The bnumexpr package

JEAN-FRANÇOIS BURNOL
jfbu (at) free (dot) fr

Package version: 1.5 (2021/05/17); documentation date: 2021/05/17.
From source file bnumexpr.dtx. Time-stamp: <17-05-2021 14:16:23 CEST>.

Contents

1 \bnumeval (\thebnumexpr), \evaltohex

LaTeX Package bnumexpr provides `\thebnumexpr`⟨*expression*⟩`\relax`: it is analogous to `\the\numexpr`⟨*expression*⟩`\relax`, with these extensions:

- it allows arbitrarily big integers,
- it computes powers (with either `**` or `^` as infix operator),
- it computes factorials (with `!` as postfix operator),
- it has an operator `//` for floored division and `/:` for the associated modulo (like `%` in Python which we can't use for obvious reasons),

- 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,
- also the underscore `_` may be used as visual digit separator,
- comma separated expressions are allowed,
- syntax is customizable and extendible.

There is also a more core-level `\bnumexpr...\relax` construct[2], which expands to a self-contained unit, rather than to explicit digit tokens (and commas). See section 6 for some related information.

There is also the alternative interface `\bnumeval{⟨expression⟩}`, where the expression is fetched as braced argument.

And there is `\evaltohex{⟨expression⟩}` which does the same as `\bnumeval` but with a conversion to hexadecimal notation of the (possibly comma separated) output. Hexadecimal input uses the `"` prefix.

This package parser is a scaled-down variant of `\xintiiexpr` from package xintexpr, dropping support for nested structures, functions, variables, booleans, etc..., but incorporating by default support for hexadecimal input as xintbinhex will be automatically loaded.

The ε-TeX extensions are required, this is the default on all modern installations for `latex|pdflatex` and also for `xelatex|lualatex`.

Further, at 1.4 (2021/05/12) the `\expanded` primitive is required. It is available in all engines since TeXLive 2019.

2 Examples

With certain languages, Babel with PDFLaTeX may make some characters active, for example the `!` with the French language. It must then be input as `\string!`.

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

```
\bnethe \bnumexpr (13_8089_1090-300_1890_2902)*(1083_1908_3901-109_8290_3⤶
890)\relax
                  -2787514672889976289932
```

```
\bnumeval {(92_874_927_979**5-31_9792_7979**6)/30!}
                 -4006240736596543944035189
```

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

```
\bnumeval {13^50//12^50, 13^50/:12^50}
       54, 6505562879010990257452210486837601617945679471401685 53
```

---

[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] Since 1.4, one can use `\bnumexpr ...\relax` directly in typesetting context, it is not mandatory to prefix it with `\bnethe` or to use `\thebnumexpr`.

```
\bnumeval {13^50/12^50, 12^50}
       55, 910043815000214977332758527534256632492715260325658624

\bnumeval {(1^10+2^10+3^10+4^10+5^10+6^10+7^10+8^10+9^10)^3}
                     118685075462698981700620828125

\bnumeval {100!/36^100}
                                 219

\bnumeval {"0010*"0100*"1000*"A0000, 16^(1+2+3+4)*10}
                     10995116277760, 10995116277760

\evaltohex {"7FFFFFFF+1, "0400^3, "ABCDEF*"0000FEDCBA, 1234}
                 80000000, 40000000, AB0A74EF03A6, 4D2

\bnumeval {"\evaltohex {12345678}FFFF, 000012345679*16**4-1}
                        809086418943, 809086418943
```

3 The custom package option and \bnumsetup

Package bnumexpr needs that some *big integer engine* provides the macros doing the actual computations. By default, it loads package xintcore (a subset of xint) and uses \bnumsetup in the following way:

```
\usepackage{xintcore}
\bnumsetup{add=\xintiiAdd, sub=\xintiiSub, mul=\xintiiMul,
           divround=\xintiiDivRound, div=\xintiiDivFloor,
           mod=\xintiiMod, pow=\xintiiPow, fac=\xintiiFac,
           opp=\xintiiOpp}
```

If using \bnumsetup, it is not necessary to specify all keys, for example one can do `\bnumsetup{mul=\MySlowerMul}`, and only multiplication will be changed.

