endcsname=\apPI
950:    \global\expandafter\let\csname apPIh:\the\apFRAC\endcsname=\apPIhalf
951: }
```

If \apFRAC > \apPIdigits then new \apPIvalue with desired decimal digits is generated using \apPIexecB macro. The Chudnovsky formula is used:

$$\pi = \frac{53360 \cdot \sqrt{640320}}{S}, \quad S = \sum_{n=0}^{\infty} \frac{(6n)!\,(13591409 + 545140134\,n)}{(3n)!\,(n!)^3\,(-262537412640768000)^n}$$

This converges very good with 14 new calculated digits per one step where new S_n is calculated. Moreover, we use the identity:

$$F_n = \frac{(6n)!}{(3n)!\,(n!)^3\,(-262537412640768000)^n}, \quad F_n = F_{n-1} \cdot \frac{8\,(6n-1)\,(6n-3)\,(6n-5)}{n^3\,(-262537412640768000)}$$

and we use auxiliary integer constants A_n, B_n, C_n with following properties:

$$A_0 = B_0 = C_0 = 1,$$
$$A_n = A_{n-1} \cdot 8\,(6n-1)\,(6n-3)\,(6n-5), \quad B_n = B_{n-1} \cdot n^3, \quad C_n = C_{n-1} \cdot (-262537412640768000),$$
$$F_n = \frac{A_n}{B_n C_n},$$
$$S_n = \frac{A_n\,(13591409 + 545140134\,n)}{B_n C_n}$$

apnum.tex

```
952: \def\apPIexecB{\apINIT
953:    \localcounts \N \a \c;%
954:    \apTOT=0 \advance\apFRAC by2
955:    \def\apSQRTxo{800.199975006248}% initial value for Newton method for SQRT
956:    \SQRT{640320}%
957:    \let\sqrtval=\OUT
958:    \N=0 \def\An{1}\def\Bn{1}\def\Cn{1}\def\S{13591409}%
959:    \loop
960:       \advance\N by 1
961:       \a=\N \multiply\a by6 \advance\a by-1 \c=\a
962:             \advance\a by-2 \multiply\c by\a            % An = An * 8 * (6N-5) *
963:             \advance\a by-2 \multiply\a by8             %      * (6N-3) * (6N-1)
964:             \apMUL\An{\apMUL{\the\a}{\the\c}}\let\An=\OUT
965:       \c=\N \multiply\c by\N                            % Bn = Bn * n^3
966:             \apMUL\Bn{\apMUL{\the\c}{\the\N}}\let\Bn=\OUT
967:             \apMUL\Cn{-262537412640768000}\let\Cn=\OUT  % Cn = Cn * K3
968:       \apDIV{\apMUL\An{\apPLUS{13591409}{\apMUL{545140134}{\the\N}}}}{\apMUL\Bn\Cn}%
969:       \let\Sn=\OUT % Sn = An * (K1 + K2 * N) / (Bn * Cn)
970:       \apTAYLOR \iftrue \repeat
971:    \advance\apFRAC by-2
972:    \apDIV{\apMUL{\sqrtval}{53360}}\S
973:    \global\let\apPIvalue=\OUT
```

\apPIexecA: 43–44 \apPIexecB: 44

```
974:    \xdef\apPIdigits{\the\apFRAC}%
975:    \apEND
976: }
```

The macros for users \PI and \PIhalf are implemented as "function-like" macros without parameters.

apnum.tex

