

The `bnumexpr` package

JEAN-FRANÇOIS BURNOL

jfbu (at) free (dot) fr

Package version: 1.6a (2025/09/07)

From source file `bnumexpr.dtx` of 07-09-2025 at 12:03:52 CEST

1	<code>\bnumeval</code>	1
2	Dependencies	2
3	Examples	2
4	Customizing how output is “printed out”	4
4.1	Printing big numbers	4
4.2	<code>\bnumprintone</code> , <code>\bnumprintonesep</code>	5
4.3	<code>\bnumprintonehex</code> , <code>\bnumprintoneoct</code> , <code>\bnumprintonebin</code>	6
5	Babel-active characters are not a problem!	7
6	Fine print (not needed to read this for regular use)	7
6.1	The <code>\bnumsetup</code> command	7
6.2	Example of customization: let's handle fractions!	8
6.3	Significant differences between <code>\bnumexpr</code> and <code>\numexpr</code>	9
6.4	For the expert user: expression syntax and its customizability	10
7	Changes	14
8	License	17
9	Commented source code	18

1 `\bnumeval`

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

Recent \TeX has `\inteval`, which is a slim wrapper for $\varepsilon\text{-}\text{\TeX}$'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`:

- It allows arbitrarily big integers, whereas `\inteval` is limited to a maximal input equal to $2^{31} - 1$, or hexadecimal `7FFFFFFF`.

2 Dependencies

- It recognizes `**` and `^` as infix operator for powers,
- It recognizes `!` as postfix operator for the factorials,
- The new operator `//` computes floored division with `/:` being the operator for the associated remainder (the operator `/` computes rounded division),
- In addition to the \TeX prefixes `'` and `"` for octal and hexadecimal, it recognizes `0b`, `0o` and `0x` for binary, octal, and hexadecimal,
- The space character is ignored¹ 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,
- Braced material `{...}` encountered in the expression is automatically unbraced,
- Comma separated expressions are allowed,
- Some idiosyncrasies of `\numexpr` such as `\inteval{-(1)}` causing an error are avoided,
- Syntax is fully customizable and extensible.

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

2 Dependencies

`bnumexpr` is a \TeX 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 \TeX .

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 or change the built-in precedences. See [subsection 6.4](#).

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

¹It 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.

3 Examples

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
```

```
\bnumeval{92_874_927_979 ** 5 - 31_9792_7979 ** 6}
-1062666812478332115682721163376486501493666601082871642222
```

```
\bnethe \bnumexpr 10!, 20!, 30!\relax
3628800, 2432902008176640000, 2652528598121910586363084800000000
```

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}
55.910043815000214977332758527534256632492715260325658624
```

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

118685075462698981700620828125

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

Let's check hexadecimal input:

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

And also hexadecimal output:

```
\bnumeval[h]{"7F_FFF_FFF+1, 0x0400^3, "ABCDEF*"0000FEDCBA, 1234}
800000000, 400000000, AB0A74EF03A6.4D2
```

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

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

4 Customizing how output is “printed out”

```
\bnumeval[b]{'75316420 * 0o44445555}
10001100010110111111000010110111100011111010000

\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!}}}
```

$\bnumeval{1000!} = 40238726007709377354370243392300398571937486421071463$
254379991042993851239862902059204420848696940480047998861019719605863166
687299480855890132382966994459099742450408707375991882362772718873251977
950595099527612087497546249704360141827809464649629105639388743788648733
711918104582578364784997701247663288983595573543251318532395846307555740
911426241747434934755342864657661166779739666882029120737914385371958824
980812686783837455973174613608537953452422158659320192809087829730843139
284440328123155861103697680135730421616874760967587134831202547858932076
716913244842623613141250878020800026168315102734182797770478463586817016
436502415369139828126481021309276124489635992870511496497541990934222156
683257208082133318611681155361583654698404670897560290095053761647584772
842188967964624494516076535340819890138544248798495995331910172335555660
213945039973628075013783761530712776192684903435262520001588853514733161
170210396817592151090778801939317811419454525722386554146106289218796022
383897147608850627686296714667469756291123408243920816015378088989396451
826324367161676217916890977991190375403127462228998800519544441428201218
736174599264295658174662830295557029902432415318161721046583203678690611
726015878352075151628422554026517048330422614397428693306169089796848259

4 Customizing how output is “printed out”

```
012545832716822645806652676995865268227280707578139185817888965220816434
834482599326604336766017699961283186078838615027946595513115655203609398
818061213855860030143569452722420634463179746059468257310379008402443243
846565724501440282188525247093519062092902313649327349756551395872055965
422874977401141334696271542284586237738753823048386568897646192738381490
014076731044664025989949022222176590433990188601856652648506179970235619
389701786004081188972991831102117122984590164192106888438712185564612496
079872290851929681937238864261483965738229112312502418664935314397013742
853192664987533721894069428143411852015801412334482801505139969429015348
307764456909907315243327828826986460278986432113908350621709500259738986
35542771967428224875758676575234422020757363056949882508796892816275384
886339690995982628095612145099487170124451646126037902930912088908694202
851064018215439945715680594187274899809425474217358240106367740459574178
516082923013535808184009699637252423056085590370062427124341690900415369
010593398383577793941097002775347200000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
```

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 \TeX '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.

4 Customizing how output is “printed out”

T_EX-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 T_EXnical 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 L^AT_EX, or Plain ϵ -T_EX (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

T_EX-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 T_EXnicity) 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, 7888609052210118054117285652827862296732064351090230047702789
306640625

T_EX-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`, `\bnumprintoneoct`, `\bnumprintonebin`

When `\bnumeval` is exerted with `[h]`, `[o]` or `[b]` it does not use `\bnumprintone` but one of `\bnumprintonehex`, `\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 output to use the `0x` prefix. This is very easy:

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

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

T_EX-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 PDF_WTeX. 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 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. More precisely:
 - a) The macro for computing factorials only has to be *x*-expandable.
 - b) Note that any *x*-expandable macro can be wrapped into an *f*-expandable one, using `\expanded`.

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

- c) If `\bnumprintonehex` is redefined and becomes `\protected` then the macro for converting to hexadecimal (value of key `dectohex`) only has to be x-expandable, and similarly for conversion to octal and binary.
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.