Naturally it is up to the user to load the appropriate package for the alternative macros.

The macros serving as custom user replacements[3] must be *f*-expandable, except for the computation of factorials, which only has to be *x*-expandable.[4]

Macro \bnumsetup can be used multiple times in the same document, thus allowing to switch math engines or to remap operators to some other arithmetic macros of the same math engine. Its effect obeys the local scope.[5]

---

[3] The replacement macros will by default receive arguments composed of explicit digit tokens, with no leading zeros, with at most one leading minus sign and no plus sign.

The format of these arguments will depend on what the other customized macros do. For example if `opp=\foo` is used and the custom `\foo` inserts a `+` when taking the opposite of a negative number, then the other custom macros for arithmetic (and the `\foo` macro itself) must be able to handle arguments starting optionally with such a `+`.

[4] Prior to `1.4`, only *x*-expandability was required. It is easy however to use an `\expanded` based wrapper to convert *x*-expandable macros into *f*-expandable ones.

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

The hexadecimal input via the " prefix is converted internally to decimal notation using \xintHexToDec from package xintbinhex, and customization is possible via redefinition of \bnumhextodec, whose default is to be an alias to \xintHexToDec.

The final conversion back to hexadecimal done by \evaltohex is handled by \bnumprintonetohex which defaults to \xintDecToHex.

These two steps can thus be customized as will. But the loading of package xintbinhex can not be canceled.

4 \bnumprintone, \bnumprintonetohex, \bnumprintonesep

The computed values are printed one by one, separated by a comma and a space (this is customizable as \bnumprintonesep), and each value being handed over to \bnumprintone. By default this does nothing else than producing its argument as is, it can be redefined at will (perhaps using macros such as in section 7 to handle the case of very long numbers).

There is also \bnumprintonetohex which is used by \evaltohex (this is its sole difference from \bnumeval). Its default definition makes it an alias to \xintDecToHex from package xintbinhex.

5 Example of customization: let the syntax 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 trailing [0]
\newcommand\myFac[1]{\xintFac{#1}}%     produces trailing [0]
\bnumsetup{add=\myIrrAdd, sub=\myIrrSub, mul=\myIrrMul,
           divround=\myDiv, div=\myDivFloor,
           mod=\myMod, pow=\myPow, fac=\myFac}%
% % if any operation happened, the result is already irreducible
% % (except power of non-irreducible) so this is overhead:
% \let\bnumprintone\xintIrr
% % but it is safe way to get rid of the trailing [0]
% % else we have to take care of case with no [0] because of no operations
% % well let's do it:
\makeatletter
\def\myPrintOne#1{\@myPrintOne#1[0]\relax}
\def\@myPrintOne#1[0]#2\relax{#1}
\let\bnumprintone\myPrintOne
\makeatother
```

