The bnumexpr package

Jean-François Burnol
jfbu (at) free (dot) fr

Package version: 1.1 (2014/09/21); documentation date: 2014/09/21.
From source file bnumexpr.dtx. Time-stamp: <21-09-2014 at 15:24:58 CEST>.

Contents

1 Introduction 1

2 Options 2

3 Examples 3

4 Package bnumexpr implementation 4

1 Introduction

Package bnumexpr provides \bnumexpr...\relax which is analogous to \numexpr ...\relax, while allowing arbitrarily big integers. Important items:

1. the \relax token ending the expression is mandatory,
2. one must use either \thebnumexpr or \bnethe\bnumexpr to get a printable result, as \bnumexpr...\relax expands to a private format; however one may embed directly one \bnumexpr...\relax in another \bnumexpr...\relax,
3. one may do \edef\tmp{\bnumexpr 1+2\relax}, and then either use \tmp in another \bnumexpr...\relax, or print it via \bnethe\tmp. The computation is done at the time of the \edef (and two expansion steps suffice),
4. tacit multiplication applies in front of parenthesized sub-expressions, or sub \bnumexpr...\relax (or \numexpr...\relax), or in front of a \count or \dimen register. This may be de-activated by option notacitmul,
5. expressions may be comma separated. On input, spaces are ignored, naturally, and on output the values are comma separated with a space after each comma. This functionality may be turned off via option nocsv,
6. even with options notacitmul and nocsv the syntax is more flexible than with \numexpr: things such as \bnumexpr -(1+1)\relax are legal.

The parser \bnumexpr is a scaled-down version of parser \xintiiexpr from package xintexpr: support for boolean operators, functions such as abs, max, lcm, the ! as factorial, handling of hexadecimal numbers, etc. . . has been removed. The goal here is to extend \numexpr only to the extent of accepting big integers. Thus by default, the syntax allows +,-,*,/, parentheses, and

also \count or \dimen registers or variables. Option allowpower enables ^ as power operator.

Of course, one needs some underlying big integer engine to provide the macros doing the actual computations. By default, bnumexpr uses package xint and its \xintiiAdd, \xintiiSub, and \xintiiMul macros (also \xintiiPow if option allowpower is made use of). As we want here / to do rounded division while xint's \xintiiQuo does Euclidean division, bnumexpr contains a few extra code lines on top of the underlying division macros from xint.sty.

See the discussion of options bigintcalc and custom in section 2 for alternatives.

The starting point for the \bnumexpr parser was not the \xintiiexpr version 1.09n as available (at the time of writing) on CTAN, but a development version for future release 1.1. This is why the version number of package bnumexpr is 1.1. It may well be that the code of the parser is in some places quite suboptimal from the fact that it was derived from code handling much more stuff.

The \xintNewExpr construct has been left out.

I recall from documentation of xintexpr that there is a potential impact on the memory of TEX (the hash table) because each arithmetic operation is done inside a dummy \csname...\endcsname used as single token to move around in one-go the possibly hundreds of digits composing a number.

2 Options

Option bigintcalc says to map the infix operators to the macros from package bigintcalc by Heiko Oberdiek. Note though that / is mapped to \bigintcalcDiv which does *truncated* (not rounded) division.

Option custom leaves it up to the user to specify the macros to be used, i.e. provide definitions for \bnumexprAdd, \bnumexprSub, \bnumexprMul, \bnumexprDiv (and possibly \bnumexprPow). Without it, the package does by default:

```
\RequirePackage{xint}
\let\bnumexprAdd\xintiiAdd
\let\bnumexprSub\xintiiSub
\let\bnumexprMul\xintiiMul
% \bnumexprDiv has custom definition on top of macros from xint.sty
\let\bnumexprPow\xintiiPow % only if option allowpower
```

To let / do euclidean division (like currently in \xintiiexpr) it is thus sufficient to do \let\bnumexprDiv\xintiiQuo after loading the package.

