The bnumexpr package

Jean-François Burnol
jfbu (at) free (dot) fr
Package version: 1.6 (2025/09/05)
From source file bnumexpr.dtx of 05-09-2025 at 12:10:31 CEST

1 \bnumeval 1

2 Dependencies 2

3 Examples 2

4 Customizing how output is "printed out" 4
4.1 Printing big numbers 4
4.2 \bnumprintone, \bnumprintonesep 5
4.3 \bnumprintonehex, \bnumprintoneoct, \bnumprintonebin 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 \bnumsetup command 7
6.2 Example of customization: let's handle fractions! 8
6.3 Significant differences between \bnumexpr and \numexpr 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 LaTeX package bnumexpr provides \bnumeval, which is an expandable parser of numerical expressions with big integers.

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

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

\the\numexpr⟨*expression*⟩\relax

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

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

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

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

- It allows arbitrarily big integers, whereas \inteval is limited to a maximal input equal to 2147483647 ($2^{31}-1$, or hexadecimal 7FFFFFFF).

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

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

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

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

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

[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.

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

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

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

\bnumexpr (13_8089_1090-300_1890_2902)*(1083_1908_3901-109_8290_3890)\relax
                        -2787514672889976289932

\bnethe \bnumexpr (92_874_927_979**5-31_9792_7979**6)/30!\relax
                      -4006240736596543944035189

\bnumeval{30!}
                   265252859812191058636308480000000

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

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

\bnumeval{13^50/12^50, 12^50}
       55, 91004381500021497733275852753425663249271526032565862 4

\bnumeval{(1^10+2^10+3^10+4^10+5^10+6^10+7^10+8^10+9^10)^3}
                    11868507546269898170062082 8125

\bnumeval{100!/36^100}
                                   219
```

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}
                  80000000, 40000000, AB0A74EF03A6, 4D2
```

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

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

```
\bnumeval[b]{'75316420 * 0o44445555}
```
```
100011000101101111110000101101111000111111010000
```

```
\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!} = 402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261683151027341827977704784636586817016436502415369139828126481021309276124489635992870511496497541990934222156683257208082133318611681155361583654698404670897560290095053761647584772842188967964624494516076535340819890138544248798495995331910172335555660213945039973628075013783761530712776192684903435262520001588853514733161170210396817592151090778801939317811419454525722386554146106289218796022383897147608850627686296714667469756291123408243920816015378088989396451826324367161676217916890977991190375403127462228998800519544441428201218736174599264295658174662830295557029902432415318161721046583203678690611726015878352075151628422554026517048330422614397428693306169089796848259
```

012545832716822645806652676995865268227280707578139185817888965220816434
834482599326604336766017699961283186078838615027946595513115655203609398
818061213855860030143569452722420634463179746059468257310379008402443243
846565724501440282188525247093519062092902313649327349756551395872055965
422874977401141334696271542284586237738753823048386568897646192738381490
014076731044664025989949022222176590433990188601856652648506179970235619
389701786004081188972991831102117122984590164192106888438712185564612496
079872290851929681937238864261483965738229112312502418664935314397013742
853192664987533721894069428143411852015801412334482801505139969429015348
307764456909907315243327828826986460278986432113908350621709500259738986
355427719674282224875758676575234422020757363056949882508796892816275384
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 LATEX's lingua):

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

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

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

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

1024, 59049, 9765625, 282475249

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

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

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

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

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

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

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

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

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

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

1267650600228229401496703205376, 515377520732011331036461129765621272702107522001, 788860905221011805411728565282786229673206435109023004770278930664062 5

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

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

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

5 Babel-active characters are not a problem!

Some languages use active characters with PDFLaTeX. For example the babel-french module turns the colon : and the exclamation mark ! into active characters (whose expansions would cause \bnumeval to crash). It used to be necessary to take preventive measures such as either turning the activation off altogether or use in the input /\string: and \string! as clumsy replacements of /: and !.

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

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

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

6.1 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.