```
977: \def\PI{\relax \apPIexec \let\OUT=\apPI}
978: \def\PIhalf{\relax \apPIexec \let\OUT=\apPIhalf}
```

The macros \SIN and \COS use the Taylor series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

These series converge good for $|x| < 1$. The main problem is to shift the given argument $x \in \mathbf{R}$ to the range $[0, 1)$ before the calculation of the series is started. This task is done by \apSINCOSa macro, the common code for both, \SIN and \COS macros.

The macro \apSINCOSa does the following steps:

- It advances \apFRAC by three and evaluates the argument.
- Note, that the macro \apSINCOSx means \apSINx or \apCOSx depending on the given task.
- The macro \signK includes 1. It can be recalculated to -1 later.
- If the argument is zero then the result is set and next computation is skipped. This test is processed by \apSINCOSo\apCOSx.
- If the argument is negative then remove minus and save \sign. This \sign will be applied to the result. The \sign is always + when \COS is calculated. This folows the identities $\sin(-x) = -\sin x$ and $\cos(-x) = \cos x$.
- The \apFRAC is saved and \apTOT=0.
- The \apPIexec is processed. The \apPI and \apPIhalf are ready after such processing.
- After \X div \apPI (rounded to integer) we have \K in \OUT, where $\X = x' + \K \cdot \pi$ and $x' \in [0, \pi)$. We set $\X := x'$ because of the identities $\sin x = (-1)^k \sin(x + k\pi)$, $\cos x = (-1)^k \cos(x + k\pi)$. The sign $(-1)^k$ is saved to \signK macro.
- If the x' is zero then the result is set by \apSINCOSo\apCOSx and the rest of calculating is skipped.
- The $|\X - \pi/2|$ is saved to \XmPIh macro.
- If $\X \in (\pi/4, \pi/2)$ then $x' = \XmPIh$. We use identities $\sin x = \cos(\pi/2 - x)$, $\cos x = \sin(\pi/2 - x)$. Set $\X = x'$. The meaning of \apSINCOSx (\apSINx or \apCOSx) is flipped in such case.
- If the x' is zero then the result is set by \apSINCOSo\apSINx and the rest of calculating is skipped.
- Now $\X \in (0, \pi/4)$, i. e. $|\X| < 1$ and we can use Taylor series. The \apSINCOSx (i. e. \apSINx or \apCOSx) macro initializes the computation of Taylor series mentioned above. The $\XX = \X^2$ is prepared. The Taylor series is processed in the loop as usually.
- The the sign of the output is \sign\signK.
- If the sign of the result is negative, the "minus" is added to the \OUT.

apnum.tex

```
980: \def\SIN{\relax \let\apSINCOSx=\apSINx \apSINCOSa}
981: \def\COS{\relax \let\apSINCOSx=\apCOSx \apSINCOSa}
982: \def\apSINCOSa#1{\apINIT
983:    \advance\apFRAC by3
984:    \evalmdef\X{#1}\apEnum\X
985:    \def\signK{1}\apSINCOSo\apCOSx
986:    \ifnum\apSIGN<0 \apREMfirst\X \def\sign{-}\else\def\sign{+}\fi
987:    \ifx\apSINCOSx\apCOSx \def\sign{+}\fi
988:    \edef\apFRACsave{\the\apFRAC}%
989:    \apPIexec
990:    \apFRAC=0 \apDIV\X\apPI            % OUT = X div PI
991:    \ifnum\apSIGN=0 \apSIGN=1 \else
992:       \let\K=\OUT
```

\PI: 3, 4, 45, 48 \PIhalf: 3, 4, 45, 48 \SIN: 3, 4–5, 8, 45–46, 48–50 \COS: 3, 4–5, 45–46, 48–50
\apSINCOSa: 45–46

