The xint packages source code

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

Package version: 1.4d (2021/03/29); documentation date: 2021/03/29.
From source file xint.dtx. Time-stamp: <29-03-2021 at 11:06:25 CEST>.

Contents

1 Timeline (in brief)

This is 1.4d of 2021/03/29.
Please refer to CHANGES.html for a (very) detailed history.
Internet: http://mirrors.ctan.org/macros/generic/xint/CHANGES.html

- Release 1.4 of 2020/01/31: xintexpr overhaul to use \expanded based expansion control. Many new features, in particular support for input and output of nested structures. Breaking changes, main ones being the (provisory) drop of x*[a, b,...], x+[a, b,...] et al. syntax and the requirement of \expanded primitive (currently required only by xintexpr).
- Release 1.3f of 2019/09/10: starred variant \xintDigits*.
- Release 1.3e of 2019/04/05: packages xinttrig, xintlog; \xintdefefunc ``non-protected'' variant of \xintdeffunc (at 1.4 the two got merged and \xintdefefunc became a deprecated alias for \xintdeffunc). Indices removed from sourcexint.pdf.
- Release 1.3d of 2019/01/06: fix of 1.2p bug for division with a zero dividend and a one-digit divisor, \xinteval et al. wrappers, gcd() and lcm() work with fractions.
- Release 1.3c of 2018/06/17: documentation better hyperlinked, indices added to sourcexint.pdf. Colon in := now optional for \xintdefvar and \xintdeffunc.
- Release 1.3b of 2018/05/18: randomness related additions (still WIP).
- Release 1.3a of 2018/03/07: efficiency fix of the mechanism for recursive functions.
- Release 1.3 of 2018/03/01: addition and subtraction use systematically least common multiple of denominators. Extensive under-the-hood refactoring of \xintNewExpr and \xintdeffunc which now allow recursive definitions. Removal of 1.2o deprecated macros.
- Release 1.2q of 2018/02/06: fix of 1.2l subtraction bug in special situation; tacit multiplication extended to cases such as 10!20!30!.
- Release 1.2p of 2017/12/05: maps // and /: to the floored, not truncated, division. Simultaneous assignments possible with \xintdefvar. Efficiency improvements in xinttools.
- Release 1.2o of 2017/08/29: massive deprecations of those macros from xintcore and xint which filtered their arguments via \xintNum.
- Release 1.2n of 2017/08/06: improvements of xintbinhex.
- Release 1.2m of 2017/07/31: rewrite of xintbinhex in the style of the 1.2 techniques.
- Release 1.2l of 2017/07/26: under the hood efficiency improvements in the style of the 1.2 techniques; subtraction refactored. Compatibility of most xintfrac macros with arguments using non-delimited \the\numexpr or \the\mathcode etc...
- Release 1.2i of 2016/12/13: under the hood efficiency improvements in the style of the 1.2 techniques.
- Release 1.2 of 2015/10/10: complete refactoring of the core arithmetic macros and faster \xintexpr parser.
- Release 1.1 of 2014/10/28: extensive changes in xintexpr. Addition and subtraction do not multiply denominators blindly but sometimes produce smaller ones. Also with that release, packages xintkernel and xintcore got extracted from xinttools and xint.

- Release 1.09g of 2013/11/22: the xinttools package is extracted from xint; addition of \xintloop and \xintiloop.
- Release 1.09c of 2013/10/09: \xintFor, \xintNewNumExpr (ancestor of \xintNewExpr/\xintdeffunc mechanism).
- Release 1.09a of 2013/09/24: support for functions by xintexpr.
- Release 1.08 of 2013/06/07: the xintbinhex package.
- Release 1.07 of 2013/05/25: support for floating point numbers added to xintfrac and first release of the xintexpr package (provided \xintexpr and \xintfloatexpr).
- Release 1.04 of 2013/04/25: the xintcfrac package.
- Release 1.03 of 2013/04/14: the xintfrac and xintseries packages.
- Release 1.0 of 2013/03/28: initial release of the xint and xintgcd packages.

Some parts of the code still date back to the initial release, and at that time I was learning my trade in expandable TeX macro programming. At some point in the future, I will have to re-examine the older parts of the code.

Warning: pay attention when looking at the code to the catcode configuration as found in \XINT_setcatcodes. Additional temporary configuration is used at some locations. For example ! is of catcode letter in xintexpr and there are locations with funny catcodes e.g. using some letters with the math shift catcode.

2 Package xintkernel implementation

This package provides the common minimal code base for loading management and catcode control and also a few programming utilities. With 1.2 a few more helper macros and all \chardef's have been moved here. The package is loaded by both xintcore.sty and xinttools.sty hence by all other packages.

1.1. separated package.

1.2i. \xintreplicate, \xintgobble, \xintLengthUpTo and \xintLastItem, and faster \xintLength.

1.3b. \xintUniformDeviate.

1.4 (2020/01/11). \xintReplicate, \xintGobble, \xintLastOne, \xintFirstOne.

2.1 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from Heiko Oberdiek's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \catcode95=11   % _
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \let\z\relax
  \expandafter
    \ifx\csname numexpr\endcsname\relax
     \y{xintkernel}{\numexpr not available, aborting input}%
```

```
    \def\z{\endgroup\endinput}%
  \else
  \expandafter
   \ifx\csname XINTsetupcatcodes\endcsname\relax
    \else
     \y{xintkernel}{I was already loaded, aborting input}%
     \def\z{\endgroup\endinput}%
    \fi
  \fi
\ifx\z\relax\else\expandafter\z\fi%
```

2.1.1 \XINT_setcatcodes, \XINT_storecatcodes, \XINT_restorecatcodes_endinput

```
\def\PrepareCatcodes
{%
    \endgroup
    \def\XINT_restorecatcodes
    {% takes care of all, to allow more economical code in modules
         \catcode0=\the\catcode0 %
         \catcode59=\the\catcode59   % ; xintexpr
         \catcode126=\the\catcode126 % ~ xintexpr
         \catcode39=\the\catcode39   % ' xintexpr
         \catcode34=\the\catcode34   % " xintbinhex, and xintexpr
         \catcode63=\the\catcode63   % ? xintexpr
         \catcode124=\the\catcode124 % | xintexpr
         \catcode38=\the\catcode38   % & xintexpr
         \catcode64=\the\catcode64   % @ xintexpr
         \catcode33=\the\catcode33   % ! xintexpr
         \catcode93=\the\catcode93   % ] -, xintfrac, xintseries, xintcfrac
         \catcode91=\the\catcode91   % [ -, xintfrac, xintseries, xintcfrac
         \catcode36=\the\catcode36   % $ xintgcd only
         \catcode94=\the\catcode94   % ^
         \catcode96=\the\catcode96   % `
         \catcode47=\the\catcode47   % /
         \catcode41=\the\catcode41   % )
         \catcode40=\the\catcode40   % (
         \catcode42=\the\catcode42   % *
         \catcode43=\the\catcode43   % +
         \catcode62=\the\catcode62   % >
         \catcode60=\the\catcode60   % <
         \catcode58=\the\catcode58   % :
         \catcode46=\the\catcode46   % .
         \catcode45=\the\catcode45   % -
         \catcode44=\the\catcode44   % ,
         \catcode35=\the\catcode35   % #
         \catcode95=\the\catcode95   % _
         \catcode125=\the\catcode125 % }
         \catcode123=\the\catcode123 % {
         \endlinechar=\the\endlinechar
         \catcode13=\the\catcode13   % ^^M
         \catcode32=\the\catcode32   %
         \catcode61=\the\catcode61\relax   % =
    }%
```

```
      \edef\XINT_restorecatcodes_endinput
      {%
           \XINT_restorecatcodes\noexpand\endinput %
      }%
      \def\XINT_setcatcodes
      {%
        \catcode61=12   % =
        \catcode32=10   % space
        \catcode13=5    % ^^M
        \endlinechar=13 %
        \catcode123=1   % {
        \catcode125=2   % }
        \catcode95=11   % _ LETTER
        \catcode35=6    % #
        \catcode44=12   % ,
        \catcode45=12   % -
        \catcode46=12   % .
        \catcode58=11   % : LETTER
        \catcode60=12   % <
        \catcode62=12   % >
        \catcode43=12   % +
        \catcode42=12   % *
        \catcode40=12   % (
        \catcode41=12   % )
        \catcode47=12   % /
        \catcode96=12   % `
        \catcode94=11   % ^ LETTER
        \catcode36=3    % $
        \catcode91=12   % [
        \catcode93=12   % ]
        \catcode33=12   % ! (xintexpr.sty will use catcode 11)
        \catcode64=11   % @ LETTER
        \catcode38=7    % & for \romannumeral`&&@ trick.
        \catcode124=12  % |
        \catcode63=11   % ? LETTER
        \catcode34=12   % "
        \catcode39=12   % '
        \catcode126=3   % ~ MATH
        \catcode59=12   % ;
        \catcode0=12    % for \romannumeral`&&@ trick
        \catcode1=3     % for ultra-safe séparateur &&A
      }%
      \XINT_setcatcodes
  }%
\PrepareCatcodes
```

Other modules could possibly be loaded under a different catcode regime.

```
\def\XINTsetupcatcodes {% for use by other modules
      \edef\XINT_restorecatcodes_endinput
      {%
           \XINT_restorecatcodes\noexpand\endinput %
      }%
      \XINT_setcatcodes
```

```
}%
```

2.2 Package identification

Inspired from Heiko Oberdiek's packages. Modified in 1.09b to allow re-use in the other modules. Also I assume now that if \ProvidesPackage exists it then does define \ver@<pkgname>.sty, code of HO for some reason escaping me (compatibility with LaTeX 2.09 or other things ??) seems to set extra precautions.

1.09c uses e-TEX \ifdefined.

```
\ifdefined\ProvidesPackage
  \let\XINT_providespackage\relax
\else
  \def\XINT_providespackage #1#2[#3]%
              {\immediate\write-1{Package: #2 #3}%
               \expandafter\xdef\csname ver@#2.sty\endcsname{#3}}%
\fi
\XINT_providespackage
\ProvidesPackage {xintkernel}%
  [2021/03/29 v1.4d Paraphernalia for the xint packages (JFB)]%
```

2.3 Constants

```
\chardef\xint_c_     0
\chardef\xint_c_i    1
\chardef\xint_c_ii   2
\chardef\xint_c_iii  3
\chardef\xint_c_iv   4
\chardef\xint_c_v    5
\chardef\xint_c_vi   6
\chardef\xint_c_vii  7
\chardef\xint_c_viii 8
\chardef\xint_c_ix     9
\chardef\xint_c_x     10
\chardef\xint_c_xii   12
\chardef\xint_c_xiv   14
\chardef\xint_c_xvi   16
\chardef\xint_c_xviii 18
\chardef\xint_c_xx    20
\chardef\xint_c_xxii  22
\chardef\xint_c_ii^v  32
\chardef\xint_c_ii^vi 64
\chardef\xint_c_ii^vii       128
\mathchardef\xint_c_ii^viii  256
\mathchardef\xint_c_ii^xii  4096
\mathchardef\xint_c_x^iv   10000
```

2.4 (WIP) \xint_texuniformdeviate and needed counts

```
\ifdefined\pdfuniformdeviate  \let\xint_texuniformdeviate\pdfuniformdeviate\fi
\ifdefined\uniformdeviate     \let\xint_texuniformdeviate\uniformdeviate   \fi
\ifx\xint_texuniformdeviate\relax\let\xint_texuniformdeviate\xint_undefined\fi
\ifdefined\xint_texuniformdeviate
  \csname newcount\endcsname\xint_c_ii^xiv
  \xint_c_ii^xiv   16384 % "4000, 2**14
```

```
  \csname newcount\endcsname\xint_c_ii^xxi
  \xint_c_ii^xxi 2097152 % "200000, 2**21
\fi
```

2.5 Token management utilities

1.3b. `\xint_gobandstop_...` macros because this is handy for `\xintRandomDigits`. `1.3g` forces `\empty` and `\space` to have their standard meanings, rather than simply alerting user in the (theoretical) case they don't that nothing will work. If some LaTeX user has `\renewcommanded` them they will be long and this will trigger xint redefinitions and warnings.

```
\def\XINT_tmpa { }%
\ifx\XINT_tmpa\space\else
   \immediate\write-1{Package xintkernel Warning:}%
   \immediate\write-1{\string\space\XINT_tmpa macro does not have its normal
     meaning from Plain or LaTeX, but:}%
   \immediate\write-1{\meaning\space}%
   \let\space\XINT_tmpa
   \immediate\write-1{\space\space\space\space
   % an exclam might let Emacs/AUCTeX think it is an error message, afair
                       Forcing \string\space\space to be the usual one.}%
\fi
\def\XINT_tmpa {}%
\ifx\XINT_tmpa\empty\else
   \immediate\write-1{Package xintkernel Warning:}%
   \immediate\write-1{\string\empty\space macro does not have its normal
     meaning from Plain or LaTeX, but:}%
   \immediate\write-1{\meaning\empty}%
   \let\empty\XINT_tmpa
   \immediate\write-1{\space\space\space\space
                       Forcing \string\empty\space to be the usual one.}%
\fi
\let\XINT_tmpa\relax
\let\xint_gobble_\empty
\long\def\xint_gobble_i    #1{}%
\long\def\xint_gobble_ii   #1#2{}%
\long\def\xint_gobble_iii  #1#2#3{}%
\long\def\xint_gobble_iv   #1#2#3#4{}%
\long\def\xint_gobble_v    #1#2#3#4#5{}%
\long\def\xint_gobble_vi   #1#2#3#4#5#6{}%
\long\def\xint_gobble_vii  #1#2#3#4#5#6#7{}%
\long\def\xint_gobble_viii #1#2#3#4#5#6#7#8{}%
\let\xint_gob_andstop_\space
\long\def\xint_gob_andstop_i    #1{ }%
\long\def\xint_gob_andstop_ii   #1#2{ }%
\long\def\xint_gob_andstop_iii  #1#2#3{ }%
\long\def\xint_gob_andstop_iv   #1#2#3#4{ }%
\long\def\xint_gob_andstop_v    #1#2#3#4#5{ }%
\long\def\xint_gob_andstop_vi   #1#2#3#4#5#6{ }%
\long\def\xint_gob_andstop_vii  #1#2#3#4#5#6#7{ }%
\long\def\xint_gob_andstop_viii #1#2#3#4#5#6#7#8{ }%
\long\def\xint_firstofone  #1{#1}%
\long\def\xint_firstoftwo  #1#2{#1}%
\long\def\xint_secondoftwo #1#2{#2}%
```

```
\long\def\xint_thirdofthree#1#2#3{#3}% 1.4d
\let\xint_stop_aftergobble\xint_gob_andstop_i
\long\def\xint_stop_atfirstofone  #1{ #1}%
\long\def\xint_stop_atfirstoftwo  #1#2{ #1}%
\long\def\xint_stop_atsecondoftwo #1#2{ #2}%
\long\def\xint_exchangetwo_keepbraces    #1#2{{#2}{#1}}%
```

2.6 "gob til" macros and UD style fork

```
\long\def\xint_gob_til_R #1\R {}%
\long\def\xint_gob_til_W #1\W {}%
\long\def\xint_gob_til_Z #1\Z {}%
\long\def\xint_gob_til_zero #10{}%
\long\def\xint_gob_til_one  #11{}%
\long\def\xint_gob_til_zeros_iii   #1000{}%
\long\def\xint_gob_til_zeros_iv    #10000{}%
\long\def\xint_gob_til_eightzeroes #100000000{}%
\long\def\xint_gob_til_dot    #1.{}%
\long\def\xint_gob_til_G     #1G{}%
\long\def\xint_gob_til_minus #1-{}%
\long\def\xint_UDzerominusfork #10-#2#3\krof {#2}%
\long\def\xint_UDzerofork       #10#2#3\krof {#2}%
\long\def\xint_UDsignfork       #1-#2#3\krof {#2}%
\long\def\xint_UDwfork         #1\W#2#3\krof {#2}%
\long\def\xint_UDXINTWfork     #1\XINT_W#2#3\krof {#2}%
\long\def\xint_UDzerosfork     #100#2#3\krof {#2}%
\long\def\xint_UDonezerofork   #110#2#3\krof {#2}%
\long\def\xint_UDsignsfork     #1--#2#3\krof {#2}%
\let\xint:\char
\long\def\xint_gob_til_xint:#1\xint:{}%
\long\def\xint_gob_til_^#1^{}%
\def\xint_bracedstopper{\xint:}%
\long\def\xint_gob_til_exclam #1!{}% documenter le catcode de ! ici
\long\def\xint_gob_til_sc #1;{}%
```

2.7 \xint_afterfi

```
\long\def\xint_afterfi #1#2\fi {\fi #1}%
```

2.8 \xint_bye, \xint_Bye

1.09. \xint_bye

1.2i. \xint_Bye for \xintDSRr and \xintRound. Also \xint_stop_afterbye.

```
\long\def\xint_bye #1\xint_bye {}%
\long\def\xint_Bye #1\xint_bye {}%
\long\def\xint_stop_afterbye #1\xint_bye { }%
```

2.9 \xintdothis, \xintorthat

1.1.

1.2. names without underscores.
To be used this way:

```
\if..\xint_dothis{..}\fi
```

```
\if..\xint_dothis{..}\fi
\if..\xint_dothis{..}\fi
...more such...
\xint_orthat{...}
```

Ancient testing indicated it is more efficient to list first the more improbable clauses.

```
\long\def\xint_dothis #1#2\xint_orthat #3{\fi #1}% 1.1
\let\xint_orthat \xint_firstofone
\long\def\xintdothis #1#2\xintorthat #3{\fi #1}%
\let\xintorthat \xint_firstofone
```

2.10 \xint_zapspaces

1.1.

This little (quite fragile in the normal sense i.e. non robust in the normal sense of programming lingua) utility zaps leading, intermediate, trailing, spaces in completely expanding context (\edef, \csname...\endcsname).

Usage: \xint_zapspaces foo<space>\xint_gobble_i

Explanation: if there are leading spaces, then the first #1 will be empty, and the first #2 being undelimited will be stripped from all the remaining leading spaces, if there was more than one to start with. Of course brace-stripping may occur. And this iterates: each time a #2 is removed, either we then have spaces and next #1 will be empty, or we have no spaces and #1 will end at the first space. Ultimately #2 will be \xint_gobble_i.

The \zap@spaces of LaTeX2e handles unexpectedly things such as

\zap@spaces 1 {22} 3 4 \@empty

(spaces are not all removed). This does not happen with \xint_zapspaces.

But for example \foo{aa} {bb} {cc} where \foo is a macro with three non-delimited arguments breaks expansion, as expansion of \foo will happen with \xint_zapspaces still around, and even if it wasn't it would have stripped the braces around {bb}, certainly breaking other things.

Despite such obvious shortcomings it is enough for our purposes. It is currently used by xintexpr at various locations e.g. cleaning up optional argument of \xintiexpr and \xintfloatexpr; maybe in future internal usage will drop this in favour of a more robust utility.

1.2e. \xint_zapspaces_o.

1.2i. made \long.

ATTENTION THAT xinttools HAS AN WHICH SHOULD NOT GET CONFUSED WITH THIS ONE

```
\long\def\xint_zapspaces #1 #2{#1#2\xint_zapspaces }% 1.1
\long\def\xint_zapspaces_o #1{\expandafter\xint_zapspaces#1 \xint_gobble_i}%
```

2.11 \odef, \oodef, \fdef

May be prefixed with \global. No parameter text.

```
\def\xintodef #1{\expandafter\def\expandafter#1\expandafter }%
\def\xintoodef #1{\expandafter\expandafter\expandafter\def
                  \expandafter\expandafter\expandafter#1%
                  \expandafter\expandafter\expandafter }%
\def\xintfdef #1#2%
    {\expandafter\def\expandafter#1\expandafter{\romannumeral`&&@#2}}%
\ifdefined\odef\else\let\odef\xintodef\fi
\ifdefined\oodef\else\let\oodef\xintoodef\fi
\ifdefined\fdef\else\let\fdef\xintfdef\fi
```

2.12 \xintReverseOrder

1.0. does not expand its argument. The whole of xint codebase now contains only two calls to \XINT_rord_main (in xintgcd).
Attention: removes brace pairs (and swallows spaces).
For digit tokens a faster reverse macro is provided by (1.2) \xintReverseDigits in xint.
For comma separated items, 1.2g has \xintCSVReverse in xinttools.

```
\def\xintReverseOrder {\romannumeral0\xintreverseorder }%
\long\def\xintreverseorder #1%
{%
    \XINT_rord_main {}#1%
      \xint:
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint:
}%
\long\def\XINT_rord_main #1#2#3#4#5#6#7#8#9%
{%
    \xint_bye #9\XINT_rord_cleanup\xint_bye
    \XINT_rord_main {#9#8#7#6#5#4#3#2#1}%
}%
\def\XINT_rord_cleanup #1{%
\long\def\XINT_rord_cleanup\xint_bye\XINT_rord_main ##1##2\xint:
{%
    \expandafter#1\xint_gob_til_xint: ##1%
}}\XINT_rord_cleanup { }%
```

2.13 \xintLength

1.0. does not expand its argument. See \xintNthElt{0} from xinttools which f-expands its argument.

1.2g. added \xintCSVLength to xinttools.

1.2i. rewrote this venerable macro. New code about 40% faster across all lengths. Syntax with \romannumeral0 adds some slight (negligible) overhead; it is done to fit some general principles of structure of the xint package macros but maybe at some point I should drop it. And in fact it is often called directly via the \numexpr access point. (bad coding...)

```
\def\xintLength {\romannumeral0\xintlength }%
\def\xintlength #1{%
\long\def\xintlength ##1%
{%
    \expandafter#1\the\numexpr\XINT_length_loop
    ##1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
       \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
       \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    \relax
}}\xintlength{ }%
\long\def\XINT_length_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint: #9\XINT_length_finish_a\xint:
    \xint_c_ix+\XINT_length_loop
}%
\def\XINT_length_finish_a\xint:\xint_c_ix+\XINT_length_loop
    #1#2#3#4#5#6#7#8#9%
```

```
{%
    #9\xint_bye
}%
```

2.14 \xintLastItem

1.2i (2016/12/10). One level of braces removed in output. Output empty if input empty. Attention! This means that an empty input or an input ending with a empty brace pair both give same output. The \xint: token must not be among items. \xintFirstItem added at 1.4 for usage in xintexpr. It must contain neither \xint: nor \xint_bye in its first item.

```
\def\xintLastItem {\romannumeral0\xintlastitem }%
\long\def\xintlastitem #1%
{%
    \XINT_last_loop {}.#1%
    {\xint:\XINT_last_loop_enda}{\xint:\XINT_last_loop_endb}%
    {\xint:\XINT_last_loop_endc}{\xint:\XINT_last_loop_endd}%
    {\xint:\XINT_last_loop_ende}{\xint:\XINT_last_loop_endf}%
    {\xint:\XINT_last_loop_endg}{\xint:\XINT_last_loop_endh}\xint_bye
}%
\long\def\XINT_last_loop #1.#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint: #9%
        {#8}{#7}{#6}{#5}{#4}{#3}{#2}{#1}\xint:
    \XINT_last_loop {#9}.%
}%
\long\def\XINT_last_loop_enda #1#2\xint_bye{ #1}%
\long\def\XINT_last_loop_endb #1#2#3\xint_bye{ #2}%
\long\def\XINT_last_loop_endc #1#2#3#4\xint_bye{ #3}%
\long\def\XINT_last_loop_endd #1#2#3#4#5\xint_bye{ #4}%
\long\def\XINT_last_loop_ende #1#2#3#4#5#6\xint_bye{ #5}%
\long\def\XINT_last_loop_endf #1#2#3#4#5#6#7\xint_bye{ #6}%
\long\def\XINT_last_loop_endg #1#2#3#4#5#6#7#8\xint_bye{ #7}%
\long\def\XINT_last_loop_endh #1#2#3#4#5#6#7#8#9\xint_bye{ #8}%
```

2.15 \xintFirstItem

1.4. There must be neither \xint: nor \xint_bye in its first item.

```
\def\xintFirstItem         {\romannumeral0\xintfirstitem }%
\long\def\xintfirstitem  #1{\XINT_firstitem #1{\xint:\XINT_firstitem_end}\xint_bye}%
\long\def\XINT_firstitem #1#2\xint_bye{\xint_gob_til_xint: #1\xint:\space #1}%
\def\XINT_firstitem_end\xint:{ }%
```

2.16 \xintLastOne

As xintexpr 1.4 uses {c1}{c2}....{cN} storage when gathering comma separated values we need to not handle identically an empty list and a list with an empty item (as the above allows hierarchical structures). But \xintLastItem removed one level of brace pair so it is anadequate for the last() function.

By the way it is logical to interpret «item» as meaning {cj} inclusive of the braces; but xint user manual was not written in this spirit. And thus \xintLastItem did brace stripping, thus we

need another name for maintaining backwards compatibility (although the cardinality of users is small).

The \xint: token must not be found (visible) among the item contents.

```
\def\xintLastOne {\romannumeral0\xintlastone }%
\long\def\xintlastone #1%
{%
    \XINT_lastone_loop {}.#1%
    {\xint:\XINT_lastone_loop_enda}{\xint:\XINT_lastone_loop_endb}%
    {\xint:\XINT_lastone_loop_endc}{\xint:\XINT_lastone_loop_endd}%
    {\xint:\XINT_lastone_loop_ende}{\xint:\XINT_lastone_loop_endf}%
    {\xint:\XINT_lastone_loop_endg}{\xint:\XINT_lastone_loop_endh}\xint_bye
}%
\long\def\XINT_lastone_loop #1.#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint: #9%
        {#8}{#7}{#6}{#5}{#4}{#3}{#2}{#1}\xint:
    \XINT_lastone_loop {{#9}}.%
}%
\long\def\XINT_lastone_loop_enda #1#2\xint_bye{{#1}}%
\long\def\XINT_lastone_loop_endb #1#2#3\xint_bye{{#2}}%
\long\def\XINT_lastone_loop_endc #1#2#3#4\xint_bye{{#3}}%
\long\def\XINT_lastone_loop_endd #1#2#3#4#5\xint_bye{{#4}}%
\long\def\XINT_lastone_loop_ende #1#2#3#4#5#6\xint_bye{{#5}}%
\long\def\XINT_lastone_loop_endf #1#2#3#4#5#6#7\xint_bye{{#6}}%
\long\def\XINT_lastone_loop_endg #1#2#3#4#5#6#7#8\xint_bye{{#7}}%
\long\def\XINT_lastone_loop_endh #1#2#3#4#5#6#7#8#9\xint_bye{ #8}%
```

2.17 \xintFirstOne

For xintexpr 1.4 too. Jan 3, 2020.

This is an experimental macro, don't use it. If input is nil (empty set) it expands to nil, if not it fetches first item and brace it. Fetching will have stripped one brace pair if item was braced to start with, which is the case in non-symbolic xintexpr data objects.

I have not given much thought to this (make it shorter, allow all tokens, (we could first test if empty via combination with \detokenize), etc...) as I need to get xint 1.4 out soon. So in particular attention that the macro assumes the \xint: token is absent from first item of input.

```
\def\xintFirstOne  {\romannumeral0\xintfirstone }%
\long\def\xintfirstone  #1{\XINT_firstone #1{\xint:\XINT_firstone_empty}\xint:}%
\long\def\XINT_firstone #1#2\xint:{\xint_gob_til_xint: #1\xint:{#1}}%
\def\XINT_firstone_empty\xint:#1{ }%
```

2.18 \xintLengthUpTo

1.2i. for use by \xintKeep and \xintTrim (xinttools). The argument N **must be non-negative**.

\xintLengthUpTo{N}{List} produces -0 if length(List)>N, else it returns N-length(List). Hence subtracting it from N always computes min(N,length(List)).

1.2j. changed ending and interface to core loop.

```
\def\xintLengthUpTo {\romannumeral0\xintlengthupto}%
\long\def\xintlengthupto #1#2%
{%
```

```
    \expandafter\XINT_lengthupto_loop
    \the\numexpr#1.#2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
         \xint_c_vii\xint_c_vi\xint_c_v\xint_c_iv
         \xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye.%
}%
\def\XINT_lengthupto_loop_a #1%
{%
    \xint_UDsignfork
      #1\XINT_lengthupto_gt
      -\XINT_lengthupto_loop
    \krof #1%
}%
\long\def\XINT_lengthupto_gt #1\xint_bye.{-0}%
\long\def\XINT_lengthupto_loop #1.#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint: #9\XINT_lengthupto_finish_a\xint:%
    \expandafter\XINT_lengthupto_loop_a\the\numexpr #1-\xint_c_viii.%
}%
\def\XINT_lengthupto_finish_a\xint:\expandafter\XINT_lengthupto_loop_a
    \the\numexpr #1-\xint_c_viii.#2#3#4#5#6#7#8#9%
{%
    \expandafter\XINT_lengthupto_finish_b\the\numexpr #1-#9\xint_bye
}%
\def\XINT_lengthupto_finish_b #1#2.%
{%
    \xint_UDsignfork
       #1{-0}%
        -{ #1#2}%
    \krof
}%
```

2.19 \xintreplicate, \xintReplicate

1.2i.
This is cloned from LaTeX3's \prg_replicate:nn, see Joseph's post at
http://tex.stackexchange.com/questions/16189/repeat-command-n-times
I posted there an alternative not using the chained \csname's but it is a bit less efficient (except perhaps for thousands of repetitions). The code in Joseph's post does abs(#1) replications when input #1 is negative and then activates an error triggering macro; here we simply do nothing when #1 is negative.

Usage: \romannumeral\xintreplicate{N}{stuff}

When N is already explicit digits (even N=0, but non-negative) one can call the macro as
\romannumeral\XINT_rep N\endcsname {foo}
to skip the \numexpr.

1.4 (2020/01/11). Added \xintReplicate ! The reason I did not before is that the prevailing habits in xint source code was to trigger with \romannumeral0 not \romannumeral which is the lowercased named macros. Thus adding the camelcase one creates a couple \xintReplicate/\xintreplicate not obeying the general mold.

```
\def\xintReplicate{\romannumeral\xintreplicate}%
\def\xintreplicate#1%
   {\expandafter\XINT_replicate\the\numexpr#1\endcsname}%
\def\XINT_replicate #1{\xint_UDsignfork
```

```
                          #1\XINT_rep_neg
                           -\XINT_rep
                        \krof #1}%
\long\def\XINT_rep_neg #1\endcsname #2{\xint_c_}%
\def\XINT_rep   #1{\csname XINT_rep_f#1\XINT_rep_a}%
\def\XINT_rep_a #1{\csname XINT_rep_#1\XINT_rep_a}%
\def\XINT_rep_\XINT_rep_a{\endcsname}%
\long\expandafter\def\csname XINT_rep_0\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}}%
\long\expandafter\def\csname XINT_rep_1\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1}%
\long\expandafter\def\csname XINT_rep_2\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1}%
\long\expandafter\def\csname XINT_rep_3\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1}%
\long\expandafter\def\csname XINT_rep_4\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_5\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_6\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_7\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_8\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_9\endcsname #1%
    {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_f0\endcsname #1%
    {\xint_c_}%
\long\expandafter\def\csname XINT_rep_f1\endcsname #1%
    {\xint_c_ #1}%
\long\expandafter\def\csname XINT_rep_f2\endcsname #1%
    {\xint_c_ #1#1}%
\long\expandafter\def\csname XINT_rep_f3\endcsname #1%
    {\xint_c_ #1#1#1}%
\long\expandafter\def\csname XINT_rep_f4\endcsname #1%
    {\xint_c_ #1#1#1#1}%
\long\expandafter\def\csname XINT_rep_f5\endcsname #1%
    {\xint_c_ #1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_f6\endcsname #1%
    {\xint_c_ #1#1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_f7\endcsname #1%
    {\xint_c_ #1#1#1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_f8\endcsname #1%
    {\xint_c_ #1#1#1#1#1#1#1#1}%
\long\expandafter\def\csname XINT_rep_f9\endcsname #1%
    {\xint_c_ #1#1#1#1#1#1#1#1#1}%
```

2.20 \xintgobble, \xintGobble

1.2i.

I hesitated about allowing as many as 9^6-1=531440 tokens to gobble, but 9^5-1=59058 is too low

for playing with long decimal expansions.
Usage: \romannumeral\xintgobble{N}...

1.4 (2020/01/11). Added \xintGobble.

```
\def\xintGobble{\romannumeral\xintgobble}%
\def\xintgobble #1%
   {\csname xint_c_\expandafter\XINT_gobble_a\the\numexpr#1.0}%
\def\XINT_gobble #1.{\csname xint_c_\XINT_gobble_a #1.0}%
\def\XINT_gobble_a #1{\xint_gob_til_zero#1\XINT_gobble_d0\XINT_gobble_b#1}%
\def\XINT_gobble_b #1.#2%
  {\expandafter\XINT_gobble_c
      \the\numexpr (#1+\xint_c_v)/\xint_c_ix-\xint_c_i\expandafter.%
      \the\numexpr #2+\xint_c_i.#1.}%
\def\XINT_gobble_c #1.#2.#3.%
  {\csname XINT_g#2\the\numexpr#3-\xint_c_ix*#1\relax\XINT_gobble_a #1.#2}%
\def\XINT_gobble_d0\XINT_gobble_b0.#1{\endcsname}%
\expandafter\let\csname XINT_g10\endcsname\endcsname
\long\expandafter\def\csname XINT_g11\endcsname#1{\endcsname}%
\long\expandafter\def\csname XINT_g12\endcsname#1#2{\endcsname}%
\long\expandafter\def\csname XINT_g13\endcsname#1#2#3{\endcsname}%
\long\expandafter\def\csname XINT_g14\endcsname#1#2#3#4{\endcsname}%
\long\expandafter\def\csname XINT_g15\endcsname#1#2#3#4#5{\endcsname}%
\long\expandafter\def\csname XINT_g16\endcsname#1#2#3#4#5#6{\endcsname}%
\long\expandafter\def\csname XINT_g17\endcsname#1#2#3#4#5#6#7{\endcsname}%
\long\expandafter\def\csname XINT_g18\endcsname#1#2#3#4#5#6#7#8{\endcsname}%
\expandafter\let\csname XINT_g20\endcsname\endcsname
\long\expandafter\def\csname XINT_g21\endcsname #1#2#3#4#5#6#7#8#9%
 {\endcsname}%
\long\expandafter\edef\csname XINT_g22\endcsname #1#2#3#4#5#6#7#8#9%
 {\expandafter\noexpand\csname XINT_g21\endcsname}%
\long\expandafter\edef\csname XINT_g23\endcsname #1#2#3#4#5#6#7#8#9%
 {\expandafter\noexpand\csname XINT_g22\endcsname}%
\long\expandafter\edef\csname XINT_g24\endcsname #1#2#3#4#5#6#7#8#9%
 {\expandafter\noexpand\csname XINT_g23\endcsname}%
\long\expandafter\edef\csname XINT_g25\endcsname #1#2#3#4#5#6#7#8#9%
 {\expandafter\noexpand\csname XINT_g24\endcsname}%
\long\expandafter\edef\csname XINT_g26\endcsname #1#2#3#4#5#6#7#8#9%
 {\expandafter\noexpand\csname XINT_g25\endcsname}%
\long\expandafter\edef\csname XINT_g27\endcsname #1#2#3#4#5#6#7#8#9%
 {\expandafter\noexpand\csname XINT_g26\endcsname}%
\long\expandafter\edef\csname XINT_g28\endcsname #1#2#3#4#5#6#7#8#9%
 {\expandafter\noexpand\csname XINT_g27\endcsname}%
\expandafter\let\csname XINT_g30\endcsname\endcsname
\long\expandafter\edef\csname XINT_g31\endcsname #1#2#3#4#5#6#7#8#9%
 {\expandafter\noexpand\csname XINT_g28\endcsname}%
\long\expandafter\edef\csname XINT_g32\endcsname #1#2#3#4#5#6#7#8#9%
 {\noexpand\csname XINT_g31\expandafter\noexpand\csname XINT_g28\endcsname}%
\long\expandafter\edef\csname XINT_g33\endcsname #1#2#3#4#5#6#7#8#9%
 {\noexpand\csname XINT_g32\expandafter\noexpand\csname XINT_g28\endcsname}%
\long\expandafter\edef\csname XINT_g34\endcsname #1#2#3#4#5#6#7#8#9%
 {\noexpand\csname XINT_g33\expandafter\noexpand\csname XINT_g28\endcsname}%
\long\expandafter\edef\csname XINT_g35\endcsname #1#2#3#4#5#6#7#8#9%
 {\noexpand\csname XINT_g34\expandafter\noexpand\csname XINT_g28\endcsname}%
```

```
\long\expandafter\edef\csname XINT_g36\endcsname #1#2#3#4#5#6#7#8#9%
 {\noexpand\csname XINT_g35\expandafter\noexpand\csname XINT_g28\endcsname}%
\long\expandafter\edef\csname XINT_g37\endcsname #1#2#3#4#5#6#7#8#9%
 {\noexpand\csname XINT_g36\expandafter\noexpand\csname XINT_g28\endcsname}%
\long\expandafter\edef\csname XINT_g38\endcsname #1#2#3#4#5#6#7#8#9%
 {\noexpand\csname XINT_g37\expandafter\noexpand\csname XINT_g28\endcsname}%
\expandafter\let\csname XINT_g40\endcsname\endcsname
\expandafter\edef\csname XINT_g41\endcsname
 {\noexpand\csname XINT_g38\expandafter\noexpand\csname XINT_g31\endcsname}%
\expandafter\edef\csname XINT_g42\endcsname
 {\noexpand\csname XINT_g41\expandafter\noexpand\csname XINT_g41\endcsname}%
\expandafter\edef\csname XINT_g43\endcsname
 {\noexpand\csname XINT_g42\expandafter\noexpand\csname XINT_g41\endcsname}%
\expandafter\edef\csname XINT_g44\endcsname
 {\noexpand\csname XINT_g43\expandafter\noexpand\csname XINT_g41\endcsname}%
\expandafter\edef\csname XINT_g45\endcsname
 {\noexpand\csname XINT_g44\expandafter\noexpand\csname XINT_g41\endcsname}%
\expandafter\edef\csname XINT_g46\endcsname
 {\noexpand\csname XINT_g45\expandafter\noexpand\csname XINT_g41\endcsname}%
\expandafter\edef\csname XINT_g47\endcsname
 {\noexpand\csname XINT_g46\expandafter\noexpand\csname XINT_g41\endcsname}%
\expandafter\edef\csname XINT_g48\endcsname
 {\noexpand\csname XINT_g47\expandafter\noexpand\csname XINT_g41\endcsname}%
\expandafter\let\csname XINT_g50\endcsname\endcsname
\expandafter\edef\csname XINT_g51\endcsname
 {\noexpand\csname XINT_g48\expandafter\noexpand\csname XINT_g41\endcsname}%
\expandafter\edef\csname XINT_g52\endcsname
 {\noexpand\csname XINT_g51\expandafter\noexpand\csname XINT_g51\endcsname}%
\expandafter\edef\csname XINT_g53\endcsname
 {\noexpand\csname XINT_g52\expandafter\noexpand\csname XINT_g51\endcsname}%
\expandafter\edef\csname XINT_g54\endcsname
 {\noexpand\csname XINT_g53\expandafter\noexpand\csname XINT_g51\endcsname}%
\expandafter\edef\csname XINT_g55\endcsname
 {\noexpand\csname XINT_g54\expandafter\noexpand\csname XINT_g51\endcsname}%
\expandafter\edef\csname XINT_g56\endcsname
 {\noexpand\csname XINT_g55\expandafter\noexpand\csname XINT_g51\endcsname}%
\expandafter\edef\csname XINT_g57\endcsname
 {\noexpand\csname XINT_g56\expandafter\noexpand\csname XINT_g51\endcsname}%
\expandafter\edef\csname XINT_g58\endcsname
 {\noexpand\csname XINT_g57\expandafter\noexpand\csname XINT_g51\endcsname}%
\expandafter\let\csname XINT_g60\endcsname\endcsname
\expandafter\edef\csname XINT_g61\endcsname
 {\noexpand\csname XINT_g58\expandafter\noexpand\csname XINT_g51\endcsname}%
\expandafter\edef\csname XINT_g62\endcsname
 {\noexpand\csname XINT_g61\expandafter\noexpand\csname XINT_g61\endcsname}%
\expandafter\edef\csname XINT_g63\endcsname
 {\noexpand\csname XINT_g62\expandafter\noexpand\csname XINT_g61\endcsname}%
\expandafter\edef\csname XINT_g64\endcsname
 {\noexpand\csname XINT_g63\expandafter\noexpand\csname XINT_g61\endcsname}%
\expandafter\edef\csname XINT_g65\endcsname
 {\noexpand\csname XINT_g64\expandafter\noexpand\csname XINT_g61\endcsname}%
\expandafter\edef\csname XINT_g66\endcsname
```

```
 {\noexpand\csname XINT_g65\expandafter\noexpand\csname XINT_g61\endcsname}%
\expandafter\edef\csname XINT_g67\endcsname
 {\noexpand\csname XINT_g66\expandafter\noexpand\csname XINT_g61\endcsname}%
\expandafter\edef\csname XINT_g68\endcsname
 {\noexpand\csname XINT_g67\expandafter\noexpand\csname XINT_g61\endcsname}%
```

2.21 (WIP) \xintUniformDeviate

1.3b. See user manual for related information.

```
\ifdefined\xint_texuniformdeviate
    \expandafter\xint_firstoftwo
\else\expandafter\xint_secondoftwo
\fi
{%
  \def\xintUniformDeviate#1%
     {\the\numexpr\expandafter\XINT_uniformdeviate_sgnfork\the\numexpr#1\xint:}%
  \def\XINT_uniformdeviate_sgnfork#1%
  {%
     \if-#1\XINT_uniformdeviate_neg\fi \XINT_uniformdeviate{}#1%
  }%
  \def\XINT_uniformdeviate_neg\fi\XINT_uniformdeviate#1-%
  {%
      \fi-\numexpr\XINT_uniformdeviate\relax
  }%
  \def\XINT_uniformdeviate#1#2\xint:
  {%(
      \expandafter\XINT_uniformdeviate_a\the\numexpr%
                    -\xint_texuniformdeviate\xint_c_ii^vii%
                    -\xint_c_ii^vii*\xint_texuniformdeviate\xint_c_ii^vii%
                    -\xint_c_ii^xiv*\xint_texuniformdeviate\xint_c_ii^vii%
                    -\xint_c_ii^xxi*\xint_texuniformdeviate\xint_c_ii^vii%
                    +\xint_texuniformdeviate#2\xint:/#2)*#2\xint:+#2\fi\relax#1%
  }%
  \def\XINT_uniformdeviate_a #1\xint:
  {%
      \expandafter\XINT_uniformdeviate_b\the\numexpr#1-(#1%
  }%
  \def\XINT_uniformdeviate_b#1#2\xint:{#1#2\if-#1}%
}%
{%
  \def\xintUniformDeviate#1%
  {%
      \the\numexpr
      \XINT_expandableerror{No uniformdeviate at engine level, returning 0.}%
      0\relax
  }%
}%
```

2.22 \xintMessage, \ifxintverbose

1.2c. for use by \xintdefvar and \xintdeffunc of xintexpr.

1.2e. uses \write128 rather than \write16 for compatibility with future extended range of output streams, in LuaTeX in particular.

1.3e. set the \newlinechar.

```
\def\xintMessage #1#2#3{%
    \edef\XINT_newlinechar{\the\newlinechar}%
    \newlinechar10
    \immediate\write128{Package #1 #2: (on line \the\inputlineno)}%
    \immediate\write128{\space\space\space\space#3}%
    \newlinechar\XINT_newlinechar\space
}%
\newif\ifxintverbose
```

2.23 \ifxintglobaldefs, \XINT_global

1.3c.

```
\newif\ifxintglobaldefs
\def\XINT_global{\ifxintglobaldefs\global\fi}%
```

2.24 (WIP) Expandable error message

1.21. but really belongs to next major release beyond 1.3.
This is copied over from l3kernel code. I am using \ ! / control sequence though, which must be left undefined. \xintError: would be 6 letters more already.

1.4 (2020/01/25). Finally rather than \ ! / I use \xint/.

```
\def\XINT_expandableerror #1#2{%
    \def\XINT_expandableerror ##1{%
        \expandafter\expandafter\expandafter
        \XINT_expandableerror_continue\xint_firstofone{#2#1##1#1}}%
    \def\XINT_expandableerror_continue ##1#1##2#1{##1}%
}%
\begingroup\lccode`$ 32 \catcode`/ 11 % $
\lowercase{\endgroup\XINT_expandableerror$\xint/\let\xint/\xint_undefined}% $
\XINT_restorecatcodes_endinput%
```

3 Package xinttools implementation

Release 1.09g of 2013/11/22 splits off xinttools.sty from xint.sty. Starting with 1.1, xinttools ceases being loaded automatically by xint.

3.1 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from Heiko Oberdiek's packages, then modified. The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xinttools.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintkernel.sty\endcsname
```

```
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xinttools}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xinttools.sty
      \ifx\w\relax % but xintkernel.sty not yet loaded.
         \def\z{\endgroup\input xintkernel.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xintkernel.sty not yet loaded.
            \def\z{\endgroup\RequirePackage{xintkernel}}%
          \fi
      \else
        \aftergroup\endinput % xinttools already loaded.
      \fi
    \fi
  \fi
\z%
\XINTsetupcatcodes% defined in xintkernel.sty
```

3.2 Package identification

```
\XINT_providespackage
\ProvidesPackage{xinttools}%
  [2021/03/29 v1.4d Expandable and non-expandable utilities (JFB)]%
```

\XINT_toks is used in macros such as \xintFor. It is not used elsewhere in the xint bundle.

```
\newtoks\XINT_toks
\xint_firstofone{\let\XINT_sptoken= } %<- space here!
```

3.3 \xintgodef, \xintgoodef, \xintgfdef

1.09i. For use in \xintAssign.

```
\def\xintgodef  {\global\xintodef }%
\def\xintgoodef {\global\xintoodef }%
\def\xintgfdef  {\global\xintfdef }%
```

3.4 \xintRevWithBraces

New with 1.06. Makes the expansion of its argument and then reverses the resulting tokens or braced tokens, adding a pair of braces to each (thus, maintaining it when it was already there.) The reason for \xint:, here and in other locations, is in case #1 expands to nothing, the \romannumeral-`0 must be stopped

```
\def\xintRevWithBraces         {\romannumeral0\xintrevwithbraces }%
\def\xintRevWithBracesNoExpand {\romannumeral0\xintrevwithbracesnoexpand }%
\long\def\xintrevwithbraces #1%
{%
    \expandafter\XINT_revwbr_loop\expandafter{\expandafter}%
    \romannumeral`&&@#1\xint:\xint:\xint:\xint:%
                       \xint:\xint:\xint:\xint:\xint_bye
}%
\long\def\xintrevwithbracesnoexpand #1%
{%
    \XINT_revwbr_loop {}%
    #1\xint:\xint:\xint:\xint:%
      \xint:\xint:\xint:\xint:\xint_bye
}%
\long\def\XINT_revwbr_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint: #9\XINT_revwbr_finish_a\xint:%
    \XINT_revwbr_loop {{#9}{#8}{#7}{#6}{#5}{#4}{#3}{#2}#1}%
}%
\long\def\XINT_revwbr_finish_a\xint:\XINT_revwbr_loop #1#2\xint_bye
{%
    \XINT_revwbr_finish_b #2\R\R\R\R\R\R\R\Z #1%
}%
\def\XINT_revwbr_finish_b #1#2#3#4#5#6#7#8\Z
{%
    \xint_gob_til_R
            #1\XINT_revwbr_finish_c \xint_gobble_viii
            #2\XINT_revwbr_finish_c \xint_gobble_vii
            #3\XINT_revwbr_finish_c \xint_gobble_vi
            #4\XINT_revwbr_finish_c \xint_gobble_v
            #5\XINT_revwbr_finish_c \xint_gobble_iv
            #6\XINT_revwbr_finish_c \xint_gobble_iii
            #7\XINT_revwbr_finish_c \xint_gobble_ii
            \R\XINT_revwbr_finish_c \xint_gobble_i\Z
}%
```

1.1c revisited this old code and improved upon the earlier endings.

```
\def\XINT_revwbr_finish_c#1{%
\def\XINT_revwbr_finish_c##1##2\Z{\expandafter#1##1}%
}\XINT_revwbr_finish_c{ }%
```

3.5 \xintZapFirstSpaces

1.09f, written [2013/11/01]. Modified (2014/10/21) for release 1.1 to correct the bug in case of an empty argument, or argument containing only spaces, which had been forgotten in first version. New version is simpler than the initial one. This macro does NOT expand its argument.

```
\def\xintZapFirstSpaces {\romannumeral0\xintzapfirstspaces }%
\def\xintzapfirstspaces#1{\long
\def\xintzapfirstspaces ##1{\XINT_zapbsp_a #1##1\xint:#1#1\xint:}%
}\xintzapfirstspaces{ }%
```

If the original #1 started with a space, the grabbed #1 is empty. Thus _again? will see #1=\xint_bye, and hand over control to _again which will loop back into \XINT_zapbsp_a, with one initial space less. If the original #1 did not start with a space, or was empty, then the #1 below will be a <sptoken>, then an extract of the original #1, not empty and not starting with a space, which contains what was up to the first <sp><sp> present in original #1, or, if none preexisted, <sptoken> and all of #1 (possibly empty) plus an ending \xint:. The added initial space will stop later the \romannumeral0. No brace stripping is possible. Control is handed over to \XINT_zapbsp_b which strips out the ending \xint:<sp><sp>\xint:

```
\def\XINT_zapbsp_a#1{\long\def\XINT_zapbsp_a ##1#1#1{%
  \XINT_zapbsp_again?##1\xint_bye\XINT_zapbsp_b ##1#1#1}%
}\XINT_zapbsp_a{ }%
\long\def\XINT_zapbsp_again? #1{\xint_bye #1\XINT_zapbsp_again }%
\xint_firstofone{\def\XINT_zapbsp_again\XINT_zapbsp_b} {\XINT_zapbsp_a }%
\long\def\XINT_zapbsp_b #1\xint:#2\xint:{#1}%
```

3.6 \xintZapLastSpaces

1.09f, written [2013/11/01].

```
\def\xintZapLastSpaces {\romannumeral0\xintzaplastspaces }%
\def\xintzaplastspaces#1{\long
\def\xintzaplastspaces ##1{\XINT_zapesp_a {}\empty##1#1#1\xint_bye\xint:}%
}\xintzaplastspaces{ }%
```

The \empty from \xintzaplastspaces is to prevent brace removal in the #2 below. The \expandafter chain removes it.

```
\xint_firstofone {\long\def\XINT_zapesp_a #1#2 } %<- second space here
    {\expandafter\XINT_zapesp_b\expandafter{#2}{#1}}%
```

Notice again an \empty added here. This is in preparation for possibly looping back to \XINT_zapesp_a. If the initial #1 had no <sp><sp>, the stuff however will not loop, because #3 will already be <some spaces>\xint_bye. Notice that this macro fetches all way to the ending \xint:. This looks not very efficient, but how often do we have to strip ending spaces from something which also has inner stretches of _multiple_ space tokens ?;-).

```
\long\def\XINT_zapesp_b #1#2#3\xint:%
    {\XINT_zapesp_end? #3\XINT_zapesp_e {#2#1}\empty #3\xint:}%
```

When we have been over all possible <sp><sp> things, we reach the ending space tokens, and #3 will be a bunch of spaces (possibly none) followed by \xint_bye. So the #1 in _end? will be \xint_bye. In all other cases #1 can not be \xint_bye (assuming naturally this token does nor arise in original input), hence control falls back to \XINT_zapesp_e which will loop back to \XINT_zapesp_a.

```
\long\def\XINT_zapesp_end? #1{\xint_bye #1\XINT_zapesp_end }%
```

We are done. The #1 here has accumulated all the previous material, and is stripped of its ending spaces, if any.

```
\long\def\XINT_zapesp_end\XINT_zapesp_e #1#2\xint:{ #1}%
```

We haven't yet reached the end, so we need to re-inject two space tokens after what we have gotten so far. Then we loop.

```
\def\XINT_zapesp_e#1{%
\long\def\XINT_zapesp_e ##1{\XINT_zapesp_a {##1#1#1}}%
}\XINT_zapesp_e{ }%
```

3.7 \xintZapSpaces

1.09f, written [2013/11/01]. Modified for 1.1, 2014/10/21 as it has the same bug as \xintZapFirstSpaces. We in effect do first \xintZapFirstSpaces, then \xintZapLastSpaces.

```
\def\xintZapSpaces {\romannumeral0\xintzapspaces }%
\def\xintzapspaces#1{%
\long\def\xintzapspaces ##1% like \xintZapFirstSpaces.
        {\XINT_zapsp_a #1##1\xint:#1#1\xint:}%
}\xintzapspaces{ }%
\def\XINT_zapsp_a#1{%
\long\def\XINT_zapsp_a ##1#1#1%
        {\XINT_zapsp_again?##1\xint_bye\XINT_zapsp_b##1#1#1}%
}\XINT_zapsp_a{ }%
\long\def\XINT_zapsp_again? #1{\xint_bye #1\XINT_zapsp_again }%
\xint_firstofone{\def\XINT_zapsp_again\XINT_zapsp_b} {\XINT_zapsp_a }%
\xint_firstofone{\def\XINT_zapsp_b} {\XINT_zapsp_c }%
\def\XINT_zapsp_c#1{%
\long\def\XINT_zapsp_c ##1\xint:##2\xint:%
        {\XINT_zapesp_a{}\empty ##1#1#1\xint_bye\xint:}%
}\XINT_zapsp_c{ }%
```

3.8 \xintZapSpacesB

1.09f, written [2013/11/01]. Strips up to one pair of braces (but then does not strip spaces inside).

```
\def\xintZapSpacesB {\romannumeral0\xintzapspacesb }%
\long\def\xintzapspacesb #1{\XINT_zapspb_one? #1\xint:\xint:%
                            \xint_bye\xintzapspaces {#1}}%
\long\def\XINT_zapspb_one? #1#2%
   {\xint_gob_til_xint: #1\XINT_zapspb_onlyspaces\xint:%
    \xint_gob_til_xint: #2\XINT_zapspb_bracedorone\xint:%
    \xint_bye {#1}}%
\def\XINT_zapspb_onlyspaces\xint:%
    \xint_gob_til_xint:\xint:\XINT_zapspb_bracedorone\xint:%
    \xint_bye #1\xint_bye\xintzapspaces #2{ }%
\long\def\XINT_zapspb_bracedorone\xint:%
    \xint_bye #1\xint:\xint_bye\xintzapspaces #2{ #1}%
```

3.9 \xintCSVtoList, \xintCSVtoListNonStripped

\xintCSVtoList transforms a,b,..,z into {a}{b}...{z}. The comma separated list may be a macro which is first f-expanded. First included in release 1.06. Here, use of \Z (and \R) perfectly safe.

[2013/11/02]: Starting with 1.09f, automatically filters items with \xintZapSpacesB to strip away all spaces around commas, and spaces at the start and end of the list. The original is kept as \xintCSVtoListNonStripped, and is faster. But ... it doesn't strip spaces.

ATTENTION: if the input is empty the output contains one item (empty, of course). This means an \xintFor loop always executes at least once the iteration, contrarily to \xintFor*.

```
\def\xintCSVtoList {\romannumeral0\xintcsvtolist }%
\long\def\xintcsvtolist #1{\expandafter\xintApply
```

```
            \expandafter\xintzapspacesb
            \expandafter{\romannumeral0\xintcsvtolistnonstripped{#1}}}%
\def\xintCSVtoListNoExpand {\romannumeral0\xintcsvtolistnoexpand }%
\long\def\xintcsvtolistnoexpand #1{\expandafter\xintApply
            \expandafter\xintzapspacesb
            \expandafter{\romannumeral0\xintcsvtolistnonstrippednoexpand{#1}}}%
\def\xintCSVtoListNonStripped {\romannumeral0\xintcsvtolistnonstripped }%
\def\xintCSVtoListNonStrippedNoExpand
          {\romannumeral0\xintcsvtolistnonstrippednoexpand }%
\long\def\xintcsvtolistnonstripped #1%
{%
    \expandafter\XINT_csvtol_loop_a\expandafter
    {\expandafter}\romannumeral`&&@#1%
        ,\xint_bye,\xint_bye,\xint_bye,\xint_bye
        ,\xint_bye,\xint_bye,\xint_bye,\xint_bye,\Z
}%
\long\def\xintcsvtolistnonstrippednoexpand #1%
{%
    \XINT_csvtol_loop_a
    {}#1,\xint_bye,\xint_bye,\xint_bye,\xint_bye
        ,\xint_bye,\xint_bye,\xint_bye,\xint_bye,\Z
}%
\long\def\XINT_csvtol_loop_a #1#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    \xint_bye #9\XINT_csvtol_finish_a\xint_bye
    \XINT_csvtol_loop_b {#1}{{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}%
}%
\long\def\XINT_csvtol_loop_b #1#2{\XINT_csvtol_loop_a {#1#2}}%
\long\def\XINT_csvtol_finish_a\xint_bye\XINT_csvtol_loop_b #1#2#3\Z
{%
    \XINT_csvtol_finish_b #3\R,\R,\R,\R,\R,\R,\Z #2{#1}%
}%
```

1.1c revisits this old code and improves upon the earlier endings. But as the _d.. macros have already nine parameters, I needed the \expandafter and \xint_gob_til_Z in finish_b (compare \XINT_keep_endb, or also \XINT_RQ_end_b).

```
\def\XINT_csvtol_finish_b #1,#2,#3,#4,#5,#6,#7,#8\Z
{%
    \xint_gob_til_R
             #1\expandafter\XINT_csvtol_finish_dviii\xint_gob_til_Z
             #2\expandafter\XINT_csvtol_finish_dvii \xint_gob_til_Z
             #3\expandafter\XINT_csvtol_finish_dvi  \xint_gob_til_Z
             #4\expandafter\XINT_csvtol_finish_dv   \xint_gob_til_Z
             #5\expandafter\XINT_csvtol_finish_div  \xint_gob_til_Z
             #6\expandafter\XINT_csvtol_finish_diii \xint_gob_til_Z
             #7\expandafter\XINT_csvtol_finish_dii  \xint_gob_til_Z
             \R\XINT_csvtol_finish_di \Z
}%
\long\def\XINT_csvtol_finish_dviii #1#2#3#4#5#6#7#8#9{ #9}%
\long\def\XINT_csvtol_finish_dvii  #1#2#3#4#5#6#7#8#9{ #9{#1}}%
\long\def\XINT_csvtol_finish_dvi   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}}%
\long\def\XINT_csvtol_finish_dv    #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}}%
```

```
\long\def\XINT_csvtol_finish_div   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}}%
\long\def\XINT_csvtol_finish_diii  #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}{#5}}%
\long\def\XINT_csvtol_finish_dii   #1#2#3#4#5#6#7#8#9%
                                           { #9{#1}{#2}{#3}{#4}{#5}{#6}}%
\long\def\XINT_csvtol_finish_di\Z  #1#2#3#4#5#6#7#8#9%
                                           { #9{#1}{#2}{#3}{#4}{#5}{#6}{#7}}%
```

3.10 \xintListWithSep

1.04. \xintListWithSep {\sep}{{a}{b}...{z}} returns a \sep b \sep ....\sep z. It f-expands its second argument. The 'sep' may be \par's: the macro \xintlistwithsep etc... are all declared long. 'sep' does not have to be a single token. It is not expanded. The "list" argument may be empty.

\xintListWithSepNoExpand does not f-expand its second argument.

This venerable macro from 1.04 remained unchanged for a long time and was finally refactored at 1.2p for increased speed. Tests done with a list of identical {\x} items and a sep of \z demonstrated a speed increase of about:

- 3x for 30 items,
- 4.5x for 100 items,
- 7.5x--8x for 1000 items.

```
\def\xintListWithSep         {\romannumeral0\xintlistwithsep }%
\def\xintListWithSepNoExpand {\romannumeral0\xintlistwithsepnoexpand }%
\long\def\xintlistwithsep #1#2%
    {\expandafter\XINT_lws\expandafter {\romannumeral`&&@#2}{#1}}%
\long\def\xintlistwithsepnoexpand #1#2%
{%
    \XINT_lws_loop_a {#1}#2{\xint_bye\XINT_lws_e_vi}%
       {\xint_bye\XINT_lws_e_v}{\xint_bye\XINT_lws_e_iv}%
       {\xint_bye\XINT_lws_e_iii}{\xint_bye\XINT_lws_e_ii}%
       {\xint_bye\XINT_lws_e_i}{\xint_bye\XINT_lws_e}%
       {\xint_bye\expandafter\space}\xint_bye
}%
\long\def\XINT_lws #1#2%
{%
    \XINT_lws_loop_a {#2}#1{\xint_bye\XINT_lws_e_vi}%
       {\xint_bye\XINT_lws_e_v}{\xint_bye\XINT_lws_e_iv}%
       {\xint_bye\XINT_lws_e_iii}{\xint_bye\XINT_lws_e_ii}%
       {\xint_bye\XINT_lws_e_i}{\xint_bye\XINT_lws_e}%
       {\xint_bye\expandafter\space}\xint_bye
}%
\long\def\XINT_lws_loop_a #1#2#3#4#5#6#7#8#9%
{%
    \xint_bye #9\xint_bye
    \XINT_lws_loop_b {#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}%
}%
\long\def\XINT_lws_loop_b #1#2#3#4#5#6#7#8#9%
{%
    \XINT_lws_loop_a {#1}{#2#1#3#1#4#1#5#1#6#1#7#1#8#1#9}%
}%
\long\def\XINT_lws_e_vi\xint_bye\XINT_lws_loop_b #1#2#3#4#5#6#7#8#9\xint_bye
    { #2#1#3#1#4#1#5#1#6#1#7#1#8}%
\long\def\XINT_lws_e_v\xint_bye\XINT_lws_loop_b #1#2#3#4#5#6#7#8\xint_bye
    { #2#1#3#1#4#1#5#1#6#1#7}%
```

```
\long\def\XINT_lws_e_iv\xint_bye\XINT_lws_loop_b #1#2#3#4#5#6#7\xint_bye
    { #2#1#3#1#4#1#5#1#6}%
\long\def\XINT_lws_e_iii\xint_bye\XINT_lws_loop_b #1#2#3#4#5#6\xint_bye
    { #2#1#3#1#4#1#5}%
\long\def\XINT_lws_e_ii\xint_bye\XINT_lws_loop_b #1#2#3#4#5\xint_bye
    { #2#1#3#1#4}%
\long\def\XINT_lws_e_i\xint_bye\XINT_lws_loop_b #1#2#3#4\xint_bye
    { #2#1#3}%
\long\def\XINT_lws_e\xint_bye\XINT_lws_loop_b #1#2#3\xint_bye
    { #2}%
```

3.11 \xintNthElt

First included in release 1.06. Last refactored in 1.2j.
  \xintNthElt {i}{List} returns the i th item from List (one pair of braces removed). The list is first f-expanded. The \xintNthEltNoExpand does no expansion of its second argument. Both variants expand i inside \numexpr.
  With i = 0, the number of items is returned using \xintLength but with the List argument f-expanded first.
  Negative values return the |i|th element from the end.
  When i is out of range, an empty value is returned.

```
\def\xintNthElt         {\romannumeral0\xintnthelt }%
\def\xintNthEltNoExpand {\romannumeral0\xintntheltnoexpand }%
\long\def\xintnthelt #1#2{\expandafter\XINT_nthelt_a\the\numexpr #1\expandafter.%
                         \expandafter{\romannumeral`&&@#2}}%
\def\xintntheltnoexpand #1{\expandafter\XINT_nthelt_a\the\numexpr #1.}%
\def\XINT_nthelt_a #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_nthelt_zero
        0#1\XINT_nthelt_neg
         0-{\XINT_nthelt_pos #1}%
    \krof
}%
\def\XINT_nthelt_zero #1.{\xintlength }%
\long\def\XINT_nthelt_neg #1.#2%
{%
    \expandafter\XINT_nthelt_neg_a\the\numexpr\xint_c_i+\XINT_length_loop
    #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    -#1.#2\xint_bye
}%
\def\XINT_nthelt_neg_a #1%
{%
    \xint_UDzerominusfork
        #1-\xint_stop_afterbye
        0#1\xint_stop_afterbye
         0-{}%
    \krof
    \expandafter\XINT_nthelt_neg_b
    \romannumeral\expandafter\XINT_gobble\the\numexpr-\xint_c_i+#1%
```

```
}%
\long\def\XINT_nthelt_neg_b #1#2\xint_bye{ #1}%
\long\def\XINT_nthelt_pos #1.#2%
{%
    \expandafter\XINT_nthelt_pos_done
    \romannumeral0\expandafter\XINT_trim_loop\the\numexpr#1-\xint_c_x.%
     #2\xint:\xint:\xint:\xint:\xint:%
       \xint:\xint:\xint:\xint:\xint:%
    \xint_bye
}%
\def\XINT_nthelt_pos_done #1{%
\long\def\XINT_nthelt_pos_done ##1##2\xint_bye{%
  \xint_gob_til_xint:##1\expandafter#1\xint_gobble_ii\xint:#1##1}%
}\XINT_nthelt_pos_done{ }%
```

3.12 \xintNthOnePy

First included in release 1.4. See relevant code comments in xintexpr.

```
\def\xintNthOnePy         {\romannumeral0\xintnthonepy }%
\def\xintNthOnePyNoExpand {\romannumeral0\xintnthonepynoexpand }%
\long\def\xintnthonepy #1#2{\expandafter\XINT_nthonepy_a\the\numexpr #1\expandafter.%
                           \expandafter{\romannumeral`&&@#2}}%
\def\xintnthonepynoexpand #1{\expandafter\XINT_nthonepy_a\the\numexpr #1.}%
\def\XINT_nthonepy_a #1%
{%
    \xint_UDsignfork
        #1\XINT_nthonepy_neg
         -{\XINT_nthonepy_nonneg #1}%
    \krof
}%
\long\def\XINT_nthonepy_neg #1.#2%
{%
    \expandafter\XINT_nthonepy_neg_a\the\numexpr\xint_c_i+\XINT_length_loop
    #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    -#1.#2\xint_bye
}%
\def\XINT_nthonepy_neg_a #1%
{%
    \xint_UDzerominusfork
        #1-\xint_stop_afterbye
        0#1\xint_stop_afterbye
         0-{}%
    \krof
    \expandafter\XINT_nthonepy_neg_b
    \romannumeral\expandafter\XINT_gobble\the\numexpr-\xint_c_i+#1%
}%
\long\def\XINT_nthonepy_neg_b #1#2\xint_bye{{#1}}%
\long\def\XINT_nthonepy_nonneg #1.#2%
{%
    \expandafter\XINT_nthonepy_nonneg_done
```

```
    \romannumeral0\expandafter\XINT_trim_loop\the\numexpr#1-\xint_c_ix.%
     #2\xint:\xint:\xint:\xint:\xint:%
       \xint:\xint:\xint:\xint:\xint:%
    \xint_bye
}%
\def\XINT_nthonepy_nonneg_done #1{%
\long\def\XINT_nthonepy_nonneg_done ##1##2\xint_bye{%
  \xint_gob_til_xint:##1\expandafter#1\xint_gobble_ii\xint:{##1}}%
}\XINT_nthonepy_nonneg_done{ }%
```

3.13 \xintKeep

First included in release 1.09m.

\xintKeep{i}{L} f-expands its second argument L. It then grabs the first i items from L and discards the rest.

ATTENTION: **each such kept item is returned inside a brace pair** Use \xintKeepUnbraced to avoid that.

For i equal or larger to the number N of items in (expanded) L, the full L is returned (with braced items). For i=0, the macro returns an empty output. For i<0, the macro discards the first N-|i| items. No brace pairs added to the remaining items. For i is less or equal to -N, the full L is returned (with no braces added.)

\xintKeepNoExpand does not expand the L argument.

Prior to 1.2i the code proceeded along a loop with no pre-computation of the length of L, for the i>0 case. The faster 1.2i version takes advantage of novel \xintLengthUpTo from xintkernel.sty.

```
\def\xintKeep          {\romannumeral0\xintkeep }%
\def\xintKeepNoExpand {\romannumeral0\xintkeepnoexpand }%
\long\def\xintkeep #1#2{\expandafter\XINT_keep_a\the\numexpr #1\expandafter.%
                        \expandafter{\romannumeral`&&@#2}}%
\def\xintkeepnoexpand #1{\expandafter\XINT_keep_a\the\numexpr #1.}%
\def\XINT_keep_a #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_keep_keepnone
        0#1\XINT_keep_neg
         0-{\XINT_keep_pos #1}%
    \krof
}%
\long\def\XINT_keep_keepnone .#1{ }%
\long\def\XINT_keep_neg #1.#2%
{%
    \expandafter\XINT_keep_neg_a\the\numexpr
    #1-\numexpr\XINT_length_loop
    #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye.#2%
}%
\def\XINT_keep_neg_a #1%
{%
    \xint_UDsignfork
        #1{\expandafter\space\romannumeral\XINT_gobble}%
         -\XINT_keep_keepall
    \krof
```

```
}%
\def\XINT_keep_keepall #1.{ }%
\long\def\XINT_keep_pos #1.#2%
{%
    \expandafter\XINT_keep_loop
    \the\numexpr#1-\XINT_lengthupto_loop
    #1.#2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
         \xint_c_vii\xint_c_vi\xint_c_v\xint_c_iv
         \xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye.%
    -\xint_c_viii.{}#2\xint_bye%
}%
\def\XINT_keep_loop #1#2.%
{%
    \xint_gob_til_minus#1\XINT_keep_loop_end-%
    \expandafter\XINT_keep_loop
    \the\numexpr#1#2-\xint_c_viii\expandafter.\XINT_keep_loop_pickeight
}%
\long\def\XINT_keep_loop_pickeight
     #1#2#3#4#5#6#7#8#9{{#1{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}}%
\def\XINT_keep_loop_end-\expandafter\XINT_keep_loop
    \the\numexpr-#1-\xint_c_viii\expandafter.\XINT_keep_loop_pickeight
    {\csname XINT_keep_end#1\endcsname}%
\long\expandafter\def\csname XINT_keep_end1\endcsname
   #1#2#3#4#5#6#7#8#9\xint_bye { #1{#2}{#3}{#4}{#5}{#6}{#7}{#8}}%
\long\expandafter\def\csname XINT_keep_end2\endcsname
   #1#2#3#4#5#6#7#8\xint_bye { #1{#2}{#3}{#4}{#5}{#6}{#7}}%
\long\expandafter\def\csname XINT_keep_end3\endcsname
   #1#2#3#4#5#6#7\xint_bye { #1{#2}{#3}{#4}{#5}{#6}}%
\long\expandafter\def\csname XINT_keep_end4\endcsname
   #1#2#3#4#5#6\xint_bye { #1{#2}{#3}{#4}{#5}}%
\long\expandafter\def\csname XINT_keep_end5\endcsname
   #1#2#3#4#5\xint_bye { #1{#2}{#3}{#4}}%
\long\expandafter\def\csname XINT_keep_end6\endcsname
   #1#2#3#4\xint_bye { #1{#2}{#3}}%
\long\expandafter\def\csname XINT_keep_end7\endcsname
   #1#2#3\xint_bye { #1{#2}}%
\long\expandafter\def\csname XINT_keep_end8\endcsname
   #1#2\xint_bye { #1}%
```

3.14 \xintKeepUnbraced

1.2a. Same as \xintKeep but will *not* add (or maintain) brace pairs around the kept items when length(L)>i>0.
  The name may cause a mis-understanding: for i<0, (i.e. keeping only trailing items), there is no brace removal at all happening.
  Modified for 1.2i like \xintKeep.

```
\def\xintKeepUnbraced         {\romannumeral0\xintkeepunbraced }%
\def\xintKeepUnbracedNoExpand {\romannumeral0\xintkeepunbracednoexpand }%
\long\def\xintkeepunbraced #1#2%
    {\expandafter\XINT_keepunbr_a\the\numexpr #1\expandafter.%
                          \expandafter{\romannumeral`&&@#2}}%
\def\xintkeepunbracednoexpand #1%
```

```
    {\expandafter\XINT_keepunbr_a\the\numexpr #1.}%
\def\XINT_keepunbr_a #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_keep_keepnone
        0#1\XINT_keep_neg
         0-{\XINT_keepunbr_pos #1}%
    \krof
}%
\long\def\XINT_keepunbr_pos #1.#2%
{%
    \expandafter\XINT_keepunbr_loop
    \the\numexpr#1-\XINT_lengthupto_loop
    #1.#2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
         \xint_c_vii\xint_c_vi\xint_c_v\xint_c_iv
         \xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye.%
    -\xint_c_viii.{}#2\xint_bye%
}%
\def\XINT_keepunbr_loop #1#2.%
{%
    \xint_gob_til_minus#1\XINT_keepunbr_loop_end-%
    \expandafter\XINT_keepunbr_loop
    \the\numexpr#1#2-\xint_c_viii\expandafter.\XINT_keepunbr_loop_pickeight
}%
\long\def\XINT_keepunbr_loop_pickeight
     #1#2#3#4#5#6#7#8#9{{#1#2#3#4#5#6#7#8#9}}%
\def\XINT_keepunbr_loop_end-\expandafter\XINT_keepunbr_loop
    \the\numexpr-#1-\xint_c_viii\expandafter.\XINT_keepunbr_loop_pickeight
    {\csname XINT_keepunbr_end#1\endcsname}%
\long\expandafter\def\csname XINT_keepunbr_end1\endcsname
   #1#2#3#4#5#6#7#8#9\xint_bye { #1#2#3#4#5#6#7#8}%
\long\expandafter\def\csname XINT_keepunbr_end2\endcsname
   #1#2#3#4#5#6#7#8\xint_bye { #1#2#3#4#5#6#7}%
\long\expandafter\def\csname XINT_keepunbr_end3\endcsname
   #1#2#3#4#5#6#7\xint_bye { #1#2#3#4#5#6}%
\long\expandafter\def\csname XINT_keepunbr_end4\endcsname
   #1#2#3#4#5#6\xint_bye { #1#2#3#4#5}%
\long\expandafter\def\csname XINT_keepunbr_end5\endcsname
   #1#2#3#4#5\xint_bye { #1#2#3#4}%
\long\expandafter\def\csname XINT_keepunbr_end6\endcsname
   #1#2#3#4\xint_bye { #1#2#3}%
\long\expandafter\def\csname XINT_keepunbr_end7\endcsname
   #1#2#3\xint_bye { #1#2}%
\long\expandafter\def\csname XINT_keepunbr_end8\endcsname
   #1#2\xint_bye { #1}%
```

3.15 \xintTrim

First included in release 1.09m.

\xintTrim{i}{L} f-expands its second argument L. It then removes the first i items from L and keeps the rest. For i equal or larger to the number N of items in (expanded) L, the macro returns an empty output. For i=0, the original (expanded) L is returned. For i<0, the macro proceeds from

the tail. It thus removes the last |i| items, i.e. it keeps the first N-|i| items. For |i|>= N, the empty list is returned.
  \xintTrimNoExpand does not expand the L argument.
  Speed improvements with 1.2i for i<0 branch (which hands over to \xintKeep). Speed improvements with 1.2j for i>0 branch which gobbles items nine by nine despite not knowing in advance if it will go too far.

```
\def\xintTrim         {\romannumeral0\xinttrim }%
\def\xintTrimNoExpand {\romannumeral0\xinttrimnoexpand }%
\long\def\xinttrim #1#2{\expandafter\XINT_trim_a\the\numexpr #1\expandafter.%
                        \expandafter{\romannumeral`&&@#2}}%
\def\xinttrimnoexpand #1{\expandafter\XINT_trim_a\the\numexpr #1.}%
\def\XINT_trim_a #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_trim_trimnone
        0#1\XINT_trim_neg
         0-{\XINT_trim_pos #1}%
    \krof
}%
\long\def\XINT_trim_trimnone .#1{ #1}%
\long\def\XINT_trim_neg #1.#2%
{%
    \expandafter\XINT_trim_neg_a\the\numexpr
    #1-\numexpr\XINT_length_loop
    #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    .{}#2\xint_bye
}%
\def\XINT_trim_neg_a #1%
{%
    \xint_UDsignfork
        #1{\expandafter\XINT_keep_loop\the\numexpr-\xint_c_viii+}%
         -\XINT_trim_trimall
    \krof
}%
\def\XINT_trim_trimall#1{%
\def\XINT_trim_trimall {\expandafter#1\xint_bye}%
}\XINT_trim_trimall{ }%
```

  This branch doesn't pre-evaluate the length of the list argument. Redone again for 1.2j, manages to trim nine by nine. Some non optimal looking aspect of the code is for allowing sharing with \xintNthElt.

```
\long\def\XINT_trim_pos #1.#2%
{%
    \expandafter\XINT_trim_pos_done\expandafter\space
    \romannumeral0\expandafter\XINT_trim_loop\the\numexpr#1-\xint_c_ix.%
     #2\xint:\xint:\xint:\xint:\xint:%
       \xint:\xint:\xint:\xint:\xint:%
    \xint_bye
}%
```

```
\def\XINT_trim_loop #1#2.%
{%
    \xint_gob_til_minus#1\XINT_trim_finish-%
    \expandafter\XINT_trim_loop\the\numexpr#1#2\XINT_trim_loop_trimnine
}%
\long\def\XINT_trim_loop_trimnine #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint: #9\XINT_trim_toofew\xint:-\xint_c_ix.%
}%
\def\XINT_trim_toofew\xint:{*\xint_c_}%
\def\XINT_trim_finish#1{%
\def\XINT_trim_finish-%
    \expandafter\XINT_trim_loop\the\numexpr-##1\XINT_trim_loop_trimnine
{%
    \expandafter\expandafter\expandafter#1%
    \csname xint_gobble_\romannumeral\numexpr\xint_c_ix-##1\endcsname
}}\XINT_trim_finish{ }%
\long\def\XINT_trim_pos_done #1\xint:#2\xint_bye {#1}%
```

3.16 \xintTrimUnbraced

1.2a. Modified in 1.2i like \xintTrim

```
\def\xintTrimUnbraced         {\romannumeral0\xinttrimunbraced }%
\def\xintTrimUnbracedNoExpand {\romannumeral0\xinttrimunbracednoexpand }%
\long\def\xinttrimunbraced #1#2%
    {\expandafter\XINT_trimunbr_a\the\numexpr #1\expandafter.%
                          \expandafter{\romannumeral`&&@#2}}%
\def\xinttrimunbracednoexpand #1%
    {\expandafter\XINT_trimunbr_a\the\numexpr #1.}%
\def\XINT_trimunbr_a #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_trim_trimnone
        0#1\XINT_trimunbr_neg
         0-{\XINT_trim_pos #1}%
    \krof
}%
\long\def\XINT_trimunbr_neg #1.#2%
{%
    \expandafter\XINT_trimunbr_neg_a\the\numexpr
    #1-\numexpr\XINT_length_loop
    #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    .{}#2\xint_bye
}%
\def\XINT_trimunbr_neg_a #1%
{%
    \xint_UDsignfork
        #1{\expandafter\XINT_keepunbr_loop\the\numexpr-\xint_c_viii+}%
         -\XINT_trim_trimall
    \krof
```

```
}%
```

3.17 \xintApply

\xintApply {\macro}{{a}{b}...{z}} returns {\macro{a}}...{\macro{b}} where each instance of \macro is f-expanded. The list itself is first f-expanded and may thus be a macro. Introduced with release 1.04.

```
\def\xintApply          {\romannumeral0\xintapply }%
\def\xintApplyNoExpand {\romannumeral0\xintapplynoexpand }%
\long\def\xintapply #1#2%
{%
    \expandafter\XINT_apply\expandafter {\romannumeral`&&@#2}%
    {#1}%
}%
\long\def\XINT_apply #1#2{\XINT_apply_loop_a {}{#2}#1\xint_bye }%
\long\def\xintapplynoexpand #1#2{\XINT_apply_loop_a {}{#1}#2\xint_bye }%
\long\def\XINT_apply_loop_a #1#2#3%
{%
    \xint_bye #3\XINT_apply_end\xint_bye
    \expandafter
    \XINT_apply_loop_b
    \expandafter {\romannumeral`&&@#2{#3}}{#1}{#2}%
}%
\long\def\XINT_apply_loop_b #1#2{\XINT_apply_loop_a {#2{#1}}}%
\long\def\XINT_apply_end\xint_bye\expandafter\XINT_apply_loop_b
    \expandafter #1#2#3{ #2}%
```

3.18 \xintApply:x (WIP, commented-out)

Done for 1.4 (2020/01/27). For usage in the NumPy-like slicing routines. Well, actually, in the end I sticked with old-fashioned (quadratic cost) \xintApply for 1.4 2020/01/31 release. See comments there.

(Comments mainly from 2020/01/27, but on 2020/02/24 I comment out the code and add an alternative)

To expand in \expanded context, and does not need to do any expansion of its second argument.

This uses techniques I had developed for 1.2i/1.2j Keep, Trim, Length, LastItem like macros, and I should revamp venerable \xintApply probably too. But the latter f-expandability (if it does not have \expanded at disposal) complicates significantly matters as it has to store material and release at very end.

Here it is simpler and I am doing it quickly as I really want to release 1.4. The \xint: token should not be located in looped over items. I could use something more exotic like the null char with catcode 3...

```
  \long\def\xintApply:x #1#2%
  {%
      \XINT_apply:x_loop {#1}#2%
      {\xint:\XINT_apply:x_loop_enda}{\xint:\XINT_apply:x_loop_endb}%
      {\xint:\XINT_apply:x_loop_endc}{\xint:\XINT_apply:x_loop_endd}%
      {\xint:\XINT_apply:x_loop_ende}{\xint:\XINT_apply:x_loop_endf}%
      {\xint:\XINT_apply:x_loop_endg}{\xint:\XINT_apply:x_loop_endh}\xint_bye
  }%
  \long\def\XINT_apply:x_loop #1#2#3#4#5#6#7#8#9%
```

```
    {%
        \xint_gob_til_xint: #9\xint:
        {#1{#2}}{#1{#3}}{#1{#4}}{#1{#5}}{#1{#6}}{#1{#7}}{#1{#8}}{#1{#9}}%
        \XINT_apply:x_loop {#1}%
    }%
    \long\def\XINT_apply:x_loop_endh\xint: #1\xint_bye{}%
    \long\def\XINT_apply:x_loop_endg\xint: #1#2\xint_bye{{#1}}%
    \long\def\XINT_apply:x_loop_endf\xint: #1#2#3\xint_bye{{#1}{#2}}%
    \long\def\XINT_apply:x_loop_ende\xint: #1#2#3#4\xint_bye{{#1}{#2}{#3}}%
    \long\def\XINT_apply:x_loop_endd\xint: #1#2#3#4#5\xint_bye{{#1}{#2}{#3}{#4}}%
    \long\def\XINT_apply:x_loop_endc\xint: #1#2#3#4#5#6\xint_bye{{#1}{#2}{#3}{#4}{#5}}%
    \long\def\XINT_apply:x_loop_endb\xint: #1#2#3#4#5#6#7\xint_bye{{#1}{#2}{#3}{#4}{#5}{#6}}%
    \long\def\XINT_apply:x_loop_enda\xint: #1#2#3#4#5#6#7#8\xint_bye{{#1}{#2}{#3}{#4}{#5}{#6}{#7}}%
For small number of items gain with respect to \xintApply is little if any (might even be a loss).
  Picking one by one is possibly better for small number of items. Like this for example, the
natural simple minded thing:
  \long\def\xintApply:x #1#2%
    {%
        \XINT_apply:x_loop {#1}#2\xint_bye\xint_bye
    }%
    \long\def\XINT_apply:x_loop #1#2%
    {%
        \xint_bye #2\xint_bye {#1{#2}}%
        \XINT_apply:x_loop {#1}%
    }%
Some variant on 2020/02/24
    \long\def\xint_Bbye#1\xint_Bye{}%
    \long\def\xintApply:x #1#2%
    {%
        \XINT_apply:x_loop {#1}#2%
        {\xint_bye}{\xint_bye}{\xint_bye}{\xint_bye}%
        {\xint_bye}{\xint_bye}{\xint_bye}{\xint_bye}\xint_bye
    }%
    \long\def\XINT_apply:x_loop #1#2#3#4#5#6#7#8#9%
    {%
        \xint_Bye #2\xint_bye {#1{#2}}%
        \xint_Bye #3\xint_bye {#1{#3}}%
        \xint_Bye #4\xint_bye {#1{#4}}%
        \xint_Bye #5\xint_bye {#1{#5}}%
        \xint_Bye #6\xint_bye {#1{#6}}%
        \xint_Bye #7\xint_bye {#1{#7}}%
        \xint_Bye #8\xint_bye {#1{#8}}%
        \xint_Bye #9\xint_bye {#1{#9}}%
        \XINT_apply:x_loop {#1}%
    }%
```

3.19 \xintApplyUnbraced

\xintApplyUnbraced {\macro}{{a}{b}...{z}} returns \macro{a}...\macro{z} where each instance of \macro is f-expanded using \romannumeral-`0. The second argument may be a macro as it is itself also f-expanded. No braces are added: this allows for example a non-expandable \def in \macro, without having to do \gdef. Introduced with release 1.06b.

```
\def\xintApplyUnbraced {\romannumeral0\xintapplyunbraced }%
\def\xintApplyUnbracedNoExpand {\romannumeral0\xintapplyunbracednoexpand }%
\long\def\xintapplyunbraced #1#2%
{%
    \expandafter\XINT_applyunbr\expandafter {\romannumeral`&&@#2}%
    {#1}%
}%
\long\def\XINT_applyunbr #1#2{\XINT_applyunbr_loop_a {}{#2}#1\xint_bye }%
\long\def\xintapplyunbracednoexpand #1#2%
   {\XINT_applyunbr_loop_a {}{#1}#2\xint_bye }%
\long\def\XINT_applyunbr_loop_a #1#2#3%
{%
    \xint_bye #3\XINT_applyunbr_end\xint_bye
    \expandafter\XINT_applyunbr_loop_b
    \expandafter {\romannumeral`&&@#2{#3}}{#1}{#2}%
}%
\long\def\XINT_applyunbr_loop_b #1#2{\XINT_applyunbr_loop_a {#2#1}}%
\long\def\XINT_applyunbr_end\xint_bye\expandafter\XINT_applyunbr_loop_b
    \expandafter #1#2#3{ #2}%
```

3.20 \xintApplyUnbraced:x (WIP, commented-out)

```
Done for 1.4, 2020/01/27. For usage in the NumPy-like slicing routines.
  The items should not contain \xint: and the applied macro should not contain \empty.
  Finally, xintexpr.sty 1.4 code did not use this macro but the f-expandable one \xintApplyUn-
braced.
  For 1.4b I prefer leave the code commented out, and classify it as WIP.
   \long\def\xintApplyUnbraced:x #1#2%
   {%
       \XINT_applyunbraced:x_loop {#1}#2%
       {\xint:\XINT_applyunbraced:x_loop_enda}{\xint:\XINT_applyunbraced:x_loop_endb}%
       {\xint:\XINT_applyunbraced:x_loop_endc}{\xint:\XINT_applyunbraced:x_loop_endd}%
       {\xint:\XINT_applyunbraced:x_loop_ende}{\xint:\XINT_applyunbraced:x_loop_endf}%
       {\xint:\XINT_applyunbraced:x_loop_endg}{\xint:\XINT_applyunbraced:x_loop_endh}\xint_bye
   }%
   \long\def\XINT_applyunbraced:x_loop #1#2#3#4#5#6#7#8#9%
   {%
       \xint_gob_til_xint: #9\xint:
               #1{#2}%
         \empty#1{#3}%
         \empty#1{#4}%
         \empty#1{#5}%
         \empty#1{#6}%
         \empty#1{#7}%
         \empty#1{#8}%
         \empty#1{#9}%
       \XINT_applyunbraced:x_loop {#1}%
   }%
   \long\def\XINT_applyunbraced:x_loop_endh\xint: #1\xint_bye{}%
   \long\def\XINT_applyunbraced:x_loop_endg\xint: #1\empty#2\xint_bye{#1}%
   \long\def\XINT_applyunbraced:x_loop_endf\xint: #1\empty
                                                  #2\empty#3\xint_bye{#1#2}%
```

```
    \long\def\XINT_applyunbraced:x_loop_ende\xint: #1\empty
                                                   #2\empty
                                                   #3\empty#4\xint_bye{#1#2#3}%
    \long\def\XINT_applyunbraced:x_loop_endd\xint: #1\empty
                                                   #2\empty
                                                   #3\empty
                                                   #4\empty#5\xint_bye{#1#2#3#4}%
    \long\def\XINT_applyunbraced:x_loop_endc\xint: #1\empty
                                                   #2\empty
                                                   #3\empty
                                                   #4\empty
                                                   #5\empty#6\xint_bye{#1#2#3#4#5}%
    \long\def\XINT_applyunbraced:x_loop_endb\xint: #1\empty
                                                   #2\empty
                                                   #3\empty
                                                   #4\empty
                                                   #5\empty
                                                   #6\empty#7\xint_bye{#1#2#3#4#5#6}%
    \long\def\XINT_applyunbraced:x_loop_enda\xint: #1\empty
                                                   #2\empty
                                                   #3\empty
                                                   #4\empty
                                                   #5\empty
                                                   #6\empty
                                                   #7\empty#8\xint_bye{#1#2#3#4#5#6#7}%
```

3.21 \xintZip (WIP, not public)

```
1.4b. (2020/02/25)
  Support for zip(). Requires \expanded.
  The implementation here thus considers the argument is already completely expanded and is a
sequence of nut-ples. I will come back at later date for more generic macros.
  Consider even the name of the function zip() as WIP.
  As per what this does, it imitates the zip() function. See xint-manual.pdf.
  I use lame terminators. Will think again later on this. I have to be careful with the used ter-
minators, in particular with the NE context in mind.
  Generally speaking I will think another day about efficiency else I will never start this.
  OK, done. More compact than I initially thought. Various things should be commented upon here.
Well, actually not so compact in the end as I basically had to double the whole thing simply to avoid
the overhead of having to grab the final result delimited by some \xint_bye\xint_bye\xint_bye\xint_bye\empty
terminator. Now actually rather \xint_bye\xint_bye\xint_bye\xint_bye\xint:
```

```
\def\xintZip #1{\expanded\XINT_zip_A#1\xint_bye\xint_bye}%
\def\XINT_zip_A#1%
{%
    \xint_bye#1{\expandafter}\xint_bye
    \expanded{\unexpanded{\XINT_ziptwo_A
        #1\xint_bye\xint_bye\xint_bye\xint_bye\xint:}\expandafter}%
    \expanded\XINT_zip_a
}%
\def\XINT_zip_a#1%
{%
```

```
    \xint_bye#1\XINT_zip_terminator\xint_bye
    \expanded{\unexpanded{\XINT_ziptwo_a
        #1\xint_bye\xint_bye\xint_bye\xint_bye\xint:}\expandafter}%
    \expanded\XINT_zip_a
}%
\def\XINT_zip_terminator\xint_bye#1\xint_bye{{}\empty\empty\empty\empty\xint:}%
\def\XINT_ziptwo_a #1#2#3#4#5\xint:#6#7#8#9%
{%
    \bgroup
    \xint_bye #1\XINT_ziptwo_e \xint_bye
    \xint_bye #6\XINT_ziptwo_e \xint_bye {{#1}#6}%
    \xint_bye #2\XINT_ziptwo_e \xint_bye
    \xint_bye #7\XINT_ziptwo_e \xint_bye {{#2}#7}%
    \xint_bye #3\XINT_ziptwo_e \xint_bye
    \xint_bye #8\XINT_ziptwo_e \xint_bye {{#3}#8}%
    \xint_bye #4\XINT_ziptwo_e \xint_bye
    \xint_bye #9\XINT_ziptwo_e \xint_bye {{#4}#9}%
```

Attention here that #6 can very well deliver no tokens at all. But the \ifx will then do the expected thing. Only mentioning!

By the way, the \xint_bye method means TeX needs to look into tokens but skipping braced groups. A conditional based method lets TeX look only at the start but then it has to find \else or \fi so here also it must looks at tokens, and actually goes into braced groups. But (written 2020/02/26) I never did serious testing comparing the two, and in xint I have usually preferred \xint_bye/\xint_gob_til_foo types of methods (they proved superior than \ifnum to check for 0000 in numerical core context for example, at the early days when xint used blocks of 4 digits, not 8), or usage of \if/\ifx only on single tokens, combined with some \xint_dothis/\xint_orthat syntax.

```
    \ifx \empty#6\expandafter\XINT_zipone_a\fi
    \XINT_ziptwo_b #5\xint:
}%
\def\XINT_zipone_a\XINT_ziptwo_b{\XINT_zipone_b}%
\def\XINT_ziptwo_b #1#2#3#4#5\xint:#6#7#8#9%
{%
    \xint_bye #1\XINT_ziptwo_e \xint_bye
    \xint_bye #6\XINT_ziptwo_e \xint_bye {{#1}#6}%
    \xint_bye #2\XINT_ziptwo_e \xint_bye
    \xint_bye #7\XINT_ziptwo_e \xint_bye {{#2}#7}%
    \xint_bye #3\XINT_ziptwo_e \xint_bye
    \xint_bye #8\XINT_ziptwo_e \xint_bye {{#3}#8}%
    \xint_bye #4\XINT_ziptwo_e \xint_bye
    \xint_bye #9\XINT_ziptwo_e \xint_bye {{#4}#9}%
    \XINT_ziptwo_b #5\xint:
}%
\def\XINT_ziptwo_e #1\XINT_ziptwo_b #2\xint:#3\xint:
    {\iffalse{\fi}\xint_bye\xint_bye\xint_bye\xint_bye\xint:}%
\def\XINT_zipone_b #1#2#3#4%
{%
    \xint_bye #1\XINT_zipone_e \xint_bye {{#1}}%
    \xint_bye #2\XINT_zipone_e \xint_bye {{#2}}%
    \xint_bye #3\XINT_zipone_e \xint_bye {{#3}}%
    \xint_bye #4\XINT_zipone_e \xint_bye {{#4}}%
    \XINT_zipone_b
```

```
}%
\def\XINT_zipone_e #1\XINT_zipone_b #2\xint:
    {\iffalse{\fi}\xint_bye\xint_bye\xint_bye\xint_bye\empty}%
\def\XINT_ziptwo_A #1#2#3#4#5\xint:#6#7#8#9%
{%
    \bgroup
    \xint_bye #1\XINT_ziptwo_end \xint_bye
    \xint_bye #6\XINT_ziptwo_end \xint_bye {{#1}#6}%
    \xint_bye #2\XINT_ziptwo_end \xint_bye
    \xint_bye #7\XINT_ziptwo_end \xint_bye {{#2}#7}%
    \xint_bye #3\XINT_ziptwo_end \xint_bye
    \xint_bye #8\XINT_ziptwo_end \xint_bye {{#3}#8}%
    \xint_bye #4\XINT_ziptwo_end \xint_bye
    \xint_bye #9\XINT_ziptwo_end \xint_bye {{#4}#9}%
    \ifx \empty#6\expandafter\XINT_zipone_A\fi
    \XINT_ziptwo_B #5\xint:
}%
\def\XINT_zipone_A\XINT_ziptwo_B{\XINT_zipone_B}%
\def\XINT_ziptwo_B #1#2#3#4#5\xint:#6#7#8#9%
{%
    \xint_bye #1\XINT_ziptwo_end \xint_bye
    \xint_bye #6\XINT_ziptwo_end \xint_bye {{#1}#6}%
    \xint_bye #2\XINT_ziptwo_end \xint_bye
    \xint_bye #7\XINT_ziptwo_end \xint_bye {{#2}#7}%
    \xint_bye #3\XINT_ziptwo_end \xint_bye
    \xint_bye #8\XINT_ziptwo_end \xint_bye {{#3}#8}%
    \xint_bye #4\XINT_ziptwo_end \xint_bye
    \xint_bye #9\XINT_ziptwo_end \xint_bye {{#4}#9}%
    \XINT_ziptwo_B #5\xint:
}%
\def\XINT_ziptwo_end #1\XINT_ziptwo_B #2\xint:#3\xint:{\iffalse{\fi}}%
\def\XINT_zipone_B #1#2#3#4%
{%
    \xint_bye #1\XINT_zipone_end \xint_bye {{#1}}%
    \xint_bye #2\XINT_zipone_end \xint_bye {{#2}}%
    \xint_bye #3\XINT_zipone_end \xint_bye {{#3}}%
    \xint_bye #4\XINT_zipone_end \xint_bye {{#4}}%
    \XINT_zipone_B
}%
\def\XINT_zipone_end #1\XINT_zipone_B #2\xint:#3\xint:{\iffalse{\fi}}%
```

3.22 \xintSeq

1.09c. Without the optional argument puts stress on the input stack, should not be used to generated thousands of terms then.

```
\def\xintSeq {\romannumeral0\xintseq }%
\def\xintseq #1{\XINT_seq_chkopt  #1\xint_bye }%
\def\XINT_seq_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_seq_opt
       \else\expandafter\XINT_seq_noopt
    \fi  #1%
```

```
}%
\def\XINT_seq_noopt #1\xint_bye #2%
{%
    \expandafter\XINT_seq\expandafter
       {\the\numexpr#1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_seq #1#2%
{%
   \ifcase\ifnum #1=#2 0\else\ifnum #2>#1 1\else -1\fi\fi\space
      \expandafter\xint_stop_atfirstoftwo
   \or
      \expandafter\XINT_seq_p
   \else
      \expandafter\XINT_seq_n
   \fi
   {#2}{#1}%
}%
\def\XINT_seq_p #1#2%
{%
    \ifnum #1>#2
      \expandafter\expandafter\expandafter\XINT_seq_p
    \else
      \expandafter\XINT_seq_e
    \fi
    \expandafter{\the\numexpr #1-\xint_c_i}{#2}{#1}%
}%
\def\XINT_seq_n #1#2%
{%
    \ifnum #1<#2
      \expandafter\expandafter\expandafter\XINT_seq_n
    \else
      \expandafter\XINT_seq_e
    \fi
     \expandafter{\the\numexpr #1+\xint_c_i}{#2}{#1}%
}%
\def\XINT_seq_e #1#2#3{ }%
\def\XINT_seq_opt [\xint_bye #1]#2#3%
{%
    \expandafter\XINT_seqo\expandafter
    {\the\numexpr #2\expandafter}\expandafter
    {\the\numexpr #3\expandafter}\expandafter
    {\the\numexpr #1}%
}%
\def\XINT_seqo #1#2%
{%
   \ifcase\ifnum #1=#2 0\else\ifnum #2>#1 1\else -1\fi\fi\space
      \expandafter\XINT_seqo_a
   \or
      \expandafter\XINT_seqo_pa
   \else
      \expandafter\XINT_seqo_na
   \fi
```

```
    {#1}{#2}%
}%
\def\XINT_seqo_a #1#2#3{ {#1}}%
\def\XINT_seqo_o #1#2#3#4{ #4}%
\def\XINT_seqo_pa #1#2#3%
{%
    \ifcase\ifnum #3=\xint_c_ 0\else\ifnum #3>\xint_c_ 1\else -1\fi\fi\space
          \expandafter\XINT_seqo_o
    \or
          \expandafter\XINT_seqo_pb
    \else
          \xint_afterfi{\expandafter\space\xint_gobble_iv}%
    \fi
    {#1}{#2}{#3}{{#1}}%
}%
\def\XINT_seqo_pb #1#2#3%
{%
    \expandafter\XINT_seqo_pc\expandafter{\the\numexpr #1+#3}{#2}{#3}%
}%
\def\XINT_seqo_pc #1#2%
{%
    \ifnum #1>#2
        \expandafter\XINT_seqo_o
    \else
        \expandafter\XINT_seqo_pd
    \fi
    {#1}{#2}%
}%
\def\XINT_seqo_pd #1#2#3#4{\XINT_seqo_pb {#1}{#2}{#3}{#4{#1}}}%
\def\XINT_seqo_na #1#2#3%
{%
    \ifcase\ifnum #3=\xint_c_ 0\else\ifnum #3>\xint_c_ 1\else -1\fi\fi\space
        \expandafter\XINT_seqo_o
    \or
        \xint_afterfi{\expandafter\space\xint_gobble_iv}%
    \else
        \expandafter\XINT_seqo_nb
    \fi
    {#1}{#2}{#3}{{#1}}%
}%
\def\XINT_seqo_nb #1#2#3%
{%
    \expandafter\XINT_seqo_nc\expandafter{\the\numexpr #1+#3}{#2}{#3}%
}%
\def\XINT_seqo_nc #1#2%
{%
    \ifnum #1<#2
        \expandafter\XINT_seqo_o
    \else
        \expandafter\XINT_seqo_nd
    \fi
    {#1}{#2}%
```

```
}%
\def\XINT_seqo_nd #1#2#3#4{\XINT_seqo_nb {#1}{#2}{#3}{#4{#1}}}%
```

3.23 \xintloop, \xintbreakloop, \xintbreakloopanddo, \xintloopskiptonext

1.09g [2013/11/22]. Made long with 1.09h.

```
\long\def\xintloop #1#2\repeat {#1#2\xintloop_again\fi\xint_gobble_i {#1#2}}%
\long\def\xintloop_again\fi\xint_gobble_i #1{\fi
                              #1\xintloop_again\fi\xint_gobble_i {#1}}%
\long\def\xintbreakloop #1\xintloop_again\fi\xint_gobble_i #2{}%
\long\def\xintbreakloopanddo #1#2\xintloop_again\fi\xint_gobble_i #3{#1}%
\long\def\xintloopskiptonext #1\xintloop_again\fi\xint_gobble_i #2{%
                         #2\xintloop_again\fi\xint_gobble_i {#2}}%
```

3.24 \xintiloop, \xintiloopindex, \xintbracediloopindex, \xintouteriloopindex, \xintbracedouteriloopindex, \xintbreakiloop, \xintbreakiloopanddo, \xintiloopskiptonext, \xintiloopskipandredo

1.09g [2013/11/22]. Made long with 1.09h.
«braced» variants added (2018/04/24) for 1.3b.

```
\def\xintiloop [#1+#2]{%
    \expandafter\xintiloop_a\the\numexpr #1\expandafter.\the\numexpr #2.}%
\long\def\xintiloop_a #1.#2.#3#4\repeat{%
    #3#4\xintiloop_again\fi\xint_gobble_iii {#1}{#2}{#3#4}}%
\def\xintiloop_again\fi\xint_gobble_iii #1#2{%
    \fi\expandafter\xintiloop_again_b\the\numexpr#1+#2.#2.}%
\long\def\xintiloop_again_b #1.#2.#3{%
    #3\xintiloop_again\fi\xint_gobble_iii {#1}{#2}{#3}}%
\long\def\xintbreakiloop #1\xintiloop_again\fi\xint_gobble_iii #2#3#4{}%
\long\def\xintbreakiloopanddo
     #1.#2\xintiloop_again\fi\xint_gobble_iii #3#4#5{#1}%
\long\def\xintiloopindex #1\xintiloop_again\fi\xint_gobble_iii #2%
                 {#2#1\xintiloop_again\fi\xint_gobble_iii {#2}}%
\long\def\xintbracediloopindex #1\xintiloop_again\fi\xint_gobble_iii #2%
     {{#2}#1\xintiloop_again\fi\xint_gobble_iii {#2}}%
\long\def\xintouteriloopindex #1\xintiloop_again
                          #2\xintiloop_again\fi\xint_gobble_iii #3%
   {#3#1\xintiloop_again #2\xintiloop_again\fi\xint_gobble_iii {#3}}%
\long\def\xintbracedouteriloopindex #1\xintiloop_again
                          #2\xintiloop_again\fi\xint_gobble_iii #3%
   {{#3}#1\xintiloop_again #2\xintiloop_again\fi\xint_gobble_iii {#3}}%
\long\def\xintiloopskiptonext #1\xintiloop_again\fi\xint_gobble_iii #2#3{%
    \expandafter\xintiloop_again_b \the\numexpr#2+#3.#3.}%
\long\def\xintiloopskipandredo #1\xintiloop_again\fi\xint_gobble_iii #2#3#4{%
    #4\xintiloop_again\fi\xint_gobble_iii {#2}{#3}{#4}}%
```

3.25 \XINT_xflet

1.09e [2013/10/29]: we f-expand unbraced tokens and swallow arising space tokens until the dust settles.

```
\def\XINT_xflet #1%
{%
    \def\XINT_xflet_macro {#1}\XINT_xflet_zapsp
}%
\def\XINT_xflet_zapsp
{%
    \expandafter\futurelet\expandafter\XINT_token
    \expandafter\XINT_xflet_sp?\romannumeral`&&@%
}%
\def\XINT_xflet_sp?
{%
    \ifx\XINT_token\XINT_sptoken
         \expandafter\XINT_xflet_zapsp
    \else\expandafter\XINT_xflet_zapspB
    \fi
}%
\def\XINT_xflet_zapspB
{%
    \expandafter\futurelet\expandafter\XINT_tokenB
    \expandafter\XINT_xflet_spB?\romannumeral`&&@%
}%
\def\XINT_xflet_spB?
{%
    \ifx\XINT_tokenB\XINT_sptoken
         \expandafter\XINT_xflet_zapspB
    \else\expandafter\XINT_xflet_eq?
    \fi
}%
\def\XINT_xflet_eq?
{%
    \ifx\XINT_token\XINT_tokenB
         \expandafter\XINT_xflet_macro
    \else\expandafter\XINT_xflet_zapsp
    \fi
}%
```

3.26 \xintApplyInline

1.09a: \xintApplyInline\macro{{a}{b}...{z}} has the same effect as executing \macro{a} and then applying again \xintApplyInline to the shortened list {{b}...{z}} until nothing is left. This is a non-expandable command which will result in quicker code than using \xintApplyUnbraced. It f-expands its second (list) argument first, which may thus be encapsulated in a macro.
  Rewritten in 1.09c. Nota bene: uses catcode 3 Z as privated list terminator.

```
\catcode`Z 3
\long\def\xintApplyInline #1#2%
{%
  \long\expandafter\def\expandafter\XINT_inline_macro
  \expandafter ##\expandafter 1\expandafter {#1{##1}}%
  \XINT_xflet\XINT_inline_b #2Z% this Z has catcode 3
}%
\def\XINT_inline_b
{%
```

```
    \ifx\XINT_token Z\expandafter\xint_gobble_i
    \else\expandafter\XINT_inline_d\fi
}%
\long\def\XINT_inline_d #1%
{%
  \long\def\XINT_item{{#1}}\XINT_xflet\XINT_inline_e
}%
\def\XINT_inline_e
{%
    \ifx\XINT_token Z\expandafter\XINT_inline_w
    \else\expandafter\XINT_inline_f\fi
}%
\def\XINT_inline_f
{%
  \expandafter\XINT_inline_g\expandafter{\XINT_inline_macro {##1}}%
}%
\long\def\XINT_inline_g #1%
{%
   \expandafter\XINT_inline_macro\XINT_item
   \long\def\XINT_inline_macro ##1{#1}\XINT_inline_d
}%
\def\XINT_inline_w #1%
{%
   \expandafter\XINT_inline_macro\XINT_item
}%
```

3.27 \xintFor, \xintFor*, \xintBreakFor, \xintBreakForAndDo

1.09c [2013/10/09]: a new kind of loop which uses macro parameters #1, #2, #3, #4 rather than macros; while not expandable it survives executing code closing groups, like what happens in an alignment with the & character. When inserted in a macro for later use, the # character must be doubled.

The non-star variant works on a csv list, which it expands once, the star variant works on a token list, which it (repeatedly) f-expands.

1.09e adds \XINT_forever with \xintintegers, \xintdimensions, \xintrationals and \xintBreakFor, \xintBreakForAndDo, \xintifForFirst, \xintifForLast. On this occasion \xint_firstoftwo and \xint_secondoftwo are made long.

1.09f: rewrites large parts of \xintFor code in order to filter the comma separated list via \xintCSVtoList which gets rid of spaces. The #1 in \XINT_for_forever? has an initial space token which serves two purposes: preventing brace stripping, and stopping the expansion made by \xintcsvtolist. If the \XINT_forever branch is taken, the added space will not be a problem there.

1.09f rewrites (2013/11/03) the code which now allows all macro parameters from #1 to #9 in \xintFor, \xintFor*, and \XINT_forever. 1.2i: slightly more robust \xintifForFirst/Last in case of nesting.

```
\def\XINT_tmpa #1#2{\ifnum #2<#1 \xint_afterfi {{#########2}}\fi}%
\def\XINT_tmpb #1#2{\ifnum #1<#2 \xint_afterfi {{#########2}}\fi}%
\def\XINT_tmpc #1%
{%
    \expandafter\edef \csname XINT_for_left#1\endcsname
                {\xintApplyUnbraced {\XINT_tmpa #1}{123456789}}%
    \expandafter\edef \csname XINT_for_right#1\endcsname
                {\xintApplyUnbraced {\XINT_tmpb #1}{123456789}}%
```

```
}%
\xintApplyInline \XINT_tmpc {123456789}%
\long\def\xintBreakFor      #1Z{}%
\long\def\xintBreakForAndDo #1#2Z{#1}%
\def\xintFor {\let\xintifForFirst\xint_firstoftwo
              \let\xintifForLast\xint_secondoftwo
              \futurelet\XINT_token\XINT_for_ifstar }%
\def\XINT_for_ifstar {\ifx\XINT_token*\expandafter\XINT_forx
                                    \else\expandafter\XINT_for \fi }%
\catcode`U 3 % with numexpr
\catcode`V 3 % with xintfrac.sty (xint.sty not enough)
\catcode`D 3 % with dimexpr
\def\XINT_flet_zapsp
{%
    \futurelet\XINT_token\XINT_flet_sp?
}%
\def\XINT_flet_sp?
{%
    \ifx\XINT_token\XINT_sptoken
         \xint_afterfi{\expandafter\XINT_flet_zapsp\romannumeral0}%
    \else\expandafter\XINT_flet_macro
    \fi
}%
\long\def\XINT_for #1#2in#3#4#5%
{%
    \expandafter\XINT_toks\expandafter
        {\expandafter\XINT_for_d\the\numexpr #2\relax {#5}}%
    \def\XINT_flet_macro {\expandafter\XINT_for_forever?\space}%
    \expandafter\XINT_flet_zapsp #3Z%
}%
\def\XINT_for_forever? #1Z%
{%
    \ifx\XINT_token U\XINT_to_forever\fi
    \ifx\XINT_token V\XINT_to_forever\fi
    \ifx\XINT_token D\XINT_to_forever\fi
    \expandafter\the\expandafter\XINT_toks\romannumeral0\xintcsvtolist {#1}Z%
}%
\def\XINT_to_forever\fi #1\xintcsvtolist #2{\fi \XINT_forever #2}%
\long\def\XINT_forx *#1#2in#3#4#5%
{%
    \expandafter\XINT_toks\expandafter
       {\expandafter\XINT_forx_d\the\numexpr #2\relax {#5}}%
    \XINT_xflet\XINT_forx_forever? #3Z%
}%
\def\XINT_forx_forever?
{%
    \ifx\XINT_token U\XINT_to_forxever\fi
    \ifx\XINT_token V\XINT_to_forxever\fi
    \ifx\XINT_token D\XINT_to_forxever\fi
    \XINT_forx_empty?
}%
\def\XINT_to_forxever\fi #1\XINT_forx_empty? {\fi \XINT_forever }%
```

```
\catcode`U 11
\catcode`D 11
\catcode`V 11
\def\XINT_forx_empty?
{%
    \ifx\XINT_token Z\expandafter\xintBreakFor\fi
    \the\XINT_toks
}%
\long\def\XINT_for_d #1#2#3%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks {{#3}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
                     \the\XINT_toks   \csname XINT_for_right#1\endcsname }%
  \XINT_toks {\XINT_x\let\xintifForFirst\xint_secondoftwo
                     \let\xintifForLast\xint_secondoftwo\XINT_for_d #1{#2}}%
  \futurelet\XINT_token\XINT_for_last?
}%
\long\def\XINT_forx_d #1#2#3%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks {{#3}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
                     \the\XINT_toks   \csname XINT_for_right#1\endcsname }%
  \XINT_toks {\XINT_x\let\xintifForFirst\xint_secondoftwo
                     \let\xintifForLast\xint_secondoftwo\XINT_forx_d #1{#2}}%
  \XINT_xflet\XINT_for_last?
}%
\def\XINT_for_last?
{%
  \ifx\XINT_token Z\expandafter\XINT_for_last?yes\fi
  \the\XINT_toks
}%
\def\XINT_for_last?yes
{%
 \let\xintifForLast\xint_firstoftwo
 \xintBreakForAndDo{\XINT_x\xint_gobble_i Z}%
}%
```

3.28 \XINT_forever, \xintintegers, \xintdimensions, \xintrationals

New with 1.09e. But this used inadvertently \xintiadd/\xintimul which have the unnecessary \xintnum overhead. Changed in 1.09f to use \xintiiadd/\xintiimul which do not have this overhead. Also 1.09f uses \xintZapSpacesB for the \xintrationals case to get rid of leading and ending spaces in the #4 and #5 delimited parameters of \XINT_forever_opt_a (for \xintintegers and \xintdimensions this is not necessary, due to the use of \numexpr resp. \dimexpr in \XINT_?expr_Ua, resp.\XINT_?expr_Da).

```
\catcode`U 3
\catcode`D 3
\catcode`V 3
\let\xintegers      U%
\let\xintintegers   U%
```

```
\let\xintdimensions D%
\let\xintrationals  V%
\def\XINT_forever #1%
{%
  \expandafter\XINT_forever_a
  \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi a\expandafter\endcsname
  \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi i\expandafter\endcsname
  \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi \endcsname
}%
\catcode`U 11
\catcode`D 11
\catcode`V 11
\def\XINT_?expr_Ua #1#2%
   {\expandafter{\expandafter\numexpr\the\numexpr #1\expandafter\relax
                               \expandafter\relax\expandafter}%
    \expandafter{\the\numexpr #2}}%
\def\XINT_?expr_Da #1#2%
   {\expandafter{\expandafter\dimexpr\number\dimexpr #1\expandafter\relax
                  \expandafter s\expandafter p\expandafter\relax\expandafter}%
    \expandafter{\number\dimexpr #2}}%
\catcode`Z 11
\def\XINT_?expr_Va #1#2%
{%
    \expandafter\XINT_?expr_Vb\expandafter
          {\romannumeral`&&@\xintrawwithzeros{\xintZapSpacesB{#2}}}%
          {\romannumeral`&&@\xintrawwithzeros{\xintZapSpacesB{#1}}}%
}%
\catcode`Z 3
\def\XINT_?expr_Vb #1#2{\expandafter\XINT_?expr_Vc #2.#1.}%
\def\XINT_?expr_Vc #1/#2.#3/#4.%
{%
     \xintifEq {#2}{#4}%
       {\XINT_?expr_Vf {#3}{#1}{#2}}%
       {\expandafter\XINT_?expr_Vd\expandafter
        {\romannumeral0\xintiimul {#2}{#4}}%
        {\romannumeral0\xintiimul {#1}{#4}}%
        {\romannumeral0\xintiimul {#2}{#3}}%
       }%
}%
\def\XINT_?expr_Vd #1#2#3{\expandafter\XINT_?expr_Ve\expandafter {#2}{#3}{#1}}%
\def\XINT_?expr_Ve #1#2{\expandafter\XINT_?expr_Vf\expandafter {#2}{#1}}%
\def\XINT_?expr_Vf #1#2#3{{#2/#3}{{0}{#1}{#2}{#3}}}%
\def\XINT_?expr_Ui {{\numexpr 1\relax}{1}}%
\def\XINT_?expr_Di {{\dimexpr 0pt\relax}{65536}}%
\def\XINT_?expr_Vi {{1/1}{0111}}%
\def\XINT_?expr_U #1#2%
   {\expandafter{\expandafter\numexpr\the\numexpr #1+#2\relax\relax}{#2}}%
\def\XINT_?expr_D #1#2%
   {\expandafter{\expandafter\dimexpr\the\numexpr #1+#2\relax sp\relax}{#2}}%
\def\XINT_?expr_V #1#2{\XINT_?expr_Vx #2}%
\def\XINT_?expr_Vx #1#2%
{%
```

```
    \expandafter\XINT_?expr_Vy\expandafter
       {\romannumeral0\xintiiadd {#1}{#2}}{#2}%
}%
\def\XINT_?expr_Vy #1#2#3#4%
{%
    \expandafter{\romannumeral0\xintiiadd {#3}{#1}/#4}{{#1}{#2}{#3}{#4}}%
}%
\def\XINT_forever_a #1#2#3#4%
{%
   \ifx #4[\expandafter\XINT_forever_opt_a
      \else\expandafter\XINT_forever_b
   \fi #1#2#3#4%
}%
\def\XINT_forever_b #1#2#3Z{\expandafter\XINT_forever_c\the\XINT_toks #2#3}%
\long\def\XINT_forever_c #1#2#3#4#5%
   {\expandafter\XINT_forever_d\expandafter #2#4#5{#3}Z}%
\def\XINT_forever_opt_a #1#2#3[#4+#5]#6Z%
{%
   \expandafter\expandafter\expandafter
   \XINT_forever_opt_c\expandafter\the\expandafter\XINT_toks
   \romannumeral`&&@#1{#4}{#5}#3%
}%
\long\def\XINT_forever_opt_c #1#2#3#4#5#6{\XINT_forever_d #2{#4}{#5}#6{#3}Z}%
\long\def\XINT_forever_d #1#2#3#4#5%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#5}%
  \XINT_toks {{#2}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
                     \the\XINT_toks   \csname XINT_for_right#1\endcsname }%
  \XINT_x
  \let\xintifForFirst\xint_secondoftwo
  \let\xintifForLast\xint_secondoftwo
  \expandafter\XINT_forever_d\expandafter #1\romannumeral`&&@#4{#2}{#3}#4{#5}%
}%
```

3.29 \xintForpair, \xintForthree, \xintForfour

1.09c.
[2013/11/02] 1.09f \xintForpair delegate to \xintCSVtoList and its \xintZapSpacesB the handling of spaces. Does not share code with \xintFor anymore.
[2013/11/03] 1.09f: \xintForpair extended to accept #1#2, #2#3 etc... up to #8#9, \xintForthree, #1#2#3 up to #7#8#9, \xintForfour id.
1.2i: slightly more robust \xintifForFirst/Last in case of nesting.

```
\catcode`j 3
\long\def\xintForpair #1#2#3in#4#5#6%
{%
    \let\xintifForFirst\xint_firstoftwo
    \let\xintifForLast\xint_secondoftwo
    \XINT_toks  {\XINT_forpair_d #2{#6}}%
    \expandafter\the\expandafter\XINT_toks #4jZ%
}%
\long\def\XINT_forpair_d #1#2#3(#4)#5%
```

```
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
      \the\XINT_toks \csname XINT_for_right\the\numexpr#1+\xint_c_i\endcsname}%
  \ifx #5j\expandafter\XINT_for_last?yes\fi
  \XINT_x
  \let\xintifForFirst\xint_secondoftwo
  \let\xintifForLast\xint_secondoftwo
  \XINT_forpair_d #1{#2}%
}%
\long\def\xintForthree #1#2#3in#4#5#6%
{%
    \let\xintifForFirst\xint_firstoftwo
    \let\xintifForLast\xint_secondoftwo
    \XINT_toks  {\XINT_forthree_d #2{#6}}%
    \expandafter\the\expandafter\XINT_toks #4jZ%
}%
\long\def\XINT_forthree_d #1#2#3(#4)#5%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
      \the\XINT_toks \csname XINT_for_right\the\numexpr#1+\xint_c_ii\endcsname}%
  \ifx #5j\expandafter\XINT_for_last?yes\fi
  \XINT_x
  \let\xintifForFirst\xint_secondoftwo
  \let\xintifForLast\xint_secondoftwo
  \XINT_forthree_d #1{#2}%
}%
\long\def\xintForfour #1#2#3in#4#5#6%
{%
    \let\xintifForFirst\xint_firstoftwo
    \let\xintifForLast\xint_secondoftwo
    \XINT_toks  {\XINT_forfour_d #2{#6}}%
    \expandafter\the\expandafter\XINT_toks #4jZ%
}%
\long\def\XINT_forfour_d #1#2#3(#4)#5%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
     \the\XINT_toks \csname XINT_for_right\the\numexpr#1+\xint_c_iii\endcsname}%
  \ifx #5j\expandafter\XINT_for_last?yes\fi
  \XINT_x
  \let\xintifForFirst\xint_secondoftwo
  \let\xintifForLast\xint_secondoftwo
  \XINT_forfour_d #1{#2}%
}%
\catcode`Z 11
\catcode`j 11
```

3.30 \xintAssign, \xintAssignArray, \xintDigitsOf

\xintAssign {a}{b}..{z}\to\A\B...\Z resp. \xintAssignArray {a}{b}..{z}\to\U.
\xintDigitsOf=\xintAssignArray.
1.1c 2015/09/12 has (belatedly) corrected some "features" of \xintAssign which didn't like the case of a space right before the "\to", or the case with the first token not an opening brace and the subsequent material containing brace groups. The new code handles gracefully these situations.

```
\def\xintAssign{\def\XINT_flet_macro {\XINT_assign_fork}\XINT_flet_zapsp }%
\def\XINT_assign_fork
{%
    \let\XINT_assign_def\def
    \ifx\XINT_token[\expandafter\XINT_assign_opt
                \else\expandafter\XINT_assign_a
    \fi
}%
\def\XINT_assign_opt [#1]%
{%
    \ifcsname #1def\endcsname
      \expandafter\let\expandafter\XINT_assign_def \csname #1def\endcsname
    \else
      \expandafter\let\expandafter\XINT_assign_def \csname xint#1def\endcsname
    \fi
    \XINT_assign_a
}%
\long\def\XINT_assign_a #1\to
{%
    \def\XINT_flet_macro{\XINT_assign_b}%
    \expandafter\XINT_flet_zapsp\romannumeral`&&@#1\xint:\to
}%
\long\def\XINT_assign_b
{%
    \ifx\XINT_token\bgroup
         \expandafter\XINT_assign_c
    \else\expandafter\XINT_assign_f
    \fi
}%
\long\def\XINT_assign_f #1\xint:\to #2%
{%
    \XINT_assign_def #2{#1}%
}%
\long\def\XINT_assign_c #1%
{%
    \def\xint_temp {#1}%
    \ifx\xint_temp\xint_bracedstopper
        \expandafter\XINT_assign_e
    \else
        \expandafter\XINT_assign_d
    \fi
}%
\long\def\XINT_assign_d #1\to #2%
{%
    \expandafter\XINT_assign_def\expandafter #2\expandafter{\xint_temp}%
```

```
    \XINT_assign_c #1\to
}%
\def\XINT_assign_e #1\to {}%
\def\xintRelaxArray #1%
{%
    \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax}%
    \escapechar -1
    \expandafter\def\expandafter\xint_arrayname\expandafter {\string #1}%
    \XINT_restoreescapechar
    \xintiloop [\csname\xint_arrayname 0\endcsname+-1]
      \global
      \expandafter\let\csname\xint_arrayname\xintiloopindex\endcsname\relax
      \ifnum \xintiloopindex > \xint_c_
    \repeat
    \global\expandafter\let\csname\xint_arrayname 00\endcsname\relax
    \global\let #1\relax
}%
\def\xintAssignArray{\def\XINT_flet_macro {\XINT_assignarray_fork}%
                     \XINT_flet_zapsp }%
\def\XINT_assignarray_fork
{%
    \let\XINT_assignarray_def\def
    \ifx\XINT_token[\expandafter\XINT_assignarray_opt
                \else\expandafter\XINT_assignarray
    \fi
}%
\def\XINT_assignarray_opt [#1]%
{%
    \ifcsname #1def\endcsname
      \expandafter\let\expandafter\XINT_assignarray_def \csname #1def\endcsname
    \else
      \expandafter\let\expandafter\XINT_assignarray_def
                                  \csname xint#1def\endcsname
    \fi
    \XINT_assignarray
}%
\long\def\XINT_assignarray #1\to #2%
{%
    \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax }%
    \escapechar -1
    \expandafter\def\expandafter\xint_arrayname\expandafter {\string #2}%
    \XINT_restoreescapechar
    \def\xint_itemcount {0}%
    \expandafter\XINT_assignarray_loop \romannumeral`&&@#1\xint:
    \csname\xint_arrayname 00\expandafter\endcsname
    \csname\xint_arrayname 0\expandafter\endcsname
    \expandafter {\xint_arrayname}#2%
}%
\long\def\XINT_assignarray_loop #1%
{%
    \def\xint_temp {#1}%
    \ifx\xint_temp\xint_bracedstopper
```

```
1225        \expandafter\def\csname\xint_arrayname 0\expandafter\endcsname
1226                    \expandafter{\the\numexpr\xint_itemcount}%
1227        \expandafter\expandafter\expandafter\XINT_assignarray_end
1228     \else
1229        \expandafter\def\expandafter\xint_itemcount\expandafter
1230                    {\the\numexpr\xint_itemcount+\xint_c_i}%
1231        \expandafter\XINT_assignarray_def
1232           \csname\xint_arrayname\xint_itemcount\expandafter\endcsname
1233              \expandafter{\xint_temp }%
1234        \expandafter\XINT_assignarray_loop
1235     \fi
1236 }%
1237 \def\XINT_assignarray_end #1#2#3#4%
1238 {%
1239     \def #4##1%
1240     {%
1241         \romannumeral0\expandafter #1\expandafter{\the\numexpr ##1}%
1242     }%
1243     \def #1##1%
1244     {%
1245         \ifnum ##1<\xint_c_
1246             \xint_afterfi{\XINT_expandableerror{Array index negative: 0 > ##1} }%
1247         \else
1248         \xint_afterfi {%
1249               \ifnum ##1>#2
1250                   \xint_afterfi
1251                   {\XINT_expandableerror{Array index beyond range: ##1 > #2} }%
1252               \else\xint_afterfi
1253        {\expandafter\expandafter\expandafter\space\csname #3##1\endcsname}%
1254               \fi}%
1255         \fi
1256      }%
1257 }%
1258 \let\xintDigitsOf\xintAssignArray
```

3.31 CSV (non user documented) variants of Length, Keep, Trim, NthElt, Reverse

These routines are for use by \xintListSel:x:csv and \xintListSel:f:csv from xintexpr, and also for the reversed and len functions. Refactored for 1.2j release, following 1.2i updates to \xintKeep, \xintTrim, ...