6.2 Example of customization: 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
```


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

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

764966897/5000000000

```
\bnumeval{2^-3^4}
```

1/2417851639229258349412352

Computations with fractions quickly give birth to big results, see [subsection 4.1](#) on how to modify `\bnumprint` to coerce \TeX into wrapping numbers too long for the available width.

6.3 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`
`\the\bnumexpr 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 not 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. `\the\bnumexpr -(1+1)\relax` is legal contrarily to `\the\numexpr -(1+1)\relax` which raises an error,

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

6. `\thebnumexpr 2+-(1+1)\relax` is legal contrarily to `\the\numexpr 2+-(1+1)\relax` which raises an error,
7. `\the\numexpr 2\cnta\relax` is illegal (with `\cnta` a `\count-variable`.) But `\thebnumexpr 2\cnta\relax` is perfectly legal and will do the tacit multiplication,
8. 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,
9. 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.

6.4 For the expert user: expression syntax and its customizability

6.4.1 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 in \TeX syntax for number assignments, i.e. using a `"` prefix and only uppercase letters `ABCDEF`. Release 1.6 added support for the `0b`, `0o`, `0x` and `'` prefixes.

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)}
```

2417851639229258349412352, 2417851639229258349412352

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

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! (or overwrite existing ones...)

6.4.2 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 `Rlast` 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
<code>+, -</code>	12	12
<code>*, /, //, /:</code>	14	14
tacit <code>*</code>	16	14
<code>**</code> , <code>^</code>	18	17
<code>!</code>	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!

6.4.3 \bnumdefinfix

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

```
\bnumdefinfix{<operator>}{<\macro>}{<L-prec>}{<R-prec>}
```

²The `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.

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

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

{<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 will be removed.^{4,5,6}

{<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}
```

% 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.⁷

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

1, 154

⁴The `_` 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.

⁵It 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 `####...`

⁶Active characters (except if they expand to innocent ones) must be prefixed by `\string` at the time of the definition of the operator whose names will use them. Same at time of use, except if they are Babel active then (new with 1.6) they need no precaution at time of use.

⁷With `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 `=`

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

```
\bnumeval{100 ++ -10 ^ 3, (100 - 10)^3, 2 ^ 5 ++ 3, 2^(5+3)}  
729000, 729000, 256, 256
```

6.4.4 \bnumdefpostfix

It is possible to define postfix unary operators of one's own choosing.⁸ 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  
\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
```

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` by default supports no infix operator 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.

⁸The effect of `\bnumdefpostfix` is global if under `\xintglobaldefstrue` setting.

7 Changes

```
\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
```

7 Changes

1.6a (2025/09/07) bugfix: 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 auto-recovering error to signal it. It will be removed at next release. Use new `\bnumeval[h]`.
- `\bnumexprsetup` was deprecated at 1.5 and renamed into `\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 fixes:

- 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.

7 Changes

- 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 `\exp\andafter"\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 `\xintglobaldeftrue` 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.

7 Changes

- 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 `LATEX 3` 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`).

8 License

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 \LaTeX 3. 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

Copyright © 2014-2022, 2025 Jean-François 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-François 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

9 Commented source code

Package identification	9.1, p. 19
Load xintkernel	9.2, p. 19
Save catcode regime and switch to our own	9.3, p. 19
Load optionally xintcore and xintbinhex	9.4, p. 19
\bnumsetup	9.5, p. 19
Some extra constants needed for user defined precedences	9.6, p. 20
\bnumexpr, \bnethe, \bnumeval	9.7, p. 21
\BNE_getnext	9.8, p. 23
Parsing decimal, hexadecimal, octal, and binary	9.9, p. 25
\BNE_getop	9.10, p. 31
Expansion spanning; opening and closing parentheses	9.11, p. 32
The comma as binary operator	9.12, p. 34
The minus as prefix operator of variable precedence level	9.13, p. 35
The infix operators.	9.14, p. 36
Extending the syntax: \bnumdefinfix, \bnumdefpostfix	9.15, p. 38
! as postfix factorial operator	9.16, p. 39
Cleanup	9.17, p. 39

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 ogle is by itself now legal, and can be injected (comma separated) in an expression, keeping it invariant, however `\bnumeval{}` ends in a `Paragraph ended by \before \BNE_print_c was complete` error because `\BNEprint` makes the tacit requirement that the 1D ogle to output has at least one item.

9.1 Package identification

```
1 \NeedsTeXFormat{LaTeX2e}%
2 \ProvidesPackage{bnumexpr}[2025/09/07 v1.6a 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.

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

9.3 Save catcode regime and switch to our own

```
4 \edef\BNErestorecatcodesendinput{\XINTrestorecatcodes\noexpand\endinput}%
5 \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`?

```
6 \def\BNE_tmpa{1}\def\BNE_tmpb{1}%
7 \DeclareOption{custom}{\def\BNE_tmpa{0}\def\BNE_tmpb{0}}%
8 \DeclareOption{customcore}{\def\BNE_tmpa{0}}%
9 \DeclareOption{custombinhex}{\def\BNE_tmpb{0}}%
10 \ProcessOptions\relax
11 \if1\BNE_tmpa\RequirePackage{xintcore}[2025/09/05]\fi
12 \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.

```
13 \catcode\! 3
14 \def\bnumsetup #1{\BNE_parsekeys #1,=!,}%
15 \def\BNE_parsekeys #1=#2#3,%
```

```

16  {%
17    \ifx!#2\expandafter\BNE_parsedone\fi
18  \XINT_global
19    \expandafter
20    \let\csname BNE_Op_\xint_zapspaces #1 \xint_gobble_i\endcsname%
21    =#2%
22  \ifxintverbose
23    \PackageInfo{bnumexpr}{assigned
24    \ifxintglobaldefs globally \fi
25    \string#2 to \xint_zapspaces #1 \xint_gobble_i\MessageBreak
Workaround for the space inserted by \on@line.
26    \expandafter\xint_firstofone}%
27  \fi
28  \BNE_parsekeys
29  }%
30 \def\BNE_parsedone #1\BNE_parsekeys {}%
31 \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.

```

32 \bnumsetup{%
33   add = \xintiiAdd, sub = \xintiiSub, opp = \xintiiOpp,
34   mul = \xintiiMul, pow = \xintiiPow, fac = \xintiiFac,
35   div = \xintiiDivFloor, mod = \xintiiMod, divround = \xintiiDivRound,
36   hextodec=\xintHexToDec, octtodec=\xintOctToDec, bintodec=\xintBinToDec,
37   dectohex=\xintDecToHex, dectooct=\xintDecToOct, dectobin=\xintDecToBin,
38 }%

```

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 `c2heckp_<op>` macros and could have been there explicit digit tokens but we will use the `\xint_c_...` `\char`-tokens.

```

39 \chardef\xint_c_xi 11
40 \chardef\xint_c_xiii 13
41 \chardef\xint_c_xv 15
42 \chardef\xint_c_xix 19
43 \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 `\xint\iiexpr...\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 `xintexpr`-expression 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` in favor of `\bnumeval[h]`.

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...