```
\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}
                         1/2417851639229258349412352
```

This last example is computed differently than it would be with `xintexpr 1.4f` because `bnumexpr 1.5` already applies right associativity to powers, whereas `xintexpr 1.4f` still applies left associativity.

Note also that the above changes break `\evaltohex` whose output routine uses by default `\xintDecToHex` which will choke on fractional input. However it is not difficult to write a macro applying separately to numerator and denominator.

Computations with fractions quickly give birth to big results, see section 7 on how to modify `\bnumprintone` to coerce TeX into wrapping numbers too long for the available width.

6 Differences from \numexpr

Apart from the extension to big integers (i.e. exceeding the TeX limit at 2147483647), 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\myVar{\thebnumexpr1+2\relax}`, the computation is of course done at time of the `\edef`. But one is also allowed to do `\edef\myVar{\bnumexpr1+2\relax}` which prepares `\myVar` as a macro which can be inserted in other bnumexpr expressions and behave there as a self-contained pre-computed unit triggering tacit multiplication, or be typeset directly if inserted in the typesetting stream.[6] There is no analog with `\numexpr` as `\edef\myVar{\numexpr1+2\relax}` does not pre-compute anything and furthermore `\the\numexpr2\myVar\relax` in typesetting flow then triggers the `You can't use `\numexpr' in horizontal mode` error.
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. `\bnumexpr -(1+1)\relax` is legal contrarily to `\numexpr -(1+1)\relax` which raises an error,
6. `\numexpr 2\cnta\relax` is illegal (with `\cnta` a `\count`-variable.) But `\bnumexpr 2\cnta\relax` is perfectly legal and will do the tacit multiplication,
7. more generally, tacit multiplication applies in front of parenthesized sub-expressions, or sub `\bnumexpr...\relax` (or `\numexpr...\relax`), or also after parentheses in front of numbers,
8. the underscore `_` is accepted within the digits composing a number and is silently ignored by `\bnumexpr`.

As hinted above `\bnumexpr...\relax` differs from `\thebnumexpr...\relax` as the latter expands to explicit digit tokens, but the former expands to a private self-contained format which can serve as sub-unit in other expressions, or be used inside `\edef`. Since 1.4 the former idiom can also be inserted directly inside the typesetting stream, or be written out to an external file where it will expand to some control sequences, braces, and character tokens, all with their standard catcodes.

One can use `\numexpr...\relax` as a sub-unit in `\bnumexpr...\relax` but the reverse does not apply: it would either cause an error or an anticipated end to the `\numexpr` which will think having hit a `\relax`.

An important thing to keep in mind is that if one has a calculation whose result is a small integer, acceptable by TeX in `\ifnum` or count assignments, this integer produced by `\thebnumexpr` is not self-delimiting, contrarily to a `\numexpr...\relax` construct: the situation is exactly as with a `\the\numexpr...\relax`, thus one may need to terminate the number to avoid premature expansion of following tokens; for example with the `\space` control sequence. When using `\bnumeval{...}` syntax as in

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

---

[6] Prior to 1.4, one would have had to use `\bnethe \myVar` for typesetting, or `\bnumeval {\myVar }`.

the end of line will (under the normal LaTeX configuration) insert a terminating space token. Again, here `\bnumeval{...}` must produce an integer acceptable to TeX, i.e. at most 2147483647 in absolute value.

7 Printing big numbers

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 }%
% \printnumber thus first ``fully'' expands its argument.
```

```
\thebnumexpr 1000!\relax= 4023872600770937735437024339230039857193748642
107146325437999104299385123986290205920442084869694048004799886101971960
586316668729948085589013238296699445909974245040870737599188236277271887
325197795059509952761208749754624970436014182780946464962910563938874378
864873371191810458257836478499770124766328898359557354325131853239584630
755574091142624174743493475534286465766116677973966688202912073791438537
195882498081268678383745597317461360853795345242215865932019280908782973
084313928444032812315586110369768013573042161687476096758713483120254785
893207671691324484262361314125087802080002616831510273418279777047846358
681701643650241536913982812648102130927612448963599287051149649754199093
422215668325720808213331861168115536158365469840467089756029009505376164
758477284218896796462449451607653534081989013854424879849599533191017233
555566021394503997362807501378376153071277619268490343526252000158885351
473316117021039681759215109077880193931781141945452572238655414610628921
879602238389714760885062768629671466746975629112340824392081601537808898
939645182632436716167621791689097799119037540312746222899880051954444142
820121873617459926429565817466283029555702990243241531816172104658320367
869061172601587835207515162842255402651704833042261439742869330616908979
684825901254583271682264580665267699586526822728070757813918581788896522
081643483448259932660433676601769996128318607883861502794659551311565520
360939881806121385586003014356945272242063446317974605946825731037900840
244324384656572450144028218852524709351906209290231364932734975655139587
205596542287497740114133469627154228458623773875382304838656889764619273
838149001407673104466402598994902222217659043399018860185665264850617997
023561938970178600408118897299183110211712298459016419210688843871218556
461249607987229085192968193723886426148396573822911231250241866493531439
701374285319266498753372189406942814341185201580141233448280150513996942
901534830776445690990731524332782882698646027898643211390835062170950025
973898635542771967428222487575867657523442202075736305694988250879689281
627538488633969099598262809561214509948717012445164612603790293091208890
869420285106401821543994571568059418727489980942547421735824010636774045
957417851608292301353580818400969963725242305608559037006242712434169090
041536901059339838357779394109700277534720000000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
00
```

8 Expression syntax and its customizability

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 in TEX syntax for number assignments, i.e. using a `"` prefix and only uppercase letters `ABCDEF`.