These macros will remain undocumented in the user manual:

-- they exist primarily for internal use by the xintexpr parsers, hence don't have to be general purpose; for example, they a priori need to handle only catcode 12 tokens (not true in \xintNewExpr, though) hence they are not really worried about controlling brace stripping (nevertheless 1.2j has paid some secondary attention to it, see below.) They are not worried about normalizing leading spaces either, because none will be encountered when the macros are used as auxiliaries to the expression parsers.

-- crucial design elements may change in future:

1. whether the handled lists must have or not have a final comma. Currently, the model is the one of comma separated lists with **no** final comma. But this means that there can not be a distinction of principle between a truly empty list and a list which contains one item which turns out to be

empty. More importantly it makes the coding more complicated as it is needed to distinguish the empty list from the single-item list, both lacking commas.

For the internal use of xintexpr, it would be ok to require all list items to be terminated by a comma, and this would bring quite some simplications here, but as initially I started with non-terminated lists, I have left it this way in the 1.2j refactoring.

2. the way to represent the empty list. I was tempted for matter of optimization and synchronization with xintexpr context to require the empty list to be always represented by a space token and to not let the macros admit a completely empty input. But there were complications so for the time being 1.2j does accept truly empty output (it is not distinguished from an input equal to a space token) and produces empty output for empty list. This means that the status of the «nil» object for the xintexpr parsers is not completely clarified (currently it is represented by a space token).

The original Python slicing code in xintexpr 1.1 used \xintCSVtoList and \xintListWithSep{,} to convert back and forth to token lists and apply \xintKeep/\xintTrim. Release 1.2g switched to devoted f-expandable macros added to xinttools. Release 1.2j refactored all these macros as a follow-up to 1.2i improvements to \xintKeep/\xintTrim. They were made \long on this occasion and auxiliary \xintLengthUpTo:f:csv was added.

Leading spaces in items are currently maintained as is by the 1.2j macros, even by \xintNthEltPy:f:csv, with the exception of the first item, as the list is f-expanded. Perhaps \xintNthEltPy:f:csv should remove a leading space if present in the picked item; anyway, there are no spaces for the lists handled internally by the Python slicer of xintexpr, except the «nil» object currently represented by exactly one space.

Kept items (with no leading spaces; but first item special as it will have lost a leading space due to f-expansion) will lose a brace pair under \xintKeep:f:csv if the first argument was positive and strictly less than the length of the list. This differs of course from \xintKeep (which always braces items it outputs when used with positive first argument) and also from \xintKeepUnbraced in the case when the whole list is kept. Actually the case of singleton list is special, and brace removal will happen then.

This behaviour was otherwise for releases earlier than 1.2j and may change again.

Directly usable names are provided, but these macros (and the behaviour as described above) are to be considered *unstable* for the time being.

3.31.1 \xintLength:f:csv

1.2g. Redone for 1.2j. Contrarily to \xintLength from xintkernel.sty, this one expands its argument.

```
\def\xintLength:f:csv {\romannumeral0\xintlength:f:csv}%
\def\xintlength:f:csv #1%
{\long\def\xintlength:f:csv ##1{%
    \expandafter#1\the\numexpr\expandafter\XINT_length:f:csv_a
    \romannumeral`&&@##1\xint:,\xint:,\xint:,\xint:,%
      \xint:,\xint:,\xint:,\xint:,\xint:,%
      \xint_c_ix,\xint_c_viii,\xint_c_vii,\xint_c_vi,%
      \xint_c_v,\xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye
    \relax
}}\xintlength:f:csv { }%
```

Must first check if empty list.

```
\long\def\XINT_length:f:csv_a #1%
{%
    \xint_gob_til_xint: #1\xint_c_\xint_bye\xint:%
```

```
    \XINT_length:f:csv_loop #1%
}%
\long\def\XINT_length:f:csv_loop #1,#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    \xint_gob_til_xint: #9\XINT_length:f:csv_finish\xint:%
    \xint_c_ix+\XINT_length:f:csv_loop
}%
\def\XINT_length:f:csv_finish\xint:\xint_c_ix+\XINT_length:f:csv_loop
    #1,#2,#3,#4,#5,#6,#7,#8,#9,{#9\xint_bye}%
```

3.31.2 \xintLengthUpTo:f:csv

1.2j. \xintLengthUpTo:f:csv{N}{comma-list}. No ending comma. Returns -0 if length>N, else returns difference N-length. **N must be non-negative!!**

Attention to the dot after \xint_bye for the loop interface.

```
\def\xintLengthUpTo:f:csv {\romannumeral0\xintlengthupto:f:csv}%
\long\def\xintlengthupto:f:csv #1#2%
{%
    \expandafter\XINT_lengthupto:f:csv_a
    \the\numexpr#1\expandafter.%
    \romannumeral`&&@#2\xint:,\xint:,\xint:,\xint:,%
         \xint:,\xint:,\xint:,\xint:,%
         \xint_c_viii,\xint_c_vii,\xint_c_vi,\xint_c_v,%
         \xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye.%
}%
```

Must first recognize if empty list. If this is the case, return N.

```
\long\def\XINT_lengthupto:f:csv_a #1.#2%
{%
    \xint_gob_til_xint: #2\XINT_lengthupto:f:csv_empty\xint:%
    \XINT_lengthupto:f:csv_loop_b #1.#2%
}%
\def\XINT_lengthupto:f:csv_empty\xint:%
    \XINT_lengthupto:f:csv_loop_b #1.#2\xint_bye.{ #1}%
\def\XINT_lengthupto:f:csv_loop_a #1%
{%
    \xint_UDsignfork
      #1\XINT_lengthupto:f:csv_gt
       -\XINT_lengthupto:f:csv_loop_b
    \krof #1%
}%
\long\def\XINT_lengthupto:f:csv_gt #1\xint_bye.{-0}%
\long\def\XINT_lengthupto:f:csv_loop_b #1.#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    \xint_gob_til_xint: #9\XINT_lengthupto:f:csv_finish_a\xint:%
    \expandafter\XINT_lengthupto:f:csv_loop_a\the\numexpr #1-\xint_c_viii.%
}%
\def\XINT_lengthupto:f:csv_finish_a\xint:
    \expandafter\XINT_lengthupto:f:csv_loop_a
    \the\numexpr #1-\xint_c_viii.#2,#3,#4,#5,#6,#7,#8,#9,%
{%
```

```
    \expandafter\XINT_lengthupto:f:csv_finish_b\the\numexpr #1-#9\xint_bye
}%
\def\XINT_lengthupto:f:csv_finish_b #1#2.%
{%
    \xint_UDsignfork
       #1{-0}%
        -{ #1#2}%
    \krof
}%
```

3.31.3 \xintKeep:f:csv

1.2g 2016/03/17. Redone for 1.2j with use of \xintLengthUpTo:f:csv. Same code skeleton as \xintKeep but handling comma separated but non terminated lists has complications. The \xintKeep in case of a negative #1 uses \xintgobble, we don't have that for comma delimited items, hence we do a special loop here (this style of loop is surely competitive with xintgobble for a few dozens items and even more). The loop knows before starting that it will not go too far.

```
\def\xintKeep:f:csv {\romannumeral0\xintkeep:f:csv }%
\long\def\xintkeep:f:csv #1#2%
{%
    \expandafter\xint_stop_aftergobble
    \romannumeral0\expandafter\XINT_keep:f:csv_a
    \the\numexpr #1\expandafter.\expandafter{\romannumeral`&&@#2}%
}%
\def\XINT_keep:f:csv_a #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_keep:f:csv_keepnone
        0#1\XINT_keep:f:csv_neg
         0-{\XINT_keep:f:csv_pos #1}%
    \krof
}%
\long\def\XINT_keep:f:csv_keepnone .#1{,}%
\long\def\XINT_keep:f:csv_neg #1.#2%
{%
    \expandafter\XINT_keep:f:csv_neg_done\expandafter,%
    \romannumeral0%
    \expandafter\XINT_keep:f:csv_neg_a\the\numexpr
    #1-\numexpr\XINT_length:f:csv_a
    #2\xint:,\xint:,\xint:,\xint:,%
      \xint:,\xint:,\xint:,\xint:,\xint:,%
      \xint_c_ix,\xint_c_viii,\xint_c_vii,\xint_c_vi,%
      \xint_c_v,\xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye
    .#2\xint_bye
}%
\def\XINT_keep:f:csv_neg_a #1%
{%
    \xint_UDsignfork
        #1{\expandafter\XINT_keep:f:csv_trimloop\the\numexpr-\xint_c_ix+}%
         -\XINT_keep:f:csv_keepall
    \krof
}%
```

```
\def\XINT_keep:f:csv_keepall #1.{ }%
\long\def\XINT_keep:f:csv_neg_done #1\xint_bye{#1}%
\def\XINT_keep:f:csv_trimloop #1#2.%
{%
    \xint_gob_til_minus#1\XINT_keep:f:csv_trimloop_finish-%
    \expandafter\XINT_keep:f:csv_trimloop
    \the\numexpr#1#2-\xint_c_ix\expandafter.\XINT_keep:f:csv_trimloop_trimnine
}%
\long\def\XINT_keep:f:csv_trimloop_trimnine #1,#2,#3,#4,#5,#6,#7,#8,#9,{}%
\def\XINT_keep:f:csv_trimloop_finish-%
    \expandafter\XINT_keep:f:csv_trimloop
    \the\numexpr-#1-\xint_c_ix\expandafter.\XINT_keep:f:csv_trimloop_trimnine
    {\csname XINT_trim:f:csv_finish#1\endcsname}%
\long\def\XINT_keep:f:csv_pos #1.#2%
{%
    \expandafter\XINT_keep:f:csv_pos_fork
    \romannumeral0\XINT_lengthupto:f:csv_a
    #1.#2\xint:,\xint:,\xint:,\xint:,%
         \xint:,\xint:,\xint:,\xint:,%
         \xint_c_viii,\xint_c_vii,\xint_c_vi,\xint_c_v,%
         \xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye.%
    .#1.{}#2\xint_bye%
}%
\def\XINT_keep:f:csv_pos_fork #1#2.%
{%
    \xint_UDsignfork
      #1{\expandafter\XINT_keep:f:csv_loop\the\numexpr-\xint_c_viii+}%
       -\XINT_keep:f:csv_pos_keepall
    \krof
}%
\long\def\XINT_keep:f:csv_pos_keepall #1.#2#3\xint_bye{,#3}%
\def\XINT_keep:f:csv_loop #1#2.%
{%
    \xint_gob_til_minus#1\XINT_keep:f:csv_loop_end-%
    \expandafter\XINT_keep:f:csv_loop
    \the\numexpr#1#2-\xint_c_viii\expandafter.\XINT_keep:f:csv_loop_pickeight
}%
\long\def\XINT_keep:f:csv_loop_pickeight
     #1#2,#3,#4,#5,#6,#7,#8,#9,{{#1,#2,#3,#4,#5,#6,#7,#8,#9}}%
\def\XINT_keep:f:csv_loop_end-\expandafter\XINT_keep:f:csv_loop
    \the\numexpr-#1-\xint_c_viii\expandafter.\XINT_keep:f:csv_loop_pickeight
    {\csname XINT_keep:f:csv_end#1\endcsname}%
\long\expandafter\def\csname XINT_keep:f:csv_end1\endcsname
   #1#2,#3,#4,#5,#6,#7,#8,#9\xint_bye {#1,#2,#3,#4,#5,#6,#7,#8}%
\long\expandafter\def\csname XINT_keep:f:csv_end2\endcsname
   #1#2,#3,#4,#5,#6,#7,#8\xint_bye {#1,#2,#3,#4,#5,#6,#7}%
\long\expandafter\def\csname XINT_keep:f:csv_end3\endcsname
   #1#2,#3,#4,#5,#6,#7\xint_bye {#1,#2,#3,#4,#5,#6}%
\long\expandafter\def\csname XINT_keep:f:csv_end4\endcsname
   #1#2,#3,#4,#5,#6\xint_bye {#1,#2,#3,#4,#5}%
\long\expandafter\def\csname XINT_keep:f:csv_end5\endcsname
   #1#2,#3,#4,#5\xint_bye {#1,#2,#3,#4}%
```

```
\long\expandafter\def\csname XINT_keep:f:csv_end6\endcsname
   #1#2,#3,#4\xint_bye {#1,#2,#3}%
\long\expandafter\def\csname XINT_keep:f:csv_end7\endcsname
   #1#2,#3\xint_bye {#1,#2}%
\long\expandafter\def\csname XINT_keep:f:csv_end8\endcsname
   #1#2\xint_bye {#1}%
```

3.31.4 \xintTrim:f:csv

1.2g 2016/03/17. Redone for 1.2j 2016/12/20 on the basis of new \xintTrim.

```
\def\xintTrim:f:csv {\romannumeral0\xinttrim:f:csv }%
\long\def\xinttrim:f:csv #1#2%
{%
    \expandafter\xint_stop_aftergobble
    \romannumeral0\expandafter\XINT_trim:f:csv_a
    \the\numexpr #1\expandafter.\expandafter{\romannumeral`&&@#2}%
}%
\def\XINT_trim:f:csv_a #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_trim:f:csv_trimnone
        0#1\XINT_trim:f:csv_neg
         0-{\XINT_trim:f:csv_pos #1}%
    \krof
}%
\long\def\XINT_trim:f:csv_trimnone .#1{,#1}%
\long\def\XINT_trim:f:csv_neg #1.#2%
{%
    \expandafter\XINT_trim:f:csv_neg_a\the\numexpr
    #1-\numexpr\XINT_length:f:csv_a
    #2\xint:,\xint:,\xint:,\xint:,%
      \xint:,\xint:,\xint:,\xint:,\xint:,%
      \xint_c_ix,\xint_c_viii,\xint_c_vii,\xint_c_vi,%
      \xint_c_v,\xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye
    .{}#2\xint_bye
}%
\def\XINT_trim:f:csv_neg_a #1%
{%
    \xint_UDsignfork
        #1{\expandafter\XINT_keep:f:csv_loop\the\numexpr-\xint_c_viii+}%
         -\XINT_trim:f:csv_trimall
    \krof
}%
\def\XINT_trim:f:csv_trimall {\expandafter,\xint_bye}%
\long\def\XINT_trim:f:csv_pos #1.#2%
{%
    \expandafter\XINT_trim:f:csv_pos_done\expandafter,%
    \romannumeral0%
    \expandafter\XINT_trim:f:csv_loop\the\numexpr#1-\xint_c_ix.%
     #2\xint:,\xint:,\xint:,\xint:,\xint:,%
       \xint:,\xint:,\xint:,\xint:,\xint:\xint_bye
}%
```

```
\def\XINT_trim:f:csv_loop #1#2.%
{%
    \xint_gob_til_minus#1\XINT_trim:f:csv_finish-%
    \expandafter\XINT_trim:f:csv_loop\the\numexpr#1#2\XINT_trim:f:csv_loop_trimnine
}%
\long\def\XINT_trim:f:csv_loop_trimnine #1,#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    \xint_gob_til_xint: #9\XINT_trim:f:csv_toofew\xint:-\xint_c_ix.%
}%
\def\XINT_trim:f:csv_toofew\xint:{*\xint_c_}%
\def\XINT_trim:f:csv_finish-%
    \expandafter\XINT_trim:f:csv_loop\the\numexpr-#1\XINT_trim:f:csv_loop_trimnine
{%
    \csname XINT_trim:f:csv_finish#1\endcsname
}%
\long\expandafter\def\csname XINT_trim:f:csv_finish1\endcsname
  #1,#2,#3,#4,#5,#6,#7,#8,{ }%
\long\expandafter\def\csname XINT_trim:f:csv_finish2\endcsname
  #1,#2,#3,#4,#5,#6,#7,{ }%
\long\expandafter\def\csname XINT_trim:f:csv_finish3\endcsname
  #1,#2,#3,#4,#5,#6,{ }%
\long\expandafter\def\csname XINT_trim:f:csv_finish4\endcsname
  #1,#2,#3,#4,#5,{ }%
\long\expandafter\def\csname XINT_trim:f:csv_finish5\endcsname
  #1,#2,#3,#4,{ }%
\long\expandafter\def\csname XINT_trim:f:csv_finish6\endcsname
  #1,#2,#3,{ }%
\long\expandafter\def\csname XINT_trim:f:csv_finish7\endcsname
  #1,#2,{ }%
\long\expandafter\def\csname XINT_trim:f:csv_finish8\endcsname
  #1,{ }%
\expandafter\let\csname XINT_trim:f:csv_finish9\endcsname\space
\long\def\XINT_trim:f:csv_pos_done #1\xint:#2\xint_bye{#1}%
```

3.31.5 \xintNthEltPy:f:csv

Counts like Python starting at zero. Last refactored with 1.2j. Attention, makes currently no effort at removing leading spaces in the picked item.

```
\def\xintNthEltPy:f:csv {\romannumeral0\xintntheltpy:f:csv }%
\long\def\xintntheltpy:f:csv #1#2%
{%
   \expandafter\XINT_nthelt:f:csv_a
   \the\numexpr #1\expandafter.\expandafter{\romannumeral`&&@#2}%
}%
\def\XINT_nthelt:f:csv_a #1%
{%
    \xint_UDsignfork
        #1\XINT_nthelt:f:csv_neg
         -\XINT_nthelt:f:csv_pos
    \krof #1%
}%
\long\def\XINT_nthelt:f:csv_neg -#1.#2%
```

```
{%
    \expandafter\XINT_nthelt:f:csv_neg_fork
    \the\numexpr\XINT_length:f:csv_a
    #2\xint:,\xint:,\xint:,\xint:,%
      \xint:,\xint:,\xint:,\xint:,\xint:,%
      \xint_c_ix,\xint_c_viii,\xint_c_vii,\xint_c_vi,%
      \xint_c_v,\xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye
    -#1.#2,\xint_bye
}%
\def\XINT_nthelt:f:csv_neg_fork #1%
{%
    \if#1-\expandafter\xint_stop_afterbye\fi
    \expandafter\XINT_nthelt:f:csv_neg_done
    \romannumeral0%
    \expandafter\XINT_keep:f:csv_trimloop\the\numexpr-\xint_c_ix+#1%
}%
\long\def\XINT_nthelt:f:csv_neg_done#1,#2\xint_bye{ #1}%
\long\def\XINT_nthelt:f:csv_pos #1.#2%
{%
    \expandafter\XINT_nthelt:f:csv_pos_done
    \romannumeral0%
    \expandafter\XINT_trim:f:csv_loop\the\numexpr#1-\xint_c_ix.%
    #2\xint:,\xint:,\xint:,\xint:,\xint:,%
       \xint:,\xint:,\xint:,\xint:,\xint:,\xint_bye
}%
\def\XINT_nthelt:f:csv_pos_done #1{%
\long\def\XINT_nthelt:f:csv_pos_done ##1,##2\xint_bye{%
  \xint_gob_til_xint:##1\XINT_nthelt:f:csv_pos_cleanup\xint:#1##1}%
}\XINT_nthelt:f:csv_pos_done{ }%
```

This strange thing is in case the picked item was the last one, hence there was an ending \xint: (we could not put a comma earlier for matters of not confusing empty list with a singleton list), and we do this here to activate brace-stripping of item as all other items may be brace-stripped if picked. This is done for coherence. Of course, in the context of the xintexpr.sty parsers, there are no braces in list items...

```
\xint_firstofone{\long\def\XINT_nthelt:f:csv_pos_cleanup\xint:} %
   #1\xint:{ #1}%
```

3.31.6 \xintReverse:f:csv

1.2g. Contrarily to \xintReverseOrder from xintkernel.sty, this one expands its argument. Handles empty list too. 2016/03/17. Made \long for 1.2j.

```
\def\xintReverse:f:csv {\romannumeral0\xintreverse:f:csv }%
\long\def\xintreverse:f:csv #1%
{%
    \expandafter\XINT_reverse:f:csv_loop
    \expandafter{\expandafter}\romannumeral`&&@#1,%
      \xint:,%
        \xint_bye,\xint_bye,\xint_bye,\xint_bye,%
        \xint_bye,\xint_bye,\xint_bye,\xint_bye,%
      \xint:
```

```
}%
\long\def\XINT_reverse:f:csv_loop #1#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    \xint_bye #9\XINT_reverse:f:csv_cleanup\xint_bye
    \XINT_reverse:f:csv_loop {,#9,#8,#7,#6,#5,#4,#3,#2#1}%
}%
\long\def\XINT_reverse:f:csv_cleanup\xint_bye\XINT_reverse:f:csv_loop #1#2\xint:
{%
    \XINT_reverse:f:csv_finish #1%
}%
\long\def\XINT_reverse:f:csv_finish #1\xint:,{ }%
```

3.31.7 \xintFirstItem:f:csv

Added with 1.2k for use by first() in \xintexpr-essions, and some amount of compatibility with \xintNewExpr.

```
\def\xintFirstItem:f:csv {\romannumeral0\xintfirstitem:f:csv}%
\long\def\xintfirstitem:f:csv #1%
{%
    \expandafter\XINT_first:f:csv_a\romannumeral`&&@#1,\xint_bye
}%
\long\def\XINT_first:f:csv_a #1,#2\xint_bye{ #1}%
```

3.31.8 \xintLastItem:f:csv

Added with 1.2k, based on and sharing code with xintkernel's \xintLastItem from 1.2i. Output empty if input empty. f-expands its argument (hence first item, if not protected.) For use by last() in \xintexpr-essions with to some extent \xintNewExpr compatibility.

```
\def\xintLastItem:f:csv {\romannumeral0\xintlastitem:f:csv}%
\long\def\xintlastitem:f:csv #1%
{%
    \expandafter\XINT_last:f:csv_loop\expandafter{\expandafter}\expandafter.%
    \romannumeral`&&@#1,%
    \xint:\XINT_last_loop_enda,\xint:\XINT_last_loop_endb,%
    \xint:\XINT_last_loop_endc,\xint:\XINT_last_loop_endd,%
    \xint:\XINT_last_loop_ende,\xint:\XINT_last_loop_endf,%
    \xint:\XINT_last_loop_endg,\xint:\XINT_last_loop_endh,\xint_bye
}%
\long\def\XINT_last:f:csv_loop #1.#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    \xint_gob_til_xint: #9%
        {#8}{#7}{#6}{#5}{#4}{#3}{#2}{#1}\xint:
    \XINT_last:f:csv_loop {#9}.%
}%
```

3.31.9 \xintKeep:x:csv

Added to xintexpr at 1.2j.
  But data model changed at 1.4, this macro moved to xinttools, not part of publicly supported macros, may be removed at any time.
  This macro is used only with positive first argument.

```
\def\xintKeep:x:csv #1#2%
{%
    \expandafter\xint_gobble_i
    \romannumeral0\expandafter\XINT_keep:x:csv_pos
    \the\numexpr #1\expandafter.\expandafter{\romannumeral`&&@#2}%
}%
\def\XINT_keep:x:csv_pos #1.#2%
{%
    \expandafter\XINT_keep:x:csv_loop\the\numexpr#1-\xint_c_viii.%
    #2\xint_Bye,\xint_Bye,\xint_Bye,\xint_Bye,%
       \xint_Bye,\xint_Bye,\xint_Bye,\xint_Bye,\xint_bye
}%
\def\XINT_keep:x:csv_loop #1%
{%
    \xint_gob_til_minus#1\XINT_keep:x:csv_finish-%
    \XINT_keep:x:csv_loop_pickeight #1%
}%
\def\XINT_keep:x:csv_loop_pickeight #1.#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    ,#2,#3,#4,#5,#6,#7,#8,#9%
    \expandafter\XINT_keep:x:csv_loop\the\numexpr#1-\xint_c_viii.%
}%
\def\XINT_keep:x:csv_finish-\XINT_keep:x:csv_loop_pickeight -#1.%
{%
    \csname XINT_keep:x:csv_finish#1\endcsname
}%
\expandafter\def\csname XINT_keep:x:csv_finish1\endcsname
  #1,#2,#3,#4,#5,#6,#7,{,#1,#2,#3,#4,#5,#6,#7\xint_Bye}%
\expandafter\def\csname XINT_keep:x:csv_finish2\endcsname
  #1,#2,#3,#4,#5,#6,{,#1,#2,#3,#4,#5,#6\xint_Bye}%
\expandafter\def\csname XINT_keep:x:csv_finish3\endcsname
  #1,#2,#3,#4,#5,{,#1,#2,#3,#4,#5\xint_Bye}%
\expandafter\def\csname XINT_keep:x:csv_finish4\endcsname
  #1,#2,#3,#4,{,#1,#2,#3,#4\xint_Bye}%
\expandafter\def\csname XINT_keep:x:csv_finish5\endcsname
  #1,#2,#3,{,#1,#2,#3\xint_Bye}%
\expandafter\def\csname XINT_keep:x:csv_finish6\endcsname
  #1,#2,{,#1,#2\xint_Bye}%
\expandafter\def\csname XINT_keep:x:csv_finish7\endcsname
  #1,{,#1\xint_Bye}%
\expandafter\let\csname XINT_keep:x:csv_finish8\endcsname\xint_Bye
```

3.31.10 Public names for the undocumented csv macros: \xintCSVLength, \xintCSVKeep, \xintCSVKeepx, \xintCSVTrim, \xintCSVNthEltPy, \xintCSVReverse, \xintCSVFirstItem, \xintCSVLastItem

Completely unstable macros: currently they expand the list argument and want no final comma. But for matters of xintexpr.sty I could as well decide to require a final comma, and then I could simplify implementation but of course this would break the macros if used with current functionalities.

```
\let\xintCSVLength   \xintLength:f:csv
\let\xintCSVKeep     \xintKeep:f:csv
```

```
\let\xintCSVKeepx    \xintKeep:x:csv
\let\xintCSVTrim     \xintTrim:f:csv
\let\xintCSVNthEltPy \xintNthEltPy:f:csv
\let\xintCSVReverse  \xintReverse:f:csv
\let\xintCSVFirstItem\xintFirstItem:f:csv
\let\xintCSVLastItem \xintLastItem:f:csv
\let\XINT_tmpa\relax \let\XINT_tmpb\relax \let\XINT_tmpc\relax
\XINT_restorecatcodes_endinput%
```

4 Package xintcore implementation

Got split off from xint with release 1.1.

The core arithmetic routines have been entirely rewritten for release 1.2. The 1.2i and 1.2l brought again some improvements.

The commenting continues (2021/03/29) to be very sparse: actually it got worse than ever with release 1.2. I will possibly add comments at a later date, but for the time being the new routines are not commented at all.

1.3 removes all macros which were deprecated at 1.2o.

4.1 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from Heiko Oberdiek's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
```

```
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintcore.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintkernel.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintcore}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintcore.sty
      \ifx\w\relax % but xintkernel.sty not yet loaded.
         \def\z{\endgroup\input xintkernel.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xintkernel.sty not yet loaded.
            \def\z{\endgroup\RequirePackage{xintkernel}}%
          \fi
      \else
        \aftergroup\endinput % xintkernel already loaded.
      \fi
    \fi
  \fi
\z%
\XINTsetupcatcodes% defined in xintkernel.sty
```

4.2 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintcore}%
  [2021/03/29 v1.4d Expandable arithmetic on big integers (JFB)]%
```

4.3 (WIP!) Error conditions and exceptions

```
As per the Mike Cowlishaw/IBM's General Decimal Arithmetic Specification
  http://speleotrove.com/decimal/decarith.html
  and the Python3 implementation in its Decimal module.
  Clamped, ConversionSyntax, DivisionByZero, DivisionImpossible, DivisionUndefined, Inexact,
InsufficientStorage, InvalidContext, InvalidOperation, Overflow, Inexact, Rounded, Subnormal,
Underflow.
  X3.274 rajoute LostDigits
  Python rajoute FloatOperation (et n'inclut pas InsufficientStorage)
  quote de decarith.pdf: The Clamped, Inexact, Rounded, and Subnormal conditions can coincide
with each other or with other conditions. In these cases then any trap enabled for another condi-
tion takes precedence over (is handled before) all of these, any Subnormal trap takes precedence
over Inexact, any Inexact trap takes precedence over Rounded, and any Rounded trap takes prece-
dence over Clamped.
```

WORK IN PROGRESS ! (1.2l, 2017/07/26)

I follow the Python terminology: a trapped signal means it raises an exception which for us means an expandable error message with some possible user interaction. In this WIP state, the interaction is commented out. A non-trapped signal or condition would activate a (presumably silent) handler.

Here, no signal-raising condition is "ignored" and all are "trapped" which means that error handlers are never activated, thus left in garbage state in the code.

Various conditions can raise the same signal.

Only signals, not conditions, raise Flags.

If a signal is ignored it does not raise a Flag, but it activates the signal handler (by default now no signal is ignored.)

If a signal is not ignored it raises a Flag and then if it is not trapped it activates the handler of the _condition_.

If trapped (which is default now) an «exception» is raised, which means an expandable error message (I copied over the LaTeX3 code for expandable error messages, basically) interrupts the TeX run. In future, user input could be solicited, but currently this is commented out.

For now macros to reset flags are done but without public interface nor documentation.

Only four conditions are currently possibly encountered:

- InvalidOperation
- DivisionByZero
- DivisionUndefined (which signals InvalidOperation)
- Underflow

I did it quickly, anyhow this will become more palpable when some of the Decimal Specification is actually implemented. The plan is to first do the X3.274 norm, then more complete implementation will follow... perhaps...

```
\csname XINT_Clamped_istrapped\endcsname
\csname XINT_ConversionSyntax_istrapped\endcsname
\csname XINT_DivisionByZero_istrapped\endcsname
\csname XINT_DivisionImpossible_istrapped\endcsname
\csname XINT_DivisionUndefined_istrapped\endcsname
\csname XINT_InvalidOperation_istrapped\endcsname
\csname XINT_Overflow_istrapped\endcsname
\csname XINT_Underflow_istrapped\endcsname
\catcode`- 11
\def\XINT_ConversionSyntax-signal  {{InvalidOperation}}%
\let\XINT_DivisionImpossible-signal\XINT_ConversionSyntax-signal
\let\XINT_DivisionUndefined-signal \XINT_ConversionSyntax-signal
\let\XINT_InvalidContext-signal    \XINT_ConversionSyntax-signal
\catcode`- 12
\def\XINT_signalcondition #1{\expandafter\XINT_signalcondition_a
    \romannumeral0\ifcsname XINT_#1-signal\endcsname
                     \xint_dothis{\csname XINT_#1-signal\endcsname}%
                  \fi\xint_orthat{{#1}}{#1}}%
\def\XINT_signalcondition_a #1#2#3#4#5{% copied over from Python Decimal module
% #1=signal, #2=condition, #3=explanation for user,
% #4=context for error handlers, #5=used
    \ifcsname XINT_#1_isignoredflag\endcsname
       \xint_dothis{\csname XINT_#1.handler\endcsname {#4}}%
    \fi
    \expandafter\xint_gobble_i\csname XINT_#1Flag_ON\endcsname
    \unless\ifcsname XINT_#1_istrapped\endcsname
       \xint_dothis{\csname XINT_#2.handler\endcsname {#4}}%
```

```
    \fi
    \xint_orthat{%
       % the flag raised is named after the signal #1, but we show condition #2
       \XINT_expandableerror{#2 (hit <RET> thrice)}%
       \XINT_expandableerror{#3}%
       \XINT_expandableerror{next: #5}%
       % not for X3.274
       %\XINT_expandableerror{<RET>, or I\xintUse{...}<RET>, or I\xintCTRLC<RET>}%
       \xint_stop_atfirstofone{#5}%
    }%
}%
%% \let\xintUse\xint_stop_atfirstofthree % defined in xint.sty
\def\XINT_ifFlagRaised #1{%
    \ifcsname XINT_#1Flag_ON\endcsname
        \expandafter\xint_firstoftwo
    \else
        \expandafter\xint_secondoftwo
    \fi}%
\def\XINT_resetFlag #1%
    {\expandafter\let\csname XINT_#1Flag_ON\endcsname\XINT_undefined}%
\def\XINT_resetFlags {% WIP
    \XINT_resetFlag{InvalidOperation}% also from DivisionUndefined
    \XINT_resetFlag{DivisionByZero}%
    \XINT_resetFlag{Underflow}% (\xintiiPow with negative exponent)
    \XINT_resetFlag{Overflow}%   not encountered so far in xint code 1.2l
    % .. others ..
}%
\def\XINT_RaiseFlag #1{\expandafter\xint_gobble_i\csname XINT_#1Flag_ON\endcsname}%
```

NOT IMPLEMENTED! WORK IN PROGRESS! (ALL SIGNALS TRAPPED, NO HANDLERS USED)

```
\catcode`. 11
\let\XINT_Clamped.handler\xint_firstofone % WIP
\def\XINT_InvalidOperation.handler#1{_NaN}% WIP
\def\XINT_ConversionSyntax.handler#1{_NaN}% WIP
\def\XINT_DivisionByZero.handler#1{_SignedInfinity(#1)}% WIP
\def\XINT_DivisionImpossible.handler#1{_NaN}% WIP
\def\XINT_DivisionUndefined.handler#1{_NaN}%  WIP
\let\XINT_Inexact.handler\xint_firstofone  %  WIP
\def\XINT_InvalidContext.handler#1{_NaN}%     WIP
\let\XINT_Rounded.handler\xint_firstofone  %  WIP
\let\XINT_Subnormal.handler\xint_firstofone%  WIP
\def\XINT_Overflow.handler#1{_NaN}%  WIP
\def\XINT_Underflow.handler#1{_NaN}% WIP
\catcode`. 12
```

4.4 Counts for holding needed constants

```
\ifdefined\m@ne\let\xint_c_mone\m@ne
          \else\csname newcount\endcsname\xint_c_mone \xint_c_mone -1 \fi
\ifdefined\xint_c_x^viii\else
\csname newcount\endcsname\xint_c_x^viii \xint_c_x^viii   100000000
\fi
\ifdefined\xint_c_x^ix\else
```

```
\csname newcount\endcsname\xint_c_x^ix    \xint_c_x^ix     1000000000
\fi
\newcount\xint_c_x^viii_mone         \xint_c_x^viii_mone     99999999
\newcount\xint_c_xii_e_viii           \xint_c_xii_e_viii   1200000000
\newcount\xint_c_xi_e_viii_mone   \xint_c_xi_e_viii_mone   1099999999
```

Routines handling integers as lists of token digits

Routines handling big integers which are lists of digit tokens with no special additional structure.

Some routines do not accept non properly terminated inputs like "\the\numexpr1", or "\the\mathcode`\-", others do.

These routines or their sub-routines are mainly for internal usage.

4.5 \XINT_cuz_small

\XINT_cuz_small removes leading zeroes from the first eight digits. Expands following \romannumeral0. At least one digit is produced.

```
\def\XINT_cuz_small#1{%
\def\XINT_cuz_small ##1##2##3##4##5##6##7##8%
{%
    \expandafter#1\the\numexpr ##1##2##3##4##5##6##7##8\relax
}}\XINT_cuz_small{ }%
```

4.6 \xintNum, \xintiNum

For example \xintNum {----+-+++---+----000000000000003}

Very old routine got completely rewritten at 1.2l.

New code uses \numexpr governed expansion and fixes some issues of former version particularly regarding inputs of the \numexpr...\relax type without \the or \number prefix, and/or possibly no terminating \relax.

\xintiNum{\numexpr 1}\foo in earlier versions caused premature expansion of \foo.

\xintiNum{\the\numexpr 1} was ok, but a bit luckily so.

Also, up to 1.2k inclusive, the macro fetched tokens eight by eight, and not nine by nine as is done now. I have no idea why.

\xintNum gets redefined by xintfrac.

```
\def\xintiNum {\romannumeral0\xintinum }%
\def\xintinum #1%
{%
    \expandafter\XINT_num_cleanup\the\numexpr\expandafter\XINT_num_loop
    \romannumeral`&&@#1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
}%
\def\xintNum {\romannumeral0\xintnum }%
\let\xintnum\xintinum
\def\XINT_num #1%
{%
    \expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
    #1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
}%
\def\XINT_num_loop #1#2#3#4#5#6#7#8#9%
{%
```

```
    \xint_gob_til_xint: #9\XINT_num_end\xint:
    #1#2#3#4#5#6#7#8#9%
    \ifnum \numexpr #1#2#3#4#5#6#7#8#9+\xint_c_ = \xint_c_
```

means that so far only signs encountered, (if syntax is legal) then possibly zeroes or a terminated or not terminated \numexpr evaluating to zero In that latter case a correct zero will be produced in the end.

```
      \expandafter\XINT_num_loop
    \else
```

non terminated \numexpr (with nine tokens total) are safe as after \fi, there is then \xint:

```
     \expandafter\relax
    \fi
}%
\def\XINT_num_end\xint:#1\xint:{#1+\xint_c_\xint:}% empty input ok
\def\XINT_num_cleanup #1\xint:#2\Z { #1}%
```

4.7 \xintiiSgn

1.2l made \xintiiSgn robust against non terminated input.
1.2o deprecates here \xintSgn (it requires xintfrac.sty).

```
\def\xintiiSgn {\romannumeral0\xintiisgn }%
\def\xintiisgn #1%
{%
    \expandafter\XINT_sgn \romannumeral`&&@#1\xint:
}%
\def\XINT_sgn #1#2\xint:
{%
    \xint_UDzerominusfork
      #1-{ 0}%
      0#1{-1}%
       0-{ 1}%
    \krof
}%
\def\XINT_Sgn #1#2\xint:
{%
    \xint_UDzerominusfork
      #1-{0}%
      0#1{-1}%
       0-{1}%
    \krof
}%
\def\XINT_cntSgn #1#2\xint:
{%
    \xint_UDzerominusfork
      #1-\xint_c_
      0#1\xint_c_mone
       0-\xint_c_i
    \krof
}%
```

4.8 \xintiiOpp

Attention, \xintiiOpp non robust against non terminated inputs. Reason is I don't want to have to grab a delimiter at the end, as everything happens "upfront".

```
\def\xintiiOpp {\romannumeral0\xintiiopp }%
\def\xintiiopp #1%
{%
    \expandafter\XINT_opp \romannumeral`&&@#1%
}%
\def\XINT_Opp #1{\romannumeral0\XINT_opp #1}%
\def\XINT_opp #1%
{%
    \xint_UDzerominusfork
      #1-{ 0}%      zero
      0#1{ }%     negative
       0-{ -#1}%  positive
    \krof
}%
```

4.9 \xintiiAbs

Attention \xintiiAbs non robust against non terminated input.

```
\def\xintiiAbs {\romannumeral0\xintiiabs }%
\def\xintiiabs #1%
{%
    \expandafter\XINT_abs \romannumeral`&&@#1%
}%
\def\XINT_abs #1%
{%
    \xint_UDsignfork
      #1{ }%
       -{ #1}%
    \krof
}%
\def\XINT_Abs #1%
{%
    \xint_UDsignfork
      #1{}%
       -{#1}%
    \krof
}%
```

4.10 \xintFDg

FIRST DIGIT.
  1.2l: \xintiiFDg made robust against non terminated input.
  1.2o deprecates \xintiiFDg, gives to \xintFDg former meaning of \xintiiFDg.

```
\def\xintFDg {\romannumeral0\xintfdg }%
\def\xintfdg #1{\expandafter\XINT_fdg \romannumeral`&&@#1\xint:\Z}%
\def\XINT_FDg #1%
```

```
222    {\romannumeral0\expandafter\XINT_fdg\romannumeral`&&@\xintnum{#1}\xint:\Z }%
223 \def\XINT_fdg #1#2#3\Z
224 {%
225     \xint_UDzerominusfork
226       #1-{ 0}%   zero
227       0#1{ #2}%  negative
228        0-{ #1}%  positive
229     \krof
230 }%
```

4.11 \xintLDg

LAST DIGIT.
Rewritten for 1.2i (2016/12/10). Surprisingly perhaps, it is faster than \xintLastItem from xintkernel.sty despite the \numexpr operations.
1.2o deprecates \xintiiLDg, gives to \xintLDg former meaning of \xintiiLDg.
Attention \xintLDg non robust against non terminated input.

```
231 \def\xintLDg {\romannumeral0\xintldg }%
232 \def\xintldg #1{\expandafter\XINT_ldg_fork\romannumeral`&&@#1%
233     \XINT_ldg_c{}{}{}{}{}{}{}{}\xint_bye\relax}%
234 \def\XINT_ldg_fork #1%
235 {%
236     \xint_UDsignfork
237       #1\XINT_ldg
238        -{\XINT_ldg#1}%
239     \krof
240 }%
241 \def\XINT_ldg #1{%
242 \def\XINT_ldg ##1##2##3##4##5##6##7##8##9%
243    {\expandafter#1%
244     \the\numexpr##9##8##7##6##5##4##3##2##1*\xint_c_+\XINT_ldg_a##9}%
245 }\XINT_ldg{ }%
246 \def\XINT_ldg_a#1#2{\XINT_ldg_cbye#2\XINT_ldg_d#1\XINT_ldg_c\XINT_ldg_b#2}%
247 \def\XINT_ldg_b#1#2#3#4#5#6#7#8#9{#9#8#7#6#5#4#3#2#1*\xint_c_+\XINT_ldg_a#9}%
248 \def\XINT_ldg_c    #1#2\xint_bye{#1}%
249 \def\XINT_ldg_cbye #1\XINT_ldg_c{}%
250 \def\XINT_ldg_d#1#2\xint_bye{#1}%
```

4.12 \xintDouble

Attention \xintDouble non robust against non terminated input.

```
251 \def\xintDouble {\romannumeral0\xintdouble}%
252 \def\xintdouble #1{\expandafter\XINT_dbl_fork\romannumeral`&&@#1%
253                    \xint_bye2345678\xint_bye*\xint_c_ii\relax}%
254 \def\XINT_dbl_fork #1%
255 {%
256     \xint_UDsignfork
257      #1\XINT_dbl_neg
258       -\XINT_dbl
259     \krof #1%
260 }%
```

```
\def\XINT_dbl_neg-{\expandafter-\romannumeral0\XINT_dbl}%
\def\XINT_dbl #1{%
\def\XINT_dbl ##1##2##3##4##5##6##7##8%
   {\expandafter#1\the\numexpr##1##2##3##4##5##6##7##8\XINT_dbl_a}%
}\XINT_dbl{ }%
\def\XINT_dbl_a #1#2#3#4#5#6#7#8%
   {\expandafter\XINT_dbl_e\the\numexpr 1#1#2#3#4#5#6#7#8\XINT_dbl_a}%
\def\XINT_dbl_e#1{*\xint_c_ii\if#13+\xint_c_i\fi\relax}%
```

4.13 \xintHalf

Attention \xintHalf non robust against non terminated input.

```
\def\xintHalf {\romannumeral0\xinthalf}%
\def\xinthalf #1{\expandafter\XINT_half_fork\romannumeral`&&@#1%
    \xint_bye\xint_Bye345678\xint_bye
    *\xint_c_v+\xint_c_v)/\xint_c_x-\xint_c_i\relax}%
\def\XINT_half_fork #1%
{%
    \xint_UDsignfork
     #1\XINT_half_neg
      -\XINT_half
    \krof #1%
}%
\def\XINT_half_neg-{\xintiiopp\XINT_half}%
\def\XINT_half #1{%
\def\XINT_half ##1##2##3##4##5##6##7##8%
   {\expandafter#1\the\numexpr(##1##2##3##4##5##6##7##8\XINT_half_a}%
}\XINT_half{ }%
\def\XINT_half_a#1{\xint_Bye#1\xint_bye\XINT_half_b#1}%
\def\XINT_half_b #1#2#3#4#5#6#7#8%
   {\expandafter\XINT_half_e\the\numexpr(1#1#2#3#4#5#6#7#8\XINT_half_a}%
\def\XINT_half_e#1{*\xint_c_v+#1-\xint_c_v)\relax}%
```

4.14 \xintInc

1.2i much delayed complete rewrite in 1.2 style.
  As we take 9 by 9 with the input save stack at 5000 this allows a bit less than 9 times 2500 = 22500 digits on input.
  Attention \xintInc non robust against non terminated input.

```
\def\xintInc {\romannumeral0\xintinc}%
\def\xintinc #1{\expandafter\XINT_inc_fork\romannumeral`&&@#1%
                 \xint_bye23456789\xint_bye+\xint_c_i\relax}%
\def\XINT_inc_fork #1%
{%
    \xint_UDsignfork
      #1\XINT_inc_neg
       -\XINT_inc
    \krof #1%
}%
\def\XINT_inc_neg-#1\xint_bye#2\relax
   {\xintiiopp\XINT_dec #1\XINT_dec_bye234567890\xint_bye}%
```

```
\def\XINT_inc #1{%
\def\XINT_inc ##1##2##3##4##5##6##7##8##9%
   {\expandafter#1\the\numexpr##1##2##3##4##5##6##7##8##9\XINT_inc_a}%
}\XINT_inc{ }%
\def\XINT_inc_a #1#2#3#4#5#6#7#8#9%
   {\expandafter\XINT_inc_e\the\numexpr 1#1#2#3#4#5#6#7#8#9\XINT_inc_a}%
\def\XINT_inc_e#1{\if#12+\xint_c_i\fi\relax}%
```

4.15 \xintDec

1.2i much delayed complete rewrite in the 1.2 style. Things are a bit more complicated than \xintInc because 2999999999 is too big for TeX.
  Attention \xintDec non robust against non terminated input.

```
\def\xintDec {\romannumeral0\xintdec}%
\def\xintdec #1{\expandafter\XINT_dec_fork\romannumeral`&&@#1%
                \XINT_dec_bye234567890\xint_bye}%
\def\XINT_dec_fork #1%
{%
    \xint_UDsignfork
      #1\XINT_dec_neg
       -\XINT_dec
    \krof #1%
}%
\def\XINT_dec_neg-#1\XINT_dec_bye#2\xint_bye
   {\expandafter-%
    \romannumeral0\XINT_inc #1\xint_bye23456789\xint_bye+\xint_c_i\relax}%
\def\XINT_dec #1{%
\def\XINT_dec ##1##2##3##4##5##6##7##8##9%
   {\expandafter#1\the\numexpr##1##2##3##4##5##6##7##8##9\XINT_dec_a}%
}\XINT_dec{ }%
\def\XINT_dec_a #1#2#3#4#5#6#7#8#9%
   {\expandafter\XINT_dec_e\the\numexpr 1#1#2#3#4#5#6#7#8#9\XINT_dec_a}%
\def\XINT_dec_bye #1\XINT_dec_a#2#3\xint_bye
         {\if#20-\xint_c_ii\relax+\else-\fi\xint_c_i\relax}%
\def\XINT_dec_e#1{\unless\if#11\xint_dothis{-\xint_c_i#1}\fi\xint_orthat\relax}%
```

4.16 \xintDSL

DECIMAL SHIFT LEFT (=MULTIPLICATION PAR 10). Rewritten for 1.2i. This was very old code... I never came back to it, but I should have rewritten it long time ago.
  Attention \xintDSL non robust against non terminated input.

```
\def\xintDSL {\romannumeral0\xintdsl }%
\def\xintdsl #1{\expandafter\XINT_dsl\romannumeral`&&@#10}%
\def\XINT_dsl#1{%
\def\XINT_dsl ##1{\xint_gob_til_zero ##1\xint_dsl_zero 0#1##1}%
}\XINT_dsl{ }%
\def\xint_dsl_zero 0 0{ }%
```

4.17 \xintDSR

Decimal shift right, truncates towards zero. Rewritten for 1.2i. Limited to 22483 digits on input.
Attention \xintDSR non robust against non terminated input.

```
\def\xintDSR{\romannumeral0\xintdsr}%
\def\xintdsr #1{\expandafter\XINT_dsr_fork\romannumeral`&&@#1%
    \xint_bye\xint_Bye3456789\xint_bye+\xint_c_v)/\xint_c_x-\xint_c_i\relax}%
\def\XINT_dsr_fork #1%
{%
    \xint_UDsignfork
      #1\XINT_dsr_neg
       -\XINT_dsr
    \krof #1%
}%
\def\XINT_dsr_neg-{\xintiiopp\XINT_dsr}%
\def\XINT_dsr #1{%
\def\XINT_dsr ##1##2##3##4##5##6##7##8##9%
   {\expandafter#1\the\numexpr(##1##2##3##4##5##6##7##8##9\XINT_dsr_a}%
}\XINT_dsr{ }%
\def\XINT_dsr_a#1{\xint_Bye#1\xint_bye\XINT_dsr_b#1}%
\def\XINT_dsr_b #1#2#3#4#5#6#7#8#9%
   {\expandafter\XINT_dsr_e\the\numexpr(1#1#2#3#4#5#6#7#8#9\XINT_dsr_a}%
\def\XINT_dsr_e #1{)\relax}%
```

4.18 \xintDSRr

New with 1.2i. Decimal shift right, rounds away from zero; done in the 1.2 spirit (with much delay, sorry). Used by \xintRound, \xintDivRound.
This is about the first time I am happy that the division in \numexpr rounds!
Attention \xintDSRr non robust against non terminated input.

```
\def\xintDSRr{\romannumeral0\xintdsrr}%
\def\xintdsrr #1{\expandafter\XINT_dsrr_fork\romannumeral`&&@#1%
                \xint_bye\xint_Bye3456789\xint_bye/\xint_c_x\relax}%
\def\XINT_dsrr_fork #1%
{%
    \xint_UDsignfork
      #1\XINT_dsrr_neg
       -\XINT_dsrr
    \krof #1%
}%
\def\XINT_dsrr_neg-{\xintiiopp\XINT_dsrr}%
\def\XINT_dsrr #1{%
\def\XINT_dsrr ##1##2##3##4##5##6##7##8##9%
   {\expandafter#1\the\numexpr##1##2##3##4##5##6##7##8##9\XINT_dsrr_a}%
}\XINT_dsrr{ }%
\def\XINT_dsrr_a#1{\xint_Bye#1\xint_bye\XINT_dsrr_b#1}%
\def\XINT_dsrr_b #1#2#3#4#5#6#7#8#9%
   {\expandafter\XINT_dsrr_e\the\numexpr1#1#2#3#4#5#6#7#8#9\XINT_dsrr_a}%
\let\XINT_dsrr_e\XINT_inc_e
```

Blocks of eight digits

The lingua of release 1.2.

4.19 \XINT_cuz

This (launched by \romannumeral0) iterately removes all leading zeroes from a sequence of 8N digits ended by \R.

Rewritten for 1.2l, now uses \numexpr governed expansion and \ifnum test rather than delimited gobbling macros.

Note 2015/11/28: with only four digits the gob_til_fourzeroes had proved in some old testing faster than \ifnum test. But with eight digits, the execution times are much closer, as I tested back then.

```
\def\XINT_cuz #1{%
\def\XINT_cuz {\expandafter#1\the\numexpr\XINT_cuz_loop}%
}\XINT_cuz{ }%
\def\XINT_cuz_loop #1#2#3#4#5#6#7#8#9%
{%
    #1#2#3#4#5#6#7#8%
       \xint_gob_til_R #9\XINT_cuz_hitend\R
       \ifnum #1#2#3#4#5#6#7#8>\xint_c_
             \expandafter\XINT_cuz_cleantoend
       \else\expandafter\XINT_cuz_loop
       \fi #9%
}%
\def\XINT_cuz_hitend\R #1\R{\relax}%
\def\XINT_cuz_cleantoend #1\R{\relax #1}%
```

4.20 \XINT_cuz_byviii

This removes eight by eight leading zeroes from a sequence of 8N digits ended by \R. Thus, we still have 8N digits on output. Expansion started by \romannumeral0

```
\def\XINT_cuz_byviii #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_R #9\XINT_cuz_byviii_e \R
    \xint_gob_til_eightzeroes #1#2#3#4#5#6#7#8\XINT_cuz_byviii_z 00000000%
    \XINT_cuz_byviii_done #1#2#3#4#5#6#7#8#9%
}%
\def\XINT_cuz_byviii_z 00000000\XINT_cuz_byviii_done 00000000{\XINT_cuz_byviii}%
\def\XINT_cuz_byviii_done #1\R { #1}%
\def\XINT_cuz_byviii_e\R #1\XINT_cuz_byviii_done #2\R{ #2}%
```

4.21 \XINT_unsep_loop

This is used as

```
   \the\numexpr0\XINT_unsep_loop (blocks of 1<8digits>!)
                \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax
```

It removes the 1's and !'s, and outputs the 8N digits with a 0 token as as prefix which will have to be cleaned out by caller.

Actually it does not matter whether the blocks contain really 8 digits, all that matters is that they have 1 as first digit (and at most 9 digits after that to obey the TeX-\numexpr bound).

Done at 1.2l for usage by other macros. The similar code in earlier releases was strangely in O(N^2) style, apparently to avoid some memory constraints. But these memory constraints related to \numexpr chaining seems to be in many places in xint code base. The 1.2l version is written in

```
the 1.2i style of \xintInc etc... and is compatible with some 1! block without digits among the
treated blocks, they will disappear.
```

```
\def\XINT_unsep_loop #1!#2!#3!#4!#5!#6!#7!#8!#9!%
{%
    \expandafter\XINT_unsep_clean
    \the\numexpr #1\expandafter\XINT_unsep_clean
    \the\numexpr #2\expandafter\XINT_unsep_clean
    \the\numexpr #3\expandafter\XINT_unsep_clean
    \the\numexpr #4\expandafter\XINT_unsep_clean
    \the\numexpr #5\expandafter\XINT_unsep_clean
    \the\numexpr #6\expandafter\XINT_unsep_clean
    \the\numexpr #7\expandafter\XINT_unsep_clean
    \the\numexpr #8\expandafter\XINT_unsep_clean
    \the\numexpr #9\XINT_unsep_loop
}%
\def\XINT_unsep_clean 1{\relax}%
```

4.22 \XINT_unsep_cuzsmall

```
This is used as
   \romannumeral0\XINT_unsep_cuzsmall (blocks of 1<8d>!)
                \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax
It removes the 1's and !'s, and removes the leading zeroes *of the first block*.
  Redone for 1.2l: the 1.2 variant was strangely in O(N^2) style.
```

```
\def\XINT_unsep_cuzsmall
{%
    \expandafter\XINT_unsep_cuzsmall_x\the\numexpr0\XINT_unsep_loop
}%
\def\XINT_unsep_cuzsmall_x #1{%
\def\XINT_unsep_cuzsmall_x 0##1##2##3##4##5##6##7##8%
{%
    \expandafter#1\the\numexpr ##1##2##3##4##5##6##7##8\relax
}}\XINT_unsep_cuzsmall_x{ }%
```

4.23 \XINT_div_unsepQ

```
This is used by division to remove separators from the produced quotient. The quotient is produced
in the correct order. The routine will also remove leading zeroes. An extra initial block of 8
zeroes is possible and thus if present must be removed. Then the next eight digits must be cleaned
of leading zeroes. Attention that there might be a single block of 8 zeroes. Expansion launched by
\romannumeral0.
  Rewritten for 1.2l in 1.2i style.
```

```
\def\XINT_div_unsepQ_delim {\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax\Z}%
\def\XINT_div_unsepQ
{%
    \expandafter\XINT_div_unsepQ_x\the\numexpr0\XINT_unsep_loop
}%
\def\XINT_div_unsepQ_x #1{%
\def\XINT_div_unsepQ_x 0##1##2##3##4##5##6##7##8##9%
{%
```

```
    \xint_gob_til_Z ##9\XINT_div_unsepQ_one\Z
    \xint_gob_til_eightzeroes ##1##2##3##4##5##6##7##8\XINT_div_unsepQ_y 00000000%
    \expandafter#1\the\numexpr ##1##2##3##4##5##6##7##8\relax ##9%
}}\XINT_div_unsepQ_x{ }%
\def\XINT_div_unsepQ_y #1{%
\def\XINT_div_unsepQ_y ##1\relax ##2##3##4##5##6##7##8##9%
{%
    \expandafter#1\the\numexpr ##2##3##4##5##6##7##8##9\relax
}}\XINT_div_unsepQ_y{ }%
\def\XINT_div_unsepQ_one#1\expandafter{\expandafter}%
```

4.24 \XINT_div_unsepR

This is used by division to remove separators from the produced remainder. The remainder is here in correct order. It must be cleaned of leading zeroes, possibly all the way.
  Also rewritten for 1.2l, the 1.2 version was O(N^2) style.
  Terminator \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax\R
  We have a need for something like \R because it is not guaranteed the thing is not actually zero.

```
\def\XINT_div_unsepR
{%
    \expandafter\XINT_div_unsepR_x\the\numexpr0\XINT_unsep_loop
}%
\def\XINT_div_unsepR_x#1{%
\def\XINT_div_unsepR_x 0{\expandafter#1\the\numexpr\XINT_cuz_loop}%
}\XINT_div_unsepR_x{ }%
```

4.25 \XINT_zeroes_forviii

 \romannumeral0\XINT_zeroes_forviii #1\R\R\R\R\R\R\R{10}0000001\W
produces a string of k 0's such that k+length(#1) is smallest bigger multiple of eight.

```
\def\XINT_zeroes_forviii #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_R #8\XINT_zeroes_forviii_end\R\XINT_zeroes_forviii
}%
\def\XINT_zeroes_forviii_end#1{%
\def\XINT_zeroes_forviii_end\R\XINT_zeroes_forviii ##1##2##3##4##5##6##7##8##9\W
{%
    \expandafter#1\xint_gob_til_one ##2##3##4##5##6##7##8%
}}\XINT_zeroes_forviii_end{ }%
```

4.26 \XINT_sepbyviii_Z

This is used as
   \the\numexpr\XINT_sepbyviii_Z <8Ndigits>\XINT_sepbyviii_Z_end 2345678\relax
It produces 1<8d>!...1<8d>!1;!
  Prior to 1.2l it used \Z as terminator not the semi-colon (hence the name). The switch to ; was done at a time I thought perhaps I would use an internal format maintaining such 8 digits blocks, and this has to be compatible with the \csname...\endcsname encapsulation in \xintexpr parsers.

```
\def\XINT_sepbyviii_Z #1#2#3#4#5#6#7#8%
{%
```

```
    1#1#2#3#4#5#6#7#8\expandafter!\the\numexpr\XINT_sepbyviii_Z
}%
\def\XINT_sepbyviii_Z_end #1\relax {;!}%
```

4.27 \XINT_sepbyviii_andcount

This is used as
   \the\numexpr\XINT_sepbyviii_andcount <8Ndigits>%
       \XINT_sepbyviii_end 2345678\relax
       \xint_c_vii!\xint_c_vi!\xint_c_v!\xint_c_iv!%
       \xint_c_iii!\xint_c_ii!\xint_c_i!\xint_c_\W
It will produce
   1<8d>!1<8d>!....1<8d>!1\xint:<count of blocks>\xint:
Used by \XINT_div_prepare_g for \XINT_div_prepare_h, and also by \xintiiCmp.

```
\def\XINT_sepbyviii_andcount
{%
    \expandafter\XINT_sepbyviii_andcount_a\the\numexpr\XINT_sepbyviii
}%
\def\XINT_sepbyviii #1#2#3#4#5#6#7#8%
{%
    1#1#2#3#4#5#6#7#8\expandafter!\the\numexpr\XINT_sepbyviii
}%
\def\XINT_sepbyviii_end #1\relax {\relax\XINT_sepbyviii_andcount_end!}%
\def\XINT_sepbyviii_andcount_a {\XINT_sepbyviii_andcount_b \xint_c_\xint:}%
\def\XINT_sepbyviii_andcount_b #1\xint:#2!#3!#4!#5!#6!#7!#8!#9!%
{%
    #2\expandafter!\the\numexpr#3\expandafter!\the\numexpr#4\expandafter
    !\the\numexpr#5\expandafter!\the\numexpr#6\expandafter!\the\numexpr
    #7\expandafter!\the\numexpr#8\expandafter!\the\numexpr#9\expandafter!\the\numexpr
    \expandafter\XINT_sepbyviii_andcount_b\the\numexpr #1+\xint_c_viii\xint:%
}%
\def\XINT_sepbyviii_andcount_end #1\XINT_sepbyviii_andcount_b\the\numexpr
    #2+\xint_c_viii\xint:#3#4\W {\expandafter\xint:\the\numexpr #2+#3\xint:}%
```

4.28 \XINT_rsepbyviii

This is used as
   \the\numexpr1\XINT_rsepbyviii <8Ndigits>%
                \XINT_rsepbyviii_end_A 2345678%
                \XINT_rsepbyviii_end_B 2345678\relax UV%
and will produce
   1<8digits>!1<8digits>\xint:1<8digits>!...
where the original digits are organized by eight, and the order inside successive pairs of blocks separated by \xint: has been reversed. Output ends either in 1<8d>!1<8d>\xint:1U\xint: (even) or 1<8d>!1<8d>\xint:1V!1<8d>\xint: (odd)
  The U an V should be \numexpr1 stoppers (or will expand and be ended by !). This macro is currently (1.2..1.2l) exclusively used in combination with \XINT_sepandrev_andcount or \XINT_sepandrev.

```
\def\XINT_rsepbyviii #1#2#3#4#5#6#7#8%
{%
    \XINT_rsepbyviii_b {#1#2#3#4#5#6#7#8}%
}%
```

```
\def\XINT_rsepbyviii_b #1#2#3#4#5#6#7#8#9%
{%
    #2#3#4#5#6#7#8#9\expandafter!\the\numexpr
    1#1\expandafter\xint:\the\numexpr 1\XINT_rsepbyviii
}%
\def\XINT_rsepbyviii_end_B #1\relax #2#3{#2\xint:}%
\def\XINT_rsepbyviii_end_A #11#2\expandafter #3\relax #4#5{#5!1#2\xint:}%
```

4.29 \XINT_sepandrev

This is used typically as
```
   \romannumeral0\XINT_sepandrev <8Ndigits>%
                \XINT_rsepbyviii_end_A 2345678%
                \XINT_rsepbyviii_end_B 2345678\relax UV%
                \R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\W
```
and will produce
```
   1<8digits>!1<8digits>!1<8digits>!...
```
where the blocks have been globally reversed. The UV here are only place holders (must be \numexpr1 stoppers) to share same syntax as \XINT_sepandrev_andcount, they are gobbled (#2 in \XINT_sepandrev_done).

```
\def\XINT_sepandrev
{%
    \expandafter\XINT_sepandrev_a\the\numexpr 1\XINT_rsepbyviii
}%
\def\XINT_sepandrev_a {\XINT_sepandrev_b {}}%
\def\XINT_sepandrev_b #1#2\xint:#3\xint:#4\xint:#5\xint:#6\xint:#7\xint:#8\xint:#9\xint:%
{%
    \xint_gob_til_R #9\XINT_sepandrev_end\R
    \XINT_sepandrev_b {#9!#8!#7!#6!#5!#4!#3!#2!#1}%
}%
\def\XINT_sepandrev_end\R\XINT_sepandrev_b #1#2\W {\XINT_sepandrev_done #1}%
\def\XINT_sepandrev_done #11#2!{ }%
```

4.30 \XINT_sepandrev_andcount

This is used typically as
```
   \romannumeral0\XINT_sepandrev_andcount <8Ndigits>%
               \XINT_rsepbyviii_end_A 2345678%
               \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
               \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
               \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
```
and will produce
```
   <length>.1<8digits>!1<8digits>!1<8digits>!...
```
where the blocks have been globally reversed and <length> is the number of blocks.

```
\def\XINT_sepandrev_andcount
{%
    \expandafter\XINT_sepandrev_andcount_a\the\numexpr 1\XINT_rsepbyviii
}%
\def\XINT_sepandrev_andcount_a {\XINT_sepandrev_andcount_b 0!{}}%
\def\XINT_sepandrev_andcount_b #1!#2#3\xint:#4\xint:#5\xint:#6\xint:#7\xint:#8\xint:#9\xint:%
{%
```

```
    \xint_gob_til_R #9\XINT_sepandrev_andcount_end\R
    \expandafter\XINT_sepandrev_andcount_b \the\numexpr #1+\xint_c_i!%
    {#9!#8!#7!#6!#5!#4!#3!#2}%
}%
\def\XINT_sepandrev_andcount_end\R
    \expandafter\XINT_sepandrev_andcount_b\the\numexpr #1+\xint_c_i!#2#3#4\W
{\expandafter\XINT_sepandrev_andcount_done\the\numexpr #3+\xint_c_xiv*#1!#2}%
\def\XINT_sepandrev_andcount_done#1{%
\def\XINT_sepandrev_andcount_done##1!##21##3!{\expandafter#1\the\numexpr##1-##3\xint:}%
}\XINT_sepandrev_andcount_done{ }%
```

4.31 \XINT_rev_nounsep

This is used as
   \romannumeral0\XINT_rev_nounsep {}<blocks 1<8d>!>\R!\R!\R!\R!\R!\R!\R!\W
It reverses the blocks, keeping the 1's and ! separators. Used multiple times in the division algorithm. The inserted {} here is not optional.

```
\def\XINT_rev_nounsep #1#2!#3!#4!#5!#6!#7!#8!#9!%
{%
    \xint_gob_til_R #9\XINT_rev_nounsep_end\R
    \XINT_rev_nounsep {#9!#8!#7!#6!#5!#4!#3!#2!#1}%
}%
\def\XINT_rev_nounsep_end\R\XINT_rev_nounsep #1#2\W {\XINT_rev_nounsep_done #1}%
\def\XINT_rev_nounsep_done #11{ 1}%
```

4.32 \XINT_unrevbyviii

Used as \romannumeral0\XINT_unrevbyviii 1<8d>!....1<8d>! terminated by
   1;!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
The \romannumeral in unrevbyviii_a is for special effects (expand some token which was put as 1<token>! at the end of the original blocks). This mechanism is used by 1.2 subtraction (still true for 1.2l).

```
\def\XINT_unrevbyviii #11#2!1#3!1#4!1#5!1#6!1#7!1#8!1#9!%
{%
    \xint_gob_til_R #9\XINT_unrevbyviii_a\R
    \XINT_unrevbyviii {#9#8#7#6#5#4#3#2#1}%
}%
\def\XINT_unrevbyviii_a#1{%
\def\XINT_unrevbyviii_a\R\XINT_unrevbyviii ##1##2\W
    {\expandafter#1\romannumeral`&&@\xint_gob_til_sc ##1}%
}\XINT_unrevbyviii_a{ }%
```

Can work with shorter ending pattern: 1;!1\R!1\R!1\R!1\R!1\R!1\R!\W but the longer one of unrevbyviii is ok here too. Used currently (1.2) only by addition, now (1.2c) with long ending pattern. Does the final clean up of leading zeroes contrarily to general \XINT_unrevbyviii.

```
\def\XINT_smallunrevbyviii 1#1!1#2!1#3!1#4!1#5!1#6!1#7!1#8!#9\W%
{%
    \expandafter\XINT_cuz_small\xint_gob_til_sc #8#7#6#5#4#3#2#1%
}%
```

Core arithmetic

The four operations have been rewritten entirely for release 1.2. The new routines works with separated blocks of eight digits. They all measure first the lengths of the arguments, even addition and subtraction (this was not the case with xintcore.sty 1.1 or earlier.)

The technique of chaining \the\numexpr induces a limitation on the maximal size depending on the size of the input save stack and the maximum expansion depth. For the current (TL2015) settings (5000, resp. 10000), the induced limit for addition of numbers is at 19968 and for multiplication it is observed to be 19959 (valid as of 2015/10/07).

Side remark: I tested that \the\numexpr was more efficient than \number. But it reduced the allowable numbers for addition from 19976 digits to 19968 digits.

4.33 \xintiiAdd

1.2l: \xintiiAdd made robust against non terminated input.

```
\def\xintiiAdd   {\romannumeral0\xintiiadd }%
\def\xintiiadd #1{\expandafter\XINT_iiadd\romannumeral`&&@#1\xint:}%
\def\XINT_iiadd #1#2\xint:#3%
{%
    \expandafter\XINT_add_nfork\expandafter#1\romannumeral`&&@#3\xint:#2\xint:
}%
\def\XINT_add_fork #1#2\xint:#3\xint:{\XINT_add_nfork #1#3\xint:#2\xint:}%
\def\XINT_add_nfork #1#2%
{%
    \xint_UDzerofork
      #1\XINT_add_firstiszero
      #2\XINT_add_secondiszero
       0{}%
    \krof
    \xint_UDsignsfork
         #1#2\XINT_add_minusminus
          #1-\XINT_add_minusplus
          #2-\XINT_add_plusminus
           --\XINT_add_plusplus
    \krof #1#2%
}%
\def\XINT_add_firstiszero  #1\krof 0#2#3\xint:#4\xint:{ #2#3}%
\def\XINT_add_secondiszero #1\krof #20#3\xint:#4\xint:{ #2#4}%
\def\XINT_add_minusminus   #1#2%
   {\expandafter-\romannumeral0\XINT_add_pp_a {}{}}%
\def\XINT_add_minusplus    #1#2{\XINT_sub_mm_a {}#2}%
\def\XINT_add_plusminus    #1#2%
   {\expandafter\XINT_opp\romannumeral0\XINT_sub_mm_a #1{}}%
\def\XINT_add_pp_a #1#2#3\xint:
{%
  \expandafter\XINT_add_pp_b
      \romannumeral0\expandafter\XINT_sepandrev_andcount
      \romannumeral0\XINT_zeroes_forviii #2#3\R\R\R\R\R\R\R\R{10}0000001\W
      #2#3\XINT_rsepbyviii_end_A 2345678%
          \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
              \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
```

```
              \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
   \X #1%
}%
\let\XINT_add_plusplus \XINT_add_pp_a
\def\XINT_add_pp_b #1\xint:#2\X #3\xint:
{%
    \expandafter\XINT_add_checklengths
    \the\numexpr #1\expandafter\xint:%
    \romannumeral0\expandafter\XINT_sepandrev_andcount
    \romannumeral0\XINT_zeroes_forviii #3\R\R\R\R\R\R\R\R{10}0000001\W
    #3\XINT_rsepbyviii_end_A 2345678%
      \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
              \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
              \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
     1;!1;!1;!1;!\W #21;!1;!1;!1;!\W
     1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
}%
```

I keep #1.#2. to check if at most 6 + 6 base 10^8 digits which can be treated faster for final reverse. But is this overhead at all useful ?

```
\def\XINT_add_checklengths #1\xint:#2\xint:%
{%
    \ifnum #2>#1
       \expandafter\XINT_add_exchange
    \else
       \expandafter\XINT_add_A
    \fi
    #1\xint:#2\xint:%
}%
\def\XINT_add_exchange #1\xint:#2\xint:#3\W #4\W
{%
    \XINT_add_A #2\xint:#1\xint:#4\W #3\W
}%
\def\XINT_add_A #1\xint:#2\xint:%
{%
    \ifnum #1>\xint_c_vi
          \expandafter\XINT_add_aa
    \else \expandafter\XINT_add_aa_small
    \fi
}%
\def\XINT_add_aa {\expandafter\XINT_add_out\the\numexpr\XINT_add_a \xint_c_ii}%
\def\XINT_add_out{\expandafter\XINT_cuz_small\romannumeral0\XINT_unrevbyviii {}}%
\def\XINT_add_aa_small
    {\expandafter\XINT_smallunrevbyviii\the\numexpr\XINT_add_a \xint_c_ii}%
```

2 as first token of #1 stands for "no carry", 3 will mean a carry (we are adding 1<8digits> to 1<8digits>.) Version 1.2c has terminators of the shape 1;!, replacing the \Z! used in 1.2.

Call: \the\numexpr\XINT_add_a 2#11;!1;!1;!1;!\W #21;!1;!1;!1;!\W where #1 and #2 are blocks of 1<8d>!, and #1 is at most as long as #2. This last requirement is a bit annoying (if one wants to do recursive algorithms but not have to check lengths), and I will probably remove it at some point.

Output: blocks of 1<8d>! representing the addition, (least significant first), and a final 1;!. In recursive algorithm this 1;! terminator can thus conveniently be reused as part of input terminator (up to the length problem).

```
\def\XINT_add_a #1!#2!#3!#4!#5\W
                #6!#7!#8!#9!%
{%
    \XINT_add_b
        #1!#6!#2!#7!#3!#8!#4!#9!%
        #5\W
}%
\def\XINT_add_b #11#2#3!#4!%
{%
    \xint_gob_til_sc #2\XINT_add_bi ;%
    \expandafter\XINT_add_c\the\numexpr#1+1#2#3+#4-\xint_c_ii\xint:%
}%
\def\XINT_add_bi;\expandafter\XINT_add_c
    \the\numexpr#1+#2+#3-\xint_c_ii\xint:#4!#5!#6!#7!#8!#9!\W
{%
    \XINT_add_k #1#3!#5!#7!#9!%
}%
\def\XINT_add_c #1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_add_d #1%
}%
\def\XINT_add_d #11#2#3!#4!%
{%
    \xint_gob_til_sc #2\XINT_add_di ;%
    \expandafter\XINT_add_e\the\numexpr#1+1#2#3+#4-\xint_c_ii\xint:%
}%
\def\XINT_add_di;\expandafter\XINT_add_e
    \the\numexpr#1+#2+#3-\xint_c_ii\xint:#4!#5!#6!#7!#8\W
{%
    \XINT_add_k #1#3!#5!#7!%
}%
\def\XINT_add_e #1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_add_f #1%
}%
\def\XINT_add_f #11#2#3!#4!%
{%
    \xint_gob_til_sc #2\XINT_add_fi ;%
    \expandafter\XINT_add_g\the\numexpr#1+1#2#3+#4-\xint_c_ii\xint:%
}%
\def\XINT_add_fi;\expandafter\XINT_add_g
    \the\numexpr#1+#2+#3-\xint_c_ii\xint:#4!#5!#6\W
{%
    \XINT_add_k #1#3!#5!%
}%
\def\XINT_add_g #1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_add_h #1%
}%
\def\XINT_add_h #11#2#3!#4!%
{%
    \xint_gob_til_sc #2\XINT_add_hi ;%
```

```
    \expandafter\XINT_add_i\the\numexpr#1+1#2#3+#4-\xint_c_ii\xint:%
}%
\def\XINT_add_hi;%
    \expandafter\XINT_add_i\the\numexpr#1+#2+#3-\xint_c_ii\xint:#4\W
{%
    \XINT_add_k #1#3!%
}%
\def\XINT_add_i #1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_add_a #1%
}%
\def\XINT_add_k #1{\if #12\expandafter\XINT_add_ke\else\expandafter\XINT_add_l \fi}%
\def\XINT_add_ke #11;#2\W {\XINT_add_kf #11;!}%
\def\XINT_add_kf 1{1\relax }%
\def\XINT_add_l 1#1#2{\xint_gob_til_sc #1\XINT_add_lf ;\XINT_add_m 1#1#2}%
\def\XINT_add_lf #1\W {1\relax 00000001!1;!}%
\def\XINT_add_m #1!{\expandafter\XINT_add_n\the\numexpr\xint_c_i+#1\xint:}%
\def\XINT_add_n #1#2\xint:{1#2\expandafter!\the\numexpr\XINT_add_o #1}%
```

Here 2 stands for "carry", and 1 for "no carry" (we have been adding 1 to 1<8digits>.)

```
\def\XINT_add_o #1{\if #12\expandafter\XINT_add_l\else\expandafter\XINT_add_ke \fi}%
```

4.34 \xintiiCmp

Moved from xint.sty to xintcore.sty and rewritten for 1.2l.
1.2l's \xintiiCmp is robust against non terminated input.
1.2o deprecates \xintCmp, with xintfrac loaded it will get overwritten anyhow.

```
\def\xintiiCmp   {\romannumeral0\xintiicmp }%
\def\xintiicmp #1{\expandafter\XINT_iicmp\romannumeral`&&@#1\xint:}%
\def\XINT_iicmp #1#2\xint:#3%
{%
    \expandafter\XINT_cmp_nfork\expandafter #1\romannumeral`&&@#3\xint:#2\xint:
}%
\def\XINT_cmp_nfork #1#2%
{%
    \xint_UDzerofork
      #1\XINT_cmp_firstiszero
      #2\XINT_cmp_secondiszero
       0{}%
    \krof
    \xint_UDsignsfork
         #1#2\XINT_cmp_minusminus
          #1-\XINT_cmp_minusplus
          #2-\XINT_cmp_plusminus
           --\XINT_cmp_plusplus
    \krof #1#2%
}%
\def\XINT_cmp_firstiszero  #1\krof 0#2#3\xint:#4\xint:
{%
    \xint_UDzerominusfork
      #2-{ 0}%
```

```
      0#2{ 1}%
       0-{ -1}%
    \krof
}%
\def\XINT_cmp_secondiszero #1\krof #20#3\xint:#4\xint:
{%
    \xint_UDzerominusfork
      #2-{ 0}%
      0#2{ -1}%
       0-{ 1}%
    \krof
}%
\def\XINT_cmp_plusminus    #1\xint:#2\xint:{ 1}%
\def\XINT_cmp_minusplus    #1\xint:#2\xint:{ -1}%
\def\XINT_cmp_minusminus
    --{\expandafter\XINT_opp\romannumeral0\XINT_cmp_plusplus {}{}}%
\def\XINT_cmp_plusplus  #1#2#3\xint:
{%
    \expandafter\XINT_cmp_pp
    \the\numexpr\expandafter\XINT_sepbyviii_andcount
    \romannumeral0\XINT_zeroes_forviii #2#3\R\R\R\R\R\R\R\R{10}0000001\W
    #2#3\XINT_sepbyviii_end 2345678\relax
        \xint_c_vii!\xint_c_vi!\xint_c_v!\xint_c_iv!%
        \xint_c_iii!\xint_c_ii!\xint_c_i!\xint_c_\W
    #1%
}%
\def\XINT_cmp_pp #1\xint:#2\xint:#3\xint:
{%
    \expandafter\XINT_cmp_checklengths
    \the\numexpr #2\expandafter\xint:%
    \the\numexpr\expandafter\XINT_sepbyviii_andcount
    \romannumeral0\XINT_zeroes_forviii #3\R\R\R\R\R\R\R\R{10}0000001\W
    #3\XINT_sepbyviii_end 2345678\relax
        \xint_c_vii!\xint_c_vi!\xint_c_v!\xint_c_iv!%
        \xint_c_iii!\xint_c_ii!\xint_c_i!\xint_c_\W
    #1;!1;!1;!1;!\W
}%
\def\XINT_cmp_checklengths #1\xint:#2\xint:#3\xint:
{%
    \ifnum #1=#3
       \expandafter\xint_firstoftwo
    \else
       \expandafter\xint_secondoftwo
    \fi
    \XINT_cmp_a {\XINT_cmp_distinctlengths {#1}{#3}}#2;!1;!1;!1;!\W
}%
\def\XINT_cmp_distinctlengths #1#2#3\W #4\W
{%
    \ifnum #1>#2
        \expandafter\xint_firstoftwo
    \else
        \expandafter\xint_secondoftwo
```

```
    \fi
    { -1}{ 1}%
}%
\def\XINT_cmp_a 1#1!1#2!1#3!1#4!#5\W 1#6!1#7!1#8!1#9!%
{%
    \xint_gob_til_sc #1\XINT_cmp_equal ;%
    \ifnum #1>#6 \XINT_cmp_gt\fi
    \ifnum #1<#6 \XINT_cmp_lt\fi
    \xint_gob_til_sc #2\XINT_cmp_equal ;%
    \ifnum #2>#7 \XINT_cmp_gt\fi
    \ifnum #2<#7 \XINT_cmp_lt\fi
    \xint_gob_til_sc #3\XINT_cmp_equal ;%
    \ifnum #3>#8 \XINT_cmp_gt\fi
    \ifnum #3<#8 \XINT_cmp_lt\fi
    \xint_gob_til_sc #4\XINT_cmp_equal ;%
    \ifnum #4>#9 \XINT_cmp_gt\fi
    \ifnum #4<#9 \XINT_cmp_lt\fi
    \XINT_cmp_a #5\W
}%
\def\XINT_cmp_lt#1{\def\XINT_cmp_lt\fi ##1\W ##2\W {\fi#1-1}}\XINT_cmp_lt{ }%
\def\XINT_cmp_gt#1{\def\XINT_cmp_gt\fi ##1\W ##2\W {\fi#11}}\XINT_cmp_gt{ }%
\def\XINT_cmp_equal #1\W #2\W { 0}%
```

4.35 \xintiiSub

Entirely rewritten for 1.2.

Refactored at 1.2l. I was initially aiming at clinching some internal format of the type 1<8digits>!....1<8digits>! for chaining the arithmetic operations (as a preliminary step to deciding upon some internal format for xintfrac macros), thus I wanted to uniformize delimiters in particular and have some core macros inputting and outputting such formats. But the way division is implemented makes it currently very hard to obtain a satisfactory solution. For subtraction I got there almost, but there was added overhead and, as the core sub-routine still assumed the shorter number will be positioned first, one would need to record the length also in the basic internal format, or add the overhead to not make assumption on which one is shorter. I thus but back-tracked my steps but in passing I improved the efficiency (probably) in the worst case branch.

Sadly this 1.2l refactoring left an extra ! in macro \XINT_sub_l_Ida. This bug shows only in rare circumstances which escaped out test suite :(Fixed at 1.2q.

The other reason for backtracking was in relation with the decimal numbers. Having a core format in base 10^8 but ultimately the radix is actually 10 leads to complications. I could use radix 10^8 for \xintiiexpr only, but then I need to make it compatible with sub-\xintiiexpr in \xintexpr, etc... there are many issues of this type.

I considered also an approach like in the 1.2l \xintiiCmp, but decided to stick with the method here for now.

```
\def\xintiiSub   {\romannumeral0\xintiisub }%
\def\xintiisub #1{\expandafter\XINT_iisub\romannumeral`&&@#1\xint:}%
\def\XINT_iisub #1#2\xint:#3%
{%
    \expandafter\XINT_sub_nfork\expandafter
    #1\romannumeral`&&@#3\xint:#2\xint:
}%
\def\XINT_sub_nfork #1#2%
{%
```

```
    \xint_UDzerofork
      #1\XINT_sub_firstiszero
      #2\XINT_sub_secondiszero
       0{}%
    \krof
    \xint_UDsignsfork
         #1#2\XINT_sub_minusminus
          #1-\XINT_sub_minusplus
          #2-\XINT_sub_plusminus
           --\XINT_sub_plusplus
    \krof #1#2%
}%
\def\XINT_sub_firstiszero  #1\krof 0#2#3\xint:#4\xint:{\XINT_opp #2#3}%
\def\XINT_sub_secondiszero #1\krof #20#3\xint:#4\xint:{ #2#4}%
\def\XINT_sub_plusminus    #1#2{\XINT_add_pp_a #1{}}%
\def\XINT_sub_plusplus   #1#2%
   {\expandafter\XINT_opp\romannumeral0\XINT_sub_mm_a #1#2}%
\def\XINT_sub_minusplus    #1#2%
   {\expandafter-\romannumeral0\XINT_add_pp_a {}#2}%
\def\XINT_sub_minusminus #1#2{\XINT_sub_mm_a {}{}}%
\def\XINT_sub_mm_a  #1#2#3\xint:
{%
  \expandafter\XINT_sub_mm_b
      \romannumeral0\expandafter\XINT_sepandrev_andcount
      \romannumeral0\XINT_zeroes_forviii #2#3\R\R\R\R\R\R\R\R{10}0000001\W
      #2#3\XINT_rsepbyviii_end_A 2345678%
          \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
              \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
              \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
  \X #1%
}%
\def\XINT_sub_mm_b #1\xint:#2\X #3\xint:
{%
    \expandafter\XINT_sub_checklengths
    \the\numexpr #1\expandafter\xint:%
    \romannumeral0\expandafter\XINT_sepandrev_andcount
    \romannumeral0\XINT_zeroes_forviii #3\R\R\R\R\R\R\R\R{10}0000001\W
    #3\XINT_rsepbyviii_end_A 2345678%
      \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
              \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
              \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
      1;!1;!1;!1;!\W
    #21;!1;!1;!1;!\W
    1;!1\R!1\R!1\R!1\R!%
    1\R!1\R!1\R!1\R!\W
}%
\def\XINT_sub_checklengths #1\xint:#2\xint:%
{%
    \ifnum #2>#1
       \expandafter\XINT_sub_exchange
    \else
       \expandafter\XINT_sub_aa
```

```
    \fi
}%
\def\XINT_sub_exchange #1\W #2\W
{%
    \expandafter\XINT_opp\romannumeral0\XINT_sub_aa #2\W #1\W
}%
\def\XINT_sub_aa
{%
    \expandafter\XINT_sub_out\the\numexpr\XINT_sub_a\xint_c_i
}%
```

The post-processing (clean-up of zeros, or rescue of situation with A-B where actually B turns out bigger than A) will be done by a macro which depends on circumstances and will be initially last token before the reversion done by \XINT_unrevbyviii.

```
\def\XINT_sub_out {\XINT_unrevbyviii{}}%
```

1 as first token of #1 stands for "no carry", 0 will mean a carry.
Call: \the\numexpr
\XINT_sub_a 1#11;!1;!1;!1;!\W
#21;!1;!1;!1;!\W
where #1 and #2 are blocks of 1<8d>!, #1 (=B) *must* be at most as long as #2 (=A), (in radix 10^8) and the routine wants to compute #2-#1 = A - B
1.2l uses 1;! delimiters to match those of addition (and multiplication). But in the end I reverted the code branch which made it possible to chain such operations keeping internal format in 8 digits blocks throughout.
\numexpr governed expansion stops with various possibilities:
- Type Ia: #1 shorter than #2, no final carry
- Type Ib: #1 shorter than #2, a final carry but next block of #2 > 1
- Type Ica: #1 shorter than #2, a final carry, next block of #2 is final and = 1
- Type Icb: as Ica except that 00000001 block from #2 was not final
- Type Id: #1 shorter than #2, a final carry, next block of #2 = 0
- Type IIa: #1 same length as #2, turns out it was <= #2.
- Type IIb: #1 same length as #2, but turned out > #2.
Various type of post actions are then needed:
- Ia: clean up of zeros in most significant block of 8 digits
- Ib: as Ia
- Ic: there may be significant blocks of 8 zeros to clean up from result. Only case Ica may have arbitrarily many of them, case Icb has only one such block.
- Id: blocks of 99999999 may propagate and there might a be final zero block created which has to be cleaned up.
- IIa: arbitrarily many zeros might have to be removed.
- IIb: We wanted #2-#1 = - (#1-#2), but we got 10^{8N}+#2 -#1 = 10^{8N}-(#1-#2). We need to do the correction then we are as in IIa situation, except that final result can not be zero.
The 1.2l method for this correction is (presumably, testing takes lots of time, which I do not have) more efficient than in 1.2 release.

```
\def\XINT_sub_a #1!#2!#3!#4!#5\W #6!#7!#8!#9!%
{%
    \XINT_sub_b
    #1!#6!#2!#7!#3!#8!#4!#9!%
    #5\W
}%
```

As 1.2l code uses 1<8digits>! blocks one has to be careful with the carry digit 1 or 0: A #11#2#3 pattern would result into an empty #1 if the carry digit which is upfront is 1, rather than setting #1=1.

```
\def\XINT_sub_b #1#2#3#4!#5!%
{%
    \xint_gob_til_sc #3\XINT_sub_bi ;%
    \expandafter\XINT_sub_c\the\numexpr#1+1#5-#3#4-\xint_c_i\xint:%
}%
\def\XINT_sub_c 1#1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_sub_d #1%
}%
\def\XINT_sub_d #1#2#3#4!#5!%
{%
    \xint_gob_til_sc #3\XINT_sub_di ;%
    \expandafter\XINT_sub_e\the\numexpr#1+1#5-#3#4-\xint_c_i\xint:
}%
\def\XINT_sub_e 1#1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_sub_f #1%
}%
\def\XINT_sub_f #1#2#3#4!#5!%
{%
    \xint_gob_til_sc #3\XINT_sub_fi ;%
    \expandafter\XINT_sub_g\the\numexpr#1+1#5-#3#4-\xint_c_i\xint:
}%
\def\XINT_sub_g 1#1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_sub_h #1%
}%
\def\XINT_sub_h #1#2#3#4!#5!%
{%
    \xint_gob_til_sc #3\XINT_sub_hi ;%
    \expandafter\XINT_sub_i\the\numexpr#1+1#5-#3#4-\xint_c_i\xint:
}%
\def\XINT_sub_i 1#1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_sub_a #1%
}%
\def\XINT_sub_bi;%
    \expandafter\XINT_sub_c\the\numexpr#1+1#2-#3\xint:
    #4!#5!#6!#7!#8!#9!\W
{%
    \XINT_sub_k #1#2!#5!#7!#9!%
}%
\def\XINT_sub_di;%
    \expandafter\XINT_sub_e\the\numexpr#1+1#2-#3\xint:
    #4!#5!#6!#7!#8\W
{%
    \XINT_sub_k #1#2!#5!#7!%
}%
\def\XINT_sub_fi;%
```

```
    \expandafter\XINT_sub_g\the\numexpr#1+1#2-#3\xint:
    #4!#5!#6\W
{%
    \XINT_sub_k #1#2!#5!%
}%
\def\XINT_sub_hi;%
    \expandafter\XINT_sub_i\the\numexpr#1+1#2-#3\xint:
    #4\W
{%
    \XINT_sub_k #1#2!%
}%
```

B terminated. Have we reached the end of A (necessarily at least as long as B) ? (we are computing A-B, digits of B come first).

If not, then we are certain that even if there is carry it will not propagate beyond the end of A. But it may propagate far transforming chains of 00000000 into 99999999, and if it does go to the final block which possibly is just 1<00000001>!, we will have those eight zeros to clean up.

If A and B have the same length (in base 10^8) then arbitrarily many zeros might have to be cleaned up, and if A<B, the whole result will have to be complemented first.

```
\def\XINT_sub_k #1#2#3%
{%
    \xint_gob_til_sc #3\XINT_sub_p;\XINT_sub_l #1#2#3%
}%
\def\XINT_sub_l #1%
   {\xint_UDzerofork #1\XINT_sub_l_carry 0\XINT_sub_l_Ia\krof}%
\def\XINT_sub_l_Ia 1#1;!#2\W{1\relax#1;!1\XINT_sub_fix_none!}%


\def\XINT_sub_l_carry 1#1!{\ifcase #1
         \expandafter \XINT_sub_l_Id
    \or  \expandafter \XINT_sub_l_Ic
    \else\expandafter \XINT_sub_l_Ib\fi 1#1!}%
\def\XINT_sub_l_Ib #1;#2\W {-\xint_c_i+#1;!1\XINT_sub_fix_none!}%
\def\XINT_sub_l_Ic 1#1!1#2#3!#4;#5\W
{%
    \xint_gob_til_sc #2\XINT_sub_l_Ica;%
    1\relax 00000000!1#2#3!#4;!1\XINT_sub_fix_none!%
}%
```

We need to add some extra delimiters at the end for post-action by \XINT_num, so we first grab the material up to \W

```
\def\XINT_sub_l_Ica#1\W
{%
    1;!1\XINT_sub_fix_cuz!%
    1;!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
    \xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
}%
\def\XINT_sub_l_Id 1#1!%
    {199999999\expandafter!\the\numexpr \XINT_sub_l_Id_a}%
\def\XINT_sub_l_Id_a 1#1!{\ifcase #1
         \expandafter \XINT_sub_l_Id
    \or  \expandafter \XINT_sub_l_Id_b
```

```
    \else\expandafter \XINT_sub_l_Ib\fi 1#1!}%
\def\XINT_sub_l_Id_b 1#1!1#2#3!#4;#5\W
{%
    \xint_gob_til_sc #2\XINT_sub_l_Ida;%
    1\relax 00000000!1#2#3!#4;!1\XINT_sub_fix_none!%
}%
\def\XINT_sub_l_Ida#1\XINT_sub_fix_none{1;!1\XINT_sub_fix_none}%
```

This is the case where both operands have same 10^8-base length.
We were handling A-B but perhaps B>A. The situation with A=B is also annoying because we then have to clean up all zeros but don't know where to stop (if A>B the first non-zero 8 digits block would tell use when).
Here again we need to grab #3\W to position the actually used terminating delimiters.

```
\def\XINT_sub_p;\XINT_sub_l #1#2\W #3\W
{%
    \xint_UDzerofork
       #1{1;!1\XINT_sub_fix_neg!%
          1;!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
          \xint_bye2345678\xint_bye1099999988\relax}% A - B, B > A
        0{1;!1\XINT_sub_fix_cuz!%
          1;!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W}%
    \krof
    \xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
}%
```

Routines for post-processing after reversal, and removal of separators. It is a matter of cleaning up zeros, and possibly in the bad case to take a complement before that.

```
\def\XINT_sub_fix_none;{\XINT_cuz_small}%
\def\XINT_sub_fix_cuz ;{\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop}%
```

Case with A and B same number of digits in base 10^8 and B>A.
1.2l subtle chaining on the model of the 1.2i rewrite of \xintInc and similar routines. After taking complement, leading zeroes need to be cleaned up as in B<=A branch.

```
\def\XINT_sub_fix_neg;%
{%
    \expandafter-\romannumeral0\expandafter
    \XINT_sub_comp_finish\the\numexpr\XINT_sub_comp_loop
}%
\def\XINT_sub_comp_finish 0{\XINT_sub_fix_cuz;}%
\def\XINT_sub_comp_loop #1#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_sub_comp_clean
    \the\numexpr \xint_c_xi_e_viii_mone-#1#2#3#4#5#6#7#8\XINT_sub_comp_loop
}%
```

#1 = 0 signifie une retenue, #1 = 1 pas de retenue, ce qui ne peut arriver que tant qu'il n'y a que des zéros du côté non significatif. Lorsqu'on est revenu au début on a forcément une retenue.

```
\def\XINT_sub_comp_clean 1#1{+#1\relax}%
```

4.36 \xintiiMul

Completely rewritten for 1.2.
1.2l: \xintiiMul made robust against non terminated input.

```
\def\xintiiMul {\romannumeral0\xintiimul }%
\def\xintiimul #1%
{%
    \expandafter\XINT_iimul\romannumeral`&&@#1\xint:
}%
\def\XINT_iimul #1#2\xint:#3%
{%
    \expandafter\XINT_mul_nfork\expandafter #1\romannumeral`&&@#3\xint:#2\xint:
}%
```

1.2 I have changed the fork, and it complicates matters elsewhere.
ATTENTION for example that 1.4 \xintiiPrd uses \XINT_mul_nfork now.

```
\def\XINT_mul_fork #1#2\xint:#3\xint:{\XINT_mul_nfork #1#3\xint:#2\xint:}%
\def\XINT_mul_nfork #1#2%
{%
    \xint_UDzerofork
      #1\XINT_mul_zero
      #2\XINT_mul_zero
       0{}%
    \krof
    \xint_UDsignsfork
         #1#2\XINT_mul_minusminus
          #1-\XINT_mul_minusplus
          #2-\XINT_mul_plusminus
           --\XINT_mul_plusplus
    \krof #1#2%
}%
\def\XINT_mul_zero  #1\krof #2#3\xint:#4\xint:{ 0}%
\def\XINT_mul_minusminus   #1#2{\XINT_mul_plusplus {}{}}%
\def\XINT_mul_minusplus    #1#2%
   {\expandafter-\romannumeral0\XINT_mul_plusplus {}#2}%
\def\XINT_mul_plusminus    #1#2%
   {\expandafter-\romannumeral0\XINT_mul_plusplus #1{}}%
\def\XINT_mul_plusplus #1#2#3\xint:
{%
  \expandafter\XINT_mul_pre_b
      \romannumeral0\expandafter\XINT_sepandrev_andcount
      \romannumeral0\XINT_zeroes_forviii #2#3\R\R\R\R\R\R\R\R{10}0000001\W
      #2#3\XINT_rsepbyviii_end_A 2345678%
          \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
              \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
              \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
  \W #1%
}%
\def\XINT_mul_pre_b #1\xint:#2\W #3\xint:
{%
    \expandafter\XINT_mul_checklengths
    \the\numexpr #1\expandafter\xint:%
```

```
    \romannumeral0\expandafter\XINT_sepandrev_andcount
    \romannumeral0\XINT_zeroes_forviii #3\R\R\R\R\R\R\R{10}0000001\W
    #3\XINT_rsepbyviii_end_A 2345678%
      \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
              \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
              \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
     1;!\W #21;!%
    1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
}%
```

Cooking recipe, 2015/10/05.

```
\def\XINT_mul_checklengths #1\xint:#2\xint:%
{%
    \ifnum #2=\xint_c_i\expandafter\XINT_mul_smallbyfirst\fi
    \ifnum #1=\xint_c_i\expandafter\XINT_mul_smallbysecond\fi
    \ifnum #2<#1
       \ifnum \numexpr (#2-\xint_c_i)*(#1-#2)<383
          \XINT_mul_exchange
       \fi
    \else
       \ifnum \numexpr (#1-\xint_c_i)*(#2-#1)>383
          \XINT_mul_exchange
       \fi
    \fi
    \XINT_mul_start
}%
\def\XINT_mul_smallbyfirst #1\XINT_mul_start 1#2!1;!\W
{%
    \ifnum#2=\xint_c_i\expandafter\XINT_mul_oneisone\fi
    \ifnum#2<\xint_c_xxii\expandafter\XINT_mul_verysmall\fi
    \expandafter\XINT_mul_out\the\numexpr\XINT_smallmul 1#2!%
}%
\def\XINT_mul_smallbysecond #1\XINT_mul_start #2\W 1#3!1;!%
{%
    \ifnum#3=\xint_c_i\expandafter\XINT_mul_oneisone\fi
    \ifnum#3<\xint_c_xxii\expandafter\XINT_mul_verysmall\fi
    \expandafter\XINT_mul_out\the\numexpr\XINT_smallmul 1#3!#2%
}%
\def\XINT_mul_oneisone #1!{\XINT_mul_out }%
\def\XINT_mul_verysmall\expandafter\XINT_mul_out
                       \the\numexpr\XINT_smallmul 1#1!%
    {\expandafter\XINT_mul_out\the\numexpr\XINT_verysmallmul 0\xint:#1!}%
\def\XINT_mul_exchange #1\XINT_mul_start #2\W #31;!%
   {\fi\fi\XINT_mul_start #31;!\W #2}%


\def\XINT_mul_start
   {\expandafter\XINT_mul_out\the\numexpr\XINT_mul_loop 100000000!1;!\W}%
\def\XINT_mul_out
   {\expandafter\XINT_cuz_small\romannumeral0\XINT_unrevbyviii {}}%
```

Call:
`\the\numexpr \XINT_mul_loop 100000000!1;!\W #11;!\W #21;!`

```
    where #1 and #2 are (globally reversed) blocks 1<8d>!. Its is generally more efficient if #1
    is the shorter one, but a better recipe is implemented in \XINT_mul_checklengths. One may call
    \XINT_mul_loop directly (but multiplication by zero will produce many 100000000! blocks on out-
    put).
      Ends after having produced: 1<8d>!....1<8d>!1;!. The last 8-digits block is significant one.
    It can not be 100000000! except if the loop was called with a zero operand.
      Thus \XINT_mul_loop can be conveniently called directly in recursive routines, as the output
    terminator can serve as input terminator, we can arrange to not have to grab the whole thing again.

1073 \def\XINT_mul_loop #1\W #2\W 1#3!%
1074 {%
1075     \xint_gob_til_sc #3\XINT_mul_e ;%
1076     \expandafter\XINT_mul_a\the\numexpr \XINT_smallmul 1#3!#2\W
1077     #1\W #2\W
1078 }%

    Each of #1 and #2 brings its 1;! for \XINT_add_a.

1079 \def\XINT_mul_a #1\W #2\W
1080 {%
1081     \expandafter\XINT_mul_b\the\numexpr
1082     \XINT_add_a \xint_c_ii #21;!1;!1;!\W #11;!1;!1;!\W\W
1083 }%
1084 \def\XINT_mul_b 1#1!{1#1\expandafter!\the\numexpr\XINT_mul_loop }%
1085 \def\XINT_mul_e;#1\W 1#2\W #3\W {1\relax #2}%

    1.2 small and mini multiplication in base 10^8 with carry. Used by the main multiplication rou-
    tines. But division, float factorial, etc.. have their own variants as they need output with spe-
    cific constraints.
      The minimulwc has 1<8digits carry>.<4 high digits>.<4 low digits!<8digits>.
      It produces a block 1<8d>! and then jump back into \XINT_smallmul_a with the new 8digits
    carry as argument. The \XINT_smallmul_a fetches a new 1<8d>! block to multiply, and calls back
    \XINT_minimul_wc having stored the multiplicand for re-use later. When the loop terminates, the
    final carry is checked for being nul, and in all cases the output is terminated by a 1;!
      Multiplication by zero will produce blocks of zeros.

1086 \def\XINT_minimulwc_a 1#1\xint:#2\xint:#3!#4#5#6#7#8\xint:%
1087 {%
1088     \expandafter\XINT_minimulwc_b
1089     \the\numexpr \xint_c_x^ix+#1+#3*#8\xint:
1090                  #3*#4#5#6#7+#2*#8\xint:
1091                        #2*#4#5#6#7\xint:%
1092 }%
1093 \def\XINT_minimulwc_b 1#1#2#3#4#5#6\xint:#7\xint:%
1094 {%
1095     \expandafter\XINT_minimulwc_c
1096     \the\numexpr \xint_c_x^ix+#1#2#3#4#5+#7\xint:#6\xint:%
1097 }%
1098 \def\XINT_minimulwc_c 1#1#2#3#4#5#6\xint:#7\xint:#8\xint:%
1099 {%
1100     1#6#7\expandafter!%
1101     \the\numexpr\expandafter\XINT_smallmul_a
1102     \the\numexpr \xint_c_x^viii+#1#2#3#4#5+#8\xint:%
1103 }%
```

```
\def\XINT_smallmul 1#1#2#3#4#5!{\XINT_smallmul_a 100000000\xint:#1#2#3#4\xint:#5!}%
\def\XINT_smallmul_a #1\xint:#2\xint:#3!1#4!%
{%
    \xint_gob_til_sc #4\XINT_smallmul_e;%
    \XINT_minimulwc_a #1\xint:#2\xint:#3!#4\xint:#2\xint:#3!%
}%
\def\XINT_smallmul_e;\XINT_minimulwc_a 1#1\xint:#2;#3!%
    {\xint_gob_til_eightzeroes #1\XINT_smallmul_f 000000001\relax #1!1;!}%
\def\XINT_smallmul_f 000000001\relax 00000000!1{1\relax}%


\def\XINT_verysmallmul #1\xint:#2!1#3!%
{%
    \xint_gob_til_sc #3\XINT_verysmallmul_e;%
    \expandafter\XINT_verysmallmul_a
    \the\numexpr #2*#3+#1\xint:#2!%
}%
\def\XINT_verysmallmul_e;\expandafter\XINT_verysmallmul_a\the\numexpr
    #1+#2#3\xint:#4!%
{\xint_gob_til_zero #2\XINT_verysmallmul_f 0\xint_c_x^viii+#2#3!1;!}%
\def\XINT_verysmallmul_f #1!1{1\relax}%
\def\XINT_verysmallmul_a #1#2\xint:%
{%
    \unless\ifnum #1#2<\xint_c_x^ix
    \expandafter\XINT_verysmallmul_bi\else
    \expandafter\XINT_verysmallmul_bj\fi
    \the\numexpr \xint_c_x^ix+#1#2\xint:%
}%
\def\XINT_verysmallmul_bj{\expandafter\XINT_verysmallmul_cj }%
\def\XINT_verysmallmul_cj 1#1#2\xint:%
    {1#2\expandafter!\the\numexpr\XINT_verysmallmul #1\xint:}%
\def\XINT_verysmallmul_bi\the\numexpr\xint_c_x^ix+#1#2#3\xint:%
    {1#3\expandafter!\the\numexpr\XINT_verysmallmul #1#2\xint:}%
```

Used by division and by squaring, not by multiplication itself.
This routine does not loop, it only does one mini multiplication with input format <4 high digits>.<4 low digits>!<8 digits>!, and on output 1<8d>!1<8d>!, with least significant block first.

```
\def\XINT_minimul_a #1\xint:#2!#3#4#5#6#7!%
{%
    \expandafter\XINT_minimul_b
    \the\numexpr \xint_c_x^viii+#2*#7\xint:#2*#3#4#5#6+#1*#7\xint:#1*#3#4#5#6\xint:%
}%
\def\XINT_minimul_b 1#1#2#3#4#5\xint:#6\xint:%
{%
    \expandafter\XINT_minimul_c
    \the\numexpr \xint_c_x^ix+#1#2#3#4+#6\xint:#5\xint:%
}%
\def\XINT_minimul_c 1#1#2#3#4#5#6\xint:#7\xint:#8\xint:%
{%
    1#6#7\expandafter!\the\numexpr \xint_c_x^viii+#1#2#3#4#5+#8!%
}%
```

4.37 \xintiiDivision

Completely rewritten for 1.2.

WARNING: some comments below try to describe the flow of tokens but they date back to xint 1.09j and I updated them on the fly while doing the 1.2 version. As the routine now works in base 10^8, not 10^4 and "drops" the quotient digits,rather than store them upfront as the earlier code, I may well have not correctly converted all such comments. At the last minute some previously #1 became stuff like #1#2#3#4, then of course the old comments describing what the macro parameters stand for are necessarily wrong.

Side remark: the way tokens are grouped was not essentially modified in 1.2, although the situation has changed. It was fine-tuned in xint 1.0/1.1 but the context has changed, and perhaps I should revisit this. As a corollary to the fact that quotient digits are now left behind thanks to the chains of \numexpr, some macros which in 1.0/1.1 fetched up to 9 parameters now need handle less such parameters. Thus, some rationale for the way the code was structured has disappeared.

1.2l: \xintiiDivision et al. made robust against non terminated input.