```
      \do\X=\apPLUS\X{-\apMUL\K\apPI};%  X := X - K * PI
      \apROLL\K{-1}\apROUND\K{0}%
      \ifodd 0\XOUT\space \def\signK{-1}\else\def\signK{1}\fi
   \fi
   \apSINCOSo\apCOSx
   \apFRAC=\apFRACsave \relax
   \do\XmPIh=\apPLUS\X{-\apPIhalf};%    XmPIh = | X - PI/2 |
   \apSINCOSo\apSINx
   \ifnum\apSIGN<0 \apREMfirst\XmPIh
   \else % X in (PI/2, PI)
      \do\X=\apPLUS\apPI{-\X};%
      \ifx\apSINCOSx\apCOSx \apSIGN=-\signK \edef\signK{\the\apSIGN}\fi
   \fi                    % X in (0, PI/2):
   \apMINUS\X{.78}%       % OUT = X - cca PI/4
   \ifnum\apSIGN<0 \else % if X in (PI/4, PI/2) :
      \let\X=\XmPIh       %    X = | X - PI/2 |;  SIN <-> COS
      \ifx\apSINCOSx\apSINx \let\apSINCOSx=\apCOSx \else \let\apSINCOSx=\apSINx \fi
   \fi
   \localcounts \N \NN;%
   \do\XX=\apPOW\X{2}\ROUND\OUT\apFRAC;%
   \apSINCOSx             % X in (0, PI/4), initialize Taylor SIN X or COS X
   \loop
      \advance\N by1 \NN=\N
      \advance\N by1 \multiply\NN by\N
      \do\Sn=\apDIV{\apMUL\Sn\XX}{-\the\NN};% Sn = - Sn * X^2 / N*(N+1)
      \apTAYLOR \iftrue\repeat
   \apSIGN=\sign\signK
   \ifnum\apTOT=0 \advance\apFRAC by-3 \else \apFRAC=\apTOT \fi
   \ifnum\apFRAC<0 \apFRAC=-\apFRAC \fi
   \apROUND\OUT\apFRAC
   \def\X{0}\ifx\OUT\X \apSIGN=0 \fi
   \ifnum\apSIGN<0 \edef\OUT{-\OUT}\fi
   \apEND
}
```

The macros \apSINx and \apCOSx initialize the calculation of the Taylor series.

apnum.tex

```
\def\apSINx{\let\S=\X \N=1 \let\Sn=\X}
\def\apCOSx{\def\S{1}\N=0 \let\Sn=\S}
```

The \apSINCOSo ⟨*sequence*⟩ macro is used three times in the \apSINCOSa. It tests if the current result is zero. If it is true then the \OUT is set as zero or it is set to \signK (if processed function is equal to the ⟨*sequence*⟩).

apnum.tex

```
\def\apSINCOSo#1{\ifnum\apSIGN=0 \ifx#1\SCgo \apSIGN=\signK \let\OUT=\signK \fi \apRETURN\fi}
```

The macro \TAN uses the identity $\tan x = \sin x / \cos x$ and calculates the denominator first. If it is zero then \apERR prints "out of range" message else the result is calculated.

apnum.tex

```
\def\TAN#1{\relax \apINIT
   \advance\apFRAC by3
   \evalmdef\X{#1}\apEnum\X
   \advance\apFRAC by-3
   \do\denom=\COS\X;%
   \ifnum\apSIGN=0 \apERR{\string\TAN: argument {\X} is out of range}\apRETURN\fi
   \SIN\X
   \apDIV{\SIN\X}\denom
   \apEND
}
```

The macro \ATAN calculates the inverse of tangens using series

$$\arctan\frac{1}{x} = \frac{x}{1+x^2} + \frac{2}{3}\frac{x}{(1+x^2)^2} + \frac{2}{3}\frac{4}{5}\frac{x}{(1+x^2)^3} + \frac{2}{3}\frac{4}{5}\frac{6}{7}\frac{x}{(1+x^2)^4} + \cdots$$

\apSINx: 45–46 \apCOSx: 45–46 \apSINCOSo: 45–46 \TAN: 3, 4, 46, 48 \ATAN: 3, 4–5, 47–48

This converges relatively good for $|x| > 1$. I was inspired by the Claudio Kozický's semestral work from the course "Typography and TEX" at ČVUT in Prague.

The macro \ATAN takes the argument x and uses identity $\arctan(-x) = -\arctan(x)$ when x is negative. If $x > 1$ then the identity

$$\arctan(x) = \frac{\pi}{2} - \arctan\frac{1}{x}$$

is used and $\arctan(1/x)$ is calculated by \apATANox macro using the series above. Else the argument is re-calculated $x := 1/x$ and the \apATANox is used. When $x = 1$ then the \apPIhalf/2 is returned directly.

apnum.tex

