The bnumexpr package

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

Package version: 1.7b (2025/09/27)
From source file bnumexpr.dtx of 27-09-2025 at 18:04:58 CEST

1 \bnumeval

This LaTeX package bnumexpr provides \bnumeval, which is an expandable parser of numerical expressions with big integers.

Recent LaTeX has \inteval, which is a slim wrapper for ε-TeX's \numexpr (embedded for twenty years in most TeX-engines except original Knuth's tex).

TeX-nical note: More precisely \inteval{⟨*expression*⟩} is equivalent (up to how TeX handles spaces located after in the source during tokenization, as tokenization of control sequences such as \relax causes TeX to ignore space characters or end-of-line space after it) to:

\the\numexpr⟨*expression*⟩\relax

In an analogous way \bnumeval{⟨*expression*⟩} has equivalent forms:

\bnethe\bnumexpr⟨*expression*⟩\relax
\thebnumexpr⟨*expression*⟩\relax

For contexts where the alternative forms may be useful, refer to the section 6. Everyday use needs only \bnumeval.

Here are the extra features from \bnumeval compared to \inteval:

(a) It allows arbitrarily big integers, whereas \inteval is limited to a maximal input equal to 2147483647 (2^{31} - 1, or hexadecimal 7FFFFFFF).
(b) It recognizes ** and ^ as infix operator for powers,
(c) It recognizes ! as postfix operator for the factorials,
(d) The new operator // computes floored division with /: being the operator for the associated remainder (the operator / computes rounded division),
(e) In addition to the TEX prefixes ' and " for octal and hexadecimal, it recognizes 0b, 0o and 0x for binary, octal, and hexadecimal,
(f) Letters in lowercase can be used for hexadecimal input,
(g) The space character is ignored[1] and can thus be used to separate in the source blocks of digits for better readability of long numbers,
(h) Also the underscore _ may be used as visual digit separator,
(i) Braced material {...} encountered in the expression is automatically unbraced,
(j) Comma separated expressions are allowed,
(k) Some idiosyncrasies of \numexpr such as \inteval{-(1)} causing an error are avoided,
(l) Syntax is fully customizable and extensible.

Note the following about octal and binary inputs:

> When parsing an octal number, if digits 8 or 9 are encountered, or when parsing binary any digit larger than 1, this does not cause an error but ends the parsing and inserts a tacit multiplication: \bnumeval{'118+3} computes $9 \times 8 + 3$, hence obtains 75.

Also note the following on precedences:

> As is explained with complete details in subsubsection 6.6.3, the multiplication operator, as well as the division operators (including modulo) are all at the same precedence level and behave in a left-associative way. Example: \bnumeval{10*10//3, 10*10/:3} evaluates to 33, 1, not to 30, 10.
>
> Also note that powers (which obviouly have a higher precedence) work in a right-associative way so that \bnumeval{2^3^4} will compute 2^{81}, not 8^4.
>
> The best among our users know how to set operator precedences to their own liking, as they have read all the fine print in subsection 6.6.

Furthermore, \bnumeval recognizes an optional argument [b], [o], [h], or [ha] which says to have the calculation result (or comma separated results) be converted to respectively binary, octal, hexadecimal (uppercase) or lowercase hexadecimal digits.

[1] It is not completely ignored, \count 37<space> will automatically be prefixed by \number and the space token delimits the integer indexing the count register. Also, devious inputs using nested braces around spaces may create unexpected internal situations and even break the parser.

2 Dependencies

bnumexpr is a LaTeX package but it can also be used with Plain TeX, thanks to miniltx. Use for this \input miniltx.tex and then \input bnumexpr.sty. Do not use \input but only \usepackage to load the package with LaTeX.

Addition, subtraction, multiplication, division(s), modulo operator, powers, and factorials are all by default executed by macros provided by the xintcore package.

Conversions between decimal, binary, octal and hexadecimal bases are done using the macros from the xintbinhex package.

\bnumeval is a scaled-down variant of \xintiieval from package xintexpr, lacking support for nested structures, functions, variables, booleans, sequence generators, etc... . The xintexpr package is NOT loaded, only as said previously xintcore and xintbinhex.

TeX-nical note: Power users can use \bnumsetup to configure usage of alternative support macros of their own choosing. Options can disable the loading of xintcore and/or xintbinhex. But xintkernel is always loaded. See section 6. Expert users can even add new operators to the syntax, even change the built-in ones or their precedences. See subsection 6.6.

3 Examples

Some of these examples use the ancient syntax \bnethe\bnumexpr...\relax from the initial release (in 2014). The \bnethe prefix converts from some private format (using braces and other things). Some examples do not even have the \bnethe prefix to \bnumexpr because it is allowed in typesetting context to omit it (but in an \edef without it expansion gives the private format). For details refer to section 6 on advanced topics.

Some further examples found in this documentation use the other ancient syntax \thebnumexpr...\relax where \thebnumexpr is equivalent to \bnethe\bnumexpr.

The recommended interface is \bnumeval, as it has optional arguments to cause conversion to hexadecimal, octal or binary. They have no equivalent with \bnethe\bnumexpr or with \thebnumexpr.

Some inputs are weird (such as the first one with three minus signs) because they served originally to check the syntax.

```
\bnumexpr --- 1 208 637 867 * (2 187 917 891 - 3 109 197 072)\relax
                          1113492904235346927

\bnumexpr  ( 13_8089_1090 - 300_1890_2902 ) * ( 1083_1908_3901 - 109_8290
_3890 )\relax
                        -2787514672889976289932

\bnumeval{92_874_927_979 ** 5 - 31_9792_7979 ** 6}
      -10626668124783321156827211633764865014936666010828716422222

\bnethe \bnumexpr 10!, 20!, 30!\relax
     3628800, 2432902008176640000, 265252859812191058636308480000000
```

Testing tacit multiplication elevated precedence:

```
\bnumeval{30!/(21*22*23*24*25)(26*27*28*29*30), 20!}
                2432902008176640000, 2432902008176640000

\bnumeval{13^50//12^50, 13^50/:12^50}
       54, 650556287901099025745221048683760161794567947140168553

\bnumeval{13^50/12^50, 12^50}
       55, 9100438150002149773327585275342566324927152603256586 24

\bnumeval{(1^10+2^10+3^10+4^10+5^10+6^10+7^10+8^10+9^10)^3}
                    1186850754626989817006208281 25

\bnumeval{100! /: 10^50}
          20827223758251185210916864000000000000000000000000
```

Let's check hexadecimal input:

```
\bnumeval{"0010 * "0100 * 0x1000 * 0xA0000, 16^(1+2+3+4)*10}
                    10995116277760, 10995116277760

\bnumeval{"000_abc_def, 0xABCDEF, "abcdef + "543210 + 1, 0x10^6}
                  11259375, 11259375, 16777216, 16777216
```

And also hexadecimal output:

```
\bnumeval[h]{"_7f_fff_fff+1, 0x_0400^3, "aBcDeF*"0000fedcba, 1234}
                  80000000, 40000000, AB0A74EF03A6, 4D2
```

And also in lowercase

```
\bnumeval[ha]{"abcdef, "ABCDEF, "999_999_999, 16^10-1, 167772160}
              abcdef, abcdef, 999999999, fffffffff, a000000
```

Let's make a few checks of octal and binary:

```
\bnumeval[o]{'75316420 * 0o44445555}
                           4305576055707720

\bnumeval[b]{'75316420 * 0o44445555}
            1000110001011011111100001011011110001111110100 00

\bnumeval[b]{0xFFFF, 0o77, 0b1000001^3}
              1111111111111111, 111111, 1000011000011000001
```

We end with some strange non-recommended things to check details of how the parser expands the input:

```
\bnumeval{"0000\bnumeval [h]{00000012345678}FFFF, 000012345679*16**4-1}
                      809086418943, 809086418943

\bnumeval[o]{0b000\bnumeval [b]{'123456}, 0x\bnumeval [h]{0o00000123456}}
                            123456, 123456
```

4 Customizing how output is “printed out”

4.1 Printing big numbers

TEX and LATEX will not split long numbers at the end of lines. I personally often use helper macros (not in the package) of the following type:

```
\def\allowsplits #1{\ifx #1\relax \else #1\hskip 0pt plus 1pt\relax
                    \expandafter\allowsplits\fi}%
\def\printnumber #1{\expandafter\allowsplits \romannumeral-`0#1\relax }%
```

Here is an example of use and its output:

```
\noindent|\bnumeval{1000!} =|
\textcolor{digitscolor}{\printnumber{\bnumeval{1000!}}}
```

\bnumeval{1000!} = 402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261683151027341827977704784635868170164365024153691398281264810213092761244896359928705114964975419909342221566832572080821333186116811553615836546984046708975602900950537616475847728421889679646244945160765353408198901385442487984959953319101723355556602139450399736280750137837615307127761926849034352625200015888535147331611702103968175921510907788019393178114194545257223865541461062892187960223838971476088506276862967146674697562911234082439208160153780889893964518263243671616762179168909779911903754031274622289988005195444414282012187361745992642956581746628302955570299024324153181617210465832036786906117260158783520751516284225540265170483304226143974286933061690897968482590125458327168226458066526769958652682272807075781391858178889652208164348344825993266043367660176999612831860788386150279465955131155655203609398818061213855860030143569452722420634463179746059468257310379008402443243846565724501440282188525247093519062092902313649327349756551395872055965422874977401141334696271542284586237738753823048386568897646192738381490014076731044664025989949022222176590433990188601856652648506179970235619389701786004081188972991831102117122984590164192106888438712185564612496079872290851929681937238864261483965738229112312502418664935314397013742853192664987533721894069428143411852015801412334482801505139969429015348307764456909907315243327828826986460278986432113908350621709500259738986355427719674282224875758676575234422020757363056949882508796892816275384886339690995982628095612145099487170124451646126037902930912088908694202851064018215439945715680594187274899809425474217358240106367740459574178516082923013535808184009699637252423056085590370062427124341690900415369010593398383577793941097002775347200000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

TEX-nical note: \bnumeval is *f*-expandable, so the \romannumeral-`0 as used here in \printnumber causes its full expansion (even if for example the output contains multiple values, separated by commas). So then \printnumber's auxiliary can simply loop over the tokens.

TEX-nical note: Note that inside math mode, the inserted \hskip's have no effect. There should be some \allowbreak's. By the way, we allow some stretch so that line endings match the actual linewidth.

4.2 \bnumprintone, \bnumprintonesep

The output values are each fetched to \bnumprintone and separated by \bnumprintonesep.

Here are the default definitions (or rather some quasi equivalents in LATEX's lingua):

```
\newcommand{\bnumprintone}[1]{#1}
\newcommand{\bnumprintonesep}{, }
```

In other terms \bnumprintone produces its argument ``as is'', and multiple values get separated by a comma and a space.

Let's say you want the output to be boxed. Doing \fbox{\bnumeval{...}} will make one single frame even in case of multiple values. Redefining \bnumprintone is the way to go:

```
\RenewDocumentCommand{\bnumprintone}{m}{\fbox{#1}}
\bnumeval{2^10, 3^10, 5^10, 7^10}
```

1024, 59049, 9765625, 282475249

It is important to have used \RenewDocumentCommand and not \renewcommand here, because \bnumprintone and \bnumprintonesep have to be compatible with expansion only context.

TEX-nical note: That means that \bnumprintone in an \edef should not give rise to any \newcommand, lower level \def, count or dimen assignments, etc....

This constraint is due to the fact that \bnumeval wraps the final print-out inside of \expanded, for TEXnical reasons.

The simplest way for \bnumprintone (considering that its argument will already have been fully expanded to digit tokens) and \bnumprintonesep to be ``safe'' is that they do not expand at all in \edef. This is the case if they are defined using \RenewDocumentCommand. With an older LATEX, or Plain ε-TEX (but having some \fbox at our disposal), we would have used here \protected\def\bnumprintone#1{\fbox{#1}}.

A more common use case will be to have the outputs be typeset according to the conventions of the document language. This is easily done redefining \bnumprintone in terms of (for example) the \np macro of package numprint.

```
\RenewDocumentCommand{\bnumprintone}{m}{\np{#1}}
\renewcommand{\bnumprintonesep}{ --- }
\bnumeval{2^10, 3^10, 5^10, 7^10}
```

```
1,024 --- 59,049 --- 9,765,625 --- 282,475,249
```

TEX-nical note: Usage of \RenewDocumentCommand for \bnumprintonesep was not needed here, obviously its expansion could cause no trouble.

Let's give another use case. Assume you are computing in one go multiple large values, too large to fit on a line. The simple-minded \printnumber of the previous section will (due to some TEXnicality) swallow the spaces injected by \bnumprintonesep. To fix this, the simplest is to redefine \bnumprintone to execute \printnumber:

```
\renewcommand{\bnumprintone}[1]{\printnumber{#1}}
\bnumeval{2^100, 3^100, 5^100, 7^100}
```

```
1267650600228229401496703205376, 515377520732011331036461129765621272702
107522001, 788860905221011805411728565282786229673206435109023004770278 9
306640625
```

TEX-nical note: Our \printnumber belongs to this family of macros causing no damage if expanding in an \edef. So, it was not needed to use \RenewDocumentCommand.

4.3 \bnumprintonehex, \bnumprintonelowerhex, \bnumprintoneoct, \bnumprintonebin

When \bnumeval is exerted with [h], [ha], [o] or [b] it does not use \bnumprintone but one of \bnumprintonehex, \bnumprintonelowerhex, \bnumprintoneoct or \bnumprintonebin. The same \bnumprintonesep is used as with decimal numbers.

The default definitions are as for \bnumprintone to ``print as is''.

To give an example of a custom definition, one may want hexadecimal to use the 0x prefix with uppercase output or the " prefix with lowercase output. This is very easy:

```
\renewcommand{\bnumprintonehex}[1]{0x#1}
\renewcommand{\bnumprintonelowerhex}[1]{"#1}
```

```
\bnumeval[h]{7^30, 13^20, 20!}
   0x12A4E415E1E1B36FF883D1, 0x40642DAC4A3F8EEB7D1, 0x21C3677C82B40000

\bnumeval[ha]{7^30, 13^20, 20!}
    "12a4e415e1e1b36ff883d1, "40642dac4a3f8eeb7d1, "21c3677c82b40000
```

TEX-nical note: It was unneeded to use \RenewDocumentCommand here because prefixing with 0x is obviously compatible with expansion-only context.