#1 = A, #2 = B. On calcule le quotient et le reste dans la division euclidienne de A par B: A=BQ+R, 0<= R < |B|.

```
\def\xintiiDivision   {\romannumeral0\xintiidivision }%
\def\xintiidivision  #1{\expandafter\XINT_iidivision \romannumeral`&&@#1\xint:}%
\def\XINT_iidivision #1#2\xint:#3{\expandafter\XINT_iidivision_a\expandafter #1%
                                 \romannumeral`&&@#3\xint:#2\xint:}%
```

On regarde les signes de A et de B.

```
\def\XINT_iidivision_a #1#2% #1 de A, #2 de B.
{%
    \if0#2\xint_dothis{\XINT_iidivision_divbyzero #1#2}\fi
    \if0#1\xint_dothis\XINT_iidivision_aiszero\fi
    \if-#2\xint_dothis{\expandafter\XINT_iidivision_bneg
                          \romannumeral0\XINT_iidivision_bpos #1}\fi
    \xint_orthat{\XINT_iidivision_bpos #1#2}%
}%
\def\XINT_iidivision_divbyzero#1#2#3\xint:#4\xint:
   {\if0#1\xint_dothis{\XINT_signalcondition{DivisionUndefined}}\fi
          \xint_orthat{\XINT_signalcondition{DivisionByZero}}%
    {Division of #1#4 by #2#3}{}{{0}{0}}}%
\def\XINT_iidivision_aiszero #1\xint:#2\xint:{{0}{0}}%
\def\XINT_iidivision_bneg #1% q->-q, r unchanged
                          {\expandafter{\romannumeral0\XINT_opp #1}}%
\def\XINT_iidivision_bpos #1%
{%
    \xint_UDsignfork
            #1\XINT_iidivision_aneg
             -{\XINT_iidivision_apos #1}%
    \krof
}%
```

Donc attention malgré son nom \XINT_div_prepare va jusqu'au bout. C'est donc en fait l'entrée principale (pour B>0, A>0) mais elle va regarder si B est < 10^8 et s'il vaut alors 1 ou 2, et si A < 10^8. Dans tous les cas le résultat est produit sous la forme {Q}{R}, avec Q et R sous leur forme final. On doit ensuite ajuster si le B ou le A initial était négatif. Je n'ai pas fait beaucoup d'efforts pour être un minimum efficace si A ou B n'est pas positif.

```
\def\XINT_iidivision_apos #1#2\xint:#3\xint:{\XINT_div_prepare {#2}{#1#3}}%
\def\XINT_iidivision_aneg #1\xint:#2\xint:
   {\expandafter
    \XINT_iidivision_aneg_b\romannumeral0\XINT_div_prepare {#1}{#2}{#1}}%
\def\XINT_iidivision_aneg_b #1#2{\if0\XINT_Sgn #2\xint:
                                   \expandafter\XINT_iidivision_aneg_rzero
                               \else
                                   \expandafter\XINT_iidivision_aneg_rpos
                               \fi {#1}{#2}}%
\def\XINT_iidivision_aneg_rzero #1#2#3{{-#1}{0}}% necessarily q was >0
\def\XINT_iidivision_aneg_rpos #1%
{%
    \expandafter\XINT_iidivision_aneg_end\expandafter
                {\expandafter-\romannumeral0\xintinc {#1}}% q-> -(1+q)
}%
\def\XINT_iidivision_aneg_end #1#2#3%
{%
     \expandafter\xint_exchangetwo_keepbraces
     \expandafter{\romannumeral0\XINT_sub_mm_a {}{}#3\xint:#2\xint:}{#1}% r-> b-r
}%
```

Le diviseur B va être étendu par des zéros pour que sa longueur soit multiple de huit. Les zéros seront mis du côté non significatif.

```
\def\XINT_div_prepare #1%
{%
    \XINT_div_prepare_a #1\R\R\R\R\R\R\R\R {10}0000001\W !{#1}%
}%
\def\XINT_div_prepare_a #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_R #9\XINT_div_prepare_small\R
    \XINT_div_prepare_b #9%
}%
```

B a au plus huit chiffres. On se débarrasse des trucs superflus. Si B>0 n'est ni 1 ni 2, le point d'entrée est \XINT_div_small_a {B}{A} (avec un A positif).

```
\def\XINT_div_prepare_small\R #1!#2%
{%
    \ifcase #2
    \or\expandafter\XINT_div_BisOne
    \or\expandafter\XINT_div_BisTwo
    \else\expandafter\XINT_div_small_a
    \fi {#2}%
}%
\def\XINT_div_BisOne #1#2{{#2}{0}}%
\def\XINT_div_BisTwo #1#2%
{%
    \expandafter\expandafter\expandafter\XINT_div_BisTwo_a
    \ifodd\xintLDg{#2} \expandafter1\else \expandafter0\fi {#2}%
}%
\def\XINT_div_BisTwo_a #1#2%
{%
    \expandafter{\romannumeral0\XINT_half
```

```
      #2\xint_bye\xint_Bye345678\xint_bye
      *\xint_c_v+\xint_c_v)/\xint_c_x-\xint_c_i\relax}{#1}%
}%
```

B a au plus huit chiffres et est au moins 3. On va l'utiliser directement, sans d'abord le multiplier par une puissance de 10 pour qu'il ait 8 chiffres.

```
\def\XINT_div_small_a #1#2%
{%
    \expandafter\XINT_div_small_b
    \the\numexpr #1/\xint_c_ii\expandafter
    \xint:\the\numexpr \xint_c_x^viii+#1\expandafter!%
    \romannumeral0%
    \XINT_div_small_ba #2\R\R\R\R\R\R\R{10}0000001\W
       #2\XINT_sepbyviii_Z_end 2345678\relax
}%
```

Le #2 poursuivra l'expansion par \XINT_div_dosmallsmall ou par \XINT_smalldivx_a suivi de \XINT_sdiv_out.

```
\def\XINT_div_small_b #1!#2{#2#1!}%
```

On ajoute des zéros avant A, puis on le prépare sous la forme de blocs 1<8d>! Au passage on repère le cas d'un A<10^8.

```
\def\XINT_div_small_ba #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_R #9\XINT_div_smallsmall\R
    \expandafter\XINT_div_dosmalldiv
    \the\numexpr\expandafter\XINT_sepbyviii_Z
           \romannumeral0\XINT_zeroes_forviii
    #1#2#3#4#5#6#7#8#9%
}%
```

Si A<10^8, on va poursuivre par \XINT_div_dosmallsmall round(B/2).10^8+B!{A}. On fait la division directe par \numexpr. Le résultat est produit sous la forme {Q}{R}.

```
\def\XINT_div_smallsmall\R
    \expandafter\XINT_div_dosmalldiv
    \the\numexpr\expandafter\XINT_sepbyviii_Z
    \romannumeral0\XINT_zeroes_forviii #1\R #2\relax
   {{\XINT_div_dosmallsmall}{#1}}%
\def\XINT_div_dosmallsmall #1\xint:1#2!#3%
{%
    \expandafter\XINT_div_smallsmallend
    \the\numexpr (#3+#1)/#2-\xint_c_i\xint:#2\xint:#3\xint:%
}%
\def\XINT_div_smallsmallend #1\xint:#2\xint:#3\xint:{\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #3-#1*#2}}%
```

Si A>=10^8, il est maintenant sous la forme 1<8d>!...1<8d>!1;! avec plus significatifs en premier. Donc on poursuit par \expandafter\XINT_sdiv_out\the\numexpr\XINT_smalldivx_a x.1B!1<8d>!...1<8d>!1;! avec x =round(B/2), 1B=10^8+B.

```
\def\XINT_div_dosmalldiv
    {{\expandafter\XINT_sdiv_out\the\numexpr\XINT_smalldivx_a}}%
```

```
    Ici B est au moins 10^8, on détermine combien de zéros lui adjoindre pour qu'il soit de longueur
  8N.

\def\XINT_div_prepare_b
   {\expandafter\XINT_div_prepare_c\romannumeral0\XINT_zeroes_forviii }%
\def\XINT_div_prepare_c #1!%
{%
     \XINT_div_prepare_d  #1.00000000!{#1}%
}%
\def\XINT_div_prepare_d #1#2#3#4#5#6#7#8#9%
{%
    \expandafter\XINT_div_prepare_e\xint_gob_til_dot #1#2#3#4#5#6#7#8#9!%
}%
\def\XINT_div_prepare_e #1!#2!#3#4%
{%
    \XINT_div_prepare_f #4#3\X {#1}{#3}%
}%
```

```
    attention qu'on calcule ici x'=x+1 (x = huit premiers chiffres du diviseur) et que si x=99999999,
  x' aura donc 9 chiffres, pas compatible avec div_mini (avant 1.2, x avait 4 chiffres, et on faisait
  la division avec x' dans un \numexpr). Bon, facile à dire après avoir laissé passer ce bug dans 1.2.
  C'est le problème lorsqu'au lieu de tout refaire à partir de zéro on recycle d'anciennes routines
  qui avaient un contexte différent.

\def\XINT_div_prepare_f #1#2#3#4#5#6#7#8#9\X
{%
    \expandafter\XINT_div_prepare_g
     \the\numexpr  #1#2#3#4#5#6#7#8+\xint_c_i\expandafter
    \xint:\the\numexpr (#1#2#3#4#5#6#7#8+\xint_c_i)/\xint_c_ii\expandafter
    \xint:\the\numexpr #1#2#3#4#5#6#7#8\expandafter
    \xint:\romannumeral0\XINT_sepandrev_andcount
    #1#2#3#4#5#6#7#8#9\XINT_rsepbyviii_end_A 2345678%
                       \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
              \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
              \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
    \X
}%
\def\XINT_div_prepare_g #1\xint:#2\xint:#3\xint:#4\xint:#5\X #6#7#8%
{%
    \expandafter\XINT_div_prepare_h
    \the\numexpr\expandafter\XINT_sepbyviii_andcount
    \romannumeral0\XINT_zeroes_forviii #8#7\R\R\R\R\R\R\R{10}0000001\W
    #8#7\XINT_sepbyviii_end 2345678\relax
     \xint_c_vii!\xint_c_vi!\xint_c_v!\xint_c_iv!%
     \xint_c_iii!\xint_c_ii!\xint_c_i!\xint_c_\W
    {#1}{#2}{#3}{#4}{#5}{#6}%
}%
\def\XINT_div_prepare_h #11\xint:#2\xint:#3#4#5#6%#7#8%
{%
    \XINT_div_start_a {#2}{#6}{#1}{#3}{#4}{#5}%{#7}{#8}%
}%
```

L, K, A, x',y,x, B, «c». Attention que K est diminué de 1 plus loin. Comme xint 1.2 a déjà repéré K=1, on a ici au minimum K=2. Attention B est à l'envers, A est à l'endroit et les deux avec séparateurs. Attention que ce n'est pas ici qu'on boucle mais en \XINT_div_I_a.

```
\def\XINT_div_start_a #1#2%
{%
    \ifnum #1 < #2
      \expandafter\XINT_div_zeroQ
    \else
      \expandafter\XINT_div_start_b
    \fi
    {#1}{#2}%
}%
\def\XINT_div_zeroQ #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_div_zeroQ_end
    \romannumeral0\XINT_unsep_cuzsmall
    #3\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax\xint:
}%
\def\XINT_div_zeroQ_end #1\xint:#2%
    {\expandafter{\expandafter0\expandafter}\XINT_div_cleanR #1#2\xint:}%
```

L, K, A, x',y,x, B, «c»->K.A.x{LK{x'y}x}B«c»

```
\def\XINT_div_start_b #1#2#3#4#5#6%
{%
    \expandafter\XINT_div_finish\the\numexpr
    \XINT_div_start_c {#2}\xint:#3\xint:{#6}{{#1}{#2}{{#4}{#5}}{#6}}%
}%
\def\XINT_div_finish
{%
    \expandafter\XINT_div_finish_a \romannumeral`&&@\XINT_div_unsepQ
}%
\def\XINT_div_finish_a #1\Z #2\xint:{\XINT_div_finish_b #2\xint:{#1}}%
```

Ici ce sont routines de fin. Le reste déjà nettoyé. R.Q«c».

```
\def\XINT_div_finish_b #1%
{%
    \if0#1%
       \expandafter\XINT_div_finish_bRzero
    \else
       \expandafter\XINT_div_finish_bRpos
    \fi
    #1%
}%
\def\XINT_div_finish_bRzero 0\xint:#1#2{{#1}{0}}%
\def\XINT_div_finish_bRpos #1\xint:#2#3%
{%
    \expandafter\xint_exchangetwo_keepbraces\XINT_div_cleanR  #1#3\xint:{#2}%
}%
\def\XINT_div_cleanR #100000000\xint:{{#1}}%
```

Kalpha.A.x{LK{x'y}x}, B, «c», au début #2=alpha est vide. On fait une boucle pour prendre K unités de A (on a au moins L égal à K) et les mettre dans alpha.

```
\def\XINT_div_start_c #1%
{%
    \ifnum #1>\xint_c_vi
       \expandafter\XINT_div_start_ca
    \else
       \expandafter\XINT_div_start_cb
    \fi {#1}%
}%
\def\XINT_div_start_ca #1#2\xint:#3!#4!#5!#6!#7!#8!#9!%
{%
    \expandafter\XINT_div_start_c\expandafter
    {\the\numexpr #1-\xint_c_vii}#2#3!#4!#5!#6!#7!#8!#9!\xint:%
}%
\def\XINT_div_start_cb #1%
   {\csname XINT_div_start_c_\romannumeral\numexpr#1\endcsname}%
\def\XINT_div_start_c_i   #1\xint:#2!%
    {\XINT_div_start_c_   #1#2!\xint:}%
\def\XINT_div_start_c_ii  #1\xint:#2!#3!%
    {\XINT_div_start_c_   #1#2!#3!\xint:}%
\def\XINT_div_start_c_iii #1\xint:#2!#3!#4!%
    {\XINT_div_start_c_   #1#2!#3!#4!\xint:}%
\def\XINT_div_start_c_iv  #1\xint:#2!#3!#4!#5!%
    {\XINT_div_start_c_   #1#2!#3!#4!#5!\xint:}%
\def\XINT_div_start_c_v   #1\xint:#2!#3!#4!#5!#6!%
    {\XINT_div_start_c_   #1#2!#3!#4!#5!#6!\xint:}%
\def\XINT_div_start_c_vi  #1\xint:#2!#3!#4!#5!#6!#7!%
    {\XINT_div_start_c_   #1#2!#3!#4!#5!#6!#7!\xint:}%
```

```
  #1=a, #2=alpha (de longueur K, à l'endroit).#3=reste de A.#4=x, #5={LK{x'y}x},#6=B,«c» -> a,
x, alpha, B, {00000000}, L, K, {x'y},x, alpha'=reste de A, B«c».
```

```
\def\XINT_div_start_c_ 1#1!#2\xint:#3\xint:#4#5#6%
{%
    \XINT_div_I_a {#1}{#4}{1#1!#2}{#6}{00000000}#5{#3}{#6}%
}%
```

```
  Ceci est le point de retour de la boucle principale. a, x, alpha, B, q0, L, K, {x'y}, x, alpha',
B«c»
```

```
\def\XINT_div_I_a #1#2%
{%
    \expandafter\XINT_div_I_b\the\numexpr #1/#2\xint:{#1}{#2}%
}%
\def\XINT_div_I_b #1%
{%
    \xint_gob_til_zero #1\XINT_div_I_czero 0\XINT_div_I_c #1%
}%
```

```
  On intercepte petit quotient nul: #1=a, x, alpha, B, #5=q0, L, K,  {x'y}, x, alpha', B«c» -> on
lâche un q puis {alpha} L, K, {x'y}, x,  alpha', B«c».
```

```
\def\XINT_div_I_czero 0\XINT_div_I_c 0\xint:#1#2#3#4#5{1#5\XINT_div_I_g {#3}}%
\def\XINT_div_I_c #1\xint:#2#3%
{%
```

```
1381     \expandafter\XINT_div_I_da\the\numexpr #2-#1*#3\xint:#1\xint:{#2}{#3}%
1382 }%
```

r.q.alpha, B, q0, L, K, {x'y}, x, alpha', B«c»

```
1383 \def\XINT_div_I_da #1\xint:%
1384 {%
1385     \ifnum #1>\xint_c_ix
1386        \expandafter\XINT_div_I_dP
1387     \else
1388        \ifnum #1<\xint_c_
1389         \expandafter\expandafter\expandafter\XINT_div_I_dN
1390        \else
1391         \expandafter\expandafter\expandafter\XINT_div_I_db
1392        \fi
1393     \fi
1394 }%
```

attention très mauvaises notations avec _b et _db.

```
1395 \def\XINT_div_I_dN #1\xint:%
1396 {%
1397     \expandafter\XINT_div_I_b\the\numexpr #1-\xint_c_i\xint:%
1398 }%
1399 \def\XINT_div_I_db #1\xint:#2#3#4#5%
1400 {%
1401     \expandafter\XINT_div_I_dc\expandafter #1%
1402     \romannumeral0\expandafter\XINT_div_sub\expandafter
1403        {\romannumeral0\XINT_rev_nounsep {}#4\R!\R!\R!\R!\R!\R!\R!\W}%
1404        {\the\numexpr\XINT_div_verysmallmul #1!#51;!}%
1405     \Z {#4}{#5}%
1406 }%
```

La soustraction spéciale renvoie simplement - si le chiffre q est trop grand. On invoque dans ce cas I_dP.

```
1407 \def\XINT_div_I_dc #1#2%
1408 {%
1409     \if-#2\expandafter\XINT_div_I_dd\else\expandafter\XINT_div_I_de\fi
1410      #1#2%
1411 }%
1412 \def\XINT_div_I_dd #1-\Z
1413 {%
1414     \if #11\expandafter\XINT_div_I_dz\fi
1415     \expandafter\XINT_div_I_dP\the\numexpr #1-\xint_c_i\xint: XX%
1416 }%
1417 \def\XINT_div_I_dz #1XX#2#3#4%
1418 {%
1419     1#4\XINT_div_I_g {#2}%
1420 }%
1421 \def\XINT_div_I_de #1#2\Z #3#4#5{1#5+#1\XINT_div_I_g {#2}}%
```

q.alpha, B, q0, L, K, {x'y},x, alpha'B«c» (q=0 has been intercepted) -> 1nouveauq.nouvel alpha, L, K, {x'y}, x, alpha',B«c»

```
\def\XINT_div_I_dP #1\xint:#2#3#4#5#6%
{%
    1#6+#1\expandafter\XINT_div_I_g\expandafter
    {\romannumeral0\expandafter\XINT_div_sub\expandafter
      {\romannumeral0\XINT_rev_nounsep {}#4\R!\R!\R!\R!\R!\R!\R!\W}%
      {\the\numexpr\XINT_div_verysmallmul #1!#51;!}%
    }%
}%
```

1#1=nouveau q. nouvel alpha, L, K, {x'y},x,alpha', BQ«c»

#1=q,#2=nouvel alpha,#3=L, #4=K, #5={x'y}, #6=x, #7= alpha',#8=B, «c» -> on laisse q puis {x'y}alpha.alpha'.{{x'y}xKL}B«c»

```
\def\XINT_div_I_g #1#2#3#4#5#6#7%
{%
     \expandafter !\the\numexpr
     \ifnum#2=#3
          \expandafter\XINT_div_exittofinish
     \else
          \expandafter\XINT_div_I_h
     \fi
     {#4}#1\xint:#6\xint:{{#4}{#5}{#3}{#2}}{#7}%
}%
```

{x'y}alpha.alpha'.{{x'y}xKL}B«c» -> Attention retour à l'envoyeur ici par terminaison des \the\numexpr. On doit reprendre le Q déjà sorti, qui n'a plus de séparateurs, ni de leading 1. Ensuite R sans leading zeros.«c»

```
\def\XINT_div_exittofinish #1#2\xint:#3\xint:#4#5%
{%
    1\expandafter\expandafter\expandafter!\expandafter\XINT_div_unsepQ_delim
    \romannumeral0\XINT_div_unsepR #2#3%
    \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax\R\xint:
}%
```

ATTENTION DESCRIPTION OBSOLÈTE. #1={x'y}alpha.#2!#3=reste de A. #4={{x'y},x,K,L},#5=B,«c» devient {x'y},alpha sur K+4 chiffres.B, {{x'y},x,K,L}, #6= nouvel alpha',B,«c»

```
\def\XINT_div_I_h #1\xint:#2!#3\xint:#4#5%
{%
    \XINT_div_II_b #1#2!\xint:{#5}{#4}{#3}{#5}%
}%
```

{x'y}alpha.B, {{x'y},x,K,L}, nouveau alpha',B,«c»

```
\def\XINT_div_II_b #11#2!#3!%
{%
    \xint_gob_til_eightzeroes #2\XINT_div_II_skipc 00000000%
    \XINT_div_II_c #1{1#2}{#3}%
}%
```

```
    x'y{100000000}{1<8>}reste de alpha.#6=B,#7={{x'y},x,K,L}, alpha',B, «c» -> {x'y}x,K,L (à diminuer de 4), {alpha sur K}B{q1=00000000}{alpha'}B,«c»

\def\XINT_div_II_skipc 00000000\XINT_div_II_c #1#2#3#4#5\xint:#6#7%
{%
    \XINT_div_II_k #7{#4!#5}{#6}{00000000}%
}%

    x'ya->1qx'yalpha.B, {{x'y},x,K,L}, nouveau alpha',B, «c». En fait, attention, ici #3 et #4 sont les 16 premiers chiffres du numérateur,sous la forme blocs 1<8chiffres>.

\def\XINT_div_II_c #1#2#3#4%
{%
     \expandafter\XINT_div_II_d\the\numexpr\XINT_div_xmini
     #1\xint:#2!#3!#4!{#1}{#2}#3!#4!%
}%
\def\XINT_div_xmini #1%
{%
    \xint_gob_til_one #1\XINT_div_xmini_a 1\XINT_div_mini #1%
}%
\def\XINT_div_xmini_a 1\XINT_div_mini 1#1%
{%
    \xint_gob_til_zero #1\XINT_div_xmini_b 0\XINT_div_mini 1#1%
}%
\def\XINT_div_xmini_b 0\XINT_div_mini 10#1#2#3#4#5#6#7%
{%
    \xint_gob_til_zero #7\XINT_div_xmini_c 0\XINT_div_mini 10#1#2#3#4#5#6#7%
}%

    x'=10^8 and we return #1=1<8digits>.

\def\XINT_div_xmini_c 0\XINT_div_mini 100000000\xint:50000000!#1!#2!{#1!}%

    1 suivi de q1 sur huit chiffres! #2=x', #3=y, #4=alpha.#5=B, {{x'y},x,K,L}, alpha', B, «c» --> nouvel alpha.x',y,B,q1,{{x'y},x,K,L}, alpha', B, «c»

\def\XINT_div_II_d 1#1#2#3#4#5!#6#7#8\xint:#9%
{%
    \expandafter\XINT_div_II_e
    \romannumeral0\expandafter\XINT_div_sub\expandafter
      {\romannumeral0\XINT_rev_nounsep {}#8\R!\R!\R!\R!\R!\R!\R!\W}%
      {\the\numexpr\XINT_div_smallmul_a 100000000\xint:#1#2#3#4\xint:#5!#91;!}%
    \xint:{#6}{#7}{#9}{#1#2#3#4#5}%
}%

    alpha.x',y,B,q1, {{x'y},x,K,L}, alpha', B, «c». Attention la soustraction spéciale doit maintenir les blocs 1<8>!

\def\XINT_div_II_e 1#1!%
{%
    \xint_gob_til_eightzeroes #1\XINT_div_II_skipf 00000000%
    \XINT_div_II_f 1#1!%
}%
```

```
    100000000! alpha sur K chiffres.#2=x',#3=y,#4=B,#5=q1, #6={{x'y},x,K,L}, #7=alpha',B«c» ->
  {x'y}x,K,L (à diminuer de 1), {alpha sur K}B{q1}{alpha'}B«c»

1490 \def\XINT_div_II_skipf 00000000\XINT_div_II_f 100000000!#1\xint:#2#3#4#5#6%
1491 {%
1492     \XINT_div_II_k #6{#1}{#4}{#5}%
1493 }%

    1<a1>!1<a2>!, alpha (sur K+1 blocs de 8). x', y, B, q1, {{x'y},x,K,L}, alpha', B,«c».
    Here also we are dividing with x' which could be 10^8 in the exceptional case x=99999999. Must
  intercept it before sending to \XINT_div_mini.

1494 \def\XINT_div_II_f #1!#2!#3\xint:%
1495 {%
1496     \XINT_div_II_fa {#1!#2!}{#1!#2!#3}%
1497 }%
1498 \def\XINT_div_II_fa #1#2#3#4%
1499 {%
1500     \expandafter\XINT_div_II_g \the\numexpr\XINT_div_xmini #3\xint:#4!#1{#2}%
1501 }%

    #1=q, #2=alpha (K+4), #3=B, #4=q1, {{x'y},x,K,L}, alpha', BQ«c»  -> 1 puis nouveau q sur 8
  chiffres. nouvel alpha sur K blocs,  B, {{x'y},x,K,L}, alpha',B«c»

1502 \def\XINT_div_II_g 1#1#2#3#4#5!#6#7#8%
1503 {%
1504     \expandafter \XINT_div_II_h
1505     \the\numexpr 1#1#2#3#4#5+#8\expandafter\expandafter\expandafter
1506     \xint:\expandafter\expandafter\expandafter
1507     {\expandafter\xint_gob_til_exclam
1508      \romannumeral0\expandafter\XINT_div_sub\expandafter
1509        {\romannumeral0\XINT_rev_nounsep {}#6\R!\R!\R!\R!\R!\R!\R!\W}%
1510        {\the\numexpr\XINT_div_smallmul_a 100000000\xint:#1#2#3#4\xint:#5!#71;!}}%
1511     {#7}%
1512 }%

    1 puis nouveau q sur 8 chiffres, #2=nouvel alpha sur K blocs, #3=B, #4={{x'y},x,K,L} avec L à
  ajuster, alpha', BQ«c» -> {x'y}x,K,L à diminuer de 1, {alpha}B{q}, alpha', BQ«c»

1513 \def\XINT_div_II_h 1#1\xint:#2#3#4%
1514 {%
1515     \XINT_div_II_k #4{#2}{#3}{#1}%
1516 }%

    {x'y}x,K,L à diminuer de 1, alpha, B{q}alpha',B«c» ->nouveau L.K,x',y,x,alpha.B,q,alpha',B,«c»
  ->{LK{x'y}x},x,a,alpha.B,q,alpha',B,«c»

1517 \def\XINT_div_II_k #1#2#3#4#5%
1518 {%
1519     \expandafter\XINT_div_II_l \the\numexpr #4-\xint_c_i\xint:{#3}#1{#2}#5\xint:%
1520 }%
1521 \def\XINT_div_II_l #1\xint:#2#3#4#51#6!%
1522 {%
1523     \XINT_div_II_m {{#1}{#2}{{#3}{#4}}{#5}}{#5}{#6}1#6!%
1524 }%
```

```
    {LK{x'y}x},x,a,alpha.B{q}alpha'B -> a, x, alpha, B, q, L, K, {x'y}, x, alpha', B«c»
```

```
\def\XINT_div_II_m #1#2#3#4\xint:#5#6%
{%
     \XINT_div_I_a {#3}{#2}{#4}{#5}{#6}#1%
}%
```

This multiplication is exactly like \XINT_smallmul -- apart from not inserting an ending 1;! --, but keeps ever a vanishing ending carry.

```
\def\XINT_div_minimulwc_a 1#1\xint:#2\xint:#3!#4#5#6#7#8\xint:%
{%
    \expandafter\XINT_div_minimulwc_b
    \the\numexpr \xint_c_x^ix+#1+#3*#8\xint:#3*#4#5#6#7+#2*#8\xint:#2*#4#5#6#7\xint:%
}%
\def\XINT_div_minimulwc_b 1#1#2#3#4#5#6\xint:#7\xint:%
{%
    \expandafter\XINT_div_minimulwc_c
    \the\numexpr \xint_c_x^ix+#1#2#3#4#5+#7\xint:#6\xint:%
}%
\def\XINT_div_minimulwc_c 1#1#2#3#4#5#6\xint:#7\xint:#8\xint:%
{%
    1#6#7\expandafter!%
    \the\numexpr\expandafter\XINT_div_smallmul_a
    \the\numexpr \xint_c_x^viii+#1#2#3#4#5+#8\xint:%
}%
\def\XINT_div_smallmul_a #1\xint:#2\xint:#3!1#4!%
{%
    \xint_gob_til_sc #4\XINT_div_smallmul_e;%
    \XINT_div_minimulwc_a #1\xint:#2\xint:#3!#4\xint:#2\xint:#3!%
}%
\def\XINT_div_smallmul_e;\XINT_div_minimulwc_a 1#1\xint:#2;#3!{1\relax #1!}%
```

Special very small multiplication for division. We only need to cater for multiplicands from 1 to 9. The ending is different from standard verysmallmul, a zero carry is not suppressed. And no final 1;! is added. If multiplicand is just 1 let's not forget to add the zero carry 100000000! at the end.

```
\def\XINT_div_verysmallmul #1%
   {\xint_gob_til_one #1\XINT_div_verysmallisone 1\XINT_div_verysmallmul_a 0\xint:#1}%
\def\XINT_div_verysmallisone 1\XINT_div_verysmallmul_a 0\xint:1!1#11;!%
   {1\relax #1100000000!}%
\def\XINT_div_verysmallmul_a #1\xint:#2!1#3!%
{%
    \xint_gob_til_sc #3\XINT_div_verysmallmul_e;%
    \expandafter\XINT_div_verysmallmul_b
    \the\numexpr \xint_c_x^ix+#2*#3+#1\xint:#2!%
}%
\def\XINT_div_verysmallmul_b 1#1#2\xint:%
    {1#2\expandafter!\the\numexpr\XINT_div_verysmallmul_a #1\xint:}%
\def\XINT_div_verysmallmul_e;#1;+#2#3!{1\relax 0000000#2!}%
```

Special subtraction for division purposes. If the subtracted thing turns out to be bigger, then just return a -. If not, then we must reverse the result, keeping the separators.

```
\def\XINT_div_sub #1#2%
{%
    \expandafter\XINT_div_sub_clean
    \the\numexpr\expandafter\XINT_div_sub_a\expandafter
    1#2;!;!;!;!;!\W #1;!;!;!;!;!\W
}%
\def\XINT_div_sub_clean #1-#2#3\W
{%
    \if1#2\expandafter\XINT_rev_nounsep\else\expandafter\XINT_div_sub_neg\fi
    {}#1\R!\R!\R!\R!\R!\R!\R!\W
}%
\def\XINT_div_sub_neg #1\W { -}%
\def\XINT_div_sub_a #1!#2!#3!#4!#5\W #6!#7!#8!#9!%
{%
    \XINT_div_sub_b #1!#6!#2!#7!#3!#8!#4!#9!#5\W
}%
\def\XINT_div_sub_b #1#2#3!#4!%
{%
    \xint_gob_til_sc #4\XINT_div_sub_bi ;%
    \expandafter\XINT_div_sub_c\the\numexpr#1-#3+1#4-\xint_c_i\xint:%
}%
\def\XINT_div_sub_c 1#1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_div_sub_d #1%
}%
\def\XINT_div_sub_d #1#2#3!#4!%
{%
    \xint_gob_til_sc #4\XINT_div_sub_di ;%
    \expandafter\XINT_div_sub_e\the\numexpr#1-#3+1#4-\xint_c_i\xint:%
}%
\def\XINT_div_sub_e 1#1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_div_sub_f #1%
}%
\def\XINT_div_sub_f #1#2#3!#4!%
{%
    \xint_gob_til_sc #4\XINT_div_sub_fi ;%
    \expandafter\XINT_div_sub_g\the\numexpr#1-#3+1#4-\xint_c_i\xint:%
}%
\def\XINT_div_sub_g 1#1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_div_sub_h #1%
}%
\def\XINT_div_sub_h #1#2#3!#4!%
{%
    \xint_gob_til_sc #4\XINT_div_sub_hi ;%
    \expandafter\XINT_div_sub_i\the\numexpr#1-#3+1#4-\xint_c_i\xint:%
}%
\def\XINT_div_sub_i 1#1#2\xint:%
{%
    1#2\expandafter!\the\numexpr\XINT_div_sub_a #1%
}%
```

```
\def\XINT_div_sub_bi;%
    \expandafter\XINT_div_sub_c\the\numexpr#1-#2+#3\xint:#4!#5!#6!#7!#8!#9!;!\W
{%
    \XINT_div_sub_l #1#2!#5!#7!#9!%
}%
\def\XINT_div_sub_di;%
    \expandafter\XINT_div_sub_e\the\numexpr#1-#2+#3\xint:#4!#5!#6!#7!#8\W
{%
    \XINT_div_sub_l #1#2!#5!#7!%
}%
\def\XINT_div_sub_fi;%
    \expandafter\XINT_div_sub_g\the\numexpr#1-#2+#3\xint:#4!#5!#6\W
{%
    \XINT_div_sub_l #1#2!#5!%
}%
\def\XINT_div_sub_hi;%
    \expandafter\XINT_div_sub_i\the\numexpr#1-#2+#3\xint:#4\W
{%
    \XINT_div_sub_l #1#2!%
}%
\def\XINT_div_sub_l #1%
{%
   \xint_UDzerofork
      #1{-2\relax}%
       0\XINT_div_sub_r
   \krof
}%
\def\XINT_div_sub_r #1!%
{%
    -\ifnum 0#1=\xint_c_ 1\else2\fi\relax
}%
```

Ici B<10^8 (et est >2). On exécute
\expandafter\XINT_sdiv_out\the\numexpr\XINT_smalldivx_a x.1B!1<8d>!...1<8d>!1;!
avec x =round(B/2), 1B=10^8+B, et A déjà en blocs 1<8d>! (non renversés). Le \the\numexpr\XINT_smalldivx_a va produire Q\Z R\W avec un R<10^8, et un Q sous forme de blocs 1<8d>! terminé par 1! et nécessitant le nettoyage du premier bloc. Dans cette branche le B n'a pas été multiplié par une puissance de 10, il peut avoir moins de huit chiffres.

```
\def\XINT_sdiv_out #1;!#2!%
    {\expandafter
     {\romannumeral0\XINT_unsep_cuzsmall
      #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax}%
     {#2}}%
```

La toute première étape fait la première division pour être sûr par la suite d'avoir un premier bloc pour A qui sera < B.

```
\def\XINT_smalldivx_a #1\xint:1#2!1#3!%
{%
    \expandafter\XINT_smalldivx_b
    \the\numexpr (#3+#1)/#2-\xint_c_i!#1\xint:#2!#3!%
}%
\def\XINT_smalldivx_b #1#2!%
```

```
{%
    \if0#1\else
          \xint_c_x^viii+#1#2\xint_afterfi{\expandafter!\the\numexpr}\fi
    \XINT_smalldiv_c #1#2!%
}%
\def\XINT_smalldiv_c #1!#2\xint:#3!#4!%
{%
    \expandafter\XINT_smalldiv_d\the\numexpr #4-#1*#3!#2\xint:#3!%
}%
```

On va boucler ici: #1 est un reste, #2 est x.B (avec B sans le 1 mais sur huit chiffres). #3#4 est le premier bloc qui reste de A. Si on a terminé avec A, alors #1 est le reste final. Le quotient lui est terminé par un 1! ce 1! disparaîtra dans le nettoyage par \XINT_unsep_cuzsmall.

```
\def\XINT_smalldiv_d #1!#2!1#3#4!%
{%
    \xint_gob_til_sc #3\XINT_smalldiv_end ;%
    \XINT_smalldiv_e #1!#2!1#3#4!%
}%
\def\XINT_smalldiv_end;\XINT_smalldiv_e #1!#2!1;!{1!;!#1!}%
```

Il est crucial que le reste #1 est < #3. J'ai documenté cette routine dans le fichier où j'ai préparé 1.2, il faudra transférer ici. Il n'est pas nécessaire pour cette routine que le diviseur B ait au moins 8 chiffres. Mais il doit être < 10^8.

```
\def\XINT_smalldiv_e #1!#2\xint:#3!%
{%
    \expandafter\XINT_smalldiv_f\the\numexpr
    \xint_c_xi_e_viii_mone+#1*\xint_c_x^viii/#3!#2\xint:#3!#1!%
}%
\def\XINT_smalldiv_f 1#1#2#3#4#5#6!#7\xint:#8!%
{%
     \xint_gob_til_zero #1\XINT_smalldiv_fz 0%
     \expandafter\XINT_smalldiv_g
     \the\numexpr\XINT_minimul_a #2#3#4#5\xint:#6!#8!#2#3#4#5#6!#7\xint:#8!%
}%
\def\XINT_smalldiv_fz 0%
    \expandafter\XINT_smalldiv_g\the\numexpr\XINT_minimul_a
    9999\xint:9999!#1!99999999!#2!0!1#3!%
{%
    \XINT_smalldiv_i \xint:#3!\xint_c_!#2!%
}%
\def\XINT_smalldiv_g 1#1!1#2!#3!#4!#5!#6!%
{%
    \expandafter\XINT_smalldiv_h\the\numexpr 1#6-#1\xint:#2!#5!#3!#4!%
}%
\def\XINT_smalldiv_h 1#1#2\xint:#3!#4!%
{%
    \expandafter\XINT_smalldiv_i\the\numexpr #4-#3+#1-\xint_c_i\xint:#2!%
}%
\def\XINT_smalldiv_i #1\xint:#2!#3!#4\xint:#5!%
{%
    \expandafter\XINT_smalldiv_j\the\numexpr (#1#2+#4)/#5-\xint_c_i!#3!#1#2!#4\xint:#5!%
}%
```

```
\def\XINT_smalldiv_j #1!#2!%
{%
    \xint_c_x^viii+#1+#2\expandafter!\the\numexpr\XINT_smalldiv_k
    #1!%
}%
```

On boucle vers \XINT_smalldiv_d.

```
\def\XINT_smalldiv_k #1!#2!#3\xint:#4!%
{%
    \expandafter\XINT_smalldiv_d\the\numexpr #2-#1*#4!#3\xint:#4!%
}%
```

Cette routine fait la division euclidienne d'un nombre de seize chiffres par #1 = C = diviseur sur huit chiffres >= 10^7, avec #2 = sa moitié utilisée dans \numexpr pour contrebalancer l'arrondi (ARRRRRRGGGGGHHHH) fait par /. Le nombre divisé XY = X*10^8+Y se présente sous la forme 1<8chiffres>!1<8chiffres>! avec plus significatif en premier.

Seul le quotient est calculé, pas le reste. En effet la routine de division principale va utiliser ce quotient pour déterminer le "grand" reste, et le petit reste ici ne nous serait d'à peu près aucune utilité.

ATTENTION UNIQUEMENT UTILISÉ POUR DES SITUATIONS OÙ IL EST GARANTI QUE X < C ! (et C au moins 10^7) le quotient euclidien de X*10^8+Y par C sera donc < 10^8. Il sera renvoyé sous la forme 1<8chiffres>.

```
\def\XINT_div_mini #1\xint:#2!1#3!%
{%
    \expandafter\XINT_div_mini_a\the\numexpr
    \xint_c_xi_e_viii_mone+#3*\xint_c_x^viii/#1!#1\xint:#2!#3!%
}%
```

Note (2015/10/08). Attention à la différence dans l'ordre des arguments avec ce que je vois en dans \XINT_smalldiv_f. Je ne me souviens plus du tout s'il y a une raison quelconque.

```
\def\XINT_div_mini_a 1#1#2#3#4#5#6!#7\xint:#8!%
{%
     \xint_gob_til_zero #1\XINT_div_mini_w 0%
     \expandafter\XINT_div_mini_b
     \the\numexpr\XINT_minimul_a #2#3#4#5\xint:#6!#7!#2#3#4#5#6!#7\xint:#8!%
}%
\def\XINT_div_mini_w 0%
    \expandafter\XINT_div_mini_b\the\numexpr\XINT_minimul_a
    9999\xint:9999!#1!99999999!#2\xint:#3!00000000!#4!%
{%
    \xint_c_x^viii_mone+(#4+#3)/#2!%
}%
\def\XINT_div_mini_b 1#1!1#2!#3!#4!#5!#6!%
{%
    \expandafter\XINT_div_mini_c
    \the\numexpr 1#6-#1\xint:#2!#5!#3!#4!%
}%
\def\XINT_div_mini_c 1#1#2\xint:#3!#4!%
{%
    \expandafter\XINT_div_mini_d
    \the\numexpr #4-#3+#1-\xint_c_i\xint:#2!%
```

```
}%
\def\XINT_div_mini_d #1\xint:#2!#3!#4\xint:#5!%
{%
    \xint_c_x^viii_mone+#3+(#1#2+#5)/#4!%
}%
```

Derived arithmetic

4.38 \xintiiQuo, \xintiiRem

```
\def\xintiiQuo {\romannumeral0\xintiiquo }%
\def\xintiiRem {\romannumeral0\xintiirem }%
\def\xintiiquo
   {\expandafter\xint_stop_atfirstoftwo\romannumeral0\xintiidivision }%
\def\xintiirem
   {\expandafter\xint_stop_atsecondoftwo\romannumeral0\xintiidivision }%
```

4.39 \xintiiDivRound

1.1, transferred from first release of bnumexpr. Rewritten for 1.2. Ending rewritten for 1.2i. (new \xintDSRr).
1.2l: \xintiiDivRound made robust against non terminated input.

```
\def\xintiiDivRound   {\romannumeral0\xintiidivround }%
\def\xintiidivround #1{\expandafter\XINT_iidivround\romannumeral`&&@#1\xint:}%
\def\XINT_iidivround #1#2\xint:#3%
    {\expandafter\XINT_iidivround_a\expandafter #1\romannumeral`&&@#3\xint:#2\xint:}%
\def\XINT_iidivround_a #1#2% #1 de A, #2 de B.
{%
    \if0#2\xint_dothis{\XINT_iidivround_divbyzero#1#2}\fi
    \if0#1\xint_dothis\XINT_iidivround_aiszero\fi
    \if-#2\xint_dothis{\XINT_iidivround_bneg #1}\fi
          \xint_orthat{\XINT_iidivround_bpos #1#2}%
}%
\def\XINT_iidivround_divbyzero #1#2#3\xint:#4\xint:
   {\XINT_signalcondition{DivisionByZero}{Division of #1#4 by #2#3}{}{0}}%
\def\XINT_iidivround_aiszero   #1\xint:#2\xint:{ 0}%
\def\XINT_iidivround_bpos #1%
{%
    \xint_UDsignfork
             #1{\xintiiopp\XINT_iidivround_pos {}}%
              -{\XINT_iidivround_pos #1}%
    \krof
}%
\def\XINT_iidivround_bneg #1%
{%
    \xint_UDsignfork
             #1{\XINT_iidivround_pos {}}%
              -{\xintiiopp\XINT_iidivround_pos #1}%
    \krof
}%
\def\XINT_iidivround_pos #1#2\xint:#3\xint:
{%
    \expandafter\expandafter\expandafter\XINT_dsrr
```

```
    \expandafter\xint_firstoftwo
    \romannumeral0\XINT_div_prepare {#2}{#1#30}%
    \xint_bye\xint_Bye3456789\xint_bye/\xint_c_x\relax
}%
```

4.40 \xintiiDivTrunc

1.2l: \xintiiDivTrunc made robust against non terminated input.

```
\def\xintiiDivTrunc   {\romannumeral0\xintiidivtrunc }%
\def\xintiidivtrunc #1{\expandafter\XINT_iidivtrunc\romannumeral`&&@#1\xint:}%
\def\XINT_iidivtrunc #1#2\xint:#3{\expandafter\XINT_iidivtrunc_a\expandafter #1%
                                  \romannumeral`&&@#3\xint:#2\xint:}%
\def\XINT_iidivtrunc_a #1#2% #1 de A, #2 de B.
{%
    \if0#2\xint_dothis{\XINT_iidivtrunc_divbyzero#1#2}\fi
    \if0#1\xint_dothis\XINT_iidivtrunc_aiszero\fi
    \if-#2\xint_dothis{\XINT_iidivtrunc_bneg #1}\fi
          \xint_orthat{\XINT_iidivtrunc_bpos #1#2}%
}%
```

Attention to not move DivRound code beyond that point.

```
\let\XINT_iidivtrunc_divbyzero\XINT_iidivround_divbyzero
\let\XINT_iidivtrunc_aiszero  \XINT_iidivround_aiszero
\def\XINT_iidivtrunc_bpos #1%
{%
    \xint_UDsignfork
            #1{\xintiiopp\XINT_iidivtrunc_pos {}}%
             -{\XINT_iidivtrunc_pos #1}%
    \krof
}%
\def\XINT_iidivtrunc_bneg #1%
{%
    \xint_UDsignfork
            #1{\XINT_iidivtrunc_pos {}}%
             -{\xintiiopp\XINT_iidivtrunc_pos #1}%
    \krof
}%
\def\XINT_iidivtrunc_pos #1#2\xint:#3\xint:
    {\expandafter\xint_stop_atfirstoftwo
     \romannumeral0\XINT_div_prepare {#2}{#1#3}}%
```

4.41 \xintiiModTrunc

Renamed from \xintiiMod to \xintiiModTrunc at 1.2p.

```
\def\xintiiModTrunc {\romannumeral0\xintiimodtrunc }%
\def\xintiimodtrunc #1{\expandafter\XINT_iimodtrunc\romannumeral`&&@#1\xint:}%
\def\XINT_iimodtrunc #1#2\xint:#3{\expandafter\XINT_iimodtrunc_a\expandafter #1%
                                  \romannumeral`&&@#3\xint:#2\xint:}%
\def\XINT_iimodtrunc_a #1#2% #1 de A, #2 de B.
{%
```

```
    \if0#2\xint_dothis{\XINT_iimodtrunc_divbyzero#1#2}\fi
    \if0#1\xint_dothis\XINT_iimodtrunc_aiszero\fi
    \if-#2\xint_dothis{\XINT_iimodtrunc_bneg #1}\fi
          \xint_orthat{\XINT_iimodtrunc_bpos #1#2}%
}%
```

Attention to not move DivRound code beyond that point. A bit of abuse here for divbyzero defaulted-to value, which happily works in both.

```
\let\XINT_iimodtrunc_divbyzero\XINT_iidivround_divbyzero
\let\XINT_iimodtrunc_aiszero  \XINT_iidivround_aiszero
\def\XINT_iimodtrunc_bpos #1%
{%
    \xint_UDsignfork
            #1{\xintiiopp\XINT_iimodtrunc_pos {}}%
             -{\XINT_iimodtrunc_pos #1}%
    \krof
}%
\def\XINT_iimodtrunc_bneg #1%
{%
    \xint_UDsignfork
            #1{\xintiiopp\XINT_iimodtrunc_pos {}}%
             -{\XINT_iimodtrunc_pos #1}%
    \krof
}%
\def\XINT_iimodtrunc_pos #1#2\xint:#3\xint:
    {\expandafter\xint_stop_atsecondoftwo\romannumeral0\XINT_div_prepare
      {#2}{#1#3}}%
```

4.42 \xintiiDivMod

1.2p. It is associated with floored division (like Python divmod function), and with the // operator in \xintiiexpr.

```
\def\xintiiDivMod   {\romannumeral0\xintiidivmod }%
\def\xintiidivmod #1{\expandafter\XINT_iidivmod\romannumeral`&&@#1\xint:}%
\def\XINT_iidivmod #1#2\xint:#3{\expandafter\XINT_iidivmod_a\expandafter #1%
                                \romannumeral`&&@#3\xint:#2\xint:}%
\def\XINT_iidivmod_a #1#2% #1 de A, #2 de B.
{%
    \if0#2\xint_dothis{\XINT_iidivmod_divbyzero#1#2}\fi
    \if0#1\xint_dothis\XINT_iidivmod_aiszero\fi
    \if-#2\xint_dothis{\XINT_iidivmod_bneg #1}\fi
          \xint_orthat{\XINT_iidivmod_bpos #1#2}%
}%
\def\XINT_iidivmod_divbyzero #1#2\xint:#3\xint:
{%
    \XINT_signalcondition{DivisionByZero}{Division by #2 of #1#3}{}%
    {{0}{0}}% à revoir...
}%
\def\XINT_iidivmod_aiszero #1\xint:#2\xint:{{0}{0}}%
\def\XINT_iidivmod_bneg #1%
{%
    \expandafter\XINT_iidivmod_bneg_finish
```

```
    \romannumeral0\xint_UDsignfork
             #1{\XINT_iidivmod_bpos {}}%
              -{\XINT_iidivmod_bpos {-#1}}%
    \krof
}%
\def\XINT_iidivmod_bneg_finish#1#2%
{%
    \expandafter\xint_exchangetwo_keepbraces\expandafter
    {\romannumeral0\xintiiopp#2}{#1}%
}%
\def\XINT_iidivmod_bpos #1#2\xint:#3\xint:{\xintiidivision{#1#3}{#2}}%
```

4.43 \xintiiDivFloor

1.2p. For bnumexpr actually, because \xintiiexpr could use \xintDivFloor which also outputs an integer in strict format.

```
\def\xintiiDivFloor {\romannumeral0\xintiidivfloor}%
\def\xintiidivfloor {\expandafter\xint_stop_atfirstoftwo
                \romannumeral0\xintiidivmod}%
```

4.44 \xintiiMod

Associated with floored division at 1.2p. Formerly was associated with truncated division.

```
\def\xintiiMod {\romannumeral0\xintiimod}%
\def\xintiimod {\expandafter\xint_stop_atsecondoftwo
                \romannumeral0\xintiidivmod}%
```

4.45 \xintiiSqr

1.2l: \xintiiSqr made robust against non terminated input.

```
\def\xintiiSqr {\romannumeral0\xintiisqr }%
\def\xintiisqr #1%
{%
    \expandafter\XINT_sqr\romannumeral0\xintiiabs{#1}\xint:
}%
\def\XINT_sqr #1\xint:
{%
    \expandafter\XINT_sqr_a
      \romannumeral0\expandafter\XINT_sepandrev_andcount
      \romannumeral0\XINT_zeroes_forviii #1\R\R\R\R\R\R\R\R{10}0000001\W
      #1\XINT_rsepbyviii_end_A 2345678%
        \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
             \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
             \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
      \xint:
}%
```

1.2c \XINT_mul_loop can now be called directly even with small arguments, thus the following check is not anymore a necessity.

```
\def\XINT_sqr_a #1\xint:
{%
    \ifnum #1=\xint_c_i \expandafter\XINT_sqr_small
                    \else\expandafter\XINT_sqr_start\fi
}%
\def\XINT_sqr_small 1#1#2#3#4#5!\xint:
{%
    \ifnum #1#2#3#4#5<46341 \expandafter\XINT_sqr_verysmall\fi
    \expandafter\XINT_sqr_small_out
    \the\numexpr\XINT_minimul_a #1#2#3#4\xint:#5!#1#2#3#4#5!%
}%
\def\XINT_sqr_verysmall#1{%
\def\XINT_sqr_verysmall
    \expandafter\XINT_sqr_small_out\the\numexpr\XINT_minimul_a ##1!##2!%
    {\expandafter#1\the\numexpr ##2*##2\relax}%
}\XINT_sqr_verysmall{ }%
\def\XINT_sqr_small_out 1#1!1#2!%
{%
    \XINT_cuz #2#1\R
}%
```

An ending 1;! is produced on output for \XINT_mul_loop and gets incorporated to the delimiter needed by the \XINT_unrevbyviii done by \XINT_mul_out.

```
\def\XINT_sqr_start #1\xint:
{%
    \expandafter\XINT_mul_out
    \the\numexpr\XINT_mul_loop
                100000000!1;!\W #11;!\W #11;!%
    1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
}%
```

4.46 \xintiiPow

The exponent is not limited but with current default settings of tex memory, with xint 1.2, the maximal exponent for 2^N is N = 2^17 = 131072.

1.2f Modifies the initial steps: 1) in order to be able to let more easily \xintiPow use \xintNum on the exponent once xintfrac.sty is loaded; 2) also because I noticed it was not very well coded. And it did only a \numexpr on the exponent, contradicting the documentation related to the "i" convention in names.

1.2l: \xintiiPow made robust against non terminated input.

```
\def\xintiiPow {\romannumeral0\xintiipow }%
\def\xintiipow #1#2%
{%
    \expandafter\xint_pow\the\numexpr #2\expandafter
    .\romannumeral`&&@#1\xint:
}%
\def\xint_pow #1.#2%#3\xint:
{%
    \xint_UDzerominusfork
      #2-\XINT_pow_AisZero
      0#2\XINT_pow_Aneg
```

```
      0-{\XINT_pow_Apos #2}%
    \krof {#1}%
}%
\def\XINT_pow_AisZero #1#2\xint:
{%
     \ifcase\XINT_cntSgn #1\xint:
         \xint_afterfi { 1}%
     \or
         \xint_afterfi { 0}%
     \else
         \xint_afterfi
        {\XINT_signalcondition{DivisionByZero}{Zero to power #1}{}{0}}%
     \fi
}%
\def\XINT_pow_Aneg #1%
{%
   \ifodd #1
       \expandafter\XINT_opp\romannumeral0%
   \fi
   \XINT_pow_Apos {}{#1}%
}%
\def\XINT_pow_Apos #1#2{\XINT_pow_Apos_a {#2}#1}%
\def\XINT_pow_Apos_a #1#2#3%
{%
    \xint_gob_til_xint: #3\XINT_pow_Apos_short\xint:
    \XINT_pow_AatleastTwo {#1}#2#3%
}%
\def\XINT_pow_Apos_short\xint:\XINT_pow_AatleastTwo #1#2\xint:
{%
    \ifcase #2
         \xintError:thiscannothappen
    \or  \expandafter\XINT_pow_AisOne
    \else\expandafter\XINT_pow_AatleastTwo
    \fi {#1}#2\xint:
}%
\def\XINT_pow_AisOne #1\xint:{ 1}%
\def\XINT_pow_AatleastTwo #1%
{%
    \ifcase\XINT_cntSgn #1\xint:
        \expandafter\XINT_pow_BisZero
    \or
        \expandafter\XINT_pow_I_in
    \else
        \expandafter\XINT_pow_BisNegative
    \fi
    {#1}%
}%
\def\XINT_pow_BisNegative #1\xint:{\XINT_signalcondition{Underflow}{Inverse power
    can not be represented by an integer}{}{0}}%
\def\XINT_pow_BisZero #1\xint:{ 1}%
```

B = #1 > 0, A = #2 > 1. Earlier code checked if size of B did not exceed a given limit (for example 131000).

```
\def\XINT_pow_I_in #1#2\xint:
{%
    \expandafter\XINT_pow_I_loop
    \the\numexpr #1\expandafter\xint:%
    \romannumeral0\expandafter\XINT_sepandrev
    \romannumeral0\XINT_zeroes_forviii #2\R\R\R\R\R\R\R{10}0000001\W
    #2\XINT_rsepbyviii_end_A 2345678%
      \XINT_rsepbyviii_end_B 2345678\relax XX%
    \R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\W
    1;!\W
    1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
}%
\def\XINT_pow_I_loop #1\xint:%
{%
    \ifnum #1 = \xint_c_i\expandafter\XINT_pow_I_exit\fi
    \ifodd #1
       \expandafter\XINT_pow_II_in
    \else
       \expandafter\XINT_pow_I_squareit
    \fi #1\xint:%
}%
\def\XINT_pow_I_exit \ifodd #1\fi #2\xint:#3\W {\XINT_mul_out #3}%
```

The 1.2c \XINT_mul_loop can be called directly even with small arguments, hence the "butcheckifsmall" is not a necessity as it was earlier with 1.2. On 2^30, it does bring roughly a 40% time gain though, and 30% gain for 2^60. The overhead on big computations should be negligible.

```
\def\XINT_pow_I_squareit #1\xint:#2\W%
{%
    \expandafter\XINT_pow_I_loop
    \the\numexpr #1/\xint_c_ii\expandafter\xint:%
    \the\numexpr\XINT_pow_mulbutcheckifsmall #2\W #2\W
}%
\def\XINT_pow_mulbutcheckifsmall #1!1#2%
{%
    \xint_gob_til_sc #2\XINT_pow_mul_small;%
    \XINT_mul_loop 100000000!1;!\W #1!1#2%
}%
\def\XINT_pow_mul_small;\XINT_mul_loop
    100000000!1;!\W 1#1!1;!\W
{%
    \XINT_smallmul 1#1!%
}%
\def\XINT_pow_II_in #1\xint:#2\W
{%
    \expandafter\XINT_pow_II_loop
    \the\numexpr #1/\xint_c_ii-\xint_c_i\expandafter\xint:%
    \the\numexpr\XINT_pow_mulbutcheckifsmall #2\W #2\W #2\W
}%
\def\XINT_pow_II_loop #1\xint:%
{%
    \ifnum #1 = \xint_c_i\expandafter\XINT_pow_II_exit\fi
    \ifodd #1
```

```
2032        \expandafter\XINT_pow_II_odda
2033    \else
2034        \expandafter\XINT_pow_II_even
2035    \fi #1\xint:%
2036 }%
2037 \def\XINT_pow_II_exit\ifodd #1\fi #2\xint:#3\W #4\W
2038 {%
2039     \expandafter\XINT_mul_out
2040     \the\numexpr\XINT_pow_mulbutcheckifsmall #4\W #3%
2041 }%
2042 \def\XINT_pow_II_even #1\xint:#2\W
2043 {%
2044     \expandafter\XINT_pow_II_loop
2045     \the\numexpr #1/\xint_c_ii\expandafter\xint:%
2046     \the\numexpr\XINT_pow_mulbutcheckifsmall #2\W #2\W
2047 }%
2048 \def\XINT_pow_II_odda #1\xint:#2\W #3\W
2049 {%
2050     \expandafter\XINT_pow_II_oddb
2051     \the\numexpr #1/\xint_c_ii-\xint_c_i\expandafter\xint:%
2052     \the\numexpr\XINT_pow_mulbutcheckifsmall #3\W #2\W #2\W
2053 }%
2054 \def\XINT_pow_II_oddb #1\xint:#2\W #3\W
2055 {%
2056     \expandafter\XINT_pow_II_loop
2057     \the\numexpr #1\expandafter\xint:%
2058     \the\numexpr\XINT_pow_mulbutcheckifsmall #3\W #3\W #2\W
2059 }%
```

4.47 \xintiiFac

Moved here from xint.sty with release 1.2 (to be usable by \bnumexpr).

An \xintiFac is needed by xintexpr.sty. Prior to 1.2o it was defined here as an alias to \xintiiFac, then redefined by xintfrac to use \xintNum. This was incoherent. Contrarily to other similarly named macros, \xintiiFac uses \numexpr on its input. This is also incoherent with the naming scheme, alas.

Partially rewritten with release 1.2 to benefit from the inner format of the 1.2 multiplication.

With current default settings of the etex memory and a.t.t.o.w (11/2015) the maximal possible computation is 5971! (which has 19956 digits).

Note (end november 2015): I also tried out a quickly written recursive (binary split) implementation

```
 \catcode`_ 11
 \catcode`^ 11
 \long\def\xint_firstofthree  #1#2#3{#1}%
 \long\def\xint_secondofthree #1#2#3{#2}%
 \long\def\xint_thirdofthree  #1#2#3{#3}%
 % quickly written factorial using binary split recursive method
 \def\tFac   {\romannumeral-`0\tfac }%
 \def\tfac #1{\expandafter\XINT_mul_out
              \romannumeral-`0\ufac {1}{#1}1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W}%
 \def\ufac #1#2{\ifcase\numexpr#2-#1\relax
                \expandafter\xint_firstofthree
```

```
                  \or
                    \expandafter\xint_secondofthree
                  \else
                    \expandafter\xint_thirdofthree
                  \fi
                  {\the\numexpr\xint_c_x^viii+#1!1;!}%
                  {\the\numexpr\xint_c_x^viii+#1*#2!1;!}%
                  {\expandafter\vfac\the\numexpr (#1+#2)/\xint_c_ii.#1.#2.}%
    }%
    \def\vfac #1.#2.#3.%
    {%
        \expandafter
        \wfac\expandafter
            {\romannumeral-`0\expandafter
             \ufac\expandafter{\the\numexpr #1+\xint_c_i}{#3}}%
            {\ufac {#2}{#1}}%
    }%
    \def\wfac #1#2{\expandafter\zfac\romannumeral-`0#2\W #1}%
    \def\zfac {\the\numexpr\XINT_mul_loop 100000000!1;!\W }% core multiplication...
    \catcode`_ 8
    \catcode`^ 7
  and I was quite surprised that it was only about 1.6x--2x slower in the range N=200 to 2000 than
  the \xintiiFac here which attempts to be smarter...
    Note (2017, 1.21): I found out some code comment of mine that the code here should be more in the
  style of \xintiiBinomial, but I left matters untouched.
    1.2o modifies \xintiFac to be coherent with \xintiBinomial: only with xintfrac.sty loaded does
  it use \xintNum. It is documented only as macro of xintfrac.sty, not as macro of xint.sty.

\def\xintiiFac {\romannumeral0\xintiifac }%
\def\xintiifac #1{\expandafter\XINT_fac_fork\the\numexpr#1.}%
\def\XINT_fac_fork #1#2.%
{%
    \xint_UDzerominusfork
     #1-\XINT_fac_zero
     0#1\XINT_fac_neg
      0-\XINT_fac_checksize
    \krof #1#2.%
}%
\def\XINT_fac_zero #1.{ 1}%
\def\XINT_fac_neg  #1.{\XINT_signalcondition{InvalidOperation}{Factorial of
    negative: (#1)!}{}{0}}%
\def\XINT_fac_checksize #1.%
{%
    \ifnum #1>\xint_c_x^iv \xint_dothis{\XINT_fac_toobig #1.}\fi
    \ifnum #1>465 \xint_dothis{\XINT_fac_bigloop_a   #1.}\fi
    \ifnum #1>101 \xint_dothis{\XINT_fac_medloop_a   #1.\XINT_mul_out}\fi
                  \xint_orthat{\XINT_fac_smallloop_a #1.\XINT_mul_out}%
    1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
}%
\def\XINT_fac_toobig #1.#2\W{\XINT_signalcondition{InvalidOperation}{Factorial
    of too big argument: #1 > 10000}{}{0}}%
\def\XINT_fac_bigloop_a #1.%
```

```
{%
    \expandafter\XINT_fac_bigloop_b \the\numexpr
    #1+\xint_c_i-\xint_c_ii*((#1-464)/\xint_c_ii).#1.%
}%
\def\XINT_fac_bigloop_b #1.#2.%
{%
    \expandafter\XINT_fac_medloop_a
        \the\numexpr #1-\xint_c_i.{\XINT_fac_bigloop_loop #1.#2.}%
}%
\def\XINT_fac_bigloop_loop #1.#2.%
{%
    \ifnum #1>#2 \expandafter\XINT_fac_bigloop_exit\fi
    \expandafter\XINT_fac_bigloop_loop
    \the\numexpr #1+\xint_c_ii\expandafter.%
    \the\numexpr #2\expandafter.\the\numexpr\XINT_fac_bigloop_mul #1!%
}%
\def\XINT_fac_bigloop_exit #1!{\XINT_mul_out}%
\def\XINT_fac_bigloop_mul #1!%
{%
    \expandafter\XINT_smallmul
        \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
}%
\def\XINT_fac_medloop_a #1.%
{%
    \expandafter\XINT_fac_medloop_b
        \the\numexpr #1+\xint_c_i-\xint_c_iii*((#1-100)/\xint_c_iii).#1.%
}%
\def\XINT_fac_medloop_b #1.#2.%
{%
    \expandafter\XINT_fac_smallloop_a
        \the\numexpr #1-\xint_c_i.{\XINT_fac_medloop_loop #1.#2.}%
}%
\def\XINT_fac_medloop_loop #1.#2.%
{%
    \ifnum #1>#2 \expandafter\XINT_fac_loop_exit\fi
    \expandafter\XINT_fac_medloop_loop
    \the\numexpr #1+\xint_c_iii\expandafter.%
    \the\numexpr #2\expandafter.\the\numexpr\XINT_fac_medloop_mul #1!%
}%
\def\XINT_fac_medloop_mul #1!%
{%
    \expandafter\XINT_smallmul
    \the\numexpr
        \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_fac_smallloop_a #1.%
{%
    \csname
       XINT_fac_smallloop_\the\numexpr #1-\xint_c_iv*(#1/\xint_c_iv)\relax
    \endcsname #1.%
}%
\expandafter\def\csname XINT_fac_smallloop_1\endcsname #1.%
```

```
{%
    \XINT_fac_smallloop_loop 2.#1.100000001!1;!%
}%
\expandafter\def\csname XINT_fac_smallloop_-2\endcsname #1.%
{%
    \XINT_fac_smallloop_loop 3.#1.100000002!1;!%
}%
\expandafter\def\csname XINT_fac_smallloop_-1\endcsname #1.%
{%
    \XINT_fac_smallloop_loop 4.#1.100000006!1;!%
}%
\expandafter\def\csname XINT_fac_smallloop_0\endcsname #1.%
{%
    \XINT_fac_smallloop_loop 5.#1.1000000024!1;!%
}%
\def\XINT_fac_smallloop_loop #1.#2.%
{%
    \ifnum #1>#2 \expandafter\XINT_fac_loop_exit\fi
    \expandafter\XINT_fac_smallloop_loop
    \the\numexpr #1+\xint_c_iv\expandafter.%
    \the\numexpr #2\expandafter.\the\numexpr\XINT_fac_smallloop_mul #1!%
}%
\def\XINT_fac_smallloop_mul #1!%
{%
    \expandafter\XINT_smallmul
    \the\numexpr
        \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_fac_loop_exit #1!#2;!#3{#3#2;!}%
```

4.48 \XINT_useiimessage

1.2o

```
\def\XINT_useiimessage #1% used in LaTeX only
{%
    \XINT_ifFlagRaised {#1}%
    {\@backslashchar#1
     (load xintfrac or use \@backslashchar xintii\xint_gobble_iv#1!)\MessageBreak}%
    {}%
}%
\XINT_restorecatcodes_endinput%
```

5 Package xint implementation

With release 1.1 the core arithmetic routines \xintiiAdd, \xintiiSub, \xintiiMul, \xintiiQuo, \xintiiPow were separated to be the main component of the then new xintcore.

At 1.3 the macros deprecated at 1.2o got all removed.

1.3b adds randomness related macros.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
```

```
\catcode58=12   % :
\let\z\endgroup
\expandafter\let\expandafter\x\csname ver@xint.sty\endcsname
\expandafter\let\expandafter\w\csname ver@xintcore.sty\endcsname
\expandafter
  \ifx\csname PackageInfo\endcsname\relax
    \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \else
    \def\y#1#2{\PackageInfo{#1}{#2}}%
  \fi
\expandafter
\ifx\csname numexpr\endcsname\relax
   \y{xint}{\numexpr not available, aborting input}%
   \aftergroup\endinput
\else
  \ifx\x\relax   % plain-TeX, first loading of xintcore.sty
    \ifx\w\relax % but xintkernel.sty not yet loaded.
       \def\z{\endgroup\input xintcore.sty\relax}%
    \fi
  \else
    \def\empty {}%
    \ifx\x\empty % LaTeX, first loading,
    % variable is initialized, but \ProvidesPackage not yet seen
        \ifx\w\relax % xintcore.sty not yet loaded.
          \def\z{\endgroup\RequirePackage{xintcore}}%
        \fi
    \else
      \aftergroup\endinput % xint already loaded.
    \fi
  \fi
\fi
\z%
\XINTsetupcatcodes% defined in xintkernel.sty (loaded by xintcore.sty)
```

5.1 Package identification

```
\XINT_providespackage
\ProvidesPackage{xint}%
  [2021/03/29 v1.4d Expandable operations on big integers (JFB)]%
```

5.2 More token management

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

5.3 (WIP) A constant needed by \xintRandomDigits et al.

```
\ifdefined\xint_texuniformdeviate
  \unless\ifdefined\xint_c_nine_x^viii
      \csname newcount\endcsname\xint_c_nine_x^viii
      \xint_c_nine_x^viii 900000000
```

```
  \fi
\fi
```

5.4 \xintLen, \xintiLen

\xintLen gets extended to fractions by xintfrac.sty: A/B is given length len(A)+len(B)-1 (somewhat arbitrary). It applies \xintNum to its argument. A minus sign is accepted and ignored.
  For parallelism with \xintiNum/\xintNum, 1.2o defines \xintiLen.
  \xintLen gets redefined by xintfrac.

```
\def\xintiLen {\romannumeral0\xintilen }%
\def\xintilen #1{\def\xintilen ##1%
{%
    \expandafter#1\the\numexpr
    \expandafter\XINT_len_fork\romannumeral0\xintinum{##1}%
      \xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye\relax
}}\xintilen{ }%
\def\xintLen {\romannumeral0\xintlen }%
\let\xintlen\xintilen


\def\XINT_len_fork #1%
{%
    \expandafter\XINT_length_loop\xint_UDsignfork#1{}-#1\krof
}%
```

5.5 \xintiiLogTen

1.3e. Support for ilog10() function in \xintiiexpr. See \XINTiLogTen in xintfrac.sty which also currently uses -"7FFF8000 as value if input is zero.

```
\def\xintiiLogTen {\the\numexpr\xintiilogten }%
\def\xintiilogten #1%
{%
    \expandafter\XINT_iilogten\romannumeral`&&@#1%
      \xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    \relax
}%
\def\XINT_iilogten #1{\if#10-"7FFF8000\fi -1+%
                      \expandafter\XINT_length_loop\xint_UDsignfork#1{}-#1\krof}%
```

5.6 \xintReverseDigits

1.2.
  This puts digits in reverse order, not suppressing leading zeros after reverse. Despite lacking the "ii" in its name, it does not apply \xintNum to its argument (contrarily to \xintLen, this is not very coherent).
  1.2l variant is robust against non terminated \the\numexpr input.
  This macro is currently not used elsewhere in xint code.

```
\def\xintReverseDigits {\romannumeral0\xintreversedigits }%
\def\xintreversedigits #1%
{%
    \expandafter\XINT_revdigits\romannumeral`&&@#1%
     {\XINT_microrevsep_end\W}\XINT_microrevsep_end
      \XINT_microrevsep_end\XINT_microrevsep_end
      \XINT_microrevsep_end\XINT_microrevsep_end
      \XINT_microrevsep_end\XINT_microrevsep_end\XINT_microrevsep_end\Z
    1\Z!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
}%
\def\XINT_revdigits #1%
{%
    \xint_UDsignfork
      #1{\expandafter-\romannumeral0\XINT_revdigits_a}%
       -{\XINT_revdigits_a #1}%
    \krof
}%
\def\XINT_revdigits_a
{%
    \expandafter\XINT_revdigits_b\expandafter{\expandafter}%
    \the\numexpr\XINT_microrevsep
}%
\def\XINT_microrevsep #1#2#3#4#5#6#7#8#9%
{%
    1#9#8#7#6#5#4#3#2#1\expandafter!\the\numexpr\XINT_microrevsep
}%
\def\XINT_microrevsep_end #1\W #2\expandafter #3\Z{\relax#2!}%
\def\XINT_revdigits_b #11#2!1#3!1#4!1#5!1#6!1#7!1#8!1#9!%
{%
    \xint_gob_til_R #9\XINT_revdigits_end\R
                        \XINT_revdigits_b {#9#8#7#6#5#4#3#2#1}%
}%
\def\XINT_revdigits_end#1{%
\def\XINT_revdigits_end\R\XINT_revdigits_b ##1##2\W
   {\expandafter#1\xint_gob_til_Z ##1}%
}\XINT_revdigits_end{ }%
\let\xintRev\xintReverseDigits
```

5.7 \xintiiE

Originally was used in \xintiiexpr. Transferred from xintfrac for 1.1. Code rewritten for 1.2i. \xintiiE{x}{e} extends x with e zeroes if e is positive and simply outputs x if e is zero or negative. Attention, le comportement pour e < 0 ne doit pas être modifié car \xintMod et autres macros en dépendent.

```
\def\xintiiE {\romannumeral0\xintiie }%
\def\xintiie #1#2%
   {\expandafter\XINT_iie_fork\the\numexpr #2\expandafter.\romannumeral`&&@#1;}%
\def\XINT_iie_fork #1%
{%
    \xint_UDsignfork
      #1\XINT_iie_neg
       -\XINT_iie_a
```

```
    \krof #1%
}%
```

le #2 a le bon pattern terminé par ; #1=0 est OK pour \XINT_rep.

```
\def\XINT_iie_a #1.%
 {\expandafter\XINT_dsx_append\romannumeral\XINT_rep #1\endcsname 0.}%
\def\XINT_iie_neg #1.#2;{ #2}%
```

5.8 \xintDecSplit

DECIMAL SPLIT
The macro \xintDecSplit {x}{A} cuts A which is composed of digits (leading zeroes ok, but no sign) (*) into two (each possibly empty) pieces L and R. The concatenation LR always reproduces A.
The position of the cut is specified by the first argument x. If x is zero or positive the cut location is x slots to the left of the right end of the number. If x becomes equal to or larger than the length of the number then L becomes empty. If x is negative the location of the cut is |x| slots to the right of the left end of the number.
(*) versions earlier than 1.2i first replaced A with its absolute value. This is not the case anymore. This macro should NOT be used for A with a leading sign (+ or -).
Entirely rewritten for 1.2i (2016/12/11).
Attention: \xintDecSplit not robust against non terminated second argument.

```
\def\xintDecSplit {\romannumeral0\xintdecsplit }%
\def\xintdecsplit #1#2%
{%
    \expandafter\XINT_split_finish
    \romannumeral0\expandafter\XINT_split_xfork
    \the\numexpr #1\expandafter.\romannumeral`&&@#2%
    \xint_bye2345678\xint_bye..%
}%
\def\XINT_split_finish  #1.#2.{{#1}{#2}}%


\def\XINT_split_xfork #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_split_zerosplit
      0#1\XINT_split_fromleft
       0-{\XINT_split_fromright #1}%
    \krof
}%
\def\XINT_split_zerosplit .#1\xint_bye#2\xint_bye..{ #1..}%
\def\XINT_split_fromleft
    {\expandafter\XINT_split_fromleft_a\the\numexpr\xint_c_viii-}%
\def\XINT_split_fromleft_a #1%
{%
    \xint_UDsignfork
      #1\XINT_split_fromleft_b
       -{\XINT_split_fromleft_end_a #1}%
    \krof
}%
\def\XINT_split_fromleft_b #1.#2#3#4#5#6#7#8#9%
{%
```

```
    \expandafter\XINT_split_fromleft_clean
    \the\numexpr1#2#3#4#5#6#7#8#9\expandafter
    \XINT_split_fromleft_a\the\numexpr\xint_c_viii-#1.%
}%
\def\XINT_split_fromleft_end_a #1.%
{%
    \expandafter\XINT_split_fromleft_clean
    \the\numexpr1\csname XINT_split_fromleft_end#1\endcsname
}%
\def\XINT_split_fromleft_clean 1{ }%
\expandafter\def\csname XINT_split_fromleft_end7\endcsname #1%
   {#1\XINT_split_fromleft_end_b}%
\expandafter\def\csname XINT_split_fromleft_end6\endcsname #1#2%
   {#1#2\XINT_split_fromleft_end_b}%
\expandafter\def\csname XINT_split_fromleft_end5\endcsname #1#2#3%
   {#1#2#3\XINT_split_fromleft_end_b}%
\expandafter\def\csname XINT_split_fromleft_end4\endcsname #1#2#3#4%
   {#1#2#3#4\XINT_split_fromleft_end_b}%
\expandafter\def\csname XINT_split_fromleft_end3\endcsname #1#2#3#4#5%
   {#1#2#3#4#5\XINT_split_fromleft_end_b}%
\expandafter\def\csname XINT_split_fromleft_end2\endcsname #1#2#3#4#5#6%
   {#1#2#3#4#5#6\XINT_split_fromleft_end_b}%
\expandafter\def\csname XINT_split_fromleft_end1\endcsname #1#2#3#4#5#6#7%
   {#1#2#3#4#5#6#7\XINT_split_fromleft_end_b}%
\expandafter\def\csname XINT_split_fromleft_end0\endcsname #1#2#3#4#5#6#7#8%
   {#1#2#3#4#5#6#7#8\XINT_split_fromleft_end_b}%


\def\XINT_split_fromleft_end_b #1\xint_bye#2\xint_bye.{.#1}% puis .
\def\XINT_split_fromright #1.#2\xint_bye
{%
    \expandafter\XINT_split_fromright_a
    \the\numexpr#1-\numexpr\XINT_length_loop
    #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    .#2\xint_bye
}%


\def\XINT_split_fromright_a #1%
{%
    \xint_UDsignfork
      #1\XINT_split_fromleft
       -\XINT_split_fromright_Lempty
    \krof
}%
\def\XINT_split_fromright_Lempty #1.#2\xint_bye#3..{.#2.}%
```

5.9 \xintDecSplitL

```
\def\xintDecSplitL {\romannumeral0\xintdecsplitl }%
\def\xintdecsplitl #1#2%
{%
```

```
    \expandafter\XINT_splitl_finish
    \romannumeral0\expandafter\XINT_split_xfork
    \the\numexpr #1\expandafter.\romannumeral`&&@#2%
    \xint_bye2345678\xint_bye..%
}%
\def\XINT_splitl_finish #1.#2.{ #1}%
```

5.10 \xintDecSplitR

```
\def\xintDecSplitR {\romannumeral0\xintdecsplitr }%
\def\xintdecsplitr #1#2%
{%
    \expandafter\XINT_splitr_finish
    \romannumeral0\expandafter\XINT_split_xfork
    \the\numexpr #1\expandafter.\romannumeral`&&@#2%
    \xint_bye2345678\xint_bye..%
}%
\def\XINT_splitr_finish #1.#2.{ #2}%
```

5.11 \xintDSHr

```
DECIMAL SHIFTS \xintDSH {x}{A}
si x <= 0, fait A -> A.10^(|x|). si x > 0, et A >=0, fait A -> quo(A,10^(x))
si x > 0, et A < 0, fait A -> -quo(-A,10^(x))
(donc pour x > 0 c'est comme DSR itéré x fois)
\xintDSHr donne le `reste' (si x<=0 donne zéro).
  Badly named macros.
  Rewritten for 1.2i, this was old code and \xintDSx has changed interface.
```

```
\def\xintDSHr {\romannumeral0\xintdshr }%


\def\xintdshr #1#2%
{%


    \expandafter\XINT_dshr_fork\the\numexpr#1\expandafter.\romannumeral`&&@#2;%
}%
\def\XINT_dshr_fork #1%
{%
    \xint_UDzerominusfork
      0#1\XINT_dshr_xzeroorneg
      #1-\XINT_dshr_xzeroorneg
       0-\XINT_dshr_xpositive
    \krof #1%
}%
\def\XINT_dshr_xzeroorneg #1;{ 0}%
\def\XINT_dshr_xpositive
{%


    \expandafter\xint_stop_atsecondoftwo\romannumeral0\XINT_dsx_xisPos
}%
```

5.12 \xintDSH

```
\def\xintDSH {\romannumeral0\xintdsh }%
\def\xintdsh #1#2%
{%

    \expandafter\XINT_dsh_fork\the\numexpr#1\expandafter.\romannumeral`&&@#2;%
}%
\def\XINT_dsh_fork #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_dsh_xiszero

      0#1\XINT_dsx_xisNeg_checkA
       0-{\XINT_dsh_xisPos #1}%
    \krof
}%
\def\XINT_dsh_xiszero #1.#2;{ #2}%
\def\XINT_dsh_xisPos
{%
  \expandafter\xint_stop_atfirstoftwo\romannumeral0\XINT_dsx_xisPos
}%
```

5.13 \xintDSx

```
--> Attention le cas x=0 est traité dans la même catégorie que x > 0 <--
   si x < 0, fait A -> A.10^(|x|)
   si x >=  0, et A >=0, fait A -> {quo(A,10^(x))}{rem(A,10^(x))}
   si x >=  0, et A < 0, d'abord on calcule {quo(-A,10^(x))}{rem(-A,10^(x))}
      puis, si le premier n'est pas nul on lui donne le signe -
            si le premier est nul on donne le signe - au second.
On peut donc toujours reconstituer l'original A par 10^x Q \pm R où il faut prendre le signe plus
si Q est positif ou nul et le signe moins si Q est strictement négatif.
  Rewritten for 1.2i, this was old code.
```

```
\def\xintDSx {\romannumeral0\xintdsx }%
\def\xintdsx #1#2%
{%

    \expandafter\XINT_dsx_fork\the\numexpr#1\expandafter.\romannumeral`&&@#2;%
}%
\def\XINT_dsx_fork #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_dsx_xisZero
      0#1\XINT_dsx_xisNeg_checkA
       0-{\XINT_dsx_xisPos #1}%
    \krof
}%
\def\XINT_dsx_xisZero #1.#2;{{#2}{0}}%
\def\XINT_dsx_xisNeg_checkA #1.#2%
{%
    \xint_gob_til_zero #2\XINT_dsx_xisNeg_Azero 0%
```

```
    \expandafter\XINT_dsx_append\romannumeral\XINT_rep #1\endcsname 0.#2%
}%
\def\XINT_dsx_xisNeg_Azero #1;{ 0}%

\def\XINT_dsx_addzeros #1%
   {\expandafter\XINT_dsx_append\romannumeral\XINT_rep#1\endcsname0.}%

\def\XINT_dsx_addzerosnofuss #1%
   {\expandafter\XINT_dsx_append\romannumeral\xintreplicate{#1}0.}%
\def\XINT_dsx_append #1.#2;{ #2#1}%

\def\XINT_dsx_xisPos #1.#2%
{%
    \xint_UDzerominusfork
      #2-\XINT_dsx_AisZero
      0#2\XINT_dsx_AisNeg
       0-\XINT_dsx_AisPos
    \krof #1.#2%
}%
\def\XINT_dsx_AisZero #1;{{0}{0}}%
\def\XINT_dsx_AisNeg #1.-#2;%
{%
    \expandafter\XINT_dsx_AisNeg_checkiffirstempty
    \romannumeral0\XINT_split_xfork #1.#2\xint_bye2345678\xint_bye..%
}%

\def\XINT_dsx_AisNeg_checkiffirstempty #1%
{%
    \xint_gob_til_dot #1\XINT_dsx_AisNeg_finish_zero.%
    \XINT_dsx_AisNeg_finish_notzero #1%
}%
\def\XINT_dsx_AisNeg_finish_zero.\XINT_dsx_AisNeg_finish_notzero.#1.%
{%
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {-#1}}{0}%
}%
\def\XINT_dsx_AisNeg_finish_notzero #1.#2.%
{%
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {#2}}{-#1}%
}%

\def\XINT_dsx_AisPos #1.#2;%
{%
    \expandafter\XINT_dsx_AisPos_finish
    \romannumeral0\XINT_split_xfork #1.#2\xint_bye2345678\xint_bye..%
}%

\def\XINT_dsx_AisPos_finish #1.#2.%
{%
```

```
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {#2}}%
                 {\romannumeral0\XINT_num {#1}}%
}%
\def\XINT_dsx_end #1#2{\expandafter{#2}{#1}}%
```

5.14 \xintiiEq

no \xintiieq.

```
\def\xintiiEq #1#2{\romannumeral0\xintiiifeq{#1}{#2}{1}{0}}%
```

5.15 \xintiiNotEq

Pour xintexpr. Pas de version en lowercase.

```
\def\xintiiNotEq #1#2{\romannumeral0\xintiiifeq {#1}{#2}{0}{1}}%
```

5.16 \xintiiGeq

PLUS GRAND OU ÉGAL attention compare les **valeurs absolues**
  1.2l made \xintiiGeq robust against non terminated items.
  1.2l rewrote \xintiiCmp, but forgot to handle \xintiiGeq too. Done at 1.2m.
  This macro should have been called \xintGEq for example.

```
\def\xintiiGeq   {\romannumeral0\xintiigeq }%
\def\xintiigeq #1{\expandafter\XINT_iigeq\romannumeral`&&@#1\xint:}%
\def\XINT_iigeq #1#2\xint:#3%
{%
    \expandafter\XINT_geq_fork\expandafter #1\romannumeral`&&@#3\xint:#2\xint:
}%


\def\XINT_geq #1#2\xint:#3%
{%
    \expandafter\XINT_geq_fork\expandafter #1\romannumeral0\xintnum{#3}\xint:#2\xint:
}%
\def\XINT_geq_fork #1#2%
{%
    \xint_UDzerofork
      #1\XINT_geq_firstiszero
      #2\XINT_geq_secondiszero
       0{}%
    \krof
    \xint_UDsignsfork
         #1#2\XINT_geq_minusminus
          #1-\XINT_geq_minusplus
          #2-\XINT_geq_plusminus
           --\XINT_geq_plusplus
    \krof #1#2%
}%
\def\XINT_geq_firstiszero  #1\krof 0#2#3\xint:#4\xint:
                              {\xint_UDzerofork #2{ 1}0{ 0}\krof }%
```

```
\def\XINT_geq_secondiszero #1\krof #20#3\xint:#4\xint:{ 1}%
\def\XINT_geq_plusminus    #1-{\XINT_geq_plusplus #1{}}%
\def\XINT_geq_minusplus    -#1{\XINT_geq_plusplus {}#1}%
\def\XINT_geq_minusminus    --{\XINT_geq_plusplus {}{}}%
\def\XINT_geq_plusplus
   {\expandafter\XINT_geq_finish\romannumeral0\XINT_cmp_plusplus}%
\def\XINT_geq_finish #1{\if-#1\expandafter\XINT_geq_no
                        \else\expandafter\XINT_geq_yes\fi}%
\def\XINT_geq_no 1{ 0}%
\def\XINT_geq_yes { 1}%
```

5.17 \xintiiGt

```
\def\xintiiGt #1#2{\romannumeral0\xintiiifgt{#1}{#2}{1}{0}}%
```

5.18 \xintiiLt

```
\def\xintiiLt #1#2{\romannumeral0\xintiiiflt{#1}{#2}{1}{0}}%
```

5.19 \xintiiGtorEq

```
\def\xintiiGtorEq #1#2{\romannumeral0\xintiiiflt {#1}{#2}{0}{1}}%
```

5.20 \xintiiLtorEq

```
\def\xintiiLtorEq #1#2{\romannumeral0\xintiiifgt {#1}{#2}{0}{1}}%
```

5.21 \xintiiIsZero

1.09a. restyled in 1.09i. 1.1 adds \xintiiIsZero, etc... for optimization in \xintexpr

```
\def\xintiiIsZero {\romannumeral0\xintiiiszero }%
\def\xintiiiszero #1{\if0\xintiiSgn{#1}\xint_afterfi{ 1}\else\xint_afterfi{ 0}\fi}%
```

5.22 \xintiiIsNotZero

1.09a. restyled in 1.09i. 1.1 adds \xintiiIsZero, etc... for optimization in \xintexpr

```
\def\xintiiIsNotZero {\romannumeral0\xintiiisnotzero }%
\def\xintiiisnotzero
          #1{\if0\xintiiSgn{#1}\xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi}%
```

5.23 \xintiiIsOne

Added in 1.03. 1.09a defines \xintIsOne. 1.1a adds \xintiiIsOne.
\XINT_isOne rewritten for 1.2g. Works with expanded strict integers, positive or negative.

```
\def\xintiiIsOne {\romannumeral0\xintiiisone }%
\def\xintiiisone #1{\expandafter\XINT_isone\romannumeral`&&@#1XY}%
\def\XINT_isone #1#2#3Y%
{%
    \unless\if#2X\xint_dothis{ 0}\fi
    \unless\if#11\xint_dothis{ 0}\fi
    \xint_orthat{ 1}%
}%
\def\XINT_isOne #1{\XINT_is_One#1XY}%
```

```
\def\XINT_is_One #1#2#3Y%
{%
    \unless\if#2X\xint_dothis0\fi
    \unless\if#11\xint_dothis0\fi
    \xint_orthat1%
}%
```

5.24 \xintiiOdd

\xintOdd is needed for the xintexpr-essions even() and odd() functions (and also by \xintNewExpr).

```
\def\xintiiOdd {\romannumeral0\xintiiodd }%
\def\xintiiodd #1%
{%
    \ifodd\xintLDg{#1} %<- intentional space
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ 0}%
    \fi
}%
```

5.25 \xintiiEven

```
\def\xintiiEven {\romannumeral0\xintiieven }%
\def\xintiieven #1%
{%
    \ifodd\xintLDg{#1} %<- intentional space
        \xint_afterfi{ 0}%
    \else
        \xint_afterfi{ 1}%
    \fi
}%
```

5.26 \xintiiMON

MINUS ONE TO THE POWER N

```
\def\xintiiMON {\romannumeral0\xintiimon }%
\def\xintiimon #1%
{%
    \ifodd\xintLDg {#1} %<- intentional space
        \xint_afterfi{ -1}%
    \else
        \xint_afterfi{ 1}%
    \fi
}%
```

5.27 \xintiiMMON

MINUS ONE TO THE POWER N-1

```
\def\xintiiMMON {\romannumeral0\xintiimmon }%
\def\xintiimmon #1%
{%
```

```
    \ifodd\xintLDg {#1} %<- intentional space
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ -1}%
    \fi
}%
```

5.28 \xintSgnFork

Expandable three-way fork added in 1.07. The argument #1 must expand to non-self-ending -1,0 or 1. 1.09i with _thenstop (now _stop_at...).

```
\def\xintSgnFork {\romannumeral0\xintsgnfork }%
\def\xintsgnfork #1%
{%
    \ifcase #1 \expandafter\xint_stop_atsecondofthree
            \or\expandafter\xint_stop_atthirdofthree
          \else\expandafter\xint_stop_atfirstofthree
    \fi
}%
```

5.29 \xintiiifSgn

Expandable three-way fork added in 1.09a. Branches expandably depending on whether <0, =0, >0. Choice of branch guaranteed in two steps.

1.09i has \xint_firstofthreeafterstop (now \xint_stop_atfirstofthree) etc for faster expansion.

1.1 adds \xintiiifSgn for optimization in xintexpr-essions. Should I move them to xintcore? (for bnumexpr)

```
\def\xintiiifSgn {\romannumeral0\xintiiifsgn }%
\def\xintiiifsgn #1%
{%
    \ifcase \xintiiSgn{#1}
               \expandafter\xint_stop_atsecondofthree
            \or\expandafter\xint_stop_atthirdofthree
          \else\expandafter\xint_stop_atfirstofthree
    \fi
}%
```

5.30 \xintiiifCmp

1.09e \xintifCmp {n}{m}{if n<m}{if n=m}{if n>m}. 1.1a adds ii variant

```
\def\xintiiifCmp {\romannumeral0\xintiiifcmp }%
\def\xintiiifcmp #1#2%
{%
    \ifcase\xintiiCmp {#1}{#2}
               \expandafter\xint_stop_atsecondofthree
            \or\expandafter\xint_stop_atthirdofthree
          \else\expandafter\xint_stop_atfirstofthree
    \fi
}%
```

5.31 \xintiiifEq

1.09a \xintifEq {n}{m}{YES if n=m}{NO if n<>m}. 1.1a adds ii variant

```
\def\xintiiifEq {\romannumeral0\xintiiifeq }%
\def\xintiiifeq #1#2%
{%
    \if0\xintiiCmp{#1}{#2}%
                \expandafter\xint_stop_atfirstoftwo
          \else\expandafter\xint_stop_atsecondoftwo
    \fi
}%
```

5.32 \xintiiifGt

1.09a \xintifGt {n}{m}{YES if n>m}{NO if n<=m}. 1.1a adds ii variant

```
\def\xintiiifGt {\romannumeral0\xintiiifgt }%
\def\xintiiifgt #1#2%
{%
    \if1\xintiiCmp{#1}{#2}%
                \expandafter\xint_stop_atfirstoftwo
          \else\expandafter\xint_stop_atsecondoftwo
    \fi
}%
```

5.33 \xintiiifLt

1.09a \xintifLt {n}{m}{YES if n<m}{NO if n>=m}. Restyled in 1.09i. 1.1a adds ii variant

```
\def\xintiiifLt {\romannumeral0\xintiiiflt }%
\def\xintiiiflt #1#2%
{%
    \ifnum\xintiiCmp{#1}{#2}<\xint_c_
          \expandafter\xint_stop_atfirstoftwo
    \else \expandafter\xint_stop_atsecondoftwo
    \fi
}%
```

5.34 \xintiiifZero

Expandable two-way fork added in 1.09a. Branches expandably depending on whether the argument is zero (branch A) or not (branch B). 1.09i restyling. By the way it appears (not thoroughly tested, though) that \if tests are faster than \ifnum tests. 1.1 adds ii versions.
1.2o deprecates \xintifZero.

```
\def\xintiiifZero {\romannumeral0\xintiiifzero }%
\def\xintiiifzero #1%
{%
    \if0\xintiiSgn{#1}%
       \expandafter\xint_stop_atfirstoftwo
    \else
       \expandafter\xint_stop_atsecondoftwo
```

```
    \fi
}%
```

5.35 \xintiiifNotZero

```
\def\xintiiifNotZero {\romannumeral0\xintiiifnotzero }%
\def\xintiiifnotzero #1%
{%
    \if0\xintiiSgn{#1}%
       \expandafter\xint_stop_atsecondoftwo
    \else
       \expandafter\xint_stop_atfirstoftwo
    \fi
}%
```

5.36 \xintiiifOne

added in 1.09i. 1.1a adds \xintiiifOne.

```
\def\xintiiifOne {\romannumeral0\xintiiifone }%
\def\xintiiifone #1%
{%
    \if1\xintiiIsOne{#1}%
       \expandafter\xint_stop_atfirstoftwo
    \else
       \expandafter\xint_stop_atsecondoftwo
    \fi
}%
```

5.37 \xintiiifOdd

1.09e. Restyled in 1.09i. 1.1a adds \xintiiifOdd.

```
\def\xintiiifOdd {\romannumeral0\xintiiifodd }%
\def\xintiiifodd #1%
{%
    \if\xintiiOdd{#1}1%
       \expandafter\xint_stop_atfirstoftwo
    \else
       \expandafter\xint_stop_atsecondoftwo
    \fi
}%
```

5.38 \xintifTrueAelseB, \xintifFalseAelseB

1.09i. 1.2i has removed deprecated \xintifTrueFalse, \xintifTrue.
1.2o uses \xintiiifNotZero, see comments to \xintAND etc... This will work fine with arguments being nested xintfrac.sty macros, without the overhead of \xintNum or \xintRaw parsing.

```
\def\xintifTrueAelseB {\romannumeral0\xintiiifnotzero}%
\def\xintifFalseAelseB{\romannumeral0\xintiiifzero}%
```

5.39 \xintIsTrue, \xintIsFalse

1.09c. Suppressed at 1.2o. They seem not to have been documented, fortunately.

```
%\let\xintIsTrue \xintIsNotZero
%\let\xintIsFalse\xintIsZero
```

5.40 \xintNOT

1.09c. But it should have been called \xintNOT, not \xintNot. Former denomination deprecated at 1.2o. Besides, the macro is now defined as ii-type.

```
\def\xintNOT{\romannumeral0\xintiiiszero}%
```

5.41 \xintAND, \xintOR, \xintXOR

Added with 1.09a. But they used \xintSgn, etc... rather than \xintiiSgn. This brings \xintNum overhead which is not really desired, and which is not needed for use by xintexpr.sty. At 1.2o I modify them to use only ii macros. This is enough for sign or zeroness even for xintfrac format, as manipulated inside the \xintexpr. Big hesitation whether there should be however \xintiiAND outputting 1 or 0 versus an \xintAND outputting 1[0] versus 0[0] for example.

```
\def\xintAND {\romannumeral0\xintand }%
\def\xintand #1#2{\if0\xintiiSgn{#1}\expandafter\xint_firstoftwo
                              \else\expandafter\xint_secondoftwo\fi
                  { 0}{\xintiiisnotzero{#2}}}%
\def\xintOR {\romannumeral0\xintor }%
\def\xintor #1#2{\if0\xintiiSgn{#1}\expandafter\xint_firstoftwo
                             \else\expandafter\xint_secondoftwo\fi
                 {\xintiiisnotzero{#2}}{ 1}}%
\def\xintXOR {\romannumeral0\xintxor }%
\def\xintxor #1#2{\if\xintiiIsZero{#1}\xintiiIsZero{#2}%
                     \xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi }%
```

5.42 \xintANDof

New with 1.09a. \xintANDof works also with an empty list. Empty items however are not accepted.
1.2l made \xintANDof robust against non terminated items.
1.2o's \xintifTrueAelseB is now an ii macro, actually.
1.4. This macro as well as ORof and XORof were formally not used by xintexpr, which uses comma separated items, but at 1.4 xintexpr uses braced items. And the macros here got slightly refactored and \XINT_ANDof added for usage by xintexpr and the NewExpr hook. For some random reason I decided to use ^ as delimiter this has to do that other macros in xintfrac in same family (such as \xintGCDof, \xintSum) also use \xint: internally and although not strictly needed having two separate ones clarifies.

```
\def\xintANDof  {\romannumeral0\xintandof }%
\def\xintandof  #1{\expandafter\XINT_andof\romannumeral`&&@#1^}%
\def\XINT_ANDof {\romannumeral0\XINT_andof}%
\def\XINT_andof #1%
{%
    \xint_gob_til_^ #1\XINT_andof_yes ^%
    \xintiiifNotZero{#1}\XINT_andof\XINT_andof_no
```

```
}%
\def\XINT_andof_no  #1^{ 0}%
\def\XINT_andof_yes ^#1\XINT_andof_no{ 1}%
```

5.43 \xintORof

New with 1.09a. Works also with an empty list. Empty items however are not accepted.
1.2l made \xintORof robust against non terminated items.
Refactored at 1.4.

```
\def\xintORof  {\romannumeral0\xintorof }%
\def\xintorof  #1{\expandafter\XINT_orof\romannumeral`&&@#1^}%
\def\XINT_ORof {\romannumeral0\XINT_orof}%
\def\XINT_orof #1%
{%
    \xint_gob_til_^ #1\XINT_orof_no ^%
    \xintiiifNotZero{#1}\XINT_orof_yes\XINT_orof
}%
\def\XINT_orof_yes#1^{ 1}%
\def\XINT_orof_no ^#1\XINT_orof{ 0}%
```

5.44 \xintXORof

New with 1.09a. Works with an empty list, too. Empty items however are not accepted. \XINT_xorof_c more efficient in 1.09i.
1.2l made \xintXORof robust against non terminated items.
Refactored at 1.4 to use \numexpr (or an \ifnum). I have not tested if more efficient or not or if one can do better without \the. \XINT_XORof for xintexpr matters.

```
\def\xintXORof  {\romannumeral0\xintxorof }%
\def\xintxorof  #1{\expandafter\XINT_xorof\romannumeral`&&@#1^}%
\def\XINT_XORof {\romannumeral0\XINT_xorof}%
\def\XINT_xorof {\if1\the\numexpr\XINT_xorof_a}%
\def\XINT_xorof_a #1%
{%
    \xint_gob_til_^ #1\XINT_xorof_e ^%
    \xintiiifNotZero{#1}{-}{}\XINT_xorof_a
}%
\def\XINT_xorof_e ^#1\XINT_xorof_a
   {1\relax\xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi}%
```

5.45 \xintiiMax

At 1.2m, a long-standing bug was fixed: \xintiiMax had the overhead of applying \xintNum to its arguments due to use of a sub-macro of \xintGeq code to which this overhead was added at some point.
And on this occasion I reduced even more number of times input is grabbed.

```
\def\xintiiMax {\romannumeral0\xintiimax }%
\def\xintiimax #1%
{%
    \expandafter\xint_iimax \romannumeral`&&@#1\xint:
}%
\def\xint_iimax #1\xint:#2%
```

```
{%
    \expandafter\XINT_max_fork\romannumeral`&&@#2\xint:#1\xint:
}%
```

#3#4 vient du *premier*, #1#2 vient du *second*. I have renamed the sub-macros at 1.2m because the terminology was quite counter-intuitive; there was no bug, but still.

```
\def\XINT_max_fork #1#2\xint:#3#4\xint:
{%
    \xint_UDsignsfork
          #1#3\XINT_max_minusminus  % A < 0, B < 0
           #1-\XINT_max_plusminus   % B < 0, A >= 0
           #3-\XINT_max_minusplus   % A < 0, B >= 0
            --{\xint_UDzerosfork
                      #1#3\XINT_max_zerozero % A = B = 0
                       #10\XINT_max_pluszero % B = 0, A > 0
                       #30\XINT_max_zeroplus % A = 0, B > 0
                        00\XINT_max_plusplus % A, B > 0
                      \krof }%
    \krof
    #3#1#2\xint:#4\xint:
      \expandafter\xint_stop_atfirstoftwo
    \else
      \expandafter\xint_stop_atsecondoftwo
    \fi
    {#3#4}{#1#2}%
}%
```

Refactored at 1.2m for avoiding grabbing arguments. Position of inputs shared with iiCmp and iiGeq code.

```
\def\XINT_max_zerozero  #1\fi{\xint_stop_atfirstoftwo }%
\def\XINT_max_zeroplus  #1\fi{\xint_stop_atsecondoftwo }%
\def\XINT_max_pluszero  #1\fi{\xint_stop_atfirstoftwo }%
\def\XINT_max_minusplus #1\fi{\xint_stop_atsecondoftwo }%
\def\XINT_max_plusminus #1\fi{\xint_stop_atfirstoftwo }%
\def\XINT_max_plusplus
{%
    \if1\romannumeral0\XINT_geq_plusplus
}%
```

Premier des testés |A|=-A, second est |B|=-B. On veut le max(A,B), c'est donc A si |A|<|B| (ou |A|=|B|, mais peu importe alors). Donc on peut faire cela avec \unless. Simple.

```
\def\XINT_max_minusminus --%
{%
    \unless\if1\romannumeral0\XINT_geq_plusplus{}{}%
}%
```

5.46 \xintiiMin

\xintnum added New with 1.09a. I add \xintiiMin in 1.1 and mark as deprecated \xintMin, renamed \xintiMin. \xintMin NOW REMOVED (1.2, as \xintMax, \xintMaxof), only provided by \xintfracnameimp.

At 1.2m, a long-standing bug was fixed: \xintiiMin had the overhead of applying \xintNum to its arguments due to use of a sub-macro of \xintGeq code to which this overhead was added at some point. And on this occasion I reduced even more number of times input is grabbed.

```
\def\xintiiMin {\romannumeral0\xintiimin }%
\def\xintiimin #1%
{%
    \expandafter\xint_iimin \romannumeral`&&@#1\xint:
}%
\def\xint_iimin #1\xint:#2%
{%
    \expandafter\XINT_min_fork\romannumeral`&&@#2\xint:#1\xint:
}%
\def\XINT_min_fork #1#2\xint:#3#4\xint:
{%
    \xint_UDsignsfork
          #1#3\XINT_min_minusminus  % A < 0, B < 0
           #1-\XINT_min_plusminus   % B < 0, A >= 0
           #3-\XINT_min_minusplus   % A < 0, B >= 0
            --{\xint_UDzerosfork
                      #1#3\XINT_min_zerozero % A = B = 0
                       #10\XINT_min_pluszero % B = 0, A > 0
                       #30\XINT_min_zeroplus % A = 0, B > 0
                        00\XINT_min_plusplus % A, B > 0
                      \krof }%
    \krof
    #3#1#2\xint:#4\xint:
      \expandafter\xint_stop_atsecondoftwo
    \else
      \expandafter\xint_stop_atfirstoftwo
    \fi
    {#3#4}{#1#2}%
}%
\def\XINT_min_zerozero  #1\fi{\xint_stop_atfirstoftwo }%
\def\XINT_min_zeroplus  #1\fi{\xint_stop_atfirstoftwo }%
\def\XINT_min_pluszero  #1\fi{\xint_stop_atsecondoftwo }%
\def\XINT_min_minusplus #1\fi{\xint_stop_atfirstoftwo }%
\def\XINT_min_plusminus #1\fi{\xint_stop_atsecondoftwo }%
\def\XINT_min_plusplus
{%
    \if1\romannumeral0\XINT_geq_plusplus
}%
\def\XINT_min_minusminus --%
{%
    \unless\if1\romannumeral0\XINT_geq_plusplus{}{}%
}%
```

5.47 \xintiiMaxof

New with 1.09a. 1.2 has NO MORE \xintMaxof, requires \xintfracname. 1.2a adds \xintiiMaxof, as \xintiiMaxof:csv is not public.
NOT compatible with empty list.
1.2l made \xintiiMaxof robust against non terminated items.

1.4 refactors code to allow empty argument. For usage by \xintiiexpr. Slight deterioration, will come back.

```
\def\xintiiMaxof {\romannumeral0\xintiimaxof }%
\def\xintiimaxof #1{\expandafter\XINT_iimaxof\romannumeral`&&@#1^}%
\def\XINT_iiMaxof{\romannumeral0\XINT_iimaxof}%
\def\XINT_iimaxof#1%
{%
    \xint_gob_til_^ #1\XINT_iimaxof_empty ^%
    \expandafter\XINT_iimaxof_loop\romannumeral`&&@#1\xint:
}%
\def\XINT_iimaxof_empty ^#1\xint:{ 0}%
\def\XINT_iimaxof_loop #1\xint:#2%
{%
    \xint_gob_til_^ #2\XINT_iimaxof_e ^%
    \expandafter\XINT_iimaxof_loop\romannumeral0\xintiimax{#1}{#2}\xint:
}%
\def\XINT_iimaxof_e ^#1\xintiimax #2#3\xint:{ #2}%
```

5.48 \xintiiMinof

1.09a. 1.2a adds \xintiiMinof which was lacking.
1.4 refactoring for \xintiiexpr matters.

```
\def\xintiiMinof {\romannumeral0\xintiiminof }%
\def\xintiiminof #1{\expandafter\XINT_iiminof\romannumeral`&&@#1^}%
\def\XINT_iiMinof{\romannumeral0\XINT_iiminof}%
\def\XINT_iiminof#1%
{%
    \xint_gob_til_^ #1\XINT_iiminof_empty ^%
    \expandafter\XINT_iiminof_loop\romannumeral`&&@#1\xint:
}%
\def\XINT_iiminof_empty ^#1\xint:{ 0}%
\def\XINT_iiminof_loop #1\xint:#2%
{%
    \xint_gob_til_^ #2\XINT_iiminof_e ^%
    \expandafter\XINT_iiminof_loop\romannumeral0\xintiimin{#1}{#2}\xint:
}%
\def\XINT_iiminof_e ^#1\xintiimin #2#3\xint:{ #2}%
```

5.49 \xintiiSum

\xintiiSum {{a}{b}...{z}} Refactored at 1.4 for matters initially related to xintexpr delimiter choice.

```
\def\xintiiSum {\romannumeral0\xintiisum }%
\def\xintiisum #1{\expandafter\XINT_iisum\romannumeral`&&@#1^}%
\def\XINT_iiSum{\romannumeral0\XINT_iisum}%
\def\XINT_iisum #1%
{%
    \expandafter\XINT_iisum_a\romannumeral`&&@#1\xint:
}%
\def\XINT_iisum_a #1%
```

```
{%
    \xint_gob_til_^ #1\XINT_iisum_empty ^%
    \XINT_iisum_loop #1%
}%
\def\XINT_iisum_empty ^#1\xint:{ 0}%
```

bad coding as it depends on internal conventions of \XINT_add_nfork

```
\def\XINT_iisum_loop #1#2\xint:#3%
{%
    \expandafter\XINT_iisum_loop_a
    \expandafter#1\romannumeral`&&@#3\xint:#2\xint:\xint:
}%
\def\XINT_iisum_loop_a #1#2%
{%
    \xint_gob_til_^ #2\XINT_iisum_loop_end ^%
    \expandafter\XINT_iisum_loop\romannumeral0\XINT_add_nfork #1#2%
}%
```

see previous comment!

```
\def\XINT_iisum_loop_end ^#1\XINT_add_nfork #2#3\xint:#4\xint:\xint:{ #2#4}%
```

5.50 \xintiiPrd

\xintiiPrd {{a}...{z}}
Macros renamed and refactored (slightly more macros here to supposedly bring micro-gain) at 1.4 to match changes in xintfrac of delimiter, in sync with some usage in xintexpr.
Contrarily to the xintfrac version \xintPrd, this one aborts as soon as it hits a zero value.

```
\def\xintiiPrd {\romannumeral0\xintiiprd }%
\def\xintiiprd #1{\expandafter\XINT_iiprd\romannumeral`&&@#1^}%
\def\XINT_iiPrd{\romannumeral0\XINT_iiprd}%
```

The above romannumeral caused f-expansion of the list argument. We f-expand below the first item and each successive items because we do not use \xintiiMul but jump directly into \XINT_mul_nfork.

```
\def\XINT_iiprd #1%
{%
    \expandafter\XINT_iiprd_a\romannumeral`&&@#1\xint:
}%
\def\XINT_iiprd_a #1%
{%
    \xint_gob_til_^ #1\XINT_iiprd_empty ^%
    \xint_gob_til_zero #1\XINT_iiprd_zero 0%
    \XINT_iiprd_loop #1%
}%
\def\XINT_iiprd_empty ^#1\xint:{ 1}%
\def\XINT_iiprd_zero  0#1^{ 0}%
```

bad coding as it depends on internal conventions of \XINT_mul_nfork

```
\def\XINT_iiprd_loop #1#2\xint:#3%
{%
    \expandafter\XINT_iiprd_loop_a
```

```
    \expandafter#1\romannumeral`&&@#3\xint:#2\xint:\xint:
}%
\def\XINT_iiprd_loop_a #1#2%
{%
    \xint_gob_til_^    #2\XINT_iiprd_loop_end ^%
    \xint_gob_til_zero #2\XINT_iiprd_zero 0%
    \expandafter\XINT_iiprd_loop\romannumeral0\XINT_mul_nfork #1#2%
}%
```

see previous comment!

```
\def\XINT_iiprd_loop_end ^#1\XINT_mul_nfork #2#3\xint:#4\xint:\xint:{ #2#4}%
```

5.51 \xintiiSquareRoot

First done with 1.08.
1.1 added \xintiiSquareRoot.
1.1a added \xintiiSqrtR.
1.2f (2016/03/01-02-03) has rewritten the implementation, the underlying mathematics remaining about the same. The routine is much faster for inputs having up to 16 digits (because it does it all with \numexpr directly now), and also much faster for very long inputs (because it now fetches only the needed new digits after the first 16 (or 17) ones, via the geometric sequence 16, then 32, then 64, etc...; earlier version did the computations with all remaining digits after a suitable starting point with correct 4 or 5 leading digits). Note however that the fetching of tokens is via intrinsically O(N^2) macros, hence inevitably inputs with thousands of digits start being treated less well.
Actually there is some room for improvements, one could prepare better input X for the upcoming treatment of fetching its digits by 16, then 32, then 64, etc...
Incidently, as \xintiiSqrt uses subtraction and subtraction was broken from 1.2 to 1.2c, then for another reason from 1.2c to 1.2f, it could get wrong in certain (relatively rare) cases. There was also a bug that made it unneedlessly slow for odd number of digits on input.
1.2f also modifies \xintFloatSqrt in xintfrac.sty which now has more code in common with here and benefits from the same speed improvements.
1.2k belatedly corrects the output to {1}{1} and not 11 when input is zero. As braces are used in all other cases they should have been used here too.
Also, 1.2k adds an \xintiSqrtR macro, for coherence as \xintiSqrt is defined (and mentioned in user manual.)

```
\def\xintiiSquareRoot {\romannumeral0\xintiisquareroot }%
\def\xintiisquareroot #1{\expandafter\XINT_sqrt_checkin\romannumeral`&&@#1\xint:}%
\def\XINT_sqrt_checkin #1%
{%
    \xint_UDzerominusfork
     #1-\XINT_sqrt_iszero
     0#1\XINT_sqrt_isneg
      0-\XINT_sqrt
    \krof #1%
}%
\def\XINT_sqrt_iszero #1\xint:{{1}{1}}%
\def\XINT_sqrt_isneg  #1\xint:{\XINT_signalcondition{InvalidOperation}{square
    root of negative: #1}{}{{0}{0}}}%
\def\XINT_sqrt #1\xint:
{%
```

```
    \expandafter\XINT_sqrt_start\romannumeral0\xintlength {#1}.#1.%
}%
\def\XINT_sqrt_start #1.%
{%
    \ifnum #1<\xint_c_x\xint_dothis\XINT_sqrt_small_a\fi
    \xint_orthat\XINT_sqrt_big_a #1.%
}%
\def\XINT_sqrt_small_a #1.{\XINT_sqrt_a #1.\XINT_sqrt_small_d }%
\def\XINT_sqrt_big_a   #1.{\XINT_sqrt_a #1.\XINT_sqrt_big_d   }%
\def\XINT_sqrt_a #1.%
{%
   \ifodd #1
     \expandafter\XINT_sqrt_bO
   \else
     \expandafter\XINT_sqrt_bE
   \fi
   #1.%
}%

\def\XINT_sqrt_bE #1.#2#3#4%
{%
    \XINT_sqrt_c {#3#4}#2{#1}#3#4%
}%

\def\XINT_sqrt_bO #1.#2#3%
{%
    \XINT_sqrt_c #3#2{#1}#3%
}%

\def\XINT_sqrt_c #1#2%
{%
    \expandafter #2%
    \the\numexpr \ifnum #1>\xint_c_ii
                 \ifnum #1>\xint_c_vi
                 \ifnum #1>12 \ifnum #1>20 \ifnum #1>30
                 \ifnum #1>42 \ifnum #1>56 \ifnum #1>72
                 \ifnum #1>90
      10\else 9\fi \else 8\fi \else 7\fi \else 6\fi \else 5\fi
        \else 4\fi \else 3\fi \else 2\fi \else 1\fi .%
}%

\def\XINT_sqrt_small_d #1.#2%
{%
   \expandafter\XINT_sqrt_small_e
   \the\numexpr #1\ifcase \numexpr #2/\xint_c_ii-\xint_c_i\relax
                    \or 0\or 00\or 000\or 0000\fi .%
}%

\def\XINT_sqrt_small_e #1.#2.%
{%
   \expandafter\XINT_sqrt_small_ea\the\numexpr #1*#1-#2.#1.%
}%
```

```
\def\XINT_sqrt_small_ea #1%
{%
    \if0#1\xint_dothis\XINT_sqrt_small_ez\fi
    \if-#1\xint_dothis\XINT_sqrt_small_eb\fi
    \xint_orthat\XINT_sqrt_small_f #1%
}%
\def\XINT_sqrt_small_ez 0.#1.{\expandafter{\the\numexpr#1+\xint_c_i
          \expandafter}\expandafter{\the\numexpr #1*\xint_c_ii+\xint_c_i}}%


\def\XINT_sqrt_small_eb -#1.#2.%
{%
    \expandafter\XINT_sqrt_small_ec \the\numexpr
    (#1-\xint_c_i+#2)/(\xint_c_ii*#2).#1.#2.%
}%


\def\XINT_sqrt_small_ec #1.#2.#3.%
{%
    \expandafter\XINT_sqrt_small_f \the\numexpr
      -#2+\xint_c_ii*#3*#1+#1*#1\expandafter.\the\numexpr #3+#1.%
}%


\def\XINT_sqrt_small_f #1.#2.%
{%
   \expandafter\XINT_sqrt_small_g
   \the\numexpr (#1+#2)/(\xint_c_ii*#2)-\xint_c_i.#1.#2.%
}%


\def\XINT_sqrt_small_g #1#2.%
{%
    \if 0#1%
       \expandafter\XINT_sqrt_small_end
    \else
       \expandafter\XINT_sqrt_small_h
    \fi
    #1#2.%
}%


\def\XINT_sqrt_small_h #1.#2.#3.%
{%
    \expandafter\XINT_sqrt_small_f
    \the\numexpr #2-\xint_c_ii*#1*#3+#1*#1\expandafter.%
    \the\numexpr #3-#1.%
}%
\def\XINT_sqrt_small_end #1.#2.#3.{{#3}{#2}}%


\def\XINT_sqrt_big_d #1.#2%
{%
   \ifodd #2 \xint_dothis{\expandafter\XINT_sqrt_big_eO}\fi
   \xint_orthat{\expandafter\XINT_sqrt_big_eE}%
```

```
    \the\numexpr (#2-\xint_c_i)/\xint_c_ii.#1;%
}%

\def\XINT_sqrt_big_eE  #1;#2#3#4#5#6#7#8#9%
{%
    \XINT_sqrt_big_eE_a #1;{#2#3#4#5#6#7#8#9}%
}%

\def\XINT_sqrt_big_eE_a #1.#2;#3%
{%
    \expandafter\XINT_sqrt_bigormed_f
    \romannumeral0\XINT_sqrt_small_e #2000.#3.#1;%
}%

\def\XINT_sqrt_big_eO #1;#2#3#4#5#6#7#8#9%
{%
    \XINT_sqrt_big_eO_a #1;{#2#3#4#5#6#7#8#9}%
}%
\def\XINT_sqrt_big_eO_a #1.#2;#3#4%
{%
    \expandafter\XINT_sqrt_bigormed_f
    \romannumeral0\XINT_sqrt_small_e #20000.#3#4.#1;%
}%

\def\XINT_sqrt_bigormed_f #1#2#3;%
{%
    \ifnum#3<\xint_c_ix
          \xint_dothis {\csname XINT_sqrt_med_f\romannumeral#3\endcsname}%
    \fi
    \xint_orthat\XINT_sqrt_big_f #1.#2.#3;%
}%
\def\XINT_sqrt_med_fv   {\XINT_sqrt_med_fa .}%
\def\XINT_sqrt_med_fvi  {\XINT_sqrt_med_fa 0.}%
\def\XINT_sqrt_med_fvii {\XINT_sqrt_med_fa 00.}%
\def\XINT_sqrt_med_fviii{\XINT_sqrt_med_fa 000.}%

\def\XINT_sqrt_med_fa #1.#2.#3.#4;%
{%
    \expandafter\XINT_sqrt_med_fb
    \the\numexpr (#30#1-5#1)/(\xint_c_ii*#2).#1.#2.#3.%
}%

\def\XINT_sqrt_med_fb #1.#2.#3.#4.#5.%
{%
    \expandafter\XINT_sqrt_small_ea
    \the\numexpr (#40#2-\xint_c_ii*#3*#1)*10#2+(#1*#1-#5)\expandafter.%
    \the\numexpr #30#2-#1.%
}%
```

```
\def\XINT_sqrt_big_f #1;#2#3#4#5#6#7#8#9%
{%
    \XINT_sqrt_big_fa #1;{#2#3#4#5#6#7#8#9}%
}%

\def\XINT_sqrt_big_fa #1.#2.#3;#4%
{%
    \expandafter\XINT_sqrt_big_ga
    \the\numexpr #3-\xint_c_viii\expandafter.%
    \romannumeral0\XINT_sqrt_med_fa 000.#1.#2.;#4.%
}%

\def\XINT_sqrt_big_ga #1.#2#3%
{%
    \ifnum #1>\xint_c_viii
      \expandafter\XINT_sqrt_big_gb\else
      \expandafter\XINT_sqrt_big_ka
    \fi #1.#3.#2.%
}%

\def\XINT_sqrt_big_gb #1.#2.#3.%
{%
    \expandafter\XINT_sqrt_big_gc
    \the\numexpr (\xint_c_ii*#2-\xint_c_i)*\xint_c_x^viii/(\xint_c_iv*#3).%
    #3.#2.#1;%
}%

\def\XINT_sqrt_big_gc #1.#2.#3.%
{%
    \expandafter\XINT_sqrt_big_gd
    \romannumeral0\xintiiadd
        {\xintiiSub {#300000000}{\xintDouble{\xintiiMul{#2}{#1}}}00000000}%
        {\xintiiSqr {#1}}.%
    \romannumeral0\xintiisub{#200000000}{#1}.%
}%

\def\XINT_sqrt_big_gd #1.#2.%
{%
    \expandafter\XINT_sqrt_big_ge #2.#1.%
}%

\def\XINT_sqrt_big_ge #1;#2#3#4#5#6#7#8#9%
   {\XINT_sqrt_big_gf #1.#2#3#4#5#6#7#8#9;}%
\def\XINT_sqrt_big_gf #1;#2#3#4#5#6#7#8#9%
   {\XINT_sqrt_big_gg #1#2#3#4#5#6#7#8#9.}%

\def\XINT_sqrt_big_gg #1.#2.#3.#4.%
{%
    \expandafter\XINT_sqrt_big_gloop
```

```
    \expandafter\xint_c_xvi\expandafter.%
    \the\numexpr #3-\xint_c_viii\expandafter.%
    \romannumeral0\xintiisub {#2}{\xintiNum{#4}}.#1.%
}%

\def\XINT_sqrt_big_gloop #1.#2.%
{%
    \unless\ifnum #1<#2 \xint_dothis\XINT_sqrt_big_ka \fi
    \xint_orthat{\XINT_sqrt_big_gi #1.}#2.%
}%

\def\XINT_sqrt_big_gi #1.%
{%
    \expandafter\XINT_sqrt_big_gj\romannumeral\xintreplicate{#1}0.#1.%
}%

\def\XINT_sqrt_big_gj #1.#2.#3.#4.#5.%
{%
    \expandafter\XINT_sqrt_big_gk
    \romannumeral0\xintiidivision {#4#1}%
                  {\XINT_dbl #5\xint_bye2345678\xint_bye*\xint_c_ii\relax}.%
    #1.#5.#2.#3.%
}%

\def\XINT_sqrt_big_gk #1#2.#3.#4.%
{%
    \expandafter\XINT_sqrt_big_gl
    \romannumeral0\xintiiadd {#2#3}{\xintiiSqr{#1}}.%
    \romannumeral0\xintiisub {#4#3}{#1}.%
}%

\def\XINT_sqrt_big_gl #1.#2.%
{%
    \expandafter\XINT_sqrt_big_gm #2.#1.%
}%

\def\XINT_sqrt_big_gm #1.#2.#3.#4.#5.%
{%
    \expandafter\XINT_sqrt_big_gn

    \romannumeral0\XINT_split_fromleft\xint_c_ii*#3.#5\xint_bye2345678\xint_bye..%
    #1.#2.#3.#4.%
}%

\def\XINT_sqrt_big_gn #1.#2.#3.#4.#5.#6.%
{%
    \expandafter\XINT_sqrt_big_gloop
    \the\numexpr \xint_c_ii*#5\expandafter.%
    \the\numexpr #6-#5\expandafter.%
    \romannumeral0\xintiisub{#4}{\xintiNum{#1}}.#3.#2.%
}%
```

```
\def\XINT_sqrt_big_ka #1.#2.#3.#4.%
{%
    \expandafter\XINT_sqrt_big_kb

    \romannumeral0\XINT_dsx_addzeros {#1}#3;.%
    \romannumeral0\xintiisub
      {\XINT_dsx_addzerosnofuss {\xint_c_ii*#1}#2;}%
      {\xintiNum{#4}}.%
}%
\def\XINT_sqrt_big_kb #1.#2.%
{%
    \expandafter\XINT_sqrt_big_kc #2.#1.%
}%

\def\XINT_sqrt_big_kc #1%
{%
    \if0#1\xint_dothis\XINT_sqrt_big_kz\fi
    \xint_orthat\XINT_sqrt_big_kloop #1%
}%
\def\XINT_sqrt_big_kz 0.#1.%
{%
    \expandafter\XINT_sqrt_big_kend
    \romannumeral0%
    \xintinc{\XINT_dbl#1\xint_bye2345678\xint_bye*\xint_c_ii\relax}.#1.%
}%
\def\XINT_sqrt_big_kend #1.#2.%
{%
    \expandafter{\romannumeral0\xintinc{#2}}{#1}%
}%

\def\XINT_sqrt_big_kloop #1.#2.%
{%
    \expandafter\XINT_sqrt_big_ke
    \romannumeral0\xintiidivision{#1}%
     {\romannumeral0\XINT_dbl #2\xint_bye2345678\xint_bye*\xint_c_ii\relax}{#2}%
}%

\def\XINT_sqrt_big_ke #1%
{%
    \if0\XINT_Sgn #1\xint:
          \expandafter \XINT_sqrt_big_end
    \else \expandafter \XINT_sqrt_big_kf
    \fi {#1}%
}%

\def\XINT_sqrt_big_kf #1#2#3%
{%
    \expandafter\XINT_sqrt_big_kg
    \romannumeral0\xintiisub {#3}{#1}.%
    \romannumeral0\xintiiadd {#2}{\xintiiSqr {#1}}.%
```

```
}%
\def\XINT_sqrt_big_kg #1.#2.%
{%
    \expandafter\XINT_sqrt_big_kloop #2.#1.%
}%

\def\XINT_sqrt_big_end #1#2#3{{#3}{#2}}%
```

5.52 \xintiiSqrt, \xintiiSqrtR

```
\def\xintiiSqrt  {\romannumeral0\xintiisqrt  }%
\def\xintiisqrt  {\expandafter\XINT_sqrt_post\romannumeral0\xintiisquareroot  }%
\def\XINT_sqrt_post #1#2{\XINT_dec #1\XINT_dec_bye234567890\xint_bye}%
\def\xintiiSqrtR {\romannumeral0\xintiisqrtr }%
\def\xintiisqrtr {\expandafter\XINT_sqrtr_post\romannumeral0\xintiisquareroot }%
```

N = (#1)^2 - #2 avec #1 le plus petit possible et #2>0 (hence #2<2*#1). (#1-.5)^2=#1^2-#1+.25=N+#2-#1+.25. Si 0<#2<#1, <= N-0.75<N, donc rounded->#1 si #2>=#1, (#1-.5)^2>=N+.25>N, donc rounded->#1-1.

```
\def\XINT_sqrtr_post #1#2%
    {\xintiiifLt {#2}{#1}{ #1}{\XINT_dec #1\XINT_dec_bye234567890\xint_bye}}%
```

5.53 \xintiiBinomial

2015/11/28-29 for 1.2f.
2016/11/19 for 1.2h: I truly can't understand why I hard-coded last year an error-message for arguments outside of the range for binomial formula. Naturally there should be no error but a rather a 0 return value for binomial(x,y), if y<0 or x<y !
I really lack some kind of infinity or NaN value.
1.2o deprecates \xintiBinomial. (which xintfrac.sty redefined to use \xintNum)

```
\def\xintiiBinomial {\romannumeral0\xintiibinomial }%
\def\xintiibinomial #1#2%
{%
    \expandafter\XINT_binom_pre\the\numexpr #1\expandafter.\the\numexpr #2.%
}%
\def\XINT_binom_pre #1.#2.%
{%
    \expandafter\XINT_binom_fork \the\numexpr#1-#2.#2.#1.%
}%
```

k.x-k.x. I hesitated to restrict maximal allowed value of x to 10000. Finally I don't. But due to using small multiplication and small division, x must have at most eight digits. If x>=2^31 an arithmetic overflow error will have happened already.

```
\def\XINT_binom_fork #1#2.#3#4.#5#6.%
{%
    \if-#5\xint_dothis{\XINT_signalcondition{InvalidOperation}{Binomial with
        negative first arg: #5#6}{}{0}}\fi
    \if-#1\xint_dothis{ 0}\fi
    \if-#3\xint_dothis{ 0}\fi
    \if0#1\xint_dothis{ 1}\fi
    \if0#3\xint_dothis{ 1}\fi
```

```
    \ifnum #5#6>\xint_c_x^viii_mone\xint_dothis
       {\XINT_signalcondition{InvalidOperation}{Binomial with too
           large argument: 99999999 < #5#6}{}{0}}\fi
    \ifnum #1#2>#3#4  \xint_dothis{\XINT_binom_a #1#2.#3#4.}\fi
                      \xint_orthat{\XINT_binom_a #3#4.#1#2.}%
}%
```

x-k.k. avec 0<k<x, k<=x-k. Les divisions produiront en extra après le quotient un terminateur 1!\Z!0!. On va procéder par petite multiplication suivie par petite division. Donc ici on met le 1!\Z!0! pour amorcer.
Le \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax est le terminateur pour le \XINT_unsep_cuzsmall final.

```
\def\XINT_binom_a #1.#2.%
{%
    \expandafter\XINT_binom_b\the\numexpr \xint_c_i+#1.1.#2.100000001!1!;!0!%
}%
```

y=x-k+1.j=1.k. On va évaluer par y/1*(y+1)/2*(y+2)/3 etc... On essaie de regrouper de manière à utiliser au mieux \numexpr. On peut aller jusqu'à x=10000 car 9999*10000<10^8. 463*464*465=99896880, 98*99*100*101=97990200. On va vérifier à chaque étape si on dépasse un seuil. Le style de l'implémentation diffère de celui que j'avais utilisé pour \xintiiFac. On pourrait tout-à-fait avoir une verybigloop, mais bon. Je rajoute aussi un verysmall. Le traitement est un peu différent pour elle afin d'aller jusqu'à x=29 (et pas seulement 26 si je suivais le modèle des autres, mais je veux pouvoir faire binomial(29,1), binomial(29,2), ... en vsmall).

```
\def\XINT_binom_b #1.%
{%
    \ifnum #1>9999 \xint_dothis\XINT_binom_vbigloop \fi
    \ifnum #1>463  \xint_dothis\XINT_binom_bigloop   \fi
    \ifnum #1>98   \xint_dothis\XINT_binom_medloop   \fi
    \ifnum #1>29   \xint_dothis\XINT_binom_smallloop \fi
                   \xint_orthat\XINT_binom_vsmallloop #1.%
}%
```

y.j.k. Au départ on avait x-k+1.1.k. Ensuite on a des blocs 1<8d>! donnant le résultat intermédiaire, dans l'ordre, et à la fin on a 1!1;!0!. Dans smallloop on peut prendre 4 par 4.

```
\def\XINT_binom_smallloop #1.#2.#3.%
{%
    \ifcase\numexpr #3-#2\relax
        \expandafter\XINT_binom_end_
    \or \expandafter\XINT_binom_end_i
    \or \expandafter\XINT_binom_end_ii
    \or \expandafter\XINT_binom_end_iii
    \else\expandafter\XINT_binom_smallloop_a
    \fi #1.#2.#3.%
}%
```

Ça m'ennuie un peu de reprendre les #1, #2, #3 ici. On a besoin de \numexpr pour \XINT_binom_div, mais de \romannumeral0 pour le unsep après \XINT_binom_mul.

```
\def\XINT_binom_smallloop_a #1.#2.#3.%
{%
    \expandafter\XINT_binom_smallloop_b
```

```
    \the\numexpr #1+\xint_c_iv\expandafter.%
    \the\numexpr #2+\xint_c_iv\expandafter.%
    \the\numexpr #3\expandafter.%
    \the\numexpr\expandafter\XINT_binom_div
    \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)*(#2+\xint_c_iii)\expandafter
    !\romannumeral0\expandafter\XINT_binom_mul
    \the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_binom_smallloop_b #1.%
{%
    \ifnum #1>98  \expandafter\XINT_binom_medloop   \else
                  \expandafter\XINT_binom_smallloop \fi #1.%
}%
```

Ici on prend trois par trois.

```
\def\XINT_binom_medloop #1.#2.#3.%
{%
    \ifcase\numexpr #3-#2\relax
        \expandafter\XINT_binom_end_
    \or \expandafter\XINT_binom_end_i
    \or \expandafter\XINT_binom_end_ii
    \else\expandafter\XINT_binom_medloop_a
    \fi #1.#2.#3.%
}%
\def\XINT_binom_medloop_a #1.#2.#3.%
{%
    \expandafter\XINT_binom_medloop_b
    \the\numexpr #1+\xint_c_iii\expandafter.%
    \the\numexpr #2+\xint_c_iii\expandafter.%
    \the\numexpr #3\expandafter.%
    \the\numexpr\expandafter\XINT_binom_div
        \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)\expandafter
    !\romannumeral0\expandafter\XINT_binom_mul
        \the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_binom_medloop_b #1.%
{%
    \ifnum #1>463 \expandafter\XINT_binom_bigloop   \else
                  \expandafter\XINT_binom_medloop   \fi #1.%
}%
```

Ici on prend deux par deux.

```
\def\XINT_binom_bigloop #1.#2.#3.%
{%
    \ifcase\numexpr #3-#2\relax
        \expandafter\XINT_binom_end_
    \or \expandafter\XINT_binom_end_i
    \else\expandafter\XINT_binom_bigloop_a
    \fi #1.#2.#3.%
}%
\def\XINT_binom_bigloop_a #1.#2.#3.%
{%
```

```
    \expandafter\XINT_binom_bigloop_b
    \the\numexpr #1+\xint_c_ii\expandafter.%
    \the\numexpr #2+\xint_c_ii\expandafter.%
    \the\numexpr #3\expandafter.%
    \the\numexpr\expandafter\XINT_binom_div
        \the\numexpr #2*(#2+\xint_c_i)\expandafter
    !\romannumeral0\expandafter\XINT_binom_mul
        \the\numexpr #1*(#1+\xint_c_i)!%
}%
\def\XINT_binom_bigloop_b #1.%
{%
    \ifnum #1>9999 \expandafter\XINT_binom_vbigloop  \else
                   \expandafter\XINT_binom_bigloop   \fi #1.%
}%
```

Et finalement un par un.

```
\def\XINT_binom_vbigloop #1.#2.#3.%
{%
    \ifnum #3=#2
         \expandafter\XINT_binom_end_
    \else\expandafter\XINT_binom_vbigloop_a
    \fi #1.#2.#3.%
}%
\def\XINT_binom_vbigloop_a #1.#2.#3.%
{%
    \expandafter\XINT_binom_vbigloop
    \the\numexpr #1+\xint_c_i\expandafter.%
    \the\numexpr #2+\xint_c_i\expandafter.%
    \the\numexpr #3\expandafter.%
    \the\numexpr\expandafter\XINT_binom_div\the\numexpr #2\expandafter
    !\romannumeral0\XINT_binom_mul #1!%
}%
```

y.j.k. La partie very small. y est au plus 26 (non 29 mais retesté dans \XINT_binom_vsmallloop_a), et tous les binomial(29,n) sont <10^8. On peut donc faire y(y+1)(y+2)(y+3) et aussi il y a le fait que etex fait a*b/c en double precision. Pour ne pas bifurquer à la fin sur smallloop, si n=27, 27, ou 29 on procède un peu différemment des autres boucles. Si je testais aussi #1 après #3-#2 pour les autres il faudrait des terminaisons différentes.