   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
```

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

764966897/5000000000

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

1/241785163922925834941235 2

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

6.3 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
   \thebnumexpr 3 5+79\relax expands (in two steps) to 114
3. With \edef\myvariable{\bnumexpr 1+2\relax}, the computation is done at time of the \edef. It prepares \myvariable as a self-contained pre-computed unit which is recognized as such when inserted in a bnumexpr expressions. It triggers tacit multiplication: 7\myvariable is like 7*\myvariable. This is different from what would happen if we had used \edef\myvariable{\bnethe\bnumexpr...} which would simply have \myvariable expand to digit tokens so 7\myvariable then constructs a number with 7 as first digit.

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

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

   590490000

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

   1000059049

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

   ```
   \bnumeval{10^10\x }
   ```

   590490000000000

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

   There is no analog with \numexpr:

   a) \edef\foo{\numexpr1+2\relax} will define \foo as \numexpr1+2\relax where the calculation is not yet done.
   b) Inserting the \foo as is in the document text causes an error.
   c) Trying \the\numexpr 7\foo\relax with such a \foo causes an error. One must use the multiplication sign * explicitly.
4. Expressions may be comma separated. On input, spaces are ignored, and on output the values are comma separated with a space after each comma,
5. \thebnumexpr -(1+1)\relax is legal contrarily to \the\numexpr -(1+1)\relax which raises an error,

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

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 R_last of the last delayed operation (which already has one argument and would like to know if it can use the new found one): if L is at most equal to it, the delayed operation is now executed, else the new-found operation is kept around to be executed first, once it will have gathered its arguments, of which only one is known at this stage.

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

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

Table of precedences

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

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

```
\bnumeval{100/4(9), (100/4)9, 1000 // (100/4) 9 (1+1) * 13}
```

3, 225, 26

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

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

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*⟩}

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

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

{⟨*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

[4] 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.

[5] 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 ####...

[6] Active 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.

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

Regarding 2 + 3! = 5, trying to let this be interpreted as 2+(3!)=5 makes sense only if a =

```
\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.

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

```
\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

1.6 (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.
- Babel-active characters (such as : and ! with French) do not need any preventive measures anymore such as using \string! in place of !.

- \bnumsetup can now be used also to customize which macros implement conversion from decimal to other bases.

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

1.5 (2021/05/17)

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

1.4a (2021/05/13)

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

1.4 (2021/05/12)

- technology transfer from xintexpr 1.4 of 2020/01/31. The \expanded primitive is now required (TeXLive 2019).
- addition to the syntax of the " prefix for hexadecimal input.
- addition of \evaltohex which is like \bnumeval with an extra conversion step to hexadecimal notation.

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

1.2d (2019/01/07)
- requires xintcore 1.3d or later (if not using option custom).
- adds \bnumeval{⟨*expression*⟩} user interface.

1.2c (2017/12/05) **Breaking changes:**
- requires xintcore 1.2p or later (if not using option custom).
- divtrunc key of \bnumexprsetup is renamed to div.
- the // and /: operators are now by default associated to the *floored* division. This is to keep in sync with the change of xintcore at 1.2p.
- for backwards compatibility, one may add to existing document: \bnumexprsetup{div=\xintiiDivTrunc, mod=\xintiiModTrunc}

1.2b (2017/07/09)
- the _ may be used to separate visually blocks of digits in long numbers.

1.2a (2015/10/14)
- requires xintcore 1.2 or later (if not using option custom).
- additions to the syntax: factorial !, truncated division //, its associated modulo /: and ** as alternative to ^.
- all options removed except custom.
- new command \bnumexprsetup which replaces the commands such as \bnumexprusesbigintcalc.
- the parser is no more limited to numbers with at most 5000 digits.

1.1b (2014/10/28)
- README converted to markdown/pandoc syntax,
- the package now loads only xintcore, which belongs to xint bundle version 1.1 and extracts from the earlier xint package the core arithmetic operations as used by bnumexpr.

1.1a (2014/09/22)
- added l3bigint option to use experimental LaTeX3 package of the same name,
- added Changes and Readme sections to the documentation,
- better \BNE_protect mechanism for use of \bnumexpr...\relax inside an \edef (without \bnethe). Previous one, inherited from xintexpr.sty 1.09n, assumed that the \.=<digits> dummy control sequence encapsulating the computation result had \relax meaning. But removing this assumption was only a matter of letting \BNE_protect protect two, not one, tokens. This will be backported to next version of xintexpr, naturally (done with xintexpr.sty 1.1).

1.1 (2014/09/21) First release. This is down-scaled from the (development version of) xintexpr. Motivation came the previous day from a chat with Joseph Wright over big int status in LaTeX3. The \bnumexpr...\relax

parser can be used on top of big int macros of one's choice. Functionalities limited to the basic operations. I leave the power operator ^ as an option.

8 License

9 Commented source code

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

9.1 Package identification