If using option custom: the four arithmetic macros \bnumexprAdd, \bnumexprSub, \bnumexprMul, \bnumexprDiv (and possibly \bnumexprPow) must be expandable, and they must allow arguments in need to be first ('f'-) expanded. They should produce on output (big) integers with no leading zeros, at most one minus sign and no plus sign (else the bnumexpr macro used for handling the - prefix operator may need to be modified). They will be expanded inside \csname...\endcsname. The macros from xint.sty (as well as those of bigintcalc.sty) are expandable in a stronger sense (only two expansion steps suffice). Perhaps speed gains are achievable from dropping these stronger requirements.

Option nocsv makes comma separated expressions illegal.
Option notacitmul removes the possibility of tacit multiplication in front of parentheses, \count registers, sub-expressions.
Option allowpower enables the ^ as power operator (left associative).

3 Examples

```
\thebnumexpr 128637867168*2187917891279\relax
```
```
281449091072838667627872
```

```
\thebnumexpr 30*(21-43*(512-67*(6133-812*2897)))\relax
```
```
-202785405180
```

```
\newcount\cnta \cnta 123
\the\numexpr \cnta*\cnta*\cnta\relax=\thebnumexpr \cnta*\cnta*\cnta\relax
```
```
1860867=1860867
```

```
\newcount\cntb \cntb 188
  \the\numexpr \cnta*\cnta*\cnta/\cntb+\cntb*\cntb*\cntb/\cnta\relax=
   \thebnumexpr \cnta*\cnta*\cnta/\cntb+\cntb*\cntb*\cntb/\cnta\relax
```
```
63920=63920
 \the\numexpr \cnta*\cnta*(\cnta/\cntb)+\cntb*\cntb*(\cntb/\cnta)\relax=
  \thebnumexpr \cnta*\cnta*(\cnta/\cntb)+\cntb*\cntb*(\cntb/\cnta)\relax
```
```
85817=85817
   \the\numexpr 123/188*188\relax, \the\numexpr 123/(188*188)\relax,
   \thebnumexpr 123/188*188\relax, \thebnumexpr 123/(188*188)\relax.
```
```
188, 0, 188, 0.
```

```
\edef\tmp {\bnumexpr 121873197*123-218137917*188\relax}
                \bnethe\tmp\ (\meaning\tmp)
     \thebnumexpr \tmp*(173197129797-\tmp)*(2179171982-\tmp)\relax
```
```
-26019525165 (macro:->!\BNE_usethe \BNE_protect \BNE_unlock \.=-26019525165 )
```
```
-14616858866321120094950781926331O
```

```
\cnta \thebnumexpr 2152966419779999/987654321\relax\space \the\cnta
```
```
2179879
```

```
\thebnumexpr 2179878*987654321-2152966419779999,2179879*987654321-2152966419779999\relax
```
```
-493827161, 493827160 (there was indeed rounding of the exact quotient)
```

An example with the power operator ^ (option allowpower):

```
\thebnumexpr (1^10+2^10+3^10+4^10+5^10+6^10)^3\relax
```
```
363084368099778773753851
```

4 Package bnumexpr implementation

Contents

4.1 Package identification and catcode setup 4
4.2 Package options 4
4.3 Mapping to the arithmetic routines. 5
4.4 Some helper macros and constants from xint 6
4.5 Encapsulation of numbers in pseudo cs names 7
4.6 \bnumexpr, \bnethe, \thebnumexpr, ... 8
4.7 \BNE_getnext 8
4.8 Parsing an integer 9
4.9 \BNE_getop 10
4.10 Until macros for global expression and parenthesized sub-ones 11
4.11 The arithmetic operators. 12
4.12 The minus as prefix operator of variable precedence level 13
4.13 The comma may separate expressions. 13
4.14 Disabling tacit multiplication 14
4.15 Cleanup 14

Comments are sparse. Error handling by the parser is kept to a minimum; if something goes wrong, the offensive token gets discarded, and some undefined control sequence attempts to trigger writing to the log of some sort of informative message. It is recommended to set \errorcontextlines to at least 2 for more meaningful context.