```
44 \def\XINTfstop {\noexpand\XINTfstop}%
45 \def\bnumexpr {\romannumeral0\bnumexpro}%
46 \def\bnumexpro {\csname BNE_wrap\expandafter\endcsname
47 \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 `\xint\eval`. 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`.

A consequence is that `\bnumexpr...\relax` can be used as sub-unit in `\xinteval` and conversely `\xintiexpr...\relax` in `\bnumeval`, as long as it does not have nested structures via bracketed inputs, which are not supported by `bnumexpr`'s syntax. But why would one do such things? Also this can only work as long as internal storage of intermediate result by `\bnumeval` is a sub-case of the way it is done for `\xinteval`.

```
48 \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`.

See above about the strange ```.'` inserted by `\BNE_wrap` and gobbled here. We also define a non `\protected` variant without the extra full stop, it will serve for `\bnumeval` 1 (and `\thebnumexpr`).

```
49 \protected\def\BNEprint.#1{{\BNE_print#1.}}%
50 \def\BNEprint_#1{{\BNE_print#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.

```
51 \def\bnethe{\expanded\expandafter\xint_gobble_i\romannumeral`&&@}%
52 \def\thebnumexpr{\expanded\csname BNEprint_\expandafter\endcsname
53 \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.

```
54 \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. Thus, no need for some `\BNEprinthe` et al. here.

```
55 \expandafter\def\csname BNEprint_[h]\endcsname#1{{\BNE_printhe#1.}}%
56 \expandafter\def\csname BNEprint_[o]\endcsname#1{{\BNE_printoct#1.}}%
57 \expandafter\def\csname BNEprint_[b]\endcsname#1{{\BNE_printbin#1.}}%
58 \expandafter\let\csname BNEprint_[]\endcsname\BNEprint_
```

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

```
59 \def\bnumeval #1#{\expanded\bnumeval_a{#1}}%
60 \def\bnumeval_a#1#2{%
61   \csname BNEprint_\xint_zapspaces #1 \xint_gobble_i\expandafter
62   \endcsname\romannumeral0\BNE_bareeval#2\relax
63 }%
```

This is deprecated at 1.6 and raises an expandable error.

```
64 \def\evaltohex {\expanded
65   \XINT_expandableerror{\evaltohex is DEPRECATED, use \bnumeval with [h]}%
66   \bnumeval_a[h]}%
67 }%
```

This code is more compact at 1.6 than at 1.5.

```
68 \def\BNE_print#1{%
69   \bnumpritone{#1}\expandafter\BNE_print_a\string
70 }%
71 \def\BNE_print_a#1{%
72   \if#1.\BNE_print_z\fi\bnumpritone\sep
73   \expandafter\BNE_print\expandafter{\iffalse}\fi
74 }%
75 \def\BNE_print_z\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`.

```

76 \def\BNE_printhex#1{%
77   \expandafter\bnumprintonehex
78   \expandafter{\romannumeral`&&\BNE_Op_dectohex{#1}}%
79   \expandafter\BNE_printhex_a\string
80 }%
81 \def\BNE_printhex_a#1{%
82   \if#1.\BNE_print_z\fi\bnumprintonesep
83   \expandafter\BNE_printhex\expandafter{\iffalse}\fi
84 }%

```

Octal and binary added at 1.6.

```

85 \def\BNE_printoct#1{%
86   \expandafter\bnumprintoneoct
87   \expandafter{\romannumeral`&&\BNE_Op_dectooct{#1}}%
88   \expandafter\BNE_printoct_a\string
89 }%
90 \def\BNE_printoct_a#1{%
91   \if#1.\BNE_print_z\fi\bnumprintonesep
92   \expandafter\BNE_printoct\expandafter{\iffalse}\fi
93 }%
94 \def\BNE_printbin#1{%
95   \expandafter\bnumprintonebin
96   \expandafter{\romannumeral`&&\BNE_Op_dectobin{#1}}%
97   \expandafter\BNE_printbin_a\string
98 }%
99 \def\BNE_printbin_a#1{%
100   \if#1.\BNE_print_z\fi\bnumprintonesep
101   \expandafter\BNE_printbin\expandafter{\iffalse}\fi
102 }%
103 \let\bnumprintone \xint_firstofone
104 \let\bnumprintonehex\xint_firstofone
105 \let\bnumprintoneoct\xint_firstofone
106 \let\bnumprintonebin\xint_firstofone
107 \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 like syntax elements, as will be seen later on.

```

108 \def\BNE_getnext #1%
109 {%
110   \expandafter\BNE_put_op_first\romannumeral`&&%
111   \expandafter\BNE_getnext_a\romannumeral`&&#1%
112 }%
113 \def\BNE_put_op_first #1#2#3{#2#3{#1}}%

```

```

114 \def\BNE_getnext_a #1%
115 {%
116   \ifx\relax #1\xint_dothis\BNE_foundprematureend\fi
117   \ifx\XINTfstop#1\xint_dothis\BNE_subexpr\fi
118   \ifcat\relax#1\xint_dothis\BNE_countetc\fi
119   \xint_orthat{}\BNE_getnextfork #1%
120 }%
121 \def\BNE_foundprematureend\BNE_getnextfork #1{{}\xint_c_\relax}%
122 \def\BNE_subexpr #1.#2%
123 {%
124   \expanded{\unexpanded{{#2}}\expandafter}\romannumeral`&&\BNE_getop
125 }%