```
\NeedsTeXFormat{LaTeX2e}%
\ProvidesPackage{bnumexpr}[2025/09/05 v1.6 Expressions with big integers (JFB)]%
```

9.2 Load xintkernel

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

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

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

9.3 Save catcode regime and switch to our own

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

9.4 Load optionally xintcore and xintbinhex

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

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

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

9.5 \bnumsetup

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

Note that a final comma will cause no harm.

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

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

Workaround for the space inserted by \on@line.

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

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

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

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

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

9.6 Some extra constants needed for user defined precedences

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

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

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

9.7 \bnumexpr, \bnethe, \bnumeval

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

1.6 deprecates \evaltohex 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.

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

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

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

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

A consequence is that \bnumexpr...\relax can be used as sub-unit in \xinteval and conversely \xintiiexpr...\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.

```
\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 (and \thebnumexpr).

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

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

I extended the \csname trick to \bnumexpr. Backporting then to xintexpr, it proved more economical to apply the trick at a lower level, at the level of the ``bareeval'' macros, because there are many more high level entry points overthere. And I am doing this here too, so I am leaving \bnumexpr (and \thebnumexpr) unchanged.

To compensate a bit the slight overhead I removed one expansion step, so no more a \BNE_start (which actually was there a bit for nicer tracing) and due to the history of the development of xintexpr.

For \BNE_check see the section ``Expansion spanning''.

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

```
\def\BNE_bareeval{%
    \csname BNE_check\expandafter\endcsname\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 \BNEprinthex et al. here.

```
\expandafter\def\csname BNEprint_[h]\endcsname#1{{\BNE_printhex#1.}}%
\expandafter\def\csname BNEprint_[o]\endcsname#1{{\BNE_printoct#1.}}%
\expandafter\def\csname BNEprint_[b]\endcsname#1{{\BNE_printbin#1.}}%
\expandafter\let\csname BNEprint_[]\endcsname\BNEprint_
```

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

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

This is deprecated at 1.6 and raises an expandable error.

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

This code is more compact at 1.6 than at 1.5.

```
\def\BNE_print#1{%
   \bnumprintone{#1}\expandafter\BNE_print_a\string
}%
\def\BNE_print_a#1{%
   \if#1.\BNE_print_z\fi\bnumprintonesep
   \expandafter\BNE_print\expandafter{\iffalse}\fi
}%
\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.

```
\def\BNE_printhex#1{%
    \expandafter\bnumprintonehex
    \expandafter{\romannumeral`&&@\BNE_Op_dectohex{#1}}%
    \expandafter\BNE_printhex_a\string
}%
\def\BNE_printhex_a#1{%
    \if#1.\BNE_print_z\fi\bnumprintonesep
    \expandafter\BNE_printhex\expandafter{\iffalse}\fi
}%
```