4.1 Package identification and catcode setup

```
\NeedsTeXFormat{LaTeX2e}%
\ProvidesPackage{bnumexpr}[2014/09/21 v1.1 Expressions with big integers (jfB)]%
\edef\BNErestorecatcodes {\catcode`\noexpand\!\the\catcode`\!
                  \catcode`\noexpand\?\the\catcode`\?
                  \catcode`\noexpand\_\the\catcode`\_
                  \catcode`\noexpand\:\the\catcode`\:\relax }%
\catcode`\! 11
\catcode`\? 11
\catcode`\_ 11
\catcode`\: 11
```

4.2 Package options

```
\def\BNE_tmpa {0}%
\DeclareOption {custom}{\def\BNE_tmpa {1}%
    \PackageWarningNoLine{bnumexpr}{^^J
  Option custom: package xint not loaded. Definitions are needed for:^^J
  \protect\bnumexprAdd, \protect\bnumexprSub,
  \protect\bnumexprMul\space and \protect\bnumexprDiv }%
}%
```

```
\DeclareOption {bigintcalc}{\def\BNE_tmpa {2}%
    \PackageWarningNoLine{bnumexpr}{^^J
    Option bigintcalc: the macros from package bigintcalc are used.^^J
    Notice that / is mapped to \protect\bigintcalcDiv\space which does truncated division}%
}%
\DeclareOption {nocsv}{%
    \PackageInfo{bnumexpr}{Comma separated expressions disabled}%
    \AtEndOfPackage{\expandafter\let\csname BNE_precedence_,\endcsname
                                    \undefined }%
}%
\DeclareOption {notacitmul}{%
    \PackageInfo{bnumexpr}{Tacit multiplication disabled}%
    \AtEndOfPackage{\BNE_notacitmultiplication}%
}%
\def\BNE_allowpower {0}%
\DeclareOption {allowpower}{%
    \PackageInfo{bnumexpr}{Power operator ^ authorized}%
    \def\BNE_allowpower {1}%
}%
\ProcessOptions\relax
```

4.3 Mapping to the arithmetic routines.

The \xintiiQuo macro from xint.sty does Euclidean division. Rounded division is available from xintfrac.sty, rather than loading it, we define directly here \bnumexprDiv as a suitable wrapper to the xint.sty division macros, to achieve, not really economically, rounded division.

Current CTAN version of xint (1.09n) has some sub-optimal code for dealing with the signs of the divisor and dividend, this has been improved in development version 1.1, not yet released at time of writing.

In case option bigintcalc is used, notice that / is mapped to the macro \bigintcalcDiv which does truncated division. We did not add the extra code for rounded division in that case.

```
\if2\BNE_tmpa % Toggle to load bigintcalc.sty
\RequirePackage{bigintcalc}%
\let\bnumexprAdd\bigintcalcAdd
\let\bnumexprSub\bigintcalcSub
\let\bnumexprMul\bigintcalcMul
\let\bnumexprDiv\bigintcalcDiv % NOTE: THIS DOES TRUNCATED DIVISION
\if1\BNE_allowpower\let\bnumexprPow\bigintcalcPow\fi
\fi
\if0\BNE_tmpa % Toggle to load xint.sty (and also xinttools.sty)
\RequirePackage{xint}%
\let\bnumexprAdd\xintiiAdd
\let\bnumexprSub\xintiiSub
\let\bnumexprMul\xintiiMul
\if1\BNE_allowpower\let\bnumexprPow\xintiiPow\fi
```

```
\def\bnumexprDiv      {\romannumeral0\bnumexprdiv }%
\def\bnumexprdiv    #1{\expandafter\BNE_div \romannumeral-`0#1\Z }%
\def\BNE_div #1#2\Z #3{\expandafter\BNE_div_a\expandafter #1%
                            \romannumeral-`0#3\Z #2\Z }%
\def\BNE_div_a #1#2% #1 de A, #2 de B.
{%
    \if0#2\xint_dothis\BNE_div_divbyzero\fi
    \if0#1\xint_dothis\BNE_div_aiszero\fi
    \if-#2\xint_dothis{\BNE_div_bneg #1}\fi
          \xint_orthat{\BNE_div_bpos #1#2}%
}%
\def\BNE_div_divbyzero #1\Z #2\Z {\BNE:DivisionByZero\space 0}%
\def\BNE_div_aiszero   #1\Z #2\Z { 0}%
\def\BNE_div_bpos #1%
{%
    \xint_UDsignfork
            #1{\xintiiopp\BNE_div_pos {}}%
             -{\BNE_div_pos #1}%
    \krof
}%
\def\BNE_div_bneg #1%
{%
    \xint_UDsignfork
            #1{\BNE_div_pos {}}%
             -{\xintiiopp\BNE_div_pos #1}%
    \krof
}%
\def\BNE_div_pos #1#2\Z #3\Z{\expandafter\BNE_div_pos_a
                           \romannumeral0\XINT_div_prepare {#2}{#1#30}}%
\def\BNE_div_pos_a #1#2{\xintReverseOrder {#1\BNE_div_pos_b}\Z }%
\def\BNE_div_pos_b #1#2{\xint_gob_til_Z #2\BNE_div_pos_small\Z
                         \BNE_div_pos_c #1#2}%
\def\BNE_div_pos_c #1#2\Z {\ifnum #1>\xint_c_iv
                              \expandafter\BNE_div_pos_up
                       \else   \expandafter\xintreverseorder
                       \fi {#2}}%
\def\BNE_div_pos_up #1{\xintinc {\xintReverseOrder{#1}}}%
\def\BNE_div_pos_small\Z\BNE_div_pos_c #1#2{\ifnum #1>\xint_c_iv\expandafter
                                  \xint_secondoftwo\else\expandafter
                                  \xint_firstoftwo\fi { 0}{ 1}}%
\fi
```