5 Babel-active characters are not a problem!

Some languages use active characters with PDFLATEX. For example the babel-french module turns the colon : and the exclamation mark ! into active characters (whose expansions would cause \bnumeval to crash). It used to be necessary to

take preventive measures such as either turning the activation off altogether or use in the input /\string: and \string! as clumsy replacements of /: and !.

Those troubled times are gone! With release 1.6 they will work fine as is in \bnumeval. The same applies to all other characters if babel-active. There are miracles sometimes!

Warning: characters made active otherwise still need the \string or other workaround to be usable as operators in the syntax.

6 Fine print (not needed to read this for regular use)

6.1 Adding support for binomial coefficients

As will be documented in the section for expert users, it is possible to extend the syntax with one's own operators.

Let's turn the semicolon into an operator which computes binomial coefficients: a;b will evaluate to ``a choose b''. The precedence will be chosen stronger than addition and multiplication but less than powers. This only needs adding those two lines to the preamble:

```
\usepackage{xint}% as xintcore does not have \xintiiBinomial
\bnumdefinfix{;}{\xintiiBinomial}{15}{15}
```

We can now test it :

```
\bnumeval{100;50}
```

100891344545564193334812497256

```
\bnumeval{3^5;2^5}
```

9812294412288780842726471233974791140221

6.2 An expandable macro for computing large Fibonacci numbers

We are going to define an expandable macro which takes on input a positive integer N (we did not bother about case of zero or negative input) and computes the Nth Fibonacci number F_N.

The algorithm to compute F_N is based upon the quadratic relations which allow to go from (F_n, F_{n-1}) to either (F_{2n}, F_{2n-1}) (operation of type A) or to (F_{2n+1}, F_{2n}) (operation of type B). We first convert N into binary base, and read the digits from most significant to least significant applying each time either type A or type B. Each one of type A or B will cost only 3 multiplications and a bunch of additions, which of course are done using \bnumexpr. In total the algorithm costs (because $3\log_2 10 \approx 10$) about 10L multiplications (doubling is not counted as a multiplication) where L is the number of decimal digits of N. Here is the code:

```
\makeatletter
\newcommand\Fibonacci[1]{%
    \expandafter\Fibo@start
    \romannumeral0\xintdectobin{\the\numexpr#1\relax};%
}
% To chain \bnumexpr's easily we use the once-expanded form which is
% \romannumeral0\bnumexpro.  We could also have used \expanded and \bnumeval
% calls, that would have been simpler but old habits die hard.
%
% Gobble the first digit which is 1 (this is where we could have tested for a
% negative or zero input).  Prepare encapsulated 1 and 0.  Then repeatedly
% dispatch according to binary digits (which are strategically located at end
```

```
% of input stream to optimize), or the semicolon which signals completion.
%
% We use \csname-governed branching (because it is cool).  We could have
% used some \if test or even an \ifcase with an {-1} at end, or any other
% of the many possibilities opened up by TeX syntax.
\def\Fibo@start 1#1{%
    \csname Fibo@#1\expandafter\endcsname
        \romannumeral0\bnumexpro 1\expandafter\relax\expandafter;%
        \romannumeral0\bnumexpro 0\relax;%
}
\@namedef{Fibo@0}#1;#2;#3{%
    \csname Fibo@#3\expandafter\endcsname
       \romannumeral0\bnumexpro (#1+2*#2)*#1\expandafter\relax\expandafter;%
       \romannumeral0\bnumexpro #1*#1+#2*#2\relax;%
}
\@namedef{Fibo@1}#1;#2;#3{%
    \csname Fibo@#3\expandafter\endcsname
        \romannumeral0\bnumexpro 2*(#1+#2)*#1+#2*#2\expandafter\relax\expandafter;%
        \romannumeral0\bnumexpro (#1+2*#2)*#1\relax;%
}
% #1 is F_N (and #2 is F_{N-1}) but in need of unpacking.
\@namedef{Fibo@;}#1;#2;{\bnethe#1}
\makeatother
```

We can now compute for example F_{10}, F_{100}, F_{1000} and F_{10000}:

```
\noindent\textcolor{cyan}{$F_{10}={}$}%
  \textcolor{digitscolor}{\Fibonacci{10}}\newline
\textcolor{cyan}{$F_{100}={}$}%
  \textcolor{digitscolor}{\Fibonacci{100}}\newline
\textcolor{cyan}{$F_{1000}={}$}%
  \textcolor{digitscolor}{\printnumber{\Fibonacci{1000}}}\newline
\textcolor{cyan}{$F_{10000} ={}$}%
  \textcolor{digitscolor}{\printnumber{\Fibonacci{10000}}}
```

F_{10} = 55
F_{100} = 354224848179261915075
F_{1000} = 43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
F_{10000} = 33644764876431783266621612005107543310302148460680063906564769974680081442166662368155595513633734025582065332680836159373734790483865268263040892463056431887354544369559827491606602099884183933864652731300088830269235673613135117579297437854413752130520504347701602264758318906527890855154366159582987279682987510631200575428783453215515103870818298969791613127856265033195487140214287532698187962046936097879900350962302291026368131493195275630227837628441540360584402572114334961180023091208287046088923962328835461505776583271252546093591128203925285393434620904245248929403901706233888991085841065183173360437470737908552631764325733993712871937587746897479926305837065742830161637408969178426378624212835258112820516370298089332099905707920064367426202389783111470054074998459250360633560933883831923386783056136435351892133279732908133732642652633989763922723407882928177953580570993691049175470808931841056146322338217465637321248226383092103297701648054726243842374862411453093812206564914032751086643394517512161526545361333111314042436854805106765843493523836959653428071768775328348234345557366719731392746273629108210679280784718035329131176778924659089938635459327894523777674406192240337638674004021330343297496902028328145933418826817683893072003634795623117103101291953169794607632

```
737589253530772552375943788434504067715555779056450443016640119462580972216729758615026
968443146952034614932291105970676243268515992834709891284706740862008587135016260312071
903172086094081298321581077282076353186624611278245537208532365305775956430072517744315
051539600905168603220349163222640885248852433158051534849622434848299380905070483482449
327453732624567755879089187190803662058009594743150052402532709746995318770724376825907
419939632265984147498193609285223945039707165443156421328157688908058783183404917434556
270520223564846495196112460268313970975069382648706613264507665074611512677522748621598
642530711298441182622661057163515069260029861704945425047491378115154139941550671256271
197133252763631939606902895650288268608362241082050562430701794976171121233066073310059
947366875
```

TEX-nical note: We did not use math mode because our \printnumber is a bit dumb and does not work in math mode. See the comment at the end of subsection 4.1.

TEX-nical note: In the code above we used ; as delimiter. But the macro definitions are located in the body of this document, not in the preamble, and ; is catcode active here due to French being among the languages used by this LATEX document. This is not a problem in itself for the uses of the semicolon as delimiter for macro definitions. It would be a problem if the semicolon had been used to terminate a \numexpr (we would then have injected a \relax). One could also be very worried seeing \@namedef{Fibo@;} where ; will expand in \csname...\endcsname, but babel has a built-in protection for such things. This all being said, it is safer to have such definitions located in the preamble of the LATEX document. But in next section we make a modification which can only be done after the original \Fibonacci has already been used (as it changes its interface), so putting all things into the preamble is not always logical nor convenient.

6.3 More fun with Fibonacci numbers

Our \Fibonacci{⟨N⟩} macro expands to compute F_N. But what if we want a whole sequence $F_N, F_{N+1}, \ldots$? Then using directly the additive recurrence $F_{n+2} = F_{n+1}+F_n$ will be more efficient than computing again and again via our fancy multiplicative algorithm new values. So it is a bit sad that expansion of \Fibonacci{⟨N⟩} actually computed both of F_N and F_{N-1} but discarded the latter. We will modify its ending to keep also F_{N-1}. And we switch to a down-to-earth style based on assignments. The xint documentation provides a variant construct which obtains expandably a sequence of braced items {F_a}{F_{a+1}}...{F_b}, but for this, expandable loops from xinttools or from LATEX3 are the appropriate tools, and we then need extra tools to handle such output.

```
\makeatletter
% reuse \Fibonacci's earlier meaning, then make \Fibonacci undefined
% because \Fibo@; will be modified.
\let\Fibonacci@\Fibonacci
\let\Fibonacci\undefined
\newcommand\SetToFibonacciPair[3]{\Fibonacci@{#1}{#2}{#3}}
\@namedef{Fibo@;}#1;#2;#3#4{\def#3{#1}\def#4{#2}}
\makeatother
```

We add the above code to our document and now execute

```
\SetToFibonacciPair{1001}{\FibB}{\FibA}
```

Now \FibB holds F_{1001} and \FibA holds F_{1000} (both being \bnumexpr-encapsulated and needing \bnethe to get converted to explicit digits). We can use LaTeX's \loop to organize the iterative computation and printing of new values. But we first abstract the ``printing'' into a LATEX ``command''.

```
\newcommand\PrintOneFib[2]{%
  \noindent\textcolor{cyan}{$F_{#1}={}$}\textcolor{digitscolor}{\printnumber{\bnethe#2}}%
}
```

```
\def\myN{1000}
\loop
  \PrintOneFib{\myN}{\FibA}\newline
  \edef\FibC{\bnumexpr \FibB + \FibA\relax}% compute and assign a new Fibonacci number
  \let\FibA\FibB \let\FibB\FibC % update the storage macros
\ifnum\myN<1010 % terminate the loop when F_{1010} has been printed
  \edef\myN{\the\numexpr\myN+1}%
\repeat
```

F_{1000} = 43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875

F_{1001} = 70330367711422815821835254877183549770181269836358732742604905087154537118196933579742249494562611733487750449241765991088186362654502236471060120533741212738673391111981393731255987676900919022452453234035 01

F_{1002} = 11379692539836027225752378255222417557274593035373051314508663417669109253614598547014612933464186690278367304232208862586339605288869009696957717369637056218040052704949710902305411477139456804004041217263237 6

F_{1003} = 18412729310978308807935903742940772534292720019008924588769153926384562965434291904988837882920447863627142349156385461695158241615414032061668318574974468345426786616069524839617971353908465994228565749603587 7

F_{1004} = 29792421850814336033688281998163190091567313054381975903277817344053672219048890452003450816384634553905509653388594324281497846904283041758626035944611524563466839321019235741923382831047922798232606966866825 3

F_{1005} = 48205151161792644841624185741103962625860033073390900492046971270438235184483182356992288699305082417532652002544979785976656088519697073820294354519585992908893625937088760581541354184956388792461172716470413 0

F_{1006} = 77997573012606980875312467739267152717427346127772876395324788614491907403532072808995739515689716971438161655933574110258153935423980115578920390464197517472360465258107996323464737016004311590693779683337238 3

F_{1007} = 12620272417439962571693665348037111534328737920116377688737175988493014258801525516598802821499479938897081365847855389623481002394367718939921474498378351038125409119519675690500609120096070038315495239980765 13

F_{1008} = 20420029718700660659224912121963826806071472532893665328269654849942204999154732797498376773068451636040897531441212800649296395936765730497813513544798102785361455645330475322847082821696501197384873208314488 96

F_{1009} = 33040302136140623230918577470000938340400210453010043017006830838435219257956258314097179594567931574937978897289068190272777398331133449437734988043176453823468684764850151013347691941792571235700368448295254 09

F_{1010} = 53460331854841283890143489591964765146471682985903708345276485688377424257110991111595556367636383210978876428730280990922073794267899179935548501587974556608848320410180626336194774763489072433085241656609743 05

At this stage we have printed up to F_{1010}. On exit of the loop \FibA holds F_{1011} and \FibB holds F_{1012}. We print them out too:

```
\PrintOneFib{\the\numexpr\myN+1}{\FibA}\newline
\PrintOneFib{\the\numexpr\myN+2}{\FibB}\par
```

F_{1011} = 86500633990981907121062067061965703486871893438913751362283316526812643515067249425692735962204314785916855326019349181194851192599032629373283489631151010432335185175030777349542466705281643668785610104904997 14

F_{1012} = 13996096584582319101120555665393046863334357642481745970755980221519006777217824053728829232984069799689573175474963017211692498686693180930883199121912556704116350558521140368573724146877071610187085176151474019

6.4 The \bnumsetup command

Package bnumexpr needs that some *big integer engine* provides the macros doing the actual computations.

By default, it loads package xintcore (a subset of xintexpr) and package xintbinhex.

```
\usepackage{xintcore}
\usepackage{xintbinhex}
```

It then uses \bnumsetup in the following way (the final comma is optional, and spaces around equal signs also; there can also be spaces before the commas but the author dislikes such style a lot so they are not used here):

```
\bnumsetup{%
  add = \xintiiAdd, sub = \xintiiSub, opp = \xintiiOpp,
  mul = \xintiiMul, pow = \xintiiPow, fac = \xintiiFac,
  div=\xintiiDivFloor, mod=\xintiiMod, divround=\xintiiDivRound,
  hextodec=\xintHexToDec, octtodec=\xintOctToDec, bintodec=\xintBinToDec,
  dectohex=\xintDecToHex, dectooct=\xintDecToOct, dectobin=\xintDecToBin,
}%
```

One can use \bnumsetup to map one, some, or all keys to macros of one's own choosing. Of course it is then up to user to load the suitable packages.

If one has alternatives for all of the above xintcore macros, so that this package is not needed at all, one can pass option customcore to bnumexpr at loading time:

```
\usepackage[customcore]{bnumexpr }
```

This tells to not load xintcore.

Similarly there is an option custombinhex to not load xintbinhex. Make sure then to provide suitable replacements to all base conversion macros!

Option custom means doing both of customcore and custombinhex. Even under this option package xintkernel will always be loaded.

Here are the conditions that the custom macros must obey:

1. They all must be *f*-expandable, with some exceptions
   a) The macro for computing factorials only has to be *x*-expandable.
   b) If \bnumprintonehex (or \bnumprintoneoct, or \bnumprintonebin) is redeinfed as a \protected, then conversion to hexadecimal (resp. octal, resp. binary) only has to be *x*-expandable, except that [ha] option will break if the dectohex macro is not *f*-expandable.