```
\def\XINT_binom_vsmallloop #1.#2.#3.%
{%
    \ifcase\numexpr #3-#2\relax
        \expandafter\XINT_binom_vsmallend_
    \or \expandafter\XINT_binom_vsmallend_i
    \or \expandafter\XINT_binom_vsmallend_ii
    \or \expandafter\XINT_binom_vsmallend_iii
    \else\expandafter\XINT_binom_vsmallloop_a
    \fi #1.#2.#3.%
}%
\def\XINT_binom_vsmallloop_a #1.%
{%
    \ifnum #1>26  \expandafter\XINT_binom_smallloop_a  \else
                  \expandafter\XINT_binom_vsmallloop_b \fi #1.%
```

```
}%
\def\XINT_binom_vsmallloop_b #1.#2.#3.%
{%
    \expandafter\XINT_binom_vsmallloop
    \the\numexpr #1+\xint_c_iv\expandafter.%
    \the\numexpr #2+\xint_c_iv\expandafter.%
    \the\numexpr #3\expandafter.%
    \the\numexpr \expandafter\XINT_binom_vsmallmuldiv
    \the\numexpr  #2*(#2+\xint_c_i)*(#2+\xint_c_ii)*(#2+\xint_c_iii)\expandafter
    !\the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_binom_mul #1!#21!;!0!%
{%
    \expandafter\XINT_rev_nounsep\expandafter{\expandafter}%
    \the\numexpr\expandafter\XINT_smallmul
    \the\numexpr\xint_c_x^viii+#1\expandafter
    !\romannumeral0\XINT_rev_nounsep {}1;!#2%
    \R!\R!\R!\R!\R!\R!\R!\W
    \R!\R!\R!\R!\R!\R!\R!\W
    1;!%
}%
\def\XINT_binom_div #1!1;!%
{%
    \expandafter\XINT_smalldivx_a
    \the\numexpr #1/\xint_c_ii\expandafter\xint:
    \the\numexpr \xint_c_x^viii+#1!%
}%
```

Vaguement envisagé d'éviter le 10^8+ mais bon.

```
\def\XINT_binom_vsmallmuldiv #1!#2!1#3!{\xint_c_x^viii+#2*#3/#1!}%
```

On a des terminaisons communes aux trois situations small, med, big, et on est sûr de pouvoir faire les multiplications dans \numexpr, car on vient ici *après* avoir comparé à 9999 ou 463 ou 98.

```
\def\XINT_binom_end_iii #1.#2.#3.%
{%
    \expandafter\XINT_binom_finish
    \the\numexpr\expandafter\XINT_binom_div
        \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)*(#2+\xint_c_iii)\expandafter
    !\romannumeral0\expandafter\XINT_binom_mul
        \the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_binom_end_ii #1.#2.#3.%
{%
    \expandafter\XINT_binom_finish
    \the\numexpr\expandafter\XINT_binom_div
        \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)\expandafter
    !\romannumeral0\expandafter\XINT_binom_mul
        \the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_binom_end_i #1.#2.#3.%
```

```
{%
    \expandafter\XINT_binom_finish
    \the\numexpr\expandafter\XINT_binom_div
        \the\numexpr #2*(#2+\xint_c_i)\expandafter
    !\romannumeral0\expandafter\XINT_binom_mul
        \the\numexpr #1*(#1+\xint_c_i)!%
}%
\def\XINT_binom_end_ #1.#2.#3.%
{%
    \expandafter\XINT_binom_finish
    \the\numexpr\expandafter\XINT_binom_div\the\numexpr #2\expandafter
    !\romannumeral0\XINT_binom_mul #1!%
}%
\def\XINT_binom_finish #1;!0!%
   {\XINT_unsep_cuzsmall #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax}%
```

Duplication de code seulement pour la boucle avec très petits coeffs, mais en plus on fait au maximum des possibilités. (on pourrait tester plus le résultat déjà obtenu).

```
\def\XINT_binom_vsmallend_iii #1.%
{%
    \ifnum #1>26  \expandafter\XINT_binom_end_iii \else
                  \expandafter\XINT_binom_vsmallend_iiib \fi #1.%
}%
\def\XINT_binom_vsmallend_iiib #1.#2.#3.%
{%
    \expandafter\XINT_binom_vsmallfinish
    \the\numexpr \expandafter\XINT_binom_vsmallmuldiv
    \the\numexpr  #2*(#2+\xint_c_i)*(#2+\xint_c_ii)*(#2+\xint_c_iii)\expandafter
    !\the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_binom_vsmallend_ii #1.%
{%
    \ifnum #1>27  \expandafter\XINT_binom_end_ii \else
                  \expandafter\XINT_binom_vsmallend_iib \fi #1.%
}%
\def\XINT_binom_vsmallend_iib #1.#2.#3.%
{%
    \expandafter\XINT_binom_vsmallfinish
    \the\numexpr \expandafter\XINT_binom_vsmallmuldiv
    \the\numexpr  #2*(#2+\xint_c_i)*(#2+\xint_c_ii)\expandafter
    !\the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_binom_vsmallend_i #1.%
{%
    \ifnum #1>28  \expandafter\XINT_binom_end_i \else
                  \expandafter\XINT_binom_vsmallend_ib \fi #1.%
}%
\def\XINT_binom_vsmallend_ib #1.#2.#3.%
{%
    \expandafter\XINT_binom_vsmallfinish
    \the\numexpr \expandafter\XINT_binom_vsmallmuldiv
    \the\numexpr  #2*(#2+\xint_c_i)\expandafter
```

```
    !\the\numexpr #1*(#1+\xint_c_i)!%
}%
\def\XINT_binom_vsmallend_ #1.%
{%
    \ifnum #1>29  \expandafter\XINT_binom_end_ \else
                  \expandafter\XINT_binom_vsmallend_b \fi #1.%
}%
\def\XINT_binom_vsmallend_b #1.#2.#3.%
{%
    \expandafter\XINT_binom_vsmallfinish
    \the\numexpr\XINT_binom_vsmallmuldiv #2!#1!%
}%
\def\XINT_binom_vsmallfinish#1{%
\def\XINT_binom_vsmallfinish1##1!1!;!0!{\expandafter#1\the\numexpr##1\relax}%
}\XINT_binom_vsmallfinish{ }%
```

5.54 \xintiiPFactorial

2015/11/29 for 1.2f. Partial factorial pfac(a,b)=(a+1)...b, only for non-negative integers with a<=b<10^8.

1.2h (2016/11/20) removes the non-negativity condition. It was a bit unfortunate that the code raised \xintError:OutOfRangePFac if 0<=a<=b<10^8 was violated. The rule now applied is to interpret pfac(a,b) as the product for a<j<=b (not as a ratio of Gamma function), hence if a>=b, return 1 because of an empty product. If a<b: if a<0, return 0 for b>=0 and (-1)^(b-a) times |b|...(|a|-1) for b<0. But only for the range 0<= a <= b < 10^8 is the macro result to be considered as stable.

```
\def\xintiiPFactorial {\romannumeral0\xintiipfactorial }%
\def\xintiipfactorial #1#2%
{%
    \expandafter\XINT_pfac_fork\the\numexpr#1\expandafter.\the\numexpr #2.%
}%
\def\xintPFactorial{\romannumeral0\xintpfactorial}%
\let\xintpfactorial\xintiipfactorial
```

Code is a simplified version of the one for \xintiiBinomial, with no attempt at implementing a "very small" branch.

```
\def\XINT_pfac_fork #1#2.#3#4.%
{%
    \unless\ifnum #1#2<#3#4 \xint_dothis\XINT_pfac_one\fi
    \if-#3\xint_dothis\XINT_pfac_neg\fi
    \if-#1\xint_dothis\XINT_pfac_zero\fi
    \ifnum #3#4>\xint_c_x^viii_mone\xint_dothis\XINT_pfac_outofrange\fi
    \xint_orthat \XINT_pfac_a #1#2.#3#4.%
}%
\def\XINT_pfac_outofrange #1.#2.%
   {\XINT_signalcondition{InvalidOperation}{PFactorial with
    too big second arg: 99999999 < #2}{}{0}}%
\def\XINT_pfac_one        #1.#2.{ 1}%
\def\XINT_pfac_zero       #1.#2.{ 0}%
\def\XINT_pfac_neg -#1.-#2.%
{%
    \ifnum #1>\xint_c_x^viii\xint_dothis\XINT_pfac_outofrange\fi
```

```
    \xint_orthat
   {\ifodd\numexpr#2-#1\relax\xint_afterfi{\expandafter-\romannumeral`&&@}\fi
    \expandafter\XINT_pfac_a }%
    \the\numexpr #2-\xint_c_i\expandafter.\the\numexpr#1-\xint_c_i.%
}%
\def\XINT_pfac_a #1.#2.%
{%
    \expandafter\XINT_pfac_b\the\numexpr \xint_c_i+#1.#2.100000001!1;!%
    1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
}%
\def\XINT_pfac_b #1.%
{%
    \ifnum #1>9999 \xint_dothis\XINT_pfac_vbigloop \fi
    \ifnum #1>463  \xint_dothis\XINT_pfac_bigloop   \fi
    \ifnum #1>98   \xint_dothis\XINT_pfac_medloop   \fi
                   \xint_orthat\XINT_pfac_smallloop #1.%
}%
\def\XINT_pfac_smallloop #1.#2.%
{%
    \ifcase\numexpr #2-#1\relax
        \expandafter\XINT_pfac_end_
    \or \expandafter\XINT_pfac_end_i
    \or \expandafter\XINT_pfac_end_ii
    \or \expandafter\XINT_pfac_end_iii
    \else\expandafter\XINT_pfac_smallloop_a
    \fi #1.#2.%
}%
\def\XINT_pfac_smallloop_a #1.#2.%
{%
    \expandafter\XINT_pfac_smallloop_b
    \the\numexpr #1+\xint_c_iv\expandafter.%
    \the\numexpr #2\expandafter.%
    \the\numexpr\expandafter\XINT_smallmul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_pfac_smallloop_b #1.%
{%
    \ifnum #1>98  \expandafter\XINT_pfac_medloop   \else
                  \expandafter\XINT_pfac_smallloop \fi #1.%
}%
\def\XINT_pfac_medloop #1.#2.%
{%
    \ifcase\numexpr #2-#1\relax
        \expandafter\XINT_pfac_end_
    \or \expandafter\XINT_pfac_end_i
    \or \expandafter\XINT_pfac_end_ii
    \else\expandafter\XINT_pfac_medloop_a
    \fi #1.#2.%
}%
\def\XINT_pfac_medloop_a #1.#2.%
{%
    \expandafter\XINT_pfac_medloop_b
```

```
    \the\numexpr #1+\xint_c_iii\expandafter.%
    \the\numexpr #2\expandafter.%
    \the\numexpr\expandafter\XINT_smallmul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_pfac_medloop_b #1.%
{%
    \ifnum #1>463 \expandafter\XINT_pfac_bigloop   \else
                  \expandafter\XINT_pfac_medloop   \fi #1.%
}%
\def\XINT_pfac_bigloop #1.#2.%
{%
    \ifcase\numexpr #2-#1\relax
        \expandafter\XINT_pfac_end_
    \or \expandafter\XINT_pfac_end_i
    \else\expandafter\XINT_pfac_bigloop_a
    \fi #1.#2.%
}%
\def\XINT_pfac_bigloop_a #1.#2.%
{%
    \expandafter\XINT_pfac_bigloop_b
    \the\numexpr #1+\xint_c_ii\expandafter.%
    \the\numexpr #2\expandafter.%
    \the\numexpr\expandafter
    \XINT_smallmul\the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
}%
\def\XINT_pfac_bigloop_b #1.%
{%
    \ifnum #1>9999 \expandafter\XINT_pfac_vbigloop  \else
                   \expandafter\XINT_pfac_bigloop   \fi #1.%
}%
\def\XINT_pfac_vbigloop #1.#2.%
{%
    \ifnum #2=#1
         \expandafter\XINT_pfac_end_
    \else\expandafter\XINT_pfac_vbigloop_a
    \fi #1.#2.%
}%
\def\XINT_pfac_vbigloop_a #1.#2.%
{%
    \expandafter\XINT_pfac_vbigloop
    \the\numexpr #1+\xint_c_i\expandafter.%
    \the\numexpr #2\expandafter.%
    \the\numexpr\expandafter\XINT_smallmul\the\numexpr\xint_c_x^viii+#1!%
}%
\def\XINT_pfac_end_iii #1.#2.%
{%
    \expandafter\XINT_mul_out
    \the\numexpr\expandafter\XINT_smallmul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_pfac_end_ii #1.#2.%
```

```
{%
    \expandafter\XINT_mul_out
    \the\numexpr\expandafter\XINT_smallmul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_pfac_end_i #1.#2.%
{%
    \expandafter\XINT_mul_out
    \the\numexpr\expandafter\XINT_smallmul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
}%
\def\XINT_pfac_end_ #1.#2.%
{%
    \expandafter\XINT_mul_out
    \the\numexpr\expandafter\XINT_smallmul\the\numexpr \xint_c_x^viii+#1!%
}%
```

5.55 \xintBool, \xintToggle

1.09c

```
\def\xintBool #1{\romannumeral`&&@%
                 \csname if#1\endcsname\expandafter1\else\expandafter0\fi }%
\def\xintToggle #1{\romannumeral`&&@\iftoggle{#1}{1}{0}}%
```

5.56 \xintiiGCD

1.3d: \xintiiGCD code from xintgcd is copied here to support gcd() function in \xintiiexpr.
1.4: removed from xintgcd the original caode as now xintgcd loads xint.

1.4d (2021/03/22). Damn'ed! Since 1.3d (2019/01/06) the code was broken if one of the arguments vanished due to a typo in macro names: "AisZero" at one location and "Aiszero" at next, and same for B...
How could this not be detected by my tests !?!
This caused \xintiiGCDof hence the gcd() function in \xintiiexpr to break as soon as one argument was zero.

```
\def\xintiiGCD {\romannumeral0\xintiigcd }%
\def\xintiigcd #1{\expandafter\XINT_iigcd\romannumeral0\xintiiabs#1\xint:}%
\def\XINT_iigcd #1#2\xint:#3%
{%
    \expandafter\XINT_gcd_fork\expandafter#1%
                \romannumeral0\xintiiabs#3\xint:#1#2\xint:
}%
\def\XINT_gcd_fork #1#2%
{%
    \xint_UDzerofork
      #1\XINT_gcd_Aiszero
      #2\XINT_gcd_Biszero
       0\XINT_gcd_loop
    \krof
    #2%
}%
\def\XINT_gcd_Aiszero #1\xint:#2\xint:{ #1}%
```

```
\def\XINT_gcd_Biszero #1\xint:#2\xint:{ #2}%
\def\XINT_gcd_loop #1\xint:#2\xint:
{%
    \expandafter\expandafter\expandafter\XINT_gcd_CheckRem
    \expandafter\xint_secondoftwo
    \romannumeral0\XINT_div_prepare {#1}{#2}\xint:#1\xint:
}%
\def\XINT_gcd_CheckRem #1%
{%
    \xint_gob_til_zero #1\XINT_gcd_end0\XINT_gcd_loop #1%
}%
\def\XINT_gcd_end0\XINT_gcd_loop #1\xint:#2\xint:{ #2}%
```

5.57 \xintiiGCDof

New with 1.09a (was located in xintgcd.sty).
1.2l adds protection against items being non-terminated \the\numexpr.
1.4 renames the macro into \xintiiGCDof and moves it here. Terminator modified to ^ for direct call by \xintiiexpr function.
1.4d fixes breakage inherited since 1.3d rom \xintiiGCD, in case any argument vanished.
Currently does not support empty list of arguments.

```
\def\xintiiGCDof    {\romannumeral0\xintiigcdof }%
\def\xintiigcdof    #1{\expandafter\XINT_iigcdof_a\romannumeral`&&@#1^}%
\def\XINT_iiGCDof   {\romannumeral0\XINT_iigcdof_a}%
\def\XINT_iigcdof_a #1{\expandafter\XINT_iigcdof_b\romannumeral`&&@#1!}%
\def\XINT_iigcdof_b #1!#2{\expandafter\XINT_iigcdof_c\romannumeral`&&@#2!{#1}!}%
\def\XINT_iigcdof_c #1{\xint_gob_til_^ #1\XINT_iigcdof_e ^\XINT_iigcdof_d #1}%
\def\XINT_iigcdof_d #1!{\expandafter\XINT_iigcdof_b\romannumeral0\xintiigcd {#1}}%
\def\XINT_iigcdof_e #1!#2!{ #2}%
```

5.58 \xintiiLCM

Copied over \xintiiLCM code from xintgcd at 1.3d in order to support lcm() function in \xintiiexpr.
At 1.4 original code removed from xintgcd as the latter now requires xint.

```
\def\xintiiLCM {\romannumeral0\xintiilcm}%
\def\xintiilcm #1{\expandafter\XINT_iilcm\romannumeral0\xintiiabs#1\xint:}%
\def\XINT_iilcm #1#2\xint:#3%
{%
    \expandafter\XINT_lcm_fork\expandafter#1%
                 \romannumeral0\xintiiabs#3\xint:#1#2\xint:
}%
\def\XINT_lcm_fork #1#2%
{%
    \xint_UDzerofork
      #1\XINT_lcm_iszero
      #2\XINT_lcm_iszero
       0\XINT_lcm_notzero
    \krof
    #2%
}%
\def\XINT_lcm_iszero #1\xint:#2\xint:{ 0}%
```

```
\def\XINT_lcm_notzero #1\xint:#2\xint:
{%
    \expandafter\XINT_lcm_end\romannumeral0%
      \expandafter\expandafter\expandafter\XINT_gcd_CheckRem
      \expandafter\xint_secondoftwo
      \romannumeral0\XINT_div_prepare {#1}{#2}\xint:#1\xint:
    \xint:#1\xint:#2\xint:
}%
\def\XINT_lcm_end #1\xint:#2\xint:#3\xint:{\xintiimul {#2}{\xintiiQuo{#3}{#1}}}%
```

5.59 \xintiiLCMof

See comments of \xintiiGCDof

.

```
\def\xintiiLCMof      {\romannumeral0\xintiilcmof }%
\def\xintiilcmof    #1{\expandafter\XINT_iilcmof_a\romannumeral`&&@#1^}%
\def\XINT_iiLCMof   {\romannumeral0\XINT_iilcmof_a}%
\def\XINT_iilcmof_a #1{\expandafter\XINT_iilcmof_b\romannumeral`&&@#1!}%
\def\XINT_iilcmof_b #1!#2{\expandafter\XINT_iilcmof_c\romannumeral`&&@#2!{#1}!}%
\def\XINT_iilcmof_c #1{\xint_gob_til_^ #1\XINT_iilcmof_e ^\XINT_iilcmof_d #1}%
\def\XINT_iilcmof_d #1!{\expandafter\XINT_iilcmof_b\romannumeral0\xintiilcm {#1}}%
\def\XINT_iilcmof_e #1!#2!{ #2}%
```

5.60 (WIP) \xintRandomDigits

1.3b. See user manual. Whether this will be part of xintkernel, xintcore, or xint is yet to be decided.

```
\def\xintRandomDigits{\romannumeral0\xintrandomdigits}%
\def\xintrandomdigits#1%
{%
    \csname xint_gob_andstop_\expandafter\XINT_randomdigits\the\numexpr#1\xint:
}%
\def\XINT_randomdigits#1\xint:
{%
    \expandafter\XINT_randomdigits_a
    \the\numexpr(#1+\xint_c_iii)/\xint_c_viii\xint:#1\xint:
}%
\def\XINT_randomdigits_a#1\xint:#2\xint:
{%
    \romannumeral\numexpr\xint_c_viii*#1-#2\csname XINT_%
      \romannumeral\XINT_replicate #1\endcsname \csname
    XINT_rdg\endcsname
}%
\def\XINT_rdg
{%
    \expandafter\XINT_rdg_aux\the\numexpr%
                  \xint_c_nine_x^viii%
                                -\xint_texuniformdeviate\xint_c_ii^vii%
                  -\xint_c_ii^vii*\xint_texuniformdeviate\xint_c_ii^vii%
                  -\xint_c_ii^xiv*\xint_texuniformdeviate\xint_c_ii^vii%
                  -\xint_c_ii^xxi*\xint_texuniformdeviate\xint_c_ii^vii%
```

```
                  +\xint_texuniformdeviate\xint_c_x^viii%
                  \relax%
}%
\def\XINT_rdg_aux#1{XINT_rdg\endcsname}%
\let\XINT_XINT_rdg\endcsname
```

5.61 (WIP) \XINT_eightrandomdigits, \xintEightRandomDigits

1.3b. 1.4 adds some public alias...

```
\def\XINT_eightrandomdigits
{%
     \expandafter\xint_gobble_i\the\numexpr%
                  \xint_c_nine_x^viii%
                                -\xint_texuniformdeviate\xint_c_ii^vii%
                  -\xint_c_ii^vii*\xint_texuniformdeviate\xint_c_ii^vii%
                  -\xint_c_ii^xiv*\xint_texuniformdeviate\xint_c_ii^vii%
                  -\xint_c_ii^xxi*\xint_texuniformdeviate\xint_c_ii^vii%
                  +\xint_texuniformdeviate\xint_c_x^viii%
                  \relax%
}%
\let\xintEightRandomDigits\XINT_eightrandomdigits
\def\xintRandBit{\xint_texuniformdeviate\xint_c_ii}%
```

5.62 (WIP) \xintRandBit

1.4 And let's add also \xintRandBit while we are at it.

```
\def\xintRandBit{\xint_texuniformdeviate\xint_c_ii}%
```

5.63 (WIP) \xintXRandomDigits

1.3b.

```
\def\xintXRandomDigits#1%
{%
    \csname xint_gobble_\expandafter\XINT_xrandomdigits\the\numexpr#1\xint:
}%
\def\XINT_xrandomdigits#1\xint:
{%
    \expandafter\XINT_xrandomdigits_a
    \the\numexpr(#1+\xint_c_iii)/\xint_c_viii\xint:#1\xint:
}%
\def\XINT_xrandomdigits_a#1\xint:#2\xint:
{%
    \romannumeral\numexpr\xint_c_viii*#1-#2\expandafter\endcsname
    \romannumeral`&&@\romannumeral
                      \XINT_replicate #1\endcsname\XINT_eightrandomdigits
}%
```

5.64 (WIP) \xintiiRandRangeAtoB

1.3b. Support for randrange() function.
  We do it f-expandably for matters of \xintNewExpr etc... The \xintexpr will add \xintNum wrapper to possible fractional input. But \xintiiexpr will call as is.
  TODO: ? implement third argument (STEP) TODO: \xintNum wrapper (which truncates) not so good in floatexpr. Use round?
  It is an error if b<=a, as in Python.

```
\def\xintiiRandRangeAtoB{\romannumeral`&&@\xintiirandrangeAtoB}%
\def\xintiirandrangeAtoB#1%
{%
    \expandafter\XINT_randrangeAtoB_a\romannumeral`&&@#1\xint:
}%
\def\XINT_randrangeAtoB_a#1\xint:#2%
{%
    \xintiiadd{\expandafter\XINT_randrange
                \romannumeral0\xintiisub{#2}{#1}\xint:}%
               {#1}%
}%
```

5.65 (WIP) \xintiiRandRange

1.3b. Support for randrange().

```
\def\xintiiRandRange{\romannumeral`&&@\xintiirandrange}%
\def\xintiirandrange#1%
{%
    \expandafter\XINT_randrange\romannumeral`&&@#1\xint:
}%
\def\XINT_randrange #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_randrange_err:empty
      0#1\XINT_randrange_err:empty
       0-\XINT_randrange_a
    \krof #1%
}%
\def\XINT_randrange_err:empty#1\xint:
{%
    \XINT_expandableerror{Empty range for randrange.} 0%
}%
\def\XINT_randrange_a #1\xint:
{%
    \expandafter\XINT_randrange_b\romannumeral0\xintlength{#1}.#1\xint:
}%
\def\XINT_randrange_b #1.%
{%
    \ifnum#1<\xint_c_x\xint_dothis{\the\numexpr\XINT_uniformdeviate{}}\fi
    \xint_orthat{\XINT_randrange_c #1.}%
}%
\def\XINT_randrange_c #1.#2#3#4#5#6#7#8#9%
{%
```

```
    \expandafter\XINT_randrange_d
    \the\numexpr\expandafter\XINT_uniformdeviate\expandafter
       {\expandafter}\the\numexpr\xint_c_i+#2#3#4#5#6#7#8#9\xint:\xint:
    #2#3#4#5#6#7#8#9\xint:#1\xint:
}%
```

This raises following annex question: immediately after setting the seed is it possible for \xintUniformDeviate{N} where N>0 has exactly eight digits to return either 0 or N-1 ? It could be that this is never the case, then there is a bias in randrange(). Of course there are anyhow only 2^28 seeds so randrange(10^X) is by necessity biased when executed immediately after setting the seed, if X is at least 9.

```
\def\XINT_randrange_d #1\xint:#2\xint:
{%
    \ifnum#1=\xint_c_\xint_dothis\XINT_randrange_Z\fi
    \ifnum#1=#2 \xint_dothis\XINT_randrange_A\fi
    \xint_orthat\XINT_randrange_e #1\xint:
}%
\def\XINT_randrange_e #1\xint:#2\xint:#3\xint:
{%
    \the\numexpr#1\expandafter\relax
    \romannumeral0\xintrandomdigits{#2-\xint_c_viii}%
}%
```

This is quite unlikely to get executed but if it does it must pay attention to leading zeros, hence the \xintinum. We don't have to be overly obstinate about removing overheads...

```
\def\XINT_randrange_Z 0\xint:#1\xint:#2\xint:
{%
    \xintinum{\xintRandomDigits{#1-\xint_c_viii}}%
}%
```

Here too, overhead is not such a problem. The idea is that we got by extraordinary same first 8 digits as upper range bound so we pick at random the remaining needed digits in one go and compare with the upper bound. If too big, we start again with another random 8 leading digits in given range. No need to aim at any kind of efficiency for the check and loop back.

```
\def\XINT_randrange_A #1\xint:#2\xint:#3\xint:
{%
    \expandafter\XINT_randrange_B
    \romannumeral0\xintrandomdigits{#2-\xint_c_viii}\xint:
    #3\xint:#2.#1\xint:
}%
\def\XINT_randrange_B #1\xint:#2\xint:#3.#4\xint:
{%
    \xintiiifLt{#1}{#2}{\XINT_randrange_E}{\XINT_randrange_again}%
    #4#1\xint:#3.#4#2\xint:
}%
\def\XINT_randrange_E #1\xint:#2\xint:{ #1}%
\def\XINT_randrange_again #1\xint:{\XINT_randrange_c}%
```

5.66 (WIP) Adjustments for engines without uniformdeviate primitive

1.3b.

```
\ifdefined\xint_texuniformdeviate
\else
  \def\xintrandomdigits#1%
  {%
      \XINT_expandableerror
      {No uniformdeviate at engine level, returning 0.} 0%
  }%
  \let\xintXRandomDigits\xintRandomDigits
  \def\XINT_randrange#1\xint:
  {%
      \XINT_expandableerror
      {No uniformdeviate at engine level, returning 0.} 0%
  }%
\fi
\XINT_restorecatcodes_endinput%
```

6 Package xintbinhex implementation

The commenting is currently (2021/03/29) very sparse.

The macros from 1.08 (2013/06/07) remained unchanged until their complete rewrite at 1.2m (2017/07/31).

At 1.2n dependencies on xintcore were removed, so now the package loads only xintkernel (this could have been done earlier).

Also at 1.2n, macros evolved again, the main improvements being in the increased allowable sizes of the input for \xintDecToHex, \xintDecToBin, \xintBinToHex. Use of \csname governed expansion at some places rather than \numexpr with some clean-up after it.

6.1 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from Heiko Oberdiek's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintbinhex.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintkernel.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintbinhex}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintbinhex.sty
      \ifx\w\relax % but xintkernel.sty not yet loaded.
         \def\z{\endgroup\input xintkernel.sty\relax}%
      \fi
    \else
```

```
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xintkernel.sty not yet loaded.
            \def\z{\endgroup\RequirePackage{xintkernel}}%
          \fi
      \else
        \aftergroup\endinput % xintbinhex already loaded.
      \fi
    \fi
  \fi
\z%
\XINTsetupcatcodes% defined in xintkernel.sty
```

6.2 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintbinhex}%
  [2021/03/29 v1.4d Expandable binary and hexadecimal conversions (JFB)]%
```

6.3 Constants, etc...

1.2n switches to \csname-governed expansion at various places.

```
\newcount\xint_c_ii^xv  \xint_c_ii^xv   32768
\newcount\xint_c_ii^xvi \xint_c_ii^xvi  65536
\def\XINT_tmpa #1{\ifx\relax#1\else
  \expandafter\edef\csname XINT_csdth_#1\endcsname
  {\endcsname\ifcase #1 0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or
              8\or 9\or A\or B\or C\or D\or E\or F\fi}%
  \expandafter\XINT_tmpa\fi }%
\XINT_tmpa {0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}\relax
\def\XINT_tmpa #1{\ifx\relax#1\else
  \expandafter\edef\csname XINT_csdtb_#1\endcsname
  {\endcsname\ifcase #1
   0000\or 0001\or 0010\or 0011\or 0100\or 0101\or 0110\or 0111\or
   1000\or 1001\or 1010\or 1011\or 1100\or 1101\or 1110\or 1111\fi}%
  \expandafter\XINT_tmpa\fi }%
\XINT_tmpa {0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}\relax
\let\XINT_tmpa\relax
\expandafter\def\csname XINT_csbth_0000\endcsname {\endcsname0}%
\expandafter\def\csname XINT_csbth_0001\endcsname {\endcsname1}%
\expandafter\def\csname XINT_csbth_0010\endcsname {\endcsname2}%
\expandafter\def\csname XINT_csbth_0011\endcsname {\endcsname3}%
\expandafter\def\csname XINT_csbth_0100\endcsname {\endcsname4}%
\expandafter\def\csname XINT_csbth_0101\endcsname {\endcsname5}%
\expandafter\def\csname XINT_csbth_0110\endcsname {\endcsname6}%
\expandafter\def\csname XINT_csbth_0111\endcsname {\endcsname7}%
\expandafter\def\csname XINT_csbth_1000\endcsname {\endcsname8}%
\expandafter\def\csname XINT_csbth_1001\endcsname {\endcsname9}%
\expandafter\def\csname XINT_csbth_1010\endcsname {\endcsname A}%
\expandafter\def\csname XINT_csbth_1011\endcsname {\endcsname B}%
\expandafter\def\csname XINT_csbth_1100\endcsname {\endcsname C}%
\expandafter\def\csname XINT_csbth_1101\endcsname {\endcsname D}%
```

```
\expandafter\def\csname XINT_csbth_1110\endcsname {\endcsname E}%
\expandafter\def\csname XINT_csbth_1111\endcsname {\endcsname F}%
\let\XINT_csbth_none \endcsname
\expandafter\def\csname XINT_cshtb_0\endcsname {\endcsname0000}%
\expandafter\def\csname XINT_cshtb_1\endcsname {\endcsname0001}%
\expandafter\def\csname XINT_cshtb_2\endcsname {\endcsname0010}%
\expandafter\def\csname XINT_cshtb_3\endcsname {\endcsname0011}%
\expandafter\def\csname XINT_cshtb_4\endcsname {\endcsname0100}%
\expandafter\def\csname XINT_cshtb_5\endcsname {\endcsname0101}%
\expandafter\def\csname XINT_cshtb_6\endcsname {\endcsname0110}%
\expandafter\def\csname XINT_cshtb_7\endcsname {\endcsname0111}%
\expandafter\def\csname XINT_cshtb_8\endcsname {\endcsname1000}%
\expandafter\def\csname XINT_cshtb_9\endcsname {\endcsname1001}%
\def\XINT_cshtb_A {\endcsname1010}%
\def\XINT_cshtb_B {\endcsname1011}%
\def\XINT_cshtb_C {\endcsname1100}%
\def\XINT_cshtb_D {\endcsname1101}%
\def\XINT_cshtb_E {\endcsname1110}%
\def\XINT_cshtb_F {\endcsname1111}%
\let\XINT_cshtb_none \endcsname
```

6.4 Helper macros

6.4.1 \XINT_zeroes_foriv

```
 \romannumeral0\XINT_zeroes_foriv #1\R{0\R}{00\R}{000\R}%
                                  \R{0\R}{00\R}{000\R}\R\W
```

expands to the <empty> or 0 or 00 or 000 needed which when adjoined to #1 extend it to length 4N.

```
\def\XINT_zeroes_foriv #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_R #8\XINT_zeroes_foriv_end\R\XINT_zeroes_foriv
}%
\def\XINT_zeroes_foriv_end\R\XINT_zeroes_foriv #1#2\W
   {\XINT_zeroes_foriv_done #1}%
\def\XINT_zeroes_foriv_done #1\R{ #1}%
```

6.5 \xintDecToHex

Complete rewrite at 1.2m in the 1.2 style. Also, 1.2m is robust against non terminated inputs.
  Improvements of coding at 1.2n, increased maximal size. Again some coding improvement at 1.2o, about 6% speed gain.
  An input without leading zeroes gives an output without leading zeroes.

```
\def\xintDecToHex {\romannumeral0\xintdectohex }%
\def\xintdectohex #1%
{%
    \expandafter\XINT_dth_checkin\romannumeral`&&@#1\xint:
}%
\def\XINT_dth_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_dth_neg
```

```
        -{\XINT_dth_main #1}%
     \krof
}%
\def\XINT_dth_neg {\expandafter-\romannumeral0\XINT_dth_main}%
\def\XINT_dth_main #1\xint:
{%
    \expandafter\XINT_dth_finish
    \romannumeral`&&@\expandafter\XINT_dthb_start
    \romannumeral0\XINT_zeroes_foriv
       #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
    #1\xint_bye\XINT_dth_tohex
}%
\def\XINT_dthb_start #1#2#3#4#5%
{%
    \xint_bye#5\XINT_dthb_small\xint_bye\XINT_dthb_start_a #1#2#3#4#5%
}%
\def\XINT_dthb_small\xint_bye\XINT_dthb_start_a #1\xint_bye#2{#2#1!}%
\def\XINT_dthb_start_a #1#2#3#4#5#6#7#8#9%
{%
    \expandafter\XINT_dthb_again\the\numexpr\expandafter\XINT_dthb_update
    \the\numexpr#1#2#3#4%
    \xint_bye#9\XINT_dthb_lastpass\xint_bye
    #5#6#7#8!\XINT_dthb_exclam\relax\XINT_dthb_nextfour #9%
}%
```

The 1.2n inserted exclamations marks, which when bumping back from \XINT_dthb_again gave rise to a \numexpr-loop which gathered the ! delimited arguments and inserted \expandafter\XINT_dthb_update\the\numexpr dynamically. The 1.2o trick is to insert it here immediately. Then at \XINT_dthb_again the \numexpr will trigger an already prepared chain.

The crux of the thing is handling of #3 at \XINT_dthb_update_a.

```
\def\XINT_dthb_exclam {!\XINT_dthb_exclam\relax
                       \expandafter\XINT_dthb_update\the\numexpr}%
\def\XINT_dthb_update #1!%
{%
    \expandafter\XINT_dthb_update_a
    \the\numexpr (#1+\xint_c_ii^xv)/\xint_c_ii^xvi-\xint_c_i\xint:
    #1\xint:%
}%
\def\XINT_dthb_update_a #1\xint:#2\xint:#3%
{%
    0000+#1\expandafter#3\the\numexpr#2-#1*\xint_c_ii^xvi
}%
```

1.2m and 1.2n had some unduly complicated ending pattern for \XINT_dthb_nextfour as inheritance of a loop needing ! separators which was pruned out at 1.2o (see previous comment).

```
\def\XINT_dthb_nextfour #1#2#3#4#5%
{%
    \xint_bye#5\XINT_dthb_lastpass\xint_bye
    #1#2#3#4!\XINT_dthb_exclam\relax\XINT_dthb_nextfour#5%
}%
\def\XINT_dthb_lastpass\xint_bye #1!#2\xint_bye#3{#1!#3!}%
\def\XINT_dth_tohex
```

```
{%
    \expandafter\expandafter\expandafter\XINT_dth_tohex_a\csname\XINT_tofourhex
}%
\def\XINT_dth_tohex_a\endcsname{!\XINT_dth_tohex!}%
\def\XINT_dthb_again #1!#2#3%
{%
    \ifx#3\relax
          \expandafter\xint_firstoftwo
      \else
          \expandafter\xint_secondoftwo
    \fi
    {\expandafter\XINT_dthb_again
     \the\numexpr
     \ifnum #1>\xint_c_
        \xint_afterfi{\expandafter\XINT_dthb_update\the\numexpr#1}%
     \fi}%
    {\ifnum #1>\xint_c_ \xint_dothis{#2#1!}\fi\xint_orthat{!#2!}}%
}%
\def\XINT_tofourhex #1!%
{%
    \expandafter\XINT_tofourhex_a
    \the\numexpr (#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i\xint:
    #1\xint:
}%
\def\XINT_tofourhex_a #1\xint:#2\xint:
{%
    \expandafter\XINT_tofourhex_c
    \the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i\xint:
    #1\xint:
    \the\numexpr #2-\xint_c_ii^viii*#1!%
}%
\def\XINT_tofourhex_c #1\xint:#2\xint:
{%
    XINT_csdth_#1%
    \csname XINT_csdth_\the\numexpr #2-\xint_c_xvi*#1\relax
    \csname \expandafter\XINT_tofourhex_d
}%
\def\XINT_tofourhex_d #1!%
{%
    \expandafter\XINT_tofourhex_e
    \the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i\xint:
    #1\xint:
}%
\def\XINT_tofourhex_e #1\xint:#2\xint:
{%
    XINT_csdth_#1%
    \csname XINT_csdth_\the\numexpr #2-\xint_c_xvi*#1\endcsname
}%
```

We only clean-up up to 3 zero hexadecimal digits, as output was produced in chunks of 4 hex digits. If input had no leading zero, output will have none either. If input had many leading zeroes, output will have some number (unspecified, but a recipe can be given...) of leading zeroes...

The coding is for varying a bit, I did not check if efficient, it does not matter.

```
\def\XINT_dth_finish !\XINT_dth_tohex!#1#2#3%
{%
    \unless\if#10\xint_dothis{ #1#2#3}\fi
    \unless\if#20\xint_dothis{ #2#3}\fi
    \unless\if#30\xint_dothis{ #3}\fi
    \xint_orthat{ }%
}%
```

6.6 \xintDecToBin

Complete rewrite at 1.2m in the 1.2 style. Also, 1.2m is robust against non terminated inputs.
Revisited at 1.2n like in \xintDecToHex: increased maximal size.
An input without leading zeroes gives an output without leading zeroes.
Most of the code canvas is shared with \xintDecToHex.

```
\def\xintDecToBin {\romannumeral0\xintdectobin }%
\def\xintdectobin #1%
{%
    \expandafter\XINT_dtb_checkin\romannumeral`&&@#1\xint:
}%
\def\XINT_dtb_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_dtb_neg
        -{\XINT_dtb_main #1}%
     \krof
}%
\def\XINT_dtb_neg {\expandafter-\romannumeral0\XINT_dtb_main}%
\def\XINT_dtb_main #1\xint:
{%
    \expandafter\XINT_dtb_finish
    \romannumeral`&&@\expandafter\XINT_dthb_start
    \romannumeral0\XINT_zeroes_foriv
       #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
    #1\xint_bye\XINT_dtb_tobin
}%
\def\XINT_dtb_tobin
{%
    \expandafter\expandafter\expandafter\XINT_dtb_tobin_a\csname\XINT_tosixteenbits
}%
\def\XINT_dtb_tobin_a\endcsname{!\XINT_dtb_tobin!}%
\def\XINT_tosixteenbits #1!%
{%
    \expandafter\XINT_tosixteenbits_a
    \the\numexpr (#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i\xint:
    #1\xint:
}%
\def\XINT_tosixteenbits_a #1\xint:#2\xint:
{%
    \expandafter\XINT_tosixteenbits_c
    \the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i\xint:
    #1\xint:
    \the\numexpr #2-\xint_c_ii^viii*#1!%
```

```
}%
\def\XINT_tosixteenbits_c #1\xint:#2\xint:
{%
    XINT_csdtb_#1%
    \csname XINT_csdtb_\the\numexpr #2-\xint_c_xvi*#1\relax
    \csname \expandafter\XINT_tosixteenbits_d
}%
\def\XINT_tosixteenbits_d #1!%
{%
    \expandafter\XINT_tosixteenbits_e
    \the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i\xint:
    #1\xint:
}%
\def\XINT_tosixteenbits_e #1\xint:#2\xint:
{%
    XINT_csdtb_#1%
    \csname XINT_csdtb_\the\numexpr #2-\xint_c_xvi*#1\endcsname
}%
\def\XINT_dtb_finish !\XINT_dtb_tobin!#1#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_dtb_finish_a\the\numexpr #1#2#3#4#5#6#7#8\relax
}%
\def\XINT_dtb_finish_a #1{%
\def\XINT_dtb_finish_a ##1##2##3##4##5##6##7##8##9%
{%
    \expandafter#1\the\numexpr ##1##2##3##4##5##6##7##8##9\relax
}}\XINT_dtb_finish_a { }%
```

6.7 \xintHexToDec

Completely (and belatedly) rewritten at 1.2m in the 1.2 style.
  1.2m version robust against non terminated inputs, but there is no primitive from TeX which may generate hexadecimal digits and provoke expansion ahead, afaik, except of course if decimal digits are treated as hexadecimal. This robustness is not on purpose but from need to expand argument and then grab it again. So we do it safely.
  Increased maximal size at 1.2n.
  1.2m version robust against non terminated inputs.
  An input without leading zeroes gives an output without leading zeroes.

```
\def\xintHexToDec {\romannumeral0\xinthextodec }%
\def\xinthextodec #1%
{%
    \expandafter\XINT_htd_checkin\romannumeral`&&@#1\xint:
}%
\def\XINT_htd_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_htd_neg
        -{\XINT_htd_main #1}%
     \krof
}%
\def\XINT_htd_neg {\expandafter-\romannumeral0\XINT_htd_main}%
\def\XINT_htd_main #1\xint:
```

```
{%
    \expandafter\XINT_htd_startb
    \the\numexpr\expandafter\XINT_htd_starta
    \romannumeral0\XINT_zeroes_foriv
       #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
    #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\relax
}%
\def\XINT_htd_starta #1#2#3#4{"#1#2#3#4+100000!}%
\def\XINT_htd_startb 1#1%
{%
    \if#10\expandafter\XINT_htd_startba\else
          \expandafter\XINT_htd_startbb
    \fi 1#1%
}%
\def\XINT_htd_startba 10#1!{\XINT_htd_again #1%
    \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\XINT_htd_nextfour}%
\def\XINT_htd_startbb 1#1#2!{\XINT_htd_again #1!#2%
    \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\XINT_htd_nextfour}%
```

It is a bit annoying to grab all to the end here. I have a version, modeled on the 1.2n variant of \xintDecToHex which solved that problem there, but it did not prove enough if at all faster in my brief testing and it had the defect of a reduced maximal allowed size of the input.

```
\def\XINT_htd_again #1\XINT_htd_nextfour #2%
{%
    \xint_bye #2\XINT_htd_finish\xint_bye
    \expandafter\XINT_htd_A\the\numexpr
    \XINT_htd_a #1\XINT_htd_nextfour #2%
}%
\def\XINT_htd_a #1!#2!#3!#4!#5!#6!#7!#8!#9!%
{%
    #1\expandafter\XINT_htd_update
    \the\numexpr #2\expandafter\XINT_htd_update
    \the\numexpr #3\expandafter\XINT_htd_update
    \the\numexpr #4\expandafter\XINT_htd_update
    \the\numexpr #5\expandafter\XINT_htd_update
    \the\numexpr #6\expandafter\XINT_htd_update
    \the\numexpr #7\expandafter\XINT_htd_update
    \the\numexpr #8\expandafter\XINT_htd_update
    \the\numexpr #9\expandafter\XINT_htd_update
    \the\numexpr \XINT_htd_a
}%
\def\XINT_htd_nextfour #1#2#3#4%
{%
    *\xint_c_ii^xvi+"#1#2#3#4+1000000000\relax\xint_bye!%
    2!3!4!5!6!7!8!9!\xint_bye\XINT_htd_nextfour
}%
```

If the innocent looking commented out #6 is left in the pattern as was the case at 1.2m, the maximal size becomes limited at 5538 digits, not 8298! (with parameter stack size = 10000.)

```
\def\XINT_htd_update 1#1#2#3#4#5%#6!%
{%
    *\xint_c_ii^xvi+10000#1#2#3#4#5!%#6!%
```

```
}%
\def\XINT_htd_A 1#1%
{%
    \if#10\expandafter\XINT_htd_Aa\else
          \expandafter\XINT_htd_Ab
    \fi 1#1%
}%
\def\XINT_htd_Aa 10#1#2#3#4{\XINT_htd_again #1#2#3#4!}%
\def\XINT_htd_Ab 1#1#2#3#4#5{\XINT_htd_again #1!#2#3#4#5!}%
\def\XINT_htd_finish\xint_bye
    \expandafter\XINT_htd_A\the\numexpr \XINT_htd_a #1\XINT_htd_nextfour
{%
    \expandafter\XINT_htd_finish_cuz\the\numexpr0\XINT_htd_unsep_loop #1%
}%
\def\XINT_htd_unsep_loop #1!#2!#3!#4!#5!#6!#7!#8!#9!%
{%
    \expandafter\XINT_unsep_clean
    \the\numexpr 1#1#2\expandafter\XINT_unsep_clean
    \the\numexpr 1#3#4\expandafter\XINT_unsep_clean
    \the\numexpr 1#5#6\expandafter\XINT_unsep_clean
    \the\numexpr 1#7#8\expandafter\XINT_unsep_clean
    \the\numexpr 1#9\XINT_htd_unsep_loop_a
}%
\def\XINT_htd_unsep_loop_a #1!#2!#3!#4!#5!#6!#7!#8!#9!%
{%
    #1\expandafter\XINT_unsep_clean
    \the\numexpr 1#2#3\expandafter\XINT_unsep_clean
    \the\numexpr 1#4#5\expandafter\XINT_unsep_clean
    \the\numexpr 1#6#7\expandafter\XINT_unsep_clean
    \the\numexpr 1#8#9\XINT_htd_unsep_loop
}%
\def\XINT_unsep_clean 1{\relax}% also in xintcore
\def\XINT_htd_finish_cuz #1{%
\def\XINT_htd_finish_cuz ##1##2##3##4##5%
   {\expandafter#1\the\numexpr ##1##2##3##4##5\relax}%
}\XINT_htd_finish_cuz{ }%
```

6.8 \xintBinToDec

Redone entirely for 1.2m. Starts by converting to hexadecimal first.
Increased maximal size at 1.2n.
An input without leading zeroes gives an output without leading zeroes.
Robust against non-terminated input.

```
\def\xintBinToDec {\romannumeral0\xintbintodec }%
\def\xintbintodec #1%
{%
    \expandafter\XINT_btd_checkin\romannumeral`&&@#1\xint:
}%
\def\XINT_btd_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_btd_N
```

```
        -{\XINT_btd_main #1}%
     \krof
}%
\def\XINT_btd_N {\expandafter-\romannumeral0\XINT_btd_main }%
\def\XINT_btd_main #1\xint:
{%
    \csname XINT_btd_htd\csname\expandafter\XINT_bth_loop
    \romannumeral0\XINT_zeroes_foriv
       #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
    #1\xint_bye2345678\xint_bye none\endcsname\xint:
}%
\def\XINT_btd_htd #1\xint:
{%
    \expandafter\XINT_htd_startb
    \the\numexpr\expandafter\XINT_htd_starta
    \romannumeral0\XINT_zeroes_foriv
       #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
    #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\relax
}%
```

6.9 \xintBinToHex

Complete rewrite for 1.2m. But input for 1.2m version limited to about 13320 binary digits (expansion depth=10000).
  Again redone for 1.2n for \csname governed expansion: increased maximal size.
  Size of output is ceil(size(input)/4), leading zeroes in output (inherited from the input) are not trimmed.
  An input without leading zeroes gives an output without leading zeroes.
  Robust against non-terminated input.

```
\def\xintBinToHex {\romannumeral0\xintbintohex }%
\def\xintbintohex #1%
{%
    \expandafter\XINT_bth_checkin\romannumeral`&&@#1\xint:
}%
\def\XINT_bth_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_bth_N
        -{\XINT_bth_main #1}%
     \krof
}%
\def\XINT_bth_N {\expandafter-\romannumeral0\XINT_bth_main }%
\def\XINT_bth_main #1\xint:
{%
    \csname space\csname\expandafter\XINT_bth_loop
    \romannumeral0\XINT_zeroes_foriv
       #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
    #1\xint_bye2345678\xint_bye none\endcsname
}%
\def\XINT_bth_loop #1#2#3#4#5#6#7#8%
{%
            XINT_csbth_#1#2#3#4%
```

```
    \csname XINT_csbth_#5#6#7#8%
    \csname\XINT_bth_loop
}%
```

6.10 \xintHexToBin

Completely rewritten for 1.2m.
  Attention this macro is not robust against arguments expanding after themselves.
  Only up to three zeros are removed on front of output: if the input had a leading zero, there will be a leading zero (and then possibly 4n of them if inputs had more leading zeroes) on output.
  Rewritten again at 1.2n for \csname governed expansion.

```
\def\xintHexToBin {\romannumeral0\xinthextobin }%
\def\xinthextobin #1%
{%
    \expandafter\XINT_htb_checkin\romannumeral`&&@#1%
    \xint_bye 23456789\xint_bye none\endcsname
}%
\def\XINT_htb_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_htb_N
        -{\XINT_htb_main #1}%
     \krof
}%
\def\XINT_htb_N {\expandafter-\romannumeral0\XINT_htb_main }%
\def\XINT_htb_main {\csname XINT_htb_cuz\csname\XINT_htb_loop}%
\def\XINT_htb_loop #1#2#3#4#5#6#7#8#9%
{%
             XINT_cshtb_#1%
     \csname XINT_cshtb_#2%
     \csname XINT_cshtb_#3%
     \csname XINT_cshtb_#4%
     \csname XINT_cshtb_#5%
     \csname XINT_cshtb_#6%
     \csname XINT_cshtb_#7%
     \csname XINT_cshtb_#8%
     \csname XINT_cshtb_#9%
     \csname \XINT_htb_loop
}%
\def\XINT_htb_cuz #1{%
\def\XINT_htb_cuz ##1##2##3##4%
   {\expandafter#1\the\numexpr##1##2##3##4\relax}%
}\XINT_htb_cuz { }%
```

6.11 \xintCHexToBin

The 1.08 macro had same functionality as \xintHexToBin, and slightly different code, the 1.2m version has the same code as \xintHexToBin except that it does not remove leading zeros from output: if the input had N hexadecimal digits, the output will have exactly 4N binary digits.
  Rewritten again at 1.2n for \csname governed expansion.

```
\def\xintCHexToBin {\romannumeral0\xintchextobin }%
```

```
\def\xintchextobin #1%
{%
    \expandafter\XINT_chtb_checkin\romannumeral`&&@#1%
    \xint_bye 23456789\xint_bye none\endcsname
}%
\def\XINT_chtb_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_chtb_N
        -{\XINT_chtb_main #1}%
     \krof
}%
\def\XINT_chtb_N {\expandafter-\romannumeral0\XINT_chtb_main }%
\def\XINT_chtb_main {\csname space\csname\XINT_htb_loop}%
\XINT_restorecatcodes_endinput%
```

7 Package xintgcd implementation

The commenting is currently (2021/03/29) very sparse.

Release 1.09h has modified a bit the \xintTypesetEuclideAlgorithm and \xintTypesetBezoutAlgorithm layout with respect to line indentation in particular. And they use the xinttools \xintloop rather than the Plain TEX or LATEX's \loop.

Breaking change at 1.2p: \xintBezout{A}{B} formerly had output {A}{B}{U}{V}{D} with AU-BV=D, now it is {U}{V}{D} with AU+BV=D.

From 1.1 to 1.3f the package loaded only xintcore. At 1.4 it now automatically loads both of xint and xinttools (the latter being in fact a requirement of \xintTypesetEuclideAlgorithm and \xintTypesetBezoutAlgorithm since 1.09h).

Changed at 1.4!

At 1.4 \xintGCD, \xintLCM, \xintGCDof, and \xintLCMof are *removed* from the package:they are provided only by xintfrac and they handle general fractions, not only integers.

The original integer-only macros have been renamed into respectively \xintiiGCD, \xintiiLCM, \xintiiGCDof, and \xintiiLCMof and got relocated into xint package.

7.1 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from Heiko Oberdiek's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\z{\endgroup}%
  \expandafter\let\expandafter\x\csname ver@xintgcd.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \expandafter\let\expandafter\t\csname ver@xinttools.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintgcd}{\numexpr not available, aborting input}%
     \aftergroup\endinput
```

```
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintgcd.sty
      \ifx\w\relax % but xint.sty not yet loaded.
         \expandafter\def\expandafter\z\expandafter{\z\input xint.sty\relax}%
      \fi
      \ifx\t\relax % but xinttools.sty not yet loaded.
         \expandafter\def\expandafter\z\expandafter{\z\input xinttools.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xint.sty not yet loaded.
            \expandafter\def\expandafter\z\expandafter{\z\RequirePackage{xint}}%
          \fi
          \ifx\t\relax % xinttools.sty not yet loaded.
            \expandafter\def\expandafter\z\expandafter{\z\RequirePackage{xinttools}}%
          \fi
      \else
        \aftergroup\endinput % xintgcd already loaded.
      \fi
    \fi
  \fi
\z%
\XINTsetupcatcodes% defined in xintkernel.sty
```

7.2 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintgcd}%
  [2021/03/29 v1.4d Euclide algorithm with xint package (JFB)]%
```

7.3 \xintBezout

```
\xintBezout{#1}{#2} produces {U}{V}{D} with UA+VB=D, D = PGCD(A,B) (non-positive), where #1 and
#2 f-expand to big integers A and B.
  I had not checked this macro for about three years when I realized in January 2017 that \xintBe-
zout{A}{B} was buggy for the cases A = 0 or B = 0. I fixed that blemish in 1.2l but overlooked the
other blemish that \xintBezout{A}{B} with A multiple of B produced a coefficient U as -0 in place
of 0.
  Hence I rewrote again for 1.2p. On this occasion I modified the output of the macro to be
{U}{V}{D} with AU+BV=D, formerly it was {A}{B}{U}{V}{D} with AU - BV = D. This is quite break-
ing change!
  Note in particular change of sign of V.
  I don't know why I had designed this macro to contain {A}{B} in its output. Perhaps I initially
intended to output {A//D}{B//D} (but forgot), as this is actually possible from outcome of the
last iteration, with no need of actually dividing. Current code however arranges to skip this last
update, as U and V are already furnished by the iteration prior to realizing that the last non-zero
remainder was found.
  Also 1.2l raised InvalidOperation if both A and B vanished, but I removed this behaviour at 1.2p.

\def\xintBezout {\romannumeral0\xintbezout }%
\def\xintbezout #1%
{%
```

```
    \expandafter\XINT_bezout\expandafter {\romannumeral0\xintnum{#1}}%
}%
\def\XINT_bezout #1#2%
{%
    \expandafter\XINT_bezout_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
```

#3#4 = A, #1#2=B. Micro improvement for 1.21.

```
\def\XINT_bezout_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerosfork
     #1#3\XINT_bezout_botharezero
      #10\XINT_bezout_secondiszero
      #30\XINT_bezout_firstiszero
       00\xint_UDsignsfork
    \krof
          #1#3\XINT_bezout_minusminus % A < 0, B < 0
           #1-\XINT_bezout_minusplus  % A > 0, B < 0
           #3-\XINT_bezout_plusminus  % A < 0, B > 0
            --\XINT_bezout_plusplus   % A > 0, B > 0
    \krof
    {#2}{#4}#1#3% #1#2=B, #3#4=A
}%
\def\XINT_bezout_botharezero #1\krof#2#300{{0}{0}{0}}%
\def\XINT_bezout_firstiszero #1\krof#2#3#4#5%
{%
    \xint_UDsignfork
      #4{{0}{-1}{#2}}%
       -{{0}{1}{#4#2}}%
    \krof
}%
\def\XINT_bezout_secondiszero #1\krof#2#3#4#5%
{%
    \xint_UDsignfork
       #5{{-1}{0}{#3}}%
        -{{1}{0}{#5#3}}%
    \krof
}%
```

#4#2= A < 0, #3#1 = B < 0

```
\def\XINT_bezout_minusminus #1#2#3#4%
{%
    \expandafter\XINT_bezout_mm_post
    \romannumeral0\expandafter\XINT_bezout_preloop_a
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
\def\XINT_bezout_mm_post #1#2%
{%
    \expandafter\XINT_bezout_mm_postb\expandafter
    {\romannumeral0\xintiiopp{#2}}{\romannumeral0\xintiiopp{#1}}%
}%
\def\XINT_bezout_mm_postb #1#2{\expandafter{#2}{#1}}%
```

```
   minusplus #4#2= A > 0, B < 0

105 \def\XINT_bezout_minusplus #1#2#3#4%
106 {%
107     \expandafter\XINT_bezout_mp_post
108     \romannumeral0\expandafter\XINT_bezout_preloop_a
109     \romannumeral0\XINT_div_prepare {#1}{#4#2}{#1}%
110 }%
111 \def\XINT_bezout_mp_post #1#2%
112 {%
113     \expandafter\xint_exchangetwo_keepbraces\expandafter
114     {\romannumeral0\xintiiopp {#2}}{#1}%
115 }%

   plusminus A < 0, B > 0

116 \def\XINT_bezout_plusminus #1#2#3#4%
117 {%
118     \expandafter\XINT_bezout_pm_post
119     \romannumeral0\expandafter\XINT_bezout_preloop_a
120     \romannumeral0\XINT_div_prepare {#3#1}{#2}{#3#1}%
121 }%
122 \def\XINT_bezout_pm_post #1{\expandafter{\romannumeral0\xintiiopp{#1}}}%

   plusplus, B = #3#1 > 0, A = #4#2 > 0

123 \def\XINT_bezout_plusplus #1#2#3#4%
124 {%
125     \expandafter\XINT_bezout_preloop_a
126     \romannumeral0\XINT_div_prepare {#3#1}{#4#2}{#3#1}%
127 }%

    n = 0: BA1001 (B, A, e=1, vv, uu, v, u)
    r(1)=B, r(0)=A, après n étapes {r(n+1)}{r(n)}{vv}{uu}{v}{u}
    q(n) quotient de r(n-1) par r(n)
    si reste nul, exit et renvoie U = -e*uu, V = e*vv, A*U+B*V=D
    sinon mise à jour
      vv, v = q * vv + v, vv
      uu, u = q * uu + u, uu
      e = -e
    puis calcul quotient reste et itération
We arrange for \xintiiMul sub-routine to be called only with positive arguments, thus skipping
some un-needed sign parsing there. For that though we have to screen out the special cases A divides
B, or B divides A. And we first want to exchange A and B if A < B. These special cases are the only
one possibly leading to U or V zero (for A and B positive which is the case here.) Thus the general
case always leads to non-zero U and V's and assigning a final sign is done simply adding a - to one
of them, with no fear of producing -0.

128 \def\XINT_bezout_preloop_a #1#2#3%
129 {%
130     \if0#1\xint_dothis\XINT_bezout_preloop_exchange\fi
131     \if0#2\xint_dothis\XINT_bezout_preloop_exit\fi
132     \xint_orthat{\expandafter\XINT_bezout_loop_B}%
133     \romannumeral0\XINT_div_prepare {#2}{#3}{#2}{#1}110%
```

```
}%
\def\XINT_bezout_preloop_exit
    \romannumeral0\XINT_div_prepare #1#2#3#4#5#6#7%
{%
    {0}{1}{#2}%
}%
\def\XINT_bezout_preloop_exchange
{%
    \expandafter\xint_exchangetwo_keepbraces
    \romannumeral0\expandafter\XINT_bezout_preloop_A
}%
\def\XINT_bezout_preloop_A #1#2#3#4%
{%
    \if0#2\xint_dothis\XINT_bezout_preloop_exit\fi
    \xint_orthat{\expandafter\XINT_bezout_loop_B}%
    \romannumeral0\XINT_div_prepare {#2}{#3}{#2}{#1}%
}%
\def\XINT_bezout_loop_B #1#2%
{%
    \if0#2\expandafter\XINT_bezout_exitA
     \else\expandafter\XINT_bezout_loop_C
    \fi {#1}{#2}%
}%
```

We use the fact that the \romannumeral-`0 (or equivalent) done by \xintiiadd will absorb the initial space token left by \XINT_mul_plusplus in its output.

We arranged for operands here to be always positive which is needed for \XINT_mul_plusplus entry point (last time I checked...). Admittedly this kind of optimization is not good for maintenance of code, but I can't resist temptation of limiting the shuffling around of tokens...

```
\def\XINT_bezout_loop_C #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_bezout_loop_D\expandafter
        {\romannumeral0\xintiiadd{\XINT_mul_plusplus{}{}#1\xint:#4\xint:}{#6}}%
        {\romannumeral0\xintiiadd{\XINT_mul_plusplus{}{}#1\xint:#5\xint:}{#7}}%
    {#2}{#3}{#4}{#5}%
}%
\def\XINT_bezout_loop_D #1#2%
{%
    \expandafter\XINT_bezout_loop_E\expandafter{#2}{#1}%
}%
\def\XINT_bezout_loop_E #1#2#3#4%
{%
    \expandafter\XINT_bezout_loop_b
    \romannumeral0\XINT_div_prepare {#3}{#4}{#3}{#2}{#1}%
}%
\def\XINT_bezout_loop_b #1#2%
{%
    \if0#2\expandafter\XINT_bezout_exita
     \else\expandafter\XINT_bezout_loop_c
    \fi {#1}{#2}%
}%
\def\XINT_bezout_loop_c #1#2#3#4#5#6#7%
```

```
{%
    \expandafter\XINT_bezout_loop_d\expandafter
        {\romannumeral0\xintiiadd{\XINT_mul_plusplus{}{}#1\xint:#4\xint:}{#6}}%
        {\romannumeral0\xintiiadd{\XINT_mul_plusplus{}{}#1\xint:#5\xint:}{#7}}%
    {#2}{#3}{#4}{#5}%
}%
\def\XINT_bezout_loop_d #1#2%
{%
    \expandafter\XINT_bezout_loop_e\expandafter{#2}{#1}%
}%
\def\XINT_bezout_loop_e #1#2#3#4%
{%
    \expandafter\XINT_bezout_loop_B
    \romannumeral0\XINT_div_prepare {#3}{#4}{#3}{#2}{#1}%
}%
```

```
  sortir U, V, D mais on a travaillé avec vv, uu, v, u dans cet ordre.
The code is structured so that #4 and #5 are guaranteed non-zero if we exit here, hence we can not
create a -0 in output.
```

```
\def\XINT_bezout_exita #1#2#3#4#5#6#7{{-#5}{#4}{#3}}%
\def\XINT_bezout_exitA #1#2#3#4#5#6#7{{#5}{-#4}{#3}}%
```

7.4 \xintEuclideAlgorithm

```
Pour Euclide: {N}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
u<2n> = u<2n+3>u<2n+2> + u<2n+4> à la n ième étape.
  Formerly, used \xintiabs, but got deprecated at 1.2o.
```

```
\def\xintEuclideAlgorithm {\romannumeral0\xinteuclidealgorithm }%
\def\xinteuclidealgorithm #1%
{%
    \expandafter\XINT_euc\expandafter{\romannumeral0\xintiiabs{\xintNum{#1}}}%
}%
\def\XINT_euc #1#2%
{%
    \expandafter\XINT_euc_fork\romannumeral0\xintiiabs{\xintNum{#2}}\Z #1\Z
}%
```

```
  Ici #3#4=A, #1#2=B
```

```
\def\XINT_euc_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_euc_BisZero
      #3\XINT_euc_AisZero
       0\XINT_euc_a
    \krof
    {0}{#1#2}{#3#4}{{#3#4}{#1#2}}{}\Z
}%
```

```
  Le {} pour protéger {{A}{B}} si on s'arrête après une étape (B divise A). On va renvoyer:
{N}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
```

```
\def\XINT_euc_AisZero #1#2#3#4#5#6{{1}{0}{#2}{#2}{0}{0}}%
\def\XINT_euc_BisZero #1#2#3#4#5#6{{1}{0}{#3}{#3}{0}{0}}%

  {n}{rn}{an}{{qn}{rn}}...{{A}{B}}{}\Z
a(n) = r(n-1). Pour n=0 on a juste {0}{B}{A}{{A}{B}}{}\Z
\XINT_div_prepare {u}{v} divise v par u

\def\XINT_euc_a #1#2#3%
{%
    \expandafter\XINT_euc_b\the\numexpr #1+\xint_c_i\expandafter.%
    \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
}%

  {n+1}{q(n+1)}{r(n+1)}{rn}{{qn}{rn}}...

\def\XINT_euc_b #1.#2#3#4%
{%
    \XINT_euc_c #3\Z {#1}{#3}{#4}{{#2}{#3}}%
}%

  r(n+1)\Z {n+1}{r(n+1)}{r(n)}{{q(n+1)}{r(n+1)}}{{qn}{rn}}...
Test si r(n+1) est nul.

\def\XINT_euc_c #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_euc_end0\XINT_euc_a
}%

  {n+1}{r(n+1)}{r(n)}{{q(n+1)}{r(n+1)}}...{}\Z Ici r(n+1) = 0. On arrête on se prépare à inverser
{n+1}{0}{r(n)}{{q(n+1)}{r(n+1)}}.....{{q1}{r1}}{{A}{B}}{}\Z
On veut renvoyer: {N=n+1}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}

\def\XINT_euc_end0\XINT_euc_a #1#2#3#4\Z%
{%
    \expandafter\XINT_euc_end_a
    \romannumeral0%
    \XINT_rord_main {}#4{{#1}{#3}}%
    \xint:
      \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_bye\xint_bye\xint_bye\xint_bye
    \xint:
}%
\def\XINT_euc_end_a #1#2#3{{#1}{#3}{#2}}%
```

7.5 \xintBezoutAlgorithm

```
Pour Bezout: objectif, renvoyer
{N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
alpha0=1, beta0=0, alpha(-1)=0, beta(-1)=1

\def\xintBezoutAlgorithm {\romannumeral0\xintbezoutalgorithm }%
\def\xintbezoutalgorithm #1%
{%
    \expandafter \XINT_bezalg
```

```
    \expandafter{\romannumeral0\xintiiabs{\xintNum{#1}}}%
}%
\def\XINT_bezalg #1#2%
{%
    \expandafter\XINT_bezalg_fork\romannumeral0\xintiiabs{\xintNum{#2}}\Z #1\Z
}%
```

Ici #3#4=A, #1#2=B

```
\def\XINT_bezalg_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_bezalg_BisZero
      #3\XINT_bezalg_AisZero
       0\XINT_bezalg_a
    \krof
    0{#1#2}{#3#4}1001{{#3#4}{#1#2}}{}\Z
}%
\def\XINT_bezalg_AisZero #1#2#3\Z{{1}{0}{0}{1}{#2}{#2}{1}{0}{0}{0}{0}{1}}%
\def\XINT_bezalg_BisZero #1#2#3#4\Z{{1}{0}{0}{1}{#3}{#3}{1}{0}{0}{0}{0}{1}}%
```

pour préparer l'étape n+1 il faut {n}{r(n)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)} {{q(n)}{r(n)}{alpha(n)}{beta(n)}}... division de #3 par #2

```
\def\XINT_bezalg_a #1#2#3%
{%
    \expandafter\XINT_bezalg_b\the\numexpr #1+\xint_c_i\expandafter.%
    \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
}%
```

{n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}...

```
\def\XINT_bezalg_b #1.#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_bezalg_c\expandafter
     {\romannumeral0\xintiiadd {\xintiiMul {#6}{#2}}{#8}}%
     {\romannumeral0\xintiiadd {\xintiiMul {#5}{#2}}{#7}}%
     {#1}{#2}{#3}{#4}{#5}{#6}%
}%
```

{beta(n+1)}{alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}

```
\def\XINT_bezalg_c #1#2#3#4#5#6%
{%
    \expandafter\XINT_bezalg_d\expandafter {#2}{#3}{#4}{#5}{#6}{#1}%
}%
```

{alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{beta(n+1)}

```
\def\XINT_bezalg_d #1#2#3#4#5#6#7#8%
{%
    \XINT_bezalg_e #4\Z {#2}{#4}{#5}{#1}{#6}{#7}{#8}{{#3}{#4}{#1}{#6}}%
}%
```

```
  r(n+1)\Z {n+1}{r(n+1)}{r(n)}{alpha(n+1)}{beta(n+1)}
{alpha(n)}{beta(n)}{q,r,alpha,beta(n+1)}
Test si r(n+1) est nul.

\def\XINT_bezalg_e #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_bezalg_end0\XINT_bezalg_a
}%

  Ici r(n+1) = 0. On arrête on se prépare à inverser.
{n+1}{r(n+1)}{r(n)}{alpha(n+1)}{beta(n+1)}{alpha(n)}{beta(n)}
{q,r,alpha,beta(n+1)}...{{A}{B}}{}\Z
On veut renvoyer
{N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}

\def\XINT_bezalg_end0\XINT_bezalg_a #1#2#3#4#5#6#7#8\Z
{%
    \expandafter\XINT_bezalg_end_a
    \romannumeral0%
    \XINT_rord_main {}#8{{#1}{#3}}%
    \xint:
      \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_bye\xint_bye\xint_bye\xint_bye
    \xint:
}%

  {N}{D}{A}{B}{q1}{r1}{alpha1=q1}{beta1=1}{q2}{r2}{alpha2}{beta2}
....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
On veut renvoyer
{N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}

\def\XINT_bezalg_end_a #1#2#3#4{{#1}{#3}{0}{1}{#2}{#4}{1}{0}}%
```

7.6 \xintTypesetEuclideAlgorithm

```
TYPESETTING
  Organisation:
  {N}{A}{D}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
\U1 = N = nombre d'étapes, \U3 = PGCD, \U2 = A, \U4=B q1 = \U5, q2 = \U7 --> qn = \U<2n+3>, rn =
\U<2n+4> bn = rn. B = r0. A=r(-1)
  r(n-2) = q(n)r(n-1)+r(n) (n e étape)
  \U{2n} = \U{2n+3} \times \U{2n+2} + \U{2n+4}, n e étape. (avec n entre 1 et N)
  1.09h uses \xintloop, and \par rather than \endgraf; and \par rather than \hfill\break

\def\xintTypesetEuclideAlgorithm {%
    \unless\ifdefined\xintAssignArray
       \errmessage
        {xintgcd: package xinttools is required for \string\xintTypesetEuclideAlgorithm}%
       \expandafter\xint_gobble_iii
    \fi
    \XINT_TypesetEuclideAlgorithm
}%
```

```
\def\XINT_TypesetEuclideAlgorithm #1#2%
{% l'algo remplace #1 et #2 par |#1| et |#2|
  \par
  \begingroup
    \xintAssignArray\xintEuclideAlgorithm {#1}{#2}\to\U
    \edef\A{\U2}\edef\B{\U4}\edef\N{\U1}%
    \setbox 0 \vbox{\halign {$##$\cr \A\cr \B \cr}}%
    \count 255 1
    \xintloop
      \indent\hbox to \wd 0 {\hfil$\U{\numexpr 2*\count255\relax}$}%
      ${} =  \U{\numexpr 2*\count255 + 3\relax}
      \times \U{\numexpr 2*\count255 + 2\relax}
          +  \U{\numexpr 2*\count255 + 4\relax}$%
    \ifnum \count255 < \N
      \par
      \advance \count255 1
    \repeat
  \endgroup
}%
```

7.7 \xintTypesetBezoutAlgorithm

```
Pour Bezout on a: {N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
  Donc 4N+8 termes: U1 = N, U2= A, U5=D, U6=B, q1 = U9, qn = U{4n+5}, n au moins 1
rn = U{4n+6}, n au moins -1
alpha(n) = U{4n+7}, n au moins -1
beta(n) = U{4n+8}, n au moins -1
  1.09h uses \xintloop, and \par rather than \endgraf; and no more \parindent0pt
```

```
\def\xintTypesetBezoutAlgorithm {%
    \unless\ifdefined\xintAssignArray
       \errmessage
        {xintgcd: package xinttools is required for \string\xintTypesetBezoutAlgorithm}%
       \expandafter\xint_gobble_iii
    \fi
    \XINT_TypesetBezoutAlgorithm
}%
\def\XINT_TypesetBezoutAlgorithm #1#2%
{%
  \par
  \begingroup
    \xintAssignArray\xintBezoutAlgorithm {#1}{#2}\to\BEZ
    \edef\A{\BEZ2}\edef\B{\BEZ6}\edef\N{\BEZ1}% A = |#1|, B = |#2|
    \setbox 0 \vbox{\halign {$##$\cr \A\cr \B \cr}}%
    \count255 1
    \xintloop
      \indent\hbox to \wd 0 {\hfil$\BEZ{4*\count255 - 2}$}%
      ${} =  \BEZ{4*\count255 + 5}
      \times \BEZ{4*\count255 + 2}
          +  \BEZ{4*\count255 + 6}$\hfill\break
      \hbox to \wd 0 {\hfil$\BEZ{4*\count255 +7}$}%
      ${} = \BEZ{4*\count255 + 5}
```

```
      \times \BEZ{4*\count255 + 3}
          +  \BEZ{4*\count255 - 1}$\hfill\break
      \hbox to \wd 0 {\hfil$\BEZ{4*\count255 +8}$}%
      ${} =  \BEZ{4*\count255 + 5}
      \times \BEZ{4*\count255 + 4}
          +  \BEZ{4*\count255 }$
      \par
    \ifnum \count255 < \N
    \advance \count255 1
  \repeat
    \edef\U{\BEZ{4*\N + 4}}%
    \edef\V{\BEZ{4*\N + 3}}%
    \edef\D{\BEZ5}%
    \ifodd\N
       $\U\times\A  - \V\times \B = -\D$%
    \else
       $\U\times\A - \V\times\B = \D$%
    \fi
    \par
  \endgroup
}%
\XINT_restorecatcodes_endinput%
```

8 Package xintfrac implementation

The commenting is currently (2021/03/29) very sparse.

8.1 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified. The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintfrac.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintfrac}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintfrac.sty
      \ifx\w\relax % but xint.sty not yet loaded.
         \def\z{\endgroup\input xint.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xint.sty not yet loaded.
            \def\z{\endgroup\RequirePackage{xint}}%
          \fi
      \else
        \aftergroup\endinput % xintfrac already loaded.
      \fi
    \fi
```

```
  \fi
\z%
\XINTsetupcatcodes% defined in xintkernel.sty
```

8.2 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintfrac}%
  [2021/03/29 v1.4d Expandable operations on fractions (JFB)]%
```

8.3 \XINT_cntSgnFork

1.09i. Used internally, #1 must expand to \m@ne, \z@, or \@ne or equivalent. \XINT_cntSgnFork does not insert a romannumeral stopper.

```
\def\XINT_cntSgnFork #1%
{%
    \ifcase #1\expandafter\xint_secondofthree
             \or\expandafter\xint_thirdofthree
           \else\expandafter\xint_firstofthree
    \fi
}%
```

8.4 \xintLen

The used formula is disputable, the idea is that A/1 and A should have same length. Venerable code rewritten for 1.2i, following updates to \xintLength in xintkernel.sty. And sadly, I forgot on this occasion that this macro is not supposed to count the sign... Fixed in 1.2k.

```
\def\xintLen {\romannumeral0\xintlen }%
\def\xintlen #1%
{%
    \expandafter\XINT_flen\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_flen#1{\def\XINT_flen ##1##2##3%
{%
    \expandafter#1%

    \the\numexpr \XINT_abs##1+%
    \XINT_len_fork ##2##3\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye-\xint_c_i
    \relax
}}\XINT_flen{ }%
```

8.5 \XINT_outfrac

Months later (2014/10/22): perhaps I should document what this macro does before I forget? from {e}{N}{D} it outputs N/D[e], checking in passing if D=0 or if N=0. It also makes sure D is not < 0. I am not sure but I don't think there is any place in the code which could call \XINT_outfrac with a D < 0, but I should check.

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

```
{%
    \ifcase\XINT_cntSgn #3\xint:
        \expandafter \XINT_outfrac_divisionbyzero
    \or
        \expandafter \XINT_outfrac_P
    \else
        \expandafter \XINT_outfrac_N
    \fi
    {#2}{#3}[#1]%
}%
\def\XINT_outfrac_divisionbyzero #1#2%
{%
    \XINT_signalcondition{DivisionByZero}{Division of #1 by #2}{}{0/1[0]}%
}%
\def\XINT_outfrac_P#1{%
\def\XINT_outfrac_P ##1##2%
   {\if0\XINT_Sgn ##1\xint:\expandafter\XINT_outfrac_Zero\fi#1##1/##2}%
}\XINT_outfrac_P{ }%
\def\XINT_outfrac_Zero #1[#2]{ 0/1[0]}%
\def\XINT_outfrac_N #1#2%
{%
    \expandafter\XINT_outfrac_N_a\expandafter
    {\romannumeral0\XINT_opp #2}{\romannumeral0\XINT_opp #1}%
}%
\def\XINT_outfrac_N_a #1#2%
{%
    \expandafter\XINT_outfrac_P\expandafter {#2}{#1}%
}%
```

8.6 \XINT_inFrac

Parses fraction, scientific notation, etc... and produces {n}{A}{B} corresponding to A/B times 10^n. No reduction to smallest terms.

Extended in 1.07 to accept scientific notation on input. With lowercase e only. The \xintexpr parser does accept uppercase E also. Ah, by the way, perhaps I should at least say what this macro does? (belated addition 2014/10/22...), before I forget! It prepares the fraction in the internal format {exponent}{Numerator}{Denominator} where Denominator is at least 1.

2015/10/09: this venerable macro from the very early days (1.03, 2013/04/14) has gotten a lifting for release 1.2. There were two kinds of issues:

1) use of \W, \Z, \T delimiters was very poor choice as this could clash with user input,

2) the new \XINT_frac_gen handles macros (possibly empty) in the input as general as \A.\Be\C/\D.\Ee\F. The earlier version would not have expanded the \B or \E: digits after decimal mark were constrained to arise from expansion of the first token. Thus the 1.03 original code would have expanded only \A, \D, \C, and \F for this input.

This reminded me think I should revisit the remaining earlier portions of code, as I was still learning TeX coding when I wrote them.

Also I thought about parsing even faster the A/B[N] input, not expanding B, but this turned out to clash with some established uses in the documentation such as 1/\xintiiSqr{...}[0]. For the implementation, careful here about potential brace removals with parameter patterns such as like #1/#2#3[#4]for example.

While I was at it 1.2 added \numexpr parsing of the N, which earlier was restricted to be only explicit digits. I allowed [] with empty N, but the way I did it in 1.2 with \the\numexpr 0#1 was

buggy, as it did not allow #1 to be a \count for example or itself a \numexpr (although such inputs were not previously allowed, I later turned out to use them in the code itself, e.g. the float factorial of version 1.2f). The better way would be \the\numexpr#1+\xint_c_ but 1.2f finally does only \the\numexpr #1 and #1 is not allowed to be empty.

The 1.2 \XINT_frac_gen had two locations with such a problematic \numexpr 0#1 which I replaced for 1.2f with \numexpr#1+\xint_c_.

Regarding calling the macro with an argument A[<expression>], a / inthe expression must be suitably hidden for example in \firstofone type constructs.

Note: when the numerator is found to be zero \XINT_infrac *always* returns {0}{0}{1}. This behaviour must not change because 1.2g \xintFloat and XINTinFloat (for example) rely upon it: if the denominator on output is not 1, then \xintFloat assumes that the numerator is not zero.

As described in the manual, if the input contains a (final) [N] part, it is assumed that it is in the shape A[N] or A/B[N] with A (and B) not containing neither decimal mark nor scientific part, moreover B must be positive and A have at most one minus sign (and no plus sign). Else there will be errors, for example -0/2[0] would not be recognized as being zero at this stage and this could cause issues afterwards. When there is no ending [N] part, both numerator and denominator will be parsed for the more general format allowing decimal digits and scientific part and possibly multiple leading signs.

1.2l fixes frailty of \XINT_infrac (hence basically of all xintfrac macros) respective to non terminated \numexpr input: \xintRaw{\the\numexpr1} for example. The issue was that \numexpr sees the / and expands what's next. But even \numexpr 1// for example creates an error, and to my mind this is a defect of \numexpr. It should be able to trace back and see that / was used as delimiter not as operator. Anyway, I thus fixed this problem belatedly here regarding \XINT_infrac.

```
\def\XINT_inFrac {\romannumeral0\XINT_infrac }%
\def\XINT_infrac #1%
{%
    \expandafter\XINT_infrac_fork\romannumeral`&&@#1\xint:/\XINT_W[\XINT_W\XINT_T
}%
\def\XINT_infrac_fork #1[#2%
{%
    \xint_UDXINTWfork
      #2\XINT_frac_gen          % input has no brackets [N]
      \XINT_W\XINT_infrac_res_a % there is some [N], must be strict A[N] or A/B[N] input
    \krof
    #1[#2%
}%
\def\XINT_infrac_res_a #1%
{%
    \xint_gob_til_zero #1\XINT_infrac_res_zero 0\XINT_infrac_res_b #1%
}%
```

Note that input exponent is here ignored and forced to be zero.

```
\def\XINT_infrac_res_zero 0\XINT_infrac_res_b #1\XINT_T {{0}{0}{1}}%
\def\XINT_infrac_res_b #1/#2%
{%
    \xint_UDXINTWfork
     #2\XINT_infrac_res_ca      % it was A[N] input
     \XINT_W\XINT_infrac_res_cb % it was A/B[N] input
    \krof
    #1/#2%
}%
```

An empty [] is not allowed. (this was authorized in 1.2, removed in 1.2f). As nobody reads xint documentation, no one will have noticed the fleeting possibility.

```
\def\XINT_infrac_res_ca #1[#2]\xint:/\XINT_W[\XINT_W\XINT_T
   {\expandafter{\the\numexpr #2}{#1}{1}}%
\def\XINT_infrac_res_cb #1/#2[%
   {\expandafter\XINT_infrac_res_cc\romannumeral`&&@#2~#1[}%
\def\XINT_infrac_res_cc #1~#2[#3]\xint:/\XINT_W[\XINT_W\XINT_T
   {\expandafter{\the\numexpr #3}{#2}{#1}}%
```

8.7 \XINT_frac_gen

Extended in 1.07 to recognize and accept scientific notation both at the numerator and (possible) denominator. Only a lowercase e will do here, but uppercase E is possible within an \xintexpr..\relax

Completely rewritten for 1.2 2015/10/10. The parsing handles inputs such as \A.\Be\C/\D.\Ee\F where each of \A, \B, \D, and \E may need f-expansion and \C and \F will end up in \numexpr.

1.2f corrects an issue to allow \C and \F to be \count variable (or expressions with \numexpr): 1.2 did a bad \numexpr0#1 which allowed only explicit digits for expanded #1.

```
\def\XINT_frac_gen #1/#2%
{%
    \xint_UDXINTWfork
      #2\XINT_frac_gen_A      % there was no /
      \XINT_W\XINT_frac_gen_B % there was a /
    \krof
    #1/#2%
}%
```

Note that #1 is only expanded so far up to decimal mark or "e".

```
\def\XINT_frac_gen_A #1\xint:/\XINT_W [\XINT_W {\XINT_frac_gen_C 0~1!#1ee.\XINT_W }%
\def\XINT_frac_gen_B #1/#2\xint:/\XINT_W[%\XINT_W
{%
    \expandafter\XINT_frac_gen_Ba
    \romannumeral`&&@#2ee.\XINT_W\XINT_Z #1ee.%\XINT_W
}%
\def\XINT_frac_gen_Ba #1.#2%
{%
    \xint_UDXINTWfork
      #2\XINT_frac_gen_Bb
      \XINT_W\XINT_frac_gen_Bc
    \krof
    #1.#2%
}%
\def\XINT_frac_gen_Bb #1e#2e#3\XINT_Z
                {\expandafter\XINT_frac_gen_C\the\numexpr #2+\xint_c_~#1!}%
\def\XINT_frac_gen_Bc #1.#2e%
{%
    \expandafter\XINT_frac_gen_Bd\romannumeral`&&@#2.#1e%
}%
\def\XINT_frac_gen_Bd #1.#2e#3e#4\XINT_Z
{%
```

```
    \expandafter\XINT_frac_gen_C\the\numexpr #3-%
    \numexpr\XINT_length_loop
    #1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    ~#2#1!%
}%
\def\XINT_frac_gen_C #1!#2.#3%
{%
    \xint_UDXINTWfork
      #3\XINT_frac_gen_Ca
      \XINT_W\XINT_frac_gen_Cb
    \krof
    #1!#2.#3%
}%
\def\XINT_frac_gen_Ca #1~#2!#3e#4e#5\XINT_T
{%
    \expandafter\XINT_frac_gen_F\the\numexpr #4-#1\expandafter
    ~\romannumeral0\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
     #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z~#3~%
}%
\def\XINT_frac_gen_Cb #1.#2e%
{%
    \expandafter\XINT_frac_gen_Cc\romannumeral`&&@#2.#1e%
}%
\def\XINT_frac_gen_Cc #1.#2~#3!#4e#5e#6\XINT_T
{%
    \expandafter\XINT_frac_gen_F\the\numexpr #5-#2-%

    \numexpr\XINT_length_loop
    #1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye

    \relax\expandafter~%
    \romannumeral0\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
    #3\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
    ~#4#1~%
}%
\def\XINT_frac_gen_F #1~#2%
{%
    \xint_UDzerominusfork
      #2-\XINT_frac_gen_Gdivbyzero
      0#2{\XINT_frac_gen_G  -{}}%
       0-{\XINT_frac_gen_G {}#2}%
    \krof  #1~%
}%
\def\XINT_frac_gen_Gdivbyzero #1~~#2~%
{%
   \expandafter\XINT_frac_gen_Gdivbyzero_a
   \romannumeral0\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
   #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z~#1~%
```

```
}%
\def\XINT_frac_gen_Gdivbyzero_a #1~#2~%
{%
    \XINT_signalcondition{DivisionByZero}{Division of #1 by zero}{}{{#2}{#1}{0}}%
}%
\def\XINT_frac_gen_G #1#2#3~#4~#5~%
{%
    \expandafter\XINT_frac_gen_Ga
    \romannumeral0\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
    #1#5\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z~#3~{#2#4}%
}%
\def\XINT_frac_gen_Ga #1#2~#3~%
{%
    \xint_gob_til_zero #1\XINT_frac_gen_zero 0%
    {#3}{#1#2}%
}%
\def\XINT_frac_gen_zero 0#1#2#3{{0}{0}{1}}%
```

8.8 \XINT_factortens

This is the core macro for \xintREZ. To be used as \romannumeral0\XINT_factortens{...}. Output is A.N. (formerly {A}{N}) where A is the integer stripped from trailing zeroes and N is the number of removed zeroes. Only for positive strict integers!

Completely rewritten at 1.3a to replace a double \xintReverseOrder by a direct \numexpr governed expansion to the end and back, à la 1.2. I should comment more... and perhaps improve again in future.

Testing shows significant gain at 100 digits or more.

```
\def\XINT_factortens #1{\expandafter\XINT_factortens_z
                        \romannumeral0\XINT_factortens_a#1%
                        \XINT_factortens_b123456789.}%
\def\XINT_factortens_z.\XINT_factortens_y{ }%
\def\XINT_factortens_a #1#2#3#4#5#6#7#8#9%
   {\expandafter\XINT_factortens_x
    \the\numexpr 1#1#2#3#4#5#6#7#8#9\XINT_factortens_a}%
\def\XINT_factortens_b#1\XINT_factortens_a#2#3.%
   {.\XINT_factortens_cc 000000000-#2.}%
\def\XINT_factortens_x1#1.#2{#2#1}%
\def\XINT_factortens_y{.\XINT_factortens_y}%
\def\XINT_factortens_cc #1#2#3#4#5#6#7#8#9%
   {\if#90\xint_dothis
     {\expandafter\XINT_factortens_d\the\numexpr #8#7#6#5#4#3#2#1\relax
      \xint_c_i 2345678.}\fi
    \xint_orthat{\XINT_factortens_yy{#1#2#3#4#5#6#7#8#9}}}%
\def\XINT_factortens_yy #1#2.{.\XINT_factortens_y#1.0.}%
\def\XINT_factortens_c #1#2#3#4#5#6#7#8#9%
   {\if#90\xint_dothis
     {\expandafter\XINT_factortens_d\the\numexpr #8#7#6#5#4#3#2#1\relax
      \xint_c_i 2345678.}\fi
    \xint_orthat{.\XINT_factortens_y #1#2#3#4#5#6#7#8#9.}}%
\def\XINT_factortens_d #1#2#3#4#5#6#7#8#9%
   {\if#10\expandafter\XINT_factortens_e\fi
    \XINT_factortens_f #9#9#8#7#6#5#4#3#2#1.}%
```

```
\def\XINT_factortens_f #1#2\xint_c_i#3.#4.#5.%
   {\expandafter\XINT_factortens_g\the\numexpr#1+#5.#3.}%
\def\XINT_factortens_g #1.#2.{.\XINT_factortens_y#2.#1.}%
\def\XINT_factortens_e #1..#2.%
   {\expandafter.\expandafter\XINT_factortens_c
    \the\numexpr\xint_c_ix+#2.}%
```

8.9 \xintEq, \xintNotEq, \xintGt, \xintLt, \xintGtorEq, \xintLtorEq, \xintIsZero, \xintIsNotZero, \xintOdd, \xintEven, \xintifSgn, \xintifCmp, \xintifEq, \xintifGt, \xintifLt, \xintifZero, \xintifNotZero, \xintifOne, \xintifOdd

Moved here at 1.3. Formerly these macros were already defined in xint.sty or even xintcore.sty. They are slim wrappers of macros defined elsewhere in xintfrac.

```
\def\xintEq   {\romannumeral0\xinteq }%
\def\xinteq   #1#2{\xintifeq{#1}{#2}{1}{0}}%
\def\xintNotEq#1#2{\romannumeral0\xintifeq {#1}{#2}{0}{1}}%
\def\xintGt {\romannumeral0\xintgt }%
\def\xintgt #1#2{\xintifgt{#1}{#2}{1}{0}}%
\def\xintLt   {\romannumeral0\xintlt }%
\def\xintlt #1#2{\xintiflt{#1}{#2}{1}{0}}%
\def\xintGtorEq #1#2{\romannumeral0\xintiflt {#1}{#2}{0}{1}}%
\def\xintLtorEq #1#2{\romannumeral0\xintifgt {#1}{#2}{0}{1}}%
\def\xintIsZero   {\romannumeral0\xintiszero }%
\def\xintiszero #1{\if0\xintSgn{#1}\xint_afterfi{ 1}\else\xint_afterfi{ 0}\fi}%
\def\xintIsNotZero{\romannumeral0\xintisnotzero }%
\def\xintisnotzero
          #1{\if0\xintSgn{#1}\xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi}%
\def\xintOdd      {\romannumeral0\xintodd }%
\def\xintodd #1%
{%
    \ifodd\xintLDg{\xintNum{#1}} %<- intentional space
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ 0}%
    \fi
}%
\def\xintEven     {\romannumeral0\xinteven }%
\def\xinteven #1%
{%
    \ifodd\xintLDg{\xintNum{#1}} %<- intentional space
        \xint_afterfi{ 0}%
    \else
        \xint_afterfi{ 1}%
    \fi
}%
\def\xintifSgn{\romannumeral0\xintifsgn }%
\def\xintifsgn #1%
{%
    \ifcase \xintSgn{#1}
               \expandafter\xint_stop_atsecondofthree
            \or\expandafter\xint_stop_atthirdofthree
```

```
295           \else\expandafter\xint_stop_atfirstofthree
296     \fi
297 }%
298 \def\xintifCmp{\romannumeral0\xintifcmp }%
299 \def\xintifcmp #1#2%
300 {%
301     \ifcase\xintCmp {#1}{#2}
302                 \expandafter\xint_stop_atsecondofthree
303             \or\expandafter\xint_stop_atthirdofthree
304           \else\expandafter\xint_stop_atfirstofthree
305     \fi
306 }%
307 \def\xintifEq {\romannumeral0\xintifeq }%
308 \def\xintifeq #1#2%
309 {%
310     \if0\xintCmp{#1}{#2}%
311                 \expandafter\xint_stop_atfirstoftwo
312           \else\expandafter\xint_stop_atsecondoftwo
313     \fi
314 }%
315 \def\xintifGt {\romannumeral0\xintifgt }%
316 \def\xintifgt #1#2%
317 {%
318     \if1\xintCmp{#1}{#2}%
319                 \expandafter\xint_stop_atfirstoftwo
320           \else\expandafter\xint_stop_atsecondoftwo
321     \fi
322 }%
323 \def\xintifLt {\romannumeral0\xintiflt }%
324 \def\xintiflt #1#2%
325 {%
326     \ifnum\xintCmp{#1}{#2}<\xint_c_
327           \expandafter\xint_stop_atfirstoftwo
328     \else \expandafter\xint_stop_atsecondoftwo
329     \fi
330 }%
331 \def\xintifZero   {\romannumeral0\xintifzero }%
332 \def\xintifzero #1%
333 {%
334     \if0\xintSgn{#1}%
335        \expandafter\xint_stop_atfirstoftwo
336     \else
337        \expandafter\xint_stop_atsecondoftwo
338     \fi
339 }%
340 \def\xintifNotZero{\romannumeral0\xintifnotzero }%
341 \def\xintifnotzero #1%
342 {%
343     \if0\xintSgn{#1}%
344        \expandafter\xint_stop_atsecondoftwo
345     \else
346        \expandafter\xint_stop_atfirstoftwo
```

```
    \fi
}%
\def\xintifOne {\romannumeral0\xintifone }%
\def\xintifone #1%
{%
    \if1\xintIsOne{#1}%
       \expandafter\xint_stop_atfirstoftwo
    \else
       \expandafter\xint_stop_atsecondoftwo
    \fi
}%
\def\xintifOdd {\romannumeral0\xintifodd }%
\def\xintifodd #1%
{%
    \if\xintOdd{#1}1%
       \expandafter\xint_stop_atfirstoftwo
    \else
       \expandafter\xint_stop_atsecondoftwo
    \fi
}%
```

8.10 \xintRaw

1.07: this macro simply prints in a user readable form the fraction after its initial scanning. Useful when put inside braces in an \xintexpr, when the input is not yet in the A/B[n] form.

```
\def\xintRaw {\romannumeral0\xintraw }%
\def\xintraw
{%
    \expandafter\XINT_raw\romannumeral0\XINT_infrac
}%
\def\XINT_raw #1#2#3{ #2/#3[#1]}%
```

8.11 \xintiLogTen

New at 1.3e

```
\def\xintiLogTen {\the\numexpr\xintilogten}%
\def\xintilogten
{%
    \expandafter\XINT_ilogten\romannumeral0\xintraw
}%
\def\XINT_ilogten #1%
{%
    \xint_UDzerominusfork
      0#1\XINT_ilogten_p
      #1-\XINT_ilogten_z
      0-{\XINT_ilogten_p#1}%
    \krof
}%
\def\XINT_ilogten_z #1[#2]{-"7FFF8000\relax}%
\def\XINT_ilogten_p #1/#2[#3]%
{%
```

```
    #3+\expandafter\XINT_ilogten_a
       \the\numexpr\xintLength{#1}\expandafter.\the\numexpr\xintLength{#2}.#1.#2.%
}%
\def\XINT_ilogten_a #1.#2.%
{%
    #1-#2\ifnum#1>#2
      \expandafter\XINT_ilogten_aa
    \else
      \expandafter\XINT_ilogten_ab
    \fi #1.#2.%
}%
\def\XINT_ilogten_aa #1.#2.#3.#4.%
{%
    \xintiiifLt{#3}{\XINT_dsx_addzerosnofuss{#1-#2}#4;}{-1}{}\relax
}%
\def\XINT_ilogten_ab #1.#2.#3.#4.%
{%
    \xintiiifLt{\XINT_dsx_addzerosnofuss{#2-#1}#3;}{#4}{-1}{}\relax
}%
```

8.12 \xintPRaw

1.09b

```
\def\xintPRaw {\romannumeral0\xintpraw }%
\def\xintpraw
{%
    \expandafter\XINT_praw\romannumeral0\XINT_infrac
}%
\def\XINT_praw #1%
{%
    \ifnum #1=\xint_c_ \expandafter\XINT_praw_a\fi \XINT_praw_A {#1}%
}%
\def\XINT_praw_A #1#2#3%
{%
    \if\XINT_isOne{#3}1\expandafter\xint_firstoftwo
                  \else\expandafter\xint_secondoftwo
    \fi { #2[#1]}{ #2/#3[#1]}%
}%
\def\XINT_praw_a\XINT_praw_A #1#2#3%
{%
    \if\XINT_isOne{#3}1\expandafter\xint_firstoftwo
                  \else\expandafter\xint_secondoftwo
    \fi { #2}{ #2/#3}%
}%
```

8.13 \xintSPRaw

This private macro was for usage by \xinttheexpr. It got moved here at 1.4 and is not used anymore by the package.

Attention that \xintSPRaw assumed that if the number has no [N] part it did not have a fraction part /B either. Indeed this was the case always with 1.3f: parsing of an integer by \xintexpr does not add the [0] as the code is shared with \xintiiexpr, and when there was /B \xintexpr always

added postfix [0]; even qfrac() parses via \xintRaw; and reduce() internally used \xintIrr which outputs A/B but it added [0].

```
\def\xintSPRaw    {\romannumeral0\xintspraw }%
\def\xintspraw  #1{\expandafter\XINT_spraw\romannumeral`&&@#1[\W]}%
\def\XINT_spraw #1[#2#3]{\xint_gob_til_W #2\XINT_spraw_a\W\XINT_spraw_p #1[#2#3]}%
\def\XINT_spraw_a\W\XINT_spraw_p #1[\W]{ #1}%
\def\XINT_spraw_p #1[\W]{\xintpraw {#1}}%
```

8.14 \xintFracToSci, \xintFracToSciE

1.4

This is the new macro used in place of \xintSPRaw (which basically was \xintPRaw) by \xintexpr typesetter. Attention that it is also used by \xintiexpr with inputs having being already converted to decimal form, hence must understand this input form. This means it must for example not think 0.123 is 0 because it starts with 0.

And indeed the code here lets 0.123 go through as is. Identification of 0 as first digit is done only in case of A[N], A/B, and A/B[N] formats.

As reduce() does not anymore append the [0] at 1.4, \xintFracToSci has indeed to recognize A, A[N], A/B and A/B[N] but does not have to parse multiple plus or minus signs or scientific part etc like \xintRaw does (delegating to \XINT_infrac like all other xintfrac macros). It has to identify say 0/5 (although I don't think that can arise) and -0 is never occuring.

The difference with \xintSPRaw is that it outputs AeN/B. It will not print the /B if B=1 and eN if N is zero.

If input is empty \xintFracToSci output is also empty, whereas \xintRaw produces 0/1[0] out of empty. But \XINTexprprint anyhow has it own special routine for empty input.

1.4b extends the macro to intercept scientific notation and thus allow customizability of the «e» via \xintFracToSciE. Without this \xintieval with a negative optional argument uses «e» in output with no possibility to modify it.

The expansion context from \xinttheexpr, \xinttheiexpr, \xinteval, \xintieval is the scope of one \expanded.

Attention indeed that this macro is not f-expandable only x-expandable.

```
\edef\xintFracToSci  #1%
   {\unexpanded{\expandafter\XINT_FracToSci\romannumeral`&&@}#1\string e%
    \unexpanded{\Z/\W[\R]}}%
\def\XINT_FracToSci #1/#2#3[#4%
{%
    \xint_gob_til_W #2\XINT_FracToSci_no\W
    \xint_gob_til_R #4\XINT_FracToSci_yesno\R
    \XINT_FracToSci_yesyes #1/#2#3[#4%
}%
\def\XINT_FracToSci_no #1\XINT_FracToSci_yesyes #2[#3%
{%
    \xint_gob_til_R #3\XINT_FracToSci_nono\R
    \XINT_FracToSci_noyes #2[#3%
}%
\edef\XINT_tmpa{##1\string e##2}%
\def\XINT_tmpb{\def\XINT_FracToSci_nono\R\XINT_FracToSci_noyes}%
\expandafter
\XINT_tmpb\XINT_tmpa
{%
    #1\xint_gob_til_Z #2\XINT_FracToSci_nonono\Z
```

```
    \XINT_FracToSci_nonoyes #2%
}%
\edef\XINT_tmpa{##1\string e}%
 \def\XINT_tmpb{\def\XINT_FracToSci_nonoyes}%
\expandafter
\XINT_tmpb\XINT_tmpa\Z/\W[\R]{\xintFracToSciE#1}%
\def\XINT_FracToSci_nonono\Z\XINT_FracToSci_nonoyes\Z/\W[\R]{}%
\edef\XINT_tmpa{##1##2[##3]\string e}%
 \def\XINT_tmpb{\def\XINT_FracToSci_noyes}%
\expandafter
\XINT_tmpb\XINT_tmpa\Z/\W[\R]%
{%
    #1\xint_gob_til_zero#1\expandafter\iffalse\xint_gobble_ii0\iftrue
    #2\ifnum #3=\xint_c_\else\xintFracToSciE#3\fi\fi
}%
\edef\XINT_tmpa{##1##2/##3\string e}%
 \def\XINT_tmpb{\def\XINT_FracToSci_yesno\R\XINT_FracToSci_yesyes}%
\expandafter
\XINT_tmpb\XINT_tmpa\Z/\W[\R]%
{%
    #1\xint_gob_til_zero#1\expandafter\iffalse\xint_gobble_ii0\iftrue
    #2\if\XINT_isOne{#3}1\else/#3\fi\fi
}%
\edef\XINT_tmpa{##1##2/##3[##4]\string e}%
 \def\XINT_tmpb{\def\XINT_FracToSci_yesyes}%
\expandafter
\XINT_tmpb\XINT_tmpa\Z/\W[\R]%
{%
    #1\xint_gob_til_zero#1\expandafter\iffalse\xint_gobble_ii0\iftrue
    #2\ifnum #4=\xint_c_\else\xintFracToSciE#4\fi
    \if\XINT_isOne{#3}1\else/#3\fi\fi
}%
\def\xintFracToSciE{e}%
```

8.15 \xintRawWithZeros

This was called \xintRaw in versions earlier than 1.07

```
\def\xintRawWithZeros {\romannumeral0\xintrawwithzeros }%
\def\xintrawwithzeros
{%
    \expandafter\XINT_rawz_fork\romannumeral0\XINT_infrac
}%


\def\XINT_rawz_fork #1%
{%
    \ifnum#1<\xint_c_
      \expandafter\XINT_rawz_Ba
    \else
      \expandafter\XINT_rawz_A
    \fi
    #1.%
```

```
}%
\def\XINT_rawz_A  #1.#2#3{\XINT_dsx_addzeros{#1}#2;/#3}%
\def\XINT_rawz_Ba -#1.#2#3{\expandafter\XINT_rawz_Bb
    \expandafter{\romannumeral0\XINT_dsx_addzeros{#1}#3;}{#2}}%
\def\XINT_rawz_Bb #1#2{ #2/#1}%
```

8.16 \xintDecToString

1.3. This is a backport from polexpr 0.4. It is definitely not in final form, consider it to be an unstable macro.

```
\def\xintDecToString{\romannumeral0\xintdectostring}%
\def\xintdectostring#1{\expandafter\XINT_dectostr\romannumeral0\xintraw{#1}}%
\def\XINT_dectostr #1/#2[#3]{\xintiiifZero {#1}%
       \XINT_dectostr_z
       {\if1\XINT_isOne{#2}\expandafter\XINT_dectostr_a
                       \else\expandafter\XINT_dectostr_b
        \fi}%
  #1/#2[#3]%
}%
\def\XINT_dectostr_z#1[#2]{ 0}%
\def\XINT_dectostr_a#1/#2[#3]{%
    \ifnum#3<\xint_c_\xint_dothis{\xinttrunc{-#3}{#1[#3]}}\fi
    \xint_orthat{\xintiie{#1}{#3}}%
}%
\def\XINT_dectostr_b#1/#2[#3]{% just to handle this somehow
    \ifnum#3<\xint_c_\xint_dothis{\xinttrunc{-#3}{#1[#3]}/#2}\fi
    \xint_orthat{\xintiie{#1}{#3}/#2}%
}%
```

8.17 \xintFloor, \xintiFloor

1.09a, 1.1 for \xintiFloor/\xintFloor. Not efficient if big negative decimal exponent. Also sub-efficient if big positive decimal exponent.

```
\def\xintFloor {\romannumeral0\xintfloor }%
\def\xintfloor #1% devrais-je faire \xintREZ?
    {\expandafter\XINT_ifloor \romannumeral0\xintrawwithzeros {#1}./1[0]}%
\def\xintiFloor {\romannumeral0\xintifloor }%
\def\xintifloor #1%
    {\expandafter\XINT_ifloor \romannumeral0\xintrawwithzeros {#1}.}%
\def\XINT_ifloor #1/#2.{\xintiiquo {#1}{#2}}%
```

8.18 \xintCeil, \xintiCeil

1.09a

```
\def\xintCeil {\romannumeral0\xintceil }%
\def\xintceil #1{\xintiiopp {\xintFloor {\xintOpp{#1}}}}%
\def\xintiCeil {\romannumeral0\xinticeil }%
\def\xinticeil #1{\xintiiopp {\xintiFloor {\xintOpp{#1}}}}%
```

8.19 \xintNumerator

```
\def\xintNumerator {\romannumeral0\xintnumerator }%
\def\xintnumerator
{%
    \expandafter\XINT_numer\romannumeral0\XINT_infrac
}%
\def\XINT_numer #1%
{%
    \ifcase\XINT_cntSgn #1\xint:
      \expandafter\XINT_numer_B
    \or
      \expandafter\XINT_numer_A
    \else
      \expandafter\XINT_numer_B
    \fi
    {#1}%
}%
\def\XINT_numer_A #1#2#3{\XINT_dsx_addzeros{#1}#2;}%
\def\XINT_numer_B #1#2#3{ #2}%
```

8.20 \xintDenominator

```
\def\xintDenominator {\romannumeral0\xintdenominator }%
\def\xintdenominator
{%
    \expandafter\XINT_denom_fork\romannumeral0\XINT_infrac
}%
\def\XINT_denom_fork #1%
{%

    \ifnum#1<\xint_c_
      \expandafter\XINT_denom_B
    \else
      \expandafter\XINT_denom_A
    \fi
    #1.%
}%
\def\XINT_denom_A #1.#2#3{ #3}%
\def\XINT_denom_B -#1.#2#3{\XINT_dsx_addzeros{#1}#3;}%
```

8.21 \xintFrac

Useless typesetting macro.

```
\def\xintFrac {\romannumeral0\xintfrac }%
\def\xintfrac #1%
{%
    \expandafter\XINT_fracfrac_A\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_fracfrac_A #1{\XINT_fracfrac_B #1\Z }%
\catcode`^=7
\def\XINT_fracfrac_B #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_fracfrac_C 0\XINT_fracfrac_D {10^{#1#2}}%
```

```
}%
\def\XINT_fracfrac_C 0\XINT_fracfrac_D #1#2#3%
{%
    \if1\XINT_isOne {#3}%
        \xint_afterfi {\expandafter\xint_stop_atfirstoftwo\xint_gobble_ii }%
    \fi
    \space
    \frac {#2}{#3}%
}%
\def\XINT_fracfrac_D #1#2#3%
{%
    \if1\XINT_isOne {#3}\XINT_fracfrac_E\fi
    \space
    \frac {#2}{#3}#1%
}%
\def\XINT_fracfrac_E \fi\space\frac #1#2{\fi \space #1\cdot }%
```

8.22 \xintSignedFrac

```
\def\xintSignedFrac {\romannumeral0\xintsignedfrac }%
\def\xintsignedfrac #1%
{%
    \expandafter\XINT_sgnfrac_a\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_sgnfrac_a #1#2%
{%
    \XINT_sgnfrac_b #2\Z {#1}%
}%
\def\XINT_sgnfrac_b #1%
{%
    \xint_UDsignfork
      #1\XINT_sgnfrac_N
       -{\XINT_sgnfrac_P #1}%
    \krof
}%
\def\XINT_sgnfrac_P #1\Z #2%
{%
    \XINT_fracfrac_A {#2}{#1}%
}%
\def\XINT_sgnfrac_N
{%
    \expandafter-\romannumeral0\XINT_sgnfrac_P
}%
```

8.23 \xintFwOver

```
\def\xintFwOver {\romannumeral0\xintfwover }%
\def\xintfwover #1%
{%
    \expandafter\XINT_fwover_A\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_fwover_A #1{\XINT_fwover_B #1\Z }%
\def\XINT_fwover_B #1#2\Z
{%
```

```
    \xint_gob_til_zero #1\XINT_fwover_C 0\XINT_fwover_D {10^{#1#2}}%
}%
\catcode`^=11
\def\XINT_fwover_C #1#2#3#4#5%
{%
    \if0\XINT_isOne {#5}\xint_afterfi { {#4\over #5}}%
                    \else\xint_afterfi { #4}%
    \fi
}%
\def\XINT_fwover_D #1#2#3%
{%
    \if0\XINT_isOne {#3}\xint_afterfi { {#2\over #3}}%
                    \else\xint_afterfi { #2\cdot }%
    \fi
    #1%
}%
```

8.24 \xintSignedFwOver

```
\def\xintSignedFwOver {\romannumeral0\xintsignedfwover }%
\def\xintsignedfwover #1%
{%
    \expandafter\XINT_sgnfwover_a\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_sgnfwover_a #1#2%
{%
    \XINT_sgnfwover_b #2\Z {#1}%
}%
\def\XINT_sgnfwover_b #1%
{%
    \xint_UDsignfork
      #1\XINT_sgnfwover_N
       -{\XINT_sgnfwover_P #1}%
    \krof
}%
\def\XINT_sgnfwover_P #1\Z #2%
{%
    \XINT_fwover_A {#2}{#1}%
}%
\def\XINT_sgnfwover_N
{%
    \expandafter-\romannumeral0\XINT_sgnfwover_P
}%
```

8.25 \xintREZ

Removes trailing zeros from A and B and adjust the N in A/B[N].
  The macro really doing the job \XINT_factortens was redone at 1.3a. But speed gain really noticeable only beyond about 100 digits.

```
\def\xintREZ {\romannumeral0\xintrez }%
\def\xintrez
{%
    \expandafter\XINT_rez_A\romannumeral0\XINT_infrac
```

```
}%
\def\XINT_rez_A #1#2%
{%
    \XINT_rez_AB #2\Z {#1}%
}%
\def\XINT_rez_AB #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_rez_zero
      0#1\XINT_rez_neg
       0-{\XINT_rez_B #1}%
    \krof
}%
\def\XINT_rez_zero #1\Z #2#3{ 0/1[0]}%
\def\XINT_rez_neg {\expandafter-\romannumeral0\XINT_rez_B }%
\def\XINT_rez_B #1\Z
{%
    \expandafter\XINT_rez_C\romannumeral0\XINT_factortens {#1}%
}%
\def\XINT_rez_C #1.#2.#3#4%
{%
    \expandafter\XINT_rez_D\romannumeral0\XINT_factortens {#4}#3+#2.#1.%
}%
\def\XINT_rez_D #1.#2.#3.%
{%
    \expandafter\XINT_rez_E\the\numexpr #3-#2.#1.%
}%
\def\XINT_rez_E #1.#2.#3.{ #3/#2[#1]}%
```

8.26 \xintE

1.07: The fraction is the first argument contrarily to \xintTrunc and \xintRound.
1.1 modifies and moves \xintiiE to xint.sty.

```
\def\xintE {\romannumeral0\xinte }%
\def\xinte #1%
{%
    \expandafter\XINT_e \romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_e #1#2#3#4%
{%
    \expandafter\XINT_e_end\the\numexpr #1+#4.{#2}{#3}%
}%
\def\XINT_e_end #1.#2#3{ #2/#3[#1]}%
```

8.27 \xintIrr, \xintPIrr

\xintPIrr (partial Irr, which ignores the decimal part) added at 1.3.

```
\def\xintIrr {\romannumeral0\xintirr }%
\def\xintPIrr{\romannumeral0\xintpirr }%
\def\xintirr #1%
{%
```

```
    \expandafter\XINT_irr_start\romannumeral0\xintrawwithzeros {#1}\Z
}%
\def\xintpirr #1%
{%
    \expandafter\XINT_pirr_start\romannumeral0\xintraw{#1}%
}%
\def\XINT_irr_start #1#2/#3\Z
{%
    \if0\XINT_isOne {#3}%
      \xint_afterfi
          {\xint_UDsignfork
               #1\XINT_irr_negative
                -{\XINT_irr_nonneg #1}%
           \krof}%
    \else
      \xint_afterfi{\XINT_irr_denomisone #1}%
    \fi
    #2\Z {#3}%
}%
\def\XINT_pirr_start #1#2/#3[%
{%
    \if0\XINT_isOne {#3}%
      \xint_afterfi
          {\xint_UDsignfork
               #1\XINT_irr_negative
                -{\XINT_irr_nonneg #1}%
           \krof}%
    \else
      \xint_afterfi{\XINT_irr_denomisone #1}%
    \fi
    #2\Z {#3}[%
}%
\def\XINT_irr_denomisone #1\Z #2{ #1/1}% changed in 1.08
\def\XINT_irr_negative   #1\Z #2{\XINT_irr_D #1\Z #2\Z -}%
\def\XINT_irr_nonneg     #1\Z #2{\XINT_irr_D #1\Z #2\Z \space}%
\def\XINT_irr_D #1#2\Z #3#4\Z
{%
    \xint_UDzerosfork
       #3#1\XINT_irr_indeterminate
       #30\XINT_irr_divisionbyzero
       #10\XINT_irr_zero
        00\XINT_irr_loop_a
    \krof
    {#3#4}{#1#2}{#3#4}{#1#2}%
}%
\def\XINT_irr_indeterminate #1#2#3#4#5%
{%
    \XINT_signalcondition{DivisionUndefined}{indeterminate: 0/0}{}{0/1}%
}%
\def\XINT_irr_divisionbyzero #1#2#3#4#5%
{%
    \XINT_signalcondition{DivisionByZero}{vanishing denominator: #5#2/0}{}{0/1}%
```

```
}%
\def\XINT_irr_zero #1#2#3#4#5{ 0/1}% changed in 1.08
\def\XINT_irr_loop_a #1#2%
{%
    \expandafter\XINT_irr_loop_d
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
\def\XINT_irr_loop_d #1#2%
{%
    \XINT_irr_loop_e #2\Z
}%
\def\XINT_irr_loop_e #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_irr_loop_exit0\XINT_irr_loop_a {#1#2}%
}%
\def\XINT_irr_loop_exit0\XINT_irr_loop_a #1#2#3#4%
{%
    \expandafter\XINT_irr_loop_exitb\expandafter
    {\romannumeral0\xintiiquo {#3}{#2}}%
    {\romannumeral0\xintiiquo {#4}{#2}}%
}%
\def\XINT_irr_loop_exitb #1#2%
{%
   \expandafter\XINT_irr_finish\expandafter {#2}{#1}%
}%
\def\XINT_irr_finish #1#2#3{#3#1/#2}% changed in 1.08
```

8.28 \xintifInt

```
\def\xintifInt   {\romannumeral0\xintifint }%
\def\xintifint #1{\expandafter\XINT_ifint\romannumeral0\xintrawwithzeros {#1}.}%
\def\XINT_ifint #1/#2.%
{%
    \if 0\xintiiRem {#1}{#2}%
     \expandafter\xint_stop_atfirstoftwo
    \else
     \expandafter\xint_stop_atsecondoftwo
    \fi
}%
```

8.29 \xintIsInt

Added at 1.3d only, for isint() xintexpr function.

```
\def\xintIsInt   {\romannumeral0\xintisint }%
\def\xintisint #1%
   {\expandafter\XINT_ifint\romannumeral0\xintrawwithzeros {#1}.10}%
```

8.30 \xintJrr

```
\def\xintJrr {\romannumeral0\xintjrr }%
\def\xintjrr #1%
{%
    \expandafter\XINT_jrr_start\romannumeral0\xintrawwithzeros {#1}\Z
```

```
}%
\def\XINT_jrr_start #1#2/#3\Z
{%
    \if0\XINT_isOne {#3}\xint_afterfi
          {\xint_UDsignfork
               #1\XINT_jrr_negative
                -{\XINT_jrr_nonneg #1}%
           \krof}%
    \else
      \xint_afterfi{\XINT_jrr_denomisone #1}%
    \fi
    #2\Z {#3}%
}%
\def\XINT_jrr_denomisone #1\Z #2{ #1/1}% changed in 1.08
\def\XINT_jrr_negative   #1\Z #2{\XINT_jrr_D #1\Z #2\Z -}%
\def\XINT_jrr_nonneg     #1\Z #2{\XINT_jrr_D #1\Z #2\Z \space}%
\def\XINT_jrr_D #1#2\Z #3#4\Z
{%
    \xint_UDzerosfork
       #3#1\XINT_jrr_indeterminate
       #30\XINT_jrr_divisionbyzero
       #10\XINT_jrr_zero
        00\XINT_jrr_loop_a
    \krof
    {#3#4}{#1#2}1001%
}%
\def\XINT_jrr_indeterminate #1#2#3#4#5#6#7%
{%
    \XINT_signalcondition{DivisionUndefined}{indeterminate: 0/0}{}{0/1}%
}%
\def\XINT_jrr_divisionbyzero #1#2#3#4#5#6#7%
{%
    \XINT_signalcondition{DivisionByZero}{Vanishing denominator: #7#2/0}{}{0/1}%
}%
\def\XINT_jrr_zero #1#2#3#4#5#6#7{ 0/1}% changed in 1.08
\def\XINT_jrr_loop_a #1#2%
{%
    \expandafter\XINT_jrr_loop_b
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
\def\XINT_jrr_loop_b #1#2#3#4#5#6#7%
{%
    \expandafter \XINT_jrr_loop_c \expandafter
        {\romannumeral0\xintiiadd{\XINT_mul_fork #4\xint:#1\xint:}{#6}}%
        {\romannumeral0\xintiiadd{\XINT_mul_fork #5\xint:#1\xint:}{#7}}%
    {#2}{#3}{#4}{#5}%
}%
\def\XINT_jrr_loop_c #1#2%
{%
    \expandafter \XINT_jrr_loop_d \expandafter{#2}{#1}%
}%
\def\XINT_jrr_loop_d #1#2#3#4%
```

```
{%
    \XINT_jrr_loop_e #3\Z {#4}{#2}{#1}%
}%
\def\XINT_jrr_loop_e #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_jrr_loop_exit0\XINT_jrr_loop_a {#1#2}%
}%
\def\XINT_jrr_loop_exit0\XINT_jrr_loop_a #1#2#3#4#5#6%
{%
    \XINT_irr_finish {#3}{#4}%
}%
```

8.31 \xintTFrac

1.09i, for frac in \xintexpr. And \xintFrac is already assigned. T for truncation. However, potentially not very efficient with numbers in scientific notations, with big exponents. Will have to think it again some day. I hesitated how to call the macro. Same convention as in maple, but some people reserve fractional part to x - floor(x). Also, not clear if I had to make it negative (or zero) if x < 0, or rather always positive. There should be in fact such a thing for each rounding function, trunc, round, floor, ceil.

```
\def\xintTFrac {\romannumeral0\xinttfrac }%
\def\xinttfrac #1{\expandafter\XINT_tfrac_fork\romannumeral0\xintrawwithzeros {#1}\Z }%
\def\XINT_tfrac_fork #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_tfrac_zero
        0#1{\xintiiopp\XINT_tfrac_P }%
         0-{\XINT_tfrac_P #1}%
    \krof
}%
\def\XINT_tfrac_zero #1\Z { 0/1[0]}%
\def\XINT_tfrac_P #1/#2\Z {\expandafter\XINT_rez_AB
                           \romannumeral0\xintiirem{#1}{#2}\Z {0}{#2}}%
```

8.32 \xintTrunc, \xintiTrunc

This of course has a long history. Only showing here some comments.

1.2i release notes: ever since its inception this macro was stupid for a decimal input: it did not handle it separately from the general fraction case A/B[N] with B>1, hence ended up doing divisions by powers of ten. But this meant that nesting \xintTrunc with itself was very inefficient.

1.2i version is better. However it still handles B>1, N<0 via adding zeros to B and dividing with this extended B. A possibly more efficient approach is implemented in \xintXTrunc, but its logic is more complicated, the code is quite longer and making it f-expandable would not shorten it... I decided for the time being to not complicate things here.

1.4a (2020/02/18) adds handling of a negative first argument.

Zero input still gives single digit 0 output as I did not want to complicate the code. But if quantization gives 0, the exponent [D] will be there. Well actually eD because of problem that sign of original is preserved in output so we can have -0 and I can not use -0[D] notation as it is not legal for strict format. So I will use -0eD hence eD generally even though this means so slight suboptimality for trunc() function in \xintexpr.

The idea to give a meaning to negative D (in the context of optional argument to \xintiexpr) was suggested a long time ago by Kpym (October 20, 2015). His suggestion was then to treat it as positive D but trim trailing zeroes. But since then, there is \xintDecToString which can be combined with \xintREZ, and I feel matters of formatting output require a whole module (or rather use existing third-party tools), and I decided to opt rather for an operation similar as the quantize() of Python Decimal module. I.e. we truncate (or round) to an integer multiple of a given power of 10.

Other reason to decide to do this is that it looks as if I don't even need to understand the original code to hack into its ending via \XINT_trunc_G or \XINT_itrunc_G. For the latter it looks as if logically I simply have to do nothing. For the former I simply have to add some eD postfix.