```

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

```

126 \def\BNE_countetc\BNE_getnextfork#1%
127 {%
128   \if0\ifx\count#1\fi
129   \ifx\numexpr#1\fi
130   \ifx\catcode#1\fi
131   \ifx\dimen#1\fi
132   \ifx\dimexpr#1\fi
133   \ifx\skip#1\fi
134   \ifx\glueexpr#1\fi
135   \ifx\fontdimen#1\fi
136   \ifx\ht#1\fi
137   \ifx\dp#1\fi
138   \ifx\wd#1\fi
139   \ifx\fontcharht#1\fi
140   \ifx\fontcharwd#1\fi
141   \ifx\fontcharhp#1\fi
142   \ifx\fontcharic#1\fi
143   0\expandafter\BNE_fetch_as_number\fi
144   \expandafter\BNE_getnext_a\number #1%
145 }%
146 \def\BNE_fetch_as_number
147   \expandafter\BNE_getnext_a\number #1%
148 {%
149   \expanded{{{\number#1}}\expandafter}\romannumeral`&&\BNE_getop
150 }%

```

In the case of hitting a (, previous release inserted directly a \BNE_oparen. But the expansion architecture imported from upstream \xintiexpr 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.

```

151 \def\BNE_getnextfork #1{%
152   \if#1-\xint_dothis {{}}{-}\fi
153   \if#1(\xint_dothis {{}}\xint_c_ii^v (}\fi
154   \ifnum\xint_c_ix<1\string#1 \xint_dothis {\BNE_startint#1}\fi
155   \xint_orthat {\BNE_getnextfork_a #1}%
156 }%

```



```

157 \def\BNE_getnextfork_a #1{%
158   \if#1\xint_dothis \BNE_getnext_a \fi
159   \if#1+\xint_dothis \BNE_getnext_a \fi
160   \if#1'\xint_dothis \BNE_startoct\fi
161   \if#1"\xint_dothis \BNE_starthex\fi
162   \xint_orthat {\BNE_unexpected #1}%
163 }%

```

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.

```

164 \def\BNE_unexpected#1%
165 {%
166   \XINT_expandableerror{Unexpected token `#1'. Ignoring.}\BNE_getnext_a
167 }%

```

9.9 Parsing decimal, hexadecimal, octal, and binary

Somewhat refactored at 1.6 compared to upstream 1.4m. Fix the case of an underscore `_` as first character in input.

```

168 \def\BNE_startint #1%
169 {%
170   \if #10\expandafter\BNE_scanint_gobz_a\else\expandafter\BNE_scanint_a\fi #1%
171 }%
172 \def\BNE_wrapint_before{\expandafter{\romannumeral`&&@iffalse}\fi}%
173 \def\BNE_wrapint_after{\iffalse{{\fi}}}%
174 \def\BNE_scanint_a #1#2%
175   {\expandafter\BNE_wrapint_before
176    \expanded\bgroup{\iffalse}\fi #1%
177    \expandafter\BNE_scanint_main\romannumeral`&&@#2}%
178 \def\BNE_scanint_gobz_a #1#2%
179   {\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. 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`.

```

180 \def\BNE_scanint_gobz_b #1%
181 {%
182   \ifx b#1\xint_dothis \BNE_startbin \fi
183   \ifx o#1\xint_dothis \BNE_startoct \fi
184   \ifx x#1\xint_dothis \BNE_starthex \fi
185   \xint_orthat {\BNE_scanint_gobz_c #1}%
186 }%
187 \def\BNE_scanint_gobz_c #1%

```

```

188 {%
189   \expandafter\BNE_wrapint_before\expanded\bgroup{\iffalse}\fi
190   \BNE_scanint_gobz_main#1%
191 }%
192 \def\BNE_scanint_main #1%
193 {%
194   \ifcat \relax #1\expandafter\BNE_scanint_hit_cs \fi
195   \ifnum\xint_c_ix<1\string#1 \else\expandafter\BNE_scanint_checkagain\fi
196   #1\BNE_scanint_again
197 }%
198 \def\BNE_scanint_again #1%
199 {%
200   \expandafter\BNE_scanint_main\romannumeral`&&@#1%
201 }%

```

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

```

202 \def\BNE_scanint_hit_cs \ifnum#1\fi#2\BNE_scanint_again
203 {%
204   \expandafter\BNE_wrapint_after\romannumeral`&&@\BNE_getop_a#2%
205 }%
206 \def\BNE_scanint_checkagain #1\BNE_scanint_again
207 {%
208   \if _#1\BNE_scanint_checkagain_skip\fi
209   \expandafter\BNE_wrapint_after\romannumeral`&&@\BNE_getop_a#1%
210 }%
211 \def\BNE_scanint_checkagain_skip#1#2\BNE_getop_a#3{#1\BNE_scanint_again}%
212 \def\BNE_scanint_gobz_main #1%
213 {%
214   \ifcat \relax #1\expandafter\BNE_scanint_gobz_hit_cs\fi
215   \ifnum\xint_c_x<1\string#1 \else\expandafter\BNE_scanint_gobz_checkagain\fi
216   #1\BNE_scanint_again
217 }%
218 \def\BNE_scanint_gobz_again #1%
219 {%
220   \expandafter\BNE_scanint_gobz_main\romannumeral`&&@#1%
221 }%

```

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. Nevertheless this means one brace-stripping less.

```

222 \def\BNE_scanint_gobz_hit_cs\ifnum#1\fi#2\BNE_scanint_again
223 {%
224   0\expandafter\BNE_wrapint_after\romannumeral`&&@\BNE_getop_a#2%
225 }%

```

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.

```

226 \def\BNE_scanint_gobz_checkagain #1\BNE_scanint_again
227 {%
228   \if _#1\xint_dothis\BNE_scanint_gobz_again\fi
229   \if 0#1\xint_dothis\BNE_scanint_gobz_again\fi
230   \xint_orthat

```

```
231 {0\expandafter\BNE_wrapint_after\romannumeral`&&\BNE_getop_a#1}%
232 }%
```

1.5 backported from [xintexpr](#) two bugfixes relative to parsing hexadecimal input. One bug had `\BNE_scanhex_a` grab an unexpanded token and used it as is in an `\ifcat...` this made syntax such as `"\foo` broken. The other bug was about leading hexadecimal zeros not being trimmed.

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

```
233 \def\BNE_starthex #1%
234 {%
235   \expandafter\BNE_starthex_i\romannumeral`&&#1%
236 }%
237 \def\BNE_starthex_i #1%
238 {%
239   \if #10\expandafter\BNE_scanhex_gobz_a\else\expandafter\BNE_scanhex_a\fi #1%
240 }%
241 \def\BNE_wraphex_before{\expandafter{\expandafter{
242   \romannumeral`&&\iffalse}}\fi\BNE_Op_hextodec}%
243 \def\BNE_wraphex_after{\iffalse{\fi}}}%
244 \def\BNE_scanhex_a #1#2%
245   {\expandafter\BNE_wraphex_before
246   \expanded\bgroup{\iffalse}\fi #1%
247   \expandafter\BNE_scanhex_main\romannumeral`&&#2}%
248 \def\BNE_scanhex_gobz_a #1#2%
249   {\expandafter\BNE_wraphex_before
250   \expanded\bgroup{\iffalse}\fi
251   \expandafter\BNE_scanhex_gobz_main\romannumeral`&&#2}%

```

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.