```
\def\ATAN#1{\relax \apINIT
   \advance\apFRAC by3
   \evalmdef\X{#1}\apEnum\X
   \ifnum\apSIGN=0 \def\OUT{0}\apRETURN\fi
   \ifnum\apSIGN<0 \def\sign{-}\apREMfirst\X \else\def\sign{}\fi
   \let\tmp=\X \apDIG\tmp\relax
   \ifnum\apnumD>0      % if X > 1:
      \apPIexec         %    OUT = apPIhalf - apATANox
      \def\tmp{1}\ifx\tmp\X \apDIV\apPIhalf2\else \apATANox \apPLUS\apPIhalf{-\OUT}\fi
   \else                % else
      \do\X=\apDIV{1}\X;%    X := 1/X
      \apATANox         %    OUT = apATANox
   \fi
   \ifnum\apTOT=0 \advance\apFRAC by-3 \else \apFRAC=\apTOT \fi
   \ifnum\apFRAC<0 \apFRAC=-\apFRAC \fi
   \apROUND\OUT\apFRAC
   \ifx\sign\empty\apSIGN=1 \else \edef\OUT{-\OUT}\apSIGN=-1 \fi
   \apEND
}
```

The macro \apATANox calculates $\arctan(1/x)$ using series mentioned above.

apnum.tex

```
\def\apATANox{%
   \localcounts \N;%
   \do\XX=\apPLUS{1}{\apPOW\X{2}}\apROUND\OUT\apFRAC;% XX = 1 + X^2
   \do\Sn=\apDIV\X\XX \apROUND\OUT\apFRAC;%          % Sn = X / (1+X^2)
   \N=1 \let\S=\Sn
   \loop
      \advance\N by1
      \do\Sn=\apMUL{\the\N}\Sn;%
      \advance\N by1
      \do\Sn=\apDIV\Sn{\apMUL{\the\N}\XX};% Sn = Sn * N / ((N+1) * (1+X^2))
      \apTAYLOR \iftrue \repeat
}
```

The macros \ASIN and \ACOS for functions $\arcsin(x)$ and $\arccos(x)$ are implemented using following identities:

$$\arcsin(x) = \arctan\frac{x}{\sqrt{1-x^2}}, \qquad \arccos(x) = \frac{\pi}{2} - \arcsin(x)$$

apnum.tex

```
\def\ASIN#1{\relax \apINIT
   \evalmdef\X{#1}\apEnum\X \edef\sign{\the\apSIGN}%
   \apPLUS 1{-\apPOW\X2}%  OUT = 1 - X^2
   \ifnum\apSIGN<0 \apERR{\string\ASIN: argument {\X} is out of range}\apRETURN\fi
   \do\sqrt=\SQRT\OUT;%    sqrt = SRQT {1 - X^1}
   \ifnum\apSIGN=0 \apPIexec
      \ifnum\sign<0 \edef\OUT{-\apPIhalf}\apSIGN=-1  % ASIN(-1) = -PI/2
      \else \let\OUT=\apPIhalf \apSIGN=1 \fi         % ASIN(1)  = PI/2
      \apRETURN \fi
   \ATAN{\X/\sqrt}%        OUT = arctan ( X / SQRT {1 - X^2} )
   \apEND
}
```

\apATANox: 47 \ASIN: 3, 4, 47–48 \ACOS: 3, 4, 48

```
1083: \def\ACOS#1{\relax \apPIexec \apPLUS\apPIhalf{-\ASIN{#1}}}
```

2.11 Printing expressions

The \eprint {⟨expression⟩}{⟨declaration⟩} macro works in the group \bgroup...\egroup. This means that the result in math mode is math-Ord atom. The macro interprets the ⟨expression⟩ in the first step like \evaldef. This is done by \apEVALb#1\limits. The result is stored in the \tmpb macro in Polish notation. Then the internal initialization is processed in \apEPi and user-space initialization is added in \apEPj and #2. Then \tmpb is processed. The \apEPe can do something end-game play but typically it is \relax.

apnum.tex

```
1088: \def\eprint#1#2{\bgroup \apnumA=0 \apnumE=1 \apEVALb#1\limits
1089:     \let\apEPe=\relax \apEPi #2\tmpb \apEPe \egroup
1090: }
```

The \apEPi macro replaces the meaning of all macros typically used in Polish notation of the expression. The original meaning is "to evaluate", the new meaning is "to print". The macro \apEPi is set to \relax in the working group because nested ⟨expressions⟩ processed by nested \eprints need not to be initialized again.

There is second initialization macro \apEPj (similar to the \apEPi) which is empty by default. Users can define their own function-like functions and they can put the printing initialization of such macros here.

apnum.tex