```
\def\xintTrunc  {\romannumeral0\xinttrunc }%
\def\xintiTrunc {\romannumeral0\xintitrunc}%
\def\xinttrunc #1{\expandafter\XINT_trunc\the\numexpr#1.\XINT_trunc_G}%
\def\xintitrunc #1{\expandafter\XINT_trunc\the\numexpr#1.\XINT_itrunc_G}%
\def\XINT_trunc #1.#2#3%
{%
    \expandafter\XINT_trunc_a\romannumeral0\XINT_infrac{#3}#1.#2%
}%
\def\XINT_trunc_a #1#2#3#4.#5%
{%
    \if0\XINT_Sgn#2\xint:\xint_dothis\XINT_trunc_zero\fi
    \if1\XINT_is_One#3XY\xint_dothis\XINT_trunc_sp_b\fi
    \xint_orthat\XINT_trunc_b #1+#4.{#2}{#3}#5#4.%
}%
\def\XINT_trunc_zero #1.#2.{ 0}%
\def\XINT_trunc_b     {\expandafter\XINT_trunc_B\the\numexpr}%
\def\XINT_trunc_sp_b  {\expandafter\XINT_trunc_sp_B\the\numexpr}%
\def\XINT_trunc_B #1%
{%
    \xint_UDsignfork
      #1\XINT_trunc_C
       -\XINT_trunc_D
    \krof #1%
}%
\def\XINT_trunc_sp_B #1%
{%
    \xint_UDsignfork
      #1\XINT_trunc_sp_C
       -\XINT_trunc_sp_D
    \krof #1%
}%
\def\XINT_trunc_C -#1.#2#3%
{%
    \expandafter\XINT_trunc_CE
    \romannumeral0\XINT_dsx_addzeros{#1}#3;.{#2}%
}%
\def\XINT_trunc_CE #1.#2{\XINT_trunc_E #2.{#1}}%


\def\XINT_trunc_sp_C -#1.#2#3{\XINT_trunc_sp_Ca #2.#1.}%
\def\XINT_trunc_sp_Ca #1%
{%
```

```
    \xint_UDsignfork
       #1{\XINT_trunc_sp_Cb -}%
        -{\XINT_trunc_sp_Cb \space#1}%
    \krof
}%
\def\XINT_trunc_sp_Cb #1#2.#3.%
{%
    \expandafter\XINT_trunc_sp_Cc
    \romannumeral0\expandafter\XINT_split_fromright_a
    \the\numexpr#3-\numexpr\XINT_length_loop
    #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
      \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
      \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
    .#2\xint_bye2345678\xint_bye..#1%
}%
\def\XINT_trunc_sp_Cc #1%
{%
    \if.#1\xint_dothis{\XINT_trunc_sp_Cd 0.}\fi
    \xint_orthat {\XINT_trunc_sp_Cd #1}%
}%
\def\XINT_trunc_sp_Cd #1.#2.#3%
{%
    \XINT_trunc_sp_F #3#1.%
}%
\def\XINT_trunc_D #1.#2%
{%
    \expandafter\XINT_trunc_E
    \romannumeral0\XINT_dsx_addzeros {#1}#2;.%
}%
\def\XINT_trunc_sp_D #1.#2#3%
{%
    \expandafter\XINT_trunc_sp_E
    \romannumeral0\XINT_dsx_addzeros {#1}#2;.%
}%
\def\XINT_trunc_E #1%
{%
    \xint_UDsignfork
       #1{\XINT_trunc_F -}%
        -{\XINT_trunc_F \space#1}%
    \krof
}%
\def\XINT_trunc_sp_E #1%
{%
    \xint_UDsignfork
       #1{\XINT_trunc_sp_F -}%
        -{\XINT_trunc_sp_F\space#1}%
    \krof
}%
\def\XINT_trunc_F #1#2.#3#4%
   {\expandafter#4\romannumeral`&&@\expandafter\xint_firstoftwo
                    \romannumeral0\XINT_div_prepare {#3}{#2}.#1}%
\def\XINT_trunc_sp_F #1#2.#3{#3#2.#1}%
```

```
\def\XINT_itrunc_G #1#2.#3#4.%
{%
    \if#10\xint_dothis{ 0}\fi
    \xint_orthat{#3#1}#2%
}%
\def\XINT_trunc_G #1.#2#3#4.%
{%
    \xint_gob_til_minus#3\XINT_trunc_Hc-%
    \expandafter\XINT_trunc_H
    \the\numexpr\romannumeral0\xintlength {#1}-#3#4.#3#4.{#1}#2%
}%
\def\XINT_trunc_Hc-\expandafter\XINT_trunc_H
    \the\numexpr\romannumeral0\xintlength #1.-#2.#3#4{#4#3e#2}%
\def\XINT_trunc_H #1.#2.%
{%
    \ifnum #1 > \xint_c_ \xint_dothis{\XINT_trunc_Ha {#2}}\fi
    \xint_orthat {\XINT_trunc_Hb {-#1}}% -0,--1,--2, ....
}%
\def\XINT_trunc_Ha%
{%
    \expandafter\XINT_trunc_Haa\romannumeral0\xintdecsplit
}%
\def\XINT_trunc_Haa #1#2#3{#3#1.#2}%
\def\XINT_trunc_Hb #1#2#3%
{%
    \expandafter #3\expandafter0\expandafter.%
    \romannumeral\xintreplicate{#1}0#2%
}%
```

8.33 \xintTTrunc

1.1. Modified in 1.2i, it does simply \xintiTrunc0 with no shortcut (the latter having been modified)

```
\def\xintTTrunc {\romannumeral0\xintttrunc }%
\def\xintttrunc {\xintitrunc\xint_c_}%
```

8.34 \xintNum

```
\let\xintnum \xintttrunc
```

8.35 \xintRound, \xintiRound

Modified in 1.2i.

It benefits first of all from the faster \xintTrunc, particularly when the input is already a decimal number (denominator B=1).

And the rounding is now done in 1.2 style (with much delay, sorry), like of the rewritten \xintInc and \xintDec.

At 1.4a, first argument can be negative. This is handled at \XINT_trunc_G.

```
\def\xintRound  {\romannumeral0\xintround }%
\def\xintiRound {\romannumeral0\xintiround }%
\def\xintround  #1{\expandafter\XINT_round\the\numexpr #1.\XINT_round_A}%
\def\xintiround #1{\expandafter\XINT_round\the\numexpr #1.\XINT_iround_A}%
```

```
\def\XINT_round #1.{\expandafter\XINT_round_aa\the\numexpr #1+\xint_c_i.#1.}%
\def\XINT_round_aa #1.#2.#3#4%
{%
    \expandafter\XINT_round_a\romannumeral0\XINT_infrac{#4}#1.#3#2.%
}%
\def\XINT_round_a #1#2#3#4.%
{%
    \if0\XINT_Sgn#2\xint:\xint_dothis\XINT_trunc_zero\fi
    \if1\XINT_is_One#3XY\xint_dothis\XINT_trunc_sp_b\fi
    \xint_orthat\XINT_trunc_b #1+#4.{#2}{#3}%
}%
\def\XINT_round_A{\expandafter\XINT_trunc_G\romannumeral0\XINT_round_B}%
\def\XINT_iround_A{\expandafter\XINT_itrunc_G\romannumeral0\XINT_round_B}%
\def\XINT_round_B #1.%
    {\XINT_dsrr #1\xint_bye\xint_Bye3456789\xint_bye/\xint_c_x\relax.}%
```

8.36 \xintXTrunc

1.09j [2014/01/06] This is completely expandable but not f-expandable. Rewritten for 1.2i (2016/12/04):
- no more use of \xintiloop from xinttools.sty (replaced by \xintreplicate... from xintkernel.sty),
- no more use in 0>N>-D case of a dummy control sequence name via \csname...\endcsname
- handles better the case of an input already a decimal number

Need to transfer code comments into public dtx.

```
\def\xintXTrunc #1%#2%
{%
    \expandafter\XINT_xtrunc_a
    \the\numexpr #1\expandafter.\romannumeral0\xintraw
}%
\def\XINT_xtrunc_a #1.% ?? faire autre chose
{%
    \expandafter\XINT_xtrunc_b\the\numexpr\ifnum#1<\xint_c_i \xint_c_i-\fi #1.%
}%


\def\XINT_xtrunc_b #1.#2{\XINT_xtrunc_c #2{#1}}%


\def\XINT_xtrunc_c #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_xtrunc_zero
        0#1{-\XINT_xtrunc_d {}}%
         0-{\XINT_xtrunc_d #1}%
    \krof
}%[
\def\XINT_xtrunc_zero #1#2]{0.\romannumeral\xintreplicate{#1}0}%


\def\XINT_xtrunc_d #1#2#3/#4[#5]%
{%
    \XINT_xtrunc_prepare_a#4\R\R\R\R\R\R\R {10}0000001\W
```

```
    !{#4};{#5}{#2}{#1#3}%
}%
\def\XINT_xtrunc_prepare_a #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_R #9\XINT_xtrunc_prepare_small\R
    \XINT_xtrunc_prepare_b #9%
}%
\def\XINT_xtrunc_prepare_small\R #1!#2;%
{%
    \ifcase #2
    \or\expandafter\XINT_xtrunc_BisOne
    \or\expandafter\XINT_xtrunc_BisTwo
    \or
    \or\expandafter\XINT_xtrunc_BisFour
    \or\expandafter\XINT_xtrunc_BisFive
    \or
    \or
    \or\expandafter\XINT_xtrunc_BisEight
    \fi\XINT_xtrunc_BisSmall {#2}%
}%


\def\XINT_xtrunc_BisOne\XINT_xtrunc_BisSmall #1#2#3#4%
   {\XINT_xtrunc_sp_e {#2}{#4}{#3}}%
\def\XINT_xtrunc_BisTwo\XINT_xtrunc_BisSmall #1#2#3#4%
{%
    \expandafter\XINT_xtrunc_sp_e\expandafter
    {\the\numexpr #2-\xint_c_i\expandafter}\expandafter
    {\romannumeral0\xintiimul 5{#4}}{#3}%
}%
\def\XINT_xtrunc_BisFour\XINT_xtrunc_BisSmall #1#2#3#4%
{%
    \expandafter\XINT_xtrunc_sp_e\expandafter
    {\the\numexpr #2-\xint_c_ii\expandafter}\expandafter
    {\romannumeral0\xintiimul {25}{#4}}{#3}%
}%
\def\XINT_xtrunc_BisFive\XINT_xtrunc_BisSmall #1#2#3#4%
{%
    \expandafter\XINT_xtrunc_sp_e\expandafter
    {\the\numexpr #2-\xint_c_i\expandafter}\expandafter
    {\romannumeral0\xintdouble {#4}}{#3}%
}%
\def\XINT_xtrunc_BisEight\XINT_xtrunc_BisSmall #1#2#3#4%
{%
    \expandafter\XINT_xtrunc_sp_e\expandafter
    {\the\numexpr #2-\xint_c_iii\expandafter}\expandafter
    {\romannumeral0\xintiimul {125}{#4}}{#3}%
}%
\def\XINT_xtrunc_BisSmall #1%
{%
    \expandafter\XINT_xtrunc_e\expandafter
    {\expandafter\XINT_xtrunc_small_a
    \the\numexpr #1/\xint_c_ii\expandafter
```

```
    .\the\numexpr \xint_c_x^viii+#1!}%
}%

\def\XINT_xtrunc_small_a #1.#2!#3%
{%
    \expandafter\XINT_div_small_b\the\numexpr #1\expandafter
    \xint:\the\numexpr #2\expandafter!%
    \romannumeral0\XINT_div_small_ba #3\R\R\R\R\R\R\R{10}0000001\W
       #3\XINT_sepbyviii_Z_end 2345678\relax
}%

\def\XINT_xtrunc_prepare_b
   {\expandafter\XINT_xtrunc_prepare_c\romannumeral0\XINT_zeroes_forviii }%
\def\XINT_xtrunc_prepare_c #1!%
{%
     \XINT_xtrunc_prepare_d  #1.00000000!{#1}%
}%
\def\XINT_xtrunc_prepare_d #1#2#3#4#5#6#7#8#9%
{%
    \expandafter\XINT_xtrunc_prepare_e
    \xint_gob_til_dot #1#2#3#4#5#6#7#8#9!%
}%
\def\XINT_xtrunc_prepare_e #1!#2!#3#4%
{%
    \XINT_xtrunc_prepare_f #4#3\X {#1}{#3}%
}%
\def\XINT_xtrunc_prepare_f #1#2#3#4#5#6#7#8#9\X
{%
    \expandafter\XINT_xtrunc_prepare_g\expandafter
    \XINT_div_prepare_g
     \the\numexpr  #1#2#3#4#5#6#7#8+\xint_c_i\expandafter
    \xint:\the\numexpr (#1#2#3#4#5#6#7#8+\xint_c_i)/\xint_c_ii\expandafter
    \xint:\the\numexpr #1#2#3#4#5#6#7#8\expandafter
    \xint:\romannumeral0\XINT_sepandrev_andcount
    #1#2#3#4#5#6#7#8#9\XINT_rsepbyviii_end_A 2345678%
                       \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
              \R\xint:\xint_c_xii \R\xint:\xint_c_x  \R\xint:\xint_c_viii \R\xint:\xint_c_vi
              \R\xint:\xint_c_iv  \R\xint:\xint_c_ii \R\xint:\xint_c_\W
    \X
}%

\def\XINT_xtrunc_prepare_g #1;{\XINT_xtrunc_e {#1}}%

\def\XINT_xtrunc_e #1#2%
{%
    \ifnum #2<\xint_c_
        \expandafter\XINT_xtrunc_I
    \else
        \expandafter\XINT_xtrunc_II
    \fi  #2\xint:{#1}%
}%
```

```
\def\XINT_xtrunc_I -#1\xint:#2#3#4%
{%
    \expandafter\XINT_xtrunc_I_a\romannumeral0#2{#4}{#2}{#1}{#3}%
}%

\def\XINT_xtrunc_I_a #1#2#3#4#5%
{%
    \expandafter\XINT_xtrunc_I_b\the\numexpr #4-#5\xint:#4\xint:{#5}{#2}{#3}{#1}%
}%

\def\XINT_xtrunc_I_b #1%
{%
    \xint_UDsignfork
      #1\XINT_xtrunc_IA_c
       -\XINT_xtrunc_IB_c
    \krof #1%
}%

\def\XINT_xtrunc_IA_c -#1\xint:#2\xint:#3#4#5#6%
{%
   \expandafter\XINT_xtrunc_IA_d
   \the\numexpr#2-\xintLength{#6}\xint:{#6}%
   \expandafter\XINT_xtrunc_IA_xd
   \the\numexpr (#1+\xint_c_ii^v)/\xint_c_ii^vi-\xint_c_i\xint:#1\xint:{#5}{#4}%
}%

\def\XINT_xtrunc_IA_d #1%
{%
    \xint_UDsignfork
      #1\XINT_xtrunc_IAA_e
       -\XINT_xtrunc_IAB_e
    \krof #1%
}%

\def\XINT_xtrunc_IAA_e -#1\xint:#2%
{%
    \romannumeral0\XINT_split_fromleft
    #1.#2\xint_gobble_i\xint_bye2345678\xint_bye..%
}%

\def\XINT_xtrunc_IAB_e #1\xint:#2%
{%
    0.\romannumeral\XINT_rep#1\endcsname0#2%
}%

\def\XINT_xtrunc_IA_xd #1\xint:#2\xint:%
{%
    \expandafter\XINT_xtrunc_IA_xe\the\numexpr #2-\xint_c_ii^vi*#1\xint:#1\xint:%
}%
```

```
\def\XINT_xtrunc_IA_xe #1\xint:#2\xint:#3#4%
{%
    \XINT_xtrunc_loop {#2}{#4}{#3}{#1}%
}%

\def\XINT_xtrunc_IB_c #1\xint:#2\xint:#3#4#5#6%
{%
    \expandafter\XINT_xtrunc_IB_d
    \romannumeral0\XINT_split_xfork #1.#6\xint_bye2345678\xint_bye..{#3}%
}%

\def\XINT_xtrunc_IB_d #1.#2.#3%
{%
    \expandafter\XINT_xtrunc_IA_d\the\numexpr#3-\xintLength {#1}\xint:{#1}%
}%

\def\XINT_xtrunc_II #1\xint:%
{%
    \expandafter\XINT_xtrunc_II_a\romannumeral\xintreplicate{#1}0\xint:%
}%
\def\XINT_xtrunc_II_a #1\xint:#2#3#4%
{%
    \expandafter\XINT_xtrunc_II_b
    \the\numexpr (#3+\xint_c_ii^v)/\xint_c_ii^vi-\xint_c_i\expandafter\xint:%
    \the\numexpr #3\expandafter\xint:\romannumeral0#2{#4#1}{#2}%
}%
\def\XINT_xtrunc_II_b #1\xint:#2\xint:%
{%
    \expandafter\XINT_xtrunc_II_c\the\numexpr #2-\xint_c_ii^vi*#1\xint:#1\xint:%
}%

\def\XINT_xtrunc_II_c #1\xint:#2\xint:#3#4#5%
{%
    #3.\XINT_xtrunc_loop {#2}{#4}{#5}{#1}%
}%

\def\XINT_xtrunc_loop #1%
{%
    \ifnum #1=\xint_c_ \expandafter\XINT_xtrunc_transition\fi
    \expandafter\XINT_xtrunc_loop_a\the\numexpr #1-\xint_c_i\xint:%
}%
\def\XINT_xtrunc_loop_a #1\xint:#2#3%
{%
    \expandafter\XINT_xtrunc_loop_b\romannumeral0#3%
    {#200000000000000000000000000000000000000000000000000000000000000000}%
    {#1}{#3}%
}%
\def\XINT_xtrunc_loop_b #1#2#3%
{%
    \romannumeral\xintreplicate{\xint_c_ii^vi-\xintLength{#1}}0#1%
    \XINT_xtrunc_loop {#3}{#2}%
}%
```

```
\def\XINT_xtrunc_transition
    \expandafter\XINT_xtrunc_loop_a\the\numexpr #1\xint:#2#3#4%
{%
    \ifnum #4=\xint_c_ \expandafter\xint_gobble_vi\fi
    \expandafter\XINT_xtrunc_finish\expandafter
    {\romannumeral0\XINT_dsx_addzeros{#4}#2;}{#3}{#4}%
}%
\def\XINT_xtrunc_finish #1#2%
{%
    \expandafter\XINT_xtrunc_finish_a\romannumeral0#2{#1}%
}%
\def\XINT_xtrunc_finish_a #1#2#3%
{%
    \romannumeral\xintreplicate{#3-\xintLength{#1}}0#1%
}%

\def\XINT_xtrunc_sp_e #1%
{%
    \ifnum #1<\xint_c_
        \expandafter\XINT_xtrunc_sp_I
    \else
        \expandafter\XINT_xtrunc_sp_II
    \fi  #1\xint:%
}%

\def\XINT_xtrunc_sp_I -#1\xint:#2#3%
{%
    \expandafter\XINT_xtrunc_sp_I_a\the\numexpr #1-#3\xint:#1\xint:{#3}{#2}%
}%

\def\XINT_xtrunc_sp_I_a #1%
{%
    \xint_UDsignfork
      #1\XINT_xtrunc_sp_IA_b
       -\XINT_xtrunc_sp_IB_b
    \krof #1%
}%

\def\XINT_xtrunc_sp_IA_b -#1\xint:#2\xint:#3#4%
{%
   \expandafter\XINT_xtrunc_sp_IA_c
   \the\numexpr#2-\xintLength{#4}\xint:{#4}\romannumeral\XINT_rep#1\endcsname0%
}%

\def\XINT_xtrunc_sp_IA_c #1%
{%
    \xint_UDsignfork
      #1\XINT_xtrunc_sp_IAA
       -\XINT_xtrunc_sp_IAB
    \krof #1%
}%
```

```
\def\XINT_xtrunc_sp_IAA -#1\xint:#2%
{%
    \romannumeral0\XINT_split_fromleft
    #1.#2\xint_gobble_i\xint_bye2345678\xint_bye..%
}%

\def\XINT_xtrunc_sp_IAB #1\xint:#2%
{%
    0.\romannumeral\XINT_rep#1\endcsname0#2%
}%

\def\XINT_xtrunc_sp_IB_b #1\xint:#2\xint:#3#4%
{%
    \expandafter\XINT_xtrunc_sp_IB_c
    \romannumeral0\XINT_split_xfork #1.#4\xint_bye2345678\xint_bye..{#3}%
}%

\def\XINT_xtrunc_sp_IB_c #1.#2.#3%
{%
    \expandafter\XINT_xtrunc_sp_IA_c\the\numexpr#3-\xintLength {#1}\xint:{#1}%
}%

\def\XINT_xtrunc_sp_II #1\xint:#2#3%
{%
    #2\romannumeral\XINT_rep#1\endcsname0.\romannumeral\XINT_rep#3\endcsname0%
}%
```

8.37 \xintAdd

Big change at 1.3: a/b+c/d uses lcm(b,d) as denominator.

```
\def\xintAdd {\romannumeral0\xintadd }%
\def\xintadd #1{\expandafter\XINT_fadd\romannumeral0\xintraw {#1}}%
\def\XINT_fadd #1{\xint_gob_til_zero #1\XINT_fadd_Azero 0\XINT_fadd_a #1}%
\def\XINT_fadd_Azero #1]{\xintraw }%
\def\XINT_fadd_a #1/#2[#3]#4%
   {\expandafter\XINT_fadd_b\romannumeral0\xintraw {#4}{#3}{#1}{#2}}%
\def\XINT_fadd_b #1{\xint_gob_til_zero #1\XINT_fadd_Bzero 0\XINT_fadd_c #1}%
\def\XINT_fadd_Bzero #1]#2#3#4{ #3/#4[#2]}%
\def\XINT_fadd_c #1/#2[#3]#4%
{%
    \expandafter\XINT_fadd_Aa\the\numexpr #4-#3.{#3}{#4}{#1}{#2}%
}%
\def\XINT_fadd_Aa #1%
{%
    \xint_UDzerominusfork
       #1-\XINT_fadd_B
        0#1\XINT_fadd_Bb
        0-\XINT_fadd_Ba
    \krof #1%
}%
```

```
\def\XINT_fadd_B   #1.#2#3#4#5#6#7{\XINT_fadd_C {#4}{#5}{#7}{#6}[#3]}%
\def\XINT_fadd_Ba  #1.#2#3#4#5#6#7%
{%
    \expandafter\XINT_fadd_C\expandafter
        {\romannumeral0\XINT_dsx_addzeros {#1}#6;}%
    {#7}{#5}{#4}[#2]%
}%
\def\XINT_fadd_Bb -#1.#2#3#4#5#6#7%
{%
    \expandafter\XINT_fadd_C\expandafter
        {\romannumeral0\XINT_dsx_addzeros {#1}#4;}%
    {#5}{#7}{#6}[#3]%
}%
\def\XINT_fadd_iszero #1[#2]{ 0/1[0]}% ou [#2] originel?
\def\XINT_fadd_C #1#2#3%
{%
    \expandafter\XINT_fadd_D_b
    \romannumeral0\XINT_div_prepare{#2}{#3}{#2}{#2}{#3}{#1}%
}%
```

Basically a clone of the \XINT_irr_loop_a loop. I should modify the output of \XINT_div_prepare perhaps to be optimized for checking if remainder vanishes.

```
\def\XINT_fadd_D_a #1#2%
{%
    \expandafter\XINT_fadd_D_b
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
\def\XINT_fadd_D_b #1#2{\XINT_fadd_D_c #2\Z}%
\def\XINT_fadd_D_c #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_fadd_D_exit0\XINT_fadd_D_a {#1#2}%
}%
\def\XINT_fadd_D_exit0\XINT_fadd_D_a #1#2#3%
{%
    \expandafter\XINT_fadd_E
    \romannumeral0\xintiiquo {#3}{#2}.{#2}%
}%
\def\XINT_fadd_E #1.#2#3%
{%
    \expandafter\XINT_fadd_F
    \romannumeral0\xintiimul{#1}{#3}.{\xintiiQuo{#3}{#2}}{#1}%
}%
\def\XINT_fadd_F #1.#2#3#4#5%
{%
    \expandafter\XINT_fadd_G
    \romannumeral0\xintiiadd{\xintiiMul{#2}{#4}}{\xintiiMul{#3}{#5}}/#1%
}%
\def\XINT_fadd_G #1{%
\def\XINT_fadd_G ##1{\if0##1\expandafter\XINT_fadd_iszero\fi#1##1}%
}\XINT_fadd_G{ }%
```

8.38 \xintSub

Since 1.3 will use least common multiple of denominators.

```
\def\xintSub   {\romannumeral0\xintsub }%
\def\xintsub #1{\expandafter\XINT_fsub\romannumeral0\xintraw {#1}}%
\def\XINT_fsub #1{\xint_gob_til_zero #1\XINT_fsub_Azero 0\XINT_fsub_a #1}%
\def\XINT_fsub_Azero #1]{\xintopp }%
\def\XINT_fsub_a #1/#2[#3]#4%
   {\expandafter\XINT_fsub_b\romannumeral0\xintraw {#4}{#3}{#1}{#2}}%
\def\XINT_fsub_b #1{\xint_UDzerominusfork
                     #1-\XINT_fadd_Bzero
                      0#1\XINT_fadd_c
                      0-{\XINT_fadd_c -#1}%
                    \krof }%
```

8.39 \xintSum

There was (not documented anymore since 1.09d, 2013/10/22) a macro \xintSumExpr, but it has been deleted at 1.2l.

Empty items in the input are not accepted by this macro, but the input may be empty.

Refactored slightly at 1.4. \XINT_Sum used in xintexpr code.

```
\def\xintSum {\romannumeral0\xintsum }%
\def\xintsum #1{\expandafter\XINT_sum\romannumeral`&&@#1^}%
\def\XINT_Sum{\romannumeral0\XINT_sum}%
\def\XINT_sum#1%
{%
    \xint_gob_til_^ #1\XINT_sum_empty ^%
    \expandafter\XINT_sum_loop\romannumeral0\xintraw{#1}\xint:
}%
\def\XINT_sum_empty ^#1\xint:{ 0/1[0]}%
\def\XINT_sum_loop #1\xint:#2%
{%
    \xint_gob_til_^ #2\XINT_sum_end ^%
    \expandafter\XINT_sum_loop
    \romannumeral0\xintadd{#1}{\romannumeral0\xintraw{#2}}\xint:
}%
\def\XINT_sum_end ^#1\xintadd #2#3\xint:{ #2}%
```

8.40 \xintMul

```
\def\xintMul {\romannumeral0\xintmul }%
\def\xintmul #1{\expandafter\XINT_fmul\romannumeral0\xintraw {#1}.}%
\def\XINT_fmul #1{\xint_gob_til_zero #1\XINT_fmul_zero 0\XINT_fmul_a #1}%
\def\XINT_fmul_a #1[#2].#3%
   {\expandafter\XINT_fmul_b\romannumeral0\xintraw {#3}#1[#2.]}%
\def\XINT_fmul_b #1{\xint_gob_til_zero #1\XINT_fmul_zero 0\XINT_fmul_c #1}%
\def\XINT_fmul_c #1/#2[#3]#4/#5[#6.]%
{%
    \expandafter\XINT_fmul_d
    \expandafter{\the\numexpr #3+#6\expandafter}%
    \expandafter{\romannumeral0\xintiimul {#5}{#2}}%
```

```
    {\romannumeral0\xintiimul {#4}{#1}}%
}%
\def\XINT_fmul_d #1#2#3%
{%
    \expandafter \XINT_fmul_e \expandafter{#3}{#1}{#2}%
}%
\def\XINT_fmul_e #1#2{\XINT_outfrac {#2}{#1}}%
\def\XINT_fmul_zero #1.#2{ 0/1[0]}%
```

8.41 \xintSqr

1.1 modifs comme xintMul.

```
\def\xintSqr {\romannumeral0\xintsqr }%
\def\xintsqr #1{\expandafter\XINT_fsqr\romannumeral0\xintraw {#1}}%
\def\XINT_fsqr #1{\xint_gob_til_zero #1\XINT_fsqr_zero 0\XINT_fsqr_a #1}%
\def\XINT_fsqr_a #1/#2[#3]%
{%
    \expandafter\XINT_fsqr_b
    \expandafter{\the\numexpr #3+#3\expandafter}%
    \expandafter{\romannumeral0\xintiisqr {#2}}%
    {\romannumeral0\xintiisqr {#1}}%
}%
\def\XINT_fsqr_b #1#2#3{\expandafter \XINT_fmul_e \expandafter{#3}{#1}{#2}}%
\def\XINT_fsqr_zero #1]{ 0/1[0]}%
```

8.42 \xintPow

1.2f: to be coherent with the "i" convention \xintiPow should parse also its exponent via \xintNum when xintfrac.sty is loaded. This was not the case so far. Cependant le problème est que le fait d'appliquer \xintNum rend impossible certains inputs qui auraient pu être gérés par \numexpr. Le \numexpr externe est ici pour intercepter trop grand input.

```
\def\xintipow #1#2%
{%
    \expandafter\xint_pow\the\numexpr \xintNum{#2}\expandafter
    .\romannumeral0\xintnum{#1}\xint:
}%
\def\xintPow {\romannumeral0\xintpow }%
\def\xintpow #1%
{%
    \expandafter\XINT_fpow\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\XINT_fpow #1#2%
{%
    \expandafter\XINT_fpow_fork\the\numexpr \xintNum{#2}\relax\Z #1%
}%
\def\XINT_fpow_fork #1#2\Z
{%
    \xint_UDzerominusfork
      #1-\XINT_fpow_zero
      0#1\XINT_fpow_neg
       0-{\XINT_fpow_pos #1}%
```

```
    \krof
    {#2}%
}%
\def\XINT_fpow_zero #1#2#3#4{ 1/1[0]}%
\def\XINT_fpow_pos #1#2#3#4#5%
{%
    \expandafter\XINT_fpow_pos_A\expandafter
    {\the\numexpr #1#2*#3\expandafter}\expandafter
    {\romannumeral0\xintiipow {#5}{#1#2}}%
    {\romannumeral0\xintiipow {#4}{#1#2}}%
}%
\def\XINT_fpow_neg #1#2#3#4%
{%
    \expandafter\XINT_fpow_pos_A\expandafter
    {\the\numexpr -#1*#2\expandafter}\expandafter
    {\romannumeral0\xintiipow {#3}{#1}}%
    {\romannumeral0\xintiipow {#4}{#1}}%
}%
\def\XINT_fpow_pos_A #1#2#3%
{%
    \expandafter\XINT_fpow_pos_B\expandafter {#3}{#1}{#2}%
}%
\def\XINT_fpow_pos_B #1#2{\XINT_outfrac {#2}{#1}}%
```

8.43 \xintFac

Factorial coefficients: variant which can be chained with other xintfrac macros. \xintiFac deprecated at 1.2o and removed at 1.3; \xintFac used by xintexpr.sty.

```
\def\xintFac  {\romannumeral0\xintfac}%
\def\xintfac  #1{\expandafter\XINT_fac_fork\the\numexpr\xintNum{#1}.[0]}%
```

8.44 \xintBinomial

1.2f. Binomial coefficients. \xintiBinomial deprecated at 1.2o and removed at 1.3; \xintBinomial needed by xintexpr.sty.

```
\def\xintBinomial {\romannumeral0\xintbinomial}%
\def\xintbinomial #1#2%
{%
    \expandafter\XINT_binom_pre
    \the\numexpr\xintNum{#1}\expandafter.\the\numexpr\xintNum{#2}.[0]%
}%
```

8.45 \xintPFactorial

1.2f. Partial factorial. For needs of xintexpr.sty.

```
\def\xintipfactorial #1#2%
{%
    \expandafter\XINT_pfac_fork
    \the\numexpr\xintNum{#1}\expandafter.\the\numexpr\xintNum{#2}.%
}%
```

```
\def\xintPFactorial {\romannumeral0\xintpfactorial}%
\def\xintpfactorial #1#2%
{%
    \expandafter\XINT_pfac_fork
    \the\numexpr\xintNum{#1}\expandafter.\the\numexpr\xintNum{#2}.[0]%
}%
```

8.46 \xintPrd

Refactored at 1.4. After some hesitation the routine still does not try to detect on the fly a zero item, to abort the loop. Indeed this would add some overhead generally (as we need normalizing the item before checking if it vanishes hence we must then grab things once more).

```
\def\xintPrd {\romannumeral0\xintprd }%
\def\xintprd #1{\expandafter\XINT_prd\romannumeral`&&@#1^}%
\def\XINT_Prd{\romannumeral0\XINT_prd}%
\def\XINT_prd#1%
{%
    \xint_gob_til_^ #1\XINT_prd_empty ^%
    \expandafter\XINT_prd_loop\romannumeral0\xintraw{#1}\xint:
}%
\def\XINT_prd_empty ^#1\xint:{ 1/1[0]}%
\def\XINT_prd_loop #1\xint:#2%
{%
    \xint_gob_til_^ #2\XINT_prd_end ^%
    \expandafter\XINT_prd_loop
    \romannumeral0\xintmul{#1}{\romannumeral0\xintraw{#2}}\xint:
}%
\def\XINT_prd_end ^#1\xintmul #2#3\xint:{ #2}%
```

8.47 \xintDiv

```
\def\xintDiv {\romannumeral0\xintdiv }%
\def\xintdiv #1%
{%
    \expandafter\XINT_fdiv\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\XINT_fdiv #1#2%
   {\expandafter\XINT_fdiv_A\romannumeral0\XINT_infrac {#2}#1}%
\def\XINT_fdiv_A #1#2#3#4#5#6%
{%
    \expandafter\XINT_fdiv_B
    \expandafter{\the\numexpr #4-#1\expandafter}%
    \expandafter{\romannumeral0\xintiimul {#2}{#6}}%
    {\romannumeral0\xintiimul {#3}{#5}}%
}%
\def\XINT_fdiv_B #1#2#3%
{%
    \expandafter\XINT_fdiv_C
    \expandafter{#3}{#1}{#2}%
}%
\def\XINT_fdiv_C #1#2{\XINT_outfrac {#2}{#1}}%
```

8.48 \xintDivFloor

1.1. Changed at 1.2p to not append /1[0] ending but rather output a big integer in strict format, like \xintDivTrunc and \xintDivRound.

```
\def\xintDivFloor     {\romannumeral0\xintdivfloor }%
\def\xintdivfloor #1#2{\xintifloor{\xintDiv {#1}{#2}}}%
```

8.49 \xintDivTrunc

1.1. \xintttrunc rather than \xintitrunc0 in 1.1a

```
\def\xintDivTrunc     {\romannumeral0\xintdivtrunc }%
\def\xintdivtrunc #1#2{\xintttrunc {\xintDiv {#1}{#2}}}%
```

8.50 \xintDivRound

1.1

```
\def\xintDivRound     {\romannumeral0\xintdivround }%
\def\xintdivround #1#2{\xintiround 0{\xintDiv {#1}{#2}}}%
```

8.51 \xintModTrunc

1.1. \xintModTrunc {q1}{q2} computes q1 - q2*t(q1/q2) with t(q1/q2) equal to the truncated division of two fractions q1 and q2.
  Its former name, prior to 1.2p, was \xintMod.
  At 1.3, uses least common multiple denominator, like \xintMod (next).

```
\def\xintModTrunc {\romannumeral0\xintmodtrunc }%
\def\xintmodtrunc #1{\expandafter\XINT_modtrunc_a\romannumeral0\xintraw{#1}.}%
\def\XINT_modtrunc_a #1#2.#3%
   {\expandafter\XINT_modtrunc_b\expandafter #1\romannumeral0\xintraw{#3}#2.}%
\def\XINT_modtrunc_b #1#2% #1 de A, #2 de B.
{%
    \if0#2\xint_dothis{\XINT_modtrunc_divbyzero #1#2}\fi
    \if0#1\xint_dothis\XINT_modtrunc_aiszero\fi
    \if-#2\xint_dothis{\XINT_modtrunc_bneg #1}\fi
          \xint_orthat{\XINT_modtrunc_bpos #1#2}%
}%
\def\XINT_modtrunc_divbyzero #1#2[#3]#4.%
{%
    \XINT_signalcondition{DivisionByZero}{Division by #2[#3] of #1#4}{}{0/1[0]}%
}%
\def\XINT_modtrunc_aiszero #1.{ 0/1[0]}%
\def\XINT_modtrunc_bneg #1%
{%
    \xint_UDsignfork
         #1{\xintiiopp\XINT_modtrunc_pos {}}%
          -{\XINT_modtrunc_pos #1}%
    \krof
}%
\def\XINT_modtrunc_bpos #1%
```

```
{%
    \xint_UDsignfork
            #1{\xintiiopp\XINT_modtrunc_pos {}}%
              -{\XINT_modtrunc_pos #1}%
    \krof
}%
```

Attention. This crucially uses that xint's \xintiiE{x}{e} is defined to return x unchanged if e is negative (and x extended by e zeroes if e >= 0).

```
\def\XINT_modtrunc_pos #1#2/#3[#4]#5/#6[#7].%
{%
    \expandafter\XINT_modtrunc_pos_a
    \the\numexpr\ifnum#7>#4 #4\else #7\fi\expandafter.%
    \romannumeral0\expandafter\XINT_mod_D_b
    \romannumeral0\XINT_div_prepare{#3}{#6}{#3}{#3}{#6}%
    {#1#5}{#7-#4}{#2}{#4-#7}%
}%
\def\XINT_modtrunc_pos_a #1.#2#3#4{\xintiirem {#3}{#4}/#2[#1]}%
```

8.52 \xintDivMod

1.2p. \xintDivMod{q1}{q2} outputs {floor(q1/q2)}{q1 - q2*floor(q1/q2)}. Attention that it relies on \xintiiE{x}{e} returning x if e < 0.

Modified (like \xintAdd and \xintSub) at 1.3 to use a l.c.m for final denominator of the "mod" part.

```
\def\xintDivMod {\romannumeral0\xintdivmod }%
\def\xintdivmod #1{\expandafter\XINT_divmod_a\romannumeral0\xintraw{#1}.}%
\def\XINT_divmod_a #1#2.#3%
   {\expandafter\XINT_divmod_b\expandafter #1\romannumeral0\xintraw{#3}#2.}%
\def\XINT_divmod_b #1#2% #1 de A, #2 de B.
{%
    \if0#2\xint_dothis{\XINT_divmod_divbyzero #1#2}\fi
    \if0#1\xint_dothis\XINT_divmod_aiszero\fi
    \if-#2\xint_dothis{\XINT_divmod_bneg #1}\fi
          \xint_orthat{\XINT_divmod_bpos #1#2}%
}%
\def\XINT_divmod_divbyzero #1#2[#3]#4.%
{%
    \XINT_signalcondition{DivisionByZero}{Division by #2[#3] of #1#4}{}%
    {{0}{0/1[0]}}% à revoir...
}%
\def\XINT_divmod_aiszero #1.{{0}{0/1[0]}}%
\def\XINT_divmod_bneg #1% f // -g = (-f) // g, f % -g = - ((-f) % g)
{%
    \expandafter\XINT_divmod_bneg_finish
    \romannumeral0\xint_UDsignfork
        #1{\XINT_divmod_bpos {}}%
          -{\XINT_divmod_bpos {-#1}}%
    \krof
}%
\def\XINT_divmod_bneg_finish#1#2%
```

```
{%
    \expandafter\xint_exchangetwo_keepbraces\expandafter
    {\romannumeral0\xintiiopp#2}{#1}%
}%
\def\XINT_divmod_bpos #1#2/#3[#4]#5/#6[#7].%
{%
    \expandafter\XINT_divmod_bpos_a
    \the\numexpr\ifnum#7>#4 #4\else #7\fi\expandafter.%
    \romannumeral0\expandafter\XINT_mod_D_b
    \romannumeral0\XINT_div_prepare{#3}{#6}{#3}{#3}{#6}%
    {#1#5}{#7-#4}{#2}{#4-#7}%
}%
\def\XINT_divmod_bpos_a #1.#2#3#4%
{%
    \expandafter\XINT_divmod_bpos_finish
    \romannumeral0\xintiidivision{#3}{#4}{/#2[#1]}%
}%
\def\XINT_divmod_bpos_finish #1#2#3{{#1}{#2#3}}%
```

8.53 \xintMod

1.2p. \xintMod{q1}{q2} computes q1 - q2*floor(q1/q2). Attention that it relies on \xintiiE{x}{e} returning x if e < 0.
Prior to 1.2p, that macro had the meaning now attributed to \xintModTrunc.
Modified (like \xintAdd and \xintSub) at 1.3 to use a l.c.m for final denominator.

```
\def\xintMod {\romannumeral0\xintmod }%
\def\xintmod #1{\expandafter\XINT_mod_a\romannumeral0\xintraw{#1}.}%
\def\XINT_mod_a #1#2.#3%
   {\expandafter\XINT_mod_b\expandafter #1\romannumeral0\xintraw{#3}#2.}%
\def\XINT_mod_b #1#2% #1 de A, #2 de B.
{%
    \if0#2\xint_dothis{\XINT_mod_divbyzero #1#2}\fi
    \if0#1\xint_dothis\XINT_mod_aiszero\fi
    \if-#2\xint_dothis{\XINT_mod_bneg #1}\fi
          \xint_orthat{\XINT_mod_bpos #1#2}%
}%
```

Attention to not move ModTrunc code beyond that point.

```
\let\XINT_mod_divbyzero\XINT_modtrunc_divbyzero
\let\XINT_mod_aiszero  \XINT_modtrunc_aiszero
\def\XINT_mod_bneg #1% f % -g = - ((-f) % g), for g > 0
{%
    \xintiiopp\xint_UDsignfork
        #1{\XINT_mod_bpos {}}%
         -{\XINT_mod_bpos {-#1}}%
    \krof
}%
\def\XINT_mod_bpos #1#2/#3[#4]#5/#6[#7].%
{%
    \expandafter\XINT_mod_bpos_a
    \the\numexpr\ifnum#7>#4 #4\else #7\fi\expandafter.%
```

```
    \romannumeral0\expandafter\XINT_mod_D_b
    \romannumeral0\XINT_div_prepare{#3}{#6}{#3}{#3}{#6}%
    {#1#5}{#7-#4}{#2}{#4-#7}%
}%
\def\XINT_mod_D_a #1#2%
{%
    \expandafter\XINT_mod_D_b
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
\def\XINT_mod_D_b #1#2{\XINT_mod_D_c #2\Z}%
\def\XINT_mod_D_c #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_mod_D_exit0\XINT_mod_D_a {#1#2}%
}%
\def\XINT_mod_D_exit0\XINT_mod_D_a #1#2#3%
{%
    \expandafter\XINT_mod_E
    \romannumeral0\xintiiquo {#3}{#2}.{#2}%
}%
\def\XINT_mod_E #1.#2#3%
{%
    \expandafter\XINT_mod_F
    \romannumeral0\xintiimul{#1}{#3}.{\xintiiQuo{#3}{#2}}{#1}%
}%
\def\XINT_mod_F #1.#2#3#4#5#6#7%
{%
    {#1}{\xintiiE{\xintiiMul{#4}{#3}}{#5}}%
        {\xintiiE{\xintiiMul{#6}{#2}}{#7}}%
}%
\def\XINT_mod_bpos_a #1.#2#3#4{\xintiirem {#3}{#4}/#2[#1]}%
```

8.54 \xintIsOne

New with 1.09a. Could be more efficient. For fractions with big powers of tens, it is better to use \xintCmp{f}{1}. Restyled in 1.09i.

```
\def\xintIsOne   {\romannumeral0\xintisone }%
\def\xintisone #1{\expandafter\XINT_fracisone
                  \romannumeral0\xintrawwithzeros{#1}\Z }%
\def\XINT_fracisone #1/#2\Z
    {\if0\xintiiCmp {#1}{#2}\xint_afterfi{ 1}\else\xint_afterfi{ 0}\fi}%
```

8.55 \xintGeq

```
\def\xintGeq {\romannumeral0\xintgeq }%
\def\xintgeq #1%
{%
    \expandafter\XINT_fgeq\expandafter {\romannumeral0\xintabs {#1}}%
}%
\def\XINT_fgeq #1#2%
{%
    \expandafter\XINT_fgeq_A \romannumeral0\xintabs {#2}#1%
}%
```

```
\def\XINT_fgeq_A #1%
{%
    \xint_gob_til_zero #1\XINT_fgeq_Zii 0%
    \XINT_fgeq_B #1%
}%
\def\XINT_fgeq_Zii 0\XINT_fgeq_B #1[#2]#3[#4]{ 1}%
\def\XINT_fgeq_B #1/#2[#3]#4#5/#6[#7]%
{%
    \xint_gob_til_zero #4\XINT_fgeq_Zi 0%
    \expandafter\XINT_fgeq_C\expandafter
    {\the\numexpr #7-#3\expandafter}\expandafter
    {\romannumeral0\xintiimul {#4#5}{#2}}%
    {\romannumeral0\xintiimul {#6}{#1}}%
}%
\def\XINT_fgeq_Zi 0#1#2#3#4#5#6#7{ 0}%
\def\XINT_fgeq_C #1#2#3%
{%
    \expandafter\XINT_fgeq_D\expandafter
    {#3}{#1}{#2}%
}%
\def\XINT_fgeq_D #1#2#3%
{%
    \expandafter\XINT_cntSgnFork\romannumeral`&&@\expandafter\XINT_cntSgn
     \the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax\xint:
    { 0}{\XINT_fgeq_E #2\Z {#3}{#1}}{ 1}%
}%
\def\XINT_fgeq_E #1%
{%
    \xint_UDsignfork
        #1\XINT_fgeq_Fd
         -{\XINT_fgeq_Fn #1}%
    \krof
}%

\def\XINT_fgeq_Fd #1\Z #2#3%
{%
    \expandafter\XINT_fgeq_Fe
    \romannumeral0\XINT_dsx_addzeros {#1}#3;\xint:#2\xint:
}%
\def\XINT_fgeq_Fe #1\xint:#2#3\xint:{\XINT_geq_plusplus #2#1\xint:#3\xint:}%
\def\XINT_fgeq_Fn #1\Z #2#3%
{%
    \expandafter\XINT_fgeq_Fo
    \romannumeral0\XINT_dsx_addzeros {#1}#2;\xint:#3\xint:
}%
\def\XINT_fgeq_Fo #1#2\xint:#3\xint:{\XINT_geq_plusplus #1#3\xint:#2\xint:}%
```

8.56 \xintMax

```
\def\xintMax {\romannumeral0\xintmax }%
\def\xintmax #1%
{%
    \expandafter\XINT_fmax\expandafter {\romannumeral0\xintraw {#1}}%
}%
```

```
\def\XINT_fmax #1#2%
{%
    \expandafter\XINT_fmax_A\romannumeral0\xintraw {#2}#1%
}%
\def\XINT_fmax_A #1#2/#3[#4]#5#6/#7[#8]%
{%
    \xint_UDsignsfork
      #1#5\XINT_fmax_minusminus
       -#5\XINT_fmax_firstneg
       #1-\XINT_fmax_secondneg
        --\XINT_fmax_nonneg_a
    \krof
    #1#5{#2/#3[#4]}{#6/#7[#8]}%
}%
\def\XINT_fmax_minusminus --%
   {\expandafter-\romannumeral0\XINT_fmin_nonneg_b }%
\def\XINT_fmax_firstneg #1-#2#3{ #1#2}%
\def\XINT_fmax_secondneg -#1#2#3{ #1#3}%
\def\XINT_fmax_nonneg_a #1#2#3#4%
{%
    \XINT_fmax_nonneg_b {#1#3}{#2#4}%
}%
\def\XINT_fmax_nonneg_b #1#2%
{%
    \if0\romannumeral0\XINT_fgeq_A #1#2%
          \xint_afterfi{ #1}%
    \else \xint_afterfi{ #2}%
    \fi
}%
```

8.57 \xintMaxof

1.2l protects \xintMaxof against items with non terminated \the\numexpr expressions.
1.4 renders the macro compatible with an empty argument and it also defines an accessor \XINT_Maxof suitable for xintexpr usage (formerly xintexpr had its own macro handling comma separated values, but it changed internal representation at 1.4).

```
\def\xintMaxof {\romannumeral0\xintmaxof }%
\def\xintmaxof #1{\expandafter\XINT_maxof\romannumeral`&&@#1^}%
\def\XINT_Maxof{\romannumeral0\XINT_maxof}%
\def\XINT_maxof#1%
{%
    \xint_gob_til_^ #1\XINT_maxof_empty ^%
    \expandafter\XINT_maxof_loop\romannumeral0\xintraw{#1}\xint:
}%
\def\XINT_maxof_empty ^#1\xint:{ 0/1[0]}%
\def\XINT_maxof_loop #1\xint:#2%
{%
    \xint_gob_til_^ #2\XINT_maxof_e ^%
    \expandafter\XINT_maxof_loop
    \romannumeral0\xintmax{#1}{\romannumeral0\xintraw{#2}}\xint:
}%
\def\XINT_maxof_e ^#1\xintmax #2#3\xint:{ #2}%
```

8.58 \xintMin

```
\def\xintMin {\romannumeral0\xintmin }%
\def\xintmin #1%
{%
    \expandafter\XINT_fmin\expandafter {\romannumeral0\xintraw {#1}}%
}%
\def\XINT_fmin #1#2%
{%
    \expandafter\XINT_fmin_A\romannumeral0\xintraw {#2}#1%
}%
\def\XINT_fmin_A #1#2/#3[#4]#5#6/#7[#8]%
{%
    \xint_UDsignsfork
      #1#5\XINT_fmin_minusminus
       -#5\XINT_fmin_firstneg
       #1-\XINT_fmin_secondneg
        --\XINT_fmin_nonneg_a
    \krof
    #1#5{#2/#3[#4]}{#6/#7[#8]}%
}%
\def\XINT_fmin_minusminus --%
   {\expandafter-\romannumeral0\XINT_fmax_nonneg_b }%
\def\XINT_fmin_firstneg #1-#2#3{ -#3}%
\def\XINT_fmin_secondneg -#1#2#3{ -#2}%
\def\XINT_fmin_nonneg_a #1#2#3#4%
{%
    \XINT_fmin_nonneg_b {#1#3}{#2#4}%
}%
\def\XINT_fmin_nonneg_b #1#2%
{%
    \if0\romannumeral0\XINT_fgeq_A #1#2%
          \xint_afterfi{ #2}%
    \else \xint_afterfi{ #1}%
    \fi
}%
```

8.59 \xintMinof

1.2l protects \xintMinof against items with non terminated \the\numexpr expressions.
1.4 version is compatible with an empty input (empty items are handled as zero).

```
\def\xintMinof {\romannumeral0\xintminof }%
\def\xintminof #1{\expandafter\XINT_minof\romannumeral`&&@#1^}%
\def\XINT_Minof{\romannumeral0\XINT_minof}%
\def\XINT_minof#1%
{%
    \xint_gob_til_^ #1\XINT_minof_empty ^%
    \expandafter\XINT_minof_loop\romannumeral0\xintraw{#1}\xint:
}%
\def\XINT_minof_empty ^#1\xint:{ 0/1[0]}%
\def\XINT_minof_loop #1\xint:#2%
{%
    \xint_gob_til_^ #2\XINT_minof_e ^%
```

```
    \expandafter\XINT_minof_loop\romannumeral0\xintmin{#1}{\romannumeral0\xintraw{#2}}\xint:
}%
\def\XINT_minof_e ^#1\xintmin #2#3\xint:{ #2}%
```

8.60 \xintCmp

```
\def\xintCmp {\romannumeral0\xintcmp }%
\def\xintcmp #1%
{%
    \expandafter\XINT_fcmp\expandafter {\romannumeral0\xintraw {#1}}%
}%
\def\XINT_fcmp #1#2%
{%
    \expandafter\XINT_fcmp_A\romannumeral0\xintraw {#2}#1%
}%
\def\XINT_fcmp_A #1#2/#3[#4]#5#6/#7[#8]%
{%
    \xint_UDsignsfork
      #1#5\XINT_fcmp_minusminus
       -#5\XINT_fcmp_firstneg
       #1-\XINT_fcmp_secondneg
        --\XINT_fcmp_nonneg_a
    \krof
    #1#5{#2/#3[#4]}{#6/#7[#8]}%
}%
\def\XINT_fcmp_minusminus --#1#2{\XINT_fcmp_B #2#1}%
\def\XINT_fcmp_firstneg #1-#2#3{ -1}%
\def\XINT_fcmp_secondneg -#1#2#3{ 1}%
\def\XINT_fcmp_nonneg_a #1#2%
{%
    \xint_UDzerosfork
      #1#2\XINT_fcmp_zerozero
       0#2\XINT_fcmp_firstzero
       #10\XINT_fcmp_secondzero
        00\XINT_fcmp_pos
    \krof
    #1#2%
}%
\def\XINT_fcmp_zerozero   #1#2#3#4{ 0}%
\def\XINT_fcmp_firstzero  #1#2#3#4{ -1}%
\def\XINT_fcmp_secondzero #1#2#3#4{ 1}%
\def\XINT_fcmp_pos #1#2#3#4%
{%
    \XINT_fcmp_B #1#3#2#4%
}%
\def\XINT_fcmp_B #1/#2[#3]#4/#5[#6]%
{%
    \expandafter\XINT_fcmp_C\expandafter
    {\the\numexpr #6-#3\expandafter}\expandafter
    {\romannumeral0\xintiimul {#4}{#2}}%
    {\romannumeral0\xintiimul {#5}{#1}}%
}%
\def\XINT_fcmp_C #1#2#3%
```

```
{%
    \expandafter\XINT_fcmp_D\expandafter
    {#3}{#1}{#2}%
}%
\def\XINT_fcmp_D #1#2#3%
{%
    \expandafter\XINT_cntSgnFork\romannumeral`&&@\expandafter\XINT_cntSgn
    \the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax\xint:
    { -1}{\XINT_fcmp_E #2\Z {#3}{#1}}{ 1}%
}%
\def\XINT_fcmp_E #1%
{%
    \xint_UDsignfork
        #1\XINT_fcmp_Fd
         -{\XINT_fcmp_Fn #1}%
    \krof
}%

\def\XINT_fcmp_Fd #1\Z #2#3%
{%
    \expandafter\XINT_fcmp_Fe
    \romannumeral0\XINT_dsx_addzeros {#1}#3;\xint:#2\xint:
}%
\def\XINT_fcmp_Fe #1\xint:#2#3\xint:{\XINT_cmp_plusplus #2#1\xint:#3\xint:}%
\def\XINT_fcmp_Fn #1\Z #2#3%
{%
    \expandafter\XINT_fcmp_Fo
    \romannumeral0\XINT_dsx_addzeros {#1}#2;\xint:#3\xint:
}%
\def\XINT_fcmp_Fo #1#2\xint:#3\xint:{\XINT_cmp_plusplus #1#3\xint:#2\xint:}%
```

8.61 \xintAbs

```
\def\xintAbs    {\romannumeral0\xintabs }%
\def\xintabs #1{\expandafter\XINT_abs\romannumeral0\xintraw {#1}}%
```

8.62 \xintOpp

```
\def\xintOpp    {\romannumeral0\xintopp }%
\def\xintopp #1{\expandafter\XINT_opp\romannumeral0\xintraw {#1}}%
```

8.63 \xintInv

1.3d.

```
\def\xintInv    {\romannumeral0\xintinv }%
\def\xintinv #1{\expandafter\XINT_inv\romannumeral0\xintraw {#1}}%
\def\XINT_inv #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_inv_iszero
      0#1\XINT_inv_a
       0-{\XINT_inv_a {}}%
    \krof #1%
}%
\def\XINT_inv_iszero #1]%
```

```
    {\XINT_signalcondition{DivisionByZero}{Division of 1 by zero (#1)}{}{0/1[0]}}%
\def\XINT_inv_a #1#2/#3[#4#5]%
{%
    \xint_UDzerominusfork
      #4-\XINT_inv_expiszero
      0#4\XINT_inv_b
       0-{\XINT_inv_b -#4}%
    \krof #5.{#1#3/#2}%
}%
\def\XINT_inv_expiszero #1.#2{ #2[0]}%
\def\XINT_inv_b #1.#2{ #2[#1]}%
```

8.64 \xintSgn

```
\def\xintSgn   {\romannumeral0\xintsgn }%
\def\xintsgn #1{\expandafter\XINT_sgn\romannumeral0\xintraw {#1}\xint:}%
```

8.65 \xintGCD

1.4. They replace the former xintgcd macros of the same names which truncated to integers their arguments. Fraction-producing gcd() and lcm() functions were available since 1.3d xintexpr, with non-public support macros handling comma separated values.

1.4d. Somewhat strangely \xintGCD was formerly \xintGCDof used with only two arguments, as the latter directly implemented a fractionl gcd algorithm using \xintMod repeatedly for two arguments.

Now \xintGCD contains the pairwise gcd routine and \xintGCDof is only a wrapper. And the pairwise gcd is reduced to integer-only computations to hopefully reduce fraction overhead.

Each input is filtered via \xintPIrr and \xintREZ to reduce size of maniuplate integers in algebra.

But hesitation about applying \xintPIrr to output, and/or \xintREZ. (as it is applied on input).

But as the code is now used for frational lcm's we actually need to do some reduction of output else lcm's of integers will not be necessarily printed by \xinteval as integers.

Well finally I apply \xintIrr (but not \xintREZ to output). Hesitations here (thinking of inputs with large [n] parts, the output will have many zeros). So I do this only for the user macro but the core routine as used by \xintGCDof will not do it.

Also at 1.4d the code uses \expanded.

```
\def\xintGCD {\romannumeral0\xintgcd}%
\def\xintgcd #1%
{%
    \expandafter\XINT_fgcd_in
    \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:
}%
\def\XINT_fgcd_in #1#2\xint:#3%
{%
    \expandafter\XINT_fgcd_out
    \romannumeral0\expandafter\XINT_fgcd_chkzeros\expandafter#1%
    \romannumeral0\xintrez{\xintPIrr{\xintAbs{#3}}}\xint:#1#2\xint:
}%
\def\XINT_fgcd_out#1[#2]{\xintirr{#1[#2]}[0]}%
\def\XINT_fgcd_chkzeros #1#2%
{%
    \xint_UDzerofork
       #1\XINT_fgcd_aiszero
```

```
        #2\XINT_fgcd_biszero
         0\XINT_fgcd_main
    \krof #2%
}%
\def\XINT_fgcd_aiszero #1\xint:#2\xint:{ #1}%
\def\XINT_fgcd_biszero #1\xint:#2\xint:{ #2}%
\def\XINT_fgcd_main #1/#2[#3]\xint:#4/#5[#6]\xint:
{%
    \expandafter\XINT_fgcd_a
      \romannumeral0\XINT_gcd_loop #2\xint:#5\xint:\xint:
      #2\xint:#5\xint:#1\xint:#4\xint:#3.#6.%
}%
\def\XINT_fgcd_a #1\xint:#2\xint:
{%
    \expandafter\XINT_fgcd_b
    \romannumeral0\xintiiquo{#2}{#1}\xint:#1\xint:#2\xint:
}%
\def\XINT_fgcd_b #1\xint:#2\xint:#3\xint:#4\xint:#5\xint:#6\xint:#7.#8.%
{%
   \expanded{%
   \xintiigcd{\xintiiE{\xintiiMul{#5}{\xintiiQuo{#4}{#2}}}{#7-#8}}%
             {\xintiiE{\xintiiMul{#6}{#1}}{#8-#7}}%
    /\xintiiMul{#1}{#4}%
    [\ifnum#7>#8 #8\else #7\fi]%
   }%
}%
```

8.66 \xintGCDof

1.4. This inherits from former non public xintexpr macro called \xintGCDof:csv, which handled comma separated items.

It handles fractions presented as braced items and is the support macro for the gcd() function in \xintexpr and \xintfloatexpr. The support macro for the gcd() function in \xintiiexpr is \xintiiGCDof, from xint.

An empty input is allowed but I have some hesitations on the return value of 1.

1.4d. Sadly the 1.4 version had multiple problems:

- broken if first argument vanished,
- broken if some argument was not in strict format, for example had leading chains of signs or zeros (\xintGCDof{2}{03}). This bug originates in the fact the original macro was used only in xintexpr sanitized context.

Also, output is now always an irreducible fraction (ending with [0]).

```
\def\xintGCDof {\romannumeral0\xintgcdof}%
\def\xintgcdof #1{\expandafter\XINT_fgcdof\romannumeral`&&@#1^}%
\def\XINT_GCDof{\romannumeral0\XINT_fgcdof}%
\def\XINT_fgcdof #1%
{%
    \expandafter\XINT_fgcdof_chkempty\romannumeral`&&@#1\xint:
}%
\def\XINT_fgcdof_chkempty #1%
{%
```

```
    \xint_gob_til_^#1\XINT_fgcdof_empty ^\XINT_fgcdof_in #1%
}%
\def\XINT_fgcdof_empty #1\xint:{ 1/1[0]}% hesitation, should it be infinity? 0?
\def\XINT_fgcdof_in #1\xint:
{%
    \expandafter\XINT_fgcd_out
    \romannumeral0\expandafter\XINT_fgcdof_loop
    \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:
}%
\def\XINT_fgcdof_loop #1\xint:#2%
{%
    \expandafter\XINT_fgcdof_chkend\romannumeral`&&@#2\xint:#1\xint:\xint:
}%
\def\XINT_fgcdof_chkend #1%
{%
    \xint_gob_til_^#1\XINT_fgcdof_end ^\XINT_fgcdof_loop_pair #1%
}%
\def\XINT_fgcdof_end #1\xint:#2\xint:\xint:{ #2}%
\def\XINT_fgcdof_loop_pair #1\xint:#2%
{%
    \expandafter\XINT_fgcdof_loop
    \romannumeral0\expandafter\XINT_fgcd_chkzeros\expandafter#2%
    \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:#2%
}%
```

8.67 \xintLCM

Same comments as for \xintGCD. Entirely redone for 1.4d. Well, actually we can express it in terms of fractional gcd.

```
\def\xintLCM {\romannumeral0\xintlcm}%
\def\xintlcm #1%
{%
    \expandafter\XINT_flcm_in
    \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:
}%
\def\XINT_flcm_in #1#2\xint:#3%
{%
    \expandafter\XINT_fgcd_out
    \romannumeral0\expandafter\XINT_flcm_chkzeros\expandafter#1%
    \romannumeral0\xintrez{\xintPIrr{\xintAbs{#3}}}\xint:#1#2\xint:
}%
\def\XINT_flcm_chkzeros #1#2%
{%
    \xint_UDzerofork
       #1\XINT_flcm_zero
       #2\XINT_flcm_zero
        0\XINT_flcm_main
    \krof #2%
}%
\def\XINT_flcm_zero #1\xint:#2\xint:{ 0/1[0]}%
\def\XINT_flcm_main #1/#2[#3]\xint:#4/#5[#6]\xint:
{%
```

```
    \xintinv
    {%
      \romannumeral0\XINT_fgcd_main #2/#1[-#3]\xint:#5/#4[-#6]\xint:
    }%
}%
```

8.68 \xintLCMof

See comments for \xintGCDof. xint provides the integer only \xintiiLCMof.
  adly, although a public xintfrac macro, it did not (since 1.4) sanitize its arguments like other xintfrac macros.

```
\def\xintLCMof {\romannumeral0\xintlcmof}%
\def\xintlcmof #1{\expandafter\XINT_flcmof\romannumeral`&&@#1^}%
\def\XINT_LCMof{\romannumeral0\XINT_flcmof}%
\def\XINT_flcmof #1%
{%
    \expandafter\XINT_flcmof_chkempty\romannumeral`&&@#1\xint:
}%
\def\XINT_flcmof_chkempty #1%
{%
    \xint_gob_til_^#1\XINT_flcmof_empty ^\XINT_flcmof_in #1%
}%
\def\XINT_flcmof_empty #1\xint:{ 0/1[0]}% hesitation
\def\XINT_flcmof_in #1\xint:
{%
    \expandafter\XINT_fgcd_out
    \romannumeral0\expandafter\XINT_flcmof_loop
    \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:
}%
\def\XINT_flcmof_loop #1\xint:#2%
{%
    \expandafter\XINT_flcmof_chkend\romannumeral`&&@#2\xint:#1\xint:\xint:
}%
\def\XINT_flcmof_chkend #1%
{%
    \xint_gob_til_^#1\XINT_flcmof_end ^\XINT_flcmof_loop_pair #1%
}%
\def\XINT_flcmof_end #1\xint:#2\xint:\xint:{ #2}%
\def\XINT_flcmof_loop_pair #1\xint:#2%
{%
    \expandafter\XINT_flcmof_chkzero
    \romannumeral0\expandafter\XINT_flcm_chkzeros\expandafter#2%
    \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:#2%
}%
\def\XINT_flcmof_chkzero #1%
{%
    \xint_gob_til_zero#1\XINT_flcmof_zero0\XINT_flcmof_loop#1%
}%
\def\XINT_flcmof_zero#1^{ 0/1[0]}%
```

8.69 Floating point macros

For a long time the float routines dating back to releases 1.07/1.08a (May-June 2013) were not modified.

Since 1.2f (March 2016) the four operations first round their arguments to \xinttheDigits-floats (or P-floats), not (\xinttheDigits+2)-floats or (P+2)-floats as was the case with earlier releases.

The four operations addition, subtraction, multiplication, division have always produced the correct rounding of the theoretical exact value to P or \xinttheDigits digits when the inputs are decimal numbers with at most P digits, and arbitrary decimal exponent part.

From 1.08a to 1.2j, \xintFloat (and \XINTinFloat which is used to parse inputs to other float macros) handled a fractional input A/B via an initial replacement to A'/B' where A' and B' were A and B truncated to Q+2 digits (where asked-for precision is Q), and then they correctly rounded A'/B' to Q digits. But this meant that this rounding of the input could differ (by up to one unit in the last place) from the correct rounding of the original A/B to the asked-for number of digits (which until 1.2f in uses as auxiliary to the macros for the basic operations was 2 more than the prevailing precision).

Since 1.2k all inputs are correctly rounded to the asked-for number of digits (this was, I think, the case in the 1.07 release -- there are no code comments -- but was, afaicr, not very efficiently done, and this is why the 1.08a release opeted for truncation of the numerator and denominator.)

Notice that in float expressions, the / is treated as operator, hence the above discussion makes a difference only for the special input form qfloat(A/B) or for an \xintexpr A/B\relax embedded in the float expression, with A or B having more digits than the prevailing float precision.

> Internally there is no inner representation of P-floats as such !!!!!
>
> The input parser will again compute the length of the mantissa on each use !!! This is obviously something that must be improved upon before implementation of higher functions.
>
> Currently, special tricks are used to quickly recognize inputs having no denominators, or fractions whose numerators and denominators are not too long compared to the target precision P, and in particular P-floats or quotients of two such.
>
> Another long-standing issue is that float multiplication will first compute the 2P or 2P-1 digits of the exact product, and then round it to P digits. This is sub-optimal for large P particularly as the multiplication algorithm is basically the schoolbook one, hence *worse* than quadratic in the TEX implementation which has extra cost of fetching long sequences of tokens.

8.70 \xintDigits, \xintSetDigits

1.3f modifies the (strange) original signature #1#2 for \xintDigits macro into #1=, allowing usage without colon. It also adds \xintSetDigits. Starred variants are added by xintexpr.sty.

```
\mathchardef\XINTdigits 16
\def\xintDigits #1=%
   {\afterassignment \xint_gobble_i \mathchardef\XINTdigits=}%
\def\xinttheDigits {\number\XINTdigits }%
\def\xintSetDigits #1{\mathchardef\XINTdigits=\numexpr#1\relax}%
```

8.71 \xintFloat

1.2f and 1.2g brought some refactoring which resulted in faster treatment of decimal inputs. 1.2i dropped use of some old routines dating back to pre 1.2 era in favor of more modern \xintDSRr for rounding. Then 1.2k improves again the handling of denominators B with few digits.

But the main change with 1.2k is a complete rewrite of the B>1 case in order to achieve again correct rounding in all cases.

The original version from 1.07 (May 2013) computed the exact rounding to P digits for all inputs. But from 1.08 on (June 2013), the macro handled A/B input by first truncating both A and B to at most P+2 digits. This meant that decimal input (arbitrarily long, with scientific part) was correctly rounded, but in case of fractional input there could be up to 0.6 unit in the last place difference of the produced rounding to the input, hence the output could differ from the correct rounding.

Example with 16 digits (the default): \xintFloat {1/17597472569900621233}
with xintfrac 1.07: 5.682634230727187e-20
with xintfrac 1.08b--1.2j: 5.682634230727188e-20
with xintfrac 1.2k: 5.682634230727187e-20
The exact value is 5.68263423072718749992412 4...e-20, showing that 1.07 and 1.2k produce the correct rounding.

Currently the code ends in a more costly branch in about 1 case among 500, where it does some extra operations (a multiplication in particular). There is a free parameter delta (here set at 4), I have yet to make some numerical explorations, to see if it could be favorable to set it to a higher value (with delta=5, there is only 1 exceptional case in 5000, etc...).

I have always hesitated about the policy of printing 10.00...0 in case of rounding upwards to the next power of ten. Already since 1.2f \XINTinFloat always produced a mantissa with exactly P digits (except for the zero value). Starting with 1.2k, \xintFloat drops this habit of printing 10.00..0 in such cases. Side note: the rounding-up detection worked when the input A/B was with numerator A and denominator B having each less than P+2 digits, or with B=1, else, it could happen that the output was a power of ten but not detected to be a rounding up of the original fraction. The value was ok, but printed 1.0...0eN with P-1 zeroes, not 10.0...0e(N-1).

I decided it was not worth the effort to enhance the algorithm to detect with 100% fiability all cases of rounding up to next power of ten, hence 1.2k dropped this.

To avoid duplication of code, and any extra burden on \XINTinFloat, which is the macro used internally by the float macros for parsing their inputs, we simply make now \xintFloat a wrapper of \XINTinFloat.

```
\def\xintFloat   {\romannumeral0\xintfloat }%
\def\xintfloat #1{\XINT_float_chkopt #1\xint:}%
\def\XINT_float_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_float_opt
       \else\expandafter\XINT_float_noopt
    \fi  #1%
}%
\def\XINT_float_noopt #1\xint:%
{%
    \expandafter\XINT_float_post
    \romannumeral0\XINTinfloat[\XINTdigits]{#1}\XINTdigits.%
}%
\def\XINT_float_opt [\xint:#1]%
{%
    \expandafter\XINT_float_opt_a\the\numexpr #1.%
}%
\def\XINT_float_opt_a #1.#2%
{%
    \expandafter\XINT_float_post
    \romannumeral0\XINTinfloat[#1]{#2}#1.%
}%
\def\XINT_float_post #1%
```

```
{%
    \xint_UDzerominusfork
      #1-\XINT_float_zero
       0#1\XINT_float_neg
       0-\XINT_float_pos
    \krof #1%
}%[
\def\XINT_float_zero #1]#2.{ 0.e0}%
\def\XINT_float_neg-{\expandafter-\romannumeral0\XINT_float_pos}%
\def\XINT_float_pos #1#2[#3]#4.%
{%
    \expandafter\XINT_float_pos_done\the\numexpr#3+#4-\xint_c_i.#1.#2;%
}%
\def\XINT_float_pos_done #1.#2;{ #2e#1}%
```

8.72 \XINTinFloat, \XINTinFloatS, \XINTiLogTen

This routine is like \xintFloat but produces an output of the shape A[N] which is then parsed faster as input to other float macros. Float operations in \xintfloatexpr...\relax use internally this format.

It must be used in form \XINTinFloat[P]{f}: the optional [P] is mandatory.

Since 1.2f, the mantissa always has exactly P digits even in case of rounding up to next power of ten. This simplifies other routines.

1.2g added a variant \XINTinFloatS which, in case of decimal input with less than the asked for precision P will not add extra zeros to the mantissa. For example it may output 2[0] even if P=500, rather than the canonical representation 200...000[-499]. This is how \xintFloatMul and \xintFloatDiv parse their inputs, which speeds-up follow-up processing. But \xintFloatAdd and \xintFloatSub still use \XINTinFloat for parsing their inputs; anyway this will have to be changed again when inner structure will carry upfront at least the length of mantissa as data.

Each time \XINTinFloat is called it at least computes a length. Naturally if we had some format for floats that would be dispensed of...
something like <letterP><length of mantissa>.mantissa.exponent, etc... not yet.

Since 1.2k, \XINTinFloat always correctly rounds its argument, even if it is a fraction with very big numerator and denominator. See the discussion of \xintFloat.

1.3e adds \XINTiLogTen.

```
\def\XINTinFloat {\romannumeral0\XINTinfloat }%
\def\XINTinfloat
   {\expandafter\XINT_infloat_clean\romannumeral0\XINT_infloat}%
```

Attention que ici le fait que l'on grabbe #1 est important car il pourrait y avoir un zéro (en particulier dans le cas où input est nul).

```
\def\XINT_infloat_clean #1%
   {\if #1!\xint_dothis\XINT_infloat_clean_a\fi\xint_orthat{ }#1}%
```

Ici on ajoute les zeros pour faire exactement avec P chiffres. Car le #1 = P - L avec L la longueur de #2, (ou de abs(#2), ici le #2 peut avoir un signe) qui est < P

```
\def\XINT_infloat_clean_a !#1.#2[#3]%
{%
    \expandafter\XINT_infloat_done
    \the\numexpr #3-#1\expandafter.%
    \romannumeral0\XINT_dsx_addzeros {#1}#2;;%
```

```
}%
\def\XINT_infloat_done #1.#2;{ #2[#1]}%
```

variant which allows output with shorter mantissas.

```
\def\XINTinFloatS {\romannumeral0\XINTinfloatS}%
\def\XINTinfloatS
   {\expandafter\XINT_infloatS_clean\romannumeral0\XINT_infloat}%
\def\XINT_infloatS_clean #1%
   {\if #1!\xint_dothis\XINT_infloatS_clean_a\fi\xint_orthat{ }#1}%
\def\XINT_infloatS_clean_a !#1.{ }%
```

1.3e ajoute \XINTiLogTen. Le comportement pour un input nul est non encore finalisé. Il changera lorsque NaN, +Inf, -Inf existeront.

```
\def\XINTFloatiLogTen {\the\numexpr\XINTfloatilogten}%
\def\XINTfloatilogten [#1]#2%
    {\expandafter\XINT_floatilogten\romannumeral0\XINT_infloat[#1]{#2}#1.}%
\def\XINT_floatilogten #1{%
    \if #10\xint_dothis\XINT_floatilogten_z\fi
    \if #1!\xint_dothis\XINT_floatilogten_a\fi
    \xint_orthat\XINT_floatilogten_b #1%
}%
\def\XINT_floatilogten_z 0[0]#1.{-"7FFF8000\relax}%
\def\XINT_floatilogten_a !#1.#2[#3]#4.{#3-#1+#4-1\relax}%
\def\XINT_floatilogten_b #1[#2]#3.{#2+#3-1\relax}%
```

début de la routine proprement dite, l'argument optionnel est obligatoire.

```
\def\XINT_infloat [#1]#2%
{%
    \expandafter\XINT_infloat_a\the\numexpr #1\expandafter.%
    \romannumeral0\XINT_infrac {#2}%
}%
```

#1=P, #2=n, #3=A, #4=B.

```
\def\XINT_infloat_a #1.#2#3#4%
{%
```

micro boost au lieu d'utiliser \XINT_isOne{#4}, mais pas bon style.

```
    \if1\XINT_is_One#4XY%
      \expandafter\XINT_infloat_sp
    \else\expandafter\XINT_infloat_fork
    \fi #3.{#1}{#2}{#4}%
}%
```

Special quick treatment of B=1 case (1.2f then again 1.2g.)
maintenant: A.{P}{N}{1} Il est possible que A soit nul.

```
\def\XINT_infloat_sp #1%
{%
    \xint_UDzerominusfork
     #1-\XINT_infloat_spzero
     0#1\XINT_infloat_spneg
```

```
2160       0-\XINT_infloat_sppos
2161     \krof #1%
2162 }%
```

Attention surtout pas 0/1[0] ici.

```
2163 \def\XINT_infloat_spzero 0.#1#2#3{ 0[0]}%
2164 \def\XINT_infloat_spneg-%
2165     {\expandafter\XINT_infloat_spnegend\romannumeral0\XINT_infloat_sppos}%
2166 \def\XINT_infloat_spnegend #1%
2167     {\if#1!\expandafter\XINT_infloat_spneg_needzeros\fi -#1}%
2168 \def\XINT_infloat_spneg_needzeros -!#1.{!#1.-}%
```

in: A.{P}{N}{1}
out: P-L.A.P.N.

```
2169 \def\XINT_infloat_sppos #1.#2#3#4%
2170 {%
2171     \expandafter\XINT_infloat_sp_b\the\numexpr#2-\xintLength{#1}.#1.#2.#3.%
2172 }%
```

#1= P-L. Si c'est positif ou nul il faut retrancher #1 à l'exposant, et ajouter autant de zéros. On regarde premier token. P-L.A.P.N.

```
2173 \def\XINT_infloat_sp_b #1%
2174 {%
2175     \xint_UDzerominusfork
2176      #1-\XINT_infloat_sp_quick
2177      0#1\XINT_infloat_sp_c
2178       0-\XINT_infloat_sp_needzeros
2179     \krof #1%
2180 }%
```

Ici P=L. Le cas usuel dans \xintfloatexpr.

```
2181 \def\XINT_infloat_sp_quick 0.#1.#2.#3.{ #1[#3]}%
```

Ici #1=P-L est >0. L'exposant sera N-(P-L). #2=A. #3=P. #4=N.
18 mars 2016. En fait dans certains contextes il est sous-optimal d'ajouter les zéros. Par exemple quand c'est appelé par la multiplication ou la division, c'est idiot de convertir 2 en 200000...00000[-499]. Donc je redéfinis addzeros en needzeroes. Si on appelle sous la forme \XINTinFloatS, on ne fait pas l'addition de zeros.

```
2182 \def\XINT_infloat_sp_needzeros #1.#2.#3.#4.{!#1.#2[#4]}%
```

L-P=#1.A=#2#3.P=#4.N=#5.
Ici P<L. Il va falloir arrondir. Attention si on va à la puissance de 10 suivante. En #1 on a L-P qui est >0. L'exposant final sera N+L-P, sauf dans le cas spécial, il sera alors N+L-P+1. L'ajustement final est fait par \XINT_infloat_Y.

```
2183 \def\XINT_infloat_sp_c -#1.#2#3.#4.#5.%
2184 {%
2185     \expandafter\XINT_infloat_Y
2186     \the\numexpr #5+#1\expandafter.%
2187     \romannumeral0\expandafter\XINT_infloat_sp_round
2188     \romannumeral0\XINT_split_fromleft
```

```
    (\xint_c_i+#4).#2#3\xint_bye2345678\xint_bye..#2%
}%
\def\XINT_infloat_sp_round #1.#2.%
{%
    \XINT_dsrr#1\xint_bye\xint_Bye3456789\xint_bye/\xint_c_x\relax.%
}%
```

General branch for A/B with B>1 inputs. It achieves correct rounding always since 1.2k (done January 2, 2017.) This branch is never taken for A=0 because \XINT_infrac will have returned B=1 then.

```
\def\XINT_infloat_fork #1%
{%
    \xint_UDsignfork
     #1\XINT_infloat_J
     -\XINT_infloat_K
    \krof #1%
}%
\def\XINT_infloat_J-{\expandafter-\romannumeral0\XINT_infloat_K }%
```

A.{P}{n}{B} avec B>1.

```
\def\XINT_infloat_K #1.#2%
{%
    \expandafter\XINT_infloat_L
    \the\numexpr\xintLength{#1}\expandafter.\the\numexpr #2+\xint_c_iv.{#1}{#2}%
}%
```

|A|.P+4.{A}{P}{n}{B}. We check if A already has length <= P+4.

```
\def\XINT_infloat_L #1.#2.%
{%
    \ifnum #1>#2
      \expandafter\XINT_infloat_Ma
    \else
      \expandafter\XINT_infloat_Mb
    \fi #1.#2.%
}%
```

|A|.P+4.{A}{P}{n}{B}. We will keep only the first P+4 digits of A, denoted A'' in what follows.
output: u=-0.A''.junk.P+4.|A|.{A}{P}{n}{B}

```
\def\XINT_infloat_Ma #1.#2.#3%
{%
    \expandafter\XINT_infloat_MtoN\expandafter-\expandafter0\expandafter.%
    \romannumeral0\XINT_split_fromleft#2.#3\xint_bye2345678\xint_bye..%
    #2.#1.{#3}%
}%
```

|A|.P+4.{A}{P}{n}{B}.
Here A is short. We set u = P+4-|A|, and A''=A (A' = 10^u A)
output: u.A''..P+4.|A|.{A}{P}{n}{B}

```
\def\XINT_infloat_Mb #1.#2.#3%
{%
```

```
2224     \expandafter\XINT_infloat_MtoN\the\numexpr#2-#1.%
2225     #3..#2.#1.{#3}%
2226 }%
```

```
   input u.A''.junk.P+4.|A|.{A}{P}{n}{B}
 output |B|.P+4.{B}u.A''.P.|A|.n.{A}{B}
```

```
2227 \def\XINT_infloat_MtoN #1.#2.#3.#4.#5.#6#7#8#9%
2228 {%
2229    \expandafter\XINT_infloat_N
2230    \the\numexpr\xintLength{#9}.#4.{#9}#1.#2.#7.#5.#8.{#6}{#9}%
2231 }%
2232 \def\XINT_infloat_N #1.#2.%
2233 {%
2234     \ifnum #1>#2
2235       \expandafter\XINT_infloat_Oa
2236     \else
2237       \expandafter\XINT_infloat_Ob
2238     \fi #1.#2.%
2239 }%
```

```
   input |B|.P+4.{B}u.A''.P.|A|.n.{A}{B}
 output v=-0.B''.junk.|B|.u.A''.P.|A|.n.{A}{B}
```

```
2240 \def\XINT_infloat_Oa #1.#2.#3%
2241 {%
2242     \expandafter\XINT_infloat_P\expandafter-\expandafter0\expandafter.%
2243     \romannumeral0\XINT_split_fromleft#2.#3\xint_bye2345678\xint_bye..%
2244     #1.%
2245 }%
```

```
   output v=P+4-|B|>=0.B''.junk.|B|.u.A''.P.|A|.n.{A}{B}
```

```
2246 \def\XINT_infloat_Ob #1.#2.#3%
2247 {%
2248     \expandafter\XINT_infloat_P\the\numexpr#2-#1.#3..#1.%
2249 }%
```

```
   input v.B''.junk.|B|.u.A''.P.|A|.n.{A}{B}
 output Q1.P.|B|.|A|.n.{A}{B}
 Q1 = division euclidienne de A''.10^{u-v+P+3} par B''.
   Special detection of cases with A and B both having length at most P+4: this will happen when
 called from \xintFloatDiv as A and B (produced then via \XINTinFloatS) will have at most P digits.
 We then only need integer division with P+1 extra zeros, not P+3.
```

```
2250 \def\XINT_infloat_P #1#2.#3.#4.#5.#6#7.#8.#9.%
2251 {%
2252    \csname XINT_infloat_Q\if-#1\else\if-#6\else q\fi\fi\expandafter\endcsname
2253    \romannumeral0\xintiiquo
2254    {\romannumeral0\XINT_dsx_addzerosnofuss
2255       {#6#7-#1#2+#9+\xint_c_iii\if-#1\else\if-#6\else-\xint_c_ii\fi\fi}#8;}%
2256    {#3}.#9.#5.%
2257 }%
```

```
    «quick» branch.

\def\XINT_infloat_Qq #1.#2.%
{%
    \expandafter\XINT_infloat_Rq
    \romannumeral0\XINT_split_fromleft#2.#1\xint_bye2345678\xint_bye..#2.%
}%
\def\XINT_infloat_Rq #1.#2#3.%
{%
    \ifnum#2<\xint_c_v
         \expandafter\XINT_infloat_SEq
    \else\expandafter\XINT_infloat_SUp
    \fi
    {\if.#3.\xint_c_\else\xint_c_i\fi}#1.%
}%

    standard branch which will have to handle undecided rounding, if too close to a mid-value.

\def\XINT_infloat_Q #1.#2.%
{%
    \expandafter\XINT_infloat_R
    \romannumeral0\XINT_split_fromleft#2.#1\xint_bye2345678\xint_bye..#2.%
}%
\def\XINT_infloat_R #1.#2#3#4#5.%
{%
    \if.#5.\expandafter\XINT_infloat_Sa\else\expandafter\XINT_infloat_Sb\fi
    #2#3#4#5.#1.%
}%

    trailing digits.Q.P.|B|.|A|.n.{A}{B}
  #1=trailing digits (they may have leading zeros.)

\def\XINT_infloat_Sa #1.%
{%
    \ifnum#1>500 \xint_dothis\XINT_infloat_SUp\fi
    \ifnum#1<499 \xint_dothis\XINT_infloat_SEq\fi
    \xint_orthat\XINT_infloat_X\xint_c_
}%
\def\XINT_infloat_Sb #1.%
{%
    \ifnum#1>5009 \xint_dothis\XINT_infloat_SUp\fi
    \ifnum#1<4990 \xint_dothis\XINT_infloat_SEq\fi
    \xint_orthat\XINT_infloat_X\xint_c_i
}%

    epsilon #2=Q.#3=P.#4=|B|.#5=|A|.#6=n.{A}{B}
  exposant final est n+|A|-|B|-P+epsilon

\def\XINT_infloat_SEq #1#2.#3.#4.#5.#6.#7#8%
{%
    \expandafter\XINT_infloat_SY
    \the\numexpr #6+#5-#4-#3+#1.#2.%
}%
\def\XINT_infloat_SY #1.#2.{ #2[#1]}%
```

initial digit #2 put aside to check for case of rounding up to next power of ten, which will need adjustment of mantissa and exponent.

```
\def\XINT_infloat_SUp #1#2#3.#4.#5.#6.#7.#8#9%
{%
    \expandafter\XINT_infloat_Y
    \the\numexpr#7+#6-#5-#4+#1\expandafter.%
    \romannumeral0\xintinc{#2#3}.#2%
}%
```

epsilon Q.P.|B|.|A|.n.{A}{B}

\xintDSH{-x}{U} multiplies U by 10^x. When x is negative, this means it truncates (i.e. it drops the last -x digits).
We don't try to optimize too much macro calls here, the odds are 2 per 1000 for this branch to be taken. Perhaps in future I will use higher free parameter d, which currently is set at 4.
#1=epsilon, #2#3=Q, #4=P, #5=|B|, #6=|A|, #7=n, #8=A, #9=B

```
\def\XINT_infloat_X #1#2#3.#4.#5.#6.#7.#8#9%
{%
   \expandafter\XINT_infloat_Y
   \the\numexpr #7+#6-#5-#4+#1\expandafter.%
   \romannumeral`&&@\romannumeral0\xintiiiflt
     {\xintDSH{#6-#5-#4+#1}{\xintDouble{#8}}}%
     {\xintiiMul{\xintInc{\xintDouble{#2#3}}}{#9}}%
   \xint_firstofone
   \xintinc{#2#3}.#2%
}%
```

check for rounding up to next power of ten.

```
\def\XINT_infloat_Y #1{%
\def\XINT_infloat_Y ##1.##2##3.##4%
{%
   \if##49\if##21\expandafter\expandafter\expandafter\XINT_infloat_Z\fi\fi
   #1##2##3[##1]%
}}\XINT_infloat_Y{ }%
```

#1=1, #2=0.

```
\def\XINT_infloat_Z #1#2#3[#4]%
{%
    \expandafter\XINT_infloat_ZZ\the\numexpr#4+\xint_c_i.#3.%
}%
\def\XINT_infloat_ZZ #1.#2.{ 1#2[#1]}%
```

8.73 \xintPFloat, \xintPFloatE

1.1. This is a prettifying printing macro for floats.
The macro applies one simple rule: x.yz...eN will drop scientific notation in favor of pure decimal notation if -5<=N<=5. This is the default behaviour of Maple. The N here is as produced on output by \xintFloat.
Special case: the zero value is printed 0. (with a dot)

The coding got simpler with 1.2k as its \xintFloat always produces a mantissa with exactly P digits (no more 10.0...0eN annoying exception).

1.4b adds \xintPFloatE allowing to customize whether to use e or E (or something else). For usage with \xintfloateval{} (anyhow only catcode 11 e is recognized by xintfrac macros proper), and to match similar \xintFracToSciE. For reasons commented upon in user manual (section «The three parsers»), I did not make an effort to let the macro be usable as a hook to grab the exponent.

Althout \xintfloateval{} will use \xintPFloat in an \expanded context we have to maintain f-expandability here.

```
\def\xintPFloat   {\romannumeral0\xintpfloat }%
\def\xintpfloat #1{\XINT_pfloat_chkopt #1\xint:}%
\def\XINT_pfloat_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_pfloat_opt
       \else\expandafter\XINT_pfloat_noopt
    \fi  #1%
}%
\def\XINT_pfloat_noopt #1\xint:%
{%
    \expandafter\XINT_pfloat_a
    \romannumeral0\xintfloat [\XINTdigits]{#1};\XINTdigits.%
}%


\def\XINT_pfloat_opt [\xint:#1]%
{%
    \expandafter\XINT_pfloat_opt_a \the\numexpr #1.%
}%
\def\XINT_pfloat_opt_a #1.#2%
{%
    \expandafter\XINT_pfloat_a\romannumeral0\xintfloat [#1]{#2};#1.%
}%
\def\XINT_pfloat_a #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_pfloat_zero
        0#1\XINT_pfloat_neg
        0-\XINT_pfloat_pos
    \krof #1%
}%


\def\XINT_pfloat_zero #1;#2.{ 0.}%
\def\XINT_pfloat_neg-{\expandafter-\romannumeral0\XINT_pfloat_pos }%
```

1.4b modifies the replacement pattern here #1{#2}{#3} in order to facilitate injection of once-expanded \xintPFloatE.

```
\def\XINT_pfloat_pos #1.#2e#3;#4.%
{%
    \ifnum #3>\xint_c_v  \xint_dothis\XINT_pfloat_no\fi
    \ifnum #3<-\xint_c_v \xint_dothis\XINT_pfloat_no\fi
    \ifnum #3<\xint_c_   \xint_dothis\XINT_pfloat_N\fi
    \ifnum #3>\numexpr #4-\xint_c_i\relax \xint_dothis\XINT_pfloat_Ps\fi
```

```
    \xint_orthat\XINT_pfloat_P #1{#2}{#3}%
}%
\def\XINT_pfloat_no
{%
    \expandafter\XINT_pfloat_no_e\expandafter{\xintPFloatE}%
}%
\def\XINT_pfloat_no_e #1#2#3#4{ #2.#3#1#4}%
\def\xintPFloatE{e}%
```

This is all simpler coded, now that 1.2k's \xintFloat always outputs a mantissa with exactly one digits before decimal mark always.

```
\def\XINT_pfloat_N #1#2#3%
{%
    \csname XINT_pfloat_N_\romannumeral-#3\endcsname #1#2%
}%
\def\XINT_pfloat_N_i  { 0.}%
\def\XINT_pfloat_N_ii { 0.0}%
\def\XINT_pfloat_N_iii{ 0.00}%
\def\XINT_pfloat_N_iv { 0.000}%
\def\XINT_pfloat_N_v  { 0.0000}%


\def\XINT_pfloat_P #1#2#3%
{%
    \csname XINT_pfloat_P_\romannumeral#3\endcsname #1#2%
}%
\def\XINT_pfloat_P_   #1{ #1.}%
\def\XINT_pfloat_P_i  #1#2{ #1#2.}%
\def\XINT_pfloat_P_ii #1#2#3{ #1#2#3.}%
\def\XINT_pfloat_P_iii#1#2#3#4{ #1#2#3#4.}%
\def\XINT_pfloat_P_iv #1#2#3#4#5{ #1#2#3#4#5.}%
\def\XINT_pfloat_P_v  #1#2#3#4#5#6{ #1#2#3#4#5#6.}%


\def\XINT_pfloat_Ps #1#2#3%
{%
    \csname XINT_pfloat_Ps\romannumeral#3\endcsname #1#200000;%
}%
\def\XINT_pfloat_Psi  #1#2#3;{ #1#2.}%
\def\XINT_pfloat_Psii #1#2#3#4;{ #1#2#3.}%
\def\XINT_pfloat_Psiii#1#2#3#4#5;{ #1#2#3#4.}%
\def\XINT_pfloat_Psiv #1#2#3#4#5#6;{ #1#2#3#4#5.}%
\def\XINT_pfloat_Psv  #1#2#3#4#5#6#7;{ #1#2#3#4#5#6.}%
```

8.74 \XINTinFloatFracdigits

1.09i, for frac function in \xintfloatexpr. This version computes exactly from the input the fractional part and then only converts it into a float with the asked-for number of digits. I will have to think it again some day, certainly.

1.1 removes optional argument for which there was anyhow no interface, for technical reasons having to do with \xintNewExpr.

1.1a renames the macro as \XINTinFloatFracdigits (from \XINTinFloatFrac) to be synchronous with the \XINTinFloatSqrt and \XINTinFloat habits related to \xintNewExpr problems.

Note to myself: I still have to rethink the whole thing about what is the best to do, the initial way of going through \xinttfrac was just a first implementation.

```
\def\XINTinFloatFracdigits {\romannumeral0\XINTinfloatfracdigits }%
\def\XINTinfloatfracdigits #1%
{%
    \expandafter\XINT_infloatfracdg_a\expandafter {\romannumeral0\xinttfrac{#1}}%
}%
\def\XINT_infloatfracdg_a {\XINTinfloat [\XINTdigits]}%
```

8.75 \xintFloatAdd, \XINTinFloatAdd

First included in release 1.07.

1.09ka improved a bit the efficiency. However the add, sub, mul, div routines were provisory and supposed to be revised soon.

Which didn't happen until 1.2f. Now, the inputs are first rounded to P digits, not P+2 as earlier.

```
\def\xintFloatAdd      {\romannumeral0\xintfloatadd }%
\def\xintfloatadd    #1{\XINT_fladd_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatAdd    {\romannumeral0\XINTinfloatadd }%
\def\XINTinfloatadd  #1{\XINT_fladd_chkopt \XINTinfloatS #1\xint:}%
\def\XINT_fladd_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_fladd_opt
       \else\expandafter\XINT_fladd_noopt
    \fi  #1#2%
}%
\def\XINT_fladd_noopt #1#2\xint:#3%
{%
    #1[\XINTdigits]%
    {\expandafter\XINT_FL_add_a
     \romannumeral0\XINTinfloat[\XINTdigits]{#2}\XINTdigits.{#3}}%
}%
\def\XINT_fladd_opt #1[\xint:#2]%#3#4%
{%
    \expandafter\XINT_fladd_opt_a\the\numexpr #2.#1%
}%
\def\XINT_fladd_opt_a #1.#2#3#4%
{%
    #2[#1]{\expandafter\XINT_FL_add_a\romannumeral0\XINTinfloat[#1]{#3}#1.{#4}}%
}%
\def\XINT_FL_add_a #1%
{%
    \xint_gob_til_zero #1\XINT_FL_add_zero 0\XINT_FL_add_b #1%
}%


\def\XINT_FL_add_zero #1.#2{#2}%[[
\def\XINT_FL_add_b #1]#2.#3%
{%
    \expandafter\XINT_FL_add_c\romannumeral0\XINTinfloat[#2]{#3}#2.#1]%
}%
```

```
\def\XINT_FL_add_c #1%
{%
    \xint_gob_til_zero #1\XINT_FL_add_zero 0\XINT_FL_add_d #1%
}%

\def\XINT_FL_add_d #1[#2]#3.#4[#5]%
{%
    \ifnum\numexpr #2-#3-#5>\xint_c_\xint_dothis\xint_firstoftwo\fi
    \ifnum\numexpr #5-#3-#2>\xint_c_\xint_dothis\xint_secondoftwo\fi
    \xint_orthat\xintAdd {#1[#2]}{#4[#5]}%
}%
```

8.76 \xintFloatSub, \XINTinFloatSub

First done 1.07.
  Starting with 1.2f the arguments undergo an intial rounding to the target precision P not P+2.

```
\def\xintFloatSub      {\romannumeral0\xintfloatsub }%
\def\xintfloatsub    #1{\XINT_flsub_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatSub    {\romannumeral0\XINTinfloatsub }%
\def\XINTinfloatsub  #1{\XINT_flsub_chkopt \XINTinfloatS #1\xint:}%
\def\XINT_flsub_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flsub_opt
       \else\expandafter\XINT_flsub_noopt
    \fi  #1#2%
}%
\def\XINT_flsub_noopt #1#2\xint:#3%
{%
    #1[\XINTdigits]%
    {\expandafter\XINT_FL_add_a
     \romannumeral0\XINTinfloat[\XINTdigits]{#2}\XINTdigits.{\xintOpp{#3}}}%
}%
\def\XINT_flsub_opt #1[\xint:#2]%#3#4%
{%
    \expandafter\XINT_flsub_opt_a\the\numexpr #2.#1%
}%
\def\XINT_flsub_opt_a #1.#2#3#4%
{%
    #2[#1]{\expandafter\XINT_FL_add_a\romannumeral0\XINTinfloat[#1]{#3}#1.{\xintOpp{#4}}}%
}%
```

8.77 \xintFloatMul, \XINTinFloatMul

1.07.
  Starting with 1.2f the arguments are rounded to the target precision P not P+2.
  1.2g handles the inputs via \XINTinFloatS which will be more efficient when the precision is large and the input is for example a small constant like 2.
  1.2k does a micro improvement to the way the macro passes over control to its output routine (former version used a higher level \xintE causing some extra un-needed processing with two calls to \XINT_infrac where one was amply enough).

```
\def\xintFloatMul   {\romannumeral0\xintfloatmul   }%
\def\xintfloatmul   #1{\XINT_flmul_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatMul {\romannumeral0\XINTinfloatmul }%
\def\XINTinfloatmul #1{\XINT_flmul_chkopt \XINTinfloatS #1\xint:}%
\def\XINT_flmul_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flmul_opt
       \else\expandafter\XINT_flmul_noopt
    \fi  #1#2%
}%
\def\XINT_flmul_noopt #1#2\xint:#3%
{%
    #1[\XINTdigits]%
    {\expandafter\XINT_FL_mul_a
     \romannumeral0\XINTinfloatS[\XINTdigits]{#2}\XINTdigits.{#3}}%
}%
\def\XINT_flmul_opt #1[\xint:#2]%#3#4%
{%
    \expandafter\XINT_flmul_opt_a\the\numexpr #2.#1%
}%
\def\XINT_flmul_opt_a #1.#2#3#4%
{%
    #2[#1]{\expandafter\XINT_FL_mul_a\romannumeral0\XINTinfloatS[#1]{#3}#1.{#4}}%
}%
\def\XINT_FL_mul_a #1[#2]#3.#4%
{%
    \expandafter\XINT_FL_mul_b\romannumeral0\XINTinfloatS[#3]{#4}#1[#2]%
}%


\def\XINT_FL_mul_b #1[#2]#3[#4]{\xintiiMul{#3}{#1}/1[#4+#2]}%
```

8.78 \XINTinFloatInv

Added belatedly at 1.3e, to support inv() function. We use Short output, for rare inv(\xintexpr 1/3\relax) case. I need to think the whole thing out at some later date.

```
\def\XINTinFloatInv#1{\XINTinFloatS[\XINTdigits]{\xintInv{#1}}}%
```

8.79 \xintFloatDiv, \XINTinFloatDiv

1.07.
  Starting with 1.2f the arguments are rounded to the target precision P not P+2.
  1.2g handles the inputs via \XINTinFloatS which will be more efficient when the precision is large and the input is for example a small constant like 2.
  The actual rounding of the quotient is handled via \xintfloat (or \XINTinfloatS).
  1.2k does the same kind of improvement in \XINT_FL_div_b as for multiplication: earlier code was unnecessarily high level.

```
\def\xintFloatDiv   {\romannumeral0\xintfloatdiv   }%
\def\xintfloatdiv   #1{\XINT_fldiv_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatDiv {\romannumeral0\XINTinfloatdiv }%
\def\XINTinfloatdiv #1{\XINT_fldiv_chkopt \XINTinfloatS #1\xint:}%
```

```
\def\XINT_fldiv_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_fldiv_opt
       \else\expandafter\XINT_fldiv_noopt
    \fi  #1#2%
}%

\def\XINT_fldiv_noopt #1#2\xint:#3%
{%
    #1[\XINTdigits]%
    {\expandafter\XINT_FL_div_a
     \romannumeral0\XINTinfloatS[\XINTdigits]{#3}\XINTdigits.{#2}}%
}%
\def\XINT_fldiv_opt #1[\xint:#2]%#3#4%
{%
    \expandafter\XINT_fldiv_opt_a\the\numexpr #2.#1%
}%

\def\XINT_fldiv_opt_a #1.#2#3#4%
{%
    #2[#1]{\expandafter\XINT_FL_div_a\romannumeral0\XINTinfloatS[#1]{#4}#1.{#3}}%
}%
\def\XINT_FL_div_a #1[#2]#3.#4%
{%
    \expandafter\XINT_FL_div_b\romannumeral0\XINTinfloatS[#3]{#4}/#1e#2%
}%

\def\XINT_FL_div_b #1[#2]{#1e#2}%
```

8.80 \xintFloatPow, \XINTinFloatPow

1.07: initial version. 1.09j has re-organized the core loop.
2015/12/07. I have hesitated to map ^ in expressions to \xintFloatPow rather than \xintFloatPower. But for 1.234567890123456 to the power 2145678912 with P=16, using Pow rather than Power seems to bring only about 5% gain.
This routine requires the exponent x to be compatible with \numexpr parsing.
1.2f has rewritten the code for better efficiency. Also, now the argument A for A^x is first rounded to P digits before switching to the increased working precision (which depends upon x).

```
\def\xintFloatPow   {\romannumeral0\xintfloatpow}%
\def\xintfloatpow #1{\XINT_flpow_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatPow {\romannumeral0\XINTinfloatpow }%
\def\XINTinfloatpow #1{\XINT_flpow_chkopt \XINTinfloatS #1\xint:}%
\def\XINT_flpow_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flpow_opt
       \else\expandafter\XINT_flpow_noopt
    \fi
    #1#2%
}%
\def\XINT_flpow_noopt  #1#2\xint:#3%
```

```
{%
   \expandafter\XINT_flpow_checkB_a
   \the\numexpr #3.\XINTdigits.{#2}{#1[\XINTdigits]}%
}%
\def\XINT_flpow_opt #1[\xint:#2]%
{%
   \expandafter\XINT_flpow_opt_a\the\numexpr #2.#1%
}%
\def\XINT_flpow_opt_a #1.#2#3#4%
{%
   \expandafter\XINT_flpow_checkB_a\the\numexpr #4.#1.{#3}{#2[#1]}%
}%
\def\XINT_flpow_checkB_a #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_flpow_BisZero
      0#1{\XINT_flpow_checkB_b -}%
       0-{\XINT_flpow_checkB_b {}#1}%
    \krof
}%
\def\XINT_flpow_BisZero .#1.#2#3{#3{1[0]}}%

\def\XINT_flpow_checkB_b #1#2.#3.%
{%
    \expandafter\XINT_flpow_checkB_c
    \the\numexpr\xintLength{#2}+\xint_c_iii.#3.#2.{#1}%
}%
\def\XINT_flpow_checkB_c #1.#2.%
{%
    \expandafter\XINT_flpow_checkB_d\the\numexpr#1+#2.#1.#2.%
}%
```

1.2f rounds input to P digits, first.

```
\def\XINT_flpow_checkB_d #1.#2.#3.#4.#5#6%
{%
    \expandafter \XINT_flpow_aa
    \romannumeral0\XINTinfloat [#3]{#6}{#2}{#1}{#4}{#5}%
}%

\def\XINT_flpow_aa #1[#2]#3%
{%
    \expandafter\XINT_flpow_ab\the\numexpr #2-#3\expandafter.%
    \romannumeral\XINT_rep #3\endcsname0.#1.%
}%

\def\XINT_flpow_ab #1.#2.#3.{\XINT_flpow_a #3#2[#1]}%

\def\XINT_flpow_a #1%
{%
    \xint_UDzerominusfork
```

```
      #1-\XINT_flpow_zero
      0#1{\XINT_flpow_b \iftrue}%
       0-{\XINT_flpow_b \iffalse#1}%
    \krof
}%
\def\XINT_flpow_zero #1[#2]#3#4#5#6%
{%
    #6{\if 1#51\xint_dothis {0[0]}\fi
       \xint_orthat
       {\XINT_signalcondition{DivisionByZero}{0 to the power #4}{}{0[0]}}%
      }%
}%

\def\XINT_flpow_b #1#2[#3]#4#5%
{%
    \XINT_flpow_loopI #5.#3.#2.#4.{#1\ifodd #5 \xint_c_i\fi\fi}%
}%

\def\XINT_flpow_truncate #1.#2.#3.%
{%
    \expandafter\XINT_flpow_truncate_a
    \romannumeral0\XINT_split_fromleft
    #3.#2\xint_bye2345678\xint_bye..#1.#3.%
}%

\def\XINT_flpow_truncate_a #1.#2.#3.{#3+\xintLength{#2}.#1.}%
\def\XINT_flpow_loopI #1.%
{%
    \ifnum #1=\xint_c_i\expandafter\XINT_flpow_ItoIII\fi
    \ifodd #1
       \expandafter\XINT_flpow_loopI_odd
    \else
       \expandafter\XINT_flpow_loopI_even
    \fi
    #1.%
}%

\def\XINT_flpow_ItoIII\ifodd #1\fi #2.#3.#4.#5.#6%
{%
    \expandafter\XINT_flpow_III\the\numexpr #6+\xint_c_.#3.#4.#5.%
}%

\def\XINT_flpow_loopI_even #1.#2.#3.%#4.%
{%
    \expandafter\XINT_flpow_loopI
    \the\numexpr #1/\xint_c_ii\expandafter.%
    \the\numexpr\expandafter\XINT_flpow_truncate
    \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.%
}%
\def\XINT_flpow_loopI_odd #1.#2.#3.#4.%
```

```
{%
    \expandafter\XINT_flpow_loopII
    \the\numexpr #1/\xint_c_ii-\xint_c_i\expandafter.%
    \the\numexpr\expandafter\XINT_flpow_truncate
    \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.#2.#3.%
}%
\def\XINT_flpow_loopII #1.%
{%
    \ifnum #1 = \xint_c_i\expandafter\XINT_flpow_IItoIII\fi
    \ifodd #1
       \expandafter\XINT_flpow_loopII_odd
    \else
       \expandafter\XINT_flpow_loopII_even
    \fi
    #1.%
}%
\def\XINT_flpow_loopII_even #1.#2.#3.%#4.%
{%
    \expandafter\XINT_flpow_loopII
    \the\numexpr #1/\xint_c_ii\expandafter.%
    \the\numexpr\expandafter\XINT_flpow_truncate
    \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.%
}%
\def\XINT_flpow_loopII_odd #1.#2.#3.#4.#5.#6.%
{%
    \expandafter\XINT_flpow_loopII_odda
    \the\numexpr\expandafter\XINT_flpow_truncate
    \the\numexpr#2+#5\expandafter.\romannumeral0\xintiimul{#3}{#6}.#4.%
    #1.#2.#3.%
}%
\def\XINT_flpow_loopII_odda #1.#2.#3.#4.#5.#6.%
{%
    \expandafter\XINT_flpow_loopII
    \the\numexpr #4/\xint_c_ii-\xint_c_i\expandafter.%
    \the\numexpr\expandafter\XINT_flpow_truncate
    \the\numexpr\xint_c_ii*#5\expandafter.\romannumeral0\xintiisqr{#6}.#3.%
    #1.#2.%
}%


\def\XINT_flpow_IItoIII\ifodd #1\fi #2.#3.#4.#5.#6.#7.#8%
{%
    \expandafter\XINT_flpow_III\the\numexpr #8+\xint_c_\expandafter.%
    \the\numexpr\expandafter\XINT_flpow_truncate
    \the\numexpr#3+#6\expandafter.\romannumeral0\xintiimul{#4}{#7}.#5.%
}%
```

This ending is common with \xintFloatPower.

In the case of negative exponent we need to inverse the Q-digits mantissa. This requires no special attention now as 1.2k's \xintFloat does correct rounding of fractions hence it is easy to bound the total error. It can be checked that the algorithm after final rounding to the target precision computes a value Z whose distance to the exact theoretical will be less than 0.52 ulp(Z) (and worst cases can only be slightly worse than 0.51 ulp(Z)).

In the case of the half-integer exponent (only via the expression interface,) the computation (which proceeds via \XINTinFloatPowerH) ends with a square root. This square root extraction is done with 3 guard digits (the power operations were done with more.) Then the value is rounded to the target precision. There is thus this rounding to 3 guard digits (in the case of negative exponent the reciprocal is computed before the square-root), then the square root is (computed with exact rounding for these 3 guard digits), and then there is the final rounding of this to the target precision. The total error (for positive as well as negative exponent) has been estimated to at worst possibly exceed slightly 0.5125 ulp(Z), and at any rate it is less than 0.52 ulp(Z).

```
\def\XINT_flpow_III #1.#2.#3.#4.#5%
{%
    \expandafter\XINT_flpow_IIIend
    \xint_UDsignfork
         #5{{1/#3[-#2]}}%
         -{{#3[#2]}}%
    \krof #1%
}%

\def\XINT_flpow_IIIend #1#2#3%
    {#3{\if#21\xint_afterfi{\expandafter-\romannumeral`&&@}\fi#1}}%
```

8.81 \xintFloatPower, \XINTinFloatPower

1.07. The core loop has been re-organized in 1.09j for some slight efficiency gain. The exponent B is given to \xintNum. The ^ in expressions is mapped to this routine.

Same modifications as in \xintFloatPow for 1.2f.

1.2f adds a special private macro for allowing half-integral exponents for use with ^ within \xintfloatexpr. The exponent will be first truncated to either an integer or an half-integer. The macro is not for general use.

1.2k does anew this 1.2f handling of half-integer exponents for the \xintfloatexpr parser: with 1.2f's code the final square-root extraction was applied to a value already rounded to the target precision, unneedlessly losing precision.

```
\def\xintFloatPower   {\romannumeral0\xintfloatpower}%
\def\xintfloatpower #1{\XINT_flpower_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatPower {\romannumeral0\XINTinfloatpower }%
\def\XINTinfloatpower #1{\XINT_flpower_chkopt \XINTinfloatS #1\xint:}%
```

First the special macro for use by the expression parser which checks if one raises to an half-integer exponent. This is always with \XINTdigits precision. Rewritten for 1.2k in order for the final square root to keep three guard digits.

We have to be careful that exponent #2 is not constrained by TeX bound. And we must allow fractions. The 1.2k variant does a rounding to nearest integer of half-integer, 1.2f did a truncation rather (this is done after truncation of #2 to fixed point with one digit after mark.) We try to recognize quickly the case of integer exponent, for speed, but there is overhead of going through \xintiTrunc1.

```
\def\XINTinFloatPowerH {\romannumeral0\XINTinfloatpowerh }%

\def\XINTinfloatpowerh #1#2%
{%
    \expandafter\XINT_flpowerh_a\romannumeral0\xintitrunc1{#2};%
    \XINTdigits.{#1}{\XINTinfloatS[\XINTdigits]}%
}%
```

```
\def\XINT_flpowerh_a #1;%
{%
    \if0\xintLDg{#1}\expandafter\XINT_flpowerh_int
        \else\expandafter\XINT_flpowerh_b
    \fi #1.%
}%
\def\XINT_flpowerh_int #1%
{%
    \if0#1\expandafter\XINT_flpower_BisZero
     \else\expandafter\XINT_flpowerh_i
    \fi #1%
}%
\def\XINT_flpowerh_i #10.{\expandafter\XINT_flpower_checkB_a#1.}%
\def\XINT_flpowerh_b #1.%
{%
   \expandafter\XINT_flpowerh_c\romannumeral0\xintdsrr{\xintDouble{#1}}.%
}%
\def\XINT_flpowerh_c #1.%
{%
    \ifodd\xintLDg{#1} %<- intentional space
        \expandafter\XINT_flpowerh_d\else\expandafter\XINT_flpowerh_e
    \fi #1.%
}%
\def\XINT_flpowerh_d #1.\XINTdigits.#2#3%
{%
   \XINT_flpower_checkB_a #1.\XINTdigits.{#2}\XINT_flpowerh_finish
}%
\def\XINT_flpowerh_finish #1%
   {\XINTinfloatS[\XINTdigits]{\XINTinFloatSqrt[\XINTdigits+\xint_c_iii]{#1}}}%
\def\XINT_flpowerh_e #1.%
   {\expandafter\XINT_flpower_checkB_a\romannumeral0\xinthalf{#1}.}%
```

Start of macro. Check for optional argument.

```
\def\XINT_flpower_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flpower_opt
       \else\expandafter\XINT_flpower_noopt
    \fi
    #1#2%
}%
\def\XINT_flpower_noopt  #1#2\xint:#3%
{%
   \expandafter\XINT_flpower_checkB_a
   \romannumeral0\xintnum{#3}.\XINTdigits.{#2}{#1[\XINTdigits]}%
}%
\def\XINT_flpower_opt #1[\xint:#2]%
{%
   \expandafter\XINT_flpower_opt_a\the\numexpr #2.#1%
}%
\def\XINT_flpower_opt_a #1.#2#3#4%
{%
   \expandafter\XINT_flpower_checkB_a
```

```
    \romannumeral0\xintnum{#4}.#1.{#3}{#2[#1]}%
}%
\def\XINT_flpower_checkB_a #1%
{%
    \xint_UDzerominusfork
      #1-{\XINT_flpower_BisZero 0}%
      0#1{\XINT_flpower_checkB_b -}%
       0-{\XINT_flpower_checkB_b {}#1}%
    \krof
}%
\def\XINT_flpower_BisZero 0.#1.#2#3{#3{1[0]}}%
\def\XINT_flpower_checkB_b #1#2.#3.%
{%
    \expandafter\XINT_flpower_checkB_c
    \the\numexpr\xintLength{#2}+\xint_c_iii.#3.#2.{#1}%
}%

\def\XINT_flpower_checkB_c #1.#2.%
{%
    \expandafter\XINT_flpower_checkB_d\the\numexpr#1+#2.#1.#2.%
}%

\def\XINT_flpower_checkB_d #1.#2.#3.#4.#5#6%
{%
    \expandafter \XINT_flpower_aa
    \romannumeral0\XINTinfloat [#3]{#6}{#2}{#1}{#4}{#5}%
}%

\def\XINT_flpower_aa #1[#2]#3%
{%
    \expandafter\XINT_flpower_ab\the\numexpr #2-#3\expandafter.%
    \romannumeral\XINT_rep #3\endcsname0.#1.%
}%
\def\XINT_flpower_ab #1.#2.#3.{\XINT_flpower_a #3#2[#1]}%
\def\XINT_flpower_a #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_flpow_zero
      0#1{\XINT_flpower_b \iftrue}%
       0-{\XINT_flpower_b \iffalse#1}%
    \krof
}%
\def\XINT_flpower_b #1#2[#3]#4#5%
{%
    \XINT_flpower_loopI #5.#3.#2.#4.{#1\xintiiOdd{#5}\fi}%
}%
\def\XINT_flpower_loopI #1.%
{%
    \if1\XINT_isOne {#1}\xint_dothis\XINT_flpower_ItoIII\fi
    \ifodd\xintLDg{#1} %<- intentional space
       \xint_dothis{\expandafter\XINT_flpower_loopI_odd}\fi
    \xint_orthat{\expandafter\XINT_flpower_loopI_even}%
```

```
2795     \romannumeral0\XINT_half
2796     #1\xint_bye\xint_Bye345678\xint_bye
2797     *\xint_c_v+\xint_c_v)/\xint_c_x-\xint_c_i\relax.%
2798 }%
2799 \def\XINT_flpower_ItoIII #1.#2.#3.#4.#5%
2800 {%
2801     \expandafter\XINT_flpow_III\the\numexpr #5+\xint_c_.#2.#3.#4.%
2802 }%
2803 \def\XINT_flpower_loopI_even #1.#2.#3.#4.%
2804 {%
2805     \expandafter\XINT_flpower_toloopI
2806     \the\numexpr\expandafter\XINT_flpow_truncate
2807     \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.#1.%
2808 }%
2809 \def\XINT_flpower_toloopI #1.#2.#3.#4.{\XINT_flpower_loopI #4.#1.#2.#3.}%
2810 \def\XINT_flpower_loopI_odd #1.#2.#3.#4.%
2811 {%
2812     \expandafter\XINT_flpower_toloopII
2813     \the\numexpr\expandafter\XINT_flpow_truncate
2814     \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.%
2815     #1.#2.#3.%
2816 }%
2817 \def\XINT_flpower_toloopII #1.#2.#3.#4.{\XINT_flpower_loopII #4.#1.#2.#3.}%
2818 \def\XINT_flpower_loopII #1.%
2819 {%
2820     \if1\XINT_isOne{#1}\xint_dothis\XINT_flpower_IItoIII\fi
2821     \ifodd\xintLDg{#1} %<- intentional space
2822         \xint_dothis{\expandafter\XINT_flpower_loopII_odd}\fi
2823     \xint_orthat{\expandafter\XINT_flpower_loopII_even}%


2824     \romannumeral0\XINT_half#1\xint_bye\xint_Bye345678\xint_bye
2825     *\xint_c_v+\xint_c_v)/\xint_c_x-\xint_c_i\relax.%
2826 }%
2827 \def\XINT_flpower_loopII_even #1.#2.#3.#4.%
2828 {%
2829     \expandafter\XINT_flpower_toloopII
2830     \the\numexpr\expandafter\XINT_flpow_truncate
2831     \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.#1.%
2832 }%
2833 \def\XINT_flpower_loopII_odd #1.#2.#3.#4.#5.#6.%
2834 {%
2835     \expandafter\XINT_flpower_loopII_odda
2836     \the\numexpr\expandafter\XINT_flpow_truncate
2837     \the\numexpr#2+#5\expandafter.\romannumeral0\xintiimul{#3}{#6}.#4.%
2838     #1.#2.#3.%
2839 }%
2840 \def\XINT_flpower_loopII_odda #1.#2.#3.#4.#5.#6.%
2841 {%
2842     \expandafter\XINT_flpower_toloopII
2843     \the\numexpr\expandafter\XINT_flpow_truncate
2844     \the\numexpr\xint_c_ii*#5\expandafter.\romannumeral0\xintiisqr{#6}.#3.%
2845     #4.#1.#2.%
```

```
}%
\def\XINT_flpower_IItoIII #1.#2.#3.#4.#5.#6.#7%
{%
    \expandafter\XINT_flpow_III\the\numexpr #7+\xint_c_\expandafter.%
    \the\numexpr\expandafter\XINT_flpow_truncate
    \the\numexpr#2+#5\expandafter.\romannumeral0\xintiimul{#3}{#6}.#4.%
}%
```

8.82 \xintFloatFac, \XINTFloatFac

Done at 1.2. At 1.3e \XINTinFloatFac outputs using \XINTinFloatS.

```
\def\xintFloatFac     {\romannumeral0\xintfloatfac}%
\def\xintfloatfac   #1{\XINT_flfac_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatFac   {\romannumeral0\XINTinfloatfac }%
\def\XINTinfloatfac #1{\XINT_flfac_chkopt \XINTinfloatS #1\xint:}%
\def\XINT_flfac_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flfac_opt
       \else\expandafter\XINT_flfac_noopt
    \fi
     #1#2%
}%
\def\XINT_flfac_noopt  #1#2\xint:
{%
   \expandafter\XINT_FL_fac_fork_a
   \the\numexpr \xintNum{#2}.\xint_c_i \XINTdigits\XINT_FL_fac_out{#1[\XINTdigits]}%
}%
\def\XINT_flfac_opt #1[\xint:#2]%
{%
   \expandafter\XINT_flfac_opt_a\the\numexpr #2.#1%
}%
\def\XINT_flfac_opt_a #1.#2#3%
{%
   \expandafter\XINT_FL_fac_fork_a\the\numexpr \xintNum{#3}.\xint_c_i {#1}\XINT_FL_fac_out{#2[#1]}%
}%
\def\XINT_FL_fac_fork_a #1%
{%
     \xint_UDzerominusfork
     #1-\XINT_FL_fac_iszero
     0#1\XINT_FL_fac_isneg
      0-{\XINT_FL_fac_fork_b #1}%
    \krof
}%
\def\XINT_FL_fac_iszero #1.#2#3#4#5{#5{1[0]}}%
```

1.2f XINT_FL_fac_isneg returns 0, earlier versions used 1 here.

```
\def\XINT_FL_fac_isneg  #1.#2#3#4#5%
{%
    #5{\XINT_signalcondition{InvalidOperation}
                      {Factorial of negative: (-#1)!}{}{0[0]}}%
}%
\def\XINT_FL_fac_fork_b #1.%
```

```
{%
    \ifnum #1>\xint_c_x^viii_mone\xint_dothis\XINT_FL_fac_toobig\fi
    \ifnum #1>\xint_c_x^iv\xint_dothis\XINT_FL_fac_vbig \fi
    \ifnum #1>465  \xint_dothis\XINT_FL_fac_big\fi
    \ifnum #1>101  \xint_dothis\XINT_FL_fac_med\fi
                   \xint_orthat\XINT_FL_fac_small
    #1.%
}%
\def\XINT_FL_fac_toobig #1.#2#3#4#5%
{%
    #5{\XINT_signalcondition{InvalidOperation}
                      {Factorial of too big: (#1)!}{}{0[0]}}%
}%
```

Computations are done with Q blocks of eight digits. When a multiplication has a carry, hence creates Q+1 blocks, the least significant one is dropped. The goal is to compute an approximate value X' to the exact value X, such that the final relative error (X-X')/X will be at most 10^{-P-1} with P the desired precision. Then, when we round X' to X'' with P significant digits, we can prove that the absolute error |X-X''| is bounded (strictly) by 0.6 ulp(X''). (ulp= unit in the last (significant) place). Let N be the number of such operations, the formula for Q deduces from the previous explanations is that 8Q should be at least P+9+k, with k the number of digits of N (in base 10). Note that 1.2 version used P+10+k, for 1.2f I reduced to P+9+k. Also, k should be the number of digits of the number N of multiplications done, hence for n<=10000 we can take N=n/2, or N/3, or N/4. This is rounded above by numexpr and always an overestimate of the actual number of approximate multiplications done (the first ones are exact). (vérifier ce que je raconte, j'ai la flemme là).

We then want ceil((P+k+n)/8). Using \numexpr rounding division (ARRRRRGGGHHHH), if m is a positive integer, ceil(m/8) can be computed as (m+3)/8. Thus with m=P+10+k, this gives Q<-(P+13+k)/8. The routine actually computes 8(Q-1) for use in \XINT_FL_fac_addzeros.

With 1.2f the formula is m=P+9+k, Q<-(P+12+k)/8, and we use now 4=12-8 rather than the earlier 5=13-8. Whatever happens, the value computed in \XINT_FL_fac_increaseP is at least 8. There will always be an extra block.

Note: with Digits:=32; Maple gives for 200!:

```
> factorial(200.);
375
0.78865786736479050355236321393218 10
```

My 1.2f routine (and also 1.2) outputs:

7.886578673647905035523632139321850622951e374

and this is the correct rounding because for 40 digits it computes

7.8865786736479050355236321393218506229510e374

Maple's result (contrarily to xint) is thus not the correct rounding but still it is less than 0.6 ulp wrong.