   Note that any *x*-expandable macro can be wrapped into an *f*-expandable one, using \expanded.
2. It is sufficient for them to be able to handle arguments in *raw normalized form*, i.e., sequences of explicit decimal (or hexadecimal for the macro associated with key hextodec) digits, no leading zeros, with at most one minus sign and no plus sign.
3. Their output format is limited only by the fact that it should be acceptable input to all the other operators, as well as to the user optional re-definition of \bnumprintone. If one cares about hexadecimal (et al.) output one must ensure the macros output format is suitable input for those macros actually doing the conversion from decimal to other bases.
4. *Important*: hence if only some macros among those associated to operators (i.e. those by default originating in xintcore), or to conversions into decimal, are custom, their output **must** be produced in *raw normalized form*, as this is the format required by the xintcore macros and by the xintbinhex macros converting from decimal to other bases. However if one does not care about producing output in binary, octal or hexadecimal (as is the case in the next section), and if one has replaced all xintcore macros, the output format can be as one likes.

5. If dectohex key is used and the associated custom macro produces *lowercase* hexadecimal, then \bnumeval[h]{...} produces as instructed hexadecimal in lowercase. Mind that the [ha] option is now broken because its internals expect the input to be in uppercase hexadecimal, and this input is computed using the macro assigned to the dectohex key. This is of course unimportant because [h] now has replaced [ha], but I figured I could mention it here.

6.5 Let's handle fractions!

I will show how to transform \bnumeval into a calculator with fractions! We will use the xintfrac macros, but coerce them into always producing fractions in lowest terms (except for powers). For optimization we use the [0] post-fix which speeds-up the input parsing by the xintfrac macros. We remove it on output via a custom \bnumprintone.

Note that the / operator is associated to divround key but of course here the used macro will simply do an exact division of fractions, not a rounded-to-an integer division. This is the whole point of using a macro of our own choosing!

```
\usepackage{xintfrac}
\newcommand\myIrrAdd[2]{\xintIrr{\xintAdd{#1}{#2}}[0]}
\newcommand\myIrrSub[2]{\xintIrr{\xintSub{#1}{#2}}[0]}
\newcommand\myIrrMul[2]{\xintIrr{\xintMul{#1}{#2}}[0]}
\newcommand\myDiv[2]{\xintIrr{\xintDiv{#1}{#2}}[0]}
\newcommand\myDivFloor[2]{\xintDivFloor{#1}{#2}[0]}
\newcommand\myMod[2]{\xintIrr{\xintMod{#1}{#2}}[0]}
\newcommand\myPow[2]{\xintPow{#1}{#2}}% will have already postfix [0]
\newcommand\myFac[1]{\xintFac{#1}}%     will have already postfix [0]
\makeatletter
\def\myRemovePostFix#1{\@myRemovePostFix#1[0]\relax}%
\def\@myRemovePostFix#1[0]#2\relax{#1}
\makeatother
\let\bnumprintone\myRemovePostFix
\bnumsetup{add=\myIrrAdd, sub=\myIrrSub, mul=\myIrrMul,
           divround=\myDiv, div=\myDivFloor,
           mod=\myMod, pow=\myPow, fac=\myFac}%

\bnumeval{1000000*(1/100+1/2^7-20/5^4)/(1/3-5/7+9/11)^2}

                                    -1514118375/20402

\bnumeval{(1-1/2)(1-1/3)(1-1/4)(1-1/5)(1-1/6)(1-1/7)}

                                           1/7

\bnumeval{(1-1/3+1/9-1/27-1/81+1/243-1/729+1/2187)^5}

                           10485760000000000/50031545098999707

\bnumeval{(1+1/10)^10 /: (1-1/10)^10}

                                   764966897/5000000000

\bnumeval{2^-3^4}
                                1/241785163922925834941235 2
```

Computations with fractions quickly give birth to big results, see subsection 4.1 on how to modify \bnumprintone to coerce TEX into wrapping numbers too long for the available width.

6.6 For the expert user: expression syntax and its customizability

6.6.1 Significant differences between \bnumexpr and \numexpr

Apart from the extension to big integers and the added operators, there are a number of important differences between \bnumexpr and \numexpr:

1. Contrarily to \numexpr, the \bnumexpr parser stops only after having found (and swallowed) a mandatory ending \relax token (it can arise from expansion),
2. In particular note that spaces between digits do not stop \bnumexpr, in contrast with \numexpr:
   \the\numexpr 3 5+79\relax expands (in one step) to 35+79\relax
   \thebnumexpr 3 5+79\relax expands (in two steps) to 114
3. With \edef\myvariable{\bnumexpr 1+2\relax}, the computation is done at time of the \edef. It prepares \myvariable as a self-contained pre-computed unit which is recognized as such when inserted in a bnumexpr expressions. It triggers tacit multiplication: 7\myvariable is like 7*\myvariable. This is different from what would happen if we had used \edef\myvariable{\bnethe\bnumexpr...} which would simply have \myvariable expand to digit tokens so 7\myvariable then constructs a number with 7 as first digit.

   Let's give an example. Note that \edef has the effect of pre-evaluating. With \def the outputs would be the same, but the computations would be delayed to \bnumeval execution.

   \edef\x{\bnumexpr 3^10\relax}% precomputes but keeps private format
   \bnumeval{10000\x }

   590490000

   \edef\y{\bnethe\bnumexpr 3^10\relax}% evaluates to explicit digits
   \bnumeval{10000\y }

   1000059049

   In the example with \x, tacit multiplication applied, whereas in the example with \y it is as if the digits had been input by hand in place of \y. Note that the tacit multiplication behaves as expected relative to powers:
   \bnumeval{10^10\x }

   590490000000000

   And we certainly do no want to try 10^10\y which is like 10^1059049.

   There is no analog with \numexpr:

   a) \edef\foo{\numexpr1+2\relax} will define \foo as \numexpr1+2\relax where the calculation is not yet done.
   b) Inserting the \foo as is in the document text causes an error.
   c) Trying \the\numexpr 7\foo\relax with such a \foo causes an error. One must use the multiplication sign * explicitly.
4. Expressions may be comma separated. On input, spaces are ignored, and on output the values are comma separated with a space after each comma,
5. \thebnumexpr -(1+1)\relax is legal contrarily to \the\numexpr -(1+1)\relax which errors (also with -(1) but -(1+1) has a more legitimate use of parentheses), and one has to employ workarounds such as \the\numexpr 0-(1+1)\relax,
6. \the\numexpr 2+-(1+1)\relax (or with -+) errors (but +- and -+ if not followed by parentheses do work), whereas \bnumexpr has no such counter-intuitive limitation,
7. \the\numexpr 2\cnta\relax is illegal (with \cnta a \count-variable) (note that \the\dimexpr 2\mydimen\relax or \the\dimexpr2\dimen0\relax do work; even \the\dimexpr 1\count0\relax does work, but \the\dimexpr \count0\relax errors!). On the other hand \thebnumexpr 2\cnta\relax is perfectly legal and does the tacit multiplication; also \thebnumexpr 2\mydimen\relax works, with \mydimen getting replaced by its value in scaled points,

8. \the\numexpr3(4+5)\relax stops at the parenthesis, i.e. it expands to 3(4+5)\relax, whereas of course \thebnumexpr 3(4+5)\relax does the computation and expands to 27,
9. \the\numexpr(4+5)3\relax expands to 93\relax whereas \thebnumexpr(4+5)3\relax does the computation and expands to 27,
10. \thebnumexpr (4+5)\count0\relax expands to nine time the current value of \count0 register, but with \the\numexpr there would be a *Missing number* error,
11. Generally speaking \numexpr does not implement the tacit multiplication which \bnumexpr does in front of parentheses and sub \numexpr...\relax expression: or after parentheses, or in-between two parenthesized expressions such as happens with (1+2)(3+4),
12. The underscore _ is accepted within the digits composing a number and is silently ignored by \bnumexpr.

Regarding constructs such as \edef\myvariable{\bnumexpr 1+2\relax}, it was explained \myvariable behaves then in a special way in another \bnumexpr expression (or \bnumeval). It is also worth mentioning that it can be used directly in the typesetting stream. But if written to an external file it will expand to some internal format which is not documented as it may vary in future.

One can NOT use a \myvariable as above in an \ifnum test, even if representing a single small integer. It will work with syntax such as \ifnum\bnethe\myvariable=7 ....

A point of note is that \bnethe\myvariable or \bnethe\bnumexpr...\relax expand to explicit digits so (assuming here there no other comma separated value computed),

```
\ifnum 3>\bnethe\bnumexpr...\relax
...
\fi
```

is dangerous, because the integer is not properly terminated. Here one could reverse the order, but the simplest way is simply to use \bnumeval:

```
\ifnum 3>\bnumeval{...}
...
\fi
```

Now, the end of line space injected by TEX will terminate the integer and make the \ifnum test safe.

6.6.2 Expression syntax

The implemented syntax is the expected one with infix operators and parentheses, the recognized operators being +, -, *, / (rounded division), ^ (power), ** (power), // (by default floored division), /: (the associated modulo) and ! (factorial).

One can input hexadecimal numbers as familiar from the TEX number assignments syntax, i.e. using the " prefix. But also lowercase letters abcdef are accepted in addition to uppercase ABCDEF (feature added at 1.7). Release 1.6 added support for the 0x prefix. It also added support for octal input via either ' or 0o prefixes, and for binary input via 0b prefix.

Commas separating multiple expressions are allowed. The whole expression is handled token by token, any component (digit, operator, parentheses... even the ending \relax) may arise on the spot from macro expansions. The underscore _ can be used to separate digits in long numbers, for readability of the input.

The precedence rules are as expected and detailed in the next section. Operators on the same level of precedence (like *, /, //, /:) behave in a left-associative way, and these examples behave as e.g. with Python analogous operators:
\bnumeval{100//3*4, 100*4//3, 100/:3*4, 100*4/:3, 100//3/:5}

132, 133, 4, 1, 3

At 1.5 a change was made to the power operators which became right-associative. Again, this matches the behaviour e.g. of Python:
```
\bnumeval{2^3^4, 2^(3^4)}
```

$$241785163922925834941235 2, 241785163922925834941235 2$$

It is possible to customize completely the behaviour of the parser, in two ways:

- via \bnumsetup which has a simple interface to replace the macros associated with +, -, *, /, //, /:, **, ^ and ! by custom macros,
- or even more completely via \bnumdefinfix and \bnumdefpostfix which allow to add new operators to the syntax! They can also be used to replace the built-in ones or modify their precedence levels.

6.6.3 Precedences

The parser implements precedence rules based on concepts which are summarized below. I am providing them for users who will use the customizing macros.

- an infix operator has two associated precedence levels, say L and R,
- the parser proceeds from left to right, pausing each time it has found a new number and an operator following it,
- the parser compares the left-precedence L of the new found operator to the right-precedence R_last of the last delayed operation (which already has one argument and would like to know if it can use the new found one): if L is at most equal to it, the delayed operation is now executed, else the new-found operation is kept around to be executed first, once it will have gathered its arguments, of which only one is known at this stage.

Although there is thus internally all the needed room for sophistication, the implemented table of precedences simply puts all of multiplication and division related operations at the same level, which means that left associativity will apply with these operators. I could see that Python behaves the same way for its analogous operators.

Here is the default table of precedences as implemented by the package:

Table of precedences

operator	left	right
+,-	12	12
*,/,//,/:	14	14
tacit *	16	14
**, ^	18	17
!	20	n/a

Tacit multiplication applies in front of parentheses, and after them, also in front of count variables or registers. As shown in the table it has an elevated precedence compared to multiplication explicitly induced by *, so 100/4(9) is computed as 100/36 and not as 25*9:
```
\bnumeval{100/4(9), (100/4)9, 1000 // (100/4) 9 (1+1) * 13}
```

$$3, 225, 26$$

More generally A/B(C)(D)(E)*F will compute (A/(B*C*D*E))*F.[2]

The unary -, as prefix, has a special behaviour: after an infix operator it will acquire a right-precedence which is the minimum of 12 (i.e. the precedence of addition and subtraction) and of the right-precedence of the infix operator. For example 2^-3^4 will be parsed as 2^(-(3^4)), raising an error because the parser is by default integer only, but see the section about \bnumsetup which explains how to let \bnumeval compute fractions!

[2] The B(C)(D)(E) product will be computed as B*(C*(D*E)) because the right-precedence of tacit multiplication is 14 but its left-precedence is 16, creating right associativity. As the underlying mathematical operation is associative this is irrelevant to final result.

6.6.4 \bnumdefinfix

It is possible to define infix binary operators of one's own choosing.[3]

For an example see the subsection 6.1 on adding ; as operator computing binomial coefficients. Other examples will also be given here.

The syntax is

\bnumdefinfix{⟨*operator*⟩}{⟨*\macro*⟩}{⟨*L-prec*⟩}{⟨*R-prec*⟩}

{⟨*operator*⟩} The characters for the operator, they may be letters or non-letters. Digits are not allowed to be first or last in ⟨*operator*⟩. The following characters are not allowed at all: \, {, }, and %. Spaces get removed.

> **Warning:** ⟨*operator*⟩ will be expanded in an \edef and then further processed. If it is (or contains) a Babel-active character chances are that the further processing by \bnumdefinfix will give breakage. There will be no problem if \bnumdefinfix is used in the document preamble. Else, use \string, for example: \bnumdefinfix{\string;}{\xintiiBinomial}{15}{15}.

The underscore and the hash-tag both are allowed, but come with their specific provisos:

> The _ character can be used, but not as first character of the operator, as it would be mis-construed on usage as part of the previous number, and ignored as such. Although \bnumdefinfix{_}... will not complain, it will remain ineffective. There must be some other character first.
>
> The # character can be used as an operator name or a character in such a name but the definition with \bnumdefinfix must be done either with \string# or with #### (or even double that if the definition is done inside the replacement text of a macro; doubling also applies with the \string method, one will need \string## there) or while having temporarily set the catcode to for example letter or other.
>
> Usage can be done without any special precaution (but as usual # may need doubling if in the replacement body of some macro definition).