```
1091: \def\apEPi{\let\apPLUS=\apEPplus \let\apMINUS=\apEPminus
1092:    \let\apMUL=\apEPmul \let\apDIV=\apEPdiv \let\apPOWx=\apEPpow \def\apPPn##1{##1}%
1093:    \let\EXP=\apEPexp \def\LN{\apEPf{ln}}\let\SQRT=\apEPsqrt
1094:    \def\SIN{\apEPf{sin}}\def\COS{\apEPf{cos}}\def\TAN{\apEPf{tan}}%
1095:    \def\ASIN{\apEPf{arcsin}}\def\ACOS{\apEPf{arccos}}\def\ATAN{\apEPf{arctan}}%
1096:    \let\PI=\pi \def\PIhalf{{\pi\over2}}%
1097:    \let\ABS=\apEPabs \let\FAC=\apEPfac \let\BINOM=\apEPbinom
1098:    \let\SGN=\apEPsgn \let\iDIV=\apEPidiv \let\iMOD=\apEPimod
1099:    \let\iFLOOR=\apEPifloor \let\iFRAC=\apEPifrac
1100:    \let\apEPk=\empty \let\apEPy=\empty \def\apEPx{.}%
1101:    \let\apEPi=\relax \apEPj
1102: }
1103: \def\apEPj{}
```

All parameters are processed in new group (excepts individual constants). For example we have \apPLUS{a}{\apDIV{b}{c}} in the \tmpb. Then the a+{\apDIV{b}{c}} is processed and thus a+{b\over c} is printed. As noted above, the outer group is set by \eprint macro itself.

When we process the \tmpb with the output of the ⟨expression⟩ interpreter then the original positions of the round brackets are definitively lost. We must to print these brackets if it is required by usual math syntax. For example \apPLUS{1}{-2} must be printed as 1+(-2). But \apPLUS{1}{2} must be printed as 1+2. So, we print all parameters using \apEPp{⟨parameter⟩}{⟨a⟩⟨b⟩⟨c⟩⟨d⟩}. This macro decides if the parameter will be surrounded by brackets or not. So, the "printing" meaning of \apPLUS (or \apMINUS respectively) and prepared in \apEPplus (or \apEPminus respectively) looks like:

apnum.tex

```
1104: \def\apEPplus#1#2{\apEPp{#1}{?...}+\apEPp{#2}{!...}}
1105: \def\apEPminus#1#2{\apEPp{#1}{?...}-\apEPp{#2}{!!..}}
```

The usage of \apEPp{⟨parameter⟩}{⟨a⟩⟨b⟩⟨c⟩⟨d⟩} has the following meaning:

- if ⟨a⟩ is ! and the ⟨parameter⟩ is a negative constant or a -⟨expression⟩ or
- if ⟨b⟩ is ! and main operator $M_{\rm op}$ of the ⟨parameter⟩ is + or - or
- if ⟨c⟩ is ! and main operator $M_{\rm op}$ of the ⟨parameter⟩ is * or
- if ⟨d⟩ is ! and main operator $M_{\rm op}$ of the ⟨parameter⟩ is / or ^

then \apEPp prints brackets around the ⟨parameter⟩ using \left(⟨parameter⟩\right). Else it doesn't use brackets around the ⟨parameter⟩ (but the ⟨parameter⟩ itself is printed in a group unless it is single element: constant, variable).

\eprint: 7, 8, 48–51 \apEPe: 48, 50 \apEPi: 48–49, 51 \apEPj: 8, 48 \apEPplus: 48, 50
\apEPminus: 48, 50

The rule for the parameter ⟨*a*⟩ has the exception: if ⟨*a*⟩ is `?` and the ⟨*parameter*⟩ is a negative constant or a `-`⟨*expression*⟩, then brackets are not used if and only if this is "very first parameter", it means that the ⟨*parameter*⟩ is first:

- at beginning of the whole ⟨*expression*⟩ given as an argument of `\eprint` or
- immediately after an opening bracket or
- at beginning of a numerator or a denominator in a fraction or
- at beginning of an exponent.

For example `-1+1` is preprocessed as `\apPLUS{-1}{1}` and printed as `-1+1` because first parameter has ⟨*a*⟩ equal to `?` and we are at beginning of the expression. But `1+-1` is preprocessed as `\apPLUS{1}{-1}` and printed as `1+(-1)` because second parameter has ⟨*a*⟩ equal to `!`. The `2*(-1+5)` is printed as `2\cdot(-1+5)` because `-1` is "very first parameter" after opening bracket. Another examples: `-1+-1+-1` is printed as `-1+(-1)+(-1)`, `a+b*c` is printed as `a+b\cdot c` (without brackets), The `1-(2+3)` is printed "as is" but `1+(2+3)` is printed as `1+2+3`.

The question about to be "very first parameter" is controlled by the value of `\apEPx` macro. It is started as `.` and it is replaced by `!` whenever ⟨*a*⟩ is `!`. It is reverted to `.` when open bracket is printed.

The unary minus in the cases like `-(a+b)` are transformed to `\apMUL{-1}{\apPLUS{a}{b}}` by the ⟨*expression*⟩ interpreter. But we don't need to print `-1\cdot(a+b)`. So, the printing version of `\apMUL` stored in the macro `\apEPmul` have an exception. First, we do the test, if `#1` is equal to `-1`. If this is true, then we print only the unary minus (no whole `-1\cdot`). Else we print the whole first parameter (enclosed in braces if its M_{op} is `+` or `-`). The first case has two sub-cases: if `\apEPx` is `!` (it means that it is not "very first parameter" then brackets are used around `-`⟨*expression*⟩.

The second parameter is enclosed in brackets if its M_{op} is `+` or `-`.

apnum.tex