```
252 \def\BNE_scanhex_main #1%
253 {%
254   \ifcat \relax #1\expandafter\BNE_scanhex_hit_cs \fi
255   \if\ifnum`#1>`/
256     \ifnum`#1>`9
257     \ifnum`#1>`@
258     \ifnum`#1>`F
259     0\else1\fi\else0\fi\else1\fi\else0\fi 1\else
260     \expandafter\BNE_scanhex_checkagain\fi
261     #1\BNE_scanhex_again
262 }%
263 \def\BNE_scanhex_again #1%
264 {%
265   \expandafter\BNE_scanhex_main\romannumeral`&&#1%
266 }%
267 \def\BNE_scanhex_hit_cs #1\BNE_scanhex_checkagain\fi#2\BNE_scanhex_again
268 {%
269   \expandafter\BNE_wraphex_after\romannumeral`&&\BNE_getop_a#2%
270 }%
```

```

271 \def\BNE_scanhex_checkagain #1\BNE_scanhex_again
272 {%
273   \if_#1\BNE_scanhex_checkagain_skip\fi
274   \expandafter\BNE_wraphex_after\romannumeral`&&\BNE_getop_a#1%
275 }%

#1 is \fi, #3 is underscore.
276 \def\BNE_scanhex_checkagain_skip#1#2\BNE_getop_a#3{#1\BNE_scanhex_again}%
277 \def\BNE_scanhex_gobz_main #1%
278 {%
279   \ifcat \relax #1\expandafter\BNE_scanhex_gobz_hit_cs\fi
280   \if\ifnum`#1>`0
281     \ifnum`#1>`9
282       \ifnum`#1>`@
283         \ifnum`#1>`F
284           0\else1\fi\else0\fi\else1\fi\else0\fi 1\else
285           \expandafter\BNE_scanhex_gobz_checkagain\fi
286           #1\BNE_scanhex_again
287 }%
288 \def\BNE_scanhex_gobz_again #1%
289 {%
290   \expandafter\BNE_scanhex_gobz_main\romannumeral`&&@#1%
291 }%
292 \def\BNE_scanhex_gobz_hit_cs#1\BNE_scanhex_gobz_checkagain\fi#2\BNE_scanhex_again
293 {%
294   0\expandafter\BNE_wraphex_after\romannumeral`&&\BNE_getop_a#2%
295 }%
296 \def\BNE_scanhex_gobz_checkagain #1\BNE_scanhex_again
297 {%
298   \if_#1\xint_dothis\BNE_scanhex_gobz_again\fi
299   \if_0#1\xint_dothis\BNE_scanhex_gobz_again\fi
300   \xint_orthat
301   {0\expandafter\BNE_wraphex_after\romannumeral`&&\BNE_getop_a#1}%
302 }%

Added at 1.6. Exact same code skeleton as for hexadecimal and decimal input. Leading
zeros are removed.
303 \def\BNE_startoct #1%
304 {%
305   \expandafter\BNE_startoct_i\romannumeral`&&@#1%
306 }%
307 \def\BNE_startoct_i #1%
308 {%
309   \if #10\expandafter\BNE_scanoct_gobz_a\else\expandafter\BNE_scanoct_a\fi #1%
310 }%
311 \def\BNE_wrapoct_before{\expandafter{\expandafter{
312   \romannumeral`&&\iffalse}}\fi\BNE_0p_octtodec}%
313 \def\BNE_wrapoct_after{\iffalse{{{ \fi}}}}}%
314 \def\BNE_scanoct_a #1#2%
315   {\expandafter\BNE_wrapoct_before
316   \expanded\bgroup{\iffalse}\fi #1%
317   \expandafter\BNE_scanoct_main\romannumeral`&&@#2}%
318 \def\BNE_scanoct_gobz_a #1#2%

```

```

319     {\expandafter\BNE_wrapoct_before
320     \expanded\bgroup{\iffalse}\fi
321     \expandafter\BNE_scanoct_gobz_main\romannumeral`&&@#2}%
322 \def\BNE_scanoct_main #1%
323 {%
324     \ifcat \relax #1\expandafter\BNE_scanoct_hit_cs \fi
325     \if\ifnum`#1>`/ \ifnum`#1>`7 0\else1\fi\else0\fi 1\else
326     \expandafter\BNE_scanoct_checkagain\fi
327     #1\BNE_scanoct_again
328 }%
329 \def\BNE_scanoct_again #1%
330 {%
331     \expandafter\BNE_scanoct_main\romannumeral`&&@#1%
332 }%
333 \def\BNE_scanoct_hit_cs #1\BNE_scanoct_checkagain\fi#2\BNE_scanoct_again
334 {%
335     \expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#2%
336 }%
337 \def\BNE_scanoct_checkagain #1\BNE_scanoct_again
338 {%
339     \if _#1\BNE_scanoct_checkagain_skip\fi
340     \expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#1%
341 }%
342 #1 is \fi, #3 is underscore.
342 \def\BNE_scanoct_checkagain_skip#1#2\BNE_getop_a#3{#1\BNE_scanoct_again}%
343 \def\BNE_scanoct_gobz_main #1%
344 {%
345     \ifcat \relax #1\expandafter\BNE_scanoct_gobz_hit_cs\fi
346     \if\ifnum`#1>`0 \ifnum`#1>`7 0\else1\fi\else0\fi 1\else
347     \expandafter\BNE_scanoct_gobz_checkagain\fi
348     #1\BNE_scanoct_again
349 }%
350 \def\BNE_scanoct_gobz_again #1%
351 {%
352     \expandafter\BNE_scanoct_gobz_main\romannumeral`&&@#1%
353 }%
354 \def\BNE_scanoct_gobz_hit_cs#1\BNE_scanoct_gobz_checkagain\fi#2\BNE_scanoct_again
355 {%
356     0\expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#2%
357 }%
358 \def\BNE_scanoct_gobz_checkagain #1\BNE_scanoct_again
359 {%
360     \if _#1\xint_dothis\BNE_scanoct_gobz_again\fi
361     \if 0#1\xint_dothis\BNE_scanoct_gobz_again\fi
362     \xint_orthat
363     {0\expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#1}%
364 }%

```

Added at 1.6. Exact same code skeleton as for octal and hexadecimal, based upon the one for decimal input.