Octal and binary added at 1.6.

```
\def\BNE_printoct#1{%
    \expandafter\bnumprintoneoct
    \expandafter{\romannumeral`&&@\BNE_Op_dectooct{#1}}%
    \expandafter\BNE_printoct_a\string
}%
\def\BNE_printoct_a#1{%
    \if#1.\BNE_print_z\fi\bnumprintonesep
    \expandafter\BNE_printoct\expandafter{\iffalse}\fi
}%
\def\BNE_printbin#1{%
    \expandafter\bnumprintonebin
    \expandafter{\romannumeral`&&@\BNE_Op_dectobin{#1}}%
    \expandafter\BNE_printbin_a\string
}%
\def\BNE_printbin_a#1{%
    \if#1.\BNE_print_z\fi\bnumprintonesep
    \expandafter\BNE_printbin\expandafter{\iffalse}\fi
}%
\let\bnumprintone    \xint_firstofone
\let\bnumprintonehex\xint_firstofone
\let\bnumprintoneoct\xint_firstofone
\let\bnumprintonebin\xint_firstofone
\def\bnumprintonesep{, }%
```

9.8 \BNE_getnext

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

This \BNE_getnext token is injected via "start" macros associated to operators or like syntax elements, as will be seen later on.

```
\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
}%
```

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

```
\def\BNE_countetc\BNE_getnextfork#1%
{%
    \if0\ifx\count#11\fi
        \ifx\numexpr#11\fi
        \ifx\catcode#11\fi
        \ifx\dimen#11\fi
        \ifx\dimexpr#11\fi
        \ifx\skip#11\fi
        \ifx\glueexpr#11\fi
        \ifx\fontdimen#11\fi
        \ifx\ht#11\fi
        \ifx\dp#11\fi
        \ifx\wd#11\fi
        \ifx\fontcharht#11\fi
        \ifx\fontcharwd#11\fi
        \ifx\fontchardp#11\fi
        \ifx\fontcharic#11\fi
       0\expandafter\BNE_fetch_as_number\fi
   \expandafter\BNE_getnext_a\number #1%
}%
\def\BNE_fetch_as_number
    \expandafter\BNE_getnext_a\number #1%
{%
    \expanded{{{\number#1}}\expandafter}\romannumeral`&&@\BNE_getop
}%
```

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

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

This merges former \BNE_getnextfork and \BNE_scan_number.

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

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

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

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

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

\BNE_unexpected replaced here \BNE_notadigit at 1.6.

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

9.9 Parsing decimal, hexadecimal, octal, and binary

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

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

It is important in case of x, o, or b to jump to \BNE_starthex (et al.) and not for example to \BNE_scanhex_a because the latter expects an f-expansion to have been applied already to what comes next. Besides, we do want to trim out leading zeroes after the 0b, 0o, or 0x prefix: although the macros of xintbinhex do accept leading zeros on input, they may then produce decimal output with leading zeros, and the ``ii'' macros of xintcore consider that an input is vanishing as soon as the first digit is 0.

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

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

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

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

#1 is \fi.

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

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.

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

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

```
\def\BNE_scanint_gobz_checkagain #1\BNE_scanint_again
{%
    \if    _#1\xint_dothis\BNE_scanint_gobz_again\fi
    \if    0#1\xint_dothis\BNE_scanint_gobz_again\fi
    \xint_orthat
```

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

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.

```
\def\BNE_starthex #1%
{%
    \expandafter\BNE_starthex_i\romannumeral`&&@#1%
}%
\def\BNE_starthex_i #1%
{%
    \if #10\expandafter\BNE_scanhex_gobz_a\else\expandafter\BNE_scanhex_a\fi #1%
}%
\def\BNE_wraphex_before{\expandafter{\expandafter{%
                          \romannumeral`&&@\iffalse}}\fi\BNE_Op_hextodec}%
\def\BNE_wraphex_after{\iffalse{{{{\fi}}}}}%
\def\BNE_scanhex_a #1#2%
    {\expandafter\BNE_wraphex_before
     \expanded\bgroup{\iffalse}\fi #1%
     \expandafter\BNE_scanhex_main\romannumeral`&&@#2}%
\def\BNE_scanhex_gobz_a #1#2%
    {\expandafter\BNE_wraphex_before
     \expanded\bgroup{\iffalse}\fi
     \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.