```
\def\apEPmul#1#2{\def\tmpa{#1}\def\tmpb{-1}%
   \ifx\tmpa\tmpb \if\apEPx!\left(-\apEPp{#2}{!!..}\right)\else
                                  -\apEPp{#2}{!!..}\fi
   \else \apEPp{#1}{?!..}\apMULop \apEPp{#2}{!!..}\fi
}
```

The `\apEPdiv` macro used for printing `\apDIV` is very easy. We needn't to set the outer group here because each parameter is enclosed in the group. We need not to add any round brackets here because fraction generated by `\over` is self explanatory from priority point of view. If you need to redefine `\apEPdiv` with the operator `/` instead `\over` then you need to redefine `\apEPmul` too because you must enclose parameters with $M_{\text{op}} =$ `\apDIV` by brackets in such case.

apnum.tex

```
\def\apEPdiv#1#2{{\def\apEPx{.}#1}\over{\def\apEPx{.}#2}}
```

The `\apEPpow` macro used for printing `^` includes another speciality. When the base (the first ⟨*parameter*⟩) is a function-like macro `\SIN`, `\COS` etc. then we need to print `\SIN{X}^2` as `\sin^2 x`. The test if the base is such special functions-like macro is performed by `\apEPpowa{`⟨*base*⟩`}\end{`⟨*exponent*⟩`}`. If this is true then `\apEPpowa` saves the ⟨*exponent*⟩ to the temporary macro `\apEPy` and only ⟨*base*⟩ is processed (the `\apEPy` is printed inside this processing) else `\apEPy` is empty and the ⟨*base*⟩ enclosed in brackets is followed by `^{`⟨*exponent*⟩`}`. Note that the ⟨*base*⟩ isn't enclosed by brackets only if the ⟨*base*⟩ is single and positive operand.

apnum.tex

```
\def\apEPpow#1#2{%
   \let\apEPy=\empty \apEPpowa{#1}\end{#2}%
   \ifx\apEPy\empty \apEPp{#1}{!!!!}^{\def\apEPx{.}#2}\else#1\fi
}
```

The `\apEPpowa` and `\apEPpowb` macros detect the special function-like macro `\SIN`, `\COS` etc. by performing one expansion step on the tested ⟨*base*⟩. If the first ⟨*token*⟩ is `\apEPf` then the special function-like macro is detected. Note that `\SIN`, `\COS` etc. are defined as `\apEPf` in the `\apEPi` macro.

apnum.tex