```
\def\XINT_FL_fac_vbig
   {\expandafter\XINT_FL_fac_vbigloop_a
    \the\numexpr \XINT_FL_fac_increaseP \xint_c_i   }%
\def\XINT_FL_fac_big
   {\expandafter\XINT_FL_fac_bigloop_a
    \the\numexpr \XINT_FL_fac_increaseP \xint_c_ii  }%
\def\XINT_FL_fac_med
   {\expandafter\XINT_FL_fac_medloop_a
    \the\numexpr \XINT_FL_fac_increaseP \xint_c_iii }%
\def\XINT_FL_fac_small
```

```
    {\expandafter\XINT_FL_fac_smallloop_a
     \the\numexpr \XINT_FL_fac_increaseP \xint_c_iv  }%
\def\XINT_FL_fac_increaseP #1#2.#3#4%
{%
    #2\expandafter.\the\numexpr\xint_c_viii*%
    ((\xint_c_iv+#4+\expandafter\XINT_FL_fac_countdigits
                    \the\numexpr #2/(#1*#3)\relax 87654321\Z)/\xint_c_viii).%
}%
\def\XINT_FL_fac_countdigits #1#2#3#4#5#6#7#8{\XINT_FL_fac_countdone }%
\def\XINT_FL_fac_countdone   #1#2\Z {#1}%
\def\XINT_FL_fac_out #1;![#2]#3%
    {#3{\romannumeral0\XINT_mul_out
         #1;!1\R!1\R!1\R!1\R!%
                     1\R!1\R!1\R!1\R!\W [#2]}}%
\def\XINT_FL_fac_vbigloop_a #1.#2.%
{%
    \XINT_FL_fac_bigloop_a \xint_c_x^iv.#2.%
    {\expandafter\XINT_FL_fac_vbigloop_loop\the\numexpr 100010001\expandafter.%
     \the\numexpr \xint_c_x^viii+#1.}%
}%
\def\XINT_FL_fac_vbigloop_loop #1.#2.%
{%
    \ifnum #1>#2 \expandafter\XINT_FL_fac_loop_exit\fi
    \expandafter\XINT_FL_fac_vbigloop_loop
    \the\numexpr #1+\xint_c_i\expandafter.%
    \the\numexpr #2\expandafter.\the\numexpr\XINT_FL_fac_mul #1!%
}%
\def\XINT_FL_fac_bigloop_a #1.%
{%
    \expandafter\XINT_FL_fac_bigloop_b \the\numexpr
    #1+\xint_c_i-\xint_c_ii*((#1-464)/\xint_c_ii).#1.%
}%
\def\XINT_FL_fac_bigloop_b #1.#2.#3.%
{%
    \expandafter\XINT_FL_fac_medloop_a
        \the\numexpr #1-\xint_c_i.#3.{\XINT_FL_fac_bigloop_loop #1.#2.}%
}%
\def\XINT_FL_fac_bigloop_loop #1.#2.%
{%
    \ifnum #1>#2 \expandafter\XINT_FL_fac_loop_exit\fi
    \expandafter\XINT_FL_fac_bigloop_loop
    \the\numexpr #1+\xint_c_ii\expandafter.%
    \the\numexpr #2\expandafter.\the\numexpr\XINT_FL_fac_bigloop_mul #1!%
}%
\def\XINT_FL_fac_bigloop_mul #1!%
{%
    \expandafter\XINT_FL_fac_mul
        \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
}%
\def\XINT_FL_fac_medloop_a #1.%
{%
    \expandafter\XINT_FL_fac_medloop_b
```

```
        \the\numexpr #1+\xint_c_i-\xint_c_iii*((#1-100)/\xint_c_iii).#1.%
}%
\def\XINT_FL_fac_medloop_b #1.#2.#3.%
{%
    \expandafter\XINT_FL_fac_smallloop_a
        \the\numexpr #1-\xint_c_i.#3.{\XINT_FL_fac_medloop_loop #1.#2.}%
}%
\def\XINT_FL_fac_medloop_loop #1.#2.%
{%
    \ifnum #1>#2 \expandafter\XINT_FL_fac_loop_exit\fi
    \expandafter\XINT_FL_fac_medloop_loop
    \the\numexpr #1+\xint_c_iii\expandafter.%
    \the\numexpr #2\expandafter.\the\numexpr\XINT_FL_fac_medloop_mul #1!%
}%
\def\XINT_FL_fac_medloop_mul #1!%
{%
    \expandafter\XINT_FL_fac_mul
    \the\numexpr
        \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_FL_fac_smallloop_a #1.%
{%
    \csname
       XINT_FL_fac_smallloop_\the\numexpr #1-\xint_c_iv*(#1/\xint_c_iv)\relax
    \endcsname #1.%
}%
\expandafter\def\csname XINT_FL_fac_smallloop_1\endcsname #1.#2.%
{%
    \XINT_FL_fac_addzeros #2.100000001!.{2.#1.}{#2}%
}%
\expandafter\def\csname XINT_FL_fac_smallloop_-2\endcsname #1.#2.%
{%
    \XINT_FL_fac_addzeros #2.100000002!.{3.#1.}{#2}%
}%
\expandafter\def\csname XINT_FL_fac_smallloop_-1\endcsname #1.#2.%
{%
    \XINT_FL_fac_addzeros #2.100000006!.{4.#1.}{#2}%
}%
\expandafter\def\csname XINT_FL_fac_smallloop_0\endcsname #1.#2.%
{%
    \XINT_FL_fac_addzeros #2.100000024!.{5.#1.}{#2}%
}%
\def\XINT_FL_fac_addzeros #1.%
{%
    \ifnum #1=\xint_c_viii \expandafter\XINT_FL_fac_addzeros_exit\fi
    \expandafter\XINT_FL_fac_addzeros
    \the\numexpr #1-\xint_c_viii.100000000!%
}%
```

We will manipulate by successive *small* multiplications Q blocks 1<8d>!, terminated by 1;!. We need a custom small multiplication which tells us when it has create a new block, and the least significant one should be dropped.

```
\def\XINT_FL_fac_addzeros_exit #1.#2.#3#4{\XINT_FL_fac_smallloop_loop #3#21;![-#4]}%
\def\XINT_FL_fac_smallloop_loop #1.#2.%
{%
    \ifnum #1>#2 \expandafter\XINT_FL_fac_loop_exit\fi
    \expandafter\XINT_FL_fac_smallloop_loop
    \the\numexpr #1+\xint_c_iv\expandafter.%
    \the\numexpr #2\expandafter.\romannumeral0\XINT_FL_fac_smallloop_mul #1!%
}%
\def\XINT_FL_fac_smallloop_mul #1!%
{%
    \expandafter\XINT_FL_fac_mul
    \the\numexpr
        \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%[[
\def\XINT_FL_fac_loop_exit #1!#2]#3{#3#2]}%
\def\XINT_FL_fac_mul 1#1!%
   {\expandafter\XINT_FL_fac_mul_a\the\numexpr\XINT_FL_fac_smallmul 10!{#1}}%
\def\XINT_FL_fac_mul_a #1-#2%
{%
    \if#21\xint_afterfi{\expandafter\space\xint_gob_til_exclam}\else
    \expandafter\space\fi #11;!%
}%
\def\XINT_FL_fac_minimulwc_a #1#2#3#4#5!#6#7#8#9%
{%
    \XINT_FL_fac_minimulwc_b {#1#2#3#4}{#5}{#6#7#8#9}%
}%
\def\XINT_FL_fac_minimulwc_b #1#2#3#4!#5%
{%
    \expandafter\XINT_FL_fac_minimulwc_c
    \the\numexpr \xint_c_x^ix+#5+#2*#4!{{#1}{#2}{#3}{#4}}%
}%
\def\XINT_FL_fac_minimulwc_c 1#1#2#3#4#5#6!#7%
{%
    \expandafter\XINT_FL_fac_minimulwc_d {#1#2#3#4#5}#7{#6}%
}%
\def\XINT_FL_fac_minimulwc_d #1#2#3#4#5%
{%
    \expandafter\XINT_FL_fac_minimulwc_e
    \the\numexpr \xint_c_x^ix+#1+#2*#5+#3*#4!{#2}{#4}%
}%
\def\XINT_FL_fac_minimulwc_e 1#1#2#3#4#5#6!#7#8#9%
{%
    1#6#9\expandafter!%
    \the\numexpr\expandafter\XINT_FL_fac_smallmul
    \the\numexpr \xint_c_x^viii+#1#2#3#4#5+#7*#8!%
}%
\def\XINT_FL_fac_smallmul 1#1!#21#3!%
{%
    \xint_gob_til_sc #3\XINT_FL_fac_smallmul_end;%
    \XINT_FL_fac_minimulwc_a #2!#3!{#1}{#2}%
}%
```

This is the crucial ending. I note that I used here an \ifnum test rather than the gob_til_eightzeroes

thing. Actually for eight digits there is much less difference than for only four.
  The "carry" situation is marked by a final !-1 rather than !-2 for no-carry. (a \numexpr muste be stopped, and leaving a - as delimiter is good as it will not arise earlier.)

```
\def\XINT_FL_fac_smallmul_end;\XINT_FL_fac_minimulwc_a #1!;!#2#3[#4]%
{%
   \ifnum #2=\xint_c_
       \expandafter\xint_firstoftwo\else
       \expandafter\xint_secondoftwo
   \fi
   {-2\relax[#4]}%
   {1#2\expandafter!\expandafter-\expandafter1\expandafter
                   [\the\numexpr #4+\xint_c_viii]}%
}%
```

8.83 \xintFloatPFactorial, \XINTinFloatPFactorial

2015/11/29 for 1.2f. Partial factorial pfactorial(a,b)=(a+1)...b, only for non-negative integers with a<=b<10^8.
  1.2h (2016/11/20) now avoids raising \xintError:OutOfRangePFac if the condition 0<=a<=b<10^8 is violated. Same as for \xintiiPFactorial.

```
\def\xintFloatPFactorial {\romannumeral0\xintfloatpfactorial}%
\def\xintfloatpfactorial #1{\XINT_flpfac_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatPFactorial {\romannumeral0\XINTinfloatpfactorial }%
\def\XINTinfloatpfactorial #1{\XINT_flpfac_chkopt \XINTinfloat #1\xint:}%
\def\XINT_flpfac_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flpfac_opt
       \else\expandafter\XINT_flpfac_noopt
    \fi
     #1#2%
}%
\def\XINT_flpfac_noopt  #1#2\xint:#3%
{%
   \expandafter\XINT_FL_pfac_fork
   \the\numexpr \xintNum{#2}\expandafter.%
   \the\numexpr \xintNum{#3}.\xint_c_i{\XINTdigits}{#1[\XINTdigits]}%
}%
\def\XINT_flpfac_opt #1[\xint:#2]%
{%
   \expandafter\XINT_flpfac_opt_b\the\numexpr #2.#1%
}%
\def\XINT_flpfac_opt_b #1.#2#3#4%
{%
   \expandafter\XINT_FL_pfac_fork
   \the\numexpr \xintNum{#3}\expandafter.%
   \the\numexpr \xintNum{#4}.\xint_c_i{#1}{#2[#1]}%
}%
\def\XINT_FL_pfac_fork #1#2.#3#4.%
{%
    \unless\ifnum #1#2<#3#4 \xint_dothis\XINT_FL_pfac_one\fi
    \if-#3\xint_dothis\XINT_FL_pfac_neg \fi
```

```
    \if-#1\xint_dothis\XINT_FL_pfac_zero\fi
    \ifnum #3#4>\xint_c_x^viii_mone\xint_dothis\XINT_FL_pfac_outofrange\fi
    \xint_orthat \XINT_FL_pfac_increaseP #1#2.#3#4.%
}%
\def\XINT_FL_pfac_outofrange #1.#2.#3#4#5%
{%
    #5{\XINT_signalcondition{InvalidOperation}
                        {pfactorial second arg too big: 99999999 < #2}{}{0[0]}}%
}%
\def\XINT_FL_pfac_one  #1.#2.#3#4#5{#5{1[0]}}%
\def\XINT_FL_pfac_zero #1.#2.#3#4#5{#5{0[0]}}%
\def\XINT_FL_pfac_neg -#1.-#2.%
{%
    \ifnum #1>\xint_c_x^viii\xint_dothis\XINT_FL_pfac_outofrange\fi
    \xint_orthat {%
    \ifodd\numexpr#2-#1\relax\xint_afterfi{\expandafter-\romannumeral`&&@}\fi
    \expandafter\XINT_FL_pfac_increaseP}%
    \the\numexpr #2-\xint_c_i\expandafter.\the\numexpr#1-\xint_c_i.%
}%
```

See the comments for \XINT_FL_pfac_increaseP. Case of b=a+1 should be filtered out perhaps. We only needed here to copy the \xintPFactorial macros and re-use \XINT_FL_fac_mul/\XINT_FL_fac_out. Had to modify a bit \XINT_FL_pfac_addzeroes. We can enter here directly with #3 equal to specify the precision (the calculated value before final rounding has a relative error less than #3.10^{-#4-1}), and #5 would hold the macro doing the final rounding (or truncating, if I make a FloatTrunc available) to a given number of digits, possibly not #4. By default the #3 is 1, but FloatBinomial calls it with #3=4.

```
\def\XINT_FL_pfac_increaseP #1.#2.#3#4%
{%
    \expandafter\XINT_FL_pfac_a
    \the\numexpr \xint_c_viii*((\xint_c_iv+#4+\expandafter
                  \XINT_FL_fac_countdigits\the\numexpr (#2-#1-\xint_c_i)%
                     /\ifnum #2>\xint_c_x^iv #3\else(#3*\xint_c_ii)\fi\relax
                  87654321\Z)/\xint_c_viii).#1.#2.%
}%
\def\XINT_FL_pfac_a #1.#2.#3.%
{%
    \expandafter\XINT_FL_pfac_b\the\numexpr \xint_c_i+#2\expandafter.%
    \the\numexpr#3\expandafter.%
    \romannumeral0\XINT_FL_pfac_addzeroes #1.100000001!1;![-#1]%
}%
\def\XINT_FL_pfac_addzeroes #1.%
{%
    \ifnum #1=\xint_c_viii \expandafter\XINT_FL_pfac_addzeroes_exit\fi
    \expandafter\XINT_FL_pfac_addzeroes\the\numexpr #1-\xint_c_viii.100000000!%
}%
\def\XINT_FL_pfac_addzeroes_exit #1.{ }%
\def\XINT_FL_pfac_b #1.%
{%
    \ifnum #1>9999 \xint_dothis\XINT_FL_pfac_vbigloop \fi
    \ifnum #1>463  \xint_dothis\XINT_FL_pfac_bigloop   \fi
    \ifnum #1>98   \xint_dothis\XINT_FL_pfac_medloop   \fi
```

```
                    \xint_orthat\XINT_FL_pfac_smallloop #1.%
}%
\def\XINT_FL_pfac_smallloop #1.#2.%
{%
    \ifcase\numexpr #2-#1\relax
        \expandafter\XINT_FL_pfac_end_
    \or \expandafter\XINT_FL_pfac_end_i
    \or \expandafter\XINT_FL_pfac_end_ii
    \or \expandafter\XINT_FL_pfac_end_iii
    \else\expandafter\XINT_FL_pfac_smallloop_a
    \fi #1.#2.%
}%
\def\XINT_FL_pfac_smallloop_a #1.#2.%
{%
    \expandafter\XINT_FL_pfac_smallloop_b
    \the\numexpr #1+\xint_c_iv\expandafter.%
    \the\numexpr #2\expandafter.%
    \romannumeral0\expandafter\XINT_FL_fac_mul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_FL_pfac_smallloop_b #1.%
{%
    \ifnum #1>98  \expandafter\XINT_FL_pfac_medloop   \else
                  \expandafter\XINT_FL_pfac_smallloop \fi #1.%
}%
\def\XINT_FL_pfac_medloop #1.#2.%
{%
    \ifcase\numexpr #2-#1\relax
        \expandafter\XINT_FL_pfac_end_
    \or \expandafter\XINT_FL_pfac_end_i
    \or \expandafter\XINT_FL_pfac_end_ii
    \else\expandafter\XINT_FL_pfac_medloop_a
    \fi #1.#2.%
}%
\def\XINT_FL_pfac_medloop_a #1.#2.%
{%
    \expandafter\XINT_FL_pfac_medloop_b
    \the\numexpr #1+\xint_c_iii\expandafter.%
    \the\numexpr #2\expandafter.%
    \romannumeral0\expandafter\XINT_FL_fac_mul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_FL_pfac_medloop_b #1.%
{%
    \ifnum #1>463 \expandafter\XINT_FL_pfac_bigloop   \else
                  \expandafter\XINT_FL_pfac_medloop   \fi #1.%
}%
\def\XINT_FL_pfac_bigloop #1.#2.%
{%
    \ifcase\numexpr #2-#1\relax
        \expandafter\XINT_FL_pfac_end_
    \or \expandafter\XINT_FL_pfac_end_i
```

```
    \else\expandafter\XINT_FL_pfac_bigloop_a
    \fi #1.#2.%
}%
\def\XINT_FL_pfac_bigloop_a #1.#2.%
{%
    \expandafter\XINT_FL_pfac_bigloop_b
    \the\numexpr #1+\xint_c_ii\expandafter.%
    \the\numexpr #2\expandafter.%
    \romannumeral0\expandafter\XINT_FL_fac_mul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
}%
\def\XINT_FL_pfac_bigloop_b #1.%
{%
    \ifnum #1>9999 \expandafter\XINT_FL_pfac_vbigloop  \else
                   \expandafter\XINT_FL_pfac_bigloop   \fi #1.%
}%
\def\XINT_FL_pfac_vbigloop #1.#2.%
{%
    \ifnum #2=#1
         \expandafter\XINT_FL_pfac_end_
    \else\expandafter\XINT_FL_pfac_vbigloop_a
    \fi #1.#2.%
}%
\def\XINT_FL_pfac_vbigloop_a #1.#2.%
{%
    \expandafter\XINT_FL_pfac_vbigloop
    \the\numexpr #1+\xint_c_i\expandafter.%
    \the\numexpr #2\expandafter.%
    \romannumeral0\expandafter\XINT_FL_fac_mul
    \the\numexpr\xint_c_x^viii+#1!%
}%
\def\XINT_FL_pfac_end_iii #1.#2.%
{%
    \expandafter\XINT_FL_fac_out
    \romannumeral0\expandafter\XINT_FL_fac_mul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
}%
\def\XINT_FL_pfac_end_ii #1.#2.%
{%
    \expandafter\XINT_FL_fac_out
    \romannumeral0\expandafter\XINT_FL_fac_mul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
}%
\def\XINT_FL_pfac_end_i #1.#2.%
{%
    \expandafter\XINT_FL_fac_out
    \romannumeral0\expandafter\XINT_FL_fac_mul
    \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
}%
\def\XINT_FL_pfac_end_ #1.#2.%
{%
    \expandafter\XINT_FL_fac_out
```

```
3255     \romannumeral0\expandafter\XINT_FL_fac_mul
3256     \the\numexpr \xint_c_x^viii+#1!%
3257 }%
```

8.84 \xintFloatBinomial, \XINTinFloatBinomial

1.2f. We compute binomial(x,y) as pfac(x-y,x)/y!, where the numerator and denominator are computed with a relative error at most 4.10^{-P-2}, then rounded (once I have a float truncation, I will use truncation rather) to P+3 digits, and finally the quotient is correctly rounded to P digits. This will guarantee that the exact value X differs from the computed one Y by at most 0.6 ulp(Y). (2015/12/01).

2016/11/19 for 1.2h. As for \xintiiBinomial, hard to understand why last year I coded this to raise an error if y<0 or y>x ! The question of the Gamma function is for another occasion, here x and y must be (small) integers.

```
3258 \def\xintFloatBinomial    {\romannumeral0\xintfloatbinomial}%
3259 \def\xintfloatbinomial   #1{\XINT_flbinom_chkopt \xintfloat #1\xint:}%
3260 \def\XINTinFloatBinomial   {\romannumeral0\XINTinfloatbinomial }%
3261 \def\XINTinfloatbinomial #1{\XINT_flbinom_chkopt \XINTinfloat #1\xint:}%
3262 \def\XINT_flbinom_chkopt #1#2%
3263 {%
3264     \ifx [#2\expandafter\XINT_flbinom_opt
3265        \else\expandafter\XINT_flbinom_noopt
3266     \fi  #1#2%
3267 }%
3268 \def\XINT_flbinom_noopt #1#2\xint:#3%
3269 {%
3270     \expandafter\XINT_FL_binom_a
3271     \the\numexpr\xintNum{#2}\expandafter.\the\numexpr\xintNum{#3}.\XINTdigits.#1%
3272 }%
3273 \def\XINT_flbinom_opt #1[\xint:#2]#3#4%
3274 {%
3275     \expandafter\XINT_FL_binom_a
3276     \the\numexpr\xintNum{#3}\expandafter.\the\numexpr\xintNum{#4}\expandafter.%
3277     \the\numexpr #2.#1%
3278 }%
3279 \def\XINT_FL_binom_a #1.#2.%
3280 {%
3281     \expandafter\XINT_FL_binom_fork \the\numexpr #1-#2.#2.#1.%
3282 }%
3283 \def\XINT_FL_binom_fork #1#2.#3#4.#5#6.%
3284 {%
3285     \if-#5\xint_dothis \XINT_FL_binom_neg\fi
3286     \if-#1\xint_dothis \XINT_FL_binom_zero\fi
3287     \if-#3\xint_dothis \XINT_FL_binom_zero\fi
3288     \if0#1\xint_dothis \XINT_FL_binom_one\fi
3289     \if0#3\xint_dothis \XINT_FL_binom_one\fi
3290     \ifnum #5#6>\xint_c_x^viii_mone \xint_dothis\XINT_FL_binom_toobig\fi
3291     \ifnum #1#2>#3#4  \xint_dothis\XINT_FL_binom_ab \fi
3292                       \xint_orthat\XINT_FL_binom_aa
3293     #1#2.#3#4.#5#6.%
3294 }%
3295 \def\XINT_FL_binom_neg #1.#2.#3.#4.#5%
```

```
{%
    #5[#4]{\XINT_signalcondition{InvalidOperation}
                           {binomial with first arg negative: #3}{}{0[0]}}%
}%
\def\XINT_FL_binom_toobig #1.#2.#3.#4.#5%
{%
    #5[#4]{\XINT_signalcondition{InvalidOperation}
                           {binomial with first arg too big: 99999999 < #3}{}{0[0]}}%
}%
\def\XINT_FL_binom_one  #1.#2.#3.#4.#5{#5[#4]{1[0]}}%
\def\XINT_FL_binom_zero #1.#2.#3.#4.#5{#5[#4]{0[0]}}%
\def\XINT_FL_binom_aa  #1.#2.#3.#4.#5%
{%
    #5[#4]{\xintDiv{\XINT_FL_pfac_increaseP
           #2.#3.\xint_c_iv{#4+\xint_c_i}{\XINTinfloat[#4+\xint_c_iii]}}%
           {\XINT_FL_fac_fork_b
           #1.\xint_c_iv{#4+\xint_c_i}\XINT_FL_fac_out{\XINTinfloat[#4+\xint_c_iii]}}}%
}%
\def\XINT_FL_binom_ab  #1.#2.#3.#4.#5%
{%
    #5[#4]{\xintDiv{\XINT_FL_pfac_increaseP
           #1.#3.\xint_c_iv{#4+\xint_c_i}{\XINTinfloat[#4+\xint_c_iii]}}%
           {\XINT_FL_fac_fork_b
           #2.\xint_c_iv{#4+\xint_c_i}\XINT_FL_fac_out{\XINTinfloat[#4+\xint_c_iii]}}}%
}%
```

8.85 \xintFloatSqrt, \XINTinFloatSqrt

First done for 1.08.

The float version was developed at the same time as the integer one and even a bit earlier. As a result the integer variant had some sub-optimal parts. Anyway, for 1.2f I have rewritten the integer variant, and the float variant delegates all preparatory wrok for it until the last step. In particular the very low precisions are not penalized anymore from doing computations for at least 17 or 18 digits. Both the large and small precisions give quite shorter computation times.

Also, after examining more closely the achieved precision I decided to extend the float version in order for it to obtain the correct rounding (for inputs already of at most P digits with P the precision) of the theoretical exact value.

Beyond about 500 digits of precision the efficiency decreases swiftly, as is the case generally speaking with xintcore/xint/xintfrac arithmetic macros.

Final note: with 1.2f the input is always first rounded to P significant places.

```
\def\xintFloatSqrt     {\romannumeral0\xintfloatsqrt }%
\def\xintfloatsqrt   #1{\XINT_flsqrt_chkopt \xintfloat #1\xint:}%
\def\XINTinFloatSqrt   {\romannumeral0\XINTinfloatsqrt }%
\def\XINTinfloatsqrt #1{\XINT_flsqrt_chkopt \XINTinfloat #1\xint:}%
\def\XINT_flsqrt_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flsqrt_opt
       \else\expandafter\XINT_flsqrt_noopt
    \fi  #1#2%
}%
\def\XINT_flsqrt_noopt #1#2\xint:%
```

```
{%
    \expandafter\XINT_FL_sqrt_a
                \romannumeral0\XINTinfloat[\XINTdigits]{#2}\XINTdigits.#1%
}%
\def\XINT_flsqrt_opt #1[\xint:#2]%#3%
{%
    \expandafter\XINT_flsqrt_opt_a\the\numexpr #2.#1%
}%
\def\XINT_flsqrt_opt_a #1.#2#3%
{%
    \expandafter\XINT_FL_sqrt_a\romannumeral0\XINTinfloat[#1]{#3}#1.#2%
}%
\def\XINT_FL_sqrt_a #1%
{%
    \xint_UDzerominusfork
     #1-\XINT_FL_sqrt_iszero
     0#1\XINT_FL_sqrt_isneg
      0-{\XINT_FL_sqrt_pos #1}%
    \krof
}%[
\def\XINT_FL_sqrt_iszero #1]#2.#3{#3[#2]{0[0]}}%
\def\XINT_FL_sqrt_isneg #1]#2.#3%
{%
   #3[#2]{\XINT_signalcondition{InvalidOperation}
                        {Square root of negative: -#1]}{}{0[0]}}%
}%


\def\XINT_FL_sqrt_pos #1[#2]#3.%
{%
    \expandafter\XINT_flsqrt
    \the\numexpr #3\ifodd #2 \xint_dothis {+\xint_c_iii.(#2+\xint_c_i).0}\fi
    \xint_orthat {+\xint_c_ii.#2.{}}#100.#3.%
}%


\def\XINT_flsqrt #1.#2.%
{%
    \expandafter\XINT_flsqrt_a
    \the\numexpr #2/\xint_c_ii-(#1-\xint_c_i)/\xint_c_ii.#1.%
}%


\def\XINT_flsqrt_a #1.#2.#3#4.#5.%
{%
    \expandafter\XINT_flsqrt_b
    \the\numexpr (#2-\xint_c_i)/\xint_c_ii\expandafter.%
    \romannumeral0\XINT_sqrt_start #2.#4#3.#5.#2.#4#3.#5.#1.%
}%


\def\XINT_flsqrt_b #1.#2#3%
{%
   \expandafter\XINT_flsqrt_c
   \romannumeral0\xintiisub
```

```
    {\XINT_dsx_addzeros {#1}#2;}%
    {\xintiiDivRound{\XINT_dsx_addzeros {#1}#3;}%
                    {\XINT_dbl#2\xint_bye2345678\xint_bye*\xint_c_ii\relax}}.%
}%

\def\XINT_flsqrt_c #1.#2.%
{%
    \expandafter\XINT_flsqrt_d
    \romannumeral0\XINT_split_fromleft#2.#1\xint_bye2345678\xint_bye..%
}%

\def\XINT_flsqrt_d #1.#2#3.%
{%
    \ifnum #2=\xint_c_v
    \expandafter\XINT_flsqrt_f\else\expandafter\XINT_flsqrt_finish\fi
    #2#3.#1.%
}%

\def\XINT_flsqrt_finish #1#2.#3.#4.#5.#6.#7.#8{#8[#6]{#3#1[#7]}}%

\def\XINT_flsqrt_f 5#1.%
   {\expandafter\XINT_flsqrt_g\romannumeral0\xintinum{#1}\relax.}%
\def\XINT_flsqrt_g #1#2#3.{\if\relax#2\xint_dothis{\XINT_flsqrt_h #1}\fi
                           \xint_orthat{\XINT_flsqrt_finish 5.}}%
\def\XINT_flsqrt_h #1{\ifnum #1<\xint_c_iii\xint_dothis{\XINT_flsqrt_again}\fi
                      \xint_orthat{\XINT_flsqrt_finish 5.}}%

\def\XINT_flsqrt_again #1.#2.%
{%
    \expandafter\XINT_flsqrt_again_a\the\numexpr #2+\xint_c_viii.%
}%

\def\XINT_flsqrt_again_a #1.#2.#3.%
{%
    \expandafter\XINT_flsqrt_b
    \the\numexpr (#1-\xint_c_i)/\xint_c_ii\expandafter.%
    \romannumeral0\XINT_sqrt_start #1.#200000000.#3.%
                                   #1.#200000000.#3.%
}%
```

8.86 \xintFloatE, \XINTinFloatE

1.07: The fraction is the first argument contrarily to \xintTrunc and \xintRound.
1.2k had to rewrite this since there is no more a \XINT_float_a macro. Attention about \XINTinFloatE: it is for use by xintexpr.sty, contrarily to other \XINTinFloat<foo> macros it inserts itself the [\XINTdigits] thing, and with value 0 it produces on output 0[N], not 0[0].

```
\def\xintFloatE   {\romannumeral0\xintfloate }%
\def\xintfloate #1{\XINT_floate_chkopt #1\xint:}%
\def\XINT_floate_chkopt #1%
```

```
{%
    \ifx [#1\expandafter\XINT_floate_opt
       \else\expandafter\XINT_floate_noopt
    \fi  #1%
}%
\def\XINT_floate_noopt #1\xint:%
{%
    \expandafter\XINT_floate_post
    \romannumeral0\XINTinfloat[\XINTdigits]{#1}\XINTdigits.%
}%
\def\XINT_floate_opt [\xint:#1]%
{%
    \expandafter\XINT_floate_opt_a\the\numexpr #1.%
}%
\def\XINT_floate_opt_a #1.#2%
{%
    \expandafter\XINT_floate_post
    \romannumeral0\XINTinfloat[#1]{#2}#1.%
}%
\def\XINT_floate_post #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_floate_zero
       0#1\XINT_floate_neg
       0-\XINT_floate_pos
    \krof #1%
}%[
\def\XINT_floate_zero #1]#2.#3{ 0.e0}%
\def\XINT_floate_neg-{\expandafter-\romannumeral0\XINT_floate_pos}%


\def\XINT_floate_pos #1#2[#3]#4.#5%
{%
    \expandafter\XINT_float_pos_done\the\numexpr#3+#4+#5-\xint_c_i.#1.#2;%
}%
\def\XINTinFloatE {\romannumeral0\XINTinfloate }%
\def\XINTinfloate
   {\expandafter\XINT_infloate\romannumeral0\XINTinfloat[\XINTdigits]}%
\def\XINT_infloate #1[#2]#3%
   {\expandafter\XINT_infloate_end\the\numexpr #3+#2.{#1}}%
\def\XINT_infloate_end #1.#2{ #2[#1]}%
```

8.87 \XINTinFloatMod

1.1. Pour emploi dans xintexpr. Code shortened at 1.2p.

```
\def\XINTinFloatMod {\romannumeral0\XINTinfloatmod [\XINTdigits]}%
\def\XINTinfloatmod [#1]#2#3%
{%
    \XINTinfloat[#1]{\xintMod
        {\romannumeral0\XINTinfloat[#1]{#2}}%
        {\romannumeral0\XINTinfloat[#1]{#3}}}%
}%
```

8.88 \XINTinFloatDivFloor

1.2p. Formerly // and /: in \xintfloatexpr used \xintDivFloor and \xintMod, hence did not round their operands to float precision beforehand.

```
\def\XINTinFloatDivFloor {\romannumeral0\XINTinfloatdivfloor [\XINTdigits]}%
\def\XINTinfloatdivfloor [#1]#2#3%
{%
    \xintdivfloor
          {\romannumeral0\XINTinfloat[#1]{#2}}%
          {\romannumeral0\XINTinfloat[#1]{#3}}%
}%
```

8.89 \XINTinFloatDivMod

1.2p. Pour emploi dans xintexpr, donc je ne prends pas la peine de faire l'expansion du modulo, qui se produira dans le \csname.

Hésitation sur le quotient, faut-il l'arrondir immédiatement ? Finalement non, le produire comme un integer.

Breaking change at 1.4 as output format is not comma separated anymore. Attention also that it uses \expanded.

No time now at the time of completion of the big 1.4 rewrite of xintexpr to test whether code efficiency here can be improved to expand the second item of output.

```
\def\XINTinFloatDivMod {\romannumeral0\XINTinfloatdivmod [\XINTdigits]}%
\def\XINTinfloatdivmod [#1]#2#3%
{%
    \expandafter\XINT_infloatdivmod
    \romannumeral0\xintdivmod
          {\romannumeral0\XINTinfloat[#1]{#2}}%
          {\romannumeral0\XINTinfloat[#1]{#3}}%
    {#1}%
}%
\def\XINT_infloatdivmod #1#2#3{\expanded{{#1}{\XINTinFloat[#3]{#2}}}}%
```

8.90 \xintifFloatInt

1.3a for ifint() function in \xintfloatexpr.

```
\def\xintifFloatInt {\romannumeral0\xintiffloatint}%
\def\xintiffloatint #1{\expandafter\XINT_iffloatint
                       \romannumeral0\xintrez{\XINTinFloat[\XINTdigits]{#1}}}%
\def\XINT_iffloatint #1#2/1[#3]%
{%
  \if 0#1\xint_dothis\xint_stop_atfirstoftwo\fi
  \ifnum#3<\xint_c_\xint_dothis\xint_stop_atsecondoftwo\fi
  \xint_orthat\xint_stop_atfirstoftwo
}%
```

8.91 \xintFloatIsInt

1.3d for isint() function in \xintfloatexpr.

```
\def\xintFloatIsInt {\romannumeral0\xintfloatisint}%
\def\xintfloatisint #1{\expandafter\XINT_iffloatint
     \romannumeral0\xintrez{\XINTinFloat[\XINTdigits]{#1}}10}%
```

8.92 \XINTinFloatdigits, \XINTinFloatSqrtdigits, \XINTinFloatFacdigits, \XINTiLogTendigits

For \xintNewExpr matters, mainly.
  At 1.3e I add \XINTinFloatSdigits and use it at various places. I also modified \XINTinFloatFac to use S(hort) output format.
  Also added \XINTiLogTendigits.
  This whole stuff moved over from xintexpr.sty at 1.4

```
\def\XINTinFloatdigits     {\XINTinFloat    [\XINTdigits]}%
\def\XINTinFloatSdigits    {\XINTinFloatS   [\XINTdigits]}%
\def\XINTinFloatSqrtdigits {\XINTinFloatSqrt[\XINTdigits]}%
\def\XINTinFloatFacdigits  {\XINTinFloatFac [\XINTdigits]}%
\def\XINTFloatiLogTendigits{\XINTFloatiLogTen[\XINTdigits]}%
```

8.93 (WIP) \XINTinRandomFloatS, \XINTinRandomFloatSdigits

1.3b. Support for random() function.
  Thus as it is a priori only for xintexpr usage, it expands inside \csname context, but as we need to get rid of initial zeros we use \xintRandomDigits not \xintXRandomDigits (\expanded would have a use case here).
  And anyway as we want to be able to use random() in \xintdeffunc/\xintNewExpr, it is good to have f-expandable macros, so we add the small overhead to make it f-expandable.
  We don't have to be very efficient in removing leading zeroes, as there is only 10% chance for each successive one. Besides we use (current) internal storage format of the type A[N], where A is not required to be with \xintDigits digits, so N will simply be -\xintDigits and needs no adjustment.
  In case we use in future with #1 something else than \xintDigits we do the 0-(#1) construct.
  I had some qualms about doing a random float like this which means that when there are leading zeros in the random digits the (virtual) mantissa ends up with trailing zeros. That did not feel right but I checked random() in Python (which of course uses radix 2), and indeed this is what happens there.

```
\def\XINTinRandomFloatS{\romannumeral0\XINTinrandomfloatS}%
\def\XINTinRandomFloatSdigits{\XINTinRandomFloatS[\XINTdigits]}%
\def\XINTinrandomfloatS[#1]%
{%
    \expandafter\XINT_inrandomfloatS\the\numexpr\xint_c_-(#1)\xint:
}%
\def\XINT_inrandomfloatS-#1\xint:
{%
    \expandafter\XINT_inrandomfloatS_a
    \romannumeral0\xintrandomdigits{#1}[-#1]%
}%
```

  We add one macro to handle a tiny bit faster 90of cases, after all we also use one extra macro for the completely improbable all 0 case.

```
\def\XINT_inrandomfloatS_a#1%
```

```
{%
    \if#10\xint_dothis{\XINT_inrandomfloatS_b}\fi
    \xint_orthat{ #1}%
}%[
\def\XINT_inrandomfloatS_b#1%
{%
    \if#1[\xint_dothis{\XINT_inrandomfloatS_zero}\fi% ]
    \if#10\xint_dothis{\XINT_inrandomfloatS_b}\fi
    \xint_orthat{ #1}%
}%[
\def\XINT_inrandomfloatS_zero#1]{ 0[0]}%
```

8.94 (WIP) \XINTinRandomFloatSixteen

1.3b. Support for qrand() function.

```
\def\XINTinRandomFloatSixteen%
{%
    \romannumeral0\expandafter\XINT_inrandomfloatS_a
    \romannumeral`&&@\expandafter\XINT_eightrandomdigits
                \romannumeral`&&@\XINT_eightrandomdigits[-16]%
}%
```

8.95 \PoorManLogBaseTen

1.3f. Code originally in poormanlog v0.4 got transferred here. It produces the logarithm in base 10 with an error (believed to be at most) about 1 unit in the 9th (i.e. last) fractional digit. Testing seems to indicate error at most 2 units.

```
\def\PoorManLogBaseTen{\romannumeral0\poormanlogbaseten}%
\def\poormanlogbaseten #1%
   {\expandafter\PML@logbaseten\romannumeral0\XINTinfloat[9]{#1}}%
\def\PML@logbaseten#1[#2]%
{%
    \xintiiadd{\xintDSx{-9}{\the\numexpr#2+8\relax}}{\the\numexpr\PML@#1.}%
    [-9]%
}%
```

8.96 \PoorManPowerOfTen

1.3f. Transferred from poormanlog v0.4. Produces the 10^#1 with 9 digits of float precision, with an error (believed to be) at most 2 units in the last place. Of course for this the input must be precise enough to have 9 fractional digits of **fixed point** precision.

Attention that this breaks with low level Number too big error if integral part of argument exceeds TeX bound on integers. Indeed some \numexpr is used in the code to subtract 8... but anyway xintfrac allows for scientific exponents only integers within TeX bounds, so even if it did not break here it would quickly elsewhere.

```
\def\PoorManPowerOfTen{\the\numexpr\poormanpoweroften}%
\def\poormanpoweroften #1%
   {\expandafter\PML@powoften\romannumeral0\xintraw{#1}}%
\def\PML@powoften#1%
{%
```

```
    \xint_UDzerominusfork
      #1-\PML@powoften@zero
      0#1\PML@powoften@neg
       0-\PML@powoften@pos
    \krof #1%
}%
\def\PML@powoften@zero 0{1\relax}%/1[0]
\def\PML@powoften@pos#1[#2]%
{%
    \expandafter\PML@powoften@pos@a\romannumeral0\xintround{9}{#1[#2]}.%
}%
\def\PML@powoften@pos@a#1.#2.{\PML@Pa#2.\expandafter[\the\numexpr-8+#1]}%
\def\PML@powoften@neg#1[#2]%
{%
    \expandafter\PML@powoften@neg@a\romannumeral0\xintround{9}{#1[#2]}.%
}%
\def\PML@powoften@neg@a#1.#2.%
{\ifnum#2=\xint_c_ \xint_afterfi{1\relax/1[#1]}\else
   \expandafter\expandafter\expandafter
   \PML@Pa\expandafter\xint_gobble_i\the\numexpr2000000000-#2.%
   \expandafter[\the\numexpr-9+#1\expandafter]\fi
}%
```

8.97 \PoorManPower

1.3f. This code originally in poormanlog v0.4 transferred here. It does #1 to the power #2.

```
\def\PoorManPower#1#2%
{%
    \PoorManPowerOfTen{\xintMul{#2}{\PoorManLogBaseTen{#1}}}%
}%
```

8.98 Support macros for natural logarithm and exponential xintexpr functions

At 1.3f, the poormanlog v0.04 extension to xintfrac.sty got transferred here. These macros from xintlog.sty 1.3e got transferred here too.

```
\def\xintLog#1{\xintMul{\PoorManLogBaseTen{#1}}{23025850923[-10]}}%
\def\XINTinFloatLog#1{\XINTinFloatMul{\PoorManLogBaseTen{#1}}{23025850923[-10]}}%
\def\xintExp#1{\PoorManPowerOfTen{\xintMul{#1}{434294481903[-12]}}}%
\def\XINTinFloatExp#1{\PoorManPowerOfTen{\XINTinFloatMul{#1}{434294481903[-12]}}}%
\let\XINTinFloatMaxof\XINT_Maxof
\let\XINTinFloatMinof\XINT_Minof
\let\XINTinFloatSum\XINT_Sum
\let\XINTinFloatPrd\XINT_Prd
\XINT_restorecatcodes_endinput%
```

9 Package xintseries implementation

The commenting is currently (2021/03/29) very sparse.

9.1 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from Heiko Oberdiek's packages, then modified. The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintseries.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintseries}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintseries.sty
      \ifx\w\relax % but xintfrac.sty not yet loaded.
         \def\z{\endgroup\input xintfrac.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xintfrac.sty not yet loaded.
            \def\z{\endgroup\RequirePackage{xintfrac}}%
          \fi
      \else
```

```
        \aftergroup\endinput % xintseries already loaded.
      \fi
    \fi
  \fi
\z%
\XINTsetupcatcodes% defined in xintkernel.sty
```

9.2 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintseries}%
  [2021/03/29 v1.4d Expandable partial sums with xint package (JFB)]%
```

9.3 \xintSeries

```
\def\xintSeries {\romannumeral0\xintseries }%
\def\xintseries #1#2%
{%
    \expandafter\XINT_series\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_series #1#2#3%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi {\XINT_series_loop {#1}{0}{#2}{#3}}%
   \fi
}%
\def\XINT_series_loop #1#2#3#4%
{%
    \ifnum #3>#1 \else \XINT_series_exit \fi
    \expandafter\XINT_series_loop\expandafter
    {\the\numexpr #1+1\expandafter }\expandafter
    {\romannumeral0\xintadd {#2}{#4{#1}}}%
    {#3}{#4}%
}%
\def\XINT_series_exit \fi #1#2#3#4#5#6#7#8%
{%
    \fi\xint_gobble_ii #6%
}%
```

9.4 \xintiSeries

```
\def\xintiSeries {\romannumeral0\xintiseries }%
\def\xintiseries #1#2%
{%
    \expandafter\XINT_iseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_iseries #1#2#3%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
```

```
      \xint_afterfi {\XINT_iseries_loop {#1}{0}{#2}{#3}}%
   \fi
}%
\def\XINT_iseries_loop #1#2#3#4%
{%
    \ifnum #3>#1 \else \XINT_iseries_exit \fi
    \expandafter\XINT_iseries_loop\expandafter
    {\the\numexpr #1+1\expandafter }\expandafter
    {\romannumeral0\xintiiadd {#2}{#4{#1}}}%
    {#3}{#4}%
}%
\def\XINT_iseries_exit \fi #1#2#3#4#5#6#7#8%
{%
    \fi\xint_gobble_ii #6%
}%
```

9.5 \xintPowerSeries

The 1.03 version was very lame and created a build-up of denominators. (this was at a time \xintAdd always multiplied denominators, by the way) The Horner scheme for polynomial evaluation is used in 1.04, this cures the denominator problem and drastically improves the efficiency of the macro. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintPowerSeries {\romannumeral0\xintpowerseries }%
\def\xintpowerseries #1#2%
{%
    \expandafter\XINT_powseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_powseries #1#2#3#4%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\XINT_powseries_loop_i {#3{#2}}{#1}{#2}{#3}{#4}}%
   \fi
}%
\def\XINT_powseries_loop_i #1#2#3#4#5%
{%
    \ifnum #3>#2 \else\XINT_powseries_exit_i\fi
    \expandafter\XINT_powseries_loop_ii\expandafter
    {\the\numexpr #3-1\expandafter}\expandafter
    {\romannumeral0\xintmul {#1}{#5}}{#2}{#4}{#5}%
}%
\def\XINT_powseries_loop_ii #1#2#3#4%
{%
   \expandafter\XINT_powseries_loop_i\expandafter
   {\romannumeral0\xintadd {#4{#1}}{#2}}{#3}{#1}{#4}%
}%
\def\XINT_powseries_exit_i\fi #1#2#3#4#5#6#7#8#9%
{%
```

```
    \fi \XINT_powseries_exit_ii  #6{#7}%
}%
\def\XINT_powseries_exit_ii #1#2#3#4#5#6%
{%
    \xintmul{\xintPow {#5}{#6}}{#4}%
}%
```

9.6 \xintPowerSeriesX

Same as \xintPowerSeries except for the initial expansion of the x parameter. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintPowerSeriesX {\romannumeral0\xintpowerseriesx }%
\def\xintpowerseriesx #1#2%
{%
    \expandafter\XINT_powseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_powseriesx #1#2#3#4%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\expandafter\XINT_powseriesx_pre\expandafter
                  {\romannumeral`&&@#4}{#1}{#2}{#3}%
      }%
   \fi
}%
\def\XINT_powseriesx_pre #1#2#3#4%
{%
    \XINT_powseries_loop_i {#4{#3}}{#2}{#3}{#4}{#1}%
}%
```

9.7 \xintRationalSeries

This computes F(a)+...+F(b) on the basis of the value of F(a) and the ratios F(n)/F(n-1). As in \xintPowerSeries we use an iterative scheme which has the great advantage to avoid denominator build-up. This makes exact computations possible with exponential type series, which would be completely inaccessible to \xintSeries. #1=a, #2=b, #3=F(a), #4=ratio function Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintRationalSeries {\romannumeral0\xintratseries }%
\def\xintratseries #1#2%
{%
    \expandafter\XINT_ratseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_ratseries #1#2#3#4%
```

```
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\XINT_ratseries_loop {#2}{1}{#1}{#4}{#3}}%
   \fi
}%
\def\XINT_ratseries_loop #1#2#3#4%
{%
    \ifnum #1>#3 \else\XINT_ratseries_exit_i\fi
    \expandafter\XINT_ratseries_loop\expandafter
    {\the\numexpr #1-1\expandafter}\expandafter
    {\romannumeral0\xintadd {1}{\xintMul {#2}{#4{#1}}}}{#3}{#4}%
}%
\def\XINT_ratseries_exit_i\fi #1#2#3#4#5#6#7#8%
{%
    \fi \XINT_ratseries_exit_ii  #6%
}%
\def\XINT_ratseries_exit_ii #1#2#3#4#5%
{%
    \XINT_ratseries_exit_iii #5%
}%
\def\XINT_ratseries_exit_iii #1#2#3#4%
{%
    \xintmul{#2}{#4}%
}%
```

9.8 \xintRationalSeriesX

a,b,initial,ratiofunction,x
This computes F(a,x)+...+F(b,x) on the basis of the value of F(a,x) and the ratios F(n,x)/F(n-1,x). The argument x is first expanded and it is the value resulting from this which is used then throughout. The initial term F(a,x) must be defined as one-parameter macro which will be given x. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintRationalSeriesX {\romannumeral0\xintratseriesx }%
\def\xintratseriesx #1#2%
{%
    \expandafter\XINT_ratseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_ratseriesx #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\expandafter\XINT_ratseriesx_pre\expandafter
                    {\romannumeral`&&@#5}{#2}{#1}{#4}{#3}%
      }%
```

```
    \fi
}%
\def\XINT_ratseriesx_pre #1#2#3#4#5%
{%
    \XINT_ratseries_loop {#2}{1}{#3}{#4{#1}}{#5{#1}}%
}%
```

9.9 \xintFxPtPowerSeries

I am not two happy with this piece of code. Will make it more economical another day. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a: forgot last time some optimization from the change to \numexpr.

```
\def\xintFxPtPowerSeries {\romannumeral0\xintfxptpowerseries }%
\def\xintfxptpowerseries #1#2%
{%
    \expandafter\XINT_fppowseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_fppowseries #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi
        {\expandafter\XINT_fppowseries_loop_pre\expandafter
           {\romannumeral0\xinttrunc {#5}{\xintPow {#4}{#1}}}%
          {#1}{#4}{#2}{#3}{#5}%
        }%
   \fi
}%
\def\XINT_fppowseries_loop_pre #1#2#3#4#5#6%
{%
    \ifnum #4>#2 \else\XINT_fppowseries_dont_i \fi
    \expandafter\XINT_fppowseries_loop_i\expandafter
    {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
    {\romannumeral0\xintitrunc {#6}{\xintMul {#5{#2}}{#1}}}%
    {#1}{#3}{#4}{#5}{#6}%
}%
\def\XINT_fppowseries_dont_i \fi\expandafter\XINT_fppowseries_loop_i
    {\fi \expandafter\XINT_fppowseries_dont_ii }%
\def\XINT_fppowseries_dont_ii #1#2#3#4#5#6#7{\xinttrunc {#7}{#2[-#7]}}%
\def\XINT_fppowseries_loop_i #1#2#3#4#5#6#7%
{%
    \ifnum #5>#1 \else \XINT_fppowseries_exit_i \fi
    \expandafter\XINT_fppowseries_loop_ii\expandafter
    {\romannumeral0\xinttrunc {#7}{\xintMul {#3}{#4}}}%
    {#1}{#4}{#2}{#5}{#6}{#7}%
}%
\def\XINT_fppowseries_loop_ii #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_fppowseries_loop_i\expandafter
```

```
    {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
    {\romannumeral0\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}}%
    {#1}{#3}{#5}{#6}{#7}%
}%
\def\XINT_fppowseries_exit_i\fi\expandafter\XINT_fppowseries_loop_ii
    {\fi \expandafter\XINT_fppowseries_exit_ii }%
\def\XINT_fppowseries_exit_ii #1#2#3#4#5#6#7%
{%
    \xinttrunc {#7}
    {\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}[-#7]}%
}%
```

9.10 \xintFxPtPowerSeriesX

a,b,coeff,x,D
Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintFxPtPowerSeriesX {\romannumeral0\xintfxptpowerseriesx }%
\def\xintfxptpowerseriesx #1#2%
{%
    \expandafter\XINT_fppowseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_fppowseriesx #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi
        {\expandafter \XINT_fppowseriesx_pre \expandafter
         {\romannumeral`&&@#4}{#1}{#2}{#3}{#5}%
        }%
   \fi
}%
\def\XINT_fppowseriesx_pre #1#2#3#4#5%
{%
    \expandafter\XINT_fppowseries_loop_pre\expandafter
       {\romannumeral0\xinttrunc {#5}{\xintPow {#1}{#2}}}%
       {#2}{#1}{#3}{#4}{#5}%
}%
```

9.11 \xintFloatPowerSeries

1.08a. I still have to re-visit \xintFxPtPowerSeries; temporarily I just adapted the code to the case of floats.

```
\def\xintFloatPowerSeries {\romannumeral0\xintfloatpowerseries }%
\def\xintfloatpowerseries #1{\XINT_flpowseries_chkopt #1\xint:}%
\def\XINT_flpowseries_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_flpowseries_opt
```

```
288         \else\expandafter\XINT_flpowseries_noopt
289     \fi
290     #1%
291 }%
292 \def\XINT_flpowseries_noopt  #1\xint:#2%
293 {%
294     \expandafter\XINT_flpowseries\expandafter
295     {\the\numexpr #1\expandafter}\expandafter
296     {\the\numexpr #2}\XINTdigits
297 }%
298 \def\XINT_flpowseries_opt [\xint:#1]#2#3%
299 {%
300     \expandafter\XINT_flpowseries\expandafter
301     {\the\numexpr #2\expandafter}\expandafter
302     {\the\numexpr #3\expandafter}{\the\numexpr #1}%
303 }%
304 \def\XINT_flpowseries #1#2#3#4#5%
305 {%
306    \ifnum #2<#1
307       \xint_afterfi { 0.e0}%
308    \else
309       \xint_afterfi
310         {\expandafter\XINT_flpowseries_loop_pre\expandafter
311            {\romannumeral0\XINTinfloatpow [#3]{#5}{#1}}%
312           {#1}{#5}{#2}{#4}{#3}%
313         }%
314    \fi
315 }%
316 \def\XINT_flpowseries_loop_pre #1#2#3#4#5#6%
317 {%
318     \ifnum #4>#2 \else\XINT_flpowseries_dont_i \fi
319     \expandafter\XINT_flpowseries_loop_i\expandafter
320     {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
321     {\romannumeral0\XINTinfloatmul [#6]{#5{#2}}{#1}}%
322     {#1}{#3}{#4}{#5}{#6}%
323 }%
324 \def\XINT_flpowseries_dont_i \fi\expandafter\XINT_flpowseries_loop_i
325     {\fi \expandafter\XINT_flpowseries_dont_ii }%
326 \def\XINT_flpowseries_dont_ii #1#2#3#4#5#6#7{\xintfloat [#7]{#2}}%
327 \def\XINT_flpowseries_loop_i #1#2#3#4#5#6#7%
328 {%
329     \ifnum #5>#1 \else \XINT_flpowseries_exit_i \fi
330     \expandafter\XINT_flpowseries_loop_ii\expandafter
331     {\romannumeral0\XINTinfloatmul [#7]{#3}{#4}}%
332     {#1}{#4}{#2}{#5}{#6}{#7}%
333 }%
334 \def\XINT_flpowseries_loop_ii #1#2#3#4#5#6#7%
335 {%
336     \expandafter\XINT_flpowseries_loop_i\expandafter
337     {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
338     {\romannumeral0\XINTinfloatadd [#7]{#4}%
339                           {\XINTinfloatmul [#7]{#6{#2}}{#1}}}%
```

```
    {#1}{#3}{#5}{#6}{#7}%
}%
\def\XINT_flpowseries_exit_i\fi\expandafter\XINT_flpowseries_loop_ii
    {\fi \expandafter\XINT_flpowseries_exit_ii }%
\def\XINT_flpowseries_exit_ii #1#2#3#4#5#6#7%
{%
    \xintfloatadd [#7]{#4}{\XINTinfloatmul [#7]{#6{#2}}{#1}}%
}%
```

9.12 \xintFloatPowerSeriesX

1.08a

```
\def\xintFloatPowerSeriesX {\romannumeral0\xintfloatpowerseriesx }%
\def\xintfloatpowerseriesx #1{\XINT_flpowseriesx_chkopt #1\xint:}%
\def\XINT_flpowseriesx_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_flpowseriesx_opt
       \else\expandafter\XINT_flpowseriesx_noopt
    \fi
    #1%
}%
\def\XINT_flpowseriesx_noopt  #1\xint:#2%
{%
    \expandafter\XINT_flpowseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter
    {\the\numexpr #2}\XINTdigits
}%
\def\XINT_flpowseriesx_opt [\xint:#1]#2#3%
{%
    \expandafter\XINT_flpowseriesx\expandafter
    {\the\numexpr #2\expandafter}\expandafter
    {\the\numexpr #3\expandafter}{\the\numexpr #1}%
}%
\def\XINT_flpowseriesx #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0.e0}%
   \else
      \xint_afterfi
        {\expandafter \XINT_flpowseriesx_pre \expandafter
         {\romannumeral`&&@#5}{#1}{#2}{#4}{#3}%
        }%
   \fi
}%
\def\XINT_flpowseriesx_pre #1#2#3#4#5%
{%
    \expandafter\XINT_flpowseries_loop_pre\expandafter
       {\romannumeral0\XINTinfloatpow [#5]{#1}{#2}}%
       {#2}{#1}{#3}{#4}{#5}%
}%
\XINT_restorecatcodes_endinput%
```

10 Package xintcfrac implementation

The commenting is currently (2021/03/29) very sparse. Release 1.09m (2014/02/26) has modified a few things: \xintFtoCs and \xintCntoCs insert spaces after the commas, \xintCstoF and \xintCstoCv authorize spaces in the input also before the commas, \xintCntoCs does not brace the produced coefficients, new macros \xintFtoC, \xintCtoF, \xintCtoCv, \xintFGtoC, and \xintGGCFrac.

There is partial dependency on xinttools due to \xintCstoF and \xintCsToCv.

10.1 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from Heiko Oberdiek's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintcfrac.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintcfrac}{\numexpr not available, aborting input}%
     \aftergroup\endinput
```

```
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintcfrac.sty
      \ifx\w\relax % but xintfrac.sty not yet loaded.
         \def\z{\endgroup\input xintfrac.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xintfrac.sty not yet loaded.
            \def\z{\endgroup\RequirePackage{xintfrac}}%
          \fi
      \else
        \aftergroup\endinput % xintcfrac already loaded.
      \fi
    \fi
  \fi
\z%
\XINTsetupcatcodes% defined in xintkernel.sty
```

10.2 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintcfrac}%
  [2021/03/29 v1.4d Expandable continued fractions with xint package (JFB)]%
```

10.3 \xintCFrac

```
\def\xintCFrac {\romannumeral0\xintcfrac }%
\def\xintcfrac #1%
{%
    \XINT_cfrac_opt_a #1\xint:
}%
\def\XINT_cfrac_opt_a #1%
{%
    \ifx[#1\XINT_cfrac_opt_b\fi \XINT_cfrac_noopt #1%
}%
\def\XINT_cfrac_noopt #1\xint:
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
    \relax\relax
}%
\def\XINT_cfrac_opt_b\fi\XINT_cfrac_noopt [\xint:#1]%
{%
    \fi\csname XINT_cfrac_opt#1\endcsname
}%
\def\XINT_cfrac_optl #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
    \relax\hfill
}%
\def\XINT_cfrac_optc #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
```

```
    \relax\relax
}%
\def\XINT_cfrac_optr #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
    \hfill\relax
}%
\def\XINT_cfrac_A #1/#2\Z
{%
    \expandafter\XINT_cfrac_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
}%
\def\XINT_cfrac_B #1#2%
{%
    \XINT_cfrac_C #2\Z {#1}%
}%
\def\XINT_cfrac_C #1%
{%
    \xint_gob_til_zero #1\XINT_cfrac_integer 0\XINT_cfrac_D #1%
}%
\def\XINT_cfrac_integer 0\XINT_cfrac_D 0#1\Z #2#3#4#5{ #2}%
\def\XINT_cfrac_D #1\Z #2#3{\XINT_cfrac_loop_a {#1}{#3}{#1}{{#2}}}%
\def\XINT_cfrac_loop_a
{%
    \expandafter\XINT_cfrac_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_cfrac_loop_d #1#2%
{%
    \XINT_cfrac_loop_e #2.{#1}%
}%
\def\XINT_cfrac_loop_e #1%
{%
    \xint_gob_til_zero #1\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1%
}%
\def\XINT_cfrac_loop_f #1.#2#3#4%
{%
    \XINT_cfrac_loop_a {#1}{#3}{#1}{{#2}#4}%
}%
\def\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1.#2#3#4#5#6%
   {\XINT_cfrac_T #5#6{#2}#4\Z }%
\def\XINT_cfrac_T #1#2#3#4%
{%
  \xint_gob_til_Z #4\XINT_cfrac_end\Z\XINT_cfrac_T #1#2{#4+\cfrac{#11#2}{#3}}%
}%
\def\XINT_cfrac_end\Z\XINT_cfrac_T #1#2#3%
{%
    \XINT_cfrac_end_b #3%
}%
\def\XINT_cfrac_end_b \Z+\cfrac#1#2{ #2}%
```

10.4 \xintGCFrac

```
\def\xintGCFrac {\romannumeral0\xintgcfrac }%
\def\xintgcfrac #1{\XINT_gcfrac_opt_a #1\xint:}%
\def\XINT_gcfrac_opt_a #1%
```

```
{%
    \ifx[#1\XINT_gcfrac_opt_b\fi \XINT_gcfrac_noopt #1%
}%
\def\XINT_gcfrac_noopt #1\xint:%
{%
    \XINT_gcfrac #1+!/\relax\relax
}%
\def\XINT_gcfrac_opt_b\fi\XINT_gcfrac_noopt [\xint:#1]%
{%
    \fi\csname XINT_gcfrac_opt#1\endcsname
}%
\def\XINT_gcfrac_optl #1%
{%
    \XINT_gcfrac #1+!/\relax\hfill
}%
\def\XINT_gcfrac_optc #1%
{%
    \XINT_gcfrac #1+!/\relax\relax
}%
\def\XINT_gcfrac_optr #1%
{%
    \XINT_gcfrac #1+!/\hfill\relax
}%
\def\XINT_gcfrac
{%
    \expandafter\XINT_gcfrac_enter\romannumeral`&&@%
}%
\def\XINT_gcfrac_enter {\XINT_gcfrac_loop {}}%
\def\XINT_gcfrac_loop #1#2+#3/%
{%
    \xint_gob_til_exclam #3\XINT_gcfrac_endloop!%
    \XINT_gcfrac_loop {{#3}{#2}#1}%
}%
\def\XINT_gcfrac_endloop!\XINT_gcfrac_loop #1#2#3%
{%
    \XINT_gcfrac_T #2#3#1!!%
}%
\def\XINT_gcfrac_T #1#2#3#4{\XINT_gcfrac_U #1#2{\xintFrac{#4}}}%
\def\XINT_gcfrac_U #1#2#3#4#5%
{%
    \xint_gob_til_exclam #5\XINT_gcfrac_end!\XINT_gcfrac_U
               #1#2{\xintFrac{#5}%
                \ifcase\xintSgn{#4}
                +\or+\else-\fi
                \cfrac{#1\xintFrac{\xintAbs{#4}}#2}{#3}}%
}%
\def\XINT_gcfrac_end!\XINT_gcfrac_U #1#2#3%
{%
    \XINT_gcfrac_end_b #3%
}%
\def\XINT_gcfrac_end_b #1\cfrac#2#3{ #3}%
```

10.5 \xintGGCFrac

New with 1.09m

```
\def\xintGGCFrac {\romannumeral0\xintggcfrac }%
\def\xintggcfrac #1{\XINT_ggcfrac_opt_a #1\xint:}%
\def\XINT_ggcfrac_opt_a #1%
{%
    \ifx[#1\XINT_ggcfrac_opt_b\fi \XINT_ggcfrac_noopt #1%
}%
\def\XINT_ggcfrac_noopt #1\xint:
{%
    \XINT_ggcfrac #1+!/\relax\relax
}%
\def\XINT_ggcfrac_opt_b\fi\XINT_ggcfrac_noopt [\xint:#1]%
{%
    \fi\csname XINT_ggcfrac_opt#1\endcsname
}%
\def\XINT_ggcfrac_optl #1%
{%
    \XINT_ggcfrac #1+!/\relax\hfill
}%
\def\XINT_ggcfrac_optc #1%
{%
    \XINT_ggcfrac #1+!/\relax\relax
}%
\def\XINT_ggcfrac_optr #1%
{%
    \XINT_ggcfrac #1+!/\hfill\relax
}%
\def\XINT_ggcfrac
{%
    \expandafter\XINT_ggcfrac_enter\romannumeral`&&@%
}%
\def\XINT_ggcfrac_enter {\XINT_ggcfrac_loop {}}%
\def\XINT_ggcfrac_loop #1#2+#3/%
{%
    \xint_gob_til_exclam #3\XINT_ggcfrac_endloop!%
    \XINT_ggcfrac_loop {{#3}{#2}#1}%
}%
\def\XINT_ggcfrac_endloop!\XINT_ggcfrac_loop #1#2#3%
{%
    \XINT_ggcfrac_T #2#3#1!!%
}%
\def\XINT_ggcfrac_T #1#2#3#4{\XINT_ggcfrac_U #1#2{#4}}%
\def\XINT_ggcfrac_U #1#2#3#4#5%
{%
    \xint_gob_til_exclam #5\XINT_ggcfrac_end!\XINT_ggcfrac_U
               #1#2{#5+\cfrac{#1#4#2}{#3}}%
}%
\def\XINT_ggcfrac_end!\XINT_ggcfrac_U #1#2#3%
{%
    \XINT_ggcfrac_end_b #3%
```

```
}%
\def\XINT_ggcfrac_end_b #1\cfrac#2#3{ #3}%
```

10.6 \xintGCtoGCx

```
\def\xintGCtoGCx {\romannumeral0\xintgctogcx }%
\def\xintgctogcx #1#2#3%
{%
    \expandafter\XINT_gctgcx_start\expandafter {\romannumeral`&&@#3}{#1}{#2}%
}%
\def\XINT_gctgcx_start #1#2#3{\XINT_gctgcx_loop_a {}{#2}{#3}#1+!/}%
\def\XINT_gctgcx_loop_a #1#2#3#4+#5/%
{%
    \xint_gob_til_exclam #5\XINT_gctgcx_end!%
    \XINT_gctgcx_loop_b {#1{#4}}{#2{#5}#3}{#2}{#3}%
}%
\def\XINT_gctgcx_loop_b #1#2%
{%
    \XINT_gctgcx_loop_a {#1#2}%
}%
\def\XINT_gctgcx_end!\XINT_gctgcx_loop_b #1#2#3#4{ #1}%
```

10.7 \xintFtoCs

Modified in 1.09m: a space is added after the inserted commas.

```
\def\xintFtoCs {\romannumeral0\xintftocs }%
\def\xintftocs #1%
{%
    \expandafter\XINT_ftc_A\romannumeral0\xintrawwithzeros {#1}\Z
}%
\def\XINT_ftc_A #1/#2\Z
{%
    \expandafter\XINT_ftc_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
}%
\def\XINT_ftc_B #1#2%
{%
    \XINT_ftc_C #2.{#1}%
}%
\def\XINT_ftc_C #1%
{%
    \xint_gob_til_zero #1\XINT_ftc_integer 0\XINT_ftc_D #1%
}%
\def\XINT_ftc_integer 0\XINT_ftc_D 0#1.#2#3{ #2}%
\def\XINT_ftc_D #1.#2#3{\XINT_ftc_loop_a {#1}{#3}{#1}{#2, }}% 1.09m adds a space
\def\XINT_ftc_loop_a
{%
    \expandafter\XINT_ftc_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_ftc_loop_d #1#2%
{%
    \XINT_ftc_loop_e #2.{#1}%
}%
\def\XINT_ftc_loop_e #1%
```

```
{%
    \xint_gob_til_zero #1\xint_ftc_loop_exit0\XINT_ftc_loop_f #1%
}%
\def\XINT_ftc_loop_f #1.#2#3#4%
{%
    \XINT_ftc_loop_a {#1}{#3}{#1}{#4#2, }% 1.09m has an added space here
}%
\def\xint_ftc_loop_exit0\XINT_ftc_loop_f #1.#2#3#4{ #4#2}%
```

10.8 \xintFtoCx

```
\def\xintFtoCx {\romannumeral0\xintftocx }%
\def\xintftocx #1#2%
{%
    \expandafter\XINT_ftcx_A\romannumeral0\xintrawwithzeros {#2}\Z {#1}%
}%
\def\XINT_ftcx_A #1/#2\Z
{%
    \expandafter\XINT_ftcx_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
}%
\def\XINT_ftcx_B #1#2%
{%
    \XINT_ftcx_C #2.{#1}%
}%
\def\XINT_ftcx_C #1%
{%
    \xint_gob_til_zero #1\XINT_ftcx_integer 0\XINT_ftcx_D #1%
}%
\def\XINT_ftcx_integer 0\XINT_ftcx_D 0#1.#2#3#4{ #2}%
\def\XINT_ftcx_D #1.#2#3#4{\XINT_ftcx_loop_a {#1}{#3}{#1}{{#2}#4}{#4}}%
\def\XINT_ftcx_loop_a
{%
    \expandafter\XINT_ftcx_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_ftcx_loop_d #1#2%
{%
    \XINT_ftcx_loop_e #2.{#1}%
}%
\def\XINT_ftcx_loop_e #1%
{%
    \xint_gob_til_zero #1\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1%
}%
\def\XINT_ftcx_loop_f #1.#2#3#4#5%
{%
    \XINT_ftcx_loop_a {#1}{#3}{#1}{#4{#2}#5}{#5}%
}%
\def\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1.#2#3#4#5{ #4{#2}}%
```

10.9 \xintFtoC

New in 1.09m: this is the same as \xintFtoCx with empty separator. I had temporarily during preparation of 1.09m removed braces from \xintFtoCx, but I recalled later why that was useful (see doc), thus let's just here do \xintFtoCx {}

```
\def\xintFtoC {\romannumeral0\xintftoc }%
\def\xintftoc {\xintftocx {}}%
```

10.10 \xintFtoGC

```
\def\xintFtoGC {\romannumeral0\xintftogc }%
\def\xintftogc {\xintftocx {+1/}}%
```

10.11 \xintFGtoC

New with 1.09m of 2014/02/26. Computes the common initial coefficients for the two fractions f and g, and outputs them as a sequence of braced items.

```
\def\xintFGtoC {\romannumeral0\xintfgtoc}%
\def\xintfgtoc#1%
{%
    \expandafter\XINT_fgtc_a\romannumeral0\xintrawwithzeros {#1}\Z
}%
\def\XINT_fgtc_a #1/#2\Z #3%
{%
    \expandafter\XINT_fgtc_b\romannumeral0\xintrawwithzeros {#3}\Z #1/#2\Z { }%
}%
\def\XINT_fgtc_b #1/#2\Z
{%
    \expandafter\XINT_fgtc_c\romannumeral0\xintiidivision {#1}{#2}{#2}%
}%
\def\XINT_fgtc_c #1#2#3#4/#5\Z
{%
    \expandafter\XINT_fgtc_d\romannumeral0\xintiidivision
                                          {#4}{#5}{#5}{#1}{#2}{#3}%
}%
\def\XINT_fgtc_d #1#2#3#4%#5#6#7%
{%
    \xintifEq {#1}{#4}{\XINT_fgtc_da {#1}{#2}{#3}{#4}}%
                      {\xint_thirdofthree}%
}%
\def\XINT_fgtc_da #1#2#3#4#5#6#7%
{%
     \XINT_fgtc_e {#2}{#5}{#3}{#6}{#7{#1}}%
}%
\def\XINT_fgtc_e #1%
{%
    \xintiiifZero {#1}{\expandafter\xint_firstofone\xint_gobble_iii}%
                     {\XINT_fgtc_f {#1}}%
}%
\def\XINT_fgtc_f #1#2%
{%
   \xintiiifZero {#2}{\xint_thirdofthree}{\XINT_fgtc_g {#1}{#2}}%
}%
\def\XINT_fgtc_g #1#2#3%
{%
    \expandafter\XINT_fgtc_h\romannumeral0\XINT_div_prepare {#1}{#3}{#1}{#2}%
}%
\def\XINT_fgtc_h #1#2#3#4#5%
```

```
{%
    \expandafter\XINT_fgtc_d\romannumeral0\XINT_div_prepare
                         {#4}{#5}{#4}{#1}{#2}{#3}%
}%
```

10.12 \xintFtoCC

```
\def\xintFtoCC {\romannumeral0\xintftocc }%
\def\xintftocc #1%
{%
    \expandafter\XINT_ftcc_A\expandafter {\romannumeral0\xintrawwithzeros {#1}}%
}%
\def\XINT_ftcc_A #1%
{%
    \expandafter\XINT_ftcc_B
    \romannumeral0\xintrawwithzeros {\xintAdd {1/2[0]}{#1[0]}}\Z {#1[0]}%
}%
\def\XINT_ftcc_B #1/#2\Z
{%
    \expandafter\XINT_ftcc_C\expandafter {\romannumeral0\xintiiquo {#1}{#2}}%
}%
\def\XINT_ftcc_C #1#2%
{%
    \expandafter\XINT_ftcc_D\romannumeral0\xintsub {#2}{#1}\Z {#1}%
}%
\def\XINT_ftcc_D #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_ftcc_integer
      0#1\XINT_ftcc_En
       0-{\XINT_ftcc_Ep #1}%
    \krof
}%
\def\XINT_ftcc_Ep #1\Z #2%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#2+1/}%
}%
\def\XINT_ftcc_En #1\Z #2%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#2+-1/}%
}%
\def\XINT_ftcc_integer #1\Z #2{ #2}%
\def\XINT_ftcc_loop_a #1%
{%
    \expandafter\XINT_ftcc_loop_b
    \romannumeral0\xintrawwithzeros {\xintAdd {1/2[0]}{#1}}\Z {#1}%
}%
\def\XINT_ftcc_loop_b #1/#2\Z
{%
    \expandafter\XINT_ftcc_loop_c\expandafter
    {\romannumeral0\xintiiquo {#1}{#2}}%
```

```
}%
\def\XINT_ftcc_loop_c #1#2%
{%
    \expandafter\XINT_ftcc_loop_d
    \romannumeral0\xintsub {#2}{#1[0]}\Z {#1}%
}%
\def\XINT_ftcc_loop_d #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_ftcc_end
      0#1\XINT_ftcc_loop_N
       0-{\XINT_ftcc_loop_P #1}%
    \krof
}%
\def\XINT_ftcc_end #1\Z #2#3{ #3#2}%
\def\XINT_ftcc_loop_P #1\Z #2#3%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#3#2+1/}%
}%
\def\XINT_ftcc_loop_N #1\Z #2#3%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#3#2+-1/}%
}%
```

10.13 \xintCtoF, \xintCstoF

1.09m uses \xintCSVtoList on the argument of \xintCstoF to allow spaces also before the commas. And the original \xintCstoF code became the one of the new \xintCtoF dealing with a braced rather than comma separated list.

```
\def\xintCstoF {\romannumeral0\xintcstof }%
\def\xintcstof #1%
{%
    \expandafter\XINT_ctf_prep \romannumeral0\xintcsvtolist{#1}!%
}%
\def\xintCtoF {\romannumeral0\xintctof }%
\def\xintctof #1%
{%
    \expandafter\XINT_ctf_prep \romannumeral`&&@#1!%
}%
\def\XINT_ctf_prep
{%
    \XINT_ctf_loop_a 1001%
}%
\def\XINT_ctf_loop_a #1#2#3#4#5%
{%
    \xint_gob_til_exclam #5\XINT_ctf_end!%
    \expandafter\XINT_ctf_loop_b
    \romannumeral0\xintrawwithzeros {#5}.{#1}{#2}{#3}{#4}%
}%
\def\XINT_ctf_loop_b #1/#2.#3#4#5#6%
{%
```

```
    \expandafter\XINT_ctf_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
    {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
    {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#6\xint:}%
                              {\XINT_mul_fork #1\xint:#4\xint:}}%
    {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#5\xint:}%
                              {\XINT_mul_fork #1\xint:#3\xint:}}%
}%
\def\XINT_ctf_loop_c #1#2%
{%
    \expandafter\XINT_ctf_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_ctf_loop_d #1#2%
{%
    \expandafter\XINT_ctf_loop_e\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_ctf_loop_e #1#2%
{%
    \expandafter\XINT_ctf_loop_a\expandafter{#2}#1%
}%
\def\XINT_ctf_end #1.#2#3#4#5{\xintrawwithzeros {#2/#3}}% 1.09b removes [0]
```

10.14 \xintiCstoF

```
\def\xintiCstoF {\romannumeral0\xinticstof }%
\def\xinticstof #1%
{%
    \expandafter\XINT_icstf_prep \romannumeral`&&@#1,!,%
}%
\def\XINT_icstf_prep
{%
    \XINT_icstf_loop_a 1001%
}%
\def\XINT_icstf_loop_a #1#2#3#4#5,%
{%
    \xint_gob_til_exclam #5\XINT_icstf_end!%
    \expandafter
    \XINT_icstf_loop_b \romannumeral`&&@#5.{#1}{#2}{#3}{#4}%
}%
\def\XINT_icstf_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_icstf_loop_c\expandafter
    {\romannumeral0\xintiiadd {#5}{\XINT_mul_fork #1\xint:#3\xint:}}%
    {\romannumeral0\xintiiadd {#4}{\XINT_mul_fork #1\xint:#2\xint:}}%
    {#2}{#3}%
}%
\def\XINT_icstf_loop_c #1#2%
{%
    \expandafter\XINT_icstf_loop_a\expandafter {#2}{#1}%
}%
\def\XINT_icstf_end#1.#2#3#4#5{\xintrawwithzeros {#2/#3}}% 1.09b removes [0]
```

10.15 \xintGCtoF

```
\def\xintGCtoF {\romannumeral0\xintgctof }%
\def\xintgctof #1%
{%
    \expandafter\XINT_gctf_prep \romannumeral`&&@#1+!/%
}%
\def\XINT_gctf_prep
{%
    \XINT_gctf_loop_a 1001%
}%
\def\XINT_gctf_loop_a #1#2#3#4#5+%
{%
    \expandafter\XINT_gctf_loop_b
    \romannumeral0\xintrawwithzeros {#5}.{#1}{#2}{#3}{#4}%
}%
\def\XINT_gctf_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctf_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
    {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
    {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#6\xint:}%
                              {\XINT_mul_fork #1\xint:#4\xint:}}%
    {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#5\xint:}%
                              {\XINT_mul_fork #1\xint:#3\xint:}}%
}%
\def\XINT_gctf_loop_c #1#2%
{%
    \expandafter\XINT_gctf_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctf_loop_d #1#2%
{%
    \expandafter\XINT_gctf_loop_e\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctf_loop_e #1#2%
{%
    \expandafter\XINT_gctf_loop_f\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctf_loop_f #1#2/%
{%
    \xint_gob_til_exclam #2\XINT_gctf_end!%
    \expandafter\XINT_gctf_loop_g
    \romannumeral0\xintrawwithzeros {#2}.#1%
}%
\def\XINT_gctf_loop_g #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctf_loop_h\expandafter
    {\romannumeral0\XINT_mul_fork #1\xint:#6\xint:}%
    {\romannumeral0\XINT_mul_fork #1\xint:#5\xint:}%
    {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
    {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
}%
\def\XINT_gctf_loop_h #1#2%
{%
```

```
    \expandafter\XINT_gctf_loop_i\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctf_loop_i #1#2%
{%
    \expandafter\XINT_gctf_loop_j\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctf_loop_j #1#2%
{%
    \expandafter\XINT_gctf_loop_a\expandafter {#2}#1%
}%
\def\XINT_gctf_end #1.#2#3#4#5{\xintrawwithzeros {#2/#3}}% 1.09b removes [0]
```

10.16 \xintiGCtoF

```
\def\xintiGCtoF {\romannumeral0\xintigctof }%
\def\xintigctof #1%
{%
    \expandafter\XINT_igctf_prep \romannumeral`&&@#1+!/%
}%
\def\XINT_igctf_prep
{%
    \XINT_igctf_loop_a 1001%
}%
\def\XINT_igctf_loop_a #1#2#3#4#5+%
{%
    \expandafter\XINT_igctf_loop_b
    \romannumeral`&&@#5.{#1}{#2}{#3}{#4}%
}%
\def\XINT_igctf_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_igctf_loop_c\expandafter
    {\romannumeral0\xintiiadd {#5}{\XINT_mul_fork #1\xint:#3\xint:}}%
    {\romannumeral0\xintiiadd {#4}{\XINT_mul_fork #1\xint:#2\xint:}}%
    {#2}{#3}%
}%
\def\XINT_igctf_loop_c #1#2%
{%
    \expandafter\XINT_igctf_loop_f\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctf_loop_f #1#2#3#4/%
{%
    \xint_gob_til_exclam #4\XINT_igctf_end!%
    \expandafter\XINT_igctf_loop_g
    \romannumeral`&&@#4.{#2}{#3}#1%
}%
\def\XINT_igctf_loop_g #1.#2#3%
{%
    \expandafter\XINT_igctf_loop_h\expandafter
    {\romannumeral0\XINT_mul_fork #1\xint:#3\xint:}%
    {\romannumeral0\XINT_mul_fork #1\xint:#2\xint:}%
}%
\def\XINT_igctf_loop_h #1#2%
{%
    \expandafter\XINT_igctf_loop_i\expandafter {#2}{#1}%
```

```
}%
\def\XINT_igctf_loop_i #1#2#3#4%
{%
    \XINT_igctf_loop_a {#3}{#4}{#1}{#2}%
}%
\def\XINT_igctf_end #1.#2#3#4#5{\xintrawwithzeros {#4/#5}}% 1.09b removes [0]
```

10.17 \xintCtoCv, \xintCstoCv

1.09m uses \xintCSVtoList on the argument of \xintCstoCv to allow spaces also before the commas. The original \xintCstoCv code became the one of the new \xintCtoF dealing with a braced rather than comma separated list.

```
\def\xintCstoCv {\romannumeral0\xintcstocv }%
\def\xintcstocv #1%
{%
    \expandafter\XINT_ctcv_prep\romannumeral0\xintcsvtolist{#1}!%
}%
\def\xintCtoCv {\romannumeral0\xintctocv }%
\def\xintctocv #1%
{%
    \expandafter\XINT_ctcv_prep\romannumeral`&&@#1!%
}%
\def\XINT_ctcv_prep
{%
    \XINT_ctcv_loop_a {}1001%
}%
\def\XINT_ctcv_loop_a #1#2#3#4#5#6%
{%
    \xint_gob_til_exclam #6\XINT_ctcv_end!%
    \expandafter\XINT_ctcv_loop_b
    \romannumeral0\xintrawwithzeros {#6}.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_ctcv_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_ctcv_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
    {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
    {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#6\xint:}%
                              {\XINT_mul_fork #1\xint:#4\xint:}}%
    {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#5\xint:}%
                              {\XINT_mul_fork #1\xint:#3\xint:}}%
}%
\def\XINT_ctcv_loop_c #1#2%
{%
    \expandafter\XINT_ctcv_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_ctcv_loop_d #1#2%
{%
    \expandafter\XINT_ctcv_loop_e\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_ctcv_loop_e #1#2%
{%
    \expandafter\XINT_ctcv_loop_f\expandafter{#2}#1%
```

```
}%
\def\XINT_ctcv_loop_f #1#2#3#4#5%
{%
    \expandafter\XINT_ctcv_loop_g\expandafter
    {\romannumeral0\xintrawwithzeros {#1/#2}}{#5}{#1}{#2}{#3}{#4}%
}%
\def\XINT_ctcv_loop_g #1#2{\XINT_ctcv_loop_a {#2{#1}}}% 1.09b removes [0]
\def\XINT_ctcv_end #1.#2#3#4#5#6{ #6}%
```

10.18 \xintiCstoCv

```
\def\xintiCstoCv {\romannumeral0\xinticstocv }%
\def\xinticstocv #1%
{%
    \expandafter\XINT_icstcv_prep \romannumeral`&&@#1,!,%
}%
\def\XINT_icstcv_prep
{%
    \XINT_icstcv_loop_a {}1001%
}%
\def\XINT_icstcv_loop_a #1#2#3#4#5#6,%
{%
    \xint_gob_til_exclam #6\XINT_icstcv_end!%
    \expandafter
    \XINT_icstcv_loop_b \romannumeral`&&@#6.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_icstcv_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_icstcv_loop_c\expandafter
    {\romannumeral0\xintiiadd {#5}{\XINT_mul_fork #1\xint:#3\xint:}}%
    {\romannumeral0\xintiiadd {#4}{\XINT_mul_fork #1\xint:#2\xint:}}%
    {{#2}{#3}}%
}%
\def\XINT_icstcv_loop_c #1#2%
{%
    \expandafter\XINT_icstcv_loop_d\expandafter {#2}{#1}%
}%
\def\XINT_icstcv_loop_d #1#2%
{%
    \expandafter\XINT_icstcv_loop_e\expandafter
    {\romannumeral0\xintrawwithzeros {#1/#2}}{{#1}{#2}}%
}%
\def\XINT_icstcv_loop_e #1#2#3#4{\XINT_icstcv_loop_a {#4{#1}}#2#3}%
\def\XINT_icstcv_end #1.#2#3#4#5#6{ #6}%  1.09b removes [0]
```

10.19 \xintGCtoCv

```
\def\xintGCtoCv {\romannumeral0\xintgctocv }%
\def\xintgctocv #1%
{%
    \expandafter\XINT_gctcv_prep \romannumeral`&&@#1+!/%
}%
\def\XINT_gctcv_prep
{%
```

```
    \XINT_gctcv_loop_a {}1001%
}%
\def\XINT_gctcv_loop_a #1#2#3#4#5#6+%
{%
    \expandafter\XINT_gctcv_loop_b
    \romannumeral0\xintrawwithzeros {#6}.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_gctcv_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctcv_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
    {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
    {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#6\xint:}%
                              {\XINT_mul_fork #1\xint:#4\xint:}}%
    {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#5\xint:}%
                              {\XINT_mul_fork #1\xint:#3\xint:}}%
}%
\def\XINT_gctcv_loop_c #1#2%
{%
    \expandafter\XINT_gctcv_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctcv_loop_d #1#2%
{%
    \expandafter\XINT_gctcv_loop_e\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctcv_loop_e #1#2%
{%
    \expandafter\XINT_gctcv_loop_f\expandafter {#2}#1%
}%
\def\XINT_gctcv_loop_f #1#2%
{%
    \expandafter\XINT_gctcv_loop_g\expandafter
    {\romannumeral0\xintrawwithzeros {#1/#2}}{{#1}{#2}}%
}%
\def\XINT_gctcv_loop_g #1#2#3#4%
{%
    \XINT_gctcv_loop_h {#4{#1}}{#2#3}% 1.09b removes [0]
}%
\def\XINT_gctcv_loop_h #1#2#3/%
{%
    \xint_gob_til_exclam #3\XINT_gctcv_end!%
    \expandafter\XINT_gctcv_loop_i
    \romannumeral0\xintrawwithzeros {#3}.#2{#1}%
}%
\def\XINT_gctcv_loop_i #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctcv_loop_j\expandafter
    {\romannumeral0\XINT_mul_fork #1\xint:#6\xint:}%
    {\romannumeral0\XINT_mul_fork #1\xint:#5\xint:}%
    {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
    {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
}%
```

```
\def\XINT_gctcv_loop_j #1#2%
{%
    \expandafter\XINT_gctcv_loop_k\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctcv_loop_k #1#2%
{%
    \expandafter\XINT_gctcv_loop_l\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctcv_loop_l #1#2%
{%
    \expandafter\XINT_gctcv_loop_m\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctcv_loop_m #1#2{\XINT_gctcv_loop_a {#2}#1}%
\def\XINT_gctcv_end #1.#2#3#4#5#6{ #6}%
```

10.20 \xintiGCtoCv

```
\def\xintiGCtoCv {\romannumeral0\xintigctocv }%
\def\xintigctocv #1%
{%
    \expandafter\XINT_igctcv_prep \romannumeral`&&@#1+!/%
}%
\def\XINT_igctcv_prep
{%
    \XINT_igctcv_loop_a {}1001%
}%
\def\XINT_igctcv_loop_a #1#2#3#4#5#6+%
{%
    \expandafter\XINT_igctcv_loop_b
    \romannumeral`&&@#6.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_igctcv_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_loop_c\expandafter
    {\romannumeral0\xintiiadd {#5}{\XINT_mul_fork #1\xint:#3\xint:}}%
    {\romannumeral0\xintiiadd {#4}{\XINT_mul_fork #1\xint:#2\xint:}}%
    {{#2}{#3}}%
}%
\def\XINT_igctcv_loop_c #1#2%
{%
    \expandafter\XINT_igctcv_loop_f\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctcv_loop_f #1#2#3#4/%
{%
    \xint_gob_til_exclam #4\XINT_igctcv_end_a!%
    \expandafter\XINT_igctcv_loop_g
    \romannumeral`&&@#4.#1#2{#3}%
}%
\def\XINT_igctcv_loop_g #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_loop_h\expandafter
    {\romannumeral0\XINT_mul_fork #1\xint:#5\xint:}%
    {\romannumeral0\XINT_mul_fork #1\xint:#4\xint:}%
    {{#2}{#3}}%
```

```
}%
\def\XINT_igctcv_loop_h #1#2%
{%
    \expandafter\XINT_igctcv_loop_i\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctcv_loop_i #1#2{\XINT_igctcv_loop_k #2{#2#1}}%
\def\XINT_igctcv_loop_k #1#2%
{%
    \expandafter\XINT_igctcv_loop_l\expandafter
    {\romannumeral0\xintrawwithzeros {#1/#2}}%
}%
\def\XINT_igctcv_loop_l #1#2#3{\XINT_igctcv_loop_a {#3{#1}}#2}%1.09i removes [0]
\def\XINT_igctcv_end_a #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_end_b\expandafter
    {\romannumeral0\xintrawwithzeros {#2/#3}}%
}%
\def\XINT_igctcv_end_b #1#2{ #2{#1}}% 1.09b removes [0]
```

10.21 \xintFtoCv

Still uses \xinticstocv \xintFtoCs rather than \xintctocv \xintFtoC.

```
\def\xintFtoCv {\romannumeral0\xintftocv }%
\def\xintftocv #1%
{%
    \xinticstocv {\xintFtoCs {#1}}%
}%
```

10.22 \xintFtoCCv

```
\def\xintFtoCCv {\romannumeral0\xintftoccv }%
\def\xintftoccv #1%
{%
    \xintigctocv {\xintFtoCC {#1}}%
}%
```

10.23 \xintCntoF

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintCntoF {\romannumeral0\xintcntof }%
\def\xintcntof #1%
{%
    \expandafter\XINT_cntf\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntf #1#2%
{%
   \ifnum #1>\xint_c_
      \xint_afterfi {\expandafter\XINT_cntf_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\romannumeral`&&@#2{#1}}{#2}}%
   \else
```

```
      \xint_afterfi
         {\ifnum #1=\xint_c_
              \xint_afterfi {\expandafter\space \romannumeral`&&@#2{0}}%
          \else \xint_afterfi { }% 1.09m now returns nothing.
          \fi}%
   \fi
}%
\def\XINT_cntf_loop #1#2#3%
{%
    \ifnum #1>\xint_c_ \else \XINT_cntf_exit \fi
    \expandafter\XINT_cntf_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\romannumeral0\xintadd {\xintDiv {1[0]}{#2}}{#3{#1}}}%
    {#3}%
}%
\def\XINT_cntf_exit \fi
    \expandafter\XINT_cntf_loop\expandafter
    #1\expandafter #2#3%
{%
    \fi\xint_gobble_ii #2%
}%
```

10.24 \xintGCntoF

Modified in 1.06 to give the N argument first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintGCntoF {\romannumeral0\xintgcntof }%
\def\xintgcntof #1%
{%
    \expandafter\XINT_gcntf\expandafter {\the\numexpr #1}%
}%
\def\XINT_gcntf #1#2#3%
{%
   \ifnum #1>\xint_c_
      \xint_afterfi {\expandafter\XINT_gcntf_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\romannumeral`&&@#2{#1}}{#2}{#3}}%
   \else
      \xint_afterfi
         {\ifnum #1=\xint_c_
              \xint_afterfi {\expandafter\space\romannumeral`&&@#2{0}}%
          \else \xint_afterfi { }% 1.09m now returns nothing rather than 0/1[0]
          \fi}%
   \fi
}%
\def\XINT_gcntf_loop #1#2#3#4%
{%
    \ifnum #1>\xint_c_ \else \XINT_gcntf_exit \fi
    \expandafter\XINT_gcntf_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\romannumeral0\xintadd {\xintDiv {#4{#1}}{#2}}{#3{#1}}}%
    {#3}{#4}%
```

```
}%
\def\XINT_gcntf_exit \fi
    \expandafter\XINT_gcntf_loop\expandafter
    #1\expandafter #2#3#4%
{%
    \fi\xint_gobble_ii #2%
}%
```

10.25 \xintCntoCs

Modified in 1.09m: added spaces after the commas in the produced list. Moreover the coefficients are not braced anymore. A slight induced limitation is that the macro argument should not contain some explicit comma (cf. \XINT_cntcs_exit_b), hence \xintCntoCs {\macro,} with \def\macro,#1{<stuff>} would crash. Not a very serious limitation, I believe.

```
\def\xintCntoCs {\romannumeral0\xintcntocs }%
\def\xintcntocs #1%
{%
    \expandafter\XINT_cntcs\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntcs #1#2%
{%
   \ifnum #1<0
      \xint_afterfi { }% 1.09i: a 0/1[0] was here, now the macro returns nothing
   \else
      \xint_afterfi {\expandafter\XINT_cntcs_loop\expandafter
                     {\the\numexpr #1-\xint_c_i\expandafter}\expandafter
                     {\romannumeral`&&@#2{#1}}{#2}}% produced coeff not braced
   \fi
}%
\def\XINT_cntcs_loop #1#2#3%
{%
    \ifnum #1>-\xint_c_i \else \XINT_cntcs_exit \fi
    \expandafter\XINT_cntcs_loop\expandafter
    {\the\numexpr #1-\xint_c_i\expandafter}\expandafter
    {\romannumeral`&&@#3{#1}, #2}{#3}% space added, 1.09m
}%
\def\XINT_cntcs_exit \fi
    \expandafter\XINT_cntcs_loop\expandafter
    #1\expandafter #2#3%
{%
    \fi\XINT_cntcs_exit_b #2%
}%
\def\XINT_cntcs_exit_b #1,{}% romannumeral stopping space already there
```

10.26 \xintCntoGC

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

1.09m maintains the braces, as the coeff are allowed to be fraction and the slash can not be naked in the GC format, contrarily to what happens in \xintCntoCs. Also the separators given to \xintGCtoGCx may then fetch the coefficients as argument, as they are braced.

```
\def\xintCntoGC {\romannumeral0\xintcntogc }%
\def\xintcntogc #1%
{%
    \expandafter\XINT_cntgc\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntgc #1#2%
{%
   \ifnum #1<0
      \xint_afterfi { }% 1.09i there was as strange 0/1[0] here, removed
   \else
      \xint_afterfi {\expandafter\XINT_cntgc_loop\expandafter
                     {\the\numexpr #1-\xint_c_i\expandafter}\expandafter
                     {\expandafter{\romannumeral`&&@#2{#1}}}{#2}}%
   \fi
}%
\def\XINT_cntgc_loop #1#2#3%
{%
    \ifnum #1>-\xint_c_i \else \XINT_cntgc_exit \fi
    \expandafter\XINT_cntgc_loop\expandafter
    {\the\numexpr #1-\xint_c_i\expandafter }\expandafter
    {\expandafter{\romannumeral`&&@#3{#1}}+1/#2}{#3}%
}%
\def\XINT_cntgc_exit \fi
    \expandafter\XINT_cntgc_loop\expandafter
    #1\expandafter #2#3%
{%
    \fi\XINT_cntgc_exit_b #2%
}%
\def\XINT_cntgc_exit_b #1+1/{ }%
```

10.27 \xintGCntoGC

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintGCntoGC {\romannumeral0\xintgcntogc }%
\def\xintgcntogc #1%
{%
    \expandafter\XINT_gcntgc\expandafter {\the\numexpr #1}%
}%
\def\XINT_gcntgc #1#2#3%
{%
   \ifnum #1<0
      \xint_afterfi { }% 1.09i now returns nothing
   \else
      \xint_afterfi {\expandafter\XINT_gcntgc_loop\expandafter
                     {\the\numexpr #1-\xint_c_i\expandafter}\expandafter
                     {\expandafter{\romannumeral`&&@#2{#1}}}{#2}{#3}}%
   \fi
}%
\def\XINT_gcntgc_loop #1#2#3#4%
{%
    \ifnum #1>-\xint_c_i \else \XINT_gcntgc_exit \fi
```

```
    \expandafter\XINT_gcntgc_loop_b\expandafter
    {\expandafter{\romannumeral`&&@#4{#1}}/#2}{#3{#1}}{#1}{#3}{#4}%
}%
\def\XINT_gcntgc_loop_b #1#2#3%
{%
    \expandafter\XINT_gcntgc_loop\expandafter
    {\the\numexpr #3-\xint_c_i \expandafter}\expandafter
    {\expandafter{\romannumeral`&&@#2}+#1}%
}%
\def\XINT_gcntgc_exit \fi
    \expandafter\XINT_gcntgc_loop_b\expandafter #1#2#3#4#5%
{%
    \fi\XINT_gcntgc_exit_b #1%
}%
\def\XINT_gcntgc_exit_b #1/{ }%
```

10.28 \xintCstoGC

```
\def\xintCstoGC {\romannumeral0\xintcstogc }%
\def\xintcstogc #1%
{%
    \expandafter\XINT_cstc_prep \romannumeral`&&@#1,!,%
}%
\def\XINT_cstc_prep #1,{\XINT_cstc_loop_a {{#1}}}%
\def\XINT_cstc_loop_a #1#2,%
{%
    \xint_gob_til_exclam #2\XINT_cstc_end!%
    \XINT_cstc_loop_b {#1}{#2}%
}%
\def\XINT_cstc_loop_b #1#2{\XINT_cstc_loop_a {#1+1/{#2}}}%
\def\XINT_cstc_end!\XINT_cstc_loop_b #1#2{ #1}%
```

10.29 \xintGCtoGC

```
\def\xintGCtoGC {\romannumeral0\xintgctogc }%
\def\xintgctogc #1%
{%
    \expandafter\XINT_gctgc_start \romannumeral`&&@#1+!/%
}%
\def\XINT_gctgc_start {\XINT_gctgc_loop_a {}}%
\def\XINT_gctgc_loop_a #1#2+#3/%
{%
    \xint_gob_til_exclam #3\XINT_gctgc_end!%
    \expandafter\XINT_gctgc_loop_b\expandafter
    {\romannumeral`&&@#2}{#3}{#1}%
}%
\def\XINT_gctgc_loop_b #1#2%
{%
    \expandafter\XINT_gctgc_loop_c\expandafter
    {\romannumeral`&&@#2}{#1}%
}%
\def\XINT_gctgc_loop_c #1#2#3%
{%
    \XINT_gctgc_loop_a {#3{#2}+{#1}/}%
```

```
}%
\def\XINT_gctgc_end!\expandafter\XINT_gctgc_loop_b
{%
    \expandafter\XINT_gctgc_end_b
}%
\def\XINT_gctgc_end_b #1#2#3{ #3{#1}}%
\XINT_restorecatcodes_endinput%
```

11 Package xintexpr implementation

This is release 1.4d of 2021/03/29.

Contents

11.1 READ ME! Important warnings and explanations relative to the status of the code source at the time of the 1.4 release

At release 1.4 the csname encapsulation of intermediate evaluations during parsing of expressions is dropped, and xintexpr requires the \expanded primitive. This means that there is no more impact on the string pool. And as internal storage now uses simply core \TeX{} syntax with braces rather than comma separated items inside a csname dummy control sequence, it became much easier to let the [...] syntax be associated to a true internal type of «tuple» or «list».

The output of \xintexpr (after \romannumeral0 or \romannumeral-`0 triggered expansion or double expansion) is thus modified at 1.4. It now looks like this:

\XINTfstop \XINTexprprint .{{<number>}} in simplest case

\XINTfstop \XINTexprprint .{{...}...{...}} in general case

where ... stands for nested braces ultimately ending in {<num. rep.>} leaves. The <num. rep.> stands for some internal representation of numeric data. It may be empty, and currently as well as probably in future uses only catcode 12 tokens (no spaces currently).

{{}} corresponds (in input as in output) to []. The external TeX braces also serve as set-theoretical braces. The comma is concatenation, so for example [], [] will become {{}{}}, or rather {}{} if sub-unit of something else.

The associated vocabulary is explained in the user manual and we avoid too much duplication here. xintfrac numerical macros receiving an empty argument usually handle it as being 0, but this is not the case of the xintcore macros supporting \xintiiexpr, they usually break if exercised on some empty argument.

The above expansion result \XINTfstop \XINTexprprint .{{<num1>}{<num2}...} uses only normal catcodes: the backslash, regular braces, and catcode 12 characters. Scientific notation is internally converted to raw xintfrac representation [N].

Additional data may be located before the dot; this is the case only for \xintfloatexpr currently. As xintexpr actually defines three parsers \xintexpr, \xintiiexpr and \xintfloatexpr but tries to share as much code as possible, some overhead is induced to fit all into the same mold.

\XINTfstop stops \romannumeral-`0 (or 0) type spanned expansion, and is invariant under \edef, but simply disappears in typesetting context. It is thus now legal to use \xintexpr directly in typesetting flow.

\XINTexprprint is \protected.

The f-expansion of an \xintexpr <expression>\relax is a complete expansion, i.e. one whose result remains invariant under \edef. But if exposed to finitely many expansion steps (at least two) there is a «blinking» \noexpand upfront depending on parity of number of steps.

\xintthe\xintexpr <expression>\relax or \xinteval{<expression>} serve as formerly to deliver the explicit digits, or more exactly some prettifying view of the actual <internal number representation>. For example \xintthe\xintboolexpr will (this is tentative) use True and False in output.

Nested contents like this

```
{{1}{{2}{3}{{4}{5}{6}}}{9}}
```

will get delivered using nested square brackets like that

```
1, [2, 3, [4, 5, 6]], 9
```

and as conversely \xintexpr 1, [2, 3, [4, 5, 6]], 9\relax expands to

```
\XINTfstop \XINTexprprint .{{1}{{2}{3}{{4}{5}{6}}}{9}}
```

we obtain the gratifying result that

```
  \xinteval{1, [2, 3, [4, 5, 6]], 9}
expands to
  1, [2, 3, [4, 5, 6]], 9
```
See user manual for explanations on the plasticity of \xintexpr syntax regarding functions with multiple arguments, and the 1.4 «unpacking» Python-like * prefix operator.

I have suppressed (from the public dtx) many big chunks of comments. Some became obsolete and need to be updated, others are currently of value only to the author as a historical record.

ATTENTION! As the removal process itself took too much time, I ended up leaving as is many comments which are obsoleted and wrong to various degrees after the 1.4 release. Precedence levels of operators have all been doubled to make room for new constructs

Even comments added during 1.4 developement may now be obsolete because the preparation of 1.4 took a few weeks and that's enough of duration to provide the author many chances to contradict in the code what has been already commented upon.

Thus don't believe (fully) anything which is said here!

Warning: in text below and also in left-over old comments I may refer to «until» and «op» macros; due to the change of data storage at 1.4, I needed to refactor a bit the way expansion is controlled, and the situation now is mainly governed by «op», «exec», «check-» and «checkp» macros the latter three replacing the two «until_a» and «until_b» of former code. This allows to diminish the number of times an accumulated result will be grabbed in order to propagate expansion to its right. Formerly this was not an issue because such things were only a single token! I do not describe here how this is all articulated but it is not hard to see it from the code (the hardest thing in all such matter was in 2013 to actually write how the expansion would be initially launched because to do that one basically has to understand the mechanism in its whole and such things are not easy to develop piecemeal). Another thing to keep in mind is that operators in truth have a left precedence (i.e. the precedence they show to operators arising earlier) and a right precedence (which determines how they react to operators coming after them from the right). Only the first one is usually encapsulated in a chardef, the second one is most of the times identical to the first one and if not it is only virtual but implemented via \ifcase of \ifnum branching. A final remark is that some things are achieved by special «op» macros, which are a favorite tool to hack into the normal regular flow of things, via injection of special syntax elements. I did not rename these macros for avoiding too large git diffs, and besides the nice thing is that the 1.4 refactoring minimally had to modify them, and all hacky things using them kept on working with not a single modification. And a post-scriptum is that advanced features crucially exploit injecting sub-\xintexpr-essions, as all is expandable there is no real «context» (only a minimal one) which one would have to perhaps store and restore and doing this sub-expression injection is rather cheap and efficient operation.

11.2 Old comments

These general comments were last updated at the end of the 1.09x series in 2014. The principles remain in place to this day but refer to CHANGES.html for some significant evolutions since.

The first version was released in June 2013. I was greatly helped in this task of writing an expandable parser of infix operations by the comments provided in l3fp-parse.dtx (in its version as available in April-May 2013). One will recognize in particular the idea of the `until' macros; I have not looked into the actual l3fp code beyond the very useful comments provided in its documentation.

A main worry was that my data has no a priori bound on its size; to keep the code reasonably efficient, I experimented with a technique of storing and retrieving data expandably as *names* of control sequences. Intermediate computation results are stored as control sequences \.=a/b[n].

Roughly speaking, the parser mechanism is as follows: at any given time the last found ``operator'' has its associated until macro awaiting some news from the token flow; first getnext expands forward in the hope to construct some number, which may come from a parenthesized sub-expression, from some braced material, or from a digit by digit scan. After this number has been formed the next operator is looked for by the getop macro. Once getop has finished its job, until is presented with three tokens: the first one is the precedence level of the new found operator (which may be an end of expression marker), the second is the operator character token (earlier versions had here already some macro name, but in order to keep as much common code to expr and floatexpr common as possible, this was modified) of the new found operator, and the third one is the newly found number (which was encountered just before the new operator).

The until macro of the earlier operator examines the precedence level of the new found one, and either executes the earlier operator (in the case of a binary operation, with the found number and a previously stored one) or it delays execution, giving the hand to the until macro of the operator having been found of higher precedence.

A minus sign acting as prefix gets converted into a (unary) operator inheriting the precedence level of the previous operator.

Once the end of the expression is found (it has to be marked by a \relax) the final result is output as four tokens (five tokens since 1.09j) the first one a catcode 11 exclamation mark, the second one an error generating macro, the third one is a protection mechanism, the fourth one a printing macro and the fifth is \.=a/b[n]. The prefix \xintthe makes the output printable by killing the first three tokens.

11.3 Catcodes, ε-TEX and reload detection

The code for reload detection was initially copied from Heiko Oberdiek's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\z {\endgroup}%
  \expandafter\let\expandafter\x\csname ver@xintexpr.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \expandafter\let\expandafter\t\csname ver@xinttools.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  % I don't think engine exists providing \expanded but not \numexpr
  \ifx\csname expanded\endcsname\relax
     \y{xintexpr}{\expanded not available, aborting input}%
     \aftergroup\endinput
```

```
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintexpr.sty
      \ifx\w\relax % but xintfrac.sty not yet loaded.
         \expandafter\def\expandafter\z\expandafter
                     {\z\input xintfrac.sty\relax}%
      \fi
      \ifx\t\relax % but xinttools.sty not yet loaded.
         \expandafter\def\expandafter\z\expandafter
                     {\z\input xinttools.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xintfrac.sty not yet loaded.
            \expandafter\def\expandafter\z\expandafter
                        {\z\RequirePackage{xintfrac}}%
          \fi
          \ifx\t\relax % xinttools.sty not yet loaded.
            \expandafter\def\expandafter\z\expandafter
                        {\z\RequirePackage{xinttools}}%
          \fi
      \else
        \aftergroup\endinput % xintexpr already loaded.
      \fi
    \fi
  \fi
\z%
\XINTsetupcatcodes%
```

11.4 Package identification

\XINT_Cmp alias for \xintiiCmp needed for some forgotten reason related to \xintNewExpr (FIX THIS!)

```
\XINT_providespackage
\ProvidesPackage{xintexpr}%
  [2021/03/29 v1.4d Expandable expression parser (JFB)]%
\catcode`! 11
\let\XINT_Cmp \xintiiCmp
\def\XINTfstop{\noexpand\XINTfstop}%
```

11.5 \xintDigits*, \xintSetDigits*

1.3f

```
\def\xintDigits {\futurelet\XINT_token\xintDigitss}%
\def\xintDigitss #1={\afterassignment\xintDigitsss\mathchardef\XINTdigits=}%
\def\xintDigitsss#1{\ifx*\XINT_token\expandafter\xintreloadxinttrig\fi}%
\let\xintfracSetDigits\xintSetDigits
\def\xintSetDigits#1#{\if\relax\detokenize{#1}\relax
                      \else\afterassignment\xintreloadxinttrig\fi
                      \xintfracSetDigits}%
```

11.6 Support for output and transform of nested braced contents as core data type

New at 1.4, of course. The former \csname.=...\endcsname encapsulation technique made very difficult implementation of nested structures.

11.6.1 Bracketed list rendering with prettifying of leaves from nested braced contents

1.4 The braces in \XINT:expr:toblistwith are there because there is an \expanded trigger.
1.4d: support for polexpr 0.8 polynomial type.

```
\def\XINT:expr:toblistwith#1#2%
{%
    {\expandafter\XINT:expr:toblist_checkempty
     \expanded{\noexpand#1!\expandafter}\detokenize{#2}^}%
}%
\def\XINT:expr:toblist_checkempty #1!#2%
{%
    \if ^#2\expandafter\xint_gob_til_^\else\expandafter\XINT:expr:toblist_a\fi
    #1!#2%
}%
\catcode`< 1 \catcode`> 2 \catcode`{ 12 \catcode`} 12
\def\XINT:expr:toblist_a #1{#2%
<%
    \if{#2\xint_dothis<[\XINT:expr:toblist_a>\fi
    \if P#2\xint_dothis<\XINT:expr:toblist_pol>\fi
    \xint_orthat\XINT:expr:toblist_b #1#2%
>%
\def\XINT:expr:toblist_pol #1!#2.{#3}}%
<%
    pol([\XINT:expr:toblist_b #1!#3}^])\XINT:expr:toblist_c #1!}%
>%
\def\XINT:expr:toblist_b #1!#2}%
<%
    \if\relax#2\relax\xintexprEmptyItem\else#1<#2>\fi\XINT:expr:toblist_c #1!}%
>%
\def\XINT:expr:toblist_c #1}#2%
<%
    \if ^#2\xint_dothis<\xint_gob_til_^>\fi
    \if{#2\xint_dothis<, \XINT:expr:toblist_a>\fi
    \xint_orthat<]\XINT:expr:toblist_c>#1#2%
>%
\catcode`{ 1 \catcode`} 2 \catcode`< 12 \catcode`> 12
```

11.6.2 Flattening nested braced contents

1.4b I hesitated whether using this technique or some variation of the method of the ListSel macros. I chose this one which I downscaled from toblistwith, I will revisit later. I only have a few minutes right now.
Call form is \expanded\XINT:expr:flatten
See \XINT_expr_func_flat. I hesitated with «flattened», but short names are faster parsed.

```
\def\XINT:expr:flatten#1%
```

```
{%
   {{\expandafter\XINT:expr:flatten_checkempty\detokenize{#1}^}}%
}%
\def\XINT:expr:flatten_checkempty #1%
{%
    \if ^#1\expandafter\xint_gobble_i\else\expandafter\XINT:expr:flatten_a\fi
    #1%
}%
\begingroup % should I check lccode s generally if corrupted context at load?
\catcode`[ 1 \catcode`] 2 \lccode`[`{ \lccode`]`}
\catcode`< 1 \catcode`> 2 \catcode`{ 12 \catcode`} 12
\lowercase<\endgroup
\def\XINT:expr:flatten_a {#1%
<%
    \if{#1\xint_dothis<\XINT:expr:flatten_a>\fi
    \xint_orthat\XINT:expr:flatten_b #1%
>%
\def\XINT:expr:flatten_b #1}%
<%
    [#1]\XINT:expr:flatten_c }%
>%
\def\XINT:expr:flatten_c }#1%
<%
    \if ^#1\xint_dothis<\xint_gobble_i>\fi
    \if{#1\xint_dothis<\XINT:expr:flatten_a>\fi
    \xint_orthat<\XINT:expr:flatten_c>#1%
>%
>% back to normal catcodes
```

11.6.3 Braced contents rendering via a TEX alignment with prettifying of leaves

1.4.

Breaking change at 1.4a as helper macros were renamed and their meanings refactored: no more \xintexpraligntab nor \xintexpraligninnercomma or \xintexpralignoutercomma but \xintexpraligninnersep, etc...

At 1.4c I remove the \protected from \xintexpralignend. I had made note a year ago that it served nothing. Let's trust myself on this one (risky one year later!) .

```
\catcode`& 4
\protected\def\xintexpralignbegin       {\halign\bgroup\tabskip2ex\hfil##&&##\hfil\cr}%
\def\xintexpralignend                   {\crcr\egroup}%
\protected\def\xintexpralignlinesep     {,\cr}%
\protected\def\xintexpralignleftbracket {[}%
\protected\def\xintexpralignrightbracket{]}%
\protected\def\xintexpralignleftsep     {&}%
\protected\def\xintexpralignrightsep    {&}%
\protected\def\xintexpraligninnersep    {,&}%
\catcode`& 7
\def\XINT:expr:toalignwith#1#2%
{%
    {\expandafter\XINT:expr:toalign_checkempty
     \expanded{\noexpand#1!\expandafter}\detokenize{#2}^\expandafter}%
    \xintexpralignend
```

```
}%
\def\XINT:expr:toalign_checkempty #1!#2%
{%
    \if ^#2\expandafter\xint_gob_til_^\else\expandafter\XINT:expr:toalign_a\fi
    #1!#2%
}%
\catcode`< 1 \catcode`> 2 \catcode`{ 12 \catcode`} 12
\def\XINT:expr:toalign_a #1{#2%
<%
    \if{#2\xint_dothis<\xintexpralignleftbracket\XINT:expr:toalign_a>\fi
    \xint_orthat<\xintexpralignleftsep\XINT:expr:toalign_b>#1#2%
>%
\def\XINT:expr:toalign_b #1!#2}%
<%
    \if\relax#2\relax\xintexprEmptyItem\else#1<#2>\fi\XINT:expr:toalign_c #1!}%
>%
\def\XINT:expr:toalign_c #1}#2%
<%
    \if ^#2\xint_dothis<\xint_gob_til_^>\fi
    \if {#2\xint_dothis<\xintexpraligninnersep\XINT:expr:toalign_A>\fi
    \xint_orthat<\xintexpralignrightsep\xintexpralignrightbracket\XINT:expr:toalign_C>#1#2%
>%
\def\XINT:expr:toalign_A #1{#2%
<%
    \if{#2\xint_dothis<\xintexpralignleftbracket\XINT:expr:toalign_A>\fi
    \xint_orthat\XINT:expr:toalign_b #1#2%
>%
\def\XINT:expr:toalign_C #1}#2%
<%
    \if ^#2\xint_dothis<\xint_gob_til_^>\fi
    \if {#2\xint_dothis<\xintexpralignlinesep\XINT:expr:toalign_a>\fi
    \xint_orthat<\xintexpralignrightbracket\XINT:expr:toalign_C>#1#2%
>%
\catcode`{ 1 \catcode`} 2 \catcode`< 12 \catcode`> 12
```

11.6.4 Transforming all leaves within nested braced contents

1.4. Leaves must be of catcode 12... This is currently not a constraint (or rather not a new constraint) for xintexpr because formerly anyhow all data went through csname encapsulation and extraction via string.

In order to share code with the functioning of universal functions, which will be allowed to transform a number into an ople, the applied macro is supposed to apply one level of bracing to its ouput. Thus to apply this with an xintfrac macro such as \xintiRound{0} one needs first to define a wrapper which will expand it into braces :

\def\foo#1{{\xintiRound{0}{#1}}

As the things will expand inside expanded, propagating expansion is not an issue.

This code is used by \xintiexpr and \xintfloatexpr in case of optional argument and by the «Universal functions».

```
\def\XINT:expr:mapwithin#1#2%
{%
    {{\expandafter\XINT:expr:mapwithin_checkempty
      \expanded{\noexpand#1!\expandafter}\detokenize{#2}^}}%
```

```
}%
\def\XINT:expr:mapwithin_checkempty #1!#2%
{%
    \if ^#2\expandafter\xint_gob_til_^\else\expandafter\XINT:expr:mapwithin_a\fi
    #1!#2%
}%
\begingroup % should I check lccode s generally if corrupted context at load?
\catcode`[ 1 \catcode`] 2 \lccode`[`{ \lccode`]`}
\catcode`< 1 \catcode`> 2 \catcode`{ 12 \catcode`} 12
\lowercase<\endgroup
\def\XINT:expr:mapwithin_a #1{#2%
<%
    \if{#2\xint_dothis<[\iffalse]\fi\XINT:expr:mapwithin_a>\fi%
    \xint_orthat\XINT:expr:mapwithin_b #1#2%
>%
\def\XINT:expr:mapwithin_b #1!#2}%
<%
    #1<#2>\XINT:expr:mapwithin_c #1!}%
>%
\def\XINT:expr:mapwithin_c #1}#2%
<%
    \if ^#2\xint_dothis<\xint_gob_til_^>\fi
    \if{#2\xint_dothis<\XINT:expr:mapwithin_a>\fi%
    \xint_orthat<\iffalse[\fi]\XINT:expr:mapwithin_c>#1#2%
>%
>% back to normal catcodes
```

11.7 Top level user TEX interface: \xinteval, \xintfloateval, \xintiieval

11.7.1 \xintexpr, \xintiexpr, \xintfloatexpr, \xintiiexpr

\xintiexpr and \xintfloatexpr have an optional argument since 1.1.
ATTENTION! 1.3d renamed \xinteval to \xintexpro etc...
The 1.4a version of optional argument [D] for \xintiexpr accepts a negative D, with same meaning as the 1.4a \xintRound from xintfrac.sty.

```
\def\xintexpr      {\romannumeral0\xintexpro      }%
\def\xintiexpr     {\romannumeral0\xintiexpro     }%
\def\xintfloatexpr {\romannumeral0\xintfloatexpro }%
```

```
\def\xintiiexpr     {\romannumeral0\xintiiexpro    }%
\def\xintexpro   {\expandafter\XINT_expr_wrap\romannumeral0\xintbareeval }%
\def\xintiiexpro {\expandafter\XINT_iiexpr_wrap\romannumeral0\xintbareiieval }%
\def\xintiexpro #1%
{%
    \ifx [#1\expandafter\XINT_iexpr_withopt\else\expandafter\XINT_iexpr_noopt
    \fi #1%
}%
\def\XINT_iexpr_noopt
{%
    \expandafter\XINT_iexpr_iiround\romannumeral0\xintbareeval
}%
\def\XINT_iexpr_iiround
{%
    \expandafter\XINT_expr_wrap
    \expanded
    \XINT:NEhook:x:mapwithin\XINT:expr:mapwithin{\XINTbracediRoundzero}%
}%
\def\XINTbracediRoundzero#1{{\xintiRound{0}{#1}}}%
\def\XINT_iexpr_withopt [#1]%
{%
    \expandafter\XINT_iexpr_round
    \the\numexpr \xint_zapspaces #1 \xint_gobble_i\expandafter.%
    \romannumeral0\xintbareeval
}%
\def\XINT_iexpr_round #1.%
{%
    \ifnum#1=\xint_c_\xint_dothis{\XINT_iexpr_iiround}\fi
    \xint_orthat{\XINT_iexpr_round_a #1.}%
}%
\def\XINT_iexpr_round_a #1.%
{%
    \expandafter\XINT_expr_wrap
    \expanded
    \XINT:NEhook:x:mapwithin\XINT:expr:mapwithin{\XINTbracedRound{#1}}%
}%
\def\XINTbracedRound#1#2{{\xintRound{#1}{#2}}}%
\def\xintfloatexpro #1%
{%
    \ifx [#1\expandafter\XINT_flexpr_withopt\else\expandafter\XINT_flexpr_noopt
    \fi #1%
}%
\def\XINT_flexpr_noopt
{%
    \expandafter\XINT_flexpr_wrap\the\numexpr\XINTdigits\expandafter.%
    \romannumeral0\xintbarefloateval
}%
\def\XINT_flexpr_withopt [#1]%
{%
    \expandafter\XINT_flexpr_withopt_a
    \the\numexpr\xint_zapspaces #1 \xint_gobble_i\expandafter.%
    \romannumeral0\xintbarefloateval
```

```
}%
\def\XINT_flexpr_withopt_a #1#2.%
{%
    \expandafter\XINT_flexpr_withopt_b\the\numexpr\if#1-\XINTdigits\fi#1#2.%
}%
\def\XINT_flexpr_withopt_b #1.%
{%
    \expandafter\XINT_flexpr_wrap
    \the\numexpr#1\expandafter.%
    \expanded
    \XINT:NEhook:x:mapwithin\XINT:expr:mapwithin{\XINTbracedinFloat[#1]}%
}%
\def\XINTbracedinFloat[#1]#2{{\XINTinFloat[#1]{#2}}}%
```

11.7.2 \XINT_expr_wrap, \XINT_iiexpr_wrap, \XINT_flexpr_wrap

1.3e removes some leading space tokens which served nothing. There is no \XINT_iexpr_wrap, because \XINT_expr_wrap is used directly.

```
\def\XINT_expr_wrap   {\XINTfstop\XINTexprprint.}%
\def\XINT_iiexpr_wrap {\XINTfstop\XINTiiexprprint.}%
\def\XINT_flexpr_wrap {\XINTfstop\XINTflexprprint}%
```

11.7.3 \XINTexprprint, \XINTiiexprprint, \XINTflexprprint

Comments currently under reconstruction.
  1.4: attention that this now requires \expanded context as the «printer» macros are not f-expandable but only e-expandable.

```
\protected\def\XINTexprprint.%
    {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintexprPrintOne}%
\let\xintexprPrintOne\xintFracToSci
\def\xintexprEmptyItem{[]}%
\protected\def\XINTiiexprprint.%
    {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintiiexprPrintOne}%
\let\xintiiexprPrintOne\xint_firstofone
\protected\def\XINTflexprprint #1.%
    {\XINT:NEhook:x:toblist\XINT:expr:toblistwith{\xintfloatexprPrintOne{#1}}}%
\def\xintfloatexprPrintOne#1%
    {\romannumeral0\XINT_pfloat_opt [\xint:#1]}% bad direct jump
\protected\def\XINTboolexprprint.%
    {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintboolexprPrintOne}%
\def\xintboolexprPrintOne#1{\xintiiifNotZero{#1}{True}{False}}%
```

11.7.4 \xintthe, \xintthealign, \xinttheexpr, \xinttheiexpr, \xintthefloatexpr, \xinttheiiexpr

The reason why \xinttheiexpr et \xintthefloatexpr are handled differently is that they admit an optional argument which acts via a custom «printing» stage.
  We exploit here that \expanded expands forward until finding an implicit or explicit brace, and that this expansion overrules \protected macros, forcing them to expand, similarly as \romannumeral expands \protected macros, and contrarily to what happens *within* the actual \expanded scope. I discovered this fact by testing (with pdftex) and I don't know where this is documented

apart from the source code of the relevant engines. This is useful to us because there are contexts where we will want to apply a complete expansion before printing, but in purely numerical context this is not needed (if I converted correctly after dropping at 1.4 the \csname governed expansions; however I rely at various places on the fact that the xint macros are f-expandable, so I have tried to not use zillions of expanded all over the place), hence it is not needed to add the expansion overhead by default. But the \expanded here will allow \xintNewExpr to create macro with suitable modification or the printing step, via some hook rather than having to duplicate all macros here with some new «NE» meaning (aliasing does not work or causes big issues due to desire to support \xinteval also in «NE» context as sub-constituent. The \XINT:NEhook:x:toblist is something else which serves to achieve this support of *sub* \xinteval, it serves nothing for the actual produced macros. For \xintdeffunc, things are simpler, but still we support the [N] optional argument of \xintiexpr and \xintfloatexpr, which required some work...