Different computations may be separated by commas. 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:[7]

```
\bnumeval {2^3^4, 2^(3^4)}
```

2417851639229258349412352, 2417851639229258349412352

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 to the operators `+`, `-`, `*`, `/`, `//`, `/:`, `**` and `^` by custom macros,
- or even more completely via `\bnumdefinfix` and `\bnumdefpostfix` which allow to add new operators to the syntax! (or overwrite existing ones...)

9 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`,

---

[7] It had been announced at xintexpr `1.4` that probably in future power operator would become right-associative, so we experiment it here in bnumexpr in advance.

- 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`.[8]

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` computes fractions!

10 \bnumdefinfix

It is possible to define infix binary operators of one's own choosing.[9] The syntax is

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

---

[8] 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.

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

{⟨*operator*⟩} The characters for the operator, they may be letters or non-letters, and must not be active or among the special characters `\`, `{`, `}`, `#` and `%`. Also, spaces will be removed.[10,11,12]

{⟨*\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 under only these two possibilities:

- 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}
% 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
```

---

[10] The _ 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.

[11] It is actually possible to use `#` as an operator name or a character in such a name but the definition with `\bnumdefinfix` must then be done either with `\string#` or `####`...

[12] Active characters must be prefixed by `\string` both at the time of the definition and at the time of use. It is probably better to use a toggle which will turn off the activity, both at time of definition and at time of use.

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.[13]

```
\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
```

11 \bnumdefpostfix

It is possible to define postfix unary operators of one's own choosing.[14] 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.

{⟨\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 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
```

---

[13] As `!=` is indeed defined out-of-the-box in the xintexpr syntax, the `3! == 8` issue applies with `\xinteval` and perhaps I should add it to the user documentation, as warning.

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

```
\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 :: :: :}
                    612716271751406378427089874211
```

```
\bnumeval {123456789.r}
                         123456789123456789123456789
```

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

```
\bnumeval {123456789.r}
              123456789123456789123456789123456789123456789
```

12 Readme

```
| Source:  bnumexpr.dtx
| Version: v1.5, 2021/05/17 (doc: 2021/05/17)
| Author:  Jean-Francois Burnol
| Info:    Expressions with big integers
| License: LPPL 1.3c

bnumexpr usage
==============

The LaTeX package `bnumexpr` allows expandable computations with
integers and the four infix operators `+`, `-`, `*`, `/` using the
expression syntax familiar from the `\numexpr` e-TeX parser, with
these extensions:

- arbitrarily big integers,
- floored division `//`,
- associated modulo `/:`,
- power operators `^` and `**`,
- factorial post-fix operator `!`,
- comma separated expressions,
- the space character as well as the underscore may serve
  to separate groups of digits,
- optional conversion of output to hexadecimal,
- customizability and extendibility of the syntax.

The expression parser is a scaled-down variant from the
`\xintiiexpr...\relax`
parser from package [xintexpr](http://ctan.org/pkg/xintexpr).

To support hexadecimal input and output, the package
[xintbinhex](http://ctan.org/pkg/xint) is loaded automatically.

The package loads by default [xintcore](http://ctan.org/pkg/xint)
but the option _custom_ together with macro `\bnumexprsetup` allow to map
the syntax elements to macros from an alternative big integer
expandable engine of the user own choosing,
and then [xintcore](http://ctan.org/pkg/xint) is not loaded.

Installation
============

Use your installation manager to install or update `bnumexpr`.

Else, obtain `bnumexpr.dtx`, from CTAN:

> <http://www.ctan.org/pkg/bnumexpr>

Run `"etex bnumexpr.dtx"` to extract these files:

`bnumexpr.sty`
  : this is the style file.

`README.md`

`bnumexprchanges.tex`
  : change history.

`bnumexpr.tex`
  : can be used to generate the documentation

To generate the documentation:
  - with latex+dvipdfmx: `"latex bnumexpr.tex"` (thrice) then
    `"dvipdfmx bnumexpr.dvi"`.

  - with pdflatex: `"pdflatex bnumexpr.tex"` (thrice).

In both cases files `README.md` and `bnumexprchanges.tex` must
be located in the same repertory as `bnumexpr.tex` and `bnumexpr.dtx`.

Without `bnumexpr.tex`:

  - `"pdflatex bnumexpr.dtx"` (thrice) extracts all files and
    simultaneously generates the pdf documentation.

Final steps:

- move files to appropriate destination:

           bnumexpr.sty   --> TDS:tex/latex/bnumexpr/

           bnumexpr.dtx   --> TDS:source/latex/bnumexpr/

           bnumexpr.pdf   --> TDS:doc/latex/bnumexpr/
              README.me   --> TDS:doc/latex/bnumexpr/

- discard auxiliary files generated during compilation.

License
=======

Copyright (C) 2014-2021 by Jean-Francois Burnol

| This Work may be distributed and/or modified under the
| conditions of the LaTeX Project Public License 1.3c.
| This version of this license is in

>   <http://www.latex-project.org/lppl/lppl-1-3c.txt>

| and version 1.3 or later is part of all distributions of
| LaTeX version 2005/12/01 or later.

This Work has the LPPL maintenance status "author-maintained".

The Author and Maintainer of this Work is Jean-Francois Burnol.

This Work consists of the main source file and its derived files

    bnumexpr.dtx, bnumexpr.sty, bnumexpr.pdf, bnumexpr.tex,
    bnumexprchanges.tex, README.md
```