{⟨*\macro*⟩} The expandable macro (expecting two mandatory arguments) which is to assign to the infix operator. This macro must be *f*-expandable. Also it must (if the default package configuration is not modified for the core operators) produce integers in the ``strict'' format which is expected by the xintcore macros for arithmetic: no leading zeros, at most one minus sign, no plus sign, no spaces.

{⟨*L-prec*⟩} An integer, minimal 4, maximal 22, which governs the left-precedence of the infix operator.

{⟨*R-prec*⟩} An integer, minimal 4, maximal 22, which governs the right-precedence of the infix operator.

Generally, the two precedences are set to the same value.

Once a multi-character operator is defined, the first characters of its name can be used if no ambiguity. In case of ambiguity, it is the earliest defined shortcut which prevails, except for the full name. So for example if $abc operator is defined, and $ab is defined next, then $ and $a will still serve as shortcuts to the original $abc, but $ab will refer to the newly defined operator.

Fully qualified names are never ambiguous, and a shortcut once defined will change meaning only under two circumstances:

- it is re-defined as the full name of a new operator,
- the original operator to which the shortcut refers is defined again; then the shortcut is automatically updated to point to the new meaning.

```
\def\equals#1#2{\ifnum\pdfstrcmp{#1}{#2}=0 \expandafter1\else
                                           \expandafter0\fi}
```

[3] The effect of \bnumdefinfix is global if under \xintglobaldefstrue setting.

```
% or:
\def\equals#1#2{\expanded{\ifnum\pdfstrcmp{#1}{#2}=0 1\else0\fi}}
\def\differ#1#2{\expanded{\ifnum\pdfstrcmp{#1}{#2}=0 0\else1\fi}}
\bnumdefinfix{==}{\equals}{10}{10}
\bnumdefinfix{!=}{\differ}{10}{10}
\bnumdefinfix{times}{\xintiiMul}{14}{14}
\bnumdefinfix{++}{\xintiiAdd}{19}{19}

\bnumeval{2 + 3! = 5, 2 + (3!) == 8}
```

0, 1

Notice in the `2+3! = 5` example that the existence of `!=` prevails on applying the factorial, so this is test whether `2+3` and `5` differ; it is not a matter of precedence here, but of input parsing ignoring spaces. And `2+3! == 8` would create an error as after having found the `!=` operator and now expecting a digit (as there is no `!==` operator) the parser would find an unexpected `=` and report an error. Hence the usage of parentheses in the input.[4]

```
\bnumeval{2^5 == 4 times 8, 11 t 14}
```

1, 154

```
\bnumeval{100 ++ -10 ^ 3, (100 - 10)^3, 2 ^ 5 ++ 3, 2^(5+3)}
```

729000, 729000, 256, 256

6.6.5 \bnumdefpostfix

It is possible to define postfix unary operators of one's own choosing.[5] The syntax is

\bnumdefpostfix{⟨*operator*⟩}{⟨*\macro*⟩}{⟨*L-prec*⟩}

{⟨*operator*⟩} The characters for the operator name: same conditions as for \bnumdefinfix. Postfix and infix operators share the same name-space, regarding abbreviated names.

> **Warning:** ⟨*operator*⟩ will be expanded in an \edef and then further processed. If it is (or contains) a Babel-active character chances are that the further processing by \bnumdefpostfix will give breakage. There will be no problem if \bnumdefpostfix is used in the document preamble.

{⟨*\macro*⟩} The one argument expandable macro to assign to the postfix operator. This macro only needs to be *x*-expandable.

{⟨*L-prec*⟩} An integer, minimal 4, maximal 22, which governs the left-precedence of the infix operator.

Examples below which use the maximal precedence are typical of what is expected of a ``function'' (and I even used `.len()` notation with parentheses in one example, the parentheses are part of the postfix operator name). And indeed such postfix operators are thus a way to implement functions in disguise, circumventing the fact that the bnumexpr parser will never be

[4] With xintexpr, whose \xinteval has a `!=` operator, `2+3!==8` is interpreted automatically as `2+(3!)==8`, thanks to internal work-around added at 1.4g. This has not been backported to bnumexpr as it does not per default support operators such as `!=` or `==` and only has generic support for adding multi-character operators.

Regarding `2 + 3! = 5`, trying to let this be interpreted as `2+(3!)=5` makes sense only if a `=` operator has been defined. If no `!=` operator exists, the magic will be automatic. If however both `=` and `!=` exist, then it would need special overhead to the parser dealings when finding `!` to avoid the `!=` interpretation. One could imagine distinguishing `! =` from `!=` but the swallowing of spaces is deeply coded in the parser. As bnumexpr default infix operators do not include one starting with `!`, it is not worth it to include in the package extra overhead to solve such issues when extending the syntax. At the level of xintexpr, there is no issue because there is no `=` operator.

[5] The effect of \bnumdefpostfix is global if under \xintglobaldefstrue setting.

extended to work with functional syntax (for this, see xintexpr). With the convention (followed in some examples) that such postfix operators start with a full stop, but never contain another one, we can chain simply by using concatenation (no need for parentheses), as there will be no ambiguity.

```
\usepackage{xint}% for \xintiiSum, \xintiiSqrt
\def\myRev#1{\xintNum{\xintReverseOrder{#1}}}% reverse and trim leading zeros
\bnumdefpostfix{$}{\myRev}{22}%     the $ will have top precedence
\bnumdefpostfix{:}{\myRev}{4}%      the : will have lowest precedence
\bnumdefpostfix{::}{\xintiiSqr}{4}% the :: is a completely different operator
\bnumdefpostfix{.len()}{\xintLength}{22}% () for fun but a single . will be enough!
\bnumdefpostfix{.sumdigits}{\xintiiSum}{22}% .s will abbreviate
\bnumdefpostfix{.sqrt}{\xintiiSqrt}{22}%     .sq will be unambiguous (but confusing)
\bnumdefpostfix{.rep}{\xintReplicate3}{22}%  .r will be unambiguous

\bnumeval{(2^31).len(), (2^31)., 2^31$, 2^31:, (2^31)$}

                          10, 10, 8192, 8463847412, 8463847412

\bnumeval{(2^31).sqrt, 100000000.sq.sq}

                                    46340, 100

\bnumeval{(2^31).sumdigits, 123456789.s, 123456789.s.s, 123456789.s.s.s}

                                    47, 45, 9, 9

\bnumeval{10^10+10000+2000+300+40+5:}

                                   54321000001

\bnumeval{1+2+3+4+5+6+7+8+9+10 :: +1 :: *2 :: :: :}

                          6127162717514063784270898742 11

\bnumeval{123456789.r}
                            12345678912345678912345678 9

\bnumdefpostfix{.rep}{\xintReplicate5}{22}% .rep modified --> .r too

\bnumeval{123456789.r}

                  12345678912345678912345678912345678912345678 9
```

7 Changes

1.7b (2025/09/27) Add to documentation an expandable macro computing Fibonacci numbers and illustrate it with the computation of F_{10000}.

1.7a (2025/09/14)

Bug fix: inputs consisting exclusively of zeros and underscores following an hexadecimal or other prefix caused a crash due to a 1.7 regression. *Yes, we have now added a long delayed test suite doing more than checking only those examples as included in the documentation.*

New feature: if the parser expects a value and finds only underscores (they are allowed as separators for blocks of digits) before hitting an operator, an opening parenthesis (or \bnumexpr sub-expression), a comma, or the end of the expression, it will consider it has fetched the value zero. Formerly, for example expressions reduced to _, __, or entirely empty, or for example _^_ all reported errors.

Other: \evaltohex which signaled via an expandable error its deprecation since 1.6 has been removed.

1.7 (2025/09/13)

Bug fix: inputs with an underscore immediately after a hexadecimal (or other) prefix caused a crash due to a 1.6 regression.

New features:

- Support for hexadecimal input using letters in lowercase.
- Optional argument [ha] for lowercase hexadecimal output.

1.6a (2025/09/07)

Bug fix: the 1.6 support for Babel-active characters worked with \bnumeval (which is recommended interface) but not with \bnumexpr.

1.6 (2025/09/05)

Breaking changes:

- Release 1.4n or later of the xint bundle is required (for those components actually used, which by default are xintkernel, xintcore and xintbinhex).
- \evaltohex is deprecated and causes an error signaling it. *It was removed at 1.7a*. Use new \bnumeval[h].
- \bnumexprsetup was deprecated at 1.5 and kept as alias of \bnumsetup. It has now been removed.
- \bnumprintonetohex and \bnumhextodec, which were documented as customizable do not exist anymore. Check the documentation for \bnumprintonehex and \bnumsetup's key hextodec.
- Under the custom option, not only xintcore but also xintbinhex are not loaded. Use customcore to avoid that. There is also custombinhex.

Bug fix: An underscore _ located in front of a number used to cause an error. It is now ignored.

New features:

- 0b, 0o and 0x are recognized as prefixes for binary, octal, and hexadecimal inputs. And ' is recognized as prefix for octal input, in addition to " for hexadecimal.
- \bnumeval accepts an optional argument [b] or [o] or [h] for automatic conversion of the calculated value (or comma separated values) to respectively binary, octal, or hexadecimal.

- Babel-active characters (such as : and ! with French) do not need any preventive measures anymore such as using \string! in place of !.
- \bnumsetup can now be used also to customize which macros implement conversion from decimal to other bases.

The documentation was extensively revised and made more user-friendly.

1.5 (2021/05/17)

- **breaking change:** the power operators act now in a right associative way; this has been announced at xintexpr as a probable future evolution, and is implemented in anticipation here now.
- **fix two bugs** (imported from upstream xintexpr) regarding hexadecimal input: impossibility to use "\foo syntax (one had to do \expandafter"\foo which is unexpected constraint; a very longstanding xintexpr bug) and issues with leading zeros (since xintexpr 1.2m).
- renamed \bnumexprsetup into \bnumsetup; the former remains available but is deprecated. [REMOVED AT 1.6]
- the customizability and extendibility is now total:
  1. \bnumprintone, \bnumprintonetohex, \bnumprintonesep, \bnumhextodec,
  2. \bnumdefinfix which allows to add extra infix operators,
  3. \bnumdefpostfix which allows to add extra postfix operators.
- \bnumsetup, \bnumdefinfix, \bnumdefpostfix obey the \xintglobaldefstrue and \xintverbosetrue settings.
- documentation is extended, providing details regarding the precedence model of the parser, as inherited from upstream xintexpr; also an example of usage of \bnumsetup is included on how to transform \bnumeval into a calculator with fractions.

1.4a (2021/05/13)

- fix undefined control sequences errors encountered by the parser in case of either extra or missing closing parenthesis (due to a problem in technology transfer at 1.4 from upstream xintexpr).
- fix \BNE_Op_opp must now be *f*-expandable (also caused as a collateral to the technology transfer).
- fix user documentation regarding the constraints applying to the user replacement macros for the core algebra, as they have changed at 1.4.

1.4 (2021/05/12)

- technology transfer from xintexpr 1.4 of 2020/01/31. The \expanded primitive is now required (TeXLive 2019).
- addition to the syntax of the " prefix for hexadecimal input.

- addition of \evaltohex which is like \bnumeval with an extra conversion step to hexadecimal notation.

1.2e (2019/01/08) Fixes a documentation glitch (extra braces when mentioning \the\numexpr or \thebnumexpr).

1.2d (2019/01/07)

- requires xintcore 1.3d or later (if not using option custom).
- adds \bnumeval{⟨*expression*⟩} user interface.

1.2c (2017/12/05) **Breaking changes:**

- requires xintcore 1.2p or later (if not using option custom).
- divtrunc key of \bnumexprsetup is renamed to div.
- the // and /: operators are now by default associated to the *floored* division. This is to keep in sync with the change of xintcore at 1.2p.
- for backwards compatibility, one may add to existing document: \bnumexprsetup{div=\xintiiDivTrunc, mod=\xintiiModTrunc}

1.2b (2017/07/09)

- the _ may be used to separate visually blocks of digits in long numbers.

1.2a (2015/10/14)

- requires xintcore 1.2 or later (if not using option custom).
- additions to the syntax: factorial !, truncated division //, its associated modulo /: and ** as alternative to ^.
- all options removed except custom.
- new command \bnumexprsetup which replaces the commands such as \bnumexprusesbigintcalc.
- the parser is no more limited to numbers with at most 5000 digits.

1.1b (2014/10/28)

- README converted to markdown/pandoc syntax,
- the package now loads only xintcore, which belongs to xint bundle version 1.1 and extracts from the earlier xint package the core arithmetic operations as used by bnumexpr.

1.1a (2014/09/22)

- added l3bigint option to use experimental LaTeX3 package of the same name,
- added Changes and Readme sections to the documentation,
- better \BNE_protect mechanism for use of \bnumexpr...\relax inside an \edef (without \bnethe). Previous one, inherited from xintexpr.sty 1.09n, assumed that the \.=<digits> dummy control sequence encapsulating the computation result had \relax meaning. But removing this assumption was only a matter of letting \BNE_protect protect two, not one, tokens. This will be backported to next version of xintexpr, naturally (done with xintexpr.sty 1.1).

1.1 (2014/09/21) First release. This is down-scaled from the (development version of) xintexpr. Motivation came the previous day from a chat with Joseph Wright over big int status in LaTeX3. The \bnumexpr...\relax parser can be used on top of big int macros of one's choice. Functionalities limited to the basic operations. I leave the power operator ^ as an option.

8 License

9 Commented source code

At 1.7, support for lowercase hexadecimal was added.

At 1.6, \bnumeval requires the 1.4n release of xintcore and xintbinhex (or at least of xintkernel if option custom is used). It adds 0b, 0o, ', and 0x to the syntax, and admits optional parameters [b], [o], and [h] to produce the output converted to binary, octal, or hexadecimal.

It is amusing that implementing the support for the optional argument had the unanticipated corollary that Babel active characters (such as ! with French) are autotaming. See the code comments.

A problem with _ if upfront in numbers was fixed.

There was some refactoring, relative to extending \bnumsetup with new keys related to base conversion macros and this lead to the removal of \bnumprintonetohex and \bnumhextodec.

At 1.5, right-associativity was enforced for powers in anticipation of upstream xintexpr 1.4g 2021/05/25, and the customizability and extendibility of the package is made total via added \bnumdefinfix and \bnumdefpostfix.