The \expanded upfront ensures \xintthe mechanism does expand completely in two steps.

```
\def\xintthe      #1{\expanded\expandafter\xint_gobble_i\romannumeral`&&@#1}%
\def\xintthealign #1{\expandafter\xintexpralignbegin
                     \expanded\expandafter\XINT:expr:toalignwith
                     \romannumeral0\expandafter\expandafter\expandafter\expandafter
                                   \expandafter\expandafter\expandafter\xint_gob_andstop_ii
                     \expandafter\xint_gobble_i\romannumeral`&&@#1}%
\def\xintttheexpr
   {\expanded\expandafter\XINTexprprint\expandafter.\romannumeral0\xintbareeval}%
\def\xintttheiexpr
   {\expanded\expandafter\xint_gobble_i\romannumeral`&&@\xintiexpr}%
\def\xintttheexpr
   {\expanded\expandafter\xint_gobble_i\romannumeral`&&@\xintfloatexpr}%
\def\xintttheiiexpr
   {\expanded\expandafter\XINTiiexprprint\expandafter.\romannumeral0\xintbareiieval}%
```

11.7.5 \thexintexpr, \thexintiexpr, \thexintfloatexpr, \thexintiiexpr

New with 1.2h. I have been for the last three years very strict regarding macros with \xint or \XINT, but well.

1.4. Definitely I don't like those. I will remove them at 1.5.

```
\let\thexintexpr     \xintttheexpr
\let\thexintiexpr    \xintttheiexpr
\let\thexintfloatexpr\xintttheexpr
\let\thexintiiexpr   \xintttheiiexpr
```

11.7.6 \xintbareeval, \xintbarefloateval, \xintbareiieval

At 1.4 added one expansion step via _start macros. Triggering is expected to be via either \romannumeral`^^@ or \romannumeral0 is also ok

```
\def\xintbareeval     {\XINT_expr_start  }%
\def\xintbarefloateval{\XINT_flexpr_start}%
\def\xintbareiieval   {\XINT_iiexpr_start}%
```

11.7.7 \xintthebareeval, \xintthebarefloateval, \xintthebareiieval

For matters of \XINT_NewFunc

```
\def\xintthebareeval      {\romannumeral0\expandafter\xint_stop_atfirstofone\romannumeral0\xintbareeval}
\def\xintthebarefloateval {\romannumeral0\expandafter\xint_stop_atfirstofone\romannumeral0\xintbarefloat
\def\xintthebareiieval    {\romannumeral0\expandafter\xint_stop_atfirstofone\romannumeral0\xintbareiieva
```

11.7.8 \xinteval, \xintieval, \xintfloateval, \xintiieval

Refactored at 1.4.
  The \expanded upfront ensures \xinteval still expands completely in two steps. No \romannumeral trigger here, in relation to the fact that \XINTexprprint is no f-expandable, only e-expandable.
  (and attention that \xintexpr\relax is now legal, and an empty ople can be produced in output also from \xintexpr [17][1]\relax for example)

```
\def\xinteval #1%
   {\expanded\expandafter\XINTexprprint\expandafter.\romannumeral0\xintbareeval#1\relax}%
\def\xintieval #1%
   {\expanded\expandafter\xint_gobble_i\romannumeral`&&@\xintiexpr#1\relax}%
\def\xintfloateval #1%
   {\expanded\expandafter\xint_gobble_i\romannumeral`&&@\xintfloatexpr#1\relax}%
\def\xintiieval #1%
   {\expanded\expandafter\XINTiiexprprint\expandafter.\romannumeral0\xintbareiieval#1\relax}%
```

11.7.9 \xintboolexpr, \XINT_boolexpr_print, \xinttheboolexpr, \thexintboolexpr

ATTENTION! 1.3d renamed \xinteval to \xintexpro etc...
  Attention, the conversion to 1 or 0 is done only by the print macro. Perhaps I should force it also inside raw result.

```
\def\xintboolexpr
{%
    \romannumeral0\expandafter\XINT_boolexpr_done\romannumeral0\xintexpro
}%
\def\XINT_boolexpr_done #1.{\XINTfstop\XINTboolexprprint.}%
\def\xinttheboolexpr
{%
    \expanded\expandafter\XINTboolexprprint\expandafter.\romannumeral0\xintbareeval
}%
\let\thexintboolexpr\xinttheboolexpr
```

11.7.10 \xintifboolexpr, \xintifboolfloatexpr, \xintifbooliiexpr

They do not accept comma separated expressions input.

```
\def\xintifboolexpr      #1{\romannumeral0\xintiiifnotzero {\xinttheexpr #1\relax}}%
\def\xintifboolfloatexpr #1{\romannumeral0\xintiiifnotzero {\xintthefloatexpr #1\relax}}%
\def\xintifbooliiexpr    #1{\romannumeral0\xintiiifnotzero {\xintttheiiexpr #1\relax}}%
```

11.7.11 \xintifsgnexpr, \xintifsgnfloatexpr, \xintifsgniiexpr

1.3d.
  They do not accept comma separated expressions.

```
\def\xintifsgnexpr      #1{\romannumeral0\xintiiifsgn {\xinttheexpr #1\relax}}%
\def\xintifsgnfloatexpr #1{\romannumeral0\xintiiifsgn {\xintthefloatexpr #1\relax}}%
\def\xintifsgniiexpr    #1{\romannumeral0\xintiiifsgn {\xintttheiiexpr #1\relax}}%
```

11.7.12 Small bits we have to put somewhere

Some renaming and modifications here with release 1.2 to switch from using chains of \romannumeral-`0 in order to gather numbers, possibly hexadecimals, to using a \csname governed expansion. In this way no more limit at 5000 digits, and besides this is a logical move because the \xintexpr parser is already based on \csname...\endcsname storage of numbers as one token.

The limitation at 5000 digits didn't worry me too much because it was not very realistic to launch computations with thousands of digits... such computations are still slow with 1.2 but less so now. Chains or \romannumeral are still used for the gathering of function names and other stuff which I have half-forgotten because the parser does many things.

In the earlier versions we used the lockscan macro after a chain of \romannumeral-`0 had ended gathering digits; this uses has been replaced by direct processing inside a \csname...\endcsname and the macro is kept only for matters of dummy variables.

Currently, the parsing of hexadecimal numbers needs two nested \csname...\endcsname, first to gather the letters (possibly with a hexadecimal fractional part), and in a second stage to apply \xintHexToDec to do the actual conversion. This should be faster than updating on the fly the number (which would be hard for the fraction part...).

```
\def\XINT_embrace#1{{#1}}%
\def\xint_gob_til_! #1!{}% ! with catcode 11
\def\xintError:noopening
{%
    \XINT_expandableerror{Extra ) found during balancing, e(X)it before the worst.}%
}%
```

\xintthecoords 1.1 Wraps up an even number of comma separated items into pairs of TikZ coordinates; for use in the following way:

coordinates {\xintthecoords\xintfloatexpr ... \relax}

The crazyness with the \csname and unlock is due to TikZ somewhat STRANGE control of the TOTAL number of expansions which should not exceed the very low value of 100 !! As we implemented \XINT_thecoords_b in an "inline" style for efficiency, we need to hide its expansions.

Not to be used as \xintthecoords\xintthefloatexpr, only as \xintthecoords\xintfloatexpr (or \xintiexpr etc...). Perhaps \xintthecoords could make an extra check, but one should not accustom users to too loose requirements!

```
\def\xintthecoords#1%
   {\romannumeral`&&@\expandafter\XINT_thecoords_a\romannumeral0#1}%
\def\XINT_thecoords_a #1#2.#3% #2.=\XINTfloatprint<digits>. etc...
   {\expanded{\expandafter\XINT_thecoords_b\expanded#2.{#3},!,!,^}}%
\def\XINT_thecoords_b #1#2,#3#4,%
   {\xint_gob_til_! #3\XINT_thecoords_c ! (#1#2, #3#4)\XINT_thecoords_b }%
\def\XINT_thecoords_c #1^{}%
```

\xintthespaceseparated 1.4a This is a utility macro which was distributed previously separately for usage with PSTricks \listplot

```
\def\xintthespaceseparated#1%
   {\romannumeral`&&@\expandafter\xintthespaceseparated_a\romannumeral0#1}%
\def\xintthespaceseparated_a #1#2.#3%
   {\expanded{\expandafter\xintthespaceseparated_b\expanded#2.{#3},!,!,!,!,!,!,!,!,!,^}}%
\def\xintthespaceseparated_b #1,#2,#3,#4,#5,#6,#7,#8,#9,%
   {\xint_gob_til_! #9\xintthespaceseparated_c !%
```

```
    #1#2#3#4#5#6#7#8#9%
    \xintthespaceseparated_b}%
```

1.4c I add a space here to stop the \romannumeral`^^@ if #1 is empty.

```
\def\xintthespaceseparated_c !#1!#2^{ #1}%
```

11.8 Hooks into the numeric parser for usage by the \xintdeffunc symbolic parser

This is new with 1.3 and considerably refactored at 1.4. See «Mysterious stuff».

```
\let\XINT:NEhook:f:one:from:one\expandafter
\let\XINT:NEhook:f:one:from:one:direct\empty
\let\XINT:NEhook:f:one:from:two\expandafter
\let\XINT:NEhook:f:one:from:two:direct\empty
\let\XINT:NEhook:x:one:from:two\empty
\let\XINT:NEhook:f:one:and:opt:direct      \empty
\let\XINT:NEhook:f:tacitzeroifone:direct   \empty
\let\XINT:NEhook:f:iitacitzeroifone:direct \empty
\let\XINT:NEhook:x:select:obey\empty
\let\XINT:NEhook:x:listsel\empty
\let\XINT:NEhook:f:reverse\empty
\def\XINT:NEhook:f:from:delim:u #1#2^{#1#2^}%
\def\XINT:NEhook:f:noeval:from:braced:u#1#2^{#1{#2}}%
\let\XINT:NEhook:branch\expandafter
\let\XINT:NEhook:seqx\empty
\let\XINT:NEhook:iter\expandafter
\let\XINT:NEhook:opx\empty
\let\XINT:NEhook:rseq\expandafter
\let\XINT:NEhook:iterr\expandafter
\let\XINT:NEhook:rrseq\expandafter
\let\XINT:NEhook:x:toblist\empty
\let\XINT:NEhook:x:mapwithin\empty
\let\XINT:NEhook:x:ndmapx\empty
```

11.9 \XINT_expr_getnext: fetch some value then an operator and present them to last waiter with the found operator precedence, then the operator, then the value

Big change in 1.1, no attempt to detect braced stuff anymore as the [N] notation is implemented otherwise. Now, braces should not be used at all; one level removed, then \romannumeral-`0 expansion.

Refactored at 1.4 to put expansion of \XINT_expr_getop after the fetched number, thus avoiding it to have to fetch it (which could happen then multiple times, it was not really important when it was only one token in pre-1.4 xintexpr).

Allow \xintexpr\relax at 1.4.

Refactored at 1.4 the articulation \XINT_expr_getnext/XINT_expr_func/XINT_expr_getop. For some legacy reason the first token picked by getnext was soon turned to catcode 12 The next ones after the first were not a priori stringified but the first token was, and this made allowing things such as \xintexpr\relax, \xintexpr,,\relax, [], 1+(), [:] etc... complicated and requiring each time specific measures.

```
\def\XINT_expr_getnext #1%
```

```
{%
    \expandafter\XINT_expr_put_op_first\romannumeral`&&@%
    \expandafter\XINT_expr_getnext_a\romannumeral`&&@#1%
}%
\def\XINT_expr_put_op_first #1#2#3{\expandafter#2\expandafter#3\expandafter{#1}}%
\def\XINT_expr_getnext_a #1%
{%
    \ifx\relax #1\xint_dothis\XINT_expr_foundprematureend\fi
    \ifx\XINTfstop#1\xint_dothis\XINT_expr_subexpr\fi
    \ifcat\relax#1\xint_dothis\XINT_expr_countetc\fi
    \xint_orthat{}\XINT_expr_getnextfork #1%
}%
\def\XINT_expr_foundprematureend\XINT_expr_getnextfork #1{{}\xint_c_\relax}%
\def\XINT_expr_subexpr #1.#2%
{%
    \expanded{\unexpanded{{#2}}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

```
  1.2 adds \ht, \dp, \wd and the eTeX font things. 1.4 avoids big nested \if's, simply for code
readability
```

```
\def\XINT_expr_countetc\XINT_expr_getnextfork#1%
{%
    \if0\ifx\count#11\fi
        \ifx\dimen#11\fi
        \ifx\numexpr#11\fi
        \ifx\dimexpr#11\fi
        \ifx\skip#11\fi
        \ifx\glueexpr#11\fi
        \ifx\fontdimen#11\fi
        \ifx\ht#11\fi
        \ifx\dp#11\fi
        \ifx\wd#11\fi
        \ifx\fontcharht#11\fi
        \ifx\fontcharwd#11\fi
        \ifx\fontchardp#11\fi
        \ifx\fontcharic#11\fi 0\expandafter\XINT_expr_fetch_as_number\fi
   \expandafter\XINT_expr_getnext_a\number #1%
}%
\def\XINT_expr_fetch_as_number
    \expandafter\XINT_expr_getnext_a\number #1%
{%
    \expanded{{{\number#1}}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

```
  This is a key component which is involved in:
   - support for \xintdeffunc via special handling of parameter character,
   - support for skipping over ignored + signs,
   - support for Python-like * «unpacking» unary operator (added at 1.4),
   - support for [..] nutple constructor (1.4, formerly [..] by itself was like (...)),
   - support for numbers starting with a decimal point,
   - support for the minus as unary operator of variable precedence level,
   - support for sub-expressions inside parenthesis (with possibly tacit multiplication)
   - else starting the scan of explicit digits or letters for a number or a function name
```

```
\begingroup
\lccode`;=`#
\lowercase{\endgroup
\def\XINT_expr_getnextfork #1{%
    \if#1;\xint_dothis {\XINT_expr_scan_macropar ;}\fi
    \if#1+\xint_dothis \XINT_expr_getnext_a \fi
    \if#1*\xint_dothis {{}\xint_c_ii^v 0}\fi
    \if#1[\xint_dothis {{}\xint_c_ii^v \XINT_expr_itself_obracket}\fi
    \if#1.\xint_dothis {\XINT_expr_startdec}\fi
    \if#1-\xint_dothis {{}{}-}\fi
    \if#1(\xint_dothis {{}\xint_c_ii^v (}\fi
    \xint_orthat {\XINT_expr_scan_nbr_or_func #1}%
}}%
\def\XINT_expr_scan_macropar #1#2%
{%
    \expandafter{\expandafter{\expandafter#1\expandafter
    #2\expandafter}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

11.10 \XINT_expr_scan_nbr_or_func: parsing the integer or decimal number or hexa-decimal number or function name or variable name or special hacky things

1.2 release has replaced chains of \romannumeral-`0 by \csname governed expansion. Thus there is no more the limit at about 5000 digits for parsed numbers.

In order to avoid having to lock and unlock in succession to handle the scientific part and adjust the exponent according to the number of digits of the decimal part, the parsing of this decimal part counts on the fly the number of digits it encounters.

There is some slight annoyance with \xintiiexpr which should never be given a [n] inside its \csname.=<digits>\endcsname storage of numbers (because its arithmetic uses the ii macros which know nothing about the [N] notation). Hence if the parser has only seen digits when hitting something else than the dot or e (or E), it will not insert a [0]. Thus we very slightly compromise the efficiency of \xintexpr and \xintfloatexpr in order to be able to share the same code with \xintiiexpr.

Indeed, the parser at this location is completely common to all, it does not know if it is working inside \xintexpr or \xintiiexpr. On the other hand if a dot or a e (or E) is met, then the (common) parser has no scrupules ending this number with a [n], this will provoke an error later if that was within an \xintiiexpr, as soon as an arithmetic macro is used.

As the gathered numbers have no spaces, no pluses, no minuses, the only remaining issue is with leading zeroes, which are discarded on the fly. The hexadecimal numbers leading zeroes are stripped in a second stage by the \xintHexToDec macro.

With 1.2, \xinttheexpr . \relax does not work anymore (it did in earlier releases). There must be digits either before or after the decimal mark. Thus both \xinttheexpr 1.\relax and \xinttheexpr .1\relax are legal.

```
  The ` syntax is here used for special constructs like `+`(..), `*`(..) where + or * will be
treated as functions. Current implementation picks only one token (could have been braced stuff),
here it will be + or *, and via \XINT_expr_op_` this then becomes a suitable \XINT_{expr|iiexpr|flexpr}_func_+
(or *). Documentation says to use `+`(...), but `+(...) is also valid. The opening parenthesis must
be there, it is not allowed to come from expansion.
  Attention at this location #1 was of catcode 12 in all versions prior to 1.4.
  Besides using principally \if tests, we will assume anyhow that catcodes of digits are 12...
```

```
\catcode96 11 % `
\def\XINT_expr_scan_nbr_or_func #1%
{%
    \if "#1\xint_dothis \XINT_expr_scanhex_I\fi
    \if `#1\xint_dothis {\XINT_expr_onliteral_`}\fi
    \ifnum \xint_c_ix<1\string#1 \xint_dothis \XINT_expr_startint\fi
    \xint_orthat \XINT_expr_scanfunc #1%
}%
\def\XINT_expr_onliteral_` #1#2#3({{#2}\xint_c_ii^v `}%
\catcode96 12 % `
\def\XINT_expr_startint #1%
{%
    \if #10\expandafter\XINT_expr_gobz_a\else\expandafter\XINT_expr_scanint_a\fi #1%
}%
\def\XINT_expr_scanint_a #1#2%
    {\expanded\bgroup{{\iffalse}}\fi #1% spare a \string
     \expandafter\XINT_expr_scanint_main\romannumeral`&&@#2}%
\def\XINT_expr_gobz_a #1#2%
    {\expanded\bgroup{{\iffalse}}\fi
     \expandafter\XINT_expr_gobz_scanint_main\romannumeral`&&@#2}%
\def\XINT_expr_startdec #1%
    {\expanded\bgroup{{\iffalse}}\fi
     \expandafter\XINT_expr_scandec_a\romannumeral`&&@#1}%
```

11.10.1 Integral part (skipping zeroes)

```
1.2 has modified the code to give highest priority to digits, the accelerating impact is non-
negligeable. I don't think the doubled \string is a serious penalty.
```

```
\def\XINT_expr_scanint_main #1%
{%
    \ifcat \relax #1\expandafter\XINT_expr_scanint_hit_cs \fi
    \ifnum\xint_c_ix<1\string#1 \else\expandafter\XINT_expr_scanint_next\fi
    #1\XINT_expr_scanint_again
}%
\def\XINT_expr_scanint_again #1%
{%
    \expandafter\XINT_expr_scanint_main\romannumeral`&&@#1%
}%
\def\XINT_expr_scanint_hit_cs \ifnum#1\fi#2\XINT_expr_scanint_again
{%
    \iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#2%
}%
```

With 1.2d the tacit multiplication in front of a variable name or function name is now done with a higher precedence, intermediate between the common one of * and / and the one of ^. Thus x/2y is like x/(2y), but x^2y is like x^2*y and 2y! is not (2y)! but 2*y!.

Finally, 1.2d has moved away from the _scan macros all the business of the tacit multiplication in one unique place via \XINT_expr_getop. For this, the ending token is not first given to \string as was done earlier before handing over back control to \XINT_expr_getop. Earlier we had to identify the catcode 11 ! signaling a sub-expression here. With no \string applied we can do it in \XINT_expr_getop. As a corollary of this displacement, parsing of big numbers should be a tiny bit faster now.

Extended for 1.2l to ignore underscore character _ if encountered within digits; so it can serve as separator for better readability.

It is not obvious at 1.4 to support [] for three things: packing, slicing, ... and raw xintfrac syntax A/B[N]. The only good way would be to actually really separate completely \xintexpr, \xintfloatexpr and \xintiiexpr code which would allow to handle both / and [] from A/B[N] as we handle e and E. But triplicating the code is something I need to think about. It is not possible as in pre 1.4 to consider [only as an operator of same precedence as multiplication and division which was the way we did this, but we can use the technique of fake operators. Thus we intercept hitting a [here, which is not too much of a problem as anyhow we dropped temporarily 3*[1,2,3]+5 syntax so we don't have to worry that 3[1,2,3] should do tacit multiplication. I think only way in future will be to really separate the code of the three parsers (or drop entirely support for A/B[N]; as 1.4 has modified output of \xinteval to not use this notation this is not too dramatic).

Anyway we find a way to inject here the former handling of [N], which will use a delimited macro to directly fetch until the closing]. We do still need some fake operator because A/B[N] is (A/B) times 10^N and the /B is allowed to be missing. We hack this using the which is not used currently as operator elsewhere in the syntax and need to hook into \XINT_expr_getop_b. No finally I use the null char. It must be of catcode 12.

```
\def\XINT_expr_scanint_next #1\XINT_expr_scanint_again
{%
    \if    [#1\xint_dothis\XINT_expr_rawxintfrac\fi
    \if    _#1\xint_dothis\XINT_expr_scanint_again\fi
    \if    e#1\xint_dothis{[\the\numexpr0\XINT_expr_scanexp_a +}\fi
    \if    E#1\xint_dothis{[\the\numexpr0\XINT_expr_scanexp_a +}\fi
    \if    .#1\xint_dothis{\XINT_expr_startdec_a .}\fi
    \xint_orthat
    {\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#1}%
}%
\def\XINT_expr_rawxintfrac
{%
    \iffalse{{{\fi}}\expandafter}\csname XINT_expr_precedence_&&@\endcsname&&@%
}%
\def\XINT_expr_gobz_scanint_main #1%
{%
    \ifcat \relax #1\expandafter\XINT_expr_gobz_scanint_hit_cs\fi
    \ifnum\xint_c_x<1\string#1 \else\expandafter\XINT_expr_gobz_scanint_next\fi
    #1\XINT_expr_scanint_again
}%
\def\XINT_expr_gobz_scanint_again #1%
{%
    \expandafter\XINT_expr_gobz_scanint_main\romannumeral`&&@#1%
}%
\def\XINT_expr_gobz_scanint_hit_cs\ifnum#1\fi#2\XINT_expr_scanint_again
```

```
{%
    0\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#2%
}%
\def\XINT_expr_gobz_scanint_next #1\XINT_expr_scanint_again
{%
    \if    [#1\xint_dothis{\expandafter0\XINT_expr_rawxintfrac}\fi
    \if    _#1\xint_dothis\XINT_expr_gobz_scanint_again\fi
    \if    e#1\xint_dothis{0[\the\numexpr0\XINT_expr_scanexp_a +}\fi
    \if    E#1\xint_dothis{0[\the\numexpr0\XINT_expr_scanexp_a +}\fi
    \if    .#1\xint_dothis{\XINT_expr_gobz_startdec_a .}\fi
    \if    0#1\xint_dothis\XINT_expr_gobz_scanint_again\fi
    \xint_orthat
    {0\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#1}%
}%
```

11.10.2 Fractional part

Annoying duplication of code to allow 0. as input.
1.2a corrects a very bad bug in 1.2 \XINT_expr_gobz_scandec_b which should have stripped leading zeroes in the fractional part but didn't; as a result \xinttheexpr 0.01\relax returned 0 =:-(((Thanks to Kroum Tzanev who reported the issue. Does it improve things if I say the bug was introduced in 1.2, it wasn't present before ?

```
\def\XINT_expr_startdec_a .#1%
{%
    \expandafter\XINT_expr_scandec_a\romannumeral`&&@#1%
}%
\def\XINT_expr_scandec_a #1%
{%
    \if .#1\xint_dothis{\iffalse{{{\fi}}\expandafter}%
                          \romannumeral`&&@\XINT_expr_getop..}\fi
    \xint_orthat {\XINT_expr_scandec_main 0.#1}%
}%
\def\XINT_expr_gobz_startdec_a .#1%
{%
    \expandafter\XINT_expr_gobz_scandec_a\romannumeral`&&@#1%
}%
\def\XINT_expr_gobz_scandec_a #1%
{%
    \if .#1\xint_dothis
    {0\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop..}\fi
    \xint_orthat {\XINT_expr_gobz_scandec_main 0.#1}%
}%
\def\XINT_expr_scandec_main #1.#2%
{%
    \ifcat \relax #2\expandafter\XINT_expr_scandec_hit_cs\fi
    \ifnum\xint_c_ix<1\string#2 \else\expandafter\XINT_expr_scandec_next\fi
    #2\expandafter\XINT_expr_scandec_again\the\numexpr #1-\xint_c_i.%
}%
\def\XINT_expr_scandec_again #1.#2%
{%
    \expandafter\XINT_expr_scandec_main
    \the\numexpr #1\expandafter.\romannumeral`&&@#2%
```

```
}%
\def\XINT_expr_scandec_hit_cs\ifnum#1\fi
    #2\expandafter\XINT_expr_scandec_again\the\numexpr#3-\xint_c_i.%
{%
    [#3]\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#2%
}%
\def\XINT_expr_scandec_next #1#2\the\numexpr#3-\xint_c_i.%
{%
    \if    _#1\xint_dothis{\XINT_expr_scandec_again#3.}\fi
    \if    e#1\xint_dothis{[\the\numexpr#3\XINT_expr_scanexp_a +}\fi
    \if    E#1\xint_dothis{[\the\numexpr#3\XINT_expr_scanexp_a +}\fi
    \xint_orthat
    {[#3]\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#1}%
}%
\def\XINT_expr_gobz_scandec_main #1.#2%
{%
    \ifcat \relax #2\expandafter\XINT_expr_gobz_scandec_hit_cs\fi
    \ifnum\xint_c_ix<1\string#2 \else\expandafter\XINT_expr_gobz_scandec_next\fi
    \if0#2\expandafter\xint_firstoftwo\else\expandafter\xint_secondoftwo\fi
    {\expandafter\XINT_expr_gobz_scandec_main}%
    {#2\expandafter\XINT_expr_scandec_again}\the\numexpr#1-\xint_c_i.%
}%
\def\XINT_expr_gobz_scandec_hit_cs \ifnum#1\fi\if0#2#3\xint_c_i.%
{%
    0[0]\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#2%
}%
\def\XINT_expr_gobz_scandec_next\if0#1#2\fi #3\numexpr#4-\xint_c_i.%
{%
    \if    _#1\xint_dothis{\XINT_expr_gobz_scandec_main #4.}\fi
    \if    e#1\xint_dothis{0[\the\numexpr0\XINT_expr_scanexp_a +}\fi
    \if    E#1\xint_dothis{0[\the\numexpr0\XINT_expr_scanexp_a +}\fi
    \xint_orthat
    {0[0]\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#1}%
}%
```

11.10.3 Scientific notation

Some pluses and minuses are allowed at the start of the scientific part, however not later, and no parenthesis.

```
\def\XINT_expr_scanexp_a #1#2%
{%
    #1\expandafter\XINT_expr_scanexp_main\romannumeral`&&@#2%
}%
\def\XINT_expr_scanexp_main #1%
{%
    \ifcat \relax #1\expandafter\XINT_expr_scanexp_hit_cs\fi
    \ifnum\xint_c_ix<1\string#1 \else\expandafter\XINT_expr_scanexp_next\fi
    #1\XINT_expr_scanexp_again
}%
\def\XINT_expr_scanexp_again #1%
{%
```

```
    \expandafter\XINT_expr_scanexp_main_b\romannumeral`&&@#1%
}%
\def\XINT_expr_scanexpr_hit_cs\ifnum#1\fi#2\XINT_expr_scanexp_again
{%
    ]\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#2%
}%
\def\XINT_expr_scanexp_next #1\XINT_expr_scanexp_again
{%
    \if    _#1\xint_dothis  \XINT_expr_scanexp_again   \fi
    \if    +#1\xint_dothis {\XINT_expr_scanexp_a +}\fi
    \if    -#1\xint_dothis {\XINT_expr_scanexp_a -}\fi
    \xint_orthat
    {]\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#1}%
}%
\def\XINT_expr_scanexp_main_b #1%
{%
    \ifcat \relax #1\expandafter\XINT_expr_scanexp_hit_cs_b\fi
    \ifnum\xint_c_ix<1\string#1 \else\expandafter\XINT_expr_scanexp_next_b\fi
    #1\XINT_expr_scanexp_again_b
}%
\def\XINT_expr_scanexp_hit_cs_b\ifnum#1\fi#2\XINT_expr_scanexp_again_b
{%
    ]\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#2%
}%
\def\XINT_expr_scanexp_again_b #1%
{%
    \expandafter\XINT_expr_scanexp_main_b\romannumeral`&&@#1%
}%
\def\XINT_expr_scanexp_next_b #1\XINT_expr_scanexp_again_b
{%
    \if _#1\xint_dothis\XINT_expr_scanexp_again\fi
    \xint_orthat
    {]\iffalse{{{\fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop#1}%
}%
```

11.10.4 Hexadecimal numbers

1.2d has moved most of the handling of tacit multiplication to \XINT_expr_getop, but we have to do some of it here, because we apply \string before calling \XINT_expr_scanhexI_aa. I do not insert the * in \XINT_expr_scanhexI_a, because it is its higher precedence variant which will is expected, to do the same as when a non-hexadecimal number prefixes a sub-expression. Tacit multiplication in front of variable or function names will not work (because of this \string).
  Extended for 1.2l to ignore underscore character _ if encountered within digits.

```
\def\XINT_expr_hex_in #1.#2#3;%
{%
    \expanded{{{\if#2>%
      \xintHexToDec{#1}%
    \else
      \xintiiMul{\xintiiPow{625}{\xintLength{#3}}}{\xintHexToDec{#1#3}}%
      [\the\numexpr-4*\xintLength{#3}]%
    \fi}}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

```
\def\XINT_expr_scanhex_I #1% #1="
{%
    \expandafter\XINT_expr_hex_in\expanded\bgroup\XINT_expr_scanhexI_a
}%
\def\XINT_expr_scanhexI_a #1%
{%
    \ifcat #1\relax\xint_dothis{.>;\iffalse{\fi}#1}\fi
    \xint_orthat {\XINT_expr_scanhexI_aa #1}%
}%
\def\XINT_expr_scanhexI_aa #1%
{%
    \if\ifnum`#1>`/
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1%
       \expandafter\XINT_expr_scanhexI_b
    \else
       \if _#1\xint_dothis{\expandafter\XINT_expr_scanhexI_bgob}\fi
       \if .#1\xint_dothis{\expandafter\XINT_expr_scanhex_transition}\fi
       \xint_orthat {\xint_afterfi {.>;\iffalse{\fi}}}%
    \fi
    #1%
}%
\def\XINT_expr_scanhexI_b #1#2%
{%
    #1\expandafter\XINT_expr_scanhexI_a\romannumeral`&&@#2%
}%
\def\XINT_expr_scanhexI_bgob #1#2%
{%
    \expandafter\XINT_expr_scanhexI_a\romannumeral`&&@#2%
}%
\def\XINT_expr_scanhex_transition .#1%
{%
    \expandafter.\expandafter.\expandafter
    \XINT_expr_scanhexII_a\romannumeral`&&@#1%
}%
\def\XINT_expr_scanhexII_a #1%
{%
    \ifcat #1\relax\xint_dothis{;\iffalse{\fi}#1}\fi
    \xint_orthat {\XINT_expr_scanhexII_aa #1}%
}%
\def\XINT_expr_scanhexII_aa #1%
{%
    \if\ifnum`#1>`/
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1%
       \expandafter\XINT_expr_scanhexII_b
    \else
       \if _#1\xint_dothis{\expandafter\XINT_expr_scanhexII_bgob}\fi
```

```
693         \xint_orthat{\xint_afterfi {;\iffalse{\fi}}}%
694     \fi
695     #1%
696 }%
697 \def\XINT_expr_scanhexII_b #1#2%
698 {%
699     #1\expandafter\XINT_expr_scanhexII_a\romannumeral`&&@#2%
700 }%
701 \def\XINT_expr_scanhexII_bgob #1#2%
702 {%
703     \expandafter\XINT_expr_scanhexII_a\romannumeral`&&@#2%
704 }%
```

11.10.5 \XINT_expr_scanfunc: collecting names of functions and variables

At 1.4 the first token left over in string has not been submitted to \string. We also know it is not a control sequence. So we can test catcode to identify if operator is found. And it is allowed to hit some operator such as a closing parenthesis we will then insert the «nil» value (which however can cause breakage of arithmetic operations, although xintfrac.sty converts empty to 0).

The @ causes a problem because it must work with both catcode 11 or 12.

The _ can be used internally for starting variables but it will have catcode 11 then.

There was prior to 1.4 solely the dispatch in \XINT_expr_scanfunc_b but now we do it immediately and issue \XINT_expr_func only in certain cases.

But we have to be careful that !(...) and ?(...) are part of the syntax and genuine functions. Because we now do earlier to getop we must filter them out.

```
705 \def\XINT_expr_scanfunc #1%
706 {%
707   \if 1\ifcat a#10\fi\if @#10\fi\if !#10\fi\if ?#10\fi 1%
708         \expandafter\xint_firstoftwo
709   \else\expandafter\xint_secondoftwo
710   \fi
711   {\expandafter{\expandafter}\romannumeral`&&@\XINT_expr_getop#1}%
712   {\expandafter\XINT_expr_func\expanded\bgroup#1\XINT_expr_scanfunc_a}%
713 }%
714 \def\XINT_expr_scanfunc_a #1%
715 {%
716     \expandafter\XINT_expr_scanfunc_b\romannumeral`&&@#1%
717 }%
```

This handles: 1) (indirectly) tacit multiplication by a variable in front a of sub-expression, 2) (indirectly) tacit multiplication in front of a \count etc..., 3) functions which are recognized via an encountered opening parenthesis (but later this must be disambiguated from variables with tacit multiplication) 4) 5) 6) 7) acceptable components of a variable or function names: @, underscore, digits, letters (or chars of category code letter.)

The short lived 1.2d which followed the even shorter lived 1.2c managed to introduce a bug here as it removed the check for catcode 11 !, which must be recognized if ! is not to be taken as part of a variable name. Don't know what I was thinking, it was the time when I was moving the handling of tacit mutliplication entirely to the \XINT_expr_getop side. Fixed in 1.2e.

I almost decided to remove the \ifcat\relax test whose rôle is to avoid the \string#1 to do something bad is the escape char is a digit! Perhaps I will remove it at some point ! I truly almost did it, but also the case of no escape char is a problem (\string\0, if \0 is a count ...)

The (indirectly) above means that via \XINT_expr_func then \XINT_expr_op__ one goes back to \XINT_expr_getop then \XINT_expr_getop_b which is the location where tacit multiplication is now centralized. This makes the treatment of tacit multiplication for situations such as <variable>\count or <variable>\xintexpr..\relax, perhaps a bit sub-optimal, but first the variable name must be gathered, second the variable must expand to its value.

```
\def\XINT_expr_scanfunc_b #1%
{%
  \ifcat \relax#1\xint_dothis{\iffalse{\fi}(_#1}\fi
  \if (#1\xint_dothis{\iffalse{\fi}(`}\fi
  \if 1\ifcat a#10\fi
       \ifnum\xint_c_ix<1\string#1 0\fi
       \if @#10\fi
       \if _#10\fi
      1%
      \xint_dothis{\iffalse{\fi}(_#1}\fi
  \xint_orthat {#1\XINT_expr_scanfunc_a}%
}%
```

11.10.6 \XINT_expr_func: dispatch to variable replacement or to function execution

Comments written 2015/11/12: earlier there was an \ifcsname test for checking if we had a variable in front of a (, for tacit multiplication for example in x(y+z(x+w)) to work. But after I had implemented functions (that was yesterday...), I had the problem if was impossible to re-declare a variable name such as "f" as a function name. The problem is that here we can not test if the function is available because we don't know if we are in expr, iiexpr or floatexpr. The \xint_c_ii^v causes all fetching operations to stop and control is handed over to the routines which will be expr, iiexpr ou floatexpr specific, i.e. the \XINT_{expr|iiexpr|flexpr}_op_{`|_} which are invoked by the until_<op>_b macros earlier in the stream. Functions may exist for one but not the two other parsers. Variables are declared via one parser and usable in the others, but naturally \xintiiexpr has its restrictions.

Thinking about this again I decided to treat a priori cases such as x(...) as functions, after having assigned to each variable a low-weight macro which will convert this into _getop\.=<value of x>*(...). To activate that macro at the right time I could for this exploit the "onliteral" intercept, which is parser independent (1.2c).

This led to me necessarily to rewrite partially the seq, add, mul, subs, iter ... routines as now the variables fetch only one token. I think the thing is more efficient.

1.2c had \def\XINT_expr_func #1(#2{\xint_c_ii^v #2{#1}}

In \XINT_expr_func the #2 is _ if #1 must be a variable name, or #2=` if #1 must be either a function name or possibly a variable name which will then have to be followed by tacit multiplication before the opening parenthesis.

The \xint_c_ii^v is there because _op_` must know in which parser it works. Dispendious for _. Hence I modify for 1.2d.

```
\def\XINT_expr_func #1(#2{\if _#2\xint_dothis{\XINT_expr_op__{#1}}\fi
                          \xint_orthat{{#1}\xint_c_ii^v #2}}%
```

11.11 \XINT_expr_op_`: launch function or pseudo-function, or evaluate variable and insert operator of multiplication in front of parenthesized contents

The "onliteral" intercepts is for bool, togl, protect, ... but also for add, mul, seq, etc... Genuine functions have expr, iiexpr and flexpr versions (or only one or two of the three).

With 1.2c "onliteral" is also used to disambiguate a variable followed by an opening parenthesis from a function and then apply tacit multiplication. However as I use only a \ifcsname test, in order to be able to re-define a variable as function, I move the check for being a function first. Each variable name now has its onliteral_<name> associated macro. This used to be decided much earlier at the time of \XINT_expr_func.

The advantage of 1.2c code is that the same name can be used for a variable or a function.

```
\def\XINT_tmpa #1#2#3{%
  \def #1##1%
  {%
    \ifcsname XINT_#3_func_##1\endcsname
      \csname XINT_#3_func_##1\expandafter\endcsname
      \romannumeral`&&@\expandafter#2%
    \else
      \ifcsname XINT_expr_onliteral_##1\endcsname
        \csname XINT_expr_onliteral_##1\expandafter\expandafter\expandafter
        \endcsname
      \else
        \csname XINT_#3_func_\XINT_expr_unknown_function {##1}%
        \expandafter\endcsname
        \romannumeral`&&@\expandafter\expandafter\expandafter#2%
      \fi
    \fi
   }%
}%
\def\XINT_expr_unknown_function #1%
   {\XINT_expandableerror{"#1" is unknown as function. (I)nsert correct name:}}%
\def\XINT_expr_func_ #1#2#3{#1#2{{0}}}%
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
     \expandafter\XINT_tmpa
                 \csname XINT_#1_op_`\expandafter\endcsname
                 \csname XINT_#1_oparen\endcsname
                 {#1}%
}%
```

11.12 \XINT_expr_op__: replace a variable by its value and then fetch next operator

The 1.1 mechanism for \XINT_expr_var_<varname> has been modified in 1.2c. The <varname> associated macro is now only expanded once, not twice. We arrive here via \XINT_expr_func.

At 1.4 \XINT_expr_getop is launched with accumulated result on its left. But the omit and abort keywords are implemented via dummy variables which rely on possibility to modify upstream tokens. If we did here something such as _var_#1\expandafter\endcsname\romannumeral`^^@\XINT_expr_getop the premature expansion of getop would break things. Thus we revert to former code which put \XINT_expr_getop (call it _legacy) in front of variable expansion (in xintexpr < 1.4 this expanded to a single token so the overhead was not serious).

Abusing variables to manipulate token stream is a bit bad, usually I prefer functions for this (such as the break() function) but then I have define 3 macros for the 3 parsers.

The situation here is not satisfactory. But 1.4 has to be released now.

```
\def\XINT_expr_op__  #1% op__ with two _'s
{%
  \ifcsname XINT_expr_var_#1\endcsname
```

```
  \expandafter\expandafter\expandafter\XINT_expr_getop_legacy
      \csname XINT_expr_var_#1\expandafter\endcsname
  \else
  \expandafter\expandafter\expandafter\XINT_expr_getop_legacy
      \csname XINT_expr_var_\XINT_expr_unknown_variable {#1}%
        \expandafter\endcsname
  \fi
}%
\def\XINT_expr_unknown_variable #1%
   {\XINT_expandableerror {"#1" is unknown as a variable. (I)nsert correct one:}}%
\def\XINT_expr_var_{{0}}%
\let\XINT_flexpr_op__ \XINT_expr_op__
\let\XINT_iiexpr_op__ \XINT_expr_op__
\def\XINT_expr_getop_legacy #1%
{%
    \expanded{\unexpanded{{#1}}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

11.13 \XINT_expr_getop: fetch the next operator or closing parenthesis or end of expression

Release 1.1 implements multi-character operators.

1.2d adds tacit mutiplication also in front of variable or functions names starting with a letter, not only a @ or a _ as was already the case. This is for (x+y)z situations. It also applies higher precedence in cases like x/2y or x/2@, or x/2max(3,5), or x/2\xintexpr 3\relax.

In fact, finally I decide that all sorts of tacit multiplication will always use the higher precedence.

Indeed I hesitated somewhat: with the current code one does not know if \XINT_expr_getop as invoked after a closing parenthesis or because a number parsing ended, and I felt distinguishing the two was unneeded extra stuff. This means cases like (a+b)/(c+d)(e+f) will first multiply the last two parenthesized terms.

1.2q adds tacit multiplication in cases such as (1+1)3 or 5!7!

1.4 has simplified coding here as \XINT_expr_getop expansion happens at a time when a fetched value has already being stored.

```
\def\XINT_expr_getop #1%
{%
    \expandafter\XINT_expr_getop_a\romannumeral`&&@#1%
}%
\catcode`* 11
\def\XINT_expr_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_thirdofthree\fi
    \if    _#1\xint_dothis      \xint_secondofthree\fi
    \if    @#1\xint_dothis      \xint_secondofthree\fi
    \if    (#1\xint_dothis      \xint_secondofthree\fi %)
    \ifcat a#1\xint_dothis      \xint_secondofthree\fi
    \xint_orthat \xint_thirdofthree
    {\XINT_expr_foundend}%
```

```
   tacit multiplication with higher precedence.

    {\XINT_expr_precedence_*** *#1}%
    {\expandafter\XINT_expr_getop_b \string#1}%
}%
\catcode`* 12

   \relax is a place holder here.

\def\XINT_expr_foundend {\xint_c_ \relax}%

   ? is a very special operator with top precedence which will check if the next token is another
 ?, while avoiding removing a brace pair from token stream due to its syntax. Pre 1.1 releases used
 : rather than ??, but we need : for Python like slices of lists.

   null char is used as hack to implement A/B[N] raw input at 1.4. See also \XINT_expr_scanint_c.

\def\XINT_expr_getop_b #1%
{%
     \if &&@#1\xint_dothis{\csname XINT_expr_precedence_&&@\endcsname&&@}\fi
     \if '#1\xint_dothis{\XINT_expr_binopwrd }\fi
     \if ?#1\xint_dothis{\XINT_expr_precedence_? ?}\fi
     \xint_orthat       {\XINT_expr_scanop_a #1}%
}%
\def\XINT_expr_binopwrd #1'%
{%
    \expandafter\XINT_expr_foundop_a
    \csname XINT_expr_itself_\xint_zapspaces #1 \xint_gobble_i\endcsname
}%
\def\XINT_expr_scanop_a #1#2%
{%
    \expandafter\XINT_expr_scanop_b\expandafter#1\romannumeral`&&@#2%
}%
\def\XINT_expr_scanop_b #1#2%
{%
    \ifcat#2\relax\xint_dothis{\XINT_expr_foundop_a #1#2}\fi
    \ifcsname XINT_expr_itself_#1#2\endcsname
    \xint_dothis
        {\expandafter\XINT_expr_scanop_c\csname XINT_expr_itself_#1#2\endcsname}\fi
    \xint_orthat {\XINT_expr_foundop_a #1#2}%
}%
\def\XINT_expr_scanop_c #1#2%
{%
    \expandafter\XINT_expr_scanop_d\expandafter#1\romannumeral`&&@#2%
}%
\def\XINT_expr_scanop_d #1#2%
{%
  \ifcat#2\relax \xint_dothis{\XINT_expr_foundop #1#2}\fi
  \ifcsname XINT_expr_itself_#1#2\endcsname
  \xint_dothis
       {\expandafter\XINT_expr_scanop_c\csname XINT_expr_itself_#1#2\endcsname }\fi
  \xint_orthat {\csname XINT_expr_precedence_#1\endcsname #1#2}%
}%
\def\XINT_expr_foundop_a #1%
```

```
{%
    \ifcsname XINT_expr_precedence_#1\endcsname
        \csname XINT_expr_precedence_#1\expandafter\endcsname
        \expandafter #1%
    \else
        \xint_afterfi{\XINT_expr_getop\romannumeral0%
        \XINT_expandableerror
        {"#1" is unknown as operator. (I)nsert one:} }%<<deliberate space
    \fi
}%
\def\XINT_expr_foundop #1{\csname XINT_expr_precedence_#1\endcsname #1}%
```

11.14 Expansion spanning; opening and closing parentheses

These comments apply to all definitions coming next relative to execution of operations from parsing of syntax.

Refactored (and unified) at 1.4. In particular the 1.4 scheme uses op, exec, check-, and checkp. Formerly it was until_a (check-) and until_b (now split into checkp and exec).

This way neither check- nor checkp have to grab the accumulated number so far (top of stack if you like) and besides one never has to go back to check- from checkp (and neither from check-).

Prior to 1.4, accumulated intermediate results were stored as one token, but now we have to use \expanded to propagate expansion beyond possibly arbitrary long braced nested data. With the 1.4 refactoring we do this only once and only grab a second time the data if we actually have to act upon it.

Version 1.1 had a hack inside the until macros for handling the omit and abort in iterations over dummy variables. This has been removed by 1.2c, see the subsection where omit and abort are discussed.

Exceptionally, the check- is here abbreviated to check.

```
\catcode`) 11
\def\XINT_tmpa #1#2#3#4#5#6%
{%
    \def#1% start
    {%
        \expandafter#2\romannumeral`&&@\XINT_expr_getnext
    }%
    \def#2##1% check
    {%
        \xint_UDsignfork
          ##1{\expandafter#3\romannumeral`&&@#4}%
            -{#3##1}%
        \krof
    }%
    \def#3##1##2% checkp
    {%
        \ifcase ##1%
           \expandafter\XINT_expr_done
        \or\expandafter#5%
        \else
           \expandafter#3\romannumeral`&&@\csname XINT_#6_op_##2\expandafter\endcsname
        \fi
    }%
    \def#5%
```

```
    {%
        \XINT_expandableerror
        {An extra ) has been removed. Hit Return, fingers crossed.}%
        \expandafter#2\romannumeral`&&@\expandafter\XINT_expr_put_op_first
        \romannumeral`&&@\XINT_expr_getop_legacy
    }%
}%
\let\XINT_expr_done\space
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
    \expandafter\XINT_tmpa
    \csname XINT_#1_start\expandafter\endcsname
    \csname XINT_#1_check\expandafter\endcsname
    \csname XINT_#1_checkp\expandafter\endcsname
    \csname XINT_#1_op_-xii\expandafter\endcsname
    \csname XINT_#1_extra_)\endcsname
    {#1}%
}%
```

Here also we take some shortcuts relative to general philosophy and have no explicit exec macro.

```
\def\XINT_tmpa #1#2#3#4#5#6#7%
{%
    \def #1##1% op_(
    {%
        \expandafter #4\romannumeral`&&@\XINT_expr_getnext
    }%
    \def #2##1% op_)
    {%
        \expanded{\unexpanded{\XINT_expr_put_op_first{##1}}\expandafter}\romannumeral`&&@\XINT_expr_geto
    }%
    \def #3% oparen
    {%
        \expandafter #4\romannumeral`&&@\XINT_expr_getnext
    }%
    \def #4##1% check-
    {%
        \xint_UDsignfork
            ##1{\expandafter#5\romannumeral`&&@#6}%
              -{#5##1}%
        \krof
    }%
    \def #5##1##2% checkp
    {%
        \ifcase ##1\expandafter\XINT_expr_missing_)
        \or \csname XINT_#7_op_##2\expandafter\endcsname
        \else
          \expandafter #5\romannumeral`&&@\csname XINT_#7_op_##2\expandafter\endcsname
        \fi
    }%
}%
\def\XINT_expr_missing_)
   {\XINT_expandableerror{Sorry to report a missing ) at the end of this journey.}%
    \xint_c_ \XINT_expr_done }%
```

```
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
    \expandafter\XINT_tmpa
    \csname XINT_#1_op_(\expandafter\endcsname
    \csname XINT_#1_op_)\expandafter\endcsname
    \csname XINT_#1_oparen\expandafter\endcsname
    \csname XINT_#1_check-_)\expandafter\endcsname
    \csname XINT_#1_checkp_)\expandafter\endcsname
    \csname XINT_#1_op_-xii\endcsname
    {#1}%
}%
\let\XINT_expr_precedence_)\xint_c_i
\catcode`) 12
```

11.15 The comma as binary operator

New with 1.09a. Refactored at 1.4.

```
\def\XINT_tmpa #1#2#3#4#5#6%
{%
    \def #1##1% \XINT_expr_op_,
    {%
      \expanded{\unexpanded{#2{##1}}\expandafter}%
      \romannumeral`&&@\expandafter#3\romannumeral`&&@\XINT_expr_getnext
    }%
    \def #2##1##2##3##4{##2##3{##1##4}}% \XINT_expr_exec_,
    \def #3##1% \XINT_expr_check-_,
    {%
      \xint_UDsignfork
        ##1{\expandafter#4\romannumeral`&&@#5}%
          -{#4##1}%
      \krof
    }%
    \def #4##1##2% \XINT_expr_checkp_,
    {%
      \ifnum ##1>\xint_c_iii
        \expandafter#4%
           \romannumeral`&&@\csname XINT_#6_op_##2\expandafter\endcsname
      \else
        \expandafter##1\expandafter##2%
      \fi
    }%
}%
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
\expandafter\XINT_tmpa
    \csname XINT_#1_op_,\expandafter\endcsname
    \csname XINT_#1_exec_,\expandafter\endcsname
    \csname XINT_#1_check-_,\expandafter\endcsname
    \csname XINT_#1_checkp_,\expandafter\endcsname
    \csname XINT_#1_op_-xii\endcsname {#1}%
}%
\expandafter\let\csname XINT_expr_precedence_,\endcsname\xint_c_iii
```

11.16 The minus as prefix operator of variable precedence level

Inherits the precedence level of the previous infix operator. Refactored at 1.4

```
\def\XINT_tmpb #1#2#3#4#5#6#7%
{%
    \def #1% \XINT_expr_op_-<level>
    {%
      \expandafter #2\romannumeral`&&@\expandafter#3%
      \romannumeral`&&@\XINT_expr_getnext
    }%
    \def #2##1##2##3% \XINT_expr_exec_-<level>
    {%
      \expandafter ##1\expandafter ##2\expandafter
       {%
        \romannumeral`&&@\XINT:NEhook:f:one:from:one
        {\romannumeral`&&@#7##3}%
       }%
    }%
    \def #3##1% \XINT_expr_check-_-<level>
    {%
      \xint_UDsignfork
        ##1{\expandafter #4\romannumeral`&&@#1}%
          -{#4##1}%
      \krof
    }%
    \def #4##1##2% \XINT_expr_checkp_-<level>
    {%
      \ifnum ##1>#5%
        \expandafter #4%
        \romannumeral`&&@\csname XINT_#6_op_##2\expandafter\endcsname
      \else
        \expandafter ##1\expandafter ##2%
      \fi
    }%
}%
\def\XINT_tmpa #1#2#3%
{%
    \expandafter\XINT_tmpb
    \csname XINT_#1_op_-#3\expandafter\endcsname
    \csname XINT_#1_exec_-#3\expandafter\endcsname
    \csname XINT_#1_check-_-#3\expandafter\endcsname
    \csname XINT_#1_checkp_-#3\expandafter\endcsname
    \csname xint_c_#3\endcsname {#1}#2%
}%
```

1.2d needs precedence 8 for *** and 9 for ^. Earlier, precedence level for ^ was only 8 but nevertheless the code did also "ix" here, which I think was unneeded back then.

```
\xintApplyInline{\XINT_tmpa {expr}\xintOpp}{{xii}{xiv}{xvi}{xviii}}%
\xintApplyInline{\XINT_tmpa {flexpr}\xintOpp}{{xii}{xiv}{xvi}{xviii}}%
\xintApplyInline{\XINT_tmpa {iiexpr}\xintiiOpp}{{xii}{xiv}{xvi}{xviii}}%
```

11.17 The * as Python-like «unpacking» prefix operator

New with 1.4. Prior to 1.4 the internal data structure was the one of \csname encapsulated comma separated numbers. No hierarchical structure was (easily) possible. At 1.4, we can use TeX braces because there is no detokenization to catcode 12.

```
\def\XINT_tmpa#1#2#3%
{%
    \def#1##1{\expandafter#2\romannumeral`&&@\XINT_expr_getnext}%
    \def#2##1##2%
    {%
      \ifnum ##1>\xint_c_xx
        \expandafter #2%
        \romannumeral`&&@\csname XINT_#3_op_##2\expandafter\endcsname
      \else
        \expandafter##1\expandafter##2\romannumeral0\expandafter\XINT:NEhook:unpack
      \fi
    }%
}%
\def\XINT:NEhook:unpack{\xint_stop_atfirstofone}%
\xintFor* #1 in {{expr}{flexpr}{iiexpr}}:
    {\expandafter\XINT_tmpa\csname XINT_#1_op_0\expandafter\endcsname
                           \csname XINT_#1_until_unpack\endcsname {#1}}%
```

11.18 Infix operators

1.2d adds the *** for tying via tacit multiplication, for example x/2y. Actually I don't need the _itself mechanism for ***, only a precedence.

1.4b subtlety with catcode of ! in \XINT_expr_itself_!=, due to chaining of comparison operators which use it to reinject into stream, but we must then have it of catcode 12 there, whereas so far the itself macros were only expanded in csname context.

```
\catcode`& 12
\xintFor* #1 in {{==}{<=}{>=}{&&}{||}{**}{//}{/:}{..}{..[}{].}{]..}}%
    \do {\expandafter\def\csname XINT_expr_itself_#1\endcsname {#1}}%
\catcode`& 7
\expandafter\edef\csname XINT_expr_itself_!=\endcsname{\string !=}%
\expandafter\let\csname XINT_expr_precedence_***\endcsname \xint_c_xvi
```

11.18.1 &&, ||, //, /:, +, −, *, /, ^, **, 'and', 'or', 'xor', and 'mod'

Usage of single character Boolean operators & and | is deprecated (for many years) and only && and || should be used. & and | will be removed at next major release after 1.4.

```
\def\XINT_expr_defbin_c #1#2#3#4#5#6#7#8%
{%
  \def #1##1% \XINT_expr_op_<op>
  {%
```

```
    \expanded{\unexpanded{#2{##1}}\expandafter}%
    \romannumeral`&&@\expandafter#3\romannumeral`&&@\XINT_expr_getnext
  }%
  \def #2##1##2##3##4% \XINT_expr_exec_<op>
  {%
    \expandafter##2\expandafter##3\expandafter
      {\romannumeral`&&@\XINT:NEhook:f:one:from:two{\romannumeral`&&@#6##1##4}}%
  }%
  \def #3##1% \XINT_expr_check-_<op>
  {%
    \xint_UDsignfork
      ##1{\expandafter#4\romannumeral`&&@#5}%
        -{#4##1}%
    \krof
  }%
  \def #4##1##2% \XINT_expr_checkp_<op>
  {%
    \ifnum ##1>#7%
      \expandafter#4%
      \romannumeral`&&@\csname XINT_#8_op_##2\expandafter\endcsname
    \else
      \expandafter ##1\expandafter ##2%
    \fi
  }%
}%
\def\XINT_expr_defbin_b #1#2#3#4#5%
{%
  \expandafter\XINT_expr_defbin_c
  \csname XINT_#1_op_#2\expandafter\endcsname
  \csname XINT_#1_exec_#2\expandafter\endcsname
  \csname XINT_#1_check-_#2\expandafter\endcsname
  \csname XINT_#1_checkp_#2\expandafter\endcsname
  \csname XINT_#1_op_-#4\expandafter\endcsname
  \csname #5\expandafter\endcsname
  \csname XINT_expr_precedence_#2\endcsname
  {#1}%
  \expandafter % done 3 times but well
  \let\csname XINT_expr_precedence_#2\expandafter\endcsname
      \csname xint_c_#3\endcsname
}%
\XINT_expr_defbin_b {expr}  {||} {vi}{xii} {xintOR}%
\XINT_expr_defbin_b {flexpr}{||} {vi}{xii} {xintOR}%
\XINT_expr_defbin_b {iiexpr}{||} {vi}{xii} {xintOR}%
\catcode`& 12
\XINT_expr_defbin_b {expr}  {&&} {viii}{xii}  {xintAND}%
\XINT_expr_defbin_b {flexpr}{&&} {viii}{xii}  {xintAND}%
\XINT_expr_defbin_b {iiexpr}{&&} {viii}{xii}  {xintAND}%
\catcode`& 7
\XINT_expr_defbin_b {expr}  {xor}{vi}{xii} {xintXOR}%
\XINT_expr_defbin_b {flexpr}{xor}{vi}{xii} {xintXOR}%
\XINT_expr_defbin_b {iiexpr}{xor}{vi}{xii} {xintXOR}%
\XINT_expr_defbin_b {expr}  {//} {xiv}{xiv}{xintDivFloor}%
```

```
\XINT_expr_defbin_b {flexpr}{//} {xiv}{xiv}{XINTinFloatDivFloor}%
\XINT_expr_defbin_b {iiexpr}{//} {xiv}{xiv}{xintiiDivFloor}%
\XINT_expr_defbin_b {expr}  {/:} {xiv}{xiv}{xintMod}%
\XINT_expr_defbin_b {flexpr}{/:} {xiv}{xiv}{XINTinFloatMod}%
\XINT_expr_defbin_b {iiexpr}{/:} {xiv}{xiv}{xintiiMod}%
\XINT_expr_defbin_b {expr}   +   {xii}{xii}{xintAdd}%
\XINT_expr_defbin_b {flexpr} +   {xii}{xii}{XINTinFloatAdd}%
\XINT_expr_defbin_b {iiexpr} +   {xii}{xii}{xintiiAdd}%
\XINT_expr_defbin_b {expr}   -   {xii}{xii}{xintSub}%
\XINT_expr_defbin_b {flexpr} -   {xii}{xii}{XINTinFloatSub}%
\XINT_expr_defbin_b {iiexpr} -   {xii}{xii}{xintiiSub}%
\XINT_expr_defbin_b {expr}   *   {xiv}{xiv}{xintMul}%
\XINT_expr_defbin_b {flexpr} *   {xiv}{xiv}{XINTinFloatMul}%
\XINT_expr_defbin_b {iiexpr} *   {xiv}{xiv}{xintiiMul}%
\XINT_expr_defbin_b {expr}   /   {xiv}{xiv}{xintDiv}%
\XINT_expr_defbin_b {flexpr} /   {xiv}{xiv}{XINTinFloatDiv}%
\XINT_expr_defbin_b {iiexpr} /   {xiv}{xiv}{xintiiDivRound}%
\XINT_expr_defbin_b {expr}   ^   {xviii}{xviii}{xintPow}%
\XINT_expr_defbin_b {flexpr} ^   {xviii}{xviii}{XINTinFloatPowerH}%
\XINT_expr_defbin_b {iiexpr} ^   {xviii}{xviii}{xintiiPow}%
\catcode`& 12
\xintFor #1 in {and,or,xor,mod} \do
{%
   \expandafter\def\csname XINT_expr_itself_#1\endcsname {#1}%
}%
\expandafter\let\csname XINT_expr_precedence_and\expandafter\endcsname
                \csname XINT_expr_precedence_&&\endcsname
\expandafter\let\csname XINT_expr_precedence_or\expandafter\endcsname
                \csname XINT_expr_precedence_||\endcsname
\expandafter\let\csname XINT_expr_precedence_mod\expandafter\endcsname
                \csname XINT_expr_precedence_/:\endcsname
\xintFor #1 in {expr, flexpr, iiexpr} \do
{%
   \expandafter\let\csname XINT_#1_op_and\expandafter\endcsname
                   \csname XINT_#1_op_&&\endcsname
   \expandafter\let\csname XINT_#1_op_or\expandafter\endcsname
                   \csname XINT_#1_op_||\endcsname
   \expandafter\let\csname XINT_#1_op_mod\expandafter\endcsname
                   \csname XINT_#1_op_/:\endcsname
}%
\expandafter\let\csname XINT_expr_precedence_&\expandafter\endcsname
                \csname XINT_expr_precedence_&&\endcsname
\expandafter\let\csname XINT_expr_precedence_|\expandafter\endcsname
                \csname XINT_expr_precedence_||\endcsname
\expandafter\let\csname XINT_expr_precedence_**\expandafter\endcsname
                \csname XINT_expr_precedence_^\endcsname
\xintFor #1 in {expr, flexpr, iiexpr} \do
{%
   \expandafter\let\csname XINT_#1_op_&\expandafter\endcsname
                   \csname XINT_#1_op_&&\endcsname
   \expandafter\let\csname XINT_#1_op_|\expandafter\endcsname
                   \csname XINT_#1_op_||\endcsname
```

```
    \expandafter\let\csname XINT_#1_op_**\expandafter\endcsname
                    \csname XINT_#1_op_^\endcsname
}%
\catcode`& 7
```

11.18.2 .., ..[, and].. for a..b and a..[b]..c syntax

The 1.4 exec_..[macros (which do no further expansion!) had silly \expandafter doing nothing for the sole reason of sharing a common \XINT_expr_defbin_c as used previously for the +, - etc... operators. At 1.4b we take the time to set things straight and do other similar simplifications.

```
\def\XINT_expr_defbin_c #1#2#3#4#5#6#7%
{%
  \def #1##1% \XINT_expr_op_..[
  {%
    \expanded{\unexpanded{#2{##1}}\expandafter}%
    \romannumeral`&&@\expandafter#3\romannumeral`&&@\XINT_expr_getnext
  }%
  \def #2##1##2##3##4% \XINT_expr_exec_..[
  {%
    ##2##3{{##1##4}}%
  }%
  \def #3##1% \XINT_expr_check-_..[
  {%
    \xint_UDsignfork
      ##1{\expandafter#4\romannumeral`&&@#5}%
        -{#4##1}%
    \krof
  }%
  \def #4##1##2% \XINT_expr_checkp_..[
  {%
    \ifnum ##1>#6%
      \expandafter#4%
      \romannumeral`&&@\csname XINT_#7_op_##2\expandafter\endcsname
    \else
      \expandafter ##1\expandafter ##2%
    \fi
  }%
}%
\def\XINT_expr_defbin_b #1%
{%
  \expandafter\XINT_expr_defbin_c
  \csname XINT_#1_op_..[\expandafter\endcsname
  \csname XINT_#1_exec_..[\expandafter\endcsname
  \csname XINT_#1_check-_..[\expandafter\endcsname
  \csname XINT_#1_checkp_..[\expandafter\endcsname
  \csname XINT_#1_op_-xii\expandafter\endcsname
  \csname XINT_expr_precedence_..[\endcsname
  {#1}%
}%
\XINT_expr_defbin_b {expr}%
\XINT_expr_defbin_b {flexpr}%
\XINT_expr_defbin_b {iiexpr}%
```

```
\expandafter\let\csname XINT_expr_precedence_..[\endcsname\xint_c_vi
\def\XINT_expr_defbin_c #1#2#3#4#5#6#7#8%
{%
  \def #1##1% \XINT_expr_op_<op>
  {%
    \expanded{\unexpanded{#2{##1}}\expandafter}%
    \romannumeral`&&@\expandafter#3\romannumeral`&&@\XINT_expr_getnext
  }%
  \def #2##1##2##3##4% \XINT_expr_exec_<op>
  {%
    \expandafter##2\expandafter##3\expanded
    {{\XINT:NEhook:x:one:from:two#8##1##4}}%
  }%
  \def #3##1% \XINT_expr_check-_<op>
  {%
    \xint_UDsignfork
      ##1{\expandafter#4\romannumeral`&&@#5}%
        -{#4##1}%
    \krof
  }%
  \def #4##1##2% \XINT_expr_checkp_<op>
  {%
    \ifnum ##1>#6%
      \expandafter#4%
      \romannumeral`&&@\csname XINT_#7_op_##2\expandafter\endcsname
    \else
      \expandafter ##1\expandafter ##2%
    \fi
  }%
}%
\def\XINT_expr_defbin_b #1#2#3%
{%
  \expandafter\XINT_expr_defbin_c
  \csname XINT_#1_op_#2\expandafter\endcsname
  \csname XINT_#1_exec_#2\expandafter\endcsname
  \csname XINT_#1_check-_#2\expandafter\endcsname
  \csname XINT_#1_checkp_#2\expandafter\endcsname
  \csname XINT_#1_op_-xii\expandafter\endcsname
  \csname XINT_expr_precedence_#2\endcsname
  {#1}#3%
  \expandafter\let
  \csname XINT_expr_precedence_#2\expandafter\endcsname\xint_c_vi
}%
\XINT_expr_defbin_b {expr}   {..}\xintSeq:tl:x
\XINT_expr_defbin_b {flexpr} {..}\xintSeq:tl:x
\XINT_expr_defbin_b {iiexpr} {..}\xintiiSeq:tl:x
\XINT_expr_defbin_b {expr}  {]..}\xintSeqB:tl:x
\XINT_expr_defbin_b {flexpr}{]..}\xintSeqB:tl:x
\XINT_expr_defbin_b {iiexpr}{]..}\xintiiSeqB:tl:x
```

11.18.3 <, >, ==, <=, >=, != with Python-like chaining

Usage of single character comparison operator = is deprecated (since many years) and only == should be used. = will be removed at next major release after 1.4.

1.4b This is preliminary implementation of chaining of comparison operators like Python and (I think) l3fp do. I am not too happy with how many times the (second) operand (already evaluated) is fetched.

```
\def\XINT_expr_defbin_d #1#2%
{%
  \def #1##1##2##3##4% \XINT_expr_exec_<op>
  {%
    \expandafter##2\expandafter##3\expandafter
      {\romannumeral`&&@\XINT:NEhook:f:one:from:two{\romannumeral`&&@#2##1##4}}%
  }%
}%
\def\XINT_expr_defbin_c #1#2#3#4#5#6#7#8#9%
{%
  \def #1##1% \XINT_expr_op_<op>
  {%
    \expanded{\unexpanded{#2{##1}}\expandafter}%
    \romannumeral`&&@\expandafter#7%
    \romannumeral`&&@\expandafter#3\romannumeral`&&@\XINT_expr_getnext
  }%
  \def #3##1% \XINT_expr_check-_<op>
  {%
    \xint_UDsignfork
      ##1{\expandafter#4\romannumeral`&&@#5}%
        -{#4##1}%
    \krof
  }%
  \def #4##1##2% \XINT_expr_checkp_<op>
  {%
    \ifnum ##1>#6%
      \expandafter#4%
      \romannumeral`&&@\csname XINT_#9_op_##2\expandafter\endcsname
    \else
      \expandafter ##1\expandafter ##2%
    \fi
  }%
  \let #6\xint_c_x
  \def #7##1% \XINT_expr_checkc_<op>
  {%
    \ifnum ##1=\xint_c_x\expandafter#8\fi ##1%
  }%
  \edef #8##1##2##3% \XINT_expr_execc_<op>
  {%
    \csname XINT_#9_precedence_\string&\string&\endcsname
    \expandafter\noexpand\csname XINT_#9_itself_\string&\string&\endcsname
    {##3}%
    \XINTfstop.{##3}##2%
  }%
  \XINT_expr_defbin_d #2% \XINT_expr_exec_<op>
}%
```

```
\def\XINT_expr_defbin_b #1#2%#3%
{%
  \expandafter\XINT_expr_defbin_c
  \csname XINT_#1_op_#2\expandafter\endcsname
  \csname XINT_#1_exec_#2\expandafter\endcsname
  \csname XINT_#1_check-_#2\expandafter\endcsname
  \csname XINT_#1_checkp_#2\expandafter\endcsname
  \csname XINT_#1_op_-xii\expandafter\endcsname
  \csname XINT_expr_precedence_#2\expandafter\endcsname
  \csname XINT_#1_checkc_#2\expandafter\endcsname
  \csname XINT_#1_execc_#2\endcsname
  {#1}%#3%
}%
```

Attention that third token here is left in stream by defbin_b, then also by defbin_c and is picked up as #2 of defbin_d. Had to work around TeX accepting only 9 arguments. Why did it not start counting at #0 like all decent mathematicians do?

```
\XINT_expr_defbin_b {expr}  <\xintLt
\XINT_expr_defbin_b {flexpr}<\xintLt
\XINT_expr_defbin_b {iiexpr}<\xintiiLt
\XINT_expr_defbin_b {expr}  >\xintGt
\XINT_expr_defbin_b {flexpr}>\xintGt
\XINT_expr_defbin_b {iiexpr}>\xintiiGt
\XINT_expr_defbin_b {expr}  {==}\xintEq
\XINT_expr_defbin_b {flexpr}{==}\xintEq
\XINT_expr_defbin_b {iiexpr}{==}\xintiiEq
\XINT_expr_defbin_b {expr}  {<=}\xintLtorEq
\XINT_expr_defbin_b {flexpr}{<=}\xintLtorEq
\XINT_expr_defbin_b {iiexpr}{<=}\xintiiLtorEq
\XINT_expr_defbin_b {expr}  {>=}\xintGtorEq
\XINT_expr_defbin_b {flexpr}{>=}\xintGtorEq
\XINT_expr_defbin_b {iiexpr}{>=}\xintiiGtorEq
\XINT_expr_defbin_b {expr}  {!=}\xintNotEq
\XINT_expr_defbin_b {flexpr}{!=}\xintNotEq
\XINT_expr_defbin_b {iiexpr}{!=}\xintiiNotEq
\expandafter\let\csname XINT_expr_precedence_=\endcsname\xint_c_x
\xintFor #1 in {expr, flexpr, iiexpr} \do
{%
   \expandafter\let\csname XINT_#1_op_=\expandafter\endcsname
                    \csname XINT_#1_op_==\endcsname
}%
```

11.18.4 Support macros for .., ..[and]..

\xintSeq:tl:x Commence par remplacer a par ceil(a) et b par floor(b) et renvoie ensuite les entiers entre les deux, possiblement en décroissant, et extrémités comprises. Si a=b est non entier en obtient donc ceil(a) et floor(a). Ne renvoie jamais une liste vide.

Note: le a..b dans \xintfloatexpr utilise cette routine.

```
\def\xintSeq:tl:x #1#2%
{%
    \expandafter\XINT_Seq:tl:x
```

```
    \the\numexpr \xintiCeil{#1}\expandafter.\the\numexpr \xintiFloor{#2}.%
}%
\def\XINT_Seq:tl:x #1.#2.%
{%
   \ifnum #2=#1 \xint_dothis\XINT_Seq:tl:x_z\fi
   \ifnum #2<#1 \xint_dothis\XINT_Seq:tl:x_n\fi
   \xint_orthat\XINT_Seq:tl:x_p
   #1.#2.%
}%
\def\XINT_Seq:tl:x_z #1.#2.{{#1/1[0]}}%
\def\XINT_Seq:tl:x_p #1.#2.%
{%
    {#1/1[0]}\ifnum #1=#2 \XINT_Seq:tl:x_e\fi
    \expandafter\XINT_Seq:tl:x_p \the\numexpr #1+\xint_c_i.#2.%
}%
\def\XINT_Seq:tl:x_n #1.#2.%
{%
    {#1/1[0]}\ifnum #1=#2 \XINT_Seq:tl:x_e\fi
    \expandafter\XINT_Seq:tl:x_n \the\numexpr #1-\xint_c_i.#2.%
}%
\def\XINT_Seq:tl:x_e#1#2.#3.{#1}%
```

\xintiiSeq:tl:x

```
\def\xintiiSeq:tl:x #1#2%
{%
    \expandafter\XINT_iiSeq:tl:x
    \the\numexpr \xintiCeil{#1}\expandafter.\the\numexpr \xintiFloor{#2}.%
}%
\def\XINT_iiSeq:tl:x #1.#2.%
{%
   \ifnum #2=#1 \xint_dothis\XINT_iiSeq:tl:x_z\fi
   \ifnum #2<#1 \xint_dothis\XINT_iiSeq:tl:x_n\fi
   \xint_orthat\XINT_iiSeq:tl:x_p
   #1.#2.%
}%
\def\XINT_iiSeq:tl:x_z #1.#2.{{#1}}%
\def\XINT_iiSeq:tl:x_p #1.#2.%
{%
    {#1}\ifnum #1=#2 \XINT_Seq:tl:x_e\fi
    \expandafter\XINT_iiSeq:tl:x_p \the\numexpr #1+\xint_c_i.#2.%
}%
\def\XINT_iiSeq:tl:x_n #1.#2.%
{%
    {#1}\ifnum #1=#2 \XINT_Seq:tl:x_e\fi
    \expandafter\XINT_iiSeq:tl:x_n \the\numexpr #1-\xint_c_i.#2.%
}%
```

Contrarily to a..b which is limited to small integers, this works with a, b, and d (big) fractions. It will produce a «nil» list, if a>b and d<0 or a<b and d>0.

\xintSeqA, \xintiiSeqA

```
\def\xintSeqA        {\expandafter\XINT_SeqA\romannumeral0\xintraw}%
```

```
\def\xintiiSeqA    #1{\expandafter\XINT_iiSeqA\romannumeral`&&@#1;}%
\def\XINT_SeqA  #1]#2{\expandafter\XINT_SeqA_a\romannumeral0\xintraw {#2}#1]}%
\def\XINT_iiSeqA#1;#2{\expandafter\XINT_SeqA_a\romannumeral`&&@#2;#1;}%
\def\XINT_SeqA_a #1{\xint_UDzerominusfork
                                    #1-{z}%
                                    0#1{n}%
                                    0-{p}%
                     \krof #1}%
```

\xintSeqB:tl:x At 1.4, delayed expansion of start and step done here and not before, for matters of \xintdeffunc and «NEhooks».

The float variant at 1.4 is made identical to the exact variant. I.e. stepping is exact and comparison to the range limit too. But recall that a/b input will be converted to a float. To handle 1/3 step for example still better to use \xintexpr 1..1/3..10\relax for example inside the \xintfloateval.

```
\def\xintSeqB:tl:x  #1{\expandafter\XINT_SeqB:tl:x\romannumeral`&&@\xintSeqA#1}%
\def\XINT_SeqB:tl:x #1{\csname XINT_SeqB#1:tl:x\endcsname}%
\def\XINT_SeqBz:tl:x #1]#2]#3{{#2]}}%
\def\XINT_SeqBp:tl:x #1]#2]#3{\expandafter\XINT_SeqBp:tl:x_a\romannumeral0\xintraw{#3}#2]#1]}%
\def\XINT_SeqBp:tl:x_a #1]#2]#3]%
{%
    \xintifCmp{#1]}{#2]}%
    {}{{#2]}}{{#2]}\expandafter\XINT_SeqBp:tl:x_b\romannumeral0\xintadd{#3]}{#2]}#1]#3]}%
}%
\def\XINT_SeqBp:tl:x_b #1]#2]#3]%
{%
    \xintifCmp{#1]}{#2]}%
    {{#1]}\expandafter\XINT_SeqBp:tl:x_b\romannumeral0\xintadd{#3]}{#1]}#2]#3]}{{#1]}}{}%
}%
\def\XINT_SeqBn:tl:x #1]#2]#3{\expandafter\XINT_SeqBn:tl:x_a\romannumeral0\xintraw{#3}#2]#1]}%
\def\XINT_SeqBn:tl:x_a #1]#2]#3]%
{%
    \xintifCmp{#1]}{#2]}%
    {{#2]}\expandafter\XINT_SeqBn:tl:x_b\romannumeral0\xintadd{#3]}{#2]}#1]#3]}{{#2]}}{}%
}%
\def\XINT_SeqBn:tl:x_b #1]#2]#3]%
{%
    \xintifCmp{#1]}{#2]}%
    {}{{#1]}}{{#1]}\expandafter\XINT_SeqBn:tl:x_b\romannumeral0\xintadd{#3]}{#1]}#2]#3]}%
}%
```

\xintiiSeqB:tl:x

```
\def\xintiiSeqB:tl:x  #1{\expandafter\XINT_iiSeqB:tl:x\romannumeral`&&@\xintiiSeqA#1}%
\def\XINT_iiSeqB:tl:x #1{\csname XINT_iiSeqB#1:tl:x\endcsname}%
\def\XINT_iiSeqBz:tl:x #1;#2;#3{{#2}}%
\def\XINT_iiSeqBp:tl:x #1;#2;#3{\expandafter\XINT_iiSeqBp:tl:x_a\romannumeral`&&@#3;#2;#1;}%
\def\XINT_iiSeqBp:tl:x_a #1;#2;#3;%
{%
    \xintiiifCmp{#1}{#2}%
    {}{{#2}}{{#2}\expandafter\XINT_iiSeqBp:tl:x_b\romannumeral0\xintiiadd{#3}{#2};#1;#3;}%
}%
```

```
\def\XINT_iiSeqBp:tl:x_b #1;#2;#3;%
{%
    \xintiiifCmp{#1}{#2}%
    {{#1}\expandafter\XINT_iiSeqBp:tl:x_b\romannumeral0\xintiiadd{#3}{#1};#2;#3;}{{#1}}{}%
}%
\def\XINT_iiSeqBn:tl:x #1;#2;#3{\expandafter\XINT_iiSeqBn:tl:x_a\romannumeral`&&@#3;#2;#1;}%
\def\XINT_iiSeqBn:tl:x_a #1;#2;#3;%
{%
    \xintiiifCmp{#1}{#2}%
    {{#2}\expandafter\XINT_iiSeqBn:tl:x_b\romannumeral0\xintiiadd{#3}{#2};#1;#3;}{{#2}}{}%
}%
\def\XINT_iiSeqBn:tl:x_b #1;#2;#3;%
{%
    \xintiiifCmp{#1}{#2}%
    {}{{#1}}{{#1}\expandafter\XINT_iiSeqBn:tl:x_b\romannumeral0\xintiiadd{#3}{#1};#2;#3;}%
}%
```

11.19 Square brackets [] both as a container and a Python slicer

Refactored at 1.4

The architecture allows to implement separately a «left» and a «right» precedence and this is crucial.

11.19.1 [...] as «oneple» constructor

In the definition of \XINT_expr_op_obracket the parameter is trash {}. The [is intercepted by the getnextfork and handled via the \xint_c_ii^v highest precedence trick to get op_obracket executed.

```
\def\XINT_expr_itself_obracket{obracket}%
\catcode`] 11 \catcode`[ 11
\def\XINT_expr_defbin_c #1#2#3#4#5#6%
{%
    \def #1##1%
    {%
      \expandafter#3\romannumeral`&&@\XINT_expr_getnext
    }%
    \def #2##1% op_]
    {%
      \expanded{\unexpanded{\XINT_expr_put_op_first{{##1}}}\expandafter}%
         \romannumeral`&&@\XINT_expr_getop
    }%
    \def #3##1%  until_cbracket_a
    {%
      \xint_UDsignfork
         ##1{\expandafter#4\romannumeral`&&@#5}%  #5 = op_-xii
           -{#4##1}%
      \krof
    }%
```

```
    \def #4##1##2%  until_cbracket_b
    {%
      \ifcase ##1\expandafter\XINT_expr_missing_]
      \or \expandafter\XINT_expr_missing_]
      \or \expandafter#2%
      \else
      \expandafter #4%
        \romannumeral`&&@\csname XINT_#6_op_##2\expandafter\endcsname
      \fi
    }%
}%
\def\XINT_expr_defbin_b #1%
{%
    \expandafter\XINT_expr_defbin_c
    \csname XINT_#1_op_obracket\expandafter\endcsname
    \csname XINT_#1_op_]\expandafter\endcsname
    \csname XINT_#1_until_cbracket_a\expandafter\endcsname
    \csname XINT_#1_until_cbracket_b\expandafter\endcsname
    \csname XINT_#1_op_-xii\endcsname
    {#1}%
}%
\XINT_expr_defbin_b {expr}%
\XINT_expr_defbin_b {flexpr}%
\XINT_expr_defbin_b {iiexpr}%
\def\XINT_expr_missing_]
   {\XINT_expandableerror{Ooops, looks like we are missing a ] here. Goodbye!}%
    \xint_c_ \XINT_expr_done}%
\let\XINT_expr_precedence_]\xint_c_ii
```

11.19.2 [...] brackets and : operator for NumPy-like slicing and item indexing syntax

The opening bracket [for the nutple constructor is filtered out by \XINT_expr_getnextfork and becomes «obracket» which behaves with precedence level 2. For the [..] Python slicer on the other hand, a real operator [is defined with precedence level 4 (it must be higher than precedence level of commas) on its right and maximal precedence on its left.

Important: although slicing and indexing shares many rules with Python/NumPy there are some significant differences: in particular there can not be any out-of-range error generated, slicing applies also to «oples» and not only to «nutple», and nested lists do not have to have their leaves at a constant depth. See the user manual.

Currently, NumPy-like nested (basic) slicing is implemented, i.e [a:b, c:d, N, e:f, M] type syntax with Python rules regarding negative integers. This is parsed as an expression and can arise from expansion or contain calculations.

Currently stepping, Ellipsis, and simultaneous multi-index extracting are not yet implemented. There are some subtle things here with possibility of variables been passed by reference.

```
\def\XINT_expr_defbin_c #1#2#3#4#5#6%
{%
    \def #1##1% \XINT_expr_op_[
    {%
        \expanded{\unexpanded{#2{##1}}\expandafter}%
        \romannumeral`&&@\expandafter#3\romannumeral`&&@\XINT_expr_getnext
    }%
    \def #2##1##2##3##4% \XINT_expr_exec_]
```

```
    {%
         \expandafter\XINT_expr_put_op_first
         \expanded
          {%
           {\XINT:NEhook:x:listsel\XINT_ListSel_top ##1__##4&({##1}\expandafter}%
           \expandafter
          }%
         \romannumeral`&&@\XINT_expr_getop
    }%
    \def #3##1% \XINT_expr_check-_]
    {%
      \xint_UDsignfork
        ##1{\expandafter#4\romannumeral`&&@#5}%
          -{#4##1}%
      \krof
    }%
    \def #4##1##2% \XINT_expr_checkp_]
    {%
      \ifcase ##1\XINT_expr_missing_]
        \or \XINT_expr_missing_]
        \or \expandafter##1\expandafter##2%
        \else \expandafter#4%
              \romannumeral`&&@\csname XINT_#6_op_##2\expandafter\endcsname
      \fi
    }%
}%
\let\XINT_expr_precedence_[ \xint_c_xx
\def\XINT_expr_defbin_b #1%
{%
    \expandafter\XINT_expr_defbin_c
    \csname XINT_#1_op_[\expandafter\endcsname
    \csname XINT_#1_exec_]\expandafter\endcsname
    \csname XINT_#1_check-_]\expandafter\endcsname
    \csname XINT_#1_checkp_]\expandafter\endcsname
    \csname XINT_#1_op_-xii\endcsname
    {#1}%
}%
\XINT_expr_defbin_b {expr}%
\XINT_expr_defbin_b {flexpr}%
\XINT_expr_defbin_b {iiexpr}%
\catcode`] 12 \catcode`[ 12
```

At 1.4 the getnext, scanint, scanfunc, getop chain got revisited to trigger automatic insertion of the nil variable if needed, without having in situations like here to define operators to support «[:» or «:]». And as we want to implement nested slicing à la NumPy, we would have had to handle also «:,» for example. Thus here we simply have to define the sole operator «:» and it will be some sort of inert joiner preparing a slicing spec.

```
\def\XINT_expr_defbin_c #1#2#3#4#5#6%
{%
    \def #1##1% \XINT_expr_op_:
    {%
        \expanded{\unexpanded{#2{##1}}\expandafter}%
```

```
1530         \romannumeral`&&@\expandafter#3\romannumeral`&&@\XINT_expr_getnext
1531     }%
1532     \def #2##1##2##3##4% \XINT_expr_exec_:
1533     {%
1534        ##2##3{:##1{0};##4:_}%
1535     }%
1536     \def #3##1% \XINT_expr_check-_:
1537     {\xint_UDsignfork
1538         ##1{\expandafter#4\romannumeral`&&@#5}%
1539           -{#4##1}%
1540      \krof
1541     }%
1542     \def #4##1##2% \XINT_expr_checkp_:
1543     {%
1544       \ifnum ##1>\XINT_expr_precedence_:
1545         \expandafter #4\romannumeral`&&@%
1546                        \csname XINT_#6_op_##2\expandafter\endcsname
1547       \else
1548         \expandafter##1\expandafter##2%
1549       \fi
1550     }%
1551 }%
1552 \let\XINT_expr_precedence_: \xint_c_vi
1553 \def\XINT_expr_defbin_b #1%
1554 {%
1555     \expandafter\XINT_expr_defbin_c
1556     \csname XINT_#1_op_:\expandafter\endcsname
1557     \csname XINT_#1_exec_:\expandafter\endcsname
1558     \csname XINT_#1_check-_:\expandafter\endcsname
1559     \csname XINT_#1_checkp_:\expandafter\endcsname
1560     \csname XINT_#1_op_-xii\endcsname {#1}%
1561 }%
1562 \XINT_expr_defbin_b {expr}%
1563 \XINT_expr_defbin_b {flexpr}%
1564 \XINT_expr_defbin_b {iiexpr}%
```

11.19.3 Macro layer implementing indexing and slicing

xintexpr applies slicing not only to «objects» (which can be passed as arguments to functions) but also to «oples».

Our «nlists» are not necessarily regular N-dimensional arrays à la NumPy. Leaves can be at arbitrary depths. If we were handling regular «ndarrays», we could proceed a bit differently.

For the related explanations, refer to the user manual.

Notice that currently the code uses f-expandable (and not using \expanded) macros \xintApply, \xintApplyUnbraced, \xintKeep, \xintTrim, \xintNthOne from xinttools.

But the whole expansion happens inside an \expanded context, so possibly some gain could be achieved with x-expandable variants (xintexpr < 1.4 had an \xintKeep:x:csv).

I coded \xintApply:x and \xintApplyUnbraced:x in xinttools, Brief testing indicated they were perhaps a bit better for 5x5x5x5 and 15x15x15x15 arrays of 8 digits numbers and for 30x30x15 with 16 digits numbers: say 1% gain... this seems to raise to between 4% and 5% for 400x400 array of 1 digit...

Currently sticking with old macros.

```
\def\XINT_ListSel_deeper #1%
{%
    \if :#1\xint_dothis\XINT_ListSel_slice_next\fi
    \xint_orthat {\XINT_ListSel_extract_next {#1}}%
}%
\def\XINT_ListSel_slice_next #1(%
{%
    \xintApply{\XINT_ListSel_recurse{:#1}}%
}%
\def\XINT_ListSel_extract_next #1(%
{%
    \xintApplyUnbraced{\XINT_ListSel_recurse{#1}}%
}%
\def\XINT_ListSel_recurse #1#2%
{%
    \XINT_ListSel_check #2__#1({#2}\expandafter\empty\empty
}%
\def\XINT_ListSel_check{\expandafter\XINT_ListSel_check_a \string}%
\def\XINT_ListSel_check_a #1%
{%
    \if #1\bgroup\xint_dothis\XINT_ListSel_check_is_ok\fi
    \xint_orthat\XINT_ListSel_check_leaf
}%
\def\XINT_ListSel_check_leaf #1\expandafter{\expandafter}%
\def\XINT_ListSel_check_is_ok
{%
    \expandafter\XINT_ListSel_check_is_ok_a\expandafter{\string}%
}%
\def\XINT_ListSel_check_is_ok_a #1__#2%
{%
    \if :#2\xint_dothis{\XINT_ListSel_slice}\fi
    \xint_orthat {\XINT_ListSel_nthone {#2}}%
}%
\def\XINT_ListSel_top #1#2%
{%
    \if _\noexpand#2%
           \expandafter\XINT_ListSel_top_one_or_none\string#1.\else
           \expandafter\XINT_ListSel_top_at_least_two\fi
}%
\def\XINT_ListSel_top_at_least_two #1__{\XINT_ListSel_top_ople}%
\def\XINT_ListSel_top_one_or_none #1%
{%
    \if #1_\xint_dothis\XINT_ListSel_top_nil\fi
    \if #1.\xint_dothis\XINT_ListSel_top_nutple_a\fi
    \if #1\bgroup\xint_dothis\XINT_ListSel_top_nutple\fi
    \xint_orthat\XINT_ListSel_top_number
}%
\def\XINT_ListSel_top_nil #1\expandafter#2\expandafter{\fi\expandafter}%
\def\XINT_ListSel_top_nutple
{%
    \expandafter\XINT_ListSel_top_nutple_a\expandafter{\string}%
}%
```

```
\def\XINT_ListSel_top_nutple_a #1_#2#3(#4%
{%
    \fi\if :#2\xint_dothis{{\XINT_ListSel_slice #3(#4}}\fi
    \xint_orthat {\XINT_ListSel_nthone {#2}#3(#4}%
}%
\def\XINT_ListSel_top_number #1_{\fi\XINT_ListSel_top_ople}%
\def\XINT_ListSel_top_ople #1%
{%
    \if :#1\xint_dothis\XINT_ListSel_slice\fi
    \xint_orthat {\XINT_ListSel_nthone {#1}}%
}%
\def\XINT_ListSel_slice #1%
{%
    \expandafter\XINT_ListSel_slice_a \expandafter{\romannumeral0\xintnum{#1}}%
}%
\def\XINT_ListSel_slice_a #1#2;#3#4%
{%
   \if _#4\expandafter\XINT_ListSel_s_b
          \else\expandafter\XINT_ListSel_slice_b\fi
   #1;#3%
}%
\def\XINT_ListSel_s_b #1#2;#3#4%
{%
    \if &#4\expandafter\XINT_ListSel_s_last\fi
    \XINT_ListSel_s_c #1{#1#2}{#4}%
}%
\def\XINT_ListSel_s_last\XINT_ListSel_s_c #1#2#3(#4%
{%
   \if-#1\expandafter\xintKeep\else\expandafter\xintTrim\fi {#2}{#4}%
}%
\def\XINT_ListSel_s_c #1#2#3(#4%
{%
   \expandafter\XINT_ListSel_deeper
   \expanded{\unexpanded{#3}(\expandafter}\expandafter{%
   \romannumeral0%
   \if-#1\expandafter\xintkeep\else\expandafter\xinttrim\fi {#2}{#4}}%
}%
```

  \xintNthElt from xinttools (knowingly) strips one level of braces when fetching kth «item» from {v1}...{vN}. If we expand {\xintNthElt{k}{{v1}...{vN}}} (notice external braces):
   if k is out of range we end up with {}
   if k is in range and the kth braced item was {} we end up with {}
   if k is in range and the kth braced item was {17} we end up with {17}
Problem is that individual numbers such as 17 are stored {{17}}. So we must have one more brace pair and in the first two cases we end up with {{}}. But in the first case we should end up with the empty ople {}, not the empty bracketed ople {{}}.
  I have thus added \xintNthOne to xinttools which does not strip brace pair from an extracted item.
  Attention: \XINT_nthonepy_a does no expansion on second argument. But here arguments are either numerical or already expanded. Normally.

```
\def\XINT_ListSel_nthone #1#2%
{%
```

```
    \if &#2\expandafter\XINT_ListSel_nthone_last\fi
    \XINT_ListSel_nthone_a {#1}{#2}%
}%
\def\XINT_ListSel_nthone_a #1#2(#3%
{%
    \expandafter\XINT_ListSel_deeper
    \expanded{\unexpanded{#2}(\expandafter}\expandafter{%
    \romannumeral0\expandafter\XINT_nthonepy_a\the\numexpr\xintNum{#1}.{#3}}%
}%
\def\XINT_ListSel_nthone_last\XINT_ListSel_nthone_a #1#2(%#3%
{%
    \romannumeral0\expandafter\XINT_nthonepy_a\the\numexpr\xintNum{#1}.%{#3}
}%
```

The macros here are basically f-expandable and use the f-expandable \xintKeep and \xintTrim. Prior to xint 1.4, there was here an x-expandable \xintKeep:x:csv dealing with comma separated items, for time being we make do with our f-expandable toolkit.

```
\def\XINT_ListSel_slice_b #1;#2_#3%
{%
    \if &#3\expandafter\XINT_ListSel_slice_last\fi
    \expandafter\XINT_ListSel_slice_c \expandafter{\romannumeral0\xintnum{#2}};#1;{#3}%
}%
\def\XINT_ListSel_slice_last\expandafter\XINT_ListSel_slice_c #1;#2;#3(%#4
{%
   \expandafter\XINT_ListSel_slice_last_c #1;#2;%{#4}
}%
\def\XINT_ListSel_slice_last_c #1;#2;#3%
{%
    \romannumeral0\XINT_ListSel_slice_d #2;#1;{#3}%
}%
\def\XINT_ListSel_slice_c #1;#2;#3(#4%
{%
   \expandafter\XINT_ListSel_deeper
   \expanded{\unexpanded{#3}(\expandafter}\expandafter{%
   \romannumeral0\XINT_ListSel_slice_d #2;#1;{#4}}%
}%
\def\XINT_ListSel_slice_d #1#2;#3#4;%
{%
    \xint_UDsignsfork
       #1#3\XINT_ListSel_N:N
        #1-\XINT_ListSel_N:P
        -#3\XINT_ListSel_P:N
         --\XINT_ListSel_P:P
    \krof #1#2;#3#4;%
}%
\def\XINT_ListSel_P:P #1;#2;#3%
{%
    \unless\ifnum #1<#2 \expandafter\xint_gob_andstop_iii\fi
    \xintkeep{#2-#1}{\xintTrim{#1}{#3}}%
}%
\def\XINT_ListSel_N:N #1;#2;#3%
{%
```

```
    \expandafter\XINT_ListSel_N:N_a
    \the\numexpr #2-#1\expandafter;\the\numexpr#1+\xintLength{#3};{#3}%
}%
\def\XINT_ListSel_N:N_a #1;#2;#3%
{%
    \unless\ifnum #1>\xint_c_ \expandafter\xint_gob_andstop_iii\fi
    \xintkeep{#1}{\xintTrim{\ifnum#2<\xint_c_\xint_c_\else#2\fi}{#3}}%
}%
\def\XINT_ListSel_N:P #1;#2;#3%
{%
    \expandafter\XINT_ListSel_N:P_a
    \the\numexpr #1+\xintLength{#3};#2;{#3}%
}%
\def\XINT_ListSel_N:P_a #1#2;%
   {\if -#1\expandafter\XINT_ListSel_O:P\fi\XINT_ListSel_P:P #1#2;}%
\def\XINT_ListSel_O:P\XINT_ListSel_P:P #1;{\XINT_ListSel_P:P 0;}%
\def\XINT_ListSel_P:N #1;#2;#3%
{%
    \expandafter\XINT_ListSel_P:N_a
    \the\numexpr #2+\xintLength{#3};#1;{#3}%
}%
\def\XINT_ListSel_P:N_a #1#2;#3;%
   {\if -#1\expandafter\XINT_ListSel_P:O\fi\XINT_ListSel_P:P #3;#1#2;}%
\def\XINT_ListSel_P:O\XINT_ListSel_P:P #1;#2;{\XINT_ListSel_P:P #1;0;}%
```

11.20 Support for raw A/B[N]

Releases earlier than 1.1 required the use of braces around A/B[N] input. The [N] is now implemented directly. *BUT* this uses a delimited macro! thus N is not allowed to be itself an expression (I could add it...). \xintE, \xintiiE, and \XINTinFloatE all put #2 in a \numexpr. But attention to the fact that \numexpr stops at spaces separating digits: \the\numexpr 3 + 7 9\relax gives 109\relax !! Hence we have to be careful.

\numexpr will not handle catcode 11 digits, but adding a \detokenize will suddenly make illicit for N to rely on macro expansion.

At 1.4, [is already overloaded and it is not easy to support this. We do this by a kludge maintaining more or less former (very not efficient) way but using $ sign which is free for time being. No, finally I use the null character, should be safe enough! (I hesitated about using R with catcode 12).

As for ? operator we needed to hack into \XINT_expr_getop_b for intercepting that pseudo operator. See also \XINT_expr_scanint_c (\XINT_expr_rawxintfrac).

```
\catcode0 11
\let\XINT_expr_precedence_&&@ \xint_c_xiv
\def\XINT_expr_op_&&@ #1#2]%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded{{{\xintE#1{\xint_zapspaces #2 \xint_gobble_i}}}%
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT_iiexpr_op_&&@ #1#2]%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded{{{\xintiiE#1{\xint_zapspaces #2 \xint_gobble_i}}}%
```

```
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT_flexpr_op_&&@ #1#2]%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded{{{\XINTinFloatE#1{\xint_zapspaces #2 \xint_gobble_i}}}%
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\catcode0 12
```

11.21 ? as two-way and ?? as three-way «short-circuit» conditionals

Comments undergoing reconstruction.

```
\let\XINT_expr_precedence_? \xint_c_xx
\catcode`- 11
\def\XINT_expr_op_?  {\XINT_expr_op__? \XINT_expr_op_-xii}%
\def\XINT_flexpr_op_?{\XINT_expr_op__? \XINT_flexpr_op_-xii}%
\def\XINT_iiexpr_op_?{\XINT_expr_op__? \XINT_iiexpr_op_-xii}%
\catcode`- 12
\def\XINT_expr_op__? #1#2#3%
    {\XINT_expr_op__?_a #3!\xint_bye\XINT_expr_exec_? {#1}{#2}{#3}}%
\def\XINT_expr_op__?_a #1{\expandafter\XINT_expr_op__?_b\detokenize{#1}}%
\def\XINT_expr_op__?_b #1%
    {\if ?#1\expandafter\XINT_expr_op__?_c\else\expandafter\xint_bye\fi }%
\def\XINT_expr_op__?_c #1{\xint_gob_til_! #1\XINT_expr_op_?? !\xint_bye}%
\def\XINT_expr_op_?? !\xint_bye\xint_bye\XINT_expr_exec_?{\XINT_expr_exec_??}%
\catcode`- 11
\def\XINT_expr_exec_? #1#2%
{%
    \expandafter\XINT_expr_check-_after?\expandafter#1%
    \romannumeral`&&@\expandafter\XINT_expr_getnext\romannumeral0\xintiiifnotzero#2%
}%
\def\XINT_expr_exec_?? #1#2#3%
{%
    \expandafter\XINT_expr_check-_after?\expandafter#1%
    \romannumeral`&&@\expandafter\XINT_expr_getnext\romannumeral0\xintiiifsgn#2%
}%
\def\XINT_expr_check-_after? #1{%
\def\XINT_expr_check-_after? ##1##2%
{%
    \xint_UDsignfork
        ##2{##1}%
        #1{##2}%
    \krof
}}\expandafter\XINT_expr_check-_after?\string -%
\catcode`- 12
```

11.22 ! as postfix factorial operator

```
\let\XINT_expr_precedence_! \xint_c_xx
\def\XINT_expr_op_! #1%
```

```
{%
    \expandafter\XINT_expr_put_op_first
    \expanded{{\romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintFac#1}}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT_flexpr_op_! #1%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded{{\romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\XINTinFloatFac#1}}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT_iiexpr_op_! #1%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded{{\romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiFac#1}}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

11.23 User defined variables

11.23.1 \xintdefvar, \xintdefiivar, \xintdeffloatvar

1.1.

1.2p (2017/12/01). extends \xintdefvar et al. to accept simultaneous assignments to multiple variables.

1.3c (2018/06/17). Use \xintexprSafeCatcodes (to palliate issue with active semi-colon from Babel+French if in body of a LaTeX document).

And allow usage with both syntaxes name:=expr; or name=expr;. Also the colon may have catcode 11, 12, or 13 with no issue. Variable names may contain letters, digits, underscores, and must not start with a digit. Names starting with @ or an underscore are reserved.

- currently @, @1, @2, @3, and @4 are reserved because they have special meanings for use in iterations,
- @@, @@@, @@@@ are also reserved but are technically functions, not variables: a user may possibly define @@ as a variable name, but if it is followed by parentheses, the function interpretation will be applied (rather than the variable interpretation followed by a tacit multiplication),
- since 1.2l, the underscore _ may be used as separator of digits in long numbers. Hence a variable whose name starts with _ will not play well with the mechanism of tacit multiplication of variables by numbers: the underscore will be removed from input stream by the number scanner, thus creating an undefined or wrong variable name, or none at all if the variable name was an initial _ followed by digits.

Note that the optional argument [P] as usable with \xintfloatexpr is **not** supported by \xintdeffloatvar. One must do \xintdeffloatvar foo = \xintfloatexpr[16] blabla \relax; to achieve the effect.

1.4 (2020/01/27). The expression will be fetched up to final semi-colon in a manner allowing inner semi-colons as used in the iter(), rseq(), subsm(), subsn() etc... syntax. They don't need to be hidden within a braced pair anymore.

TODO: prior to 1.4 a variable «value» was passed along as a single token. Now it is managed, like everything else, as explicit braced contents. But most of the code is ready for passing it along

again as a single (braced, now) token again, because all needed \expanded/\unexpanded things are in place. However this is «most of the code». I am really eager to get 1.4 released now, because I can't devote more time in immediate future. It is too late to engage into an umpteenth deep refactoring at a time where things work and many new features were added and most aspects of inner working got adapted. However in future it could be that variables holding large data will be managed much faster.

1.4c 2021/02/20. One year later I realized I had broken tacit multiplication for situations such as variable(1+2). As hinted at in comments above before 1.4 release I had been doing some deep refactoring here, which I cancelled almost completely in the end... but not quite, and as a result there was a problem that some macro holding braced contents was expanded to late, once it was in old core routines of xintfrac not expecting other things than digits. I do an emergency bugfix here with some \expandafter's but I don't have the code in my brain at this time, and don't have the luxury now to invest into it. Let's hope this does not induce breakage elsewhere, and that the February 2020 1.4 did not break something else.

```
\catcode`* 11
\def\XINT_expr_defvar_one #1#2%
{%
    \XINT_global
    \expandafter\edef\csname XINT_expr_varvalue_#1\endcsname {#2}%
    \XINT_expr_defvar_one_b {#1}%
}%
\def\XINT_expr_defvar_one_b #1%
{%
     \XINT_global
     \expandafter\edef\csname XINT_expr_var_#1\endcsname
        {{\expandafter\noexpand\csname XINT_expr_varvalue_#1\endcsname}}%
    \XINT_global
    \expandafter\edef\csname XINT_expr_onliteral_#1\endcsname
        {\unexpanded{\expandafter\expandafter\expandafter}%
         \XINT_expr_precedence_***
         \unexpanded{\expandafter\expandafter\expandafter}%
         *\unexpanded{\expandafter\expandafter}%
         \expandafter\noexpand\csname XINT_expr_var_#1\endcsname(}%
    \ifxintverbose\xintMessage{xintexpr}{Info}
       {Variable #1 \ifxintglobaldefs globally \fi
        defined with value \csname XINT_expr_varvalue_#1\endcsname.}%
    \fi
}%
\catcode`* 12
\catcode`~ 13
\catcode`: 12
\def\XINT_expr_defvar_getname #1:#2~%
{%
    \endgroup
    \def\XINT_defvar_tmpa{#1}\edef\XINT_defvar_tmpc{\xintCSVLength{#1}}%
}%
\def\XINT_expr_defvar #1#2%
{%
    \def\XINT_defvar_tmpa{#2}%
    \expandafter\XINT_expr_defvar_a\expandafter#1\romannumeral\XINT_expr_fetch_to_semicolon
}%
\def\XINT_expr_defvar_a #1#2%
```

```
{%
    \xintexprRestoreCatcodes
```

Maybe SafeCatcodes was without effect because the colon and the rest are from some earlier macro definition. Give a safe definition to active colon (even if in math mode with a math active colon..).
The \XINT_expr_defvar_getname closes the group opened here.

```
    \begingroup\lccode`~`: \lowercase{\let~}\empty
    \edef\XINT_defvar_tmpa{\XINT_defvar_tmpa}%
    \edef\XINT_defvar_tmpa{\xint_zapspaces_o\XINT_defvar_tmpa}%
    \expandafter\XINT_expr_defvar_getname
                \detokenize\expandafter{\XINT_defvar_tmpa}:~%
    \ifcase\XINT_defvar_tmpc\space
      \xintMessage {xintexpr}{Error}
      {Aborting: not allowed to declare variable with empty name.}%
    \or
     \XINT_global
     \expandafter\edef\csname XINT_expr_varvalue_\XINT_defvar_tmpa\endcsname
                 {\romannumeral0#1#2\relax}%
     \XINT_expr_defvar_one_b\XINT_defvar_tmpa
    \else
     \edef\XINT_defvar_tmpb{\romannumeral0#1#2\relax}%
     \edef\XINT_defvar_tmpd{\expandafter\xintLength\expandafter{\XINT_defvar_tmpb}}%
     \let\XINT_defvar_tmpe\empty
     \if1\XINT_defvar_tmpd
         \def\XINT_defvar_tmpe{unpacked }%
         \oodef\XINT_defvar_tmpb{\expandafter\xint_firstofone\XINT_defvar_tmpb}%
         \edef\XINT_defvar_tmpd{\expandafter\xintLength\expandafter{\XINT_defvar_tmpb}}%
     \fi
     \ifnum\XINT_defvar_tmpc=\XINT_defvar_tmpd\space
       \xintAssignArray\xintCSVtoList\XINT_defvar_tmpa\to\XINT_defvar_tmpvar
       \xintAssignArray\xintApply\XINT_embrace{\XINT_defvar_tmpb}\to\XINT_defvar_tmpval
       \def\XINT_defvar_tmpd{1}%
       \xintloop
           \expandafter\XINT_expr_defvar_one
           \csname XINT_defvar_tmpvar\XINT_defvar_tmpd\expandafter\endcsname
           \csname XINT_defvar_tmpval\XINT_defvar_tmpd\endcsname
       \ifnum\XINT_defvar_tmpd<\XINT_defvar_tmpc\space
           \edef\XINT_defvar_tmpd{\the\numexpr\XINT_defvar_tmpd+1}%
       \repeat
       \xintRelaxArray\XINT_defvar_tmpvar
       \xintRelaxArray\XINT_defvar_tmpval
     \else
       \xintMessage {xintexpr}{Error}
        {Aborting: mismatch between number of variables (\XINT_defvar_tmpc)
         and number of \XINT_defvar_tmpe values (\XINT_defvar_tmpd).}%
     \fi
    \fi
    \let\XINT_defvar_tmpa\empty
    \let\XINT_defvar_tmpb\empty
    \let\XINT_defvar_tmpc\empty
    \let\XINT_defvar_tmpd\empty
```

```
}%
\catcode`~ 3
\catcode`: 11
```

This SafeCatcodes is mainly in the hope that semi-colon ending the expression can still be sanitized.

```
\def\xintdefvar      {\xintexprSafeCatcodes\xintdefvar_a}%
\def\xintdefiivar    {\xintexprSafeCatcodes\xintdefiivar_a}%
\def\xintdeffloatvar {\xintexprSafeCatcodes\xintdeffloatvar_a}%
\def\xintdefvar_a      #1={\XINT_expr_defvar\xintthebareeval      {#1}}%
\def\xintdefiivar_a    #1={\XINT_expr_defvar\xintthebareiieval    {#1}}%
\def\xintdeffloatvar_a #1={\XINT_expr_defvar\xintthebarefloateval {#1}}%
```

11.23.2 \xintunassignvar

1.2e.

1.3d. Embarrassingly I had for a long time a misunderstanding of \ifcsname (let's blame its documentation) and I was not aware that it chooses FALSE branch if tested control sequence has been \let to \undefined... So earlier version didn't do the right thing (and had another bug: failure to protect \.=0 from expansion).
The \ifcsname tests are done in \XINT_expr_op__ and \XINT_expr_op_`.

```
\def\xintunassignvar #1{%
   \edef\XINT_unvar_tmpa{#1}%
   \edef\XINT_unvar_tmpa {\xint_zapspaces_o\XINT_unvar_tmpa}%
   \ifcsname XINT_expr_var_\XINT_unvar_tmpa\endcsname
       \ifnum\expandafter\xintLength\expandafter{\XINT_unvar_tmpa}=\@ne
         \expandafter\xintnewdummy\XINT_unvar_tmpa
       \else
         \XINT_global\expandafter
           \let\csname XINT_expr_varvalue_\XINT_unvar_tmpa\endcsname\xint_undefined
         \XINT_global\expandafter
           \let\csname XINT_expr_var_\XINT_unvar_tmpa\endcsname\xint_undefined
         \XINT_global\expandafter
           \let\csname XINT_expr_onliteral_\XINT_unvar_tmpa\endcsname\xint_undefined
         \ifxintverbose\xintMessage {xintexpr}{Info}
           {Variable \XINT_unvar_tmpa\space has been
            \ifxintglobaldefs globally \fi ``unassigned''.}%
         \fi
       \fi
   \else
       \xintMessage {xintexpr}{Warning}
           {Error: there was no such variable \XINT_unvar_tmpa\space to unassign.}%
   \fi
}%
```

11.24 Support for dummy variables

11.24.1 \xintnewdummy

Comments under reconstruction.
1.4 adds multi-letter names as usable dummy variables!

```
\catcode`* 11
\def\XINT_expr_makedummy #1%
{%
   \edef\XINT_tmpa{\xint_zapspaces #1 \xint_gobble_i}%
   \ifcsname XINT_expr_var_\XINT_tmpa\endcsname
     \XINT_global
     \expandafter\let\csname XINT_expr_var_\XINT_tmpa/old\expandafter\endcsname
                     \csname XINT_expr_var_\XINT_tmpa\expandafter\endcsname
   \fi
   \ifcsname XINT_expr_onliteral_\XINT_tmpa\endcsname
     \XINT_global
     \expandafter\let\csname XINT_expr_onliteral_\XINT_tmpa/old\expandafter\endcsname
                     \csname XINT_expr_onliteral_\XINT_tmpa\expandafter\endcsname
   \fi
   \expandafter\XINT_global
   \expanded
   {\edef\expandafter\noexpand
         \csname XINT_expr_var_\XINT_tmpa\endcsname ##1\relax !\XINT_tmpa##2}%
      {{##2}##1\relax !\XINT_tmpa{##2}}%
   \expandafter\XINT_global
   \expanded
   {\edef\expandafter\noexpand
         \csname XINT_expr_onliteral_\XINT_tmpa\endcsname ##1\relax !\XINT_tmpa##2}%
      {\XINT_expr_precedence_*** *{##2}(##1\relax !\XINT_tmpa{##2}}%)
}%
\xintApplyUnbraced \XINT_expr_makedummy {abcdefghijklmnopqrstuvwxyz}%
\xintApplyUnbraced \XINT_expr_makedummy {ABCDEFGHIJKLMNOPQRSTUVWXYZ}%
\def\xintnewdummy #1{%
    \XINT_expr_makedummy{#1}%
    \ifxintverbose\xintMessage {xintexpr}{Info}%
       {\XINT_tmpa\space now
        \ifxintglobaldefs globally \fi usable as dummy variable.}%
    \fi
}%
%    \begin{macrocode}
% Je ne définis pas de onliteral for them (it only serves for allowing
% tacit multiplication if variable name is in front of an opening
% parenthesis).
%
% The |nil| variable was need in |xint < 1.4| (with some other meaning)
% in places the syntax could not allow emptiness, such as |,,|, and
```

```
% other things, but at |1.4| meaning as changed.
%
% The other variables are new with |1.4|.
% Don't use the |None|, it is tentative, and may be input as |[]|.
%    \begin{macrocode}
\def\XINT_expr_var_nil{{}}%
\def\XINT_expr_var_None{{{}}}% ? tentative
\def\XINT_expr_var_false{{{0}}}% Maple, TeX
\def\XINT_expr_var_true{{{1}}}%
\def\XINT_expr_var_False{{{0}}}% Python
\def\XINT_expr_var_True{{{1}}}%
\catcode`* 12
```

11.24.2 \xintensuredummy, \xintrestorevariable

1.3e \xintensuredummy differs from \xintnewdummy only in the informational message... Attention that this is not meant to be nested.

1.4 fixes that the message mentioned non-existent \xintrestoredummy (real name was \xintrestorelettervar and renames the latter to \xintrestorevariable as it applies also to multi-letter names.

```
\def\xintensuredummy #1{%
    \XINT_expr_makedummy{#1}%
    \ifxintverbose\xintMessage {xintexpr}{Info}%
       {\XINT_tmpa\space now
        \ifxintglobaldefs globally \fi usable as dummy variable.&&J
        Issue \string\xintrestorevariable{\XINT_tmpa} to restore former meaning.}%
    \fi
}%
\def\xintrestorevariablesilently #1{%
   \edef\XINT_tmpa{\xint_zapspaces #1 \xint_gobble_i}%
   \ifcsname XINT_expr_var_\XINT_tmpa/old\endcsname
     \XINT_global
     \expandafter\let\csname XINT_expr_var_\XINT_tmpa\expandafter\endcsname
                     \csname XINT_expr_var_\XINT_tmpa/old\expandafter\endcsname
   \fi
   \ifcsname XINT_expr_onliteral_\XINT_tmpa/old\endcsname
     \XINT_global
     \expandafter\let\csname XINT_expr_onliteral_\XINT_tmpa\expandafter\endcsname
                     \csname XINT_expr_onliteral_\XINT_tmpa/old\expandafter\endcsname
   \fi
}%
\def\xintrestorevariable #1{%
    \xintrestorevariablesilently {#1}%
    \ifxintverbose\xintMessage {xintexpr}{Info}%
       {\XINT_tmpa\space
        \ifxintglobaldefs globally \fi restored to its earlier status, if any.}%
    \fi
}%
```

11.24.3 Checking (without expansion) that a symbolic expression contains correctly nested parentheses

Expands to \xint_c_mone in case a closing) had no opening (matching it, to \@ne if opening) had no closing) matching it, to \z@ if expression was balanced. Call it as:
\XINT_isbalanced_a \relax #1(\xint_bye)\xint_bye
This is legacy f-expandable code not using \expanded even at 1.4.

```
\def\XINT_isbalanced_a #1({\XINT_isbalanced_b #1)\xint_bye }%
\def\XINT_isbalanced_b #1)#2%
   {\xint_bye #2\XINT_isbalanced_c\xint_bye\XINT_isbalanced_error }%
```

if #2 is not \xint_bye, a) was found, but there was no (. Hence error -> -1

```
\def\XINT_isbalanced_error #1)\xint_bye {\xint_c_mone}%
```

#2 was \xint_bye, was there a) in original #1?

```
\def\XINT_isbalanced_c\xint_bye\XINT_isbalanced_error #1%
    {\xint_bye #1\XINT_isbalanced_yes\xint_bye\XINT_isbalanced_d #1}%
```

#1 is \xint_bye, there was never (nor) in original #1, hence OK.

```
\def\XINT_isbalanced_yes\xint_bye\XINT_isbalanced_d\xint_bye )\xint_bye {\xint_c_ }%
```

#1 is not \xint_bye, there was indeed a (in original #1. We check if we see a). If we do, we then loop until no (nor) is to be found.

```
\def\XINT_isbalanced_d #1)#2%
   {\xint_bye #2\XINT_isbalanced_no\xint_bye\XINT_isbalanced_a #1#2}%
```

#2 was \xint_bye, we did not find a closing) in original #1. Error.

```
\def\XINT_isbalanced_no\xint_bye #1\xint_bye\xint_bye {\xint_c_i }%
```

11.24.4 Fetching balanced expressions E1, E2 and a variable name Name from E1, Name=E2)

Multi-letter dummy variables added at 1.4.

```
\def\XINT_expr_fetch_E_comma_V_equal_E_a #1#2,%
{%
    \ifcase\XINT_isbalanced_a \relax #1#2(\xint_bye)\xint_bye
           \expandafter\XINT_expr_fetch_E_comma_V_equal_E_c
       \or\expandafter\XINT_expr_fetch_E_comma_V_equal_E_b
     \else\expandafter\xintError:noopening
    \fi {#1#2},%
}%
\def\XINT_expr_fetch_E_comma_V_equal_E_b #1,%
   {\XINT_expr_fetch_E_comma_V_equal_E_a {#1,}}%
\def\XINT_expr_fetch_E_comma_V_equal_E_c #1,#2#3=%
{%
    \expandafter\XINT_expr_fetch_E_comma_V_equal_E_d\expandafter
    {\expanded{{\xint_zapspaces #2#3 \xint_gobble_i}}{#1}}{}%
}%
\def\XINT_expr_fetch_E_comma_V_equal_E_d #1#2#3)%
{%
    \ifcase\XINT_isbalanced_a \relax #2#3(\xint_bye)\xint_bye
        \or\expandafter\XINT_expr_fetch_E_comma_V_equal_E_e
      \else\expandafter\xintError:noopening
```

```
2029     \fi
2030     {#1}{#2#3}%
2031 }%
2032 \def\XINT_expr_fetch_E_comma_V_equal_E_e #1#2{\XINT_expr_fetch_E_comma_V_equal_E_d {#1}{#2)}}%
```

11.24.5 Fetching a balanced expression delimited by a semi-colon

1.4. For subsn() leaner syntax of nested substitutions.
Will also serve to \xintdeffunc, to not have to hide inner semi-colons in for example an iter() from \xintdeffunc.
Adding brace removal protection for no serious reason, anyhow the xintexpr parsers always removes braces when moving forward, but well.
Trigger by \romannumeral\XINT_expr_fetch_to_semicolon upfront.

```
2033 \def\XINT_expr_fetch_to_semicolon {\XINT_expr_fetch_to_semicolon_a {}\empty}%
2034 \def\XINT_expr_fetch_to_semicolon_a #1#2;%
2035 {%
2036     \ifcase\XINT_isbalanced_a \relax #1#2(\xint_bye)\xint_bye
2037           \xint_dothis{\expandafter\XINT_expr_fetch_to_semicolon_c}%
2038        \or\xint_dothis{\expandafter\XINT_expr_fetch_to_semicolon_b}%
2039      \else\expandafter\xintError:noopening
2040     \fi\xint_orthat{}\expandafter{#2}{#1}%
2041 }%
2042 \def\XINT_expr_fetch_to_semicolon_b #1#2{\XINT_expr_fetch_to_semicolon_a {#2#1;}\empty}%
2043 \def\XINT_expr_fetch_to_semicolon_c #1#2{\xint_c_{#2#1}}%
```

11.24.6 Low-level support for omit and abort keywords, the break() function, the n++ construct and the semi-colon as used in the syntax of seq(), add(), mul(), iter(), rseq(), iterr(), rrseq(), subsm(), subsn(), ndseq(), ndmap()

There is some clever play simply based on setting suitable precedence levels combined with special meanings given to op macros.
The special !? internal operator is a helper for omit and abort keywords in list generators.
Prior to 1.4 support for +[, *[, ...,]+,]*, had some elements here.