13 Changes

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.
- 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.

14 bnumexpr implementation

Contents

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 unconditional loading of 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 `File ended while scanning use of \BNE_print_c` error because `\BNEprint` makes the tacit requirement that the 1D ople to output has at least one item.

At 1.5, right-associativity is implemented for powers in anticipation of upstream, and the customizability and extendibility of the package is made total via added `\bnumdefinfix` and `\bnumdefpostfix`.

14.1 Package identification

```
\NeedsTeXFormat{LaTeX2e}%
\ProvidesPackage{bnumexpr}[2021/05/17 v1.5 Expressions with big integers (JFB)]%
```

14.2 Load unconditionally xintbinhex

Newly done at 1.4. Formerly, bnumexpr had no dependency if loaded with option custom. But for 1.4 release I have decided to add unconditional support for hexadecimal notation.

Let's require the most recent xint date at time of writing. We should check for availability of `\expanded` but well.

```
\RequirePackage{xintbinhex}[2021/05/10]%
```

14.3 Package options

```
\def\BNEtmpa {0}%
\DeclareOption {custom}{\def\BNEtmpa {1}}%
\ProcessOptions\relax
```

14.4 \bnumsetup and conditional loading of xintcore

The keys should have been Add, Sub, . . . , not add, sub, . . . , so internally macros `\BNE_Op_Add` etc. . . macro names would be used, but well, let's simply live with this.

`\bnumsetup` replaces at 1.5 deprecated `\bnumexprsetup` which is kept as an alias.

```
{\catcode`! 3 \catcode`_ 11 %
  \gdef\bnumsetup #1{\BNE_parsekeys #1,=!,}%
  \gdef\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 the space inserted by \on@line.

```
     \expandafter\xint_firstofone}%
  \fi
  \BNE_parsekeys
  }%
  \gdef\BNE_parsedone #1\BNE_parsekeys {}%
}%
\let\bnumexprsetup\bnumsetup
\if0\BNEtmpa\expandafter\@secondoftwo\fi
\@gobble{%
    \RequirePackage{xintcore}[2021/05/10]%
    \bnumsetup{add=\xintiiAdd, sub=\xintiiSub, mul=\xintiiMul,
                   divround=\xintiiDivRound, div=\xintiiDivFloor,
                   mod=\xintiiMod, pow=\xintiiPow, fac=\xintiiFac,
                   opp=\xintiiOpp}%
}%
```

14.5 Activate usual xint catcodes for code source

```
\edef\BNErestorecatcodes{\XINTrestorecatcodes}%
\XINTsetcatcodes%
```

Strangely those three are not defined in xintkernel.sty, but only in xint.sty