```
\def\apEPpowa#1{\expandafter\apEPpowb#1;}
\def\apEPpowb#1#2;\end#3{\ifx#1\apEPf \def\apEPy{\let\apEPy=\empty\def\apEPx{.}#3}\fi}
```

`\apEPx`: 48–50 `\apEPmul`: 48–50 `\apEPdiv`: 48–50 `\apEPpow`: 48–50 `\apEPy`: 48–50
`\apEPpowa`: 49 `\apEPpowb`: 49

The functions like \SIN{⟨expression⟩} are printed by \apEPf {⟨name⟩}{⟨expression⟩}. First, the \mathop{⟨name⟩}\nolimits is printed. If \apEPy is non-empty then the exponent is printed by ^{\apEPy}. Finally, the nested ⟨expression⟩ is printed by the nested \eprint.

apnum.tex

```
\def\apEPf#1#2{\begingroup
   \mathop{\rm#1}\nolimits
   \ifx\apEPy\empty \else ^{\apEPy}\let\apEPy=\empty \fi
   \def\apEPk{\mskip-\thinmuskip}%
   \def\apEPx{.}%
   \eprint{#2}{\expandafter\apEPb}\endgroup
}
```

The space-correction macro \apEPk is set to remove the \thinmuskip after \mathop. This will be processed only if the \left(follows: we want to print \sin\apEPk\left(⟨expression⟩\right) because this gives the same result as \sin(⟨expression⟩). On the other hand we don't use \apEPk when simple \sin x is printed without brackets.

By default the ⟨expression⟩ (this is an argument of common function-like macros \SIN, \COS, \EXP etc.) will be printed in brackets (see the default \next definition where closing bracket is printed by \apEPe macro used after expanded \tmpb). But if

- the ⟨expression⟩ is single non-negatie object (a constant or a variable without unary minus) or
- the ⟨expression⟩ is a fraction of the form {⟨nominator⟩\over⟨denominator⟩}

then no brackets are printed around the ⟨expression⟩.

This rule is implemented by the usage of \expandafter\apEPb in the ⟨declaration⟩ part of \eprint (in the code of \apEPf above). It expands the following \tmpb (the result of the expression scanner) and checks the first token and the following parameter. Note that the ⟨expression⟩ scanner generates \apPPn{⟨operand⟩} if and only if the whole ⟨expression⟩ is a single operand.

apnum.tex

```
\def\apEPb#1#2{\def\next{\apEPk\left(\def\apEPe{\right)}}%
   \ifx\apPPn#1\expandafter\apEPd#2.\end{}{\let\next=\relax}.\fi
   \ifx\apDIV#1\let\next=\relax \fi
   \next\let\apEPk=\empty #1{#2}%
}
```

The meaning of \apEPp {⟨parameter⟩}{⟨a⟩⟨b⟩⟨c⟩⟨d⟩} is explained above, see the text where \apEPplus is introduced. Now, we focus to the implementation of this feature. The auxiliary macro \apEPa ⟨first token⟩⟨rest⟩\end{⟨normal⟩}{⟨bracket⟩}⟨a⟩⟨b⟩⟨c⟩⟨d⟩ is used twice: before processing the ⟨parameter⟩ and after processing. The \apEPa inserts the ⟨normal⟩ or ⟨bracket⟩ depending on the condition described above where $M_{\rm op}$ (or unary - when ⟨parameter⟩ is negative constant) is equal to the ⟨first token⟩.

apnum.tex

```
\def\apEPp#1#2{\apEPq#1\end\bgroup{\left(\def\apEPx{.}}#2#1\apEPq#1\end\egroup{\right)}#2}
\def\apEPq#1#2\end#3#4#5#6#7#8{
   \ifx#5!\def\apEPx{!}\fi
   \ifx#1\apEPplus  \ifx#6!#4\else#3\fi\else
   \ifx#1\apEPminus \ifx#6!#4\else#3\fi\else
   \ifx#1\apEPmul   \ifx#7!#4\else#3\fi\else
   \ifx#1\apEPdiv   \ifx#8!#4\else#3\fi\else
   \ifx#1\apEPpow   \ifx#8!#4\else#3\fi\else
   \expandafter\apEPd#1.\end#3{}\apEPx\fi\fi\fi\fi\fi
}
```

If we have variables like \def\X{-17} and the expression looks like 1+\X and the constants stored in the varaibales must to be printed then we have \apPLUS{1}{\X} after expression scanner and we need to print 1+(-17). So we need to try to expand the ⟨paramter⟩ and to test if there is the unary - as a ⟨first-tok⟩. This is done by \apEPd ⟨first-tok⟩⟨rest⟩\end{⟨group-type⟩}{⟨else-part⟩}{⟨dot-or-exclam⟩}.

apnum.tex