Older comments at time of 1.4 and 1.4a releases:

I transferred mid-May 2021 from xintexpr its \expanded based infra-structure from its own 1.4 release of January 2020 and bumped version to 1.4. Also I added support for hexadecimal input and output, via xintbinhex.

A few comments added here at 1.4a:

- It looked a bit costly and probably would have been mostly useless to end users to integrate in bnumexpr support for nested structures via square brackets [,], which is in xintexpr since its January 2020 1.4 release. But some of the related architecture remains here; we could make some gains probably but diverging from upstream code would make maintenance a nightmare.
- Formerly, the \csname...\endcsname encapsulation technique had the after-effect to allow the macros supporting the infix operators to be only *x*-expandable. At 1.4, I could have still allowed support

macros being only *x*-expandable, but, keeping in sync with upstream, I have used only a \romannumeral trigger and did not insert an \expanded, so now the support macros must be *f*-expandable. The 1.4a release fixes the related user documentation of \bnumsetup which was not updated at 1.4. The support macro for the factorial however needs only be *x*-expandable.

- Also, I simply do not understand why the legacy (1.2e) user documentation said that the support macros were supposed to *f*-expand their arguments, as they are used only with arguments being explicit digit tokens (and optional minus sign).
- The \bnumexpr\relax syntax creating an empty ople is by itself now legal, and can be injected (comma separated) in an expression, keeping it invariant, however \bnumeval{} ends in a Paragraph ended before \BNE_print_c was complete error because \BNEprint makes the tacit requirement that the 1D ople to output has at least one item.

9.1 Package identification

```
\NeedsTeXFormat{LaTeX2e}%
\ProvidesPackage{bnumexpr}[2025/09/27 v1.7b Expressions with big integers (JFB)]%
```

9.2 Load xintkernel

At 1.6, in order to make the base conversion macros also customizable, hence not mandate loading of xintbinhex, we only load unconditionally xintkernel.

We then switch to the familiar catcode regime of the xintexpr sources.

```
\RequirePackage{xintkernel}[2025/09/05]%
```

9.3 Save catcode regime and switch to our own

```
\edef\BNErestorecatcodesendinput{\XINTrestorecatcodes\noexpand\endinput}%
\XINTsetcatcodes%
```

9.4 Load optionally xintcore and xintbinhex

1.6 adds customcore as alias of legacy custom. It adds custombinhex to add possibility of not loading xintbinhex either. Option custom now means both of customcore and custombinhex.

But who on Earth isn't going to use with delight both my xintcore and xintbinhex?

```
\def\BNE_tmpa{1}\def\BNE_tmpb{1}%
\DeclareOption{custom}{\def\BNE_tmpa{0}\def\BNE_tmpb{0}}%
\DeclareOption{customcore}{\def\BNE_tmpa{0}}%
\DeclareOption{custombinhex}{\def\BNE_tmpb{0}}%
\ProcessOptions\relax
\if1\BNE_tmpa\RequirePackage{xintcore}[2025/09/05]\fi
\if1\BNE_tmpb\RequirePackage{xintbinhex}[2025/09/05]\fi
```

9.5 \bnumsetup

\bnumsetup is the new name at 1.5 of \bnumexprsetup. The old name was kept as an alias at 1.5, and deleted at 1.6.

Note that a final comma will cause no harm.

```
\catcode`! 3
\def\bnumsetup #1{\BNE_parsekeys #1,=!,}%
```

```
\def\BNE_parsekeys #1=#2#3,%
  {%
    \ifx!#2\expandafter\BNE_parsedone\fi
  \XINT_global
    \expandafter
    \let\csname BNE_Op_\xint_zapspaces #1 \xint_gobble_i\endcsname%
    =#2%
  \ifxintverbose
    \PackageInfo{bnumexpr}{assigned
    \ifxintglobaldefs globally \fi
     \string#2 to \xint_zapspaces #1 \xint_gobble_i\MessageBreak
```

Workaround for the space inserted by \on@line.

```
     \expandafter\xint_firstofone}%
  \fi
  \BNE_parsekeys
  }%
\def\BNE_parsedone #1\BNE_parsekeys {}%
\catcode`! 12
```

Final comma and spaces are only to check if it does work. But I will NOT insert spaces before commas, even though they are allowed!

1.6 also handles base conversion macros here. Prior to 1.6 this \bnumsetup configuration was not executed if package received option custom (now customcore). But as the user is then responsible for redefining all keys, why bother.

```
\bnumsetup{%
  add = \xintiiAdd, sub = \xintiiSub, opp = \xintiiOpp,
  mul = \xintiiMul, pow = \xintiiPow, fac = \xintiiFac,
  div = \xintiiDivFloor, mod = \xintiiMod, divround = \xintiiDivRound,
  hextodec=\xintHexToDec, octtodec=\xintOctToDec, bintodec=\xintBinToDec,
  dectohex=\xintDecToHex, dectooct=\xintDecToOct, dectobin=\xintDecToBin,
}%
```

By the way the keys should have been Add, Sub, ..., not add, sub, ..., so internally \BNE_Op_Add etc... would have been the macros defined by \bnumsetup and used in the code, not \BNE_Op_add (et al.) whose casing does not match my naming conventions.

9.6 Some extra constants needed for user defined precedences

For the mechanism of \bnumdefinfix we need precedence levels to be available as \chardef's. xintkernel already provides 0-10, 12, 14, 16, 17, 18, 20, 22.

Left levels need to be represented by one token; right levels are hard-coded into checkp_<op> macros and could have been there explicit digit tokens but we will use the \xint_c_... \char-tokens.

```
\chardef\xint_c_xi   11
\chardef\xint_c_xiii 13
\chardef\xint_c_xv   15
\chardef\xint_c_xix  19
\chardef\xint_c_xxi  21
```

9.7 \bnumexpr, \bnethe, \bnumeval

\XINTfstop has to be the same as defined in xintexpr, in order for a subexpression \xintiiexpr...\relax to get recognized in \bnumeval or conversely for \bnumexpr...\relax to possibly serve inside an \xinteval. But why use bnumexpr then? Besides a sub xintexpression will break \bnumeval if it is anything else than a 1D flat sequence. And even then it can work only if internal storage format are kept in sync.

1.6 deprecates \evaltohex (and 1.7a removed it) and has \bnumeval[h] as equivalent replacement.

The \protected \BNEprint will survive to \bnumexpr being expanded in a \write or \edef. But its expansion will be forced by the \expanded from \bnethe.

I now really dislike \thebnumexpr macro name and at some point had replaced it with \bnumtheexpr but this got reverted.

1.6a uses the strange \csname in place of directly \BNE_wrap in order to fix the 1.6 blunder which had done a similar thing, but too late. This is to tame Babel active characters. The \bnumeval was OK, though. Sadly the blunder was first done in xintexpr, after I had reverted perfectly valid implementation there, having thought I could apply a shortcut, which was simply a brain fault. And I backported it here... alas...

```
\def\XINTfstop {\noexpand\XINTfstop}%
\def\bnumexpr  {\romannumeral0\bnumexpro}%
\def\bnumexpro {\csname BNE_wrap\expandafter\endcsname
                \romannumeral0\BNE_bareeval}%
```

While preparing 1.6 I wondered why the ``.'' after \BNEprint in \BNE_wrap which is then gobbled by \BNEprint. It was clear it came from xintexpr, but why was it kept here?

The reason is to support having a sub \bnumexpr...\relax inside \bnumeval or \xinteval. Indeed such a sub-expression is identified via the presence of the \XINTfstop after its expansion, and the code inside bnumexpr handling this is inherited from xintexpr, so it expects the structure \XINTfstop then a ``print'' macro, then possibly some stuff delimited by a full stop (this is related to the implementation of the optional arguments of \xintfloateval and \xintieval).

As we keep this stuff handled the same way we must inject the seemingly silly full stop here for \bnumexpr...\relax (or a macro defined from it via an \edef) to be usable inside \bnumval or another \bnumexpr...\relax.

```
\def\BNE_wrap  {\XINTfstop\BNEprint.}%
```

It is important to keep in mind that #1 has the structure {{...}{...}...{...}} with an external brace pair, which here gets removed. In the replacement the external {...} are for \expanded.

We also define a non \protected variant without the strange extra full stop, it will serve for \bnumeval (and \thebnumexpr) and thus does not need it.

```
\protected\def\BNEprint.#1{{\BNE_unpack#1.}}%
\def\BNEprint_#1{{\BNE_unpack#1.}}%
```

\bnethe removes the \XINTfstop and activates the printing via \BNEprint.

Attention that prior to 1.6 \bnethe grabbed a #1, hence would work to print a braced \bnumexpr...\relax, but I don't see the reason for doing that. Removed.

1.6a modifies \thebnumexpr here for the Babel active thing.

```
\def\bnethe{\expanded\expandafter\xint_gobble_i\romannumeral`&&@}%
```

```
\def\thebnumexpr{\expanded\csname BNEprint_\expandafter\endcsname
                 \romannumeral0\BNE_bareeval}%
```

At 1.6 after implementing the [h] optional argument of \bnumeval, there was the unanticipated result that this tamed Babel active characters. This is explained by the expansion happening while a \csname is not yet closed. And by the fact that during its expansion \bnumeval does not use delimited macros, for example to fetch up to a closing parenthesis.

There used to be here a \BNE_start but it got replaced by its expansion.

The \BNE_check is defined in the section ``Expansion spanning''.

Prior to 1.6 \BNE_bareeval was named \bnebareeval, but this was outside of the package namespace (it should have been \bnumbareeval, or \bnumexprbareeval). Upstream has \xintbareeval without underscores for legacy reasons.

```
\def\BNE_bareeval{\expandafter\BNE_check\romannumeral`&&@\BNE_getnext}%
```

These next are not \protected because they are only used with \bnumeval, there is no analog of the private format which \bnumexpr expands to. This also spares us having to define macros with names which can be written to an external file and re-read using the standard catcodes.

MEMO: \BNEprint_ (with the trailing underscore) will be used in case of absence of optional argument and has been defined already. It is important for compatibility with the others here that it did not use the strange full stop in its parameter pattern. It is also important that it is *not* \protected, as we want \bnumeval to expand fully in an \edef.

```
\expandafter\def\csname BNEprint_[h]\endcsname#1{{\BNE_unpack_tohex#1.}}%
\expandafter\def\csname BNEprint_[ha]\endcsname#1{{\BNE_unpack_tolowhex#1.}}%
\expandafter\def\csname BNEprint_[o]\endcsname#1{{\BNE_unpack_tooct#1.}}%
\expandafter\def\csname BNEprint_[b]\endcsname#1{{\BNE_unpack_tobin#1.}}%
\expandafter\let\csname BNEprint_[]\endcsname\BNEprint_
```

[b], [o] and [h] added at 1.6.

We break here the xint legacy pattern to always use \romannumeral0 trigger first, via a CamelCase/lowercase pair of macros.

The \expanded will *not* control the actual computation, only launch it via propagation across the \csname...\endcsname to a \romannumeral0 for the *f*-expansion of \BNE_bareeval. After computation is done the \BNEprint_[...] will present the braced result (maybe containing multiple items) to this \expanded, and the unpacking is thus done via an \edef-like expansion. This does not mean that all helpers doing conversion to the binary bases only have to be *x*-expandable, because their full expansion must be achieved prior to the ``printone'' macros activate. If the latter are \protected, then conversions do only need *x*-expandability. Except for the support of the [ha] option which needs \BNE_Op_dectohex to be *f*-expandable as the helper doing conversion to lowercase must act on a fully prepared uppercase hexadecimal input.

```
\def\bnumeval #1#{\expanded\bnumeval_a{#1}}%
\def\bnumeval_a#1#2{%
   \csname BNEprint_\xint_zapspaces #1 \xint_gobble_i\expandafter
   \endcsname\romannumeral0\BNE_bareeval#2\relax
}%
```

This code is more compact at 1.6 than at 1.5. Various renamings at 1.7 and addition of the [ha] optional argument.

```
\def\BNE_unpack#1{%
   \bnumprintone{#1}\expandafter\BNE_unpack_a\string
}%
\def\BNE_unpack_a#1{%
   \if#1.\BNE_allitemsdone\fi\bnumprintonesep
   \expandafter\BNE_unpack\expandafter{\iffalse}\fi
}%
\def\BNE_allitemsdone\fi#1\fi{\fi}%
```

There is a breaking change at 1.6 as formerly there was a \bnumprintonetohex. Now, the decimal to hexadecimal conversion is done always, and the customizable wrapper was thus renamed to \bnumprintonehex.

```
\def\BNE_unpack_tohex#1{%
    \expandafter\bnumprintonehex
    \expandafter{\romannumeral`&&@\BNE_Op_dectohex{#1}}%
    \expandafter\BNE_unpack_tohex_a\string
}%
\def\BNE_unpack_tohex_a#1{%
    \if#1.\BNE_allitemsdone\fi\bnumprintonesep
    \expandafter\BNE_unpack_tohex\expandafter{\iffalse}\fi
}%
```

Conversion to lowercase hexadecimal added at 1.7. By the way [ha] option is broken (but also is not needed) if the \BNE_Op_dectohex is customized to produce lowercase, not uppercase as \xintiiDecToHex.