```

365 \def\BNE_startbin #1%
366 {%

```

```

367     \expandafter\BNE_startbin_i\romannumeral`&&@#1%
368 }%
369 \def\BNE_startbin_i #1%
370 {%
371     \if #10\expandafter\BNE_scanbin_gobz_a\else\expandafter\BNE_scanbin_a\fi #1%
372 }%
373 \def\BNE_wrapbin_before{\expandafter{\expandafter{%
374     \romannumeral`&&@\iffalse}}\fi\BNE_Op_bintodec}%
375 \def\BNE_wrapbin_after{\iffalse{{{ \fi}}}}}%
376 \def\BNE_scanbin_a #1#2%
377     {\expandafter\BNE_wrapbin_before
378     \expanded\bgroup{\iffalse}\fi #1%
379     \expandafter\BNE_scanbin_main\romannumeral`&&@#2}%
380 \def\BNE_scanbin_gobz_a #1#2%
381     {\expandafter\BNE_wrapbin_before
382     \expanded\bgroup{\iffalse}\fi
383     \expandafter\BNE_scanbin_gobz_main\romannumeral`&&@#2}%
384 \def\BNE_scanbin_main #1%
385 {%
386     \ifcat \relax #1\expandafter\BNE_scanbin_hit_cs \fi
387     \if1\if0#11\else\if1#11\else0\fi\fi\else
388         \expandafter\BNE_scanbin_checkagain\fi
389     #1\BNE_scanbin_again
390 }%
391 \def\BNE_scanbin_again #1%
392 {%
393     \expandafter\BNE_scanbin_main\romannumeral`&&@#1%
394 }%
395 \def\BNE_scanbin_hit_cs #1\BNE_scanbin_checkagain\fi#2\BNE_scanbin_again
396 {%
397     \expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#2%
398 }%
399 \def\BNE_scanbin_checkagain #1\BNE_scanbin_again
400 {%
401     \if _#1\BNE_scanbin_checkagain_skip\fi
402     \expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#1%
403 }%
404
405 #1 is \fi, #3 is underscore.
404 \def\BNE_scanbin_checkagain_skip#1#2\BNE_getop_a#3{#1\BNE_scanbin_again}%
405 \def\BNE_scanbin_gobz_main #1%
406 {%
407     \ifcat \relax #1\expandafter\BNE_scanbin_gobz_hit_cs\fi
408     \if1#1\else\expandafter\BNE_scanbin_gobz_checkagain\fi
409     #1\BNE_scanbin_again
410 }%
411 \def\BNE_scanbin_gobz_again #1%
412 {%
413     \expandafter\BNE_scanbin_gobz_main\romannumeral`&&@#1%
414 }%
415 \def\BNE_scanbin_gobz_hit_cs#1\BNE_scanbin_gobz_checkagain\fi#2\BNE_scanbin_again
416 {%
417     0\expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#2%

```

```

418 }%
419 \def\BNE_scanbin_gobz_checkagain #1\BNE_scanbin_again
420 {%
421   \if _#1\xint_dothis\BNE_scanbin_gobz_again\fi
422   \if 0#1\xint_dothis\BNE_scanbin_gobz_again\fi
423   \xint_orthat
424   {0\expandafter\BNE_wrapbin_after\romannumeral`&&\BNE_getop_a#1}%
425 }%

```

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.

```

426 \def\BNE_getop #1%
427 {%
428   \expandafter\BNE_getop_a\romannumeral`&&@#1%
429 }%
430 \catcode`* 11
431 \def\BNE_getop_a #1%
432 {%
433   \ifx \relax #1\xint_dothis\xint_firstofthree\fi
434   \ifcat \relax #1\xint_dothis\xint_secondofthree\fi
435   \ifnum\xint_c_ix<1\string#1 \xint_dothis\xint_secondofthree\fi
436   \if (#1\xint_dothis \xint_secondofthree\fi %)
437     \xint_orthat \xint_thirdofthree
438     \xint_c_
439     {\BNE_prec_tacit *}%
440     \BNE_scanop_a
441     #1%
442 }%
443 \catcode`* 12
444 \def\BNE_scanop_a #1#2%
445 {%
446   \expandafter\BNE_scanop_b\expandafter#1\romannumeral`&&@#2%
447 }%
448 \def\BNE_scanop_b #1#2%
449 {%
450   \unless\ifcat#2\relax
451     \ifcsname BNE_itself_#1#2\endcsname
452     \BNE_scanop_c
453   \fi\fi
454   \BNE_foundop_a #1#2%
455 }%
456 \def\BNE_scanop_c #1#2#3#4#5% #1#2=\fi\fi
457 {%
458   #1#2%
459   \expandafter\BNE_scanop_d\csname BNE_itself_#4#5\expandafter\endcsname

```

```

460 \romannumeral`&&@%
461 }%
462 \def\BNE_scanop_d #1#2%
463 {%
464 \unless\ifcat#2\relax
465 \ifcsname BNE_itself_#1#2\endcsname
466 \BNE_scanop_c
467 \fi\fi
468 \BNE_foundop #1#2%
469 }%

```

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.