4.4 Some helper macros and constants from xint

These macros from xint should not change, hence overwriting them here should not be cause for alarm. I opted against renaming everything with \BNE_ prefix rather than \xint_. The \xint_dothis/\xint_orthat thing is a new style I have adopted for expandably forking. The least probable branches should be specified first, for better efficiency.

See examples of uses in the present code.

```
\chardef\xint_c_     0
\chardef\xint_c_i    1
\chardef\xint_c_ii   2
% \chardef\xint_c_iii  3
% \chardef\xint_c_iv   4
% \chardef\xint_c_v    5
\chardef\xint_c_vi   6
\chardef\xint_c_vii  7
\chardef\xint_c_viii 8
\chardef\xint_c_ix   9
% \chardef\xint_c_x     10
% \chardef\xint_c_xviii 18
\long\def\xint_gobble_i      #1{}%
\long\def\xint_gobble_iii    #1#2#3{}%
\long\def\xint_firstofone    #1{#1}%
\long\def\xint_firstoftwo    #1#2{#1}%
\long\def\xint_secondoftwo   #1#2{#2}%
\long\def\xint_firstofthree  #1#2#3{#1}%
\long\def\xint_secondofthree #1#2#3{#2}%
\long\def\xint_thirdofthree  #1#2#3{#3}%
\def\xint_gob_til_!   #1!{}% this ! has catcode 11
\def\xint_UDsignfork  #1-#2#3\krof {#2}%
\long\def\xint_afterfi       #1#2\fi {\fi #1}%
\long\def\xint_dothis        #1#2\xint_orthat #3{\fi #1}% new in v1.1
\let\xint_orthat             \xint_firstofone
```

4.5 Encapsulation of numbers in pseudo cs names

We define here a `\BNE_num` to not have to invoke `\xintNum`; hence dependency on `xint.sty` is kept to the actual arithmetic operations. We only need to get rid of leading zeros as plus and minus signs have already been stripped off; generally speaking user input will have no leading zeros thus the macro is designed to go fast when it is not needed... and as everything happens inside a `\csname...\endcsname`, we can leave some trailing `\fi`'s.