As we inject ending pattern already here, to skip one ``grab argument'' step, we do not provide a wrapper which could be configured via \bnumsetup with something such as dectolowhex=\macro.

```
\def\BNE_unpack_tolowhex#1{%
    \expandafter\bnumprintonelowerhex
    \expanded{{\BNE_dectolowhex_a{#1}\xint_bye23456789\xint_bye\endcsname}}%
    \expandafter\BNE_unpack_tolowhex_a\string
}%
\def\BNE_unpack_tolowhex_a#1{%
    \if#1.\BNE_allitemsdone\fi\bnumprintonesep
    \expandafter\BNE_unpack_tolowhex\expandafter{\iffalse}\fi
}%
```

So the dectohex must be *f*-expandable: even with a \bnumprintonelowerhex redefined \protected, the conversion to lowercase needs the \BNE_Op_dectohex action to be complete before \BNE_dectolowhex_b kicks in.

```
\def\BNE_dectolowhex_a{%
    \expandafter\BNE_dectolowhex_b\romannumeral`&&@\BNE_Op_dectohex
}%
```

When preparing the 1.7 release I hesitated about testing the token to check if a digit or a letter (which could have been done on the basis of the catcode only, but would have then put a constraint on the catcodes used by the \BNE_Op_dectohex macro), and let digits go through ``as is''. But I was not willing to think too much and the advantage of the approach chosen is the elegance of its termination. I did not do comparative

efficiency tests, which would have cost me at least one or two hours in setting them up and comparing with alternative implementation, besides I would have had to decide if I base decision on behavior of very long inputs or of the short ones. All of that for something perhaps nobody on Earth will ever use anyhow.

```
\def\BNE_dectolowhex_b #1#2#3#4#5#6#7#8#9{%
    \csname BNE_lower #1\endcsname
    \csname BNE_lower #2\endcsname
    \csname BNE_lower #3\endcsname
    \csname BNE_lower #4\endcsname
    \csname BNE_lower #5\endcsname
    \csname BNE_lower #6\endcsname
    \csname BNE_lower #7\endcsname
    \csname BNE_lower #8\endcsname
    \csname BNE_lower #9\endcsname
    \BNE_dectolowhex_b
}%
\expandafter\let\csname BNE_lower \endcsname\empty
\expandafter\def\csname BNE_lower 0\endcsname{0}%
\expandafter\def\csname BNE_lower 1\endcsname{1}%
\expandafter\def\csname BNE_lower 2\endcsname{2}%
\expandafter\def\csname BNE_lower 3\endcsname{3}%
\expandafter\def\csname BNE_lower 4\endcsname{4}%
\expandafter\def\csname BNE_lower 5\endcsname{5}%
\expandafter\def\csname BNE_lower 6\endcsname{6}%
\expandafter\def\csname BNE_lower 7\endcsname{7}%
\expandafter\def\csname BNE_lower 8\endcsname{8}%
\expandafter\def\csname BNE_lower 9\endcsname{9}%
\expandafter\def\csname BNE_lower A\endcsname{a}%
\expandafter\def\csname BNE_lower B\endcsname{b}%
\expandafter\def\csname BNE_lower C\endcsname{c}%
\expandafter\def\csname BNE_lower D\endcsname{d}%
\expandafter\def\csname BNE_lower E\endcsname{e}%
\expandafter\def\csname BNE_lower F\endcsname{f}%
```

Octal and binary added at 1.6.

```
\def\BNE_unpack_tooct#1{%
    \expandafter\bnumprintoneoct
    \expandafter{\romannumeral`&&@\BNE_Op_dectooct{#1}}%
    \expandafter\BNE_unpack_tooct_a\string
}%
\def\BNE_unpack_tooct_a#1{%
    \if#1.\BNE_allitemsdone\fi\bnumprintonesep
    \expandafter\BNE_unpack_tooct\expandafter{\iffalse}\fi
}%
\def\BNE_unpack_tobin#1{%
    \expandafter\bnumprintonebin
    \expandafter{\romannumeral`&&@\BNE_Op_dectobin{#1}}%
    \expandafter\BNE_unpack_tobin_a\string
}%
\def\BNE_unpack_tobin_a#1{%
    \if#1.\BNE_allitemsdone\fi\bnumprintonesep
    \expandafter\BNE_unpack_tobin\expandafter{\iffalse}\fi
```

```
}%
\let\bnumprintone   \xint_firstofone
\let\bnumprintonehex\xint_firstofone
\let\bnumprintonelowerhex\xint_firstofone
\let\bnumprintoneoct\xint_firstofone
\let\bnumprintonebin\xint_firstofone
\def\bnumprintonesep{, }%
```

9.8 \BNE_getnext

The upstream \BNE_put_op_first has a string of included \expandafter, which was imported here at 1.4 and 1.4a but they serve nothing in our context. Removed this useless overhead at 1.5.

This \BNE_getnext token is injected via "start" macros associated to operators or similar syntax elements, as will be seen later on. It tries to get next operand.

```
\def\BNE_getnext #1%
{%
    \expandafter\BNE_put_op_first\romannumeral`&&@%
    \expandafter\BNE_getnext_a\romannumeral`&&@#1%
}%
\def\BNE_put_op_first #1#2#3{#2#3{#1}}%
\def\BNE_getnext_a #1%
{%
    \ifx\relax #1\xint_dothis\BNE_foundprematureend\fi
    \ifx\XINTfstop#1\xint_dothis\BNE_subexpr\fi
    \ifcat\relax#1\xint_dothis\BNE_countetc\fi
    \xint_orthat{}\BNE_getnextfork #1%
}%
```

This is executed when the parser was trying to find an operand but ended up hitting the \relax end-marker. This happens if the expression is empty for example. It trickled down here from xintexpr, but \xinteval has a notion of empty value. This is not the case of the bnumexpr parser. So let's at 1.7a allow \bnumeval{} or \bnumeval{_}. Indeed \bnumeval{"} and \bnumeval{"_} which use an other codebranch worked already. This goes via replacing {} by {0}.

```
\def\BNE_foundprematureend\BNE_getnextfork #1{{0}\xint_c_\relax}%
\def\BNE_subexpr #1.#2%
{%
    \expanded{\unexpanded{{#2}}\expandafter}\romannumeral`&&@\BNE_getop
}%
```

At 1.6 this also filters for \catcode (as per xint 1.4g 2021/05/25).

```
\def\BNE_countetc\BNE_getnextfork#1%
{%
    \if0\ifx\count#11\fi
        \ifx\numexpr#11\fi
        \ifx\catcode#11\fi
        \ifx\dimen#11\fi
        \ifx\dimexpr#11\fi
        \ifx\skip#11\fi
        \ifx\glueexpr#11\fi
        \ifx\fontdimen#11\fi
```

```
        \ifx\ht#11\fi
        \ifx\dp#11\fi
        \ifx\wd#11\fi
        \ifx\fontcharht#11\fi
        \ifx\fontcharwd#11\fi
        \ifx\fontchardp#11\fi
        \ifx\fontcharic#11\fi
       0\expandafter\BNE_fetch_as_number\fi
   \expandafter\BNE_getnext_a\number #1%
}%
\def\BNE_fetch_as_number
    \expandafter\BNE_getnext_a\number #1%
{%
    \expanded{{{\number#1}}\expandafter}\romannumeral`&&@\BNE_getop
}%
```

In the case of hitting a `(`, previous release inserted directly a `\BNE_oparen`. But the expansion architecture imported from upstream `\xintiiexpr` has been refactored, and the `..._oparen` meaning and usage evolved. We stick with `{}\xint_c_ii^v (` from upstream.

Also, at `1.6`, slight refactoring to handle digit tokens and opening parenthesis a bit faster (but this is only first token...); and to ignore an underscore as first character (rather than raise an error in this case).

This merges former `\BNE_getnextfork` and `\BNE_scan_number`.

```
\def\BNE_getnextfork #1{%
    \if#1-\xint_dothis {{}{}-}\fi
    \if#1(\xint_dothis {{}\xint_c_ii^v (}\fi
    \ifnum\xint_c_ix<1\string#1 \xint_dothis {\BNE_startint#1}\fi
    \xint_orthat {\BNE_getnextfork_a #1}%
}%
```

Prior to `1.7a`, the `_` was handled as is a `+` encountered while looking for a value, i.e. was simply ignored and expansion continued with `\BNE_get_next_a`. This made `_^` raise an ``unexpected token'' error when hitting the `^`. But in contrast `"_^` does work without complaining and is the same as `0^`. So at `1.7a` we jump to `\BNE_startint`. Mind that the latter differs from `\BNE_startoct` and `\BNE_starthex` and expects to grab an already expanded argument, and naturally we inject a zero. As a corollary of this `1.7a` change, also `_x`, `_o` and `_b` can serve as prefixes for hexadecimal, octal and binary.

```
\def\BNE_getnextfork_a #1{%
    \if#1_\xint_dothis {\BNE_startint 0}\fi
    \if#1+\xint_dothis \BNE_getnext_a \fi
    \if#1'\xint_dothis \BNE_startoct\fi
    \if#1"\xint_dothis \BNE_starthex\fi
    \xint_orthat {\BNE_unexpected #1}%
}%
```

If user employs `\bnumdefinfix` with `\string#`, and then tries `100##3`, the first `#` will be interpreted as operator (assuming no operator starting with `##` has actually been defined) and the error "message" (which is not using \message or a \write) will then be

```
! xint error: Unexpected token `##'. Ignoring.
```

because the parser is actually looking for a digit but finds the second `#`, and TeX displays it doubled. This is doubly confusing, but well, let's not dwell on that.

\BNE_unexpected replaced here \BNE_notadigit at 1.6.

```
\def\BNE_unexpected#1%
{%
    \XINT_expandableerror{Unexpected token `#1'. Ignoring.}\BNE_getnext_a
}%
```

9.9 Parsing decimal, hexadecimal, octal, and binary

Somewhat refactored at 1.6 compared to upstream 1.4m, on the occasion of adding support for prefixes beyond only " as earlier.

Ironically, 1.6 fixed the case of an underscore _ as first character in decimal input, but caused a regression for " prefixed hexadecimal, which ceased to allow a _ right after the ". Fixed at 1.7.

But 1.7 introduced a regression causing "0 input to now break... pfff... hence 1.7a. A secondary (longstanding) issue is that the test suite was limited to examples as shown in the docs (which used a special mark-up to check automatically for regressions), but this way made inconvenient to test for special inputs such as an hexadecimal prefix only followed with zeros and underscores, which is what got broken in a too speedy refactoring at 1.7. The development sources now have a better way of checking against regression prior to releases.

9.9.1 Prefix dispatch

```
\def\BNE_startint #1%
{%
    \if #10\expandafter\BNE_scanint_gobz_a\else\expandafter\BNE_scanint_a\fi #1%
}%
\def\BNE_wrapint_after{\iffalse{{{\fi}}}}%
\def\BNE_scanint_gobz_a #1#2{%
     \expandafter\BNE_scanint_gobz_b\romannumeral`&&@#2%
}%
```

It is important in case of x, o, or b to jump to \BNE_starthex (et al.) and not for example to \BNE_scanhex_a because the latter expects an *f*-expansion to have been applied already to what comes next (this comment is half-obsolete at 1.7 which has no \BNE_scanhex_a anymore).

Besides, we do want to trim out leading zeroes after the 0b, 0o, or 0x prefix: although the macros of xintbinhex do accept leading zeros on input, they may then produce decimal output with leading zeros, and the ``ii'' macros of xintcore consider that an input is vanishing as soon as the first digit is 0.

```
\def\BNE_scanint_gobz_b #1%
{%
    \ifx b#1\xint_dothis \BNE_startbin \fi
    \ifx o#1\xint_dothis \BNE_startoct \fi
    \ifx x#1\xint_dothis \BNE_starthex \fi
    \xint_orthat {\BNE_scanint_gobz_c #1}%
}%
\def\BNE_scanint_gobz_c #1%
{%
    \expandafter{\expanded{{\iffalse}}}\fi
```

```
    \BNE_scanint_gobz_main#1%
}%
```

9.9.2 Decimal

No \BNE_wrapint_before, it has been inlined here at 1.7 (and in \BNE_scanint_gobz_c).

```
\def\BNE_scanint_a #1#2{%
     \expandafter{\expanded{{\iffalse}}}\fi #1%
     \expandafter\BNE_scanint_main\romannumeral`&&@#2%
}%
\def\BNE_scanint_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanint_hit_cs \fi
    \ifnum\xint_c_ix<1\string#1 \else\expandafter\BNE_scanint_checkagain\fi
    #1\BNE_scanint_again
}%
\def\BNE_scanint_again #1%
{%
    \expandafter\BNE_scanint_main\romannumeral`&&@#1%
}%
```

Upstream (at 1.4f) has _getop here, but let's jump directly to BNE_getop_a.

```
\def\BNE_scanint_hit_cs \ifnum#1\fi#2\BNE_scanint_again
{%
    \expandafter\BNE_wrapint_after\romannumeral`&&@\BNE_getop_a#2%
}%
\def\BNE_scanint_checkagain #1\BNE_scanint_again
{%
    \if _#1\BNE_scanint_checkagain_skip\fi
    \expandafter\BNE_wrapint_after\romannumeral`&&@\BNE_getop_a#1%
}%
```

#1 is \fi.

```
\def\BNE_scanint_checkagain_skip#1#2\BNE_getop_a#3{#1\BNE_scanint_again}%
\def\BNE_scanint_gobz_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanint_gobz_hit_cs\fi
    \ifnum\xint_c_x<1\string#1 \else\expandafter\BNE_scanint_gobz_checkagain\fi
    #1\BNE_scanint_again
}%
```

Upstream (at 1.4f) has _getop here, but let's jump directly to BNE_getop_a. The #2 has been grabbed already and f-expanded. But this means one less brace-stripping.

```
\def\BNE_scanint_gobz_hit_cs\ifnum#1\fi#2\BNE_scanint_again
{%
    0\expandafter\BNE_wrapint_after\romannumeral`&&@\BNE_getop_a#2%
}%
```

Fix at 1.6 for when an underscore is used as first character followed by digits. No need to worry about being very efficient here.

```
\def\BNE_scanint_gobz_checkagain #1\BNE_scanint_again
{%
    \if    _#1\xint_dothis\BNE_scanint_gobz_again\fi
```

```
    \if    0#1\xint_dothis\BNE_scanint_gobz_again\fi
    \xint_orthat
    {0\expandafter\BNE_wrapint_after\romannumeral`&&@\BNE_getop_a#1}%
}%
\def\BNE_scanint_gobz_again #1%
{%
    \expandafter\BNE_scanint_gobz_main\romannumeral`&&@#1%
}%
```

9.9.3 Hexadecimal

At 1.6 the code here is refactored to follow closely the scanint one, rather than downscaling upstream xintexpr which also has to handle fractional input. This avoids gathering the hexadecimal digits then grabbing them again as a whole via a delimited macro.

Removing leading zeros and leading underscores is the job of \BNE_scanhex_gobz_main.