```
\long\def\xint_firstofthree  #1#2#3{#1}%
\long\def\xint_secondofthree #1#2#3{#2}%
\long\def\xint_thirdofthree  #1#2#3{#3}%
```

For the mechanism of \bnumdefinfix we need [1] precedence levels to be available as \chardef's. xintkernel already provides 0-10, 12, 14, 16, 18, 20, 22. Admittedly they could be created only dynamically, and then I would not have to set a cap at 22, but well, that's already a large supported range for a functionality nobody will use, as nobody probably uses the package to start with.

.. [1] 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_xvii 17
\chardef\xint_c_xix  19
\chardef\xint_c_xxi  21
```

14.6 \bnumexpr, \thebnumexpr, \bnethe, \bnumeval

```
\def\XINTfstop {\noexpand\XINTfstop}%
\def\bnumexpr  {\romannumeral0\bnumexpro}%
\def\bnumexpro {\expandafter\BNE_wrap\romannumeral0\bnebareeval }%
\def\BNE_wrap  {\XINTfstop\BNEprint.}%
\def\bnumeval #1%
   {\expanded\expandafter\BNEprint\expandafter.\romannumeral0\bnebareeval#1\relax}%
\def\evaltohex #1%
   {\expanded\expandafter\BNEprinthex\expandafter.\romannumeral0\bnebareeval#1\relax}%
```

```
\def\thebnumexpr
   {\expanded\expandafter\BNEprint\expandafter.\romannumeral0\bnebareeval}%
\def\bnebareeval{\BNE_start}%
\def\bnethe#1{\expanded\expandafter\xint_gobble_i\romannumeral`&&@#1}%
\protected\def\BNEprint.#1{{\BNE_print#1.}}%
\def\BNE_print#1{\bnumprintone{#1}\expandafter\BNE_print_a\string}%
\def\BNE_print_a#1{\unless\if#1.\expandafter\BNE_print_b\fi}%
\def\BNE_print_b
   {\expandafter\BNE_print_c\expandafter{\expandafter\xint_gobble_i\string}}%
\def\BNE_print_c#1{\bnumprintonesep\bnumprintone{#1}\expandafter\BNE_print_a\string}%
\protected\def\BNEprinthex.#1{{\BNE_printhex#1.}}%
\def\BNE_printhex#1{\bnumprintonetohex{#1}\expandafter\BNE_printhex_a\string}%
\def\BNE_printhex_a#1{\unless\if#1.\expandafter\BNE_printhex_b\fi}%
\def\BNE_printhex_b
   {\expandafter\BNE_printhex_c\expandafter{\expandafter\xint_gobble_i\string}}%
\def\BNE_printhex_c#1{\bnumprintonesep\bnumprintonetohex{#1}\expandafter\BNE_printhex_a\string
\let\bnumprintone\xint_firstofone
\let\bnumprintonetohex\xintDecToHex
\def\bnumprintonesep{, }%
```

14.7 \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.

```
\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%
}%
\def\BNE_foundprematureend\BNE_getnextfork #1{{}\xint_c_\relax}%
\def\BNE_subexpr #1.#2%
{%
    \expanded{\unexpanded{{#2}}\expandafter}\romannumeral`&&@\BNE_getop
}%
\def\BNE_countetc\BNE_getnextfork#1%
{%
    \if0\ifx\count#11\fi
        \ifx\dimen#11\fi
        \ifx\numexpr#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.

```
\def\BNE_getnextfork #1{%
    \if#1+\xint_dothis \BNE_getnext_a \fi
    \if#1-\xint_dothis {{}{}-}\fi
    \if#1(\xint_dothis {{}\xint_c_ii^v (}\fi
    \xint_orthat {\BNE_scan_number #1}%
}%
```

14.8 Parsing an integer in decimal or hexadecimal notation

```
\def\BNE_scan_number #1%
{%
    \if "#1\xint_dothis \BNE_scanhex\fi
    \ifnum \xint_c_ix<1\string#1 \xint_dothis \BNE_startint\fi
    \xint_orthat \BNE_notadigit #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