Note: the 1.09n `\xintiiexpr` currently on CTAN has a bug related to leading zeros, `\xinttheiiexpr 001+1\relax` does not return 2. This bug is absent from `\xintexpr`, `\xintfloatexpr`, `\xintiexpr` and only present in `\xintiiexpr`.

```
\edef\BNE_lock #1!{\noexpand\expandafter\space\noexpand
                           \csname .=\noexpand\BNE_num #1\endcsname }%
\def\BNE_num #1{\if #10\expandafter\BNE_num\else
                \ifcat #1\relax 0\expandafter\expandafter\expandafter #1\else
                #1\fi\fi }%
\def\BNE_unlock   {\expandafter\BNE_unlock_a\string }%
\def\BNE_unlock_a #1.={}%
```

4.6 \bnumexpr, \bnethe, \thebnumexpr, ...

In the full \xintexpr, the final unlocking may involve post-treatment of the comma separated values, hence there are _print macros to handle the possibly comma separated values. Here we may just identify _print with _unlock.

```
\def\bnumexpr {\romannumeral0\bnumeval }%
\def\bnumeval {\expandafter\BNE_wrap\romannumeral0\BNE_eval }%
\def\BNE_eval {\expandafter\BNE_until_end_a\romannumeral-`0\BNE_getnext }%
\def\BNE_wrap { !\BNE_usethe\BNE_protect\BNE_unlock }%
\protected\def\BNE_usethe\BNE_protect {\BNE:missing_bnethe!}%
\def\BNE_protect {\noexpand\BNE_protect\noexpand }%
\let\BNE_done\space
\def\thebnumexpr
              {\romannumeral-`0\expandafter\BNE_unlock\romannumeral0\BNE_eval }%
\def\bnethe #1{\romannumeral-`0\expandafter\xint_gobble_iii\romannumeral-`0#1}%
```

4.7 \BNE_getnext

The getnext scans forward to find a number: after expansion of what comes next, an opening parenthesis signals a parenthesized sub-expression, a ! with catcode 11 signals there was there a sub \bnumexpr...\relax (now evaluated), a minus sign is treated as a prefix operator inheriting its precedence level from the previous operator, a plus sign is swallowed, a \count or \dimen will get fetched to \number (in case of a count variable, this provides a full locked number but \count0 1 for example is like 1231 if \count0's value is 123); a digit triggers the number scanner. After the digit scanner finishes the integer is trimmed of leading zeros and locked as a single token into a \csname .=...\endcsname. The flow then proceeds with \BNE_getop which looks for the next operator or possibly the end of the expression. Note: \bnumexpr\relax is illegal.

```
\def\BNE_getnext #1%
{%
    \expandafter\BNE_getnext_a\romannumeral-`0#1%
}%
\def\BNE_getnext_a #1%
{%
    \xint_gob_til_! #1\BNE_gn_foundexpr !% this ! has catcode 11
    \ifcat\relax#1% \count or \numexpr etc... token or count, dimen, skip cs
       \expandafter\BNE_gn_countetc
    \else
       \expandafter\expandafter\expandafter\BNE_gn_fork\expandafter\string
    \fi
    #1%
}%
\def\BNE_gn_foundexpr !#1\fi !{\expandafter\BNE_getop\xint_gobble_iii }%
\def\BNE_gn_countetc #1%
{%
    \ifx\count#1\else\ifx#1\dimen\else\ifx#1\numexpr\else\ifx#1\dimexpr\else
    \ifx\skip#1\else\ifx\glueexpr#1\else\ifx\fontdimen#1\else
```

```
      \BNE_gn_unpackvar
    \fi\fi\fi\fi\fi\fi\fi
    \expandafter\BNE_getnext\number #1%
}%
\def\BNE_gn_unpackvar\fi\fi\fi\fi\fi\fi\fi\expandafter
                            \BNE_getnext\number #1%
{%
  \fi\fi\fi\fi\fi\fi\fi
  \expandafter\BNE_getop\csname .=\number#1\endcsname
}%
```

This is quite simplified here compared to \xintexpr, for various reasons: we have dropped the \xintNewExpr thing, and we can treat the (directly as we don't have to get back to check if we are in an \xintexpr, \xintfloatexpr, etc..

```
\def\BNE_gn_fork #1{%
    \if#1+\xint_dothis \BNE_getnext\fi
    \if#1-\xint_dothis -\fi
    \if#1(\xint_dothis \BNE_oparen \fi
    \xint_orthat       {\BNE_scan_number #1}%
}%
```