Things could be done better if \BNE_Op_hextodec was only required to be *x*-expandable. Already now we could inline the \BNE_wraphex_before sparing its use of \iffalse..\fi to move closing braces to the one after \expanded. But we would still have three \expandafter's in total.

```
\def\BNE_starthex #1%
{%
    \expandafter\BNE_wraphex_before
    \expanded{{\iffalse}}\fi
    \expandafter\BNE_scanhex_gobz_main\romannumeral`&&@#1%
}%
\def\BNE_wraphex_before{\expandafter{\expandafter{%
                          \romannumeral`&&@\iffalse}}\fi\BNE_Op_hextodec}%
\def\BNE_wraphex_after{\iffalse{{{{\fi}}}}}%
```

At 1.6 we apply exact same scheme as for the scanint code. The sole difference is the more complicated test for recognizing a digit.

At 1.7 the code evolved to support a..f as hexadecimal input. As \bnumeval supports no functions or user variables, there is no breaking change with tacit multiplication as would be the case for \xinteval if we did the same addition to it.

The expansion is \expanded-governed which spares us quite some annoyances with exiting from TEX nested conditionals.

Hesitation whether to check for lowercase hexadecimal before or after checking for the underscore.

```
\def\BNE_scanhex_main #1%
{%
    \ifcat \relax #1\BNE_scanhex_done_hit_cs #1\fi
    \if\ifnum`#1>`/
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1%
    #1%
    \else
      \if _#1\else
        \if\ifnum`#1>``
           \ifnum`#1>`f 0\else1\fi\else0\fi 1%
```

```
            \csname BNE_upper #1\endcsname
          \else
            \BNE_scanhex_done #1%
          \fi
        \fi
      \fi
      \BNE_scanhex_again
}%
\expandafter\def\csname BNE_upper a\endcsname{A}%
\expandafter\def\csname BNE_upper b\endcsname{B}%
\expandafter\def\csname BNE_upper c\endcsname{C}%
\expandafter\def\csname BNE_upper d\endcsname{D}%
\expandafter\def\csname BNE_upper e\endcsname{E}%
\expandafter\def\csname BNE_upper f\endcsname{F}%
```

#2 is \fi.

```
\def\BNE_scanhex_done_hit_cs #1#2#3\BNE_scanhex_again
{%
    #2\expandafter\BNE_wraphex_after\romannumeral`&&@\BNE_getop_a#1%
}%
```

#2 is \fi\fi\fi or \fi\fi\fi\fi (if called from the gobz variant).

```
\def\BNE_scanhex_done #1#2\BNE_scanhex_again
{%
    #2\expandafter\BNE_wraphex_after\romannumeral`&&@\BNE_getop_a#1%
}%
\def\BNE_scanhex_again #1%
{%
    \expandafter\BNE_scanhex_main\romannumeral`&&@#1%
}%
```

Initial scanning which skips leading zeros and underscores. Priority in efficiency is when there are none. Slight refactoring to check for lowercase hexadecimal earlier at 1.7a. But the \BNE_scanhex_main will then again check for underscores prior to them.

Also 1.7a fixes the unfortunate 1.7 regression when only zeros (or underscores) are found after the hexadecimal (or other) prefixes.

```
\def\BNE_scanhex_gobz_main #1%
{%
    \ifcat \relax #1%
        0\BNE_scanhex_done_hit_cs #1\fi
    \if\ifnum`#1>`0
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1%
    #1%
    \else
      \if\ifnum`#1>`` \ifnum`#1>`f 0\else1\fi\else0\fi 1%
        \csname BNE_upper #1\endcsname
      \else
        \if 0#1\BNE_scanhex_gobzero\else
        \if _#1\BNE_scanhex_gobunderscore\else
           0\BNE_scanhex_done #1%
```

```
        \fi\fi
      \fi
    \fi
    \BNE_scanhex_again
}%
\def\BNE_scanhex_gobzero #1\BNE_scanhex_again #2%
{%
    \fi\fi\fi
    \expandafter\BNE_scanhex_gobz_main\romannumeral`&&@#2%
}%
\def\BNE_scanhex_gobunderscore #1\BNE_scanhex_again #2%
{%
    \fi\fi\fi\fi
    \expandafter\BNE_scanhex_gobz_main\romannumeral`&&@#2%
}%
```

9.9.4 Octal

Added at 1.6. Leading zeros are removed.

At 1.7 the code for hexadecimal was a bit refactored and the one here was changed to follow same pattern.

```
\def\BNE_startoct #1%
{%
    \expandafter\BNE_wrapoct_before
    \expanded{{\iffalse}}\fi
    \expandafter\BNE_scanoct_gobz_main\romannumeral`&&@#1%
}%
\def\BNE_wrapoct_before{\expandafter{\expandafter{%
                          \romannumeral`&&@\iffalse}}\fi\BNE_Op_octtodec}%
\def\BNE_wrapoct_after{\iffalse{{{{\fi}}}}}%
\def\BNE_scanoct_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanoct_done_hit_cs #1\fi
    \if\ifnum`#1>`/ \ifnum`#1>`7 0\else1\fi\else0\fi 1%
    #1%
    \else
      \if _#1\else
        \BNE_scanoct_done #1%
      \fi
    \fi
    \BNE_scanoct_again
}%
```

#2 is \fi.

```
\def\BNE_scanoct_done_hit_cs #1#2#3\BNE_scanoct_again
{%
    #2\expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#1%
}%
```

#2 is \fi\fi or \fi\fi\fi (if called from the gobz variant).

```
\def\BNE_scanoct_done #1#2\BNE_scanoct_again
{%
    #2\expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#1%
```

```
}%
\def\BNE_scanoct_again #1%
{%
    \expandafter\BNE_scanoct_main\romannumeral`&&@#1%
}%
\def\BNE_scanoct_gobz_main #1%
{%
    \ifcat \relax #1%
      0\BNE_scanoct_done_hit_cs #1\fi
    \if\ifnum`#1>`0 \ifnum`#1>`7 0\else1\fi\else0\fi 1%
    #1%
    \else
      \if 0#1\BNE_scanoct_gobzero\else
      \if _#1\BNE_scanoct_gobunderscore\else
        0\BNE_scanoct_done #1%
      \fi\fi
    \fi
    \BNE_scanoct_again
}%
\def\BNE_scanoct_gobzero #1\BNE_scanoct_again #2%
{%
    \fi\fi
    \expandafter\BNE_scanoct_gobz_main\romannumeral`&&@#2%
}%
\def\BNE_scanoct_gobunderscore #1\BNE_scanoct_again #2%
{%
    \fi\fi\fi
    \expandafter\BNE_scanoct_gobz_main\romannumeral`&&@#2%
}%
```

9.9.5 Binary

Added at 1.6. Same code skeleton as for octal and hexadecimal.

```
\def\BNE_startbin #1%
{%
    \expandafter\BNE_wrapbin_before
    \expanded{{\iffalse}}\fi
    \expandafter\BNE_scanbin_gobz_main\romannumeral`&&@#1%
}%
\def\BNE_wrapbin_before{\expandafter{\expandafter{%
                          \romannumeral`&&@\iffalse}}\fi\BNE_Op_bintodec}%
\def\BNE_wrapbin_after{\iffalse{{{{\fi}}}}}%
\def\BNE_scanbin_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanbin_done_hit_cs #1\fi
    \if \if0#11\else\if1#11\else0\fi\fi 1%
    #1%
    \else
      \if _#1\else
        \BNE_scanbin_done #1%
      \fi
    \fi
```

```
    \BNE_scanbin_again
}%
```

#2 is \fi.

```
\def\BNE_scanbin_done_hit_cs #1#2#3\BNE_scanbin_again
{%
    #2\expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#1%
}%
```

#2 is \fi\fi or \fi\fi\fi (if called from the gobz variant).

```
\def\BNE_scanbin_done #1#2\BNE_scanbin_again
{%
    #2\expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#1%
}%
\def\BNE_scanbin_again #1%
{%
    \expandafter\BNE_scanbin_main\romannumeral`&&@#1%
}%
\def\BNE_scanbin_gobz_main #1%
{%
    \ifcat \relax #1%
      0\BNE_scanbin_done_hit_cs #1\fi
    \if \if0#11\else\if1#11\else0\fi\fi 1%
    #1%
    \else
      \if 0#1\BNE_scanbin_gobzero\else
      \if _#1\BNE_scanbin_gobunderscore\else
        0\BNE_scanbin_done #1%
      \fi\fi
    \fi
    \BNE_scanbin_again
}%
\def\BNE_scanbin_gobzero #1\BNE_scanbin_again #2%
{%
    \fi\fi
    \expandafter\BNE_scanbin_gobz_main\romannumeral`&&@#2%
}%
\def\BNE_scanbin_gobunderscore #1\BNE_scanbin_again #2%
{%
    \fi\fi\fi
    \expandafter\BNE_scanbin_gobz_main\romannumeral`&&@#2%
}%
```

9.10 \BNE_getop

The upstream analog to \BNE_getop_a applies \string to #1 in its thirdofthree branch before handing over to analog of \BNE_scanop_a, but I see no reason for doing it here (and I do have to check if upstream has any valid reason to do it). Removed. First branch was a \BNE_foundend, used only here, and expanding to \xint_c_\relax, let's move the #1 (which will be \relax) last and simply insert \xint_c_.

The _scanop macros have been refactored at upstream and here 1.5.

```
\def\BNE_getop #1%
```

```
{%
    \expandafter\BNE_getop_a\romannumeral`&&@#1%
}%
\catcode`* 11
\def\BNE_getop_a #1%
{%
    \ifx   \relax #1\xint_dothis\xint_firstofthree\fi
    \ifcat \relax #1\xint_dothis\xint_secondofthree\fi
    \ifnum\xint_c_ix<1\string#1 \xint_dothis\xint_secondofthree\fi
    \if    (#1\xint_dothis      \xint_secondofthree\fi %)
    \xint_orthat \xint_thirdofthree
    \xint_c_
    {\BNE_prec_tacit *}%
    \BNE_scanop_a
    #1%
}%
\catcode`* 12
\def\BNE_scanop_a #1#2%
{%
    \expandafter\BNE_scanop_b\expandafter#1\romannumeral`&&@#2%
}%
\def\BNE_scanop_b #1#2%
{%
    \unless\ifcat#2\relax
           \ifcsname BNE_itself_#1#2\endcsname
           \BNE_scanop_c
    \fi\fi
    \BNE_foundop_a #1#2%
}%
\def\BNE_scanop_c #1#2#3#4#5% #1#2=\fi\fi
{%
    #1#2%
    \expandafter\BNE_scanop_d\csname BNE_itself_#4#5\expandafter\endcsname
    \romannumeral`&&@%
}%
\def\BNE_scanop_d #1#2%
{%
    \unless\ifcat#2\relax
           \ifcsname BNE_itself_#1#2\endcsname
           \BNE_scanop_c
    \fi\fi
    \BNE_foundop #1#2%
}%
```

If a postfix say ?s is defined and ?r is encountered the ? will have been interpreted as a shortcut to ?s and then the r will be found with the parser (after having executed the already found postfix) now looking for another operator so the error message will be Operator? (got `r') which is doubly confusing... well, let's not dwell on that.

Update 2021/05/22, I have changed the message, as part of a systematic removal of I<something> invites, in part because xint 1.4g changed its expandable error method and now has a nice message saying xint will try to recover by itself. And now I have about 55 characters available for the message.

```
\def\BNE_foundop_a #1%
{%
    \ifcsname BNE_precedence_#1\endcsname
        \csname BNE_precedence_#1\expandafter\endcsname
        \expandafter #1%
    \else
        \expandafter\BNE_getop_a\romannumeral`&&@%
        \xint_afterfi{\XINT_expandableerror
        {Expected an operator but got `#1'. Ignoring.}}%
     \fi
}%
\def\BNE_foundop #1{\csname BNE_precedence_#1\endcsname #1}%
```

9.11 Expansion spanning; opening and closing parentheses

There was refactoring of expandable error messages at xint 1.4g and I can now use up to 55 characters, but should not really invite user to Insert something as it does not fit well with generic message saying xint will go ahead "hoping repair was complete".

At 1.6, we removed the \BNE_start, current \BNE_bareeval has its meaning rather than expanding to it as formerly.

Also here macros are defined one by one so that it is easier to understand what is happening. Formerly \BNE_tmpa defined all of them in one go (as is still the case in upstream xintexpr).

```
\def\BNE_tmpa#1{%
    \def\BNE_check##1%
    {%
        \xint_UDsignfork
          ##1{\expandafter\BNE_checkp\romannumeral`&&@#1}%
            -{\BNE_checkp##1}%
        \krof
    }%
}\expandafter\BNE_tmpa\csname BNE_op_-xii\endcsname
\def\BNE_tmpa#1{%
    \def\BNE_checkp##1##2%
    {%
        \ifcase ##1%
           \expandafter\BNE_done
        \or\expandafter#1%
        \else
           \expandafter\BNE_checkp
           \romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
        \fi
    }%
}\expandafter\BNE_tmpa\csname BNE_extra_)\endcsname
\expandafter\def\csname BNE_extra_)\endcsname{%
        \XINT_expandableerror
        {An extra ) was removed. Hit <return>, fingers crossed.}%
        \expandafter\BNE_check\romannumeral`&&@\expandafter\BNE_put_op_first
        \romannumeral`&&@\BNE_getop_legacy
}%
\let\BNE_done\space
```

```
\def\BNE_getop_legacy #1%
{%
    \expanded{\unexpanded{{#1}}\expandafter}\romannumeral`&&@\BNE_getop
}%
```

Code style left untouched at 1.6.

```
\catcode`) 11
\def\BNE_tmpa #1#2#3#4#5#6%
{%
    \def #1##1% op_(
    {%
        \expandafter #4\romannumeral`&&@\BNE_getnext
    }%
    \def #2##1% op_)
    {%
        \expanded{\unexpanded{\BNE_put_op_first{##1}}\expandafter}%
        \romannumeral`&&@\BNE_getop
    }%
    \def #3% oparen
    {%
        \expandafter #4\romannumeral`&&@\BNE_getnext
    }%
    \def #4##1% check-
    {%
        \xint_UDsignfork
            ##1{\expandafter#5\romannumeral`&&@#6}%
              -{#5##1}%
        \krof
    }%
    \def #5##1##2% checkp
    {%
        \ifcase ##1\expandafter\BNE_missing_)
        \or \csname BNE_op_##2\expandafter\endcsname
        \else
          \expandafter #5\romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
        \fi
    }%
}%
\expandafter\BNE_tmpa
    \csname BNE_op_(\expandafter\endcsname
    \csname BNE_op_)\expandafter\endcsname
    \csname BNE_oparen\expandafter\endcsname
    \csname BNE_check-_)\expandafter\endcsname
    \csname BNE_checkp_)\expandafter\endcsname
    \csname BNE_op_-xii\endcsname
\let\BNE_precedence_)\xint_c_i
\def\BNE_missing_)
   {\XINT_expandableerror{Missing ). Hit <return> to proceed.}%
    \xint_c_ \BNE_done }%
\catcode`) 12
```

9.12 The comma as binary operator

At 1.4, it is simply a union operator for 1D oples. Inserting directly here a <comma><space> separator (as in earlier releases) in accumulated result would avoid having to do it on output but to the cost of diverging from xintexpr upstream code, and to have to let the ``unpack'' (new name at 1.7) routines handle comma separated values rather than braced values.