``Digit? (got `##')``

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.

```
\def\BNE_notadigit#1%
{%
    \expandafter\BNE_scan_number
    \romannumeral`&&@\XINT_expandableerror{Digit? (got `#1'). Hit I<RET><digit>}%
}%
\def\BNE_startint #1%
{%
    \if #10\expandafter\BNE_gobz_a\else\expandafter\BNE_scanint_a\fi #1%
}%
\def\BNE_scanint_a #1#2%
    {\expanded\bgroup{{\iffalse}}\fi #1%
     \expandafter\BNE_scanint_main\romannumeral`&&@#2}%
```

```
\def\BNE_gobz_a #1#2%
    {\expanded\bgroup{{\iffalse}}\fi
     \expandafter\BNE_gobz_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_next\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
{%
    \iffalse{{{\fi}}\expandafter}\romannumeral`&&@\BNE_getop_a#2%
}%
\def\BNE_scanint_next #1\BNE_scanint_again
{%
    \if    _#1\xint_dothis\BNE_scanint_again\fi
    \xint_orthat
    {\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\BNE_getop_a#1}%
}%
\def\BNE_gobz_scanint_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_gobz_scanint_hit_cs\fi
    \ifnum\xint_c_x<1\string#1 \else\expandafter\BNE_gobz_scanint_next\fi
    #1\BNE_scanint_again
}%
\def\BNE_gobz_scanint_again #1%
{%
    \expandafter\BNE_gobz_scanint_main\romannumeral`&&@#1%
}%
```

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

```
\def\BNE_gobz_scanint_hit_cs\ifnum#1\fi#2\BNE_scanint_again
{%
    0\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\BNE_getop_a#2%
}%
\def\BNE_gobz_scanint_next #1\BNE_scanint_again
{%
    \if    _#1\xint_dothis\BNE_gobz_scanint_again\fi
    \if    0#1\xint_dothis\BNE_gobz_scanint_again\fi
    \xint_orthat
    {0\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\BNE_getop_a#1}%
}%
\def\BNE_hex_in #1.%
{%
    \expanded{{{\bnumhextodec{#1}}}\expandafter}\romannumeral`&&@\BNE_getop
}%
\let\bnumhextodec\xintHexToDec
```

Upstream (until 1.4f) had a long-standing bug in its hexadecimal input, which was inherited here at 1.4: the \BNE_scanhex triggered \BNE_scanhex_a which then grabbed an unexpanded token and used it as is in an \ifcat... this made syntax such as "\foo broken. Fixed here at 1.5.

And there was a further long-standing bug in upstream (from 1.2m to 1.4f) about leading hexadecimal zeros not being trimmed. This was inherited here at 1.4. Fixed also at 1.5.

```
\def\BNE_scanhex #1#2% #1="
{%
    \expandafter\BNE_hex_in\expanded\bgroup
    \expandafter\BNE_scanhexgobz_a\romannumeral`&&@#2%
}%
\def\BNE_scanhexgobz_a #1%
{%
    \ifcat #1\relax0.\iffalse{\fi\expandafter}\expandafter\xint_gobble_i\fi
    \BNE_scanhexgobz_aa #1%
}%
\def\BNE_scanhexgobz_aa #1%
{%
    \if\ifnum`#1>`0
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1%
       \xint_dothis\BNE_scanhex_b
    \fi
    \if 0#1\xint_dothis\BNE_scanhexgobz_bgob\fi
    \if _#1\xint_dothis\BNE_scanhexgobz_bgob\fi
    \if .#1\xint_dothis\BNE_scanhexgobz_toII\fi
    \xint_orthat
     {\XINT_expandableerror
        {HexDigit was expected but saw `#1'. Using 0, hit <RET>}%
      0.>;\iffalse{\fi}}%
    #1%
}%
\def\BNE_scanhexgobz_bgob #1#2%
{%
    \expandafter\BNE_scanhexgobz_a\romannumeral`&&@#2%
}%
\def\BNE_scanhex_a #1%
{%
    \ifcat #1\relax.\iffalse{\fi\expandafter}\expandafter\xint_gobble_i\fi
    \BNE_scanhex_aa #1%
}%
\def\BNE_scanhex_aa #1%
{%
    \if\ifnum`#1>`/
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1%
       \expandafter\BNE_scanhex_b
    \else
       \if _#1\xint_dothis{\expandafter\BNE_scanhex_bgob}\fi
       \xint_orthat {.\iffalse{\fi\expandafter}}%
    \fi
    #1%
}%
\def\BNE_scanhex_b #1#2%
{%
    #1\expandafter\BNE_scanhex_a\romannumeral`&&@#2%
}%
\def\BNE_scanhex_bgob #1#2%
{%
    \expandafter\BNE_scanhex_a\romannumeral`&&@#2%
}%
```