```
\def\BNE_scanhex_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanhex_hit_cs \fi
    \if\ifnum`#1>`/
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1\else
        \expandafter\BNE_scanhex_checkagain\fi
    #1\BNE_scanhex_again
}%
\def\BNE_scanhex_again #1%
{%
    \expandafter\BNE_scanhex_main\romannumeral`&&@#1%
}%
\def\BNE_scanhex_hit_cs #1\BNE_scanhex_checkagain\fi#2\BNE_scanhex_again
{%
    \expandafter\BNE_wraphex_after\romannumeral`&&@\BNE_getop_a#2%
}%
```

```
\def\BNE_scanhex_checkagain #1\BNE_scanhex_again
{%
    \if _#1\BNE_scanhex_checkagain_skip\fi
    \expandafter\BNE_wraphex_after\romannumeral`&&@\BNE_getop_a#1%
}%
```

#1 is \fi, #3 is underscore.

```
\def\BNE_scanhex_checkagain_skip#1#2\BNE_getop_a#3{#1\BNE_scanhex_again}%
\def\BNE_scanhex_gobz_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanhex_gobz_hit_cs\fi
    \if\ifnum`#1>`0
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1\else
       \expandafter\BNE_scanhex_gobz_checkagain\fi
    #1\BNE_scanhex_again
}%
\def\BNE_scanhex_gobz_again #1%
{%
    \expandafter\BNE_scanhex_gobz_main\romannumeral`&&@#1%
}%
\def\BNE_scanhex_gobz_hit_cs#1\BNE_scanhex_gobz_checkagain\fi#2\BNE_scanhex_again
{%
    0\expandafter\BNE_wraphex_after\romannumeral`&&@\BNE_getop_a#2%
}%
\def\BNE_scanhex_gobz_checkagain #1\BNE_scanhex_again
{%
    \if    _#1\xint_dothis\BNE_scanhex_gobz_again\fi
    \if    0#1\xint_dothis\BNE_scanhex_gobz_again\fi
    \xint_orthat
    {0\expandafter\BNE_wraphex_after\romannumeral`&&@\BNE_getop_a#1}%
}%
```

Added at 1.6. Exact same code skeleton as for hexadecimal and decimal input. Leading zeros are removed.

```
\def\BNE_startoct #1%
{%
    \expandafter\BNE_startoct_i\romannumeral`&&@#1%
}%
\def\BNE_startoct_i #1%
{%
    \if #10\expandafter\BNE_scanoct_gobz_a\else\expandafter\BNE_scanoct_a\fi #1%
}%
\def\BNE_wrapoct_before{\expandafter{\expandafter{%
                          \romannumeral`&&@\iffalse}}\fi\BNE_Op_octtodec}%
\def\BNE_wrapoct_after{\iffalse{{{{\fi}}}}}%
\def\BNE_scanoct_a #1#2%
    {\expandafter\BNE_wrapoct_before
     \expanded\bgroup{\iffalse}\fi #1%
     \expandafter\BNE_scanoct_main\romannumeral`&&@#2}%
\def\BNE_scanoct_gobz_a #1#2%
```

```
    {\expandafter\BNE_wrapoct_before
     \expanded\bgroup{\iffalse}\fi
     \expandafter\BNE_scanoct_gobz_main\romannumeral`&&@#2}%
\def\BNE_scanoct_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanoct_hit_cs \fi
    \if\ifnum`#1>`/ \ifnum`#1>`7 0\else1\fi\else0\fi 1\else
        \expandafter\BNE_scanoct_checkagain\fi
    #1\BNE_scanoct_again
}%
\def\BNE_scanoct_again #1%
{%
    \expandafter\BNE_scanoct_main\romannumeral`&&@#1%
}%
\def\BNE_scanoct_hit_cs #1\BNE_scanoct_checkagain\fi#2\BNE_scanoct_again
{%
    \expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#2%
}%
\def\BNE_scanoct_checkagain #1\BNE_scanoct_again
{%
    \if _#1\BNE_scanoct_checkagain_skip\fi
    \expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#1%
}%
```

#1 is \fi, #3 is underscore.