Note bene: contiguous commas ,, will cause the parser to raise an ``unexpected token'' error on the second comma, contrarily to \xinteval which silently ignores it since its 1.4 release (see the first subsection about basic terminology in *Oples and nutples: the 1.4 terminology* in xint.pdf).

It does work to use \bnumdefinfix{,,}{\macro}{N}{N}. Then ,, acts as a custom infix operator without breaking the use of a single comma to separate inputs.

```
\def\BNE_tmpa #1#2#3#4#5%
{%
    \def #1##1% \BNE_op_,
    {%
      \expanded{\unexpanded{#2{##1}}\expandafter}%
      \romannumeral`&&@\expandafter#3\romannumeral`&&@\BNE_getnext
    }%
    \def #2##1##2##3##4{##2##3{##1##4}}% \BNE_exec_,
    \def #3##1% \BNE_check-_,
    {%
      \xint_UDsignfork
        ##1{\expandafter#4\romannumeral`&&@#5}%
          -{#4##1}%
      \krof
    }%
    \def #4##1##2% \BNE_checkp_,
    {%
      \ifnum ##1>\xint_c_iii
        \expandafter#4%
           \romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
      \else
        \expandafter##1\expandafter##2%
      \fi
    }%
}%
\expandafter\BNE_tmpa
    \csname BNE_op_,\expandafter\endcsname
    \csname BNE_exec_,\expandafter\endcsname
    \csname BNE_check-_,\expandafter\endcsname
    \csname BNE_checkp_,\expandafter\endcsname
    \csname BNE_op_-xii\endcsname
\expandafter\let\csname BNE_precedence_,\endcsname\xint_c_iii
```

9.13 The minus as prefix operator of variable precedence level

This \BNE_Op_opp caused trouble at 1.4 as it must be *f*-expandable, whereas earlier it expanded inside \csname...\endcsname context, so I could define it as

```
\if-#1\else\if0#10\else-#1\fi\fi
```

where #1 was the first token of unbraced argument but this meant at 1.4 an added \xint_firstofone here. Well let's return to sanity at 1.4a and not add the \xint_firstofone and simply default \BNE_Op_opp to \xintiiOpp, which it should have been all along! And on this occasion let's trim user documentation of complications.

The package used to need to define unary minus operator with precedences 12, 14, and 18. It also defined it at level 16 but this was unneedeed actually, no operator possibly generating usage of an op_-xvi.

At 1.5 the right precedence of powers was lowered to 17, so we now need here only 12, 14, and 17.

Due to \bnumdefinfix it is needed to support also, perhaps, the other levels 13, 15, 16, 18, ... This will be done only if necessary and is the reason why the macros \BNE_defminus_a and \BNE_defminus_b are given permanent names. In fact it is now \BNE_defbin_b which will decide to invoke or not the \BNE_defminus_a, and we activate it here only for the base precedence 12.

The \XINT_global's are absent from upstream xintexpr as it does not incorporate yet some analog to \bnumdefinfix/\bnumdefpostfix.

```
\def\BNE_defminus_b #1#2#3#4#5%
{%
    \XINT_global\def #1% \BNE_op_-<level>
    {%
      \expandafter #2\romannumeral`&&@\expandafter#3%
      \romannumeral`&&@\BNE_getnext
    }%
    \XINT_global\def #2##1##2##3% \BNE_exec_-<level>
    {%
      \expandafter ##1\expandafter ##2\expandafter
       {\expandafter{\romannumeral`&&@\BNE_Op_opp##3}}%
    }%
    \XINT_global\def #3##1% \BNE_check-_-<level>
    {%
      \xint_UDsignfork
        ##1{\expandafter #4\romannumeral`&&@#1}%
          -{#4##1}%
      \krof
    }%
    \XINT_global\def #4##1##2% \BNE_checkp_-<level>
    {%
      \ifnum ##1>#5%
        \expandafter #4%
        \romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
      \else
        \expandafter ##1\expandafter ##2%
      \fi
    }%
}%
\def\BNE_defminus_a #1%
{%
    \expandafter\BNE_defminus_b
    \csname BNE_op_-#1\expandafter\endcsname
    \csname BNE_exec_-#1\expandafter\endcsname
```

```
    \csname BNE_check-_-#1\expandafter\endcsname
    \csname BNE_checkp_-#1\expandafter\endcsname
    \csname xint_c_#1\endcsname
}%
\BNE_defminus_a {xii}%
```

9.14 The infix operators.

I could have at the 1.4 refactoring injected usage of \expanded here, but kept in sync with upstream xintexpr code. Any *x*-expandable macro can easily be converted into an *f*-expandable one using \expanded, so this is no serious limitation.

Macro names are somewhat bad and there is much risk of confusion in future maintenance of \BNE_Op_ prefix (used for \BNE_Op_add etc...; besides this should have been \BNE_Op_Add) and \BNE_op_ prefix (used for \BNE_op_+ etc...).

At 1.5 decision is made to anticipate the announced upstream change to let the power operators be right associative, matching Python behaviour. This change is simply implemented by hardcoding in \BNE_checkp_<op> the right precedence which so far, for such operators, had been identical with the left precedence (upstream has examples of direct coding without formalization). In fact the right precedence existed already as argument to \BNE_defbin_b as the precedence to assign to unary minus following <op>.

Note1: although it is easy to change the left precedence at user level, the right precedence is now more inaccessible. But on the other hand bnumexpr provides \bnumdefinfix so all is customizable at user level.

Note2: Tacit multiplication is not really a separate operator, it is the * with an elevated left precedence, which costs nothing to create and this precedence is stored in chardef token \BNE_prec_tacit.

Compared to upstream, we use here numbers as arguments to \BNE_defbin_b, and convert to roman numerals internally, also the operator macro is passed as a control sequence not as its name (and #6 and #7 are permuted in \BNE_defbin_c).

```
\def\BNE_defbin_c #1#2#3#4#5#6#7%
{%
  \XINT_global\def #1##1% \BNE_op_<op>
  {%
    \expanded{\unexpanded{#2{##1}}\expandafter}%
    \romannumeral`&&@\expandafter#3\romannumeral`&&@\BNE_getnext
  }%
  \XINT_global\def #2##1##2##3##4% \BNE_exec_<op>
  {%
    \expandafter##2\expandafter##3\expandafter
      {\expandafter{\romannumeral`&&@#7##1##4}}%
  }%
  \XINT_global\def #3##1% \BNE_check-_<op>
  {%
    \xint_UDsignfork
      ##1{\expandafter#4\romannumeral`&&@#5}%
        -{#4##1}%
    \krof
  }%
  \XINT_global\def #4##1##2% \BNE_checkp_<op>
```

```
689   {%
690     \ifnum ##1>#6%
691       \expandafter#4%
692       \romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
693     \else
694       \expandafter ##1\expandafter ##2%
695     \fi
696   }%
697 }%
698 \def\BNE_defbin_b #1#2#3#4%
699 {%
700     \expandafter\BNE_defbin_c
701     \csname BNE_op_#1\expandafter\endcsname
702     \csname BNE_exec_#1\expandafter\endcsname
703     \csname BNE_check-_#1\expandafter\endcsname
704     \csname BNE_checkp_#1\expandafter\endcsname
705     \csname BNE_op_-\romannumeral\ifnum#3>12 #3\else 12\fi
706             \expandafter\endcsname
707     \csname xint_c_\romannumeral#3\endcsname #4%
708   \XINT_global
709     \expandafter
710     \let\csname BNE_precedence_#1\expandafter\endcsname
711         \csname xint_c_\romannumeral#2\endcsname
712     \unless
713     \ifcsname BNE_exec_-\romannumeral\ifnum#3>12 #3\else 12\fi\endcsname
```

This will execute only for `#3>12` as `\BNE_exec_-xii` exists.

```
714      \expandafter\BNE_defminus_a\expandafter{\romannumeral#3}%
715     \fi
716 }%
717 \BNE_defbin_b  +   {12} {12}  \BNE_Op_add
718 \BNE_defbin_b  -   {12} {12}  \BNE_Op_sub
719 \BNE_defbin_b  *   {14} {14}  \BNE_Op_mul
720 \BNE_defbin_b  /   {14} {14}  \BNE_Op_divround
721 \BNE_defbin_b {//} {14} {14}  \BNE_Op_div
722 \BNE_defbin_b {/:} {14} {14}  \BNE_Op_mod
723 \BNE_defbin_b  ^   {18} {17}  \BNE_Op_pow
```

xintexpr uses shortcut

```
\expandafter\def\csname XINT_expr_itself_**\endcsname {^}
```

But doing it would mean that any redefinition of `^` propagates to `**`. And it creates a special case which would need consideration by `\BNE_dotheitselves`, or special restrictions to add to user documentation. Better to simply handle `**` as a full operator.

```
724 \BNE_defbin_b {**} {18} {17}  \BNE_Op_pow
725 \expandafter\def\csname BNE_itself_**\endcsname {**}%
726 \expandafter\def\csname BNE_itself_//\endcsname {//}%
727 \expandafter\def\csname BNE_itself_/:\endcsname {/:}%
728 \let\BNE_prec_tacit\xint_c_xvi
```

9.15 Extending the syntax: \bnumdefinfix, \bnumdefpostfix

9.15.1 \bnumdefinfix

#1 gives the operator characters, #2 the associated macro, #3 its left-precedence and #4 its right precedence (as integers).

The "itself" definitions are done in such a way that unambiguous abbreviations work; but in case of ambiguity the first defined operator is used.

However, if for example operator $a was defined after $ab, then although $ will use $ab which was defined first, $a will use as expected the second defined operator.

The mismatch \BNE_defminus_a vs \BNE_defbin_b is inherited from upstream, I keep it to simplify maintenance.

```
\def\bnumdefinfix #1#2#3#4%
{%
    \edef\BNE_tmpa{#1}%
    \edef\BNE_tmpa{\xint_zapspaces_o\BNE_tmpa}%
    \edef\BNE_tmpL{\the\numexpr#3\relax}%
    \edef\BNE_tmpL{\ifnum\BNE_tmpL<4 4\else\ifnum\BNE_tmpL<23 \BNE_tmpL\else 22\fi\fi}%
    \edef\BNE_tmpR{\the\numexpr#4\relax}%
    \edef\BNE_tmpR{\ifnum\BNE_tmpR<4 4\else\ifnum\BNE_tmpR<23 \BNE_tmpR\else 22\fi\fi}%
    \BNE_defbin_b \BNE_tmpa\BNE_tmpL\BNE_tmpR #2%
    \expandafter\BNE_dotheitselves\BNE_tmpa\relax
  \ifxintverbose
    \PackageInfo{bnumexpr}{infix operator \BNE_tmpa\space
    \ifxintglobaldefs globally \fi
        does
        \unexpanded{#2}\MessageBreak with precedences \BNE_tmpL, \BNE_tmpR;}%
  \fi
}%
\def\BNE_dotheitselves#1#2%
{%
    \if#2\relax\expandafter\xint_gobble_ii
    \else
  \XINT_global
      \expandafter\edef\csname BNE_itself_#1#2\endcsname{#1#2}%
      \unless\ifcsname BNE_precedence_#1\endcsname
  \XINT_global
        \expandafter\edef\csname BNE_precedence_#1\endcsname
                        {\csname BNE_precedence_\BNE_tmpa\endcsname}%
  \XINT_global
        \expandafter\odef\csname BNE_op_#1\endcsname
                        {\csname BNE_op_\BNE_tmpa\endcsname}%
      \fi
    \fi
    \BNE_dotheitselves{#1#2}%
}%
```

9.15.2 \bnumdefpostfix

Support macros for postfix operators only need to be *x*-expandable.

```
\def\bnumdefpostfix #1#2#3%
```

```
{%
    \edef\BNE_tmpa{#1}%
    \edef\BNE_tmpa{\xint_zapspaces_o\BNE_tmpa}%
    \edef\BNE_tmpL{\the\numexpr#3\relax}%
    \edef\BNE_tmpL{\ifnum\BNE_tmpL<4 4\else\ifnum\BNE_tmpL<23 \BNE_tmpL\else 22\fi\fi}%
  \XINT_global
    \expandafter\let\csname BNE_precedence_\BNE_tmpa\expandafter\endcsname
                     \csname xint_c_\romannumeral\BNE_tmpL\endcsname
  \XINT_global
    \expandafter\def\csname BNE_op_\BNE_tmpa\endcsname ##1%
    {%
        \expandafter\BNE_put_op_first
        \expanded{{{#2##1}}\expandafter}\romannumeral`&&@\BNE_getop
    }%
    \expandafter\BNE_dotheitselves\BNE_tmpa\relax
  \ifxintverbose
    \PackageInfo{bnumexpr}{postfix operator \BNE_tmpa\space
    \ifxintglobaldefs globally \fi
        does \unexpanded{#2}\MessageBreak
        with precedence \BNE_tmpL;}%
  \fi
}%
```

9.16 ! as postfix factorial operator

```
\bnumdefpostfix{!}{\BNE_Op_fac}{20}%
```

9.17 Cleanup

```
\let\BNE_tmpa\relax \let\BNE_tmpb\relax \let\BNE_tmpc\relax
\let\BNE_tmpR\relax \let\BNE_tmpL\relax
\BNErestorecatcodesendinput%
```