4.8 Parsing an integer

We gather a string of digits, plus and minus prefixes have already been swallowed. There might be some leading string of zeros which will have to be removed. In the full \xint expr the situation is more involved as it has to recognize and accept decimal numbers, numbers in scientific notation, also hexadecimal numbers, function names, etc... and variable names in current development version 1.1 (not yet finished).

```
\def\BNE_scan_number #1% this #1 has necessarily here catcode 12
{%
    \ifnum \xint_c_ix<1#1 \expandafter \BNE_scan_nbr\else
                          \expandafter \BNE_notadigit\fi #1%
}%
\def\BNE_notadigit #1{\BNE:not_a_digit! \xint_gobble_i {#1}}%
```

Scanning for a number. Once gathered, lock it and do _getop. If we hit against some catcode eleven !, this means there was a sub \bnumexpr..\relax. We then apply tacit multiplication.

```
\def\BNE_scan_nbr
{%
    \expandafter\BNE_getop\romannumeral-‘0\expandafter
    \BNE_lock\romannumeral-‘0\BNE_scan_nbr_c
}%
\def\BNE_scan_nbr_a #1%
{% careful that ! has catcode letter here
    \ifcat \relax #1\xint_dothis{!#1}\fi % stops the scan
    \ifx         !#1\xint_dothis{!*!}\fi % tacit multiplication before subexpr
    \xint_orthat {\expandafter\BNE_scan_nbr_b\string #1}%
```

```
}%
\def\BNE_scan_nbr_b #1% #1 with catcode 12
{%
    \ifnum \xint_c_ix<1#1 \expandafter\BNE_scan_nbr_c
    \else\expandafter !\fi #1%
}%
\def\BNE_scan_nbr_c #1#2%
{%
    \expandafter #1\romannumeral-‘0\expandafter
                    \BNE_scan_nbr_a\romannumeral-‘0#2%
}%
```

4.9 \BNE_getop

This finds the next infix operator or closing parenthesis or expression end. It then leaves in the token flow <precedence> <operator> <locked number>. The <precedence> stops expansion and ultimately gives back control to a \BNE_until_<op> command. The code here is derived from more involved context where the actual macro associated to the operator may vary, depending if we are in \xintexpr, \xintfloatexpr or \xintiiexpr. Here things are simpler but I have kept the general scheme, thus the actual macro to be used for the <operator> is not decided immediately (development version of xintexpr.sty has extra things to allow multi-characters operators like &&).

```
\def\BNE_getop #1#2% this #1 is the current locked computed value
{%
    \expandafter\BNE_getop_a\expandafter #1\romannumeral-‘0#2%
}%
\catcode‘* 11
\def\BNE_getop_a #1#2%
{%  if a control sequence is found, must be \relax, or possibly register or
 %  variable if tacit multiplication is allowed
    \ifx \relax #2\xint_dothis\xint_firstofthree\fi
    % tacit multiplications:
    \ifcat \relax #2\xint_dothis\xint_secondofthree\fi
    \if    (#2\xint_dothis      \xint_secondofthree\fi
    \ifx   !#2\xint_dothis      \xint_secondofthree\fi
    \xint_orthat \xint_thirdofthree
    {\BNE_foundend #1}%
    {\BNE_precedence_* *#1#2}% tacit multiplication
    {\BNE_foundop #2#1}%
}%
\catcode‘* 12
\def\BNE_foundend {\xint_c_ \relax }% \relax is only a place-holder here.
\def\BNE_foundop #1%
{%
    \ifcsname BNE_precedence_#1\endcsname
        \csname BNE_precedence_#1\expandafter\endcsname
        \expandafter #1%
```

```
    \else
        \BNE_notanoperator {#1}\expandafter\BNE_getop
    \fi
}%
\def\BNE_notanoperator #1{\BNE:not_an_operator! \xint_gobble_i {#1}}%
```