```
\def\BNE_scanoct_checkagain_skip#1#2\BNE_getop_a#3{#1\BNE_scanoct_again}%
\def\BNE_scanoct_gobz_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanoct_gobz_hit_cs\fi
    \if\ifnum`#1>`0 \ifnum`#1>`7 0\else1\fi\else0\fi 1\else
       \expandafter\BNE_scanoct_gobz_checkagain\fi
    #1\BNE_scanoct_again
}%
\def\BNE_scanoct_gobz_again #1%
{%
    \expandafter\BNE_scanoct_gobz_main\romannumeral`&&@#1%
}%
\def\BNE_scanoct_gobz_hit_cs#1\BNE_scanoct_gobz_checkagain\fi#2\BNE_scanoct_again
{%
    0\expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#2%
}%
\def\BNE_scanoct_gobz_checkagain #1\BNE_scanoct_again
{%
    \if    _#1\xint_dothis\BNE_scanoct_gobz_again\fi
    \if    0#1\xint_dothis\BNE_scanoct_gobz_again\fi
    \xint_orthat
    {0\expandafter\BNE_wrapoct_after\romannumeral`&&@\BNE_getop_a#1}%
}%
```

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

```
\def\BNE_startbin #1%
{%
```

```
    \expandafter\BNE_startbin_i\romannumeral`&&@#1%
}%
\def\BNE_startbin_i #1%
{%
    \if #10\expandafter\BNE_scanbin_gobz_a\else\expandafter\BNE_scanbin_a\fi #1%
}%
\def\BNE_wrapbin_before{\expandafter{\expandafter{%
                          \romannumeral`&&@\iffalse}}\fi\BNE_Op_bintodec}%
\def\BNE_wrapbin_after{\iffalse{{{{\fi}}}}}%
\def\BNE_scanbin_a #1#2%
    {\expandafter\BNE_wrapbin_before
     \expanded\bgroup{\iffalse}\fi #1%
     \expandafter\BNE_scanbin_main\romannumeral`&&@#2}%
\def\BNE_scanbin_gobz_a #1#2%
    {\expandafter\BNE_wrapbin_before
     \expanded\bgroup{\iffalse}\fi
     \expandafter\BNE_scanbin_gobz_main\romannumeral`&&@#2}%
\def\BNE_scanbin_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanbin_hit_cs \fi
    \if1\if0#11\else\if1#11\else0\fi\fi\else
        \expandafter\BNE_scanbin_checkagain\fi
    #1\BNE_scanbin_again
}%
\def\BNE_scanbin_again #1%
{%
    \expandafter\BNE_scanbin_main\romannumeral`&&@#1%
}%
\def\BNE_scanbin_hit_cs #1\BNE_scanbin_checkagain\fi#2\BNE_scanbin_again
{%
    \expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#2%
}%
\def\BNE_scanbin_checkagain #1\BNE_scanbin_again
{%
    \if _#1\BNE_scanbin_checkagain_skip\fi
    \expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#1%
}%
```

#1 is \fi, #3 is underscore.

```
\def\BNE_scanbin_checkagain_skip#1#2\BNE_getop_a#3{#1\BNE_scanbin_again}%
\def\BNE_scanbin_gobz_main #1%
{%
    \ifcat \relax #1\expandafter\BNE_scanbin_gobz_hit_cs\fi
    \if1#1\else\expandafter\BNE_scanbin_gobz_checkagain\fi
    #1\BNE_scanbin_again
}%
\def\BNE_scanbin_gobz_again #1%
{%
    \expandafter\BNE_scanbin_gobz_main\romannumeral`&&@#1%
}%
\def\BNE_scanbin_gobz_hit_cs#1\BNE_scanbin_gobz_checkagain\fi#2\BNE_scanbin_again
{%
    0\expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#2%
```