```

470 \def\BNE_foundop_a #1%
471 {%
472 \ifcsname BNE_precedence_#1\endcsname
473 \csname BNE_precedence_#1\expandafter\endcsname
474 \expandafter #1%
475 \else
476 \expandafter\BNE_getop_a\romannumeral`&&@%
477 \xint_afterfi{\XINT_expandableerror
478 {Expected an operator but got `#1'. Ignoring.}}%
479 \fi
480 }%
481 \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`).

```

482 \def\BNE_tmpa#1{%
483 \def\BNE_check##1%
484 {%
485 \xint_UDsignfork
486 ##1{\expandafter\BNE_checkp\romannumeral`&&@#1}%
487 -{\BNE_checkp##1}%
488 \krof
489 }%
490 }\expandafter\BNE_tmpa\csname BNE_op_-xii\endcsname

```



```

491 \def\BNE_tmpa#1{%
492   \def\BNE_checkp##1##2%
493   {%
494     \ifcase ##1%
495       \expandafter\BNE_done
496     \or\expandafter#1%
497     \else
498       \expandafter\BNE_checkp
499       \romannumeral`&&\csname BNE_op_##2\expandafter\endcsname
500     \fi
501   }%
502 }\expandafter\BNE_tmpa\csname BNE_extra_\endcsname
503 \expandafter\def\csname BNE_extra_\endcsname{%
504   \XINT_expandableerror
505   {An extra ) was removed. Hit <return>, fingers crossed.}%
506   \expandafter\BNE_check\romannumeral`&&\expandafter\BNE_put_op_first
507   \romannumeral`&&\BNE_getop_legacy
508 }%
509 \let\BNE_done\space
510 \def\BNE_getop_legacy #1%
511 {%
512   \expanded{\unexpanded{{#1}}\expandafter}\romannumeral`&&\BNE_getop
513 }%

```

Code style left untouched at 1.6.

```

514 \catcode`) 11
515 \def\BNE_tmpa #1#2#3#4#5#6%
516 {%
517   \def #1##1% op_(
518   {%
519     \expandafter #4\romannumeral`&&\BNE_getnext
520   }%
521   \def #2##1% op_)
522   {%
523     \expanded{\unexpanded{\BNE_put_op_first{##1}}\expandafter}%
524     \romannumeral`&&\BNE_getop
525   }%
526   \def #3% oparen
527   {%
528     \expandafter #4\romannumeral`&&\BNE_getnext
529   }%
530   \def #4##1% check-
531   {%
532     \xint_UDsignfork
533     ##1{\expandafter#5\romannumeral`&&#6}%
534     -{#5##1}%
535     \krof
536   }%
537   \def #5##1##2% checkp
538   {%
539     \ifcase ##1\expandafter\BNE_missing_)
540     \or \csname BNE_op_##2\expandafter\endcsname
541     \else

```

```

542     \expandafter #5\romannumeral`&&\csname BNE_op_#2\expandafter\endcsname
543     \fi
544 }%
545 }%
546 \expandafter\BNE_tmpa
547     \csname BNE_op_(\expandafter\endcsname
548     \csname BNE_op_)\expandafter\endcsname
549     \csname BNE_oparen\expandafter\endcsname
550     \csname BNE_check_)\expandafter\endcsname
551     \csname BNE_checkp_)\expandafter\endcsname
552     \csname BNE_op_-xii\endcsname
553 \let\BNE_precedence_\xint_c_i
554 \def\BNE_missing_
555     {\XINT_expandableerror{Missing }. Hit <return> to proceed.}%
556     \xint_c_ \BNE_done }%
557 \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 **\evaltohex** output routine handle comma separated values rather than braced values.

```

558 \def\BNE_tmpa #1#2#3#4#5%
559 {%
560     \def #1##1% \BNE_op_,
561     {%
562         \expanded{\unexpanded{#2{##1}}\expandafter}%
563         \romannumeral`&&\expandafter#3\romannumeral`&&\BNE_getnext
564     }%
565     \def #2##1##2##3##4{##2##3{##1##4}}% \BNE_exec_,
566     \def #3##1% \BNE_check_-,
567     {%
568         \xint_UDsignfork
569         ##1{\expandafter#4\romannumeral`&&#5}%
570         -{#4##1}%
571         \krof
572     }%
573     \def #4##1##2% \BNE_checkp_,
574     {%
575         \ifnum ##1>\xint_c_iii
576             \expandafter#4%
577             \romannumeral`&&\csname BNE_op_#2\expandafter\endcsname
578         \else
579             \expandafter##1\expandafter##2%
580         \fi
581     }%
582 }%
583 \expandafter\BNE_tmpa
584     \csname BNE_op_,\expandafter\endcsname
585     \csname BNE_exec_,\expandafter\endcsname
586     \csname BNE_check_-, \expandafter\endcsname

```

```

587 \csname BNE_checkp_,\expandafter\endcsname
588 \csname BNE_op_-xii\endcsname
589 \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_2firstofone` here. Well let's return to sanity at 1.4a and not add the `\xint_firstofone` and simply default `\BNE_Op_opp` to `\xintiio_opp`, 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 unneeded 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`.

```

590 \def\BNE_defminus_b #1#2#3#4#5%
591 {%
592   \XINT_global\def #1% \BNE_op_-<level>
593   {%
594     \expandafter #2\romannumeral`&&\expandafter#3%
595     \romannumeral`&&\BNE_getnext
596   }%
597   \XINT_global\def #2##1##2##3% \BNE_exec_-<level>
598   {%
599     \expandafter ##1\expandafter ##2\expandafter
600     {\expandafter{\romannumeral`&&\BNE_Op_opp##3}}%
601   }%
602   \XINT_global\def #3##1% \BNE_check--<level>
603   {%
604     \xint_UDsignfork
605     ##1{\expandafter #4\romannumeral`&&##1}%
606     -{##4##1}%
607   \krof
608   }%
609   \XINT_global\def #4##1##2% \BNE_checkp_-<level>
610   {%
611     \ifnum ##1>#5%
612       \expandafter #4%
613       \romannumeral`&&\csname BNE_op_-##2\expandafter\endcsname
614     \else

```

```

615     \expandafter ##1\expandafter ##2%
616     \fi
617   }%
618 }%
619 \def\BNE_defminus_a #1%
620 {%
621   \expandafter\BNE_defminus_b
622   \csname BNE_op_-#1\expandafter\endcsname
623   \csname BNE_exec_-#1\expandafter\endcsname
624   \csname BNE_check_-#1\expandafter\endcsname
625   \csname BNE_checkp_-#1\expandafter\endcsname
626   \csname xint_c_#1\endcsname
627 }%
628 \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 `\bnumdef infix` 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`).