4.10 Until macros for global expression and parenthesized sub-ones

The minus sign as prefix is treated here.

```
\catcode`) 11
\def\BNE_tmpa #1{%  #1=\BNE_op_-vi token
    \def\BNE_until_end_a ##1%
    {%
        \xint_UDsignfork
            ##1{\expandafter\BNE_until_end_a\romannumeral-`0#1}%
              -{\BNE_until_end_b ##1}%
        \krof
    }%
}\expandafter\BNE_tmpa\csname BNE_op_-vi\endcsname
\def\BNE_until_end_b #1#2%
    {%
        \ifcase #1\expandafter\BNE_done
        \or
        \xint_afterfi{\BNE:extra_)_?\expandafter
                      \BNE_until_end_a\romannumeral-`0\BNE_getop }%
        \else
        \xint_afterfi{\expandafter\BNE_until_end_a
                      \romannumeral-`0\csname BNE_op_#2\endcsname }%
        \fi
    }%
\catcode`( 11
\def\BNE_op_( {\expandafter\BNE_until_)_a\romannumeral-`0\BNE_getnext }%
\let\BNE_oparen\BNE_op_(
\catcode`( 12
\def\BNE_tmpa #1{% #1=\BNE_op_-vi
    \def\BNE_until_)_a ##1{\xint_UDsignfork
                           ##1{\expandafter \BNE_until_)_a\romannumeral-`0#1}%
                             -{\BNE_until_)_b ##1}%
                      \krof }%
}\expandafter\BNE_tmpa\csname BNE_op_-vi\endcsname
\def \BNE_until_)_b #1#2%
    {%
     \ifcase  #1\expandafter    \BNE_missing_)_? % missing ) ?
                \or\expandafter \BNE_getop       % found closing )
                \else \xint_afterfi
      {\expandafter \BNE_until_)_a\romannumeral-`0\csname BNE_op_#2\endcsname }%
        \fi
    }%
\def\BNE_missing_)_? {\BNE:missing_)_inserted \xint_c_ \BNE_done }%
```

```
\let\BNE_precedence_) \xint_c_i
\let\BNE_op_)   \BNE_getop
\catcode`) 12
```

4.11 The arithmetic operators.

This is where the infix operators are mapped to actual macros. These macros must ``f-expand'' their arguments, and know how to handle then big integers having no leading zeros and at most a minus sign.

```
\def\BNE_tmpc #1#2#3#4#5#6#7%
{%
  \def #1##1% \BNE_op_<op>
  {% keep value, get next number and operator, then do until
    \expandafter #2\expandafter ##1\romannumeral-`0\expandafter\BNE_getnext }%
  \def #2##1##2% \BNE_until_<op>_a
  {\xint_UDsignfork
    ##2{\expandafter #2\expandafter ##1\romannumeral-`0#4}%
      -{#3##1##2}%
   \krof }%
  \def #3##1##2##3##4% \BNE_until_<op>_b
  {% either execute next operation now, or first do next (possibly unary)
    \ifnum ##2>#5%
    \xint_afterfi {\expandafter #2\expandafter ##1\romannumeral-`0%
      \csname BNE_op_##3\endcsname {##4}}%
    \else \xint_afterfi {\expandafter ##2\expandafter ##3%
      \csname .=#6{\BNE_unlock ##1}{\BNE_unlock ##4}\endcsname }%
    \fi }%
  \let #7#5%
}%
\def\BNE_tmpb #1#2#3%
{%
  \expandafter\BNE_tmpc
  \csname BNE_op_#1\expandafter\endcsname
  \csname BNE_until_#1_a\expandafter\endcsname
  \csname BNE_until_#1_b\expandafter\endcsname
  \csname BNE_op_-#2\expandafter\endcsname
  \csname xint_c_#2\expandafter\endcsname
  \csname #3\expandafter\endcsname
  \csname BNE_precedence_#1\endcsname
}%
\BNE_tmpb  +{vi}{bnumexprAdd}%
\BNE_tmpb  -{vi}{bnumexprSub}%
\BNE_tmpb  *{vii}{bnumexprMul}%
\BNE_tmpb  /{vii}{bnumexprDiv}%
\if1\BNE_allowpower\BNE_tmpb ^{viii}{bnumexprPow}\fi
```