```
\def\apEPd#1#2\end#3#4#5{\ifx-#1\if#5!\ifx#3\bgroup\left(\else\right)\fi\fi\else#4\fi}
```

The \apMULop is used as an operation mark for multiplying. It is \cdot by default but user can change this.

\apEPf: 48–50 \apEPk: 48, 50 \apEPb: 50–51 \apEPp: 48–50 \apEPa: 50 \apEPd: 50
\apMULop: 8, 49, 51

apnum.tex
```
\let\apMULop=\cdot
```

The single operand like 2.18 or \X or \FAC{10} is processed directly without any additional material. User can define "variables" as desired. The function-like macros provided by apnum.tex is initialized in \apEPi macro and the "printing macros" \apEPabs, \apEPfac, \apEPbinom, \apEPsqrt, \apEPexp, \apEPsgn, \apEPdivmod, \apEPidiv, \apEPimod, \apEPifloor, \apEPifrac are defined here. The trick with \expandafter\apEPb in the declaration part of the nested \eprint was explained above. Users can re-define these macros if they want.

apnum.tex
```
\def\apEPabs#1{\left|\eprint{#1}{}\right|}
\def\apEPfac#1{\eprint{#1}{\expandafter\apEPb}\,!}
\def\apEPbinom#1#2{{\eprint{#1}{}\choose\eprint{#2}{}}}
\def\apEPsqrt#1{\sqrt{\eprint{#1}{}}}
\def\apEPexp#1{{\rm e}^{\eprint{#1}{}}}
\def\apEPsgn#1{\mathop{\rm sign}\eprint{#1}{\expandafter\apEPb}}
\def\apEPdivmod#1#2#3{\left[\eprint{#2}{\expandafter\apEPb}%
    \mathbin{\rm #1}\eprint{#3}{\expandafter\apEPb}\right]}
\def\apEPidiv{\apEPdivmod{div}}
\def\apEPimod{\apEPdivmod{mod}}
\def\apEPifloor#1{\left\lfloor\eprint{#1}{}\right\rfloor}
\def\apEPifrac#1{\left\{\eprint{#1}{}\right\}}
```

The \corrnum ⟨*token*⟩ macro expects ⟨*token*⟩ as a macro with number. It adds zero before decimal point if the sequence of ⟨*digits*⟩ before decimal point is empty. It uses a macro \apEPc which works at expansion level. First, the occurrence of the - is tested. If it is true then - is expanded and the \apEPc is called again. Else the zero is added if the first token is dot (this means if the ⟨*digits*⟩ before dot is empty).

apnum.tex
```
\def\corrnum#1{\edef#1{\expandafter\apEPc#1\end}}
\def\apEPc#1#2\end{\ifx#1-{-}\apEPc#2\end\else \ifx#1.0.#2\else #1#2\fi\fi}
```

2.12 Conclusion

This code is here only for backward compatibility with old versions of apnum.tex. Don't use these sequences if you are implementing an internal feature because users can re-define these sequences.

apnum.tex
```
\let\PLUS=\apPLUS \let\MINUS=\apMINUS \let\MUL=\apMUL \let\DIV=\apDIV \let\POW=\apPOW
\let\SIGN=\apSIGN \let\ROUND=\apROUND \let\NORM=\apNORM \let\ROLL=\apROLL
```

Here is my little joke. Of course, this macro file works in LaTeX without problems because only TeX primitives (from classical 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{WARNING: the author of apnum package recommends: Never use LaTeX.}\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. The items marked by ≻ are mentioned in user documentation.