14.9 \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.

```
\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
        {Operator? (got `#1'). Hit I<RET><operator>}}%
     \fi
}%
\def\BNE_foundop #1{\csname BNE_precedence_#1\endcsname #1}%
```

14.10 Expansion spanning; opening and closing parentheses

```
\def\BNE_tmpa #1#2#3#4#5%
{%
    \def#1% start
    {%
        \expandafter#2\romannumeral`&&@\BNE_getnext
    }%
    \def#2##1% check
    {%
        \xint_UDsignfork
          ##1{\expandafter#3\romannumeral`&&@#4}%
            -{#3##1}%
        \krof
    }%
    \def#3##1##2% checkp
    {%
```

```
        \ifcase ##1%
           \expandafter\BNE_done
        \or\expandafter#5%
        \else
           \expandafter#3\romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
        \fi
    }%
    \def#5%
    {%
        \XINT_expandableerror
        {An extra ) has been removed. Hit <RET>, fingers crossed.}%
        \expandafter#2\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
}%
\expandafter\BNE_tmpa
    \csname BNE_start\expandafter\endcsname
    \csname BNE_check\expandafter\endcsname
    \csname BNE_checkp\expandafter\endcsname
    \csname BNE_op_-xii\expandafter\endcsname
    \csname BNE_extra_)\endcsname
\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 <RET> to proceed}%
    \xint_c_ \BNE_done }%
\catcode`) 12
```

14.11 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 \evaltohex output routine handle comma separated values rather than braced values.

```
\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
```

14.12 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` are inexistent in upstream at 1.4f 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}%
```

14.13 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>
  {%
    \ifnum ##1>#6%
      \expandafter#4%
      \romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
    \else
      \expandafter ##1\expandafter ##2%
    \fi
  }%
}%
\def\BNE_defbin_b #1#2#3#4%
{%
    \expandafter\BNE_defbin_c
    \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 BNE_op_-\romannumeral\ifnum#3>12 #3\else 12\fi
            \expandafter\endcsname
    \csname xint_c_\romannumeral#3\endcsname #4%
  \XINT_global
    \expandafter
    \let\csname BNE_precedence_#1\expandafter\endcsname
        \csname xint_c_\romannumeral#2\endcsname
    \unless
    \ifcsname BNE_exec_-\romannumeral\ifnum#3>12 #3\else 12\fi\endcsname
```

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

```
     \expandafter\BNE_defminus_a\expandafter{\romannumeral#3}%
    \fi
}%
\BNE_defbin_b  +   {12} {12}  \BNE_Op_add
\BNE_defbin_b  -   {12} {12}  \BNE_Op_sub
\BNE_defbin_b  *   {14} {14}  \BNE_Op_mul
\BNE_defbin_b  /   {14} {14}  \BNE_Op_divround
\BNE_defbin_b {//} {14} {14}  \BNE_Op_div
\BNE_defbin_b {/:} {14} {14}  \BNE_Op_mod
\BNE_defbin_b  ^   {18} {17}  \BNE_Op_pow
```

At upstream, we can use shortcut

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

but it means then that any redefinition of ^ propagates to `**`, besides 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.

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

14.14 \bnumdefinfix: extending the syntax

`#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}%
}%
```

14.15 \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
}%
```

14.16 ! as postfix factorial operator

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

14.17 Cleanup

```
\let\BNEtmpa\relax
\let\BNE_tmpa\relax \let\BNE_tmpb\relax \let\BNE_tmpc\relax
\let\BNE_tmpR\relax \let\BNE_tmpL\relax
\BNErestorecatcodes%
```