4.12 The minus as prefix operator of variable precedence level

We only need here two levels of precedence, vi and vii. If the power ^ operation is authorized, then one further level viii is needed.

```
\def\BNE_tmpa #1% #1=vi or vii
{%
\expandafter\BNE_tmpb
    \csname BNE_op_-#1\expandafter\endcsname
    \csname BNE_until_-#1_a\expandafter\endcsname
    \csname BNE_until_-#1_b\expandafter\endcsname
    \csname xint_c_#1\endcsname
}%
\def\BNE_tmpb #1#2#3#4%
{%
    \def #1% \BNE_op_-<level>
    {%  get next number+operator then switch to _until macro
        \expandafter #2\romannumeral-`0\BNE_getnext
    }%
    \def #2##1% \BNE_until_-<level>_a
    {\xint_UDsignfork
        ##1{\expandafter #2\romannumeral-`0#1}%
          -{#3##1}%
     \krof }%
    \def #3##1##2##3% \BNE_until_-<level>_b
    {%
        \ifnum ##1>#4%
         \xint_afterfi {\expandafter #2\romannumeral-`0%
                        \csname BNE_op_##2\endcsname {##3}}%
        \else
         \xint_afterfi {\expandafter ##1\expandafter ##2%
                        \csname .=\expandafter\BNE_Opp
                                  \romannumeral-`0\BNE_unlock ##3\endcsname }%
        \fi
    }%
}%
\BNE_tmpa {vi}%
\BNE_tmpa {vii}%
\if1\BNE_allowpower\BNE_tmpa {viii}\fi
\def\BNE_Opp #1{\if-#1\else\if0#10\else-#1\fi\fi }%
```

4.13 The comma may separate expressions.

It suffices to treat the comma as a binary operator of precedence ii. We insert a space after the comma. The current code in \xintexpr does not do it at this stage, but only later during the final unlocking, as there is anyhow need for some processing for final formatting and was considered to be as well the opportunity to insert the space. Here, let's do it immediately. These spaces are not an issue when \bnumexpr is identified as a sub-expression in \xintexpr, for example in: \xinttheiiexpr lcm(\bnumexpr 175-12,1

23+34,56*31\relax)\relax (this example requires package xintgcd).

```
\catcode`, 11
\def\BNE_op_, #1%
{%
    \expandafter \BNE_until_,_a\expandafter #1\romannumeral-`0\BNE_getnext
}%
\def\BNE_tmpa #1{% #1 = \BNE_op_-vi
  \def\BNE_until_,_a ##1##2%
  {%
    \xint_UDsignfork
        ##2{\expandafter \BNE_until_,_a\expandafter ##1\romannumeral-`0#1}%
          -{\BNE_until_,_b ##1##2}%
     \krof }%
}\expandafter\BNE_tmpa\csname BNE_op_-vi\endcsname
\def\BNE_until_,_b #1#2#3#4%
{%
    \ifnum #2>\xint_c_ii
        \xint_afterfi {\expandafter \BNE_until_,_a
                   \expandafter #1\romannumeral-`0%
                   \csname BNE_op_#3\endcsname {#4}}%
    \else
        \xint_afterfi {\expandafter #2\expandafter #3%
                        \csname .=\BNE_unlock #1, \BNE_unlock #4\endcsname }%
    \fi
}%
\let \BNE_precedence_, \xint_c_ii
\catcode`, 12
```

4.14 Disabling tacit multiplication

```
\def\BNE_notacitmultiplication{%
  \def\BNE_getop_a ##1##2{%
    \ifx \relax ##2\expandafter\xint_firstoftwo\else
                   \expandafter\xint_secondoftwo\fi
    {\BNE_foundend ##1}%
    {\BNE_foundop  ##2##1}%
  }%
  \def\BNE_scan_nbr_a ##1{%
    \ifcat \relax ##1\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {!##1}{\expandafter\BNE_scan_nbr_b\string ##1}%
  }%
}%
```

4.15 Cleanup

```
\let\BNE_tmpa\relax \let\BNE_tmpb\relax \let\BNE_tmpc\relax \let\BNE_allowpower\relax
\BNErestorecatcodes
```