```
}%
\def\BNE_scanbin_gobz_checkagain #1\BNE_scanbin_again
{%
    \if    _#1\xint_dothis\BNE_scanbin_gobz_again\fi
    \if    0#1\xint_dothis\BNE_scanbin_gobz_again\fi
    \xint_orthat
    {0\expandafter\BNE_wrapbin_after\romannumeral`&&@\BNE_getop_a#1}%
}%
```

9.10 \BNE_getop

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

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

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

```
    \romannumeral`&&@%
}%
\def\BNE_scanop_d #1#2%
{%
    \unless\ifcat#2\relax
           \ifcsname BNE_itself_#1#2\endcsname
           \BNE_scanop_c
    \fi\fi
    \BNE_foundop #1#2%
}%
```

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

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

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

9.11 Expansion spanning; opening and closing parentheses

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

At 1.6, we define one less macro, see comment at location of definition of \BNE_bareeval. Upstream code has \BNE_tmpa do all three definitions, (and for the three parsers via an \xintFor loop) here we do things one by one.

```
\def\BNE_tmpa#1{%
    \def\BNE_check##1%
    {%
        \xint_UDsignfork
          ##1{\expandafter\BNE_checkp\romannumeral`&&@#1}%
            -{\BNE_checkp##1}%
        \krof
    }%
}\expandafter\BNE_tmpa\csname BNE_op_-xii\endcsname
\def\BNE_tmpa#1{%
    \def\BNE_checkp##1##2%
```

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

Code style left untouched at 1.6.

```
\catcode`) 11
\def\BNE_tmpa #1#2#3#4#5#6%
{%
    \def #1##1% op_(
    {%
        \expandafter #4\romannumeral`&&@\BNE_getnext
    }%
    \def #2##1% op_)
    {%
        \expanded{\unexpanded{\BNE_put_op_first{##1}}\expandafter}%
        \romannumeral`&&@\BNE_getop
    }%
    \def #3% oparen
    {%
        \expandafter #4\romannumeral`&&@\BNE_getnext
    }%
    \def #4##1% check-
    {%
        \xint_UDsignfork
            ##1{\expandafter#5\romannumeral`&&@#6}%
              -{#5##1}%
        \krof
    }%
    \def #5##1##2% checkp
    {%
        \ifcase ##1\expandafter\BNE_missing_)
        \or \csname BNE_op_##2\expandafter\endcsname
        \else
          \expandafter #5\romannumeral`&&@\csname BNE_op_##2\expandafter\endcsname
        \fi
```

```
    }%
}%
\expandafter\BNE_tmpa
    \csname BNE_op_(\expandafter\endcsname
    \csname BNE_op_)\expandafter\endcsname
    \csname BNE_oparen\expandafter\endcsname
    \csname BNE_check-_)\expandafter\endcsname
    \csname BNE_checkp_)\expandafter\endcsname
    \csname BNE_op_-xii\endcsname
\let\BNE_precedence_)\xint_c_i
\def\BNE_missing_)
   {\XINT_expandableerror{Missing ). Hit <return> to proceed.}%
    \xint_c_ \BNE_done }%
\catcode`) 12
```

9.12 The comma as binary operator

At 1.4, it is simply a union operator for 1D oples. Inserting directly here a <comma><space> separator (as in earlier releases) in accumulated result would avoid having to do it on output but to the cost of diverging from xintexpr upstream code, and to have to let the \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
```

9.13 The minus as prefix operator of variable precedence level

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

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

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

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

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

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

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

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

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

9.14 The infix operators.

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

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

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

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

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

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

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

```
  {%
    \xint_UDsignfork
      ##1{\expandafter#4\romannumeral`&&@#5}%
        -{#4##1}%
    \krof
  }%
  \XINT_global\def #4##1##2% \BNE_checkp_<op>
  {%
    \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
```

xintexpr uses shortcut

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

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

```
\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
```

9.15 Extending the syntax: \bnumdefinfix, \bnumdefpostfix

9.15.1 \bnumdefinfix

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

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

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

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

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

9.15.2 \bnumdefpostfix

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

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

9.16 ! as postfix factorial operator

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

9.17 Cleanup

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