```

629 \def\BNE_defbin_c #1#2#3#4#5#6#7%
630 {%
631   \XINT_global\def #1##1% \BNE_op_<op>
632   {%
633     \expanded{\unexpanded{#2{##1}}\expandafter}%
634     \romannumeral`&&\expandafter#3\romannumeral`&&\BNE_getnext
635   }%
636   \XINT_global\def #2##1##2##3##4% \BNE_exec_<op>
637   {%
638     \expandafter##2\expandafter##3\expandafter
639     {\expandafter{\romannumeral`&&#7##1##4}}%

```

```

640 }%
641 \XINT_global\def #3##1% \BNE_check-_{op>
642 {%
643   \xint_UDsignfork
644     ##1{\expandafter#4\romannumeral`&&@#5}%
645     -{#4##1}%
646   \krof
647 }%
648 \XINT_global\def #4##1##2% \BNE_checkp_{op>
649 {%
650   \ifnum ##1>#6%
651     \expandafter#4%
652     \romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
653   \else
654     \expandafter ##1\expandafter ##2%
655   \fi
656 }%
657 }%
658 \def\BNE_defbin_b #1#2#3#4%
659 {%
660   \expandafter\BNE_defbin_c
661   \csname BNE_op_#1\expandafter\endcsname
662   \csname BNE_exec_#1\expandafter\endcsname
663   \csname BNE_check_#1\expandafter\endcsname
664   \csname BNE_checkp_#1\expandafter\endcsname
665   \csname BNE_op_-\romannumeral\ifnum#3>12 #3\else 12\fi
666   \expandafter\endcsname
667   \csname xint_c_\romannumeral#3\endcsname #4%
668 \XINT_global
669 \expandafter
670 \let\csname BNE_precedence_#1\expandafter\endcsname
671   \csname xint_c_\romannumeral#2\endcsname
672 \unless
673 \ifcsname BNE_exec_-\romannumeral\ifnum#3>12 #3\else 12\fi\endcsname

```

This will execute only for #3>12 as \BNE_exec-*xii* exists.

```

674   \expandafter\BNE_defminus_a\expandafter{\romannumeral#3}%
675   \fi
676 }%
677 \BNE_defbin_b + {12} {12} \BNE_Op_add
678 \BNE_defbin_b - {12} {12} \BNE_Op_sub
679 \BNE_defbin_b * {14} {14} \BNE_Op_mul
680 \BNE_defbin_b / {14} {14} \BNE_Op_divround
681 \BNE_defbin_b {/} {14} {14} \BNE_Op_div
682 \BNE_defbin_b {/:} {14} {14} \BNE_Op_mod
683 \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_dothet *itselfes*, or special restrictions to add to user documentation. Better to simply handle **** as a full operator.

```

684 \BNE_defbin_b {**} {18} {17} \BNE_Op_pow

```

```

685 \expandafter\def\csname BNE_itself_**\endcsname {**}%
686 \expandafter\def\csname BNE_itself_//\endcsname {//}%
687 \expandafter\def\csname BNE_itself_/\endcsname {/}%
688 \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.

```

689 \def\bnumdefinfix #1#2#3#4%
690 {%
691   \edef\BNE_tmpa{#1}%
692   \edef\BNE_tmpa{\xint_zapspaces_o\BNE_tmpa}%
693   \edef\BNE_tmpl{\the\numexpr#3\relax}%
694   \edef\BNE_tmpl{\ifnum\BNE_tmpl<4 4\else\ifnum\BNE_tmpl<23 \BNE_tmpl\else 22\fi\fi}%
695   \edef\BNE_tmpr{\the\numexpr#4\relax}%
696   \edef\BNE_tmpr{\ifnum\BNE_tmpr<4 4\else\ifnum\BNE_tmpr<23 \BNE_tmpr\else 22\fi\fi}%
697   \BNE_defbin_b \BNE_tmpa\BNE_tmpl\BNE_tmpr #2%
698   \expandafter\BNE_dotheitselfes\BNE_tmpa\relax
699   \ifxintverbose
700     \PackageInfo{bnumexpr}{infix operator \BNE_tmpa\space
701     \ifxintglobaldefs globally \fi
702     does
703     \unexpanded{#2}\MessageBreak with precedences \BNE_tmpl, \BNE_tmpr;}%
704   \fi
705}%
706 \def\BNE_dotheitselfes#1#2%
707 {%
708   \if#2\relax\expandafter\xint_gobble_ii
709   \else
710     \XINT_global
711     \expandafter\edef\csname BNE_itself_#1#2\endcsname{#1#2}%
712     \unless\ifcsname BNE_precedence_#1\endcsname
713     \XINT_global
714     \expandafter\edef\csname BNE_precedence_#1\endcsname
715     {\csname BNE_precedence_\BNE_tmpa\endcsname}%
716     \XINT_global
717     \expandafter\odef\csname BNE_op_#1\endcsname
718     {\csname BNE_op_\BNE_tmpa\endcsname}%
719     \fi
720     \fi
721     \BNE_dotheitselfes{#1#2}%
722}%

```

9.15.2 \bnumdefpostfix

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

```

723 \def\bnumdefpostfix #1#2#3%
724 {%
725   \edef\BNE_tmpa{#1}%
726   \edef\BNE_tmpa{\xint_zapspace_o\BNE_tmpa}%
727   \edef\BNE_tmpl{\the\numexpr#3\relax}%
728   \edef\BNE_tmpl{\ifnum\BNE_tmpl<4 4\else\ifnum\BNE_tmpl<23 \BNE_tmpl\else 22\fi\fi}%
729   \XINT_global
730   \expandafter\let\csname BNE_precedence_\BNE_tmpa\expandafter\endcsname
731     \csname xint_c_\romannumeral\BNE_tmpl\endcsname
732   \XINT_global
733   \expandafter\def\csname BNE_op_\BNE_tmpa\endcsname ##1%
734   {%
735     \expandafter\BNE_put_op_first
736     \expanded{{{#2##1}}}\expandafter\romannumeral`&&\BNE_getop
737   }%
738   \expandafter\BNE_dotheitselfes\BNE_tmpa\relax
739   \ifxintverbose
740     \PackageInfo{bnumexpr}{postfix operator \BNE_tmpa\space}
741     \ifxintglobaldefs globally \fi
742     does \unexpanded{#2}\MessageBreak
743     with precedence \BNE_tmpl;}%
744   \fi
745 }%
```

9.16 ! as postfix factorial operator

```

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

9.17 Cleanup

```

747 \let\BNE_tmpa\relax \let\BNE_tmpl\relax \let\BNE_tmpr\relax
748 \let\BNE_tmpr\relax \let\BNE_tmpl\relax
749 \BNE_restorecatcodesendinginput%
```