The n++ construct 1.1 2014/10/29 did \expandafter\.=+\xintiCeil which transformed it into \romannumeral0\xinticeil, which seems a bit weird. This exploited the fact that dummy variables macros could back then pick braced material (which in the case at hand here ended being {\romannumeral0\xinticeil...} and were submitted to two expansions. The result of this was to provide a not value which got expanded only in the first loop of the :_A and following macros of seq, iter, rseq, etc...
Anyhow with 1.2c I have changed the implementation of dummy variables which now need to fetch a single locked token, which they do not expand.
The \xintiCeil appears a bit dispendious, but I need the starting value in a \numexpr compatible form in the iteration loops.

```
2044 \expandafter\def\csname XINT_expr_itself_++\endcsname {++}%
2045 \expandafter\def\csname XINT_expr_itself_++)\endcsname {++)}%
2046 \expandafter\let\csname XINT_expr_precedence_++)\endcsname \xint_c_i
2047 \xintFor #1 in {expr,flexpr,iiexpr} \do {%
2048     \expandafter\def\csname XINT_#1_op_++)\endcsname ##1##2\relax
2049      {\expandafter\XINT_expr_foundend
```

```
                    \expanded{{+{\XINT:NEhook:f:one:from:one:direct\xintiCeil##1}}}%
    }%
}%
```

The break() function break is a true function, the parsing via expansion of the enclosed material proceeds via _oparen macros as with any other function.

```
\catcode`? 3
\def\XINT_expr_func_break #1#2#3{#1#2{?#3}}%
\catcode`? 11
\let\XINT_flexpr_func_break \XINT_expr_func_break
\let\XINT_iiexpr_func_break \XINT_expr_func_break
```

The omit and abort keywords Comments are currently undergoing reconstruction.

```
\edef\XINT_expr_var_omit  #1\relax !{1\string !?!\relax !}%
\edef\XINT_expr_var_abort #1\relax !{1\string !?^\relax !}%
\def\XINT_expr_itself_!? {!?}%
\def\XINT_expr_op_!? #1#2\relax{\XINT_expr_foundend{#2}}%
\let\XINT_iiexpr_op_!? \XINT_expr_op_!?
\let\XINT_flexpr_op_!? \XINT_expr_op_!?
\let\XINT_expr_precedence_!? \xint_c_iv
```

The semi-colon Obsolete comments undergoing re-construction

```
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
    \expandafter\def\csname XINT_#1_op_;\endcsname {\xint_c_i ;}%
}%
\expandafter\let\csname XINT_expr_precedence_;\endcsname\xint_c_i
\expandafter\def\csname XINT_expr_itself_;)\endcsname {)}%
\expandafter\let\csname XINT_expr_precedence_;)\endcsname\xint_c_i
```

11.24.7 Reserved dummy variables @, @1, @2, @3, @4, @@, @@(1), . . . , @@@, @@@(1), . . . for recursions

Comments currently under reconstruction.
1.4 breaking change: @ and @1 behave differently and one can not use @ in place of @1 in iterr() and rrseq(). Formerly @ and @1 had the same definition.
Brace stripping in \XINT_expr_func_@@ is prevented by some ending 0 or other token see iterr() and rrseq() code.
For the record, the ~ and ? have catcode 3 in this code.

```
\catcode`* 11
\def\XINT_expr_var_@ #1~#2{{#2}#1~{#2}}%
\def\XINT_expr_onliteral_@ #1~#2{\XINT_expr_precedence_*** *{#2}(#1~{#2}}%
\expandafter
\def\csname XINT_expr_var_@1\endcsname #1~#2{{{#2}}#1~{#2}}%
\expandafter
\def\csname XINT_expr_var_@2\endcsname #1~#2#3{{{#3}}#1~{#2}{#3}}%
\expandafter
\def\csname XINT_expr_var_@3\endcsname #1~#2#3#4{{{#4}}#1~{#2}{#3}{#4}}%
\expandafter
```

```
\def\csname XINT_expr_var_@4\endcsname #1~#2#3#4#5{{{#5}}#1~{#2}{#3}{#4}{#5}}%
\expandafter\def\csname XINT_expr_onliteral_@1\endcsname #1~#2%
                {\XINT_expr_precedence_*** *{{#2}}(#1~{#2}}%
\expandafter\def\csname XINT_expr_onliteral_@2\endcsname #1~#2#3%
                {\XINT_expr_precedence_*** *{{#3}}(#1~{#2}{#3}}%
\expandafter\def\csname XINT_expr_onliteral_@3\endcsname #1~#2#3#4%
                {\XINT_expr_precedence_*** *{{#4}}(#1~{#2}{#3}{#4}}%
\expandafter\def\csname XINT_expr_onliteral_@4\endcsname #1~#2#3#4#5%
                {\XINT_expr_precedence_*** *{{#5}}(#1~{#2}{#3}{#4}{#5}}%
\catcode`* 12
\catcode`? 3
\def\XINT_expr_func_@@ #1#2#3#4~#5?%
{%
   \expandafter#1\expandafter#2\expandafter{\expandafter{%
         \romannumeral0\xintntheltnoexpand{\xintNum#3}{#5}}}#4~#5?%
}%
\def\XINT_expr_func_@@@ #1#2#3#4~#5~#6?%
{%
   \expandafter#1\expandafter#2\expandafter{\expandafter{%
         \romannumeral0\xintntheltnoexpand{\xintNum#3}{#6}}}#4~#5~#6?%
}%
\def\XINT_expr_func_@@@@ #1#2#3#4~#5~#6~#7?%
{%
   \expandafter#1\expandafter#2\expandafter{\expandafter{%
         \romannumeral0\xintntheltnoexpand{\xintNum#3}{#7}}}#4~#5~#6~#7?%
}%
\let\XINT_flexpr_func_@@\XINT_expr_func_@@
\let\XINT_flexpr_func_@@@\XINT_expr_func_@@@
\let\XINT_flexpr_func_@@@@\XINT_expr_func_@@@@
\def\XINT_iiexpr_func_@@ #1#2#3#4~#5?%
{%
   \expandafter#1\expandafter#2\expandafter{\expandafter{%
         \romannumeral0\xintntheltnoexpand{\xint_firstofone#3}{#5}}}#4~#5?%
}%
\def\XINT_iiexpr_func_@@@ #1#2#3#4~#5~#6?%
{%
   \expandafter#1\expandafter#2\expandafter{\expandafter{%
         \romannumeral0\xintntheltnoexpand{\xint_firstofone#3}{#6}}}#4~#5~#6?%
}%
\def\XINT_iiexpr_func_@@@@ #1#2#3#4~#5~#6~#7?%
{%
   \expandafter#1\expandafter#2\expandafter{\expandafter{%
         \romannumeral0\xintntheltnoexpand{\xint_firstofone#3}{#7}}}#4~#5~#6~#7?%
}%
\catcode`? 11
```

11.25 Pseudo-functions involving dummy variables and generating scalars or sequences

11.25.1 Comments

Comments added 2020/01/16.

The mechanism for «seq» is the following. When the parser encounters «seq», which means it parsed these letters and encountered (from expansion) an opening parenthesis, the \XINT_expr_func mechanism triggers the «`» operator which realizes that «seq» is a pseudo-function (there is no _func_seq) and thus spans the \XINT_expr_onliteral_seq macro (currently this means however that the knowledge of which parser we are in is lost, see comments of \XINT_expr_op_` code). The latter will use delimited macros and parenthesis check to fetch (without any expansion), the symbolic expression ExprSeq to evaluate, the Name (now possibly multi-letter) of the variable and the expression ExprValues to evaluate which will give the values to assign to the dummy variable Name. It then positions upstream ExprValues suitably terminated (see next) and after it {{Name}{ExprSeq}}. Then it inserts a second call to the «`» operator with now «seqx» as argument hence the appropriate «{,fl,ii}expr_func_seqx» macros gets executed. The general way function macros work is that first all their arguments are evaluated via a call not to \xintbare{,float,ii}eval but to the suitable \XINT_{expr,flexpr,iiexpr}_oparen core macro which does almost same excepts it expects a final closing parenthesis (of course allowing nested parenthesis in-between) and stops there. Here, this closing parenthesis got positioned deliberately with a \relax after it, so the parser, which always after having gathered a value looks ahead to find the next operator, thinks it has hit the end of the expression and as result inserts a \xint_c_ (i.e. \z@) token for precedence level and a dummy \relax token (place-holder for a non-existing operator). Generally speaking «func_foo» macros expect to be executed with three parameters #1#2#3, #1 = precedence, #2 = operator, #3 = values (call it «args») i.e. the fully evaluated list of all its arguments. The special «func_seqx» and cousins know that the first two tokens are trash and they now proceed forward, having thus lying before them upstream the values to loop over, now fully evaluated, and {{Name}{ExprSeq}}. It then positions appropriately ExprSeq inside a sub-expression and after it, following suitable delimiter, Name and the evaluated values to assign to Name.

Dummy variables are essentially simply delimited macros where the delimiter is the variable name preceded by a \relax token and a catcode 11 exclamation point. Thus the various «subsx», «seqx», «iterx» position the tokens appropriately and launch suitable loops.

All of this nests well, inner «seq»'s (or more often in practice «subs»'s) being allowed to refer to the dummy variables used by outer «seq»'s because the outer «seq»'s have the values to assign to their variables evaluated first and their ExprSeq evaluated last. For inner dummy variables to be able to refer to outer dummy variables the author must be careful of course to not use in the implementation braces { and } which would break dummy variables to fetch values beyond the closing brace.

The above «seq» mechanism was done around June 15-25th 2014 at the time of the transition from 1.09n to 1.1 but already in October 2014 I made a note that I had a hard time to understand it again:

« [START OF YEAR 2014 COMMENTS]

All of seq, add, mul, rseq, etc... (actually all of the extensive changes from xintexpr 1.09n to 1.1) was done around June 15-25th 2014, but the problem is that I did not document the code enough, and I had a hard time understanding in October what I had done in June. Despite the lesson, again being short on time, I do not document enough my current understanding of the innards of the beast...

I added subs, and iter in October (also the [:n], [n:] list extractors), proving I did at least understand a bit (or rather could imitate) my earlier code (but don't ask me to explain \xintNewExpr !)

The \XINT_expr_fetch_E_comma_V_equal_E_a parses: "expression, variable=list)" (when it is called the opening (has been swallowed, and it looks for the ending one.) Both expression and list may themselves contain parentheses and commas, we allow nesting. For example "x^2,x=1..10)", at the end of seq_a we have {variable{expression}}{list}, in this example {x{x^2}}{1..10}, or more complicated "seq(add(y,y=1..x),x=1..10)" will work too. The variable is a single lowercase Latin letter.

The complications with \xint_c_ii^v in seq_f is for the recurrent thing that we don't know in what type of expressions we are, hence we must move back up, with some loss of efficiency (superfluous check for minus sign, etc...). But the code manages simultaneously expr, flexpr and iiexpr.

[END OF YEAR 2014 OLD COMMENTS]»

On Jeudi 16 janvier 2020 à 15:13:32 I finally did the documentation as above.

The case of «iter», «rseq», «iterr», «rrseq» differs slightly because the initial values need evaluation. This is done by genuine functions \XINT_<parser>_func_iter etc... (there was no \XINT_<parser>_func_seq). The trick is via the semi-colon ; which is a genuine operator having the precedence of a closing parenthesis and whose action is only to stop expansion. Thus this first step of gathering the initial values is done as part of the reguler expansion job of the parser not using delimited macros and the ; can be hidden in braces {;} because the three parsers when moving forward remove one level of braces always. Thus \XINT_<parser>_func_seq simply hand over to \XINT_allexpr_iter which will then trigger the fetching without expansion of ExprIter, Name=ExprValues as described previously for «seq».

With 1.4, multi-letter names for dummy variables are allowed.

Also there is the additional 1.4 ambition to make the whole thing parsable by \xintNewExpr/\xintdeffunc. This is done by checking if all is numerical, because the omit, abort and break() mechanisms have no translation into macros, and the only solution for symbolic material is to simply keep it as is, so that expansion will again activate the xintexpr parsers. At 1.4 this approach is fine although the initial goals of \xintNewExpr/\xintdeffunc was to completely replace the parsers (whose storage method hit the string pool formerly) by macros. Now that 1.4 does not impact the string pool we can make \xintdeffunc much more powerful but it will not be a construct using only xintfrac macros, it will still be partially the \xintexpr etc... parsers in such cases.

Got simpler with 1.2c as now the dummy variable fetches an already encapsulated value, which is anyhow the form in which we get it.

Refactored at 1.4 using \expanded rather than \csname.

And support for multi-letter variables, which means function declarations can now use multi-letter variables !

11.25.2 subs(): substitution of one variable

```
\def\XINT_expr_onliteral_subs
{%
    \expandafter\XINT_allexpr_subs_f
    \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
}%
\def\XINT_allexpr_subs_f #1#2{\xint_c_ii^v `{subsx}#2)\relax #1}%
\def\XINT_expr_func_subsx   #1#2{\XINT_allexpr_subsx \xintbareeval }%
\def\XINT_flexpr_func_subsx #1#2{\XINT_allexpr_subsx \xintbarefloateval}%
\def\XINT_iiexpr_func_subsx #1#2{\XINT_allexpr_subsx \xintbareiieval }%
```

#2 is the value to assign to the dummy variable #3 is the dummy variable name (possibly multi-letter), #4 is the expression to evaluate

```
\def\XINT_allexpr_subsx #1#2#3#4%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded
    \bgroup\romannumeral0#1#4\relax \iffalse\relax !#3{#2}{\fi
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

11.25.3 subsm(): simultaneous independent substitutions

New with 1.4. Globally the var1=expr1; var2=expr2; var2=expr3;... part can arise from expansion, except that once a semi-colon has been found (from expansion) the varK= thing following it must be there. And as for subs() the final parenthesis must be there from the start.

```
\def\XINT_expr_onliteral_subsm
{%
    \expandafter\XINT_allexpr_subsm_f
    \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
}%
\def\XINT_allexpr_subsm_f #1#2{\xint_c_ii^v `{subsmx}#2)\relax #1}%
\def\XINT_expr_func_subsmx
{%
    \expandafter\XINT_allexpr_subsmx\expandafter\xintbareeval
    \expanded\bgroup{\iffalse}\fi\XINT_allexpr_subsm_A\XINT_expr_oparen
}%
\def\XINT_flexpr_func_subsmx
{%
    \expandafter\XINT_allexpr_subsmx\expandafter\xintbarefloateval
    \expanded\bgroup{\iffalse}\fi\XINT_allexpr_subsm_A\XINT_flexpr_oparen
}%
\def\XINT_iiexpr_func_subsmx
{%
    \expandafter\XINT_allexpr_subsmx\expandafter\xintbareiieval
    \expanded\bgroup{\iffalse}\fi\XINT_allexpr_subsm_A\XINT_iiexpr_oparen
}%
\def\XINT_allexpr_subsm_A #1#2#3%
{%
    \ifx#2\xint_c_
       \expandafter\XINT_allexpr_subsm_done
    \else
       \expandafter\XINT_allexpr_subsm_B
    \fi #1%
}%
\def\XINT_allexpr_subsm_B #1#2#3#4=%
{%
    {#2}\relax !\xint_zapspaces#3#4 \xint_gobble_i
    \expandafter\XINT_allexpr_subsm_A\expandafter#1\romannumeral`&&@#1%
}%
```

#1 = \xintbareeval, or \xintbarefloateval or \xintbareiieval #2 = evaluation of last variable assignment

```
\def\XINT_allexpr_subsm_done #1#2{{#2}\iffalse{{\fi}}}%
```

```
   #1 = \xintbareeval or \xintbarefloateval or \xintbareiieval  #2 = {value1}\relax !var2{value2}....\relax
 !varN{valueN} (value's may be oples)  #3 = {var1}  #4 = the expression to evaluate
```

```
\def\XINT_allexpr_subsmx #1#2#3#4%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded
    \bgroup\romannumeral0#1#4\relax \iffalse\relax !#3#2{\fi
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

11.25.4 subsn(): leaner syntax for nesting (possibly dependent) substitutions

New with 1.4. 2020/01/24

```
\def\XINT_expr_onliteral_subsn
{%
    \expandafter\XINT_allexpr_subsn_f
    \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
}%
\def\XINT_allexpr_subsn_f #1{\XINT_allexpr_subsn_g #1}%
```

```
    #1 = Name1
    #2 = Expression in all variables which is to evaluate
    #3 = all the stuff after Name1 =  and up to final parenthesis
```

```
\def\XINT_allexpr_subsn_g #1#2#3%
{%
    \expandafter\XINT_allexpr_subsn_h
    \expanded\bgroup{\iffalse}\fi\expandafter\XINT_allexpr_subsn_B
    \romannumeral\XINT_expr_fetch_to_semicolon #1=#3;\hbox=;;^{#2}%
}%
\def\XINT_allexpr_subsn_B #1{\XINT_allexpr_subsn_C #1\vbox}%
\def\XINT_allexpr_subsn_C #1#2=#3\vbox
{%
    \ifx\hbox#1\iffalse{{\fi}\expandafter}\else
    {{\xint_zapspaces #1#2 \xint_gobble_i}};\unexpanded{{{#3}}}%
    \expandafter\XINT_allexpr_subsn_B
    \romannumeral\expandafter\XINT_expr_fetch_to_semicolon\fi
}%
\def\XINT_allexpr_subsn_h
{%
    \xint_c_ii^v `{subsnx}\romannumeral0\xintreverseorder
}%
\def\XINT_expr_func_subsnx #1#2#3#4#5;#6%
{%
    \xint_gob_til_^ #6\XINT_allexpr_subsnx_H ^%
    \expandafter\XINT_allexpr_subsnx\expandafter
    \xintbareeval\romannumeral0\xintbareeval #5\relax !#4{#3}\xintundefined
    {\relax !#4{#3}\relax !#6}%
}%
\def\XINT_iiexpr_func_subsnx #1#2#3#4#5;#6%
```

```
{%
    \xint_gob_til_^ #6\XINT_allexpr_subsnx_H ^%
    \expandafter\XINT_allexpr_subsnx\expandafter
    \xintbareiieval\romannumeral0\xintbareiieval #5\relax !#4{#3}\xintundefined
    {\relax !#4{#3}\relax !#6}%
}%
\def\XINT_flexpr_func_subsnx #1#2#3#4#5;#6%
{%
    \xint_gob_til_^ #6\XINT_allexpr_subsnx_H ^%
    \expandafter\XINT_allexpr_subsnx\expandafter
    \xintbarefloateval\romannumeral0\xintbarefloateval #5\relax !#4{#3}\xintundefined
    {\relax !#4{#3}\relax !#6}%
}%
\def\XINT_allexpr_subsnx #1#2!#3\xintundefined#4#5;#6%
{%
    \xint_gob_til_^ #6\XINT_allexpr_subsnx_I ^%
    \expandafter\XINT_allexpr_subsnx\expandafter
    #1\romannumeral0#1#5\relax !#4{#2}\xintundefined
    {\relax !#4{#2}\relax !#6}%
}%
\def\XINT_allexpr_subsnx_H ^#1\romannumeral0#2#3!#4\xintundefined #5#6%
{%
    \expandafter\XINT_allexpr_subsnx_J\romannumeral0#2#6#5%
}%
\def\XINT_allexpr_subsnx_I ^#1\romannumeral0#2#3\xintundefined #4#5%
{%
    \expandafter\XINT_allexpr_subsnx_J\romannumeral0#2#5#4%
}%
\def\XINT_allexpr_subsnx_J #1#2^%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded{\unexpanded{{#1}}\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
```

11.25.5 seq(): sequences from assigning values to a dummy variable

In seq_f, the #2 is the ExprValues expression which needs evaluation to provide the values to the dummy variable and #1 is {Name}{ExprSeq} where Name is the name of dummy variable and {ExprSeq} the expression which will have to be evaluated.

```
\def\XINT_allexpr_seq_f #1#2{\xint_c_ii^v `{seqx}#2)\relax #1}%
\def\XINT_expr_onliteral_seq
 {\expandafter\XINT_allexpr_seq_f\romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}}%
\def\XINT_expr_func_seqx   #1#2{\XINT:NEhook:seqx\XINT_allexpr_seqx\xintbareeval }%
\def\XINT_flexpr_func_seqx #1#2{\XINT:NEhook:seqx\XINT_allexpr_seqx\xintbarefloateval}%
\def\XINT_iiexpr_func_seqx #1#2{\XINT:NEhook:seqx\XINT_allexpr_seqx\xintbareiieval }%
\def\XINT_allexpr_seqx #1#2#3#4%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded \bgroup {\iffalse}\fi\XINT_expr_seq:_b {#1#4\relax !#3}#2^%
    \XINT_expr_cb_and_getop
}%
\def\XINT_expr_cb_and_getop{\iffalse{\fi\expandafter}\romannumeral`&&@\XINT_expr_getop}%
```

Comments undergoing reconstruction.

```
\catcode`? 3
\def\XINT_expr_seq:_b #1#2%
{%
    \ifx +#2\xint_dothis\XINT_expr_seq:_Ca\fi
    \ifx !#2!\xint_dothis\XINT_expr_seq:_noop\fi
    \ifx ^#2\xint_dothis\XINT_expr_seq:_end\fi
    \xint_orthat{\XINT_expr_seq:_c}{#2}{#1}%
}%
\def\XINT_expr_seq:_noop #1{\XINT_expr_seq:_b }%
\def\XINT_expr_seq:_end  #1#2{\iffalse{\fi}}%
\def\XINT_expr_seq:_c #1#2{\expandafter\XINT_expr_seq:_d\romannumeral0#2{{#1}}{#2}}%
\def\XINT_expr_seq:_d #1{\ifx ^#1\xint_dothis\XINT_expr_seq:_abort\fi
                         \ifx ?#1\xint_dothis\XINT_expr_seq:_break\fi
                         \ifx !#1\xint_dothis\XINT_expr_seq:_omit\fi
                         \xint_orthat{\XINT_expr_seq:_goon {#1}}}%
\def\XINT_expr_seq:_abort #1!#2^{\iffalse{\fi}}%
\def\XINT_expr_seq:_break #1!#2^{#1\iffalse{\fi}}%
\def\XINT_expr_seq:_omit  #1!#2#{\expandafter\XINT_expr_seq:_b\xint_gobble_i}%
\def\XINT_expr_seq:_goon  #1!#2#{#1\expandafter\XINT_expr_seq:_b\xint_gobble_i}%
\def\XINT_expr_seq:_Ca #1#2#3{\XINT_expr_seq:_Cc#3.{#2}}%
\def\XINT_expr_seq:_Cb #1{\expandafter\XINT_expr_seq:_Cc\the\numexpr#1+\xint_c_i.}%
\def\XINT_expr_seq:_Cc #1.#2{\expandafter\XINT_expr_seq:_D\romannumeral0#2{{#1}}{#1}{#2}}%
\def\XINT_expr_seq:_D #1{\ifx ^#1\xint_dothis\XINT_expr_seq:_abort\fi
                         \ifx ?#1\xint_dothis\XINT_expr_seq:_break\fi
                         \ifx !#1\xint_dothis\XINT_expr_seq:_Omit\fi
                         \xint_orthat{\XINT_expr_seq:_Goon {#1}}}%
\def\XINT_expr_seq:_Omit #1!#2#{\expandafter\XINT_expr_seq:_Cb\xint_gobble_i}%
\def\XINT_expr_seq:_Goon #1!#2#{#1\expandafter\XINT_expr_seq:_Cb\xint_gobble_i}%
```

11.25.6 iter()

Prior to 1.2g, the iter keyword was what is now called iterr, analogous with rrseq. Somehow I forgot an iter functioning like rseq with the sole difference of printing only the last iteration. Both rseq and iter work well with list selectors, as @ refers to the whole comma separated sequence of the initial values. I have thus deliberately done the backwards incompatible renaming of iter to iterr, and the new iter.

To understand the tokens which are presented to \XINT_allexpr_iter it is needed to check elsewhere in the source code how the ; hack is done.

The #2 in \XINT_allexpr_iter is \xint_c_i from the ; hack. Formerly (xint < 1.4) there was no such token. The change is motivated to using ; also in subsm() syntax.

```
\def\XINT_expr_func_iter   {\XINT_allexpr_iter \xintbareeval      }%
\def\XINT_flexpr_func_iter {\XINT_allexpr_iter \xintbarefloateval }%
\def\XINT_iiexpr_func_iter {\XINT_allexpr_iter \xintbareiieval    }%
\def\XINT_allexpr_iter #1#2#3#4%
{%
    \expandafter\XINT_expr_iterx
    \expandafter#1\expanded{\unexpanded{{#4}}\expandafter}%
    \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
}%
\def\XINT_expr_iterx #1#2#3#4%
```

```
{%
    \XINT:NEhook:iter\XINT_expr_itery\romannumeral0#1(#4)\relax {#2}#3#1%
}%
\def\XINT_expr_itery #1#2#3#4#5%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded \bgroup {\iffalse}\fi
    \XINT_expr_iter:_b {#5#4\relax !#3}#1^~{#2}\XINT_expr_cb_and_getop
}%
\def\XINT_expr_iter:_b #1#2%
{%
    \ifx +#2\xint_dothis\XINT_expr_iter:_Ca\fi
    \ifx !#2!\xint_dothis\XINT_expr_iter:_noop\fi
    \ifx ^#2\xint_dothis\XINT_expr_iter:_end\fi
    \xint_orthat{\XINT_expr_iter:_c}{#2}{#1}%
}%
\def\XINT_expr_iter:_noop #1{\XINT_expr_iter:_b }%
\def\XINT_expr_iter:_end  #1#2~#3{#3\iffalse{\fi}}%
\def\XINT_expr_iter:_c #1#2{\expandafter\XINT_expr_iter:_d\romannumeral0#2{{#1}}{#2}}%
\def\XINT_expr_iter:_d #1{\ifx ^#1\xint_dothis\XINT_expr_iter:_abort\fi
                          \ifx ?#1\xint_dothis\XINT_expr_iter:_break\fi
                          \ifx !#1\xint_dothis\XINT_expr_iter:_omit\fi
                          \xint_orthat{\XINT_expr_iter:_goon {#1}}}%
\def\XINT_expr_iter:_abort #1!#2^~#3{#3\iffalse{\fi}}%
\def\XINT_expr_iter:_break #1!#2^~#3{#1\iffalse{\fi}}%
\def\XINT_expr_iter:_omit  #1!#2#{\expandafter\XINT_expr_iter:_b\xint_gobble_i}%
\def\XINT_expr_iter:_goon  #1!#2#{\XINT_expr_iter:_goon_a {#1}}%
\def\XINT_expr_iter:_goon_a #1#2#3~#4{\XINT_expr_iter:_b #3~{#1}}%
\def\XINT_expr_iter:_Ca #1#2#3{\XINT_expr_iter:_Cc#3.{#2}}%
\def\XINT_expr_iter:_Cb #1{\expandafter\XINT_expr_iter:_Cc\the\numexpr#1+\xint_c_i.}%
\def\XINT_expr_iter:_Cc #1.#2{\expandafter\XINT_expr_iter:_D\romannumeral0#2{{#1}}{#1}{#2}}%
\def\XINT_expr_iter:_D #1{\ifx ^#1\xint_dothis\XINT_expr_iter:_abort\fi
                          \ifx ?#1\xint_dothis\XINT_expr_iter:_break\fi
                          \ifx !#1\xint_dothis\XINT_expr_iter:_Omit\fi
                          \xint_orthat{\XINT_expr_iter:_Goon {#1}}}%
\def\XINT_expr_iter:_Omit  #1!#2#{\expandafter\XINT_expr_iter:_Cb\xint_gobble_i}%
\def\XINT_expr_iter:_Goon  #1!#2#{\XINT_expr_iter:_Goon_a {#1}}%
\def\XINT_expr_iter:_Goon_a #1#2#3~#4{\XINT_expr_iter:_Cb #3~{#1}}%
```

11.25.7 add(), mul()

Comments under reconstruction.

These were a bit anomalous as they did not implement omit and abort keyword and the break() function (and per force then neither the n++ syntax).

At 1.4 they are simply mapped to using adequately iter(). Thus, there is small loss in efficiency, but supporting omit, abort and break is important. Using dedicated macros here would have caused also slight efficiency drop. Simpler to remove the old approach.

```
\def\XINT_expr_onliteral_add
 {\expandafter\XINT_allexpr_add_f\romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}}%
\def\XINT_allexpr_add_f #1#2{\xint_c_ii^v `{opx}#2)\relax #1{+}{0}}%
\def\XINT_expr_onliteral_mul
 {\expandafter\XINT_allexpr_mul_f\romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}}%
```

```
\def\XINT_allexpr_mul_f #1#2{\xint_c_ii^v `{opx}#2)\relax #1{*}{1}}%
\def\XINT_expr_func_opx    {\XINT:NEhook:opx \XINT_allexpr_opx \xintbareeval     }%
\def\XINT_flexpr_func_opx {\XINT:NEhook:opx \XINT_allexpr_opx \xintbarefloateval}%
\def\XINT_iiexpr_func_opx {\XINT:NEhook:opx \XINT_allexpr_opx \xintbareiieval   }%
```

1.4a In case of usage of omit (did I not test it? obviously I didn't as neither omit nor abort could work; and break neither), 1.4 code using (#6) syntax caused a (somewhat misleading) «missing)» error message which originated in the #6. This is non-obvious problem (perhaps explained why prior to 1.4 I had not added support for omit and break() to add() and mul()...

Allowing () is not enough as it would have to be 0 or 1 depending on whether we are using add() or mul(). Hence the somewhat complicated detour (relying on precise way var_omit and var_abort work) via \XINT_allexpr_opx_ifnotomitted.

\break() has special meaning here as it is used as last operand, not as last value. The code is very unsatisfactory and inefficient but this is hotfix for 1.4a.

```
\def\XINT_allexpr_opx #1#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded \bgroup {\iffalse}\fi
    \XINT_expr_iter:_b {#1%
    \expandafter\XINT_allexpr_opx_ifnotomitted
    \romannumeral0#1#6\relax#7@\relax !#5}#4^~{{#8}}\XINT_expr_cb_and_getop
}%
\def\XINT_allexpr_opx_ifnotomitted #1%
{%
    \ifx !#1\xint_dothis{@\relax}\fi
    \ifx ^#1\xint_dothis{\XINTfstop. ^\relax}\fi
    \if ?\xintFirstItem{#1}\xint_dothis{\XINT_allexpr_opx_break{#1}}\fi
    \xint_orthat{\XINTfstop.{#1}}%
}%
\def\XINT_allexpr_opx_break #1#2\relax
{%
    break(\expandafter\XINTfstop\expandafter.\expandafter{\xint_gobble_i#1}#2)\relax
}%
```

11.25.8 rseq()

When func_rseq has its turn, initial segment has been scanned by oparen, the ; mimicking the rôle of a closing parenthesis, and stopping further expansion (and leaving a \xint_c_i left-over token since 1.4). The ; is discovered during standard parsing mode, it may be for example {;} or arise from expansion as rseq does not use a delimited macro to locate it.

```
\def\XINT_expr_func_rseq    {\XINT_allexpr_rseq \xintbareeval      }%
\def\XINT_flexpr_func_rseq {\XINT_allexpr_rseq \xintbarefloateval }%
\def\XINT_iiexpr_func_rseq {\XINT_allexpr_rseq \xintbareiieval    }%
\def\XINT_allexpr_rseq #1#2#3#4%
{%
    \expandafter\XINT_expr_rseqx
    \expandafter #1\expanded{\unexpanded{{#4}}\expandafter}%
    \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
}%
\def\XINT_expr_rseqx #1#2#3#4%
{%
```

```
    \XINT:NEhook:rseq \XINT_expr_rseqy\romannumeral0#1(#4)\relax {#2}#3#1%
}%
\def\XINT_expr_rseqy #1#2#3#4#5%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded \bgroup {\iffalse}\fi
    #2%
    \XINT_expr_rseq:_b {#5#4\relax !#3}#1^~{#2}\XINT_expr_cb_and_getop
}%
\def\XINT_expr_rseq:_b #1#2%
{%
    \ifx +#2\xint_dothis\XINT_expr_rseq:_Ca\fi
    \ifx !#2!\xint_dothis\XINT_expr_rseq:_noop\fi
    \ifx ^#2\xint_dothis\XINT_expr_rseq:_end\fi
    \xint_orthat{\XINT_expr_rseq:_c}{#2}{#1}%
}%
\def\XINT_expr_rseq:_noop #1{\XINT_expr_rseq:_b }%
\def\XINT_expr_rseq:_end  #1#2~#3{\iffalse{\fi}}%
\def\XINT_expr_rseq:_c #1#2{\expandafter\XINT_expr_rseq:_d\romannumeral0#2{{#1}}{#2}}%
\def\XINT_expr_rseq:_d #1{\ifx ^#1\xint_dothis\XINT_expr_rseq:_abort\fi
                          \ifx ?#1\xint_dothis\XINT_expr_rseq:_break\fi
                          \ifx !#1\xint_dothis\XINT_expr_rseq:_omit\fi
                          \xint_orthat{\XINT_expr_rseq:_goon {#1}}}%
\def\XINT_expr_rseq:_abort #1!#2^~#3{\iffalse{\fi}}%
\def\XINT_expr_rseq:_break #1!#2^~#3{#1\iffalse{\fi}}%
\def\XINT_expr_rseq:_omit  #1!#2#{\expandafter\XINT_expr_rseq:_b\xint_gobble_i}%
\def\XINT_expr_rseq:_goon  #1!#2#{\XINT_expr_rseq:_goon_a {#1}}%
\def\XINT_expr_rseq:_goon_a #1#2#3~#4{#1\XINT_expr_rseq:_b #3~{#1}}%
\def\XINT_expr_rseq:_Ca #1#2#3{\XINT_expr_rseq:_Cc#3.{#2}}%
\def\XINT_expr_rseq:_Cb #1{\expandafter\XINT_expr_rseq:_Cc\the\numexpr#1+\xint_c_i.}%
\def\XINT_expr_rseq:_Cc #1.#2{\expandafter\XINT_expr_rseq:_D\romannumeral0#2{{#1}}{#1}{#2}}%
\def\XINT_expr_rseq:_D #1{\ifx ^#1\xint_dothis\XINT_expr_rseq:_abort\fi
                          \ifx ?#1\xint_dothis\XINT_expr_rseq:_break\fi
                          \ifx !#1\xint_dothis\XINT_expr_rseq:_Omit\fi
                          \xint_orthat{\XINT_expr_rseq:_Goon {#1}}}%
\def\XINT_expr_rseq:_Omit  #1!#2#{\expandafter\XINT_expr_rseq:_Cb\xint_gobble_i}%
\def\XINT_expr_rseq:_Goon  #1!#2#{\XINT_expr_rseq:_Goon_a {#1}}%
\def\XINT_expr_rseq:_Goon_a #1#2#3~#4{#1\XINT_expr_rseq:_Cb #3~{#1}}%
```

11.25.9 iterr()

ATTENTION! at 1.4 the @ and @1 are not synonymous anymore. One *must* use @1 in iterr() context.

```
\def\XINT_expr_func_iterr   {\XINT_allexpr_iterr \xintbareeval      }%
\def\XINT_flexpr_func_iterr {\XINT_allexpr_iterr \xintbarefloateval }%
\def\XINT_iiexpr_func_iterr {\XINT_allexpr_iterr \xintbareiieval    }%
\def\XINT_allexpr_iterr #1#2#3#4%
{%
    \expandafter\XINT_expr_iterrx
    \expandafter #1\expanded{{\xintRevWithBraces{#4}}\expandafter}%
    \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
}%
\def\XINT_expr_iterrx #1#2#3#4%
```

```
{%
    \XINT:NEhook:iterr\XINT_expr_iterry\romannumeral0#1(#4)\relax {#2}#3#1%
}%
\def\XINT_expr_iterry #1#2#3#4#5%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded \bgroup {\iffalse}\fi
    \XINT_expr_iterr:_b {#5#4\relax !#3}#1^~#20?\XINT_expr_cb_and_getop
}%
\def\XINT_expr_iterr:_b #1#2%
{%
    \ifx +#2\xint_dothis\XINT_expr_iterr:_Ca\fi
    \ifx !#2!\xint_dothis\XINT_expr_iterr:_noop\fi
    \ifx ^#2\xint_dothis\XINT_expr_iterr:_end\fi
    \xint_orthat{\XINT_expr_iterr:_c}{#2}{#1}%
}%
\def\XINT_expr_iterr:_noop #1{\XINT_expr_iterr:_b }%
\def\XINT_expr_iterr:_end  #1#2~#3#4?{{#3}\iffalse{\fi}}%
\def\XINT_expr_iterr:_c #1#2{\expandafter\XINT_expr_iterr:_d\romannumeral0#2{{#1}}{#2}}%
\def\XINT_expr_iterr:_d #1{\ifx ^#1\xint_dothis\XINT_expr_iterr:_abort\fi
                           \ifx ?#1\xint_dothis\XINT_expr_iterr:_break\fi
                           \ifx !#1\xint_dothis\XINT_expr_iterr:_omit\fi
                           \xint_orthat{\XINT_expr_iterr:_goon {#1}}}%
\def\XINT_expr_iterr:_abort #1!#2^~#3?{\iffalse{\fi}}%
\def\XINT_expr_iterr:_break #1!#2^~#3?{#1\iffalse{\fi}}%
\def\XINT_expr_iterr:_omit  #1!#2#{\expandafter\XINT_expr_iterr:_b\xint_gobble_i}%
\def\XINT_expr_iterr:_goon  #1!#2#{\XINT_expr_iterr:_goon_a{#1}}%
\def\XINT_expr_iterr:_goon_a  #1#2#3~#4?%
{%
    \expandafter\XINT_expr_iterr:_b \expanded{\unexpanded{#3~}\xintTrim{-2}{#1#4}}0?%
}%
\def\XINT_expr_iterr:_Ca #1#2#3{\XINT_expr_iterr:_Cc#3.{#2}}%
\def\XINT_expr_iterr:_Cb #1{\expandafter\XINT_expr_iterr:_Cc\the\numexpr#1+\xint_c_i.}%
\def\XINT_expr_iterr:_Cc #1.#2{\expandafter\XINT_expr_iterr:_D\romannumeral0#2{{#1}}{#1}{#2}}%
\def\XINT_expr_iterr:_D #1{\ifx ^#1\xint_dothis\XINT_expr_iterr:_abort\fi
                           \ifx ?#1\xint_dothis\XINT_expr_iterr:_break\fi
                           \ifx !#1\xint_dothis\XINT_expr_iterr:_Omit\fi
                           \xint_orthat{\XINT_expr_iterr:_Goon {#1}}}%
\def\XINT_expr_iterr:_Omit #1!#2#{\expandafter\XINT_expr_iterr:_Cb\xint_gooble_i}%
\def\XINT_expr_iterr:_Goon  #1!#2#{\XINT_expr_iterr:_Goon_a{#1}}%
\def\XINT_expr_iterr:_Goon_a  #1#2#3~#4?%
{%
    \expandafter\XINT_expr_iterr:_Cb \expanded{\unexpanded{#3~}\xintTrim{-2}{#1#4}}0?%
}%
```

11.25.10 rrseq()

When func_rrseq has its turn, initial segment has been scanned by oparen, the ; mimicking the rôle of a closing parenthesis, and stopping further expansion. #2 = \xint_c_i and #3 are left-over trash.

```
\def\XINT_expr_func_rrseq   {\XINT_allexpr_rrseq \xintbareeval      }%
\def\XINT_flexpr_func_rrseq {\XINT_allexpr_rrseq \xintbarefloateval }%
```

```
\def\XINT_iiexpr_func_rrseq {\XINT_allexpr_rrseq \xintbareiieval    }%
\def\XINT_allexpr_rrseq #1#2#3#4%
{%
    \expandafter\XINT_expr_rrseqx\expandafter#1\expanded
       {\unexpanded{{#4}}{\xintRevWithBraces{#4}}\expandafter}%
    \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
}%
\def\XINT_expr_rrseqx #1#2#3#4#5%
{%
    \XINT:NEhook:rrseq\XINT_expr_rrseqy\romannumeral0#1(#5)\relax {#2}{#3}#4#1%
}%
\def\XINT_expr_rrseqy #1#2#3#4#5#6%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded \bgroup {\iffalse}\fi
    #2\XINT_expr_rrseq:_b {#6#5\relax !#4}#1^~#30?\XINT_expr_cb_and_getop
}%
\def\XINT_expr_rrseq:_b #1#2%
{%
    \ifx +#2\xint_dothis\XINT_expr_rrseq:_Ca\fi
    \ifx !#2!\xint_dothis\XINT_expr_rrseq:_noop\fi
    \ifx ^#2\xint_dothis\XINT_expr_rrseq:_end\fi
    \xint_orthat{\XINT_expr_rrseq:_c}{#2}{#1}%
}%
\def\XINT_expr_rrseq:_noop #1{\XINT_expr_rrseq:_b }%
\def\XINT_expr_rrseq:_end  #1#2~#3?{\iffalse{\fi}}%
\def\XINT_expr_rrseq:_c #1#2{\expandafter\XINT_expr_rrseq:_d\romannumeral0#2{{#1}}{#2}}%
\def\XINT_expr_rrseq:_d #1{\ifx ^#1\xint_dothis\XINT_expr_rrseq:_abort\fi
                           \ifx ?#1\xint_dothis\XINT_expr_rrseq:_break\fi
                           \ifx !#1\xint_dothis\XINT_expr_rrseq:_omit\fi
                           \xint_orthat{\XINT_expr_rrseq:_goon {#1}}}%
\def\XINT_expr_rrseq:_abort #1!#2^~#3?{\iffalse{\fi}}%
\def\XINT_expr_rrseq:_break #1!#2^~#3?{#1\iffalse{\fi}}%
\def\XINT_expr_rrseq:_omit  #1!#2#{\expandafter\XINT_expr_rrseq:_b\xint_gobble_i}%
\def\XINT_expr_rrseq:_goon  #1!#2#{\XINT_expr_rrseq:_goon_a {#1}}%
\def\XINT_expr_rrseq:_goon_a  #1#2#3~#4?%
{%
    #1\expandafter\XINT_expr_rrseq:_b\expanded{\unexpanded{#3~}\xintTrim{-2}{#1#4}}0?%
}%
\def\XINT_expr_rrseq:_Ca #1#2#3{\XINT_expr_rrseq:_Cc#3.{#2}}%
\def\XINT_expr_rrseq:_Cb #1{\expandafter\XINT_expr_rrseq:_Cc\the\numexpr#1+\xint_c_i.}%
\def\XINT_expr_rrseq:_Cc #1.#2{\expandafter\XINT_expr_rrseq:_D\romannumeral0#2{{#1}}{#1}{#2}}%
\def\XINT_expr_rrseq:_D #1{\ifx ^#1\xint_dothis\XINT_expr_rrseq:_abort\fi
                           \ifx ?#1\xint_dothis\XINT_expr_rrseq:_break\fi
                           \ifx !#1\xint_dothis\XINT_expr_rrseq:_Omit\fi
                           \xint_orthat{\XINT_expr_rrseq:_Goon {#1}}}%
\def\XINT_expr_rrseq:_Omit  #1!#2#{\expandafter\XINT_expr_rrseq:_Cb\xint_gobble_i}%
\def\XINT_expr_rrseq:_Goon  #1!#2#{\XINT_expr_rrseq:_Goon_a {#1}}%
\def\XINT_expr_rrseq:_Goon_a  #1#2#3~#4?%
{%
    #1\expandafter\XINT_expr_rrseq:_Cb\expanded{\unexpanded{#3~}\xintTrim{-2}{#1#4}}0?%
}%
```

```
\catcode`? 11
```

11.26 Pseudo-functions related to N-dimensional hypercubic lists

11.26.1 ndseq()

New with 1.4. 2020/01/23. It is derived from subsm() but instead of evaluating one expression according to one value per variable, it constructs a nested bracketed seq... this means the expression is parsed each time ! Anyway, proof of concept. Nota Bene : omit, abort, break() work !

```
\def\XINT_expr_onliteral_ndseq
{%
    \expandafter\XINT_allexpr_ndseq_f
    \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
}%
\def\XINT_allexpr_ndseq_f #1#2{\xint_c_ii^v `{ndseqx}#2)\relax #1}%
\def\XINT_expr_func_ndseqx
{%
    \expandafter\XINT_allexpr_ndseqx\expandafter\xintbareeval
    \expandafter{\romannumeral0\expandafter\xint_gobble_i\string}%
    \expandafter\xintrevwithbraces
    \expanded\bgroup{\iffalse}\fi\XINT_allexpr_ndseq_A\XINT_expr_oparen
}%
\def\XINT_flexpr_func_ndseqx
{%
    \expandafter\XINT_allexpr_ndseqx\expandafter\xintbarefloateval
    \expandafter{\romannumeral0\expandafter\xint_gobble_i\string}%
    \expandafter\xintrevwithbraces
    \expanded\bgroup{\iffalse}\fi\XINT_allexpr_ndseq_A\XINT_flexpr_oparen
}%
\def\XINT_iiexpr_func_ndseqx
{%
    \expandafter\XINT_allexpr_ndseqx\expandafter\xintbareiieval
    \expandafter{\romannumeral0\expandafter\xint_gobble_i\string}%
    \expandafter\xintrevwithbraces
    \expanded\bgroup{\iffalse}\fi\XINT_allexpr_ndseq_A\XINT_iiexpr_oparen
}%
\def\XINT_allexpr_ndseq_A #1#2#3%
{%
    \ifx#2\xint_c_
       \expandafter\XINT_allexpr_ndseq_C
    \else
       \expandafter\XINT_allexpr_ndseq_B
    \fi #1%
}%
\def\XINT_allexpr_ndseq_B #1#2#3#4=%
{%
    {#2}{\xint_zapspaces#3#4 \xint_gobble_i}%
    \expandafter\XINT_allexpr_ndseq_A\expandafter#1\romannumeral`&&@#1%
}%
```

#1 = \xintbareeval, or \xintbarefloateval or \xintbareiieval #2 = values for last coordinate

```
\def\XINT_allexpr_ndseq_C #1#2{{#2}\iffalse{{{\fi}}}}%
```

```
    #1 = \xintbareeval or \xintbarefloateval or \xintbareiieval  #2 = {valuesN}...{values2}{var2}{values1}
  #3 = {var1}  #4 = the expression to evaluate
```

```
\def\XINT_allexpr_ndseqx #1#2#3#4%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded
    \bgroup
      \romannumeral0#1\empty
      \expanded{\xintReplicate{\xintLength{{#3}#2}/2}{[seq(}%
                \unexpanded{#4}%
                \XINT_allexpr_ndseqx_a #2{#3}^^%
                }%
      \relax
    \iffalse{\fi\expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT_allexpr_ndseqx_a #1#2%
{%
    \xint_gob_til_^ #1\XINT_allexpr_ndseqx_e ^%
    \unexpanded{,#2=\XINTfstop.{#1})]}\XINT_allexpr_ndseqx_a
}%
\def\XINT_allexpr_ndseqx_e ^#1\XINT_allexpr_ndseqx_a{}%
```

11.26.2 ndmap()

New with 1.4. 2020/01/24.

```
\def\XINT_expr_onliteral_ndmap #1,{\xint_c_ii^v `{ndmapx}\XINTfstop.{#1};}%
\def\XINT_expr_func_ndmapx #1#2#3%
{%
    \expandafter\XINT_allexpr_ndmapx
    \csname XINT_expr_func_\xint_zapspaces #3 \xint_gobble_i\endcsname
    \XINT_expr_oparen
}%
\def\XINT_flexpr_func_ndmapx #1#2#3%
{%
    \expandafter\XINT_allexpr_ndmapx
    \csname XINT_flexpr_func_\xint_zapspaces #3 \xint_gobble_i\endcsname
    \XINT_flexpr_oparen
}%
\def\XINT_iiexpr_func_ndmapx #1#2#3%
{%
    \expandafter\XINT_allexpr_ndmapx
    \csname XINT_iiexpr_func_\xint_zapspaces #3 \xint_gobble_i\endcsname
    \XINT_iiexpr_oparen
}%
\def\XINT_allexpr_ndmapx #1#2%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded\bgroup{\iffalse}\fi
    \expanded
```

```
2608       {\noexpand\XINT:NEhook:x:ndmapx
2609        \noexpand\XINT_allexpr_ndmapx_a
2610        \noexpand#1{}\expandafter}%
2611     \expanded\bgroup\expandafter\XINT_allexpr_ndmap_A
2612                     \expandafter#2\romannumeral`&&@#2%
2613 }%
2614 \def\XINT_allexpr_ndmap_A #1#2#3%
2615 {%
2616     \ifx#3;%
2617        \expandafter\XINT_allexpr_ndmap_B
2618     \else
2619        \xint_afterfi{\XINT_allexpr_ndmap_C#2#3}%
2620     \fi #1%
2621 }%
2622 \def\XINT_allexpr_ndmap_B #1#2%
2623 {%
2624     {#2}\expandafter\XINT_allexpr_ndmap_A\expandafter#1\romannumeral`&&@#1%
2625 }%
2626 \def\XINT_allexpr_ndmap_C #1#2#3#4%
2627 {%
2628     {#4}^\relax\iffalse{{{\fi}}}#1#2%
2629 }%
2630 \def\XINT_allexpr_ndmapx_a #1#2#3%
2631 {%
2632     \xint_gob_til_^ #3\XINT_allexpr_ndmapx_l ^%
2633     \XINT_allexpr_ndmapx_b #1{#2}{#3}%
2634 }%
2635 \def\XINT_allexpr_ndmapx_l ^#1\XINT_allexpr_ndmapx_b #2#3#4\relax
2636 {%
2637     #2\empty\xint_firstofone{#3}%
2638 }%
2639 \def\XINT_allexpr_ndmapx_b #1#2#3#4\relax
2640 {%
2641     {\iffalse}\fi\XINT_allexpr_ndmapx_c {#4\relax}#1{#2}#3^%
2642 }%
2643 \def\XINT_allexpr_ndmapx_c #1#2#3#4%
2644 {%
2645     \xint_gob_til_^ #4\XINT_allexpr_ndmapx_e ^%
2646     \XINT_allexpr_ndmapx_a #2{#3{#4}}#1%
2647     \XINT_allexpr_ndmapx_c  {#1}#2{#3}%
2648 }%
2649 \def\XINT_allexpr_ndmapx_e ^#1\XINT_allexpr_ndmapx_c
2650    {\iffalse{\fi}\xint_gobble_iii}%
```

11.26.3 ndfillraw()

New with 1.4. 2020/01/24. J'hésite à autoriser un #1 quelconque, ou plutôt à le wrapper dans un \xintbareval. Mais il faut alors distinguer les trois. De toute façon les variables ne marcheraient pas donc j'hésite à mettre un wrapper automatique. Mais ce n'est pas bien d'autoriser l'injection de choses quelconques.

Pour des choses comme ndfillraw(\xintRandomBit,[10,10]).

Je n'aime pas le nom !. Le changer. ndconst? Surtout je n'aime pas que dans le premier argument il faut rajouter explicitement si nécessaire \xintiiexpr wrap.

```
\def\XINT_expr_onliteral_ndfillraw #1,{\xint_c_ii^v `{ndfillrawx}\XINTfstop.{{#1}},}%
\def\XINT_expr_func_ndfillrawx #1#2#3%
{%
    \expandafter#1\expandafter#2\expanded{{{\XINT_allexpr_ndfillrawx_a #3}}}%
}%
\let\XINT_iiexpr_func_ndfillrawx\XINT_expr_func_ndfillrawx
\let\XINT_flexpr_func_ndfillrawx\XINT_expr_func_ndfillrawx
\def\XINT_allexpr_ndfillrawx_a #1#2%
{%
    \expandafter\XINT_allexpr_ndfillrawx_b
    \romannumeral0\xintApply{\xintNum}{#2}^\relax {#1}%
}%
\def\XINT_allexpr_ndfillrawx_b #1#2\relax#3%
{%
    \xint_gob_til_^ #1\XINT_allexpr_ndfillrawx_c ^%
    \xintReplicate{#1}{{\XINT_allexpr_ndfillrawx_b #2\relax {#3}}}%
}%
\def\XINT_allexpr_ndfillrawx_c ^\xintReplicate #1#2%
{%
    \expandafter\XINT_allexpr_ndfillrawx_d\xint_firstofone #2%
}%
\def\XINT_allexpr_ndfillrawx_d\XINT_allexpr_ndfillrawx_b \relax #1{#1}%
```

11.27 Other pseudo-functions: bool(), togl(), protect(), qraw(), qint(), qfrac(), qfloat(), qrand(), random(), rbit()

bool, togl and protect use delimited macros. They are not true functions, they turn off the parser to gather their "variable".

1.2. adds qint(), qfrac(), qfloat().

1.3c. adds qraw(). Useful to limit impact on TEX memory from abuse of \csname's storage when generating many comma separated values from a loop.

1.3e. qfloat() keeps a short mantissa if possible.

They allow the user to hand over quickly a big number to the parser, spaces not immediately removed but should be harmless in general. The qraw() does no post-processing at all apart complete expansion, useful for comma-separated values, but must be obedient to (non really documented) expected format. Each uses a delimited macro, the closing parenthesis can not emerge from expansion.

1.3b. random(), qrand() Function-like syntax but with no argument currently, so let's use fast parsing which requires though the closing parenthesis to be explicit.

Attention that qraw() which pre-supposes knowledge of internal storage model is fragile and may break at any release.
1.4 adds rbit(). Short for random bit.

```
\def\XINT_expr_onliteral_bool #1)%
    {\expandafter\XINT_expr_put_op_first\expanded{{{\xintBool{#1}}}\expandafter
     }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_togl #1)%
    {\expandafter\XINT_expr_put_op_first\expanded{{{\xintToggle{#1}}}\expandafter
     }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_protect #1)%
    {\expandafter\XINT_expr_put_op_first\expanded{{{\detokenize{#1}}}\expandafter
```

```
      }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_qint #1)%
    {\expandafter\XINT_expr_put_op_first\expanded{{{\xintiNum{#1}}}\expandafter
      }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_qfrac #1)%
    {\expandafter\XINT_expr_put_op_first\expanded{{{\xintRaw{#1}}}\expandafter
      }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_qfloat #1)%
    {\expandafter\XINT_expr_put_op_first\expanded{{{\XINTinFloatSdigits{#1}}}\expandafter
      }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_qraw #1)%
    {\expandafter\XINT_expr_put_op_first\expanded{{#1}\expandafter
      }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_random #1)%
    {\expandafter\XINT_expr_put_op_first\expanded{{{\XINTinRandomFloatSdigits}}\expandafter
      }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_qrand #1)%
   {\expandafter\XINT_expr_put_op_first\expanded{{{\XINTinRandomFloatSixteen}}\expandafter
      }\romannumeral`&&@\XINT_expr_getop}%
\def\XINT_expr_onliteral_rbit #1)%
   {\expandafter\XINT_expr_put_op_first\expanded{{{\xintRandBit}}\expandafter
      }\romannumeral`&&@\XINT_expr_getop}%
```

11.28 Regular built-in functions: num(), reduce(), preduce(), abs(), sgn(), frac(), floor(), ceil(), sqr(), ?(), !(), not(), odd(), even(), isint(), isone(), factorial(), sqrt(), sqrtr(), inv(), round(), trunc(), float(), sfloat(), ilog10(), divmod(), mod(), binomial(), pfactorial(), randrange(), iquo(), irem(), gcd(), lcm(), max(), min(), `+`(), `*`(), all(), any(), xor(), len(), first(), last(), reversed(), if(), ifint(), ifone(), ifsgn(), nuple(), unpack(), flat() and zip()

```
\def\XINT:expr:f:one:and:opt #1#2#3!#4#5%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                       \expandafter\xint_secondoftwo\fi
    {#4}{#5[\xintNum{#2}]}{#1}%
}%
\def\XINT:expr:f:tacitzeroifone #1#2#3!#4#5%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                       \expandafter\xint_secondoftwo\fi
    {#4{0}}{#5{\xintNum{#2}}}{#1}%
}%
\def\XINT:expr:f:iitacitzeroifone #1#2#3!#4%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                       \expandafter\xint_secondoftwo\fi
    {#4{0}}{#4{#2}}{#1}%
}%
\def\XINT_expr_func_num #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
```

```
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintNum#3}}%
}%
\let\XINT_flexpr_func_num\XINT_expr_func_num
\let\XINT_iiexpr_func_num\XINT_expr_func_num
\def\XINT_expr_func_reduce #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintIrr#3}}%
}%
\let\XINT_flexpr_func_reduce\XINT_expr_func_reduce
\def\XINT_expr_func_preduce #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintPIrr#3}}%
}%
\let\XINT_flexpr_func_preduce\XINT_expr_func_preduce
\def\XINT_expr_func_abs #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintAbs#3}}%
}%
\let\XINT_flexpr_func_abs\XINT_expr_func_abs
\def\XINT_iiexpr_func_abs #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiAbs#3}}%
}%
\def\XINT_expr_func_sgn #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintSgn#3}}%
}%
\let\XINT_flexpr_func_sgn\XINT_expr_func_sgn
\def\XINT_iiexpr_func_sgn #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiSgn#3}}%
}%
\def\XINT_expr_func_frac #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintTFrac#3}}%
}%
\def\XINT_flexpr_func_frac #1#2#3%
```

```
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\XINTinFloatFracdigits#3}}%
}%
```

no \XINT_iiexpr_func_frac

```
\def\XINT_expr_func_floor #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintFloor#3}}%
}%
\let\XINT_flexpr_func_floor\XINT_expr_func_floor
```

The floor and ceil functions in \xintiiexpr require protect(a/b) or, better, \qfrac(a/b); else the / will be executed first and do an integer rounded division.

```
\def\XINT_iiexpr_func_floor #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiFloor#3}}%
}%
\def\XINT_expr_func_ceil #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintCeil#3}}%
}%
\let\XINT_flexpr_func_ceil\XINT_expr_func_ceil
\def\XINT_iiexpr_func_ceil #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiCeil#3}}%
}%
\def\XINT_expr_func_sqr #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintSqr#3}}%
}%
\def\XINTinFloatSqr#1{\XINTinFloatMul{#1}{#1}}%
\def\XINT_flexpr_func_sqr #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\XINTinFloatSqr#3}}%
}%
\def\XINT_iiexpr_func_sqr #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
```

```
    {\romannumeral`&&@\xintiiSqr#3}}%
}%
\def\XINT_expr_func_? #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiIsNotZero#3}}%
}%
\let\XINT_flexpr_func_? \XINT_expr_func_?
\let\XINT_iiexpr_func_? \XINT_expr_func_?
\def\XINT_expr_func_! #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiIsZero#3}}%
}%
\let\XINT_flexpr_func_! \XINT_expr_func_!
\let\XINT_iiexpr_func_! \XINT_expr_func_!
\def\XINT_expr_func_not #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiIsZero#3}}%
}%
\let\XINT_flexpr_func_not \XINT_expr_func_not
\let\XINT_iiexpr_func_not \XINT_expr_func_not
\def\XINT_expr_func_odd #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintOdd#3}}%
}%
\let\XINT_flexpr_func_odd\XINT_expr_func_odd
\def\XINT_iiexpr_func_odd #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiOdd#3}}%
}%
\def\XINT_expr_func_even #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintEven#3}}%
}%
\let\XINT_flexpr_func_even\XINT_expr_func_even
\def\XINT_iiexpr_func_even #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiEven#3}}%
}%
```

```
\def\XINT_expr_func_isint #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintIsInt#3}}%
}%
\def\XINT_flexpr_func_isint #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintFloatIsInt#3}}%
}%
\let\XINT_iiexpr_func_isint\XINT_expr_func_isint % ? perhaps rather always 1
\def\XINT_expr_func_isone #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintIsOne#3}}%
}%
\let\XINT_flexpr_func_isone\XINT_expr_func_isone
\def\XINT_iiexpr_func_isone #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiIsOne#3}}%
}%
\def\XINT_expr_func_factorial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:and:opt:direct
    \XINT:expr:f:one:and:opt #3,!\xintFac\XINTinFloatFac
    }}%
}%
\def\XINT_flexpr_func_factorial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:and:opt:direct
    \XINT:expr:f:one:and:opt#3,!\XINTinFloatFacdigits\XINTinFloatFac
    }}%
}%
\def\XINT_iiexpr_func_factorial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiFac#3}}%
}%
\def\XINT_expr_func_sqrt #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:and:opt:direct
    \XINT:expr:f:one:and:opt #3,!\XINTinFloatSqrtdigits\XINTinFloatSqrt
    }}%
```

```
}%
\let\XINT_flexpr_func_sqrt\XINT_expr_func_sqrt
\def\XINT_expr_func_sqrt_ #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\XINTinFloatSqrtdigits#3}}%
}%
\let\XINT_flexpr_func_sqrt_\XINT_expr_func_sqrt_
\def\XINT_iiexpr_func_sqrt #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiSqrt#3}}%
}%
\def\XINT_iiexpr_func_sqrtr #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiSqrtR#3}}%
}%
\def\XINT_expr_func_inv #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintInv#3}}%
}%
\def\XINT_flexpr_func_inv #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\XINTinFloatInv#3}}%
}%
\def\XINT_expr_func_round #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:tacitzeroifone:direct
    \XINT:expr:f:tacitzeroifone #3,!\xintiRound\xintRound
    }}%
}%
\let\XINT_flexpr_func_round\XINT_expr_func_round
\def\XINT_iiexpr_func_round #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:iitacitzeroifone:direct
    \XINT:expr:f:iitacitzeroifone #3,!\xintiRound
    }}%
}%
\def\XINT_expr_func_trunc #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:tacitzeroifone:direct
```

```
    \XINT:expr:f:tacitzeroifone #3,!\xintiTrunc\xintTrunc
    }}%
}%
\let\XINT_flexpr_func_trunc\XINT_expr_func_trunc
\def\XINT_iiexpr_func_trunc #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:iitacitzeroifone:direct
    \XINT:expr:f:iitacitzeroifone #3,!\xintiTrunc
    }}%
}%
```

Hesitation at 1.3e about using \XINTinFloatSdigits and \XINTinFloatS. Finally I add a sfloat() function. It helps for xinttrig.sty.

```
\def\XINT_expr_func_float #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:and:opt:direct
    \XINT:expr:f:one:and:opt #3,!\XINTinFloatdigits\XINTinFloat
    }}%
}%
\let\XINT_flexpr_func_float\XINT_expr_func_float
```

float_() added at 1.4. Does not check for optional argument. Useful to transfer functions defined with \xintdeffunc to functions usable in \xintfloateval. I hesitated briefly about notation but here we go. Unfortunately I will have to document it (contrarily to sqrt_).

No need to do same for sfloat() currently used in xinttrig.sty to go from float to expr, because sfloat(x) sees there is no optional argument.

Still I wonder if better would not be to have some function «single()» which signals to outer one it is a single argument? Must think about this. Too late now for 1.4.

```
\def\XINT_expr_func_float_ #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\XINTinFloatdigits#3}}%
}%
\let\XINT_flexpr_func_float_\XINT_expr_func_float_
\def\XINT_expr_func_sfloat #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:and:opt:direct
    \XINT:expr:f:one:and:opt #3,!\XINTinFloatSdigits\XINTinFloatS
    }}%
}%
\let\XINT_flexpr_func_sfloat\XINT_expr_func_sfloat
% \XINT_iiexpr_func_sfloat not defined
\expandafter\def\csname XINT_expr_func_ilog10\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:and:opt:direct
    \XINT:expr:f:one:and:opt #3,!\xintiLogTen\XINTFloatiLogTen
    }}%
}%
```

```
\expandafter\def\csname XINT_flexpr_func_ilog10\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:and:opt:direct
    \XINT:expr:f:one:and:opt #3,!\XINTFloatiLogTendigits\XINTFloatiLogTen
    }}%
}%
\expandafter\def\csname XINT_iiexpr_func_ilog10\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintiiLogTen#3}}%
}%
\def\XINT_expr_func_divmod #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintDivMod #3}}%
}%
\def\XINT_flexpr_func_divmod #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\XINTinFloatDivMod #3}}%
}%
\def\XINT_iiexpr_func_divmod #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintiiDivMod #3}}%
}%
\def\XINT_expr_func_mod #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintMod#3}}%
}%
\def\XINT_flexpr_func_mod #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\XINTinFloatMod#3}}%
}%
\def\XINT_iiexpr_func_mod #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintiiMod#3}}%
}%
\def\XINT_expr_func_binomial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
```

```
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintBinomial #3}}%
}%
\def\XINT_flexpr_func_binomial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\XINTinFloatBinomial #3}}%
}%
\def\XINT_iiexpr_func_binomial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintiiBinomial #3}}%
}%
\def\XINT_expr_func_pfactorial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintPFactorial #3}}%
}%
\def\XINT_flexpr_func_pfactorial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\XINTinFloatPFactorial #3}}%
}%
\def\XINT_iiexpr_func_pfactorial #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintiiPFactorial #3}}%
}%
\def\XINT_expr_func_randrange #1#2#3%
{%
    \expandafter #1\expandafter #2\expanded{{{%
    \XINT:expr:randrange #3,!%
    }}}%
}%
\let\XINT_flexpr_func_randrange\XINT_expr_func_randrange
\def\XINT_iiexpr_func_randrange #1#2#3%
{%
    \expandafter #1\expandafter #2\expanded{{{%
    \XINT:iiexpr:randrange #3,!%
    }}}%
}%
\def\XINT:expr:randrange #1#2#3!%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                       \expandafter\xint_secondoftwo\fi
    {\xintiiRandRange{\XINT:NEhook:f:one:from:one:direct\xintNum{#1}}}%
    {\xintiiRandRangeAtoB{\XINT:NEhook:f:one:from:one:direct\xintNum{#1}}%
```

```
                          {\XINT:NEhook:f:one:from:one:direct\xintNum{#2}}%
     }%
}%
\def\XINT:iiexpr:randrange #1#2#3!%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {\xintiiRandRange{#1}}%
    {\xintiiRandRangeAtoB{#1}{#2}}%
}%
\def\XINT_iiexpr_func_iquo #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintiiQuo #3}}%
}%
\def\XINT_iiexpr_func_irem #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
    \XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\xintiiRem #3}}%
}%
\def\XINT_expr_func_gcd #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_GCDof#3^}}%
}%
\let\XINT_flexpr_func_gcd\XINT_expr_func_gcd
\def\XINT_iiexpr_func_gcd #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_iiGCDof#3^}}%
}%
\def\XINT_expr_func_lcm #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_LCMof#3^}}%
}%
\let\XINT_flexpr_func_lcm\XINT_expr_func_lcm
\def\XINT_iiexpr_func_lcm #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_iiLCMof#3^}}%
}%
\def\XINT_expr_func_max #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_Maxof#3^}}%
}%
\def\XINT_iiexpr_func_max #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
```

```
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_iiMaxof#3^}}%
}%
\def\XINT_flexpr_func_max #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINTinFloatMaxof#3^}}%
}%
\def\XINT_expr_func_min #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_Minof#3^}}%
}%
\def\XINT_iiexpr_func_min #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_iiMinof#3^}}%
}%
\def\XINT_flexpr_func_min #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINTinFloatMinof#3^}}%
}%
\expandafter
\def\csname XINT_expr_func_+\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_Sum#3^}}%
}%
\expandafter
\def\csname XINT_flexpr_func_+\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINTinFloatSum#3^}}%
}%
\expandafter
\def\csname XINT_iiexpr_func_+\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_iiSum#3^}}%
}%
\expandafter
\def\csname XINT_expr_func_*\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_Prd#3^}}%
}%
\expandafter
\def\csname XINT_flexpr_func_*\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINTinFloatPrd#3^}}%
}%
```

```
\expandafter
\def\csname XINT_iiexpr_func_*\endcsname #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_iiPrd#3^}}%
}%
\def\XINT_expr_func_all #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_ANDof#3^}}%
}%
\let\XINT_flexpr_func_all\XINT_expr_func_all
\let\XINT_iiexpr_func_all\XINT_expr_func_all
\def\XINT_expr_func_any #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_ORof#3^}}%
}%
\let\XINT_flexpr_func_any\XINT_expr_func_any
\let\XINT_iiexpr_func_any\XINT_expr_func_any
\def\XINT_expr_func_xor #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{\expandafter
    {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_XORof#3^}}%
}%
\let\XINT_flexpr_func_xor\XINT_expr_func_xor
\let\XINT_iiexpr_func_xor\XINT_expr_func_xor
\def\XINT_expr_func_len #1#2#3%
{%
    \expandafter#1\expandafter#2\expandafter{\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:noeval:from:braced:u\xintLength#3^%
    }}%
}%
\let\XINT_flexpr_func_len \XINT_expr_func_len
\let\XINT_iiexpr_func_len \XINT_expr_func_len
\def\XINT_expr_func_first #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:noeval:from:braced:u\xintFirstOne#3^%
    }%
}%
\let\XINT_flexpr_func_first\XINT_expr_func_first
\let\XINT_iiexpr_func_first\XINT_expr_func_first
\def\XINT_expr_func_last #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:noeval:from:braced:u\xintLastOne#3^%
    }%
}%
\let\XINT_flexpr_func_last\XINT_expr_func_last
\let\XINT_iiexpr_func_last\XINT_expr_func_last
\def\XINT_expr_func_reversed #1#2#3%
```

```
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:reverse\XINT_expr_reverse
    #3^^#3\xint:\xint:\xint:\xint:
          \xint:\xint:\xint:\xint:\xint_bye
    }%
}%
\def\XINT_expr_reverse #1#2%
{%
    \if ^\noexpand#2%
        \expandafter\XINT_expr_reverse:_one_or_none\string#1.%
    \else
        \expandafter\XINT_expr_reverse:_at_least_two
    \fi
}%
\def\XINT_expr_reverse:_at_least_two #1^^{\XINT_revwbr_loop {}}%
\def\XINT_expr_reverse:_one_or_none #1%
{%
    \if #1\bgroup\xint_dothis\XINT_expr_reverse:_nutple\fi
    \if #1^\xint_dothis\XINT_expr_reverse:_nil\fi
    \xint_orthat\XINT_expr_reverse:_leaf
}%
\edef\XINT_expr_reverse:_nil #1\xint_bye{\noexpand\fi\space}%
\def\XINT_expr_reverse:_leaf#1\fi #2\xint:#3\xint_bye{\fi\xint_gob_andstop_i#2}%
\def\XINT_expr_reverse:_nutple%
{%
    \expandafter\XINT_expr_reverse:_nutple_a\expandafter{\string}%
}%
\def\XINT_expr_reverse:_nutple_a #1^#2\xint:#3\xint_bye
{%
    \fi\expandafter
    {\romannumeral0\XINT_revwbr_loop{}#2\xint:#3\xint_bye}%
}%
\let\XINT_flexpr_func_reversed\XINT_expr_func_reversed
\let\XINT_iiexpr_func_reversed\XINT_expr_func_reversed
\def\XINT_expr_func_if #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:branch{\romannumeral`&&@\xintiiifNotZero #3}}%
}%
\let\XINT_flexpr_func_if\XINT_expr_func_if
\let\XINT_iiexpr_func_if\XINT_expr_func_if
\def\XINT_expr_func_ifint #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:branch{\romannumeral`&&@\xintifInt #3}}%
}%
\let\XINT_iiexpr_func_ifint\XINT_expr_func_ifint
\def\XINT_flexpr_func_ifint #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:branch{\romannumeral`&&@\xintifFloatInt #3}}%
```

```
}%
\def\XINT_expr_func_ifone #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:branch{\romannumeral`&&@\xintifOne #3}}%
}%
\let\XINT_flexpr_func_ifone\XINT_expr_func_ifone
\def\XINT_iiexpr_func_ifone #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:branch{\romannumeral`&&@\xintiiifOne #3}}%
}%
\def\XINT_expr_func_ifsgn #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:branch{\romannumeral`&&@\xintiiifSgn #3}}%
}%
\let\XINT_flexpr_func_ifsgn\XINT_expr_func_ifsgn
\let\XINT_iiexpr_func_ifsgn\XINT_expr_func_ifsgn
\def\XINT_expr_func_nuple #1#2#3{#1#2{{#3}}}%
\let\XINT_flexpr_func_nuple\XINT_expr_func_nuple
\let\XINT_iiexpr_func_nuple\XINT_expr_func_nuple
\def\XINT_expr_func_unpack #1#2%#3%
   {\expandafter#1\expandafter#2\romannumeral0\XINT:NEhook:unpack}%
\let\XINT_flexpr_func_unpack\XINT_expr_func_unpack
\let\XINT_iiexpr_func_unpack\XINT_expr_func_unpack
\def\XINT_expr_func_flat #1#2%#3%
{%
    \expandafter#1\expandafter#2\expanded
    \XINT:NEhook:x:flatten\XINT:expr:flatten
}%
\let\XINT_flexpr_func_flat\XINT_expr_func_flat
\let\XINT_iiexpr_func_flat\XINT_expr_func_flat
\let\XINT:NEhook:x:flatten\empty
\def\XINT_expr_func_zip #1#2%#3%
{%
    \expandafter#1\expandafter#2\romannumeral`&&@%
    \XINT:NEhook:x:zip\XINT:expr:zip
}%
\let\XINT_flexpr_func_zip\XINT_expr_func_zip
\let\XINT_iiexpr_func_zip\XINT_expr_func_zip
\let\XINT:NEhook:x:zip\empty
\def\XINT:expr:zip#1{\expandafter{\expanded\XINT_zip_A#1\xint_bye\xint_bye}}%
```

11.29 User declared functions

It is possible that the author actually does understand at this time the \xintNewExpr/\xintdeffunc refactored code and mechanisms for the first time since 2014: past evolutions such as the 2018 1.3 refactoring were done a bit in the fog (although they did accomplish a crucial step).

The 1.4 version of function and macro definitions is much more powerful than 1.3 one. But the mechanisms such as «omit», «abort» and «break()» in iter() et al. can't be translated into much else than their actual code when they potentially have to apply to non-numeric only context. The 1.4 \xintdeffunc is thus apparently able to digest them but its pre-parsing benefits are limited

compared to simply assigning such parts of an expression to a mock-function created by \xintNewFunction (which creates simply a TeX macro from its substitution expression in macro parameters and add syntactic sugar to let it appear to \xintexpr as a genuine «function» although nothing of the syntax has really been pre-parsed.)

At 1.4 fetching the expression up to final semi-colon is done using \XINT_expr_fetch_to_semicolon, hence semi-colons arising in the syntax do not need to be hidden inside braces.

11.29.1 \xintdeffunc, \xintdefiifunc, \xintdeffloatfunc

1.2c (2015/11/12).

Note: it is possible to have same name assigned both to a variable and a function: things such as add(f(f), f=1..10) are possible.

1.2c (2015/11/13).

Function names first expanded then detokenized and cleaned of spaces.

1.2e (2015/11/21).

No \detokenize anymore on the function names. And #1(#2)#3=#4 parameter pattern to avoid to have to worry if a : is there and it is active.

1.2f (2016/02/22).

La macro associée à la fonction ne débute plus par un \romannumeral, car de toute façon elle est pour emploi dans \csname..\endcsname.

1.2f (2016/03/08).

Comma separated expressions allowed (formerly this required using parenthesis \xintdeffunc foo(x,..):=(.., .., ..);

1.3c (2018/06/17).

Usage of \xintexprSafeCatcodes to be compatible with an active semi-colon at time of use; the colon was not a problem (see ##3) already.

1.3e (??).

\xintdefefunc variant added for functions which will expand completely if used with numeric arguments in other function definitions. They can't be used for recursive definitions.

1.4 (2020/01/10).

Multi-letter variables can be used (with no prior declaration)

1.4 (2020/01/11).

The new internal data model has caused many worries initially (such as whether to allow functions with «ople» outputs in contrast to «numbers» or «nutples») but in the end all is simpler again and the refactoring of ? and ?? in function definitions allows to fuse inert functions (allowing recursive definitions) and expanding functions (expanding completely if with numeric arguments) into a single entity.

Thus the 1.3e \xintdefefunc, \xintdefiiefunc, \xintdeffloatefunc constructors of «expanding» functions are kept only as aliases of legacy \xintdeffunc et al. and deprecated.

A special situation is with functions of no variables. In that case it will be handled as an inert entity, else they would not be different from variables.

```
\def\XINT_tmpa #1#2#3#4#5%
{%
  \def #1##1(##2)##3={%
   \edef\XINT_deffunc_tmpa {##1}%
   \edef\XINT_deffunc_tmpa {\xint_zapspaces_o \XINT_deffunc_tmpa}%
   \def\XINT_deffunc_tmpb {0}%
   \edef\XINT_deffunc_tmpd {##2}%
   \edef\XINT_deffunc_tmpd {\xint_zapspaces_o\XINT_deffunc_tmpd}%
   \def\XINT_deffunc_tmpe {0}%
   \expandafter#5\romannumeral\XINT_expr_fetch_to_semicolon
  }%  end of \xintdeffunc_a definition
  \def#5##1{%
   \def\XINT_deffunc_tmpc{(##1)}%
   \ifnum\xintLength:f:csv{\XINT_deffunc_tmpd}>\xint_c_
     \xintFor ####1 in {\XINT_deffunc_tmpd}\do
      {%
       \xintifForFirst{\let\XINT_deffunc_tmpd\empty}{}%
       \def\XINT_deffunc_tmpf{####1}%
       \if*\xintFirstItem{####1}%
         \xintifForLast
         {%
          \def\XINT_deffunc_tmpe{1}%
          \edef\XINT_deffunc_tmpf{\xintTrim{1}{####1}}%
         }%
         {%
          \edef\XINT_deffunc_tmpf{\xintTrim{1}{####1}}%
          \xintMessage{xintexpr}{Error}
          {Only the last positional argument can be variadic. Trimmed ####1 to
          \XINT_deffunc_tmpf}%
         }%
       \fi
       \XINT_expr_makedummy{\XINT_deffunc_tmpf}%
       \edef\XINT_deffunc_tmpd{\XINT_deffunc_tmpd{\XINT_deffunc_tmpf}}%
       \edef\XINT_deffunc_tmpb {\the\numexpr\XINT_deffunc_tmpb+\xint_c_i}%
       \edef\XINT_deffunc_tmpc {subs(\unexpanded\expandafter{\XINT_deffunc_tmpc},%
                                \XINT_deffunc_tmpf=################\XINT_deffunc_tmpb)}%
      }%
   \fi
```

Place holder for comments. Logic at 1.4 is simplified here compared to earlier releases.

```
   \ifcase\XINT_deffunc_tmpb\space
     \expandafter\XINT_expr_defuserfunc_none\csname
   \else
     \expandafter\XINT_expr_defuserfunc\csname
   \fi
           XINT_#2_func_\XINT_deffunc_tmpa\expandafter\endcsname
   \csname XINT_#2_userfunc_\XINT_deffunc_tmpa\expandafter\endcsname
   \expandafter{\XINT_deffunc_tmpa}{#2}%
```

```
    \expandafter#3\csname XINT_#2_userfunc_\XINT_deffunc_tmpa\endcsname
                                [\XINT_deffunc_tmpb]{\XINT_deffunc_tmpc}%
    \ifxintverbose\xintMessage {xintexpr}{Info}
         {Function \XINT_deffunc_tmpa\space for \string\xint #4 parser
          associated to \string\XINT_#2_userfunc_\XINT_deffunc_tmpa\space
          with \ifxintglobaldefs global \fi meaning \expandafter\meaning
          \csname XINT_#2_userfunc_\XINT_deffunc_tmpa\endcsname}%
    \fi
    \xintFor* ####1 in {\XINT_deffunc_tmpd}:{\xintrestorevariablesilently{####1}}%
    \xintexprRestoreCatcodes
  }% end of \xintdeffunc_b definition
}%
\def\xintdeffunc      {\xintexprSafeCatcodes\xintdeffunc_a}%
\def\xintdefiifunc    {\xintexprSafeCatcodes\xintdefiifunc_a}%
\def\xintdeffloatfunc {\xintexprSafeCatcodes\xintdeffloatfunc_a}%
\XINT_tmpa\xintdeffunc_a     {expr}  \XINT_NewFunc     {expr}\xintdeffunc_b
\XINT_tmpa\xintdefiifunc_a   {iiexpr}\XINT_NewIIFunc   {iiexpr}\xintdefiifunc_b
\XINT_tmpa\xintdeffloatfunc_a{flexpr}\XINT_NewFloatFunc{floatexpr}\xintdeffloatfunc_b
\def\XINT_expr_defuserfunc_none #1#2#3#4%
{%
    \XINT_global
    \def #1##1##2##3%
    {%
         \expandafter##1\expandafter##2\expanded{%
           {\XINT:NEhook:usernoargfunc\csname XINT_#4_userfunc_#3\endcsname}%
         }%
    }%
}%
\let\XINT:NEhook:usernoargfunc \empty
\def\XINT_expr_defuserfunc #1#2#3#4%
{%
    \if0\XINT_deffunc_tmpe
    \XINT_global
    \def #1##1##2%##3%
    {%
         \expandafter ##1\expandafter##2\expanded\bgroup{\iffalse}\fi
         \XINT:NEhook:userfunc{XINT_#4_userfunc_#3}#2%##3%
    }%
    \else
    \def #1##1{%
    \XINT_global\def #1####1####2%####3%
    {%
         \expandafter ####1\expandafter####2\expanded\bgroup{\iffalse}\fi
         \XINT:NEhook:userfunc:argv{##1}{XINT_#4_userfunc_#3}#2%####3%
    }}\expandafter#1\expandafter{\the\numexpr\XINT_deffunc_tmpb-1}%
    \fi
}%
\def\XINT:NEhook:userfunc #1#2#3{#2#3\iffalse{{\fi}}}%
\def\XINT:NEhook:userfunc:argv #1#2#3#4%
   {\expandafter#3\expanded{\xintKeep{#1}{#4}{\xintTrim{#1}{#4}}}\iffalse{{\fi}}}%
\let\xintdefefunc\xintdeffunc
\let\xintdefiifunc\xintdefiifunc
```

```
\let\xintdeffloatefunc\xintdeffloatfunc
```

11.29.2 \xintdefufunc, \xintdefiiufunc, \xintdeffloatufunc

1.4

```
\def\XINT_tmpa #1#2#3#4#5#6%
{%
  \def #1##1(##2)##3={%
   \edef\XINT_defufunc_tmpa {##1}%
   \edef\XINT_defufunc_tmpa {\xint_zapspaces_o \XINT_defufunc_tmpa}%
   \edef\XINT_defufunc_tmpd {##2}%
   \edef\XINT_defufunc_tmpd {\xint_zapspaces_o\XINT_defufunc_tmpd}%
   \expandafter#5\romannumeral\XINT_expr_fetch_to_semicolon
  }% end of \xint_defufunc_a
  \def#5##1{%
   \def\XINT_defufunc_tmpc{(##1)}%
   \ifnum\xintLength:f:csv{\XINT_defufunc_tmpd}=\xint_c_i
     \expandafter#6%
   \else
   \xintMessage {xintexpr}{ERROR}
        {Universal functions must be functions of one argument only,
         but the declaration of \XINT_defufunc_tmpa\space
         has \xintLength:f:csv{\XINT_defufunc_tmpd} of them. Cancelled.}%
   \xintexprRestoreCatcodes
   \fi
   }% end of \xint_defufunc_b
  \def #6{%
   \XINT_expr_makedummy{\XINT_defufunc_tmpd}%
   \edef\XINT_defufunc_tmpc {subs(\unexpanded\expandafter{\XINT_defufunc_tmpc},%
                              \XINT_defufunc_tmpd=########1)}%
   \expandafter\XINT_expr_defuserufunc
   \csname XINT_#2_func_\XINT_defufunc_tmpa\expandafter\endcsname
   \csname XINT_#2_userufunc_\XINT_defufunc_tmpa\expandafter\endcsname
   \expandafter{\XINT_defufunc_tmpa}{#2}%
   \expandafter#3\csname XINT_#2_userufunc_\XINT_defufunc_tmpa\endcsname
                              [1]{\XINT_defufunc_tmpc}%
   \ifxintverbose\xintMessage {xintexpr}{Info}
        {Universal function \XINT_defufunc_tmpa\space for \string\xint #4 parser
         associated to \string\XINT_#2_userufunc_\XINT_defufunc_tmpa\space
         with \ifxintglobaldefs global \fi meaning \expandafter\meaning
         \csname XINT_#2_userufunc_\XINT_defufunc_tmpa\endcsname}%
   \fi
  }% end of \xint_defufunc_c
}%
\def\xintdefufunc      {\xintexprSafeCatcodes\xintdefufunc_a}%
\def\xintdefiiufunc    {\xintexprSafeCatcodes\xintdefiiufunc_a}%
\def\xintdeffloatufunc {\xintexprSafeCatcodes\xintdeffloatufunc_a}%
\XINT_tmpa\xintdefufunc_a     {expr}  \XINT_NewFunc     {expr}%
          \xintdefufunc_b\xintdefufunc_c
\XINT_tmpa\xintdefiiufunc_a   {iiexpr}\XINT_NewIIFunc   {iiexpr}%
          \xintdefiiufunc_b\xintdefiiufunc_c
\XINT_tmpa\xintdeffloatufunc_a{flexpr}\XINT_NewFloatFunc{floatexpr}%
```

```
3523           \xintdeffloatufunc_b\xintdeffloatufunc_c
3524 \def\XINT_expr_defuserufunc #1#2#3#4%
3525 {%
3526     \XINT_global
3527     \def #1##1##2%##3%
3528     {%
3529          \expandafter ##1\expandafter##2\expanded
3530          \XINT:NEhook:userufunc{XINT_#4_userufunc_#3}#2%##3%
3531     }%
3532 }%
3533 \def\XINT:NEhook:userufunc #1{\XINT:expr:mapwithin}%
```

11.29.3 \xintunassignexprfunc, \xintunassigniiexprfunc, \xintunassignfloatexprfunc

See the \xintunassignvar for the embarrassing explanations why I had not done that earlier. A bit lazy here, no warning if undefining something not defined, and attention no precaution respective built-in functions.

```
3534 \def\XINT_tmpa #1{\expandafter\def\csname xintunassign#1func\endcsname ##1{%
3535    \edef\XINT_unfunc_tmpa{##1}%
3536    \edef\XINT_unfunc_tmpa {\xint_zapspaces_o\XINT_unfunc_tmpa}%
3537    \XINT_global\expandafter
3538      \let\csname XINT_#1_func_\XINT_unfunc_tmpa\endcsname\xint_undefined
3539    \XINT_global\expandafter
3540      \let\csname XINT_#1_userfunc_\XINT_unfunc_tmpa\endcsname\xint_undefined
3541    \XINT_global\expandafter
3542      \let\csname XINT_#1_userufunc_\XINT_unfunc_tmpa\endcsname\xint_undefined
3543    \ifxintverbose\xintMessage {xintexpr}{Info}
3544     {Function \XINT_unfunc_tmpa\space for \string\xint #1 parser now
3545      \ifxintglobaldefs globally \fi undefined.}%
3546    \fi}}%
3547 \XINT_tmpa{expr}\XINT_tmpa{iiexpr}\XINT_tmpa{floatexpr}%
```

11.29.4 \xintNewFunction

1.2h (2016/11/20). Syntax is \xintNewFunction{<name>}[nb of arguments]{expression with #1, #2,... as in \xintNewExpr}. This defines a function for all three parsers but the expression parsing is delayed until function execution. Hence the expression admits all constructs, contrarily to \xintNewExpr or \xintdeffunc.

As the letters used for variables in \xintdeffunc, #1, #2, etc... can not stand for non numeric «oples», because at time of function call f(a, b, c, ...) how to decide if #1 stands for a or a, b etc... ? Or course «a» can be packed and thus the macro function can handle #1 as a «nutple» and for this be defined with the * unpacking operator being applied to it.

```
3548 \def\xintNewFunction #1#2[#3]#4%
3549 {%
3550   \edef\XINT_newfunc_tmpa {#1}%
3551   \edef\XINT_newfunc_tmpa {\xint_zapspaces_o \XINT_newfunc_tmpa}%
3552   \def\XINT_newfunc_tmpb ##1##2##3##4##5##6##7##8##9{#4}%
3553   \begingroup
3554     \ifcase #3\relax
3555         \toks0{}%
3556     \or \toks0{##1}%
3557     \or \toks0{##1##2}%
```

```
    \or \toks0{##1##2##3}%
    \or \toks0{##1##2##3##4}%
    \or \toks0{##1##2##3##4##5}%
    \or \toks0{##1##2##3##4##5##6}%
    \or \toks0{##1##2##3##4##5##6##7}%
    \or \toks0{##1##2##3##4##5##6##7##8}%
    \else \toks0{##1##2##3##4##5##6##7##8##9}%
    \fi
    \expandafter
  \endgroup\expandafter
  \XINT_global\expandafter
  \def\csname XINT_expr_macrofunc_\XINT_newfunc_tmpa\expandafter\endcsname
  \the\toks0\expandafter{\XINT_newfunc_tmpb
    {\XINTfstop.{{##1}}}{\XINTfstop.{{##2}}}{\XINTfstop.{{##3}}}%
    {\XINTfstop.{{##4}}}{\XINTfstop.{{##5}}}{\XINTfstop.{{##6}}}%
    {\XINTfstop.{{##7}}}{\XINTfstop.{{##8}}}{\XINTfstop.{{##9}}}}%
  \expandafter\XINT_expr_newfunction
    \csname XINT_expr_func_\XINT_newfunc_tmpa\expandafter\endcsname
    \expandafter{\XINT_newfunc_tmpa}\xintbareeval
  \expandafter\XINT_expr_newfunction
    \csname XINT_iiexpr_func_\XINT_newfunc_tmpa\expandafter\endcsname
    \expandafter{\XINT_newfunc_tmpa}\xintbareiieval
  \expandafter\XINT_expr_newfunction
    \csname XINT_flexpr_func_\XINT_newfunc_tmpa\expandafter\endcsname
    \expandafter{\XINT_newfunc_tmpa}\xintbarefloateval
  \ifxintverbose
    \xintMessage {xintexpr}{Info}
        {Function \XINT_newfunc_tmpa\space for the expression parsers is
         associated to \string\XINT_expr_macrofunc_\XINT_newfunc_tmpa\space
         with \ifxintglobaldefs global \fi meaning \expandafter\meaning
         \csname XINT_expr_macrofunc_\XINT_newfunc_tmpa\endcsname}%
  \fi
}%
\def\XINT_expr_newfunction #1#2#3%
{%
    \XINT_global
    \def#1##1##2##3%
      {\expandafter ##1\expandafter ##2%
       \romannumeral0\XINT:NEhook:macrofunc
       #3{\csname XINT_expr_macrofunc_#2\endcsname##3}\relax
      }%
}%
\let\XINT:NEhook:macrofunc\empty
```

11.29.5 Mysterious stuff

There was an \xintNewExpr already in 1.07 from May 2013, which was modified in September 2013 to work with the # macro parameter character, and then refactored into a more powerful version in June 2014 for 1.1 release of 2014/10/28.

It is always too soon to try to comment and explain. In brief, this attempts to hack into the *purely numeric* \xintexpr parsers to transform them into *symbolic* parsers, allowing to do once and for all the parsing job and inherit a gigantic nested macro. Originally only f-expandable nesting. The initial motivation was that the \csname encapsulation impacted the string pool memory. Later

this work proved to be the basis to provide support for implementing user-defined functions and it is now its main purpose.

Deep refactorings happened at 1.3 and 1.4.

At 1.3 the crucial idea of the «hook» macros was introduced, reducing considerably the preparatory work done by \xintNewExpr.

At 1.4 further considerable simplifications happened, and it is possible that the author currently does at long last understand the code!

The 1.3 code had serious complications with trying to identify would-be «list» arguments, distinguishing them from «single» arguments (things like parsing #2+[[#1..[#3]..#4][#5:#6]]*#7 and convert it to a single nested f-exandable macro...)

The conversion at 1.4 is both more powerful and simpler, due in part to the new storage model which from \csname encapsulated comma separated values up to 1.3f became simply a braced list of braced values, and also crucially due to the possibilities opened up by usage of \expanded primitive.

```
\catcode`~ 12
\def\XINT:NE:hastilde#1~#2#3\relax{\unless\if !#21\fi}%
\def\XINT:NE:hashash#1{%
\def\XINT:NE:hashash##1#1##2##3\relax{\unless\if !##21\fi}%
}\expandafter\XINT:NE:hashash\string#%
\def\XINT:NE:unpack #1{%
\def\XINT:NE:unpack ##1%
{%
    \if0\XINT:NE:hastilde ##1~!\relax
        \XINT:NE:hashash  ##1#1!\relax 0\else
        \expandafter\XINT:NE:unpack:p\fi
    \xint_stop_atfirstofone{##1}%
}}\expandafter\XINT:NE:unpack\string#%
\def\XINT:NE:unpack:p#1#2%
    {{~romannumeral0~expandafter~xint_stop_atfirstofone~expanded{#2}}}%
\def\XINT:NE:f:one:from:one #1{%
\def\XINT:NE:f:one:from:one ##1%
{%
    \if0\XINT:NE:hastilde ##1~!\relax
        \XINT:NE:hashash  ##1#1!\relax 0\else
        \xint_dothis\XINT:NE:f:one:from:one_a\fi
    \xint_orthat\XINT:NE:f:one:from:one_b
    ##1&&A%
}}\expandafter\XINT:NE:f:one:from:one\string#%
\def\XINT:NE:f:one:from:one_a\romannumeral`&&@#1#2&&A%
{%
    \expandafter{\detokenize{\expandafter#1}#2}%
}%
\def\XINT:NE:f:one:from:one_b#1{%
\def\XINT:NE:f:one:from:one_b\romannumeral`&&@##1##2&&A%
{%
    \expandafter{\romannumeral`&&@%
        \if0\XINT:NE:hastilde ##2~!\relax
            \XINT:NE:hashash  ##2#1!\relax 0\else
        \expandafter\string\fi
    ##1{##2}}%
}}\expandafter\XINT:NE:f:one:from:one_b\string#%
\def\XINT:NE:f:one:from:one:direct #1#2{\XINT:NE:f:one:from:one:direct_a #2&&A{#1}}%
```

```
\def\XINT:NE:f:one:from:one:direct_a #1#2&&A#3%
{%
    \if ###1\xint_dothis {\detokenize{#3}}\fi
    \if  ~#1\xint_dothis {\detokenize{#3}}\fi
    \xint_orthat {#3}{#1#2}%
}%
\def\XINT:NE:f:one:from:two #1{%
\def\XINT:NE:f:one:from:two ##1%
{%
    \if0\XINT:NE:hastilde ##1~!\relax
        \XINT:NE:hashash  ##1#1!\relax 0\else
        \xint_dothis\XINT:NE:f:one:from:two_a\fi
    \xint_orthat\XINT:NE:f:one:from:two_b ##1&&A%
}}\expandafter\XINT:NE:f:one:from:two\string#%
\def\XINT:NE:f:one:from:two_a\romannumeral`&&@#1#2&&A%
{%
    \expandafter{\detokenize{\expandafter#1\expanded}{#2}}%
}%
\def\XINT:NE:f:one:from:two_b#1{%
\def\XINT:NE:f:one:from:two_b\romannumeral`&&@##1##2##3&&A%
{%
    \expandafter{\romannumeral`&&@%
        \if0\XINT:NE:hastilde ##2##3~!\relax
            \XINT:NE:hashash  ##2##3#1!\relax 0\else
        \expandafter\string\fi
    ##1{##2}{##3}}%
}}\expandafter\XINT:NE:f:one:from:two_b\string#%
\def\XINT:NE:f:one:from:two:direct #1#2#3{\XINT:NE:two_fork #2&&A#3&&A#1{#2}{#3}}%
\def\XINT:NE:two_fork #1#2&&A#3#4&&A{\XINT:NE:two_fork_nn#1#3}%
\def\XINT:NE:two_fork_nn #1#2%
{%
    \if #1##\xint_dothis\string\fi
    \if  #1~\xint_dothis\string\fi
    \if #2##\xint_dothis\string\fi
    \if  #2~\xint_dothis\string\fi
    \xint_orthat{}%
}%
\def\XINT:NE:f:one:and:opt:direct#1{%
\def\XINT:NE:f:one:and:opt:direct##1!%
{%
    \if0\XINT:NE:hastilde ##1~!\relax
        \XINT:NE:hashash  ##1#1!\relax 0\else
        \xint_dothis\XINT:NE:f:one:and:opt_a\fi
    \xint_orthat\XINT:NE:f:one:and:opt_b ##1&&A%
}}\expandafter\XINT:NE:f:one:and:opt:direct\string#%
\def\XINT:NE:f:one:and:opt_a #1#2&&A#3#4%
{%
    \detokenize{\romannumeral-`0\expandafter#1\expanded{#2}$XINT_expr_exclam#3#4}%$
}%
\def\XINT:NE:f:one:and:opt_b\XINT:expr:f:one:and:opt #1#2#3&&A#4#5%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
```

```
                    \expandafter\xint_secondoftwo\fi
    {\XINT:NE:f:one:from:one:direct#4}%
    {\expandafter\XINT:NE:f:onewithopttoone\expandafter#5%
        \expanded{{\XINT:NE:f:one:from:one:direct\xintNum{#2}}}}%
    {#1}%
}%
\def\XINT:NE:f:onewithopttoone#1#2#3{\XINT:NE:two_fork #2&&A#3&&A#1[#2]{#3}}%
\def\XINT:NE:f:tacitzeroifone:direct#1{%
\def\XINT:NE:f:tacitzeroifone:direct##1!%
{%
    \if0\XINT:NE:hastilde ##1~!\relax
        \XINT:NE:hashash  ##1#1!\relax 0\else
        \xint_dothis\XINT:NE:f:one:and:opt_a\fi
    \xint_orthat\XINT:NE:f:tacitzeroifone_b ##1&&A%
}}\expandafter\XINT:NE:f:tacitzeroifone:direct\string#%
\def\XINT:NE:f:tacitzeroifone_b\XINT:expr:f:tacitzeroifone #1#2#3&&A#4#5%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {\XINT:NE:f:one:from:two:direct#4{0}}%
    {\expandafter\XINT:NE:f:one:from:two:direct\expandafter#5%
        \expanded{{\XINT:NE:f:one:from:one:direct\xintNum{#2}}}}%
    {#1}%
}%
\def\XINT:NE:f:iitacitzeroifone:direct#1{%
\def\XINT:NE:f:iitacitzeroifone:direct##1!%
{%
    \if0\XINT:NE:hastilde ##1~!\relax
        \XINT:NE:hashash  ##1#1!\relax 0\else
        \xint_dothis\XINT:NE:f:iitacitzeroifone_a\fi
    \xint_orthat\XINT:NE:f:iitacitzeroifone_b ##1&&A%
}}\expandafter\XINT:NE:f:iitacitzeroifone:direct\string#%
\def\XINT:NE:f:iitacitzeroifone_a #1#2&&A#3%
{%
    \detokenize{\romannumeral`$XINT_expr_null\expandafter#1\expanded{#2}$XINT_expr_exclam#3}%
}%
\def\XINT:NE:f:iitacitzeroifone_b\XINT:expr:f:iitacitzeroifone #1#2#3&&A#4%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {\XINT:NE:f:one:from:two:direct#4{0}}%
    {\XINT:NE:f:one:from:two:direct#4{#2}}%
    {#1}%
}%
\def\XINT:NE:x:one:from:two #1#2#3{\XINT:NE:x:one:from:two_fork #2&&A#3&&A#1{#2}{#3}}%
\def\XINT:NE:x:one:from:two_fork #1{%
\def\XINT:NE:x:one:from:two_fork ##1##2&&A##3##4&&A%
{%
    \if0\XINT:NE:hastilde ##1##3~!\relax\XINT:NE:hashash ##1##3#1!\relax 0%
    \else
        \expandafter\XINT:NE:x:one:from:two:p
    \fi
```

```
}}\expandafter\XINT:NE:x:one:from:two_fork\string#%
\def\XINT:NE:x:one:from:two:p #1#2#3%
   {~expanded{\detokenize{\expandafter#1}~expanded{{#2}{#3}}}}%
\def\XINT:NE:x:listsel #1{%
\def\XINT:NE:x:listsel ##1##2&%
{%
    \if0\expandafter\XINT:NE:hastilde\detokenize{##2}~!\relax
        \expandafter\XINT:NE:hashash\detokenize{##2}#1!\relax 0%
    \else
        \expandafter\XINT:NE:x:listsel:p
    \fi
    ##1##2&%
}}\expandafter\XINT:NE:x:listsel\string#%
\def\XINT:NE:x:listsel:p #1#2_#3&(#4%
{%
    \detokenize{\expanded\XINT:expr:ListSel{{#3}{#4}}}%
}%
\def\XINT:expr:ListSel{\expandafter\XINT:expr:ListSel_i\expanded}%
\def\XINT:expr:ListSel_i #1#2{{\XINT_ListSel_top #2_#1&({#2}}}%
\def\XINT:NE:f:reverse #1{%
\def\XINT:NE:f:reverse ##1^%
{%
    \if0\expandafter\XINT:NE:hastilde\detokenize\expandafter{\xint_gobble_i##1}~!\relax
        \expandafter\XINT:NE:hashash\detokenize{##1}#1!\relax 0%
    \else
        \expandafter\XINT:NE:f:reverse:p
    \fi
    ##1^%
}}\expandafter\XINT:NE:f:reverse\string#%
\def\XINT:NE:f:reverse:p #1^#2\xint_bye
{%
    \expandafter\XINT:NE:f:reverse:p_i\expandafter{\xint_gobble_i#1}%
}%
\def\XINT:NE:f:reverse:p_i #1%
{%
    \detokenize{\romannumeral0\XINT:expr:f:reverse{{#1}}}%
}%
\def\XINT:expr:f:reverse{\expandafter\XINT:expr:f:reverse_i\expanded}%
\def\XINT:expr:f:reverse_i #1%
{%
    \XINT_expr_reverse #1^^#1\xint:\xint:\xint:\xint:
                           \xint:\xint:\xint:\xint:\xint_bye
}%
\def\XINT:NE:f:from:delim:u #1{%
\def\XINT:NE:f:from:delim:u ##1##2^%
{%
    \if0\expandafter\XINT:NE:hastilde\detokenize{##2}~!\relax
        \expandafter\XINT:NE:hashash\detokenize{##2}#1!\relax 0%
        \expandafter##1%
    \else
        \xint_afterfi{\XINT:NE:f:from:delim:u:p##1\empty}%
    \fi
```

```
    ##2^%
}}\expandafter\XINT:NE:f:from:delim:u\string#%
\def\XINT:NE:f:from:delim:u:p #1#2^%
    {\detokenize{\expandafter#1}~expanded{#2}$XINT_expr_caret}%$
\def\XINT:NE:f:noeval:from:braced:u #1{%
\def\XINT:NE:f:noeval:from:braced:u ##1##2^%
{%
    \if0\XINT:NE:hastilde ##2~!\relax\XINT:NE:hashash ##2#1!\relax 0%
    \else
        \expandafter\XINT:NE:f:noeval:from:braced:u:p
    \fi
    ##1{##2}%
}}\expandafter\XINT:NE:f:noeval:from:braced:u\string#%
\def\XINT:NE:f:noeval:from:braced:u:p #1#2%
    {\detokenize{\romannumeral`$XINT_expr_null\expandafter#1}~expanded{{#2}}}%
\catcode`- 11
\def\XINT:NE:exec_? #1#2%
{%
    \XINT:NE:exec_?_b #2&&A#1{#2}%
}%
\def\XINT:NE:exec_?_b #1{%
\def\XINT:NE:exec_?_b ##1&&A%
{%
    \if0\XINT:NE:hastilde ##1~!\relax
        \XINT:NE:hashash  ##1#1!\relax 0%
    \xint_dothis\XINT:NE:exec_?:x\fi
    \xint_orthat\XINT:NE:exec_?:p
}}\expandafter\XINT:NE:exec_?_b\string#%
\def\XINT:NE:exec_?:x #1#2#3%
{%
    \expandafter\XINT_expr_check-_after?\expandafter#1%
    \romannumeral`&&@\expandafter\XINT_expr_getnext\romannumeral0\xintiiifnotzero#3%
}%
\def\XINT:NE:exec_?:p #1#2#3#4#5%
{%
    \csname XINT_expr_func_*If\expandafter\endcsname
    \romannumeral`&&@#2\XINTfstop.{#3},[#4],[#5])%
}%
\expandafter\def\csname XINT_expr_func_*If\endcsname #1#2#3%
{%
    #1#2{~expanded{~xintiiifNotZero#3}}%
}%
\def\XINT:NE:exec_?? #1#2#3%
{%
    \XINT:NE:exec_??_b #2&&A#1{#2}%
}%
\def\XINT:NE:exec_??_b #1{%
\def\XINT:NE:exec_??_b ##1&&A%
{%
    \if0\XINT:NE:hastilde ##1~!\relax
        \XINT:NE:hashash  ##1#1!\relax 0%
    \xint_dothis\XINT:NE:exec_??:x\fi
```

```
    \xint_orthat\XINT:NE:exec_??:p
}}\expandafter\XINT:NE:exec_??_b\string#%
\def\XINT:NE:exec_??:x #1#2#3%
{%
    \expandafter\XINT_expr_check-_after?\expandafter#1%
    \romannumeral`&&@\expandafter\XINT_expr_getnext\romannumeral0\xintiiifsgn#3%
}%
\def\XINT:NE:exec_??:p #1#2#3#4#5#6%
{%
    \csname XINT_expr_func_*IfSgn\expandafter\endcsname
    \romannumeral`&&@#2\XINTfstop.{#3},[#4],[#5],[#6])%
}%
\expandafter\def\csname XINT_expr_func_*IfSgn\endcsname #1#2#3%
{%
    #1#2{~expanded{~xintiiifSgn#3}}%
}%
\catcode`- 12
\def\XINT:NE:branch #1%
{%
    \if0\XINT:NE:hastilde #1~!\relax 0\else
        \xint_dothis\XINT:NE:branch_a\fi
    \xint_orthat\XINT:NE:branch_b #1&&A%
}%
\def\XINT:NE:branch_a\romannumeral`&&@#1#2&&A%
{%
    \expandafter{\detokenize{\expandafter#1\expanded}{#2}}%
}%
\def\XINT:NE:branch_b#1{%
\def\XINT:NE:branch_b\romannumeral`&&@##1##2##3&&A%
{%
    \expandafter{\romannumeral`&&@%
        \if0\XINT:NE:hastilde ##2~!\relax
            \XINT:NE:hashash  ##2#1!\relax 0\else
        \expandafter\string\fi
    ##1{##2}##3}%
}}\expandafter\XINT:NE:branch_b\string#%
\def\XINT:NE:seqx#1{%
\def\XINT:NE:seqx\XINT_allexpr_seqx##1##2%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##2}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##2}#1!\relax 0%
    \else
      \expandafter\XINT:NE:seqx:p
    \fi \XINT_allexpr_seqx{##1}{##2}%
}}\expandafter\XINT:NE:seqx\string#%
\def\XINT:NE:seqx:p\XINT_allexpr_seqx #1#2#3#4%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded {%
    {%
        \detokenize
        {%
```

```
            \expanded\bgroup
            \expanded
            {\unexpanded{\XINT_expr_seq:_b{#1#4\relax $XINT_expr_exclam #3}}%
              #2$XINT_expr_caret}%
        }%
    }%
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT:NE:opx#1{%
\def\XINT:NE:opx\XINT_allexpr_opx ##1##2##3##4%##5##6##7##8%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##4}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##4}#1!\relax 0%
    \else
      \expandafter\XINT:NE:opx:p
    \fi \XINT_allexpr_opx ##1{##2}{##3}{##4}% en fait ##2 = \xint_c_, ##3 = \relax
}}\expandafter\XINT:NE:opx\string#%
\def\XINT:NE:opx:p\XINT_allexpr_opx #1#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded {%
    {%
        \detokenize
        {%
            \expanded\bgroup
            \expanded{\unexpanded{\XINT_expr_iter:_b
               {#1\expandafter\XINT_allexpr_opx_ifnotomitted
                  \romannumeral0#1#6\relax#7@\relax $XINT_expr_exclam #5}}%
                      #4$XINT_expr_caret$XINT_expr_tilde{{#8}}}%$
        }%
    }%
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT:NE:iter{\expandafter\XINT:NE:itery\expandafter}%
\def\XINT:NE:itery#1{%
\def\XINT:NE:itery\XINT_expr_itery##1##2%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##1##2}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##1##2}#1!\relax 0%
    \else
      \expandafter\XINT:NE:itery:p
    \fi \XINT_expr_itery{##1}{##2}%
}}\expandafter\XINT:NE:itery\string#%
\def\XINT:NE:itery:p\XINT_expr_itery #1#2#3#4#5%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded {%
    {%
        \detokenize
        {%
          \expanded\bgroup
          \expanded{\unexpanded{\XINT_expr_iter:_b {#5#4\relax $XINT_expr_exclam #3}}%
```

```
                    #1$XINT_expr_caret$XINT_expr_tilde{#2}}%$
        }%
    }%
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT:NE:rseq{\expandafter\XINT:NE:rseqy\expandafter}%
\def\XINT:NE:rseqy#1{%
\def\XINT:NE:rseqy\XINT_expr_rseqy##1##2%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##1##2}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##1##2}#1!\relax 0%
    \else
      \expandafter\XINT:NE:rseqy:p
    \fi \XINT_expr_rseqy{##1}{##2}%
}}\expandafter\XINT:NE:rseqy\string#%
\def\XINT:NE:rseqy:p\XINT_expr_rseqy #1#2#3#4#5%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded {%
    {%
        \detokenize
        {%
          \expanded\bgroup
          \expanded{#2\unexpanded{\XINT_expr_rseq:_b {#5#4\relax $XINT_expr_exclam #3}}%
                      #1$XINT_expr_caret$XINT_expr_tilde{#2}}%$
         }%
    }%
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT:NE:iterr{\expandafter\XINT:NE:iterry\expandafter}%
\def\XINT:NE:iterry#1{%
\def\XINT:NE:iterry\XINT_expr_iterry##1##2%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##1##2}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##1##2}#1!\relax 0%
    \else
      \expandafter\XINT:NE:iterry:p
    \fi \XINT_expr_iterry{##1}{##2}%
}}\expandafter\XINT:NE:iterry\string#%
\def\XINT:NE:iterry:p\XINT_expr_iterry #1#2#3#4#5%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded {%
    {%
        \detokenize
        {%
          \expanded\bgroup
          \expanded{\unexpanded{\XINT_expr_iterr:_b {#5#4\relax $XINT_expr_exclam #3}}%
                    #1$XINT_expr_caret$XINT_expr_tilde #20$XINT_expr_qmark}%
        }%
    }%
    \expandafter}\romannumeral`&&@\XINT_expr_getop
```

```
}%
\def\XINT:NE:rrseq{\expandafter\XINT:NE:rrseqy\expandafter}%
\def\XINT:NE:rrseqy#1{%
\def\XINT:NE:rrseqy\XINT_expr_rrseqy##1##2%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##1##2}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##1##2}#1!\relax 0%
    \else
      \expandafter\XINT:NE:rrseqy:p
    \fi \XINT_expr_rrseqy{##1}{##2}%
}}\expandafter\XINT:NE:rrseqy\string#%
\def\XINT:NE:rrseqy:p\XINT_expr_rrseqy #1#2#3#4#5#6%
{%
    \expandafter\XINT_expr_put_op_first
    \expanded {%
    {%
        \detokenize
        {%
           \expanded\bgroup
           \expanded{#2\unexpanded{\XINT_expr_rrseq:_b {#6#5\relax $XINT_expr_exclam #4}}%
                       #1$XINT_expr_caret$XINT_expr_tilde #30$XINT_expr_qmark}%
         }%
    }%
    \expandafter}\romannumeral`&&@\XINT_expr_getop
}%
\def\XINT:NE:x:toblist#1{%
\def\XINT:NE:x:toblist\XINT:expr:toblistwith##1##2%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##2}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##2}#1!\relax 0%
    \else
      \expandafter\XINT:NE:x:toblist:p
    \fi \XINT:expr:toblistwith{##1}{##2}%
}}\expandafter\XINT:NE:x:toblist\string#%
\def\XINT:NE:x:toblist:p\XINT:expr:toblistwith #1#2{{\XINTfstop.{#2}}}%
\def\XINT:NE:x:flatten#1{%
\def\XINT:NE:x:flatten\XINT:expr:flatten##1%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##1}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##1}#1!\relax 0%
    \else
      \expandafter\XINT:NE:x:flatten:p
    \fi \XINT:expr:flatten{##1}%
}}\expandafter\XINT:NE:x:flatten\string#%
\def\XINT:NE:x:flatten:p\XINT:expr:flatten #1%
{%
    {{%
     \detokenize
     {%
         \expandafter\XINT:expr:flatten_checkempty
         \detokenize\expandafter{\expanded{#1}}$XINT_expr_caret%$
     }%
```

```
    }}%
}%
\def\XINT:NE:x:zip#1{%
\def\XINT:NE:x:zip\XINT:expr:zip##1%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##1}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##1}#1!\relax 0%
    \else
      \expandafter\XINT:NE:x:zip:p
    \fi \XINT:expr:zip{##1}%
}}\expandafter\XINT:NE:x:zip\string#%
\def\XINT:NE:x:zip:p\XINT:expr:zip #1%
{%
    \expandafter{%
     \detokenize
     {%
        \expanded\expandafter\XINT_zip_A\expanded{#1}\xint_bye\xint_bye
     }%
    }%
}%
\def\XINT:NE:x:mapwithin#1{%
\def\XINT:NE:x:mapwithin\XINT:expr:mapwithin ##1##2%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##2}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##2}#1!\relax 0%
    \else
      \expandafter\XINT:NE:x:mapwithin:p
    \fi \XINT:expr:mapwithin {##1}{##2}%
}}\expandafter\XINT:NE:x:mapwithin\string#%
\def\XINT:NE:x:mapwithin:p \XINT:expr:mapwithin  #1#2%
{%
    {{%
     \detokenize
     {%
%%      \expanded
%%      {%
         \expandafter\XINT:expr:mapwithin_checkempty
         \expanded{\noexpand#1$XINT_expr_exclam\expandafter}%$
         \detokenize\expandafter{\expanded{#2}}$XINT_expr_caret%$
%%      }%
     }%
    }}%
}%
\def\XINT:NE:x:ndmapx#1{%
\def\XINT:NE:x:ndmapx\XINT_allexpr_ndmapx_a ##1##2^%
{%
    \if 0\expandafter\XINT:NE:hastilde\detokenize{##2}~!\relax
         \expandafter\XINT:NE:hashash \detokenize{##2}#1!\relax 0%
    \else
      \expandafter\XINT:NE:x:ndmapx:p
    \fi \XINT_allexpr_ndmapx_a ##1##2^%
}}\expandafter\XINT:NE:x:ndmapx\string#%
```

```
\def\XINT:NE:x:ndmapx:p #1#2#3^\relax
{%
    \detokenize
    {%
      \expanded{%
      \expandafter#1\expandafter#2\expanded{#3}$XINT_expr_caret\relax %$
                }%
    }%
}%
```

Attention here that user function names may contain digits, so we don't use a \detokenize or ~ approach.

This syntax means that a function defined by \xintdeffunc never expands when used in another definition, so it can implement recursive definitions.

\XINT:NE:userefunc et al. added at 1.3e.

I added at \xintdefefunc, \xintdefiiefunc, \xintdeffloatefunc at 1.3e to on the contrary expand if possible (i.e. if used only with numeric arguments) in another definition.

The \XINTusefunc uses \expanded. Its ancestor \xintExpandArgs (xinttools 1.3) had some more primitive f-expansion technique.

```
\def\XINTusenoargfunc #1%
{%
    0\csname #1\endcsname
}%
\def\XINT:NE:usernoargfunc\csname #1\endcsname
{%
    ~romannumeral~XINTusenoargfunc{#1}%
}%
\def\XINTusefunc #1%
{%
    0\csname #1\expandafter\endcsname\expanded
}%
\def\XINT:NE:usefunc #1#2#3%
{%
    ~romannumeral~XINTusefunc{#1}{#3}\iffalse{{\fi}}%
}%
\def\XINTuseufunc #1%
{%
    \expanded\expandafter\XINT:expr:mapwithin\csname #1\expandafter\endcsname\expanded
}%
\def\XINT:NE:useufunc #1#2#3%
{%
    {{~expanded~XINTuseufunc{#1}{#3}}}%
}%
\def\XINT:NE:userfunc #1{%
\def\XINT:NE:userfunc ##1##2##3%
{%
    \if0\expandafter\XINT:NE:hastilde\detokenize{##3}~!\relax
        \expandafter\XINT:NE:hashash\detokenize{##3}#1!\relax 0%
      \expandafter\XINT:NE:userfunc_x
    \else
      \expandafter\XINT:NE:usefunc
    \fi {##1}{##2}{##3}%
```

```
}}\expandafter\XINT:NE:userfunc\string#%
\def\XINT:NE:userfunc_x #1#2#3{#2#3\iffalse{{\fi}}}%
\def\XINT:NE:userufunc #1{%
\def\XINT:NE:userufunc ##1##2##3%
{%
    \if0\expandafter\XINT:NE:hastilde\detokenize{##3}~!\relax
        \expandafter\XINT:NE:hashash\detokenize{##3}#1!\relax 0%
      \expandafter\XINT:NE:userufunc_x
    \else
      \expandafter\XINT:NE:useufunc
    \fi {##1}{##2}{##3}%
}}\expandafter\XINT:NE:userufunc\string#%
\def\XINT:NE:userufunc_x #1{\XINT:expr:mapwithin}%
\def\XINT:NE:macrofunc #1#2%
   {\expandafter\XINT:NE:macrofunc:a\string#1#2\empty&}%
\def\XINT:NE:macrofunc:a#1\csname #2\endcsname#3&%
   {{~XINTusemacrofunc{#1}{#2}{#3}}}%
\def\XINTusemacrofunc #1#2#3%
{%
    \romannumeral0\expandafter\xint_stop_atfirstofone
    \romannumeral0#1\csname #2\endcsname#3\relax
}%
```

11.29.6 \XINT_expr_redefinemacros

Completely refactored at 1.3.
  Again refactored at 1.4. The availability of \expanded allows more powerful mechanisms and more importantly I better thought out the root problems caused by the handling of list operations in this context and this helped simplify considerably the code.

```
\catcode`- 11
\def\XINT_expr_redefinemacros {%
  \let\XINT:NEhook:unpack           \XINT:NE:unpack
  \let\XINT:NEhook:f:one:from:one       \XINT:NE:f:one:from:one
  \let\XINT:NEhook:f:one:from:one:direct \XINT:NE:f:one:from:one:direct
  \let\XINT:NEhook:f:one:from:two       \XINT:NE:f:one:from:two
  \let\XINT:NEhook:f:one:from:two:direct \XINT:NE:f:one:from:two:direct
  \let\XINT:NEhook:x:one:from:two       \XINT:NE:x:one:from:two
  \let\XINT:NEhook:f:one:and:opt:direct     \XINT:NE:f:one:and:opt:direct
  \let\XINT:NEhook:f:tacitzeroifone:direct  \XINT:NE:f:tacitzeroifone:direct
  \let\XINT:NEhook:f:iitacitzeroifone:direct \XINT:NE:f:iitacitzeroifone:direct
  \let\XINT:NEhook:x:listsel            \XINT:NE:x:listsel
  \let\XINT:NEhook:f:reverse            \XINT:NE:f:reverse
  \let\XINT:NEhook:f:from:delim:u       \XINT:NE:f:from:delim:u
  \let\XINT:NEhook:f:noeval:from:braced:u\XINT:NE:f:noeval:from:braced:u
  \let\XINT:NEhook:branch           \XINT:NE:branch
  \let\XINT:NEhook:seqx             \XINT:NE:seqx
  \let\XINT:NEhook:opx              \XINT:NE:opx
  \let\XINT:NEhook:rseq             \XINT:NE:rseq
  \let\XINT:NEhook:iter             \XINT:NE:iter
  \let\XINT:NEhook:rrseq            \XINT:NE:rrseq
  \let\XINT:NEhook:iterr            \XINT:NE:iterr
  \let\XINT:NEhook:x:toblist        \XINT:NE:x:toblist
```

```
\let\XINT:NEhook:x:flatten        \XINT:NE:x:flatten
\let\XINT:NEhook:x:zip            \XINT:NE:x:zip
\let\XINT:NEhook:x:mapwithin      \XINT:NE:x:mapwithin
\let\XINT:NEhook:x:ndmapx         \XINT:NE:x:ndmapx
\let\XINT:NEhook:userfunc         \XINT:NE:userfunc
\let\XINT:NEhook:userufunc        \XINT:NE:userufunc
\let\XINT:NEhook:usernoargfunc    \XINT:NE:usernoargfunc
\let\XINT:NEhook:macrofunc        \XINT:NE:macrofunc
\def\XINTinRandomFloatSdigits{~XINTinRandomFloatSdigits }%
\def\XINTinRandomFloatSixteen{~XINTinRandomFloatSixteen }%
\def\xintiiRandRange{~xintiiRandRange }%
\def\xintiiRandRangeAtoB{~xintiiRandRangeAtoB }%
\def\xintRandBit{~xintRandBit }%
\let\XINT_expr_exec_?  \XINT:NE:exec_?
\let\XINT_expr_exec_?? \XINT:NE:exec_??
\def\XINT_expr_op_?  {\XINT_expr_op__?{\XINT_expr_op_-xii\XINT_expr_oparen}}%
\def\XINT_flexpr_op_?{\XINT_expr_op__?{\XINT_flexpr_op_-xii\XINT_flexpr_oparen}}%
\def\XINT_iiexpr_op_?{\XINT_expr_op__?{\XINT_iiexpr_op_-xii\XINT_iiexpr_oparen}}%
}%
\catcode`- 12
```

11.29.7 \xintNewExpr, \xintNewIExpr, \xintNewFloatExpr, \xintNewIIExpr

1.2c modifications to accomodate \XINT_expr_deffunc_newexpr etc..
1.2f adds token \XINT_newexpr_clean to be able to have a different \XINT_newfunc_clean.
As \XINT_NewExpr always execute \XINT_expr_redefineprints since 1.3e whether with \xintNewExpr or \XINT_NewFunc, it has been moved from argument to hardcoded in replacement text.
NO MORE \XINT_expr_redefineprints at 1.4 ! This allows better support for \xinteval, \xintthe-expr as sub-entities inside an \xintNewExpr. And the «cleaning» will remove the new \XINTfstop (detokenized from \meaning output), to maintain backwards compatibility with former behaviour that created macros expand to explicit digits and not an encapsulated result.
The #2#3 in clean stands for \noexpand\XINTfstop (where the actual scantoken-ized input uses $ originally with catcode letter as the escape character).

```
\def\xintNewExpr     {\XINT_NewExpr\xint_firstofone\xintexpr     \XINT_newexpr_clean}%
\def\xintNewFloatExpr{\XINT_NewExpr\xint_firstofone\xintfloatexpr\XINT_newexpr_clean}%
\def\xintNewIExpr    {\XINT_NewExpr\xint_firstofone\xintiexpr    \XINT_newexpr_clean}%
\def\xintNewIIExpr   {\XINT_NewExpr\xint_firstofone\xintiiexpr   \XINT_newexpr_clean}%
\def\xintNewBoolExpr {\XINT_NewExpr\xint_firstofone\xintboolexpr \XINT_newexpr_clean}%
\def\XINT_newexpr_clean #1>#2#3{\noexpand\expanded\noexpand\xintNEprinthook}%
\def\xintNEprinthook#1.#2{\expanded{\unexpanded{#1.}{#2}}}%
```

1.2c for \xintdeffunc, \xintdefiifunc, \xintdeffloatfunc.
At 1.3, NewFunc does not use anymore a comma delimited pattern for the arguments to the macro being defined.
At 1.4 we use \xintthebareeval, whose meaning now does not mean unlock from csname but firstofone to remove a level of braces This is involved in functioning of expr:userfunc and expr:userefunc

```
\def\XINT_NewFunc     {\XINT_NewExpr\xint_gobble_i\xintthebareeval\XINT_newfunc_clean}%
\def\XINT_NewFloatFunc{\XINT_NewExpr\xint_gobble_i\xintthebarefloateval\XINT_newfunc_clean}%
\def\XINT_NewIIFunc   {\XINT_NewExpr\xint_gobble_i\xintthebareiieval\XINT_newfunc_clean}%
\def\XINT_newfunc_clean #1>{}%
```

1.2c adds optional logging. For this needed to pass to _NewExpr_a the macro name as parameter. Up to and including 1.2c the definition was global. Starting with 1.2d it is done locally. Modified at 1.3c so that \XINT_NewFunc et al. do not execute the \xintexprSafeCatcodes, as it is now already done earlier by \xintdeffunc.

```
\def\XINT_NewExpr #1#2#3#4#5[#6]%
{%
 \begingroup
    \ifcase #6\relax
        \toks0 {\endgroup\XINT_global\def#4}%
    \or \toks0 {\endgroup\XINT_global\def#4##1}%
    \or \toks0 {\endgroup\XINT_global\def#4##1##2}%
    \or \toks0 {\endgroup\XINT_global\def#4##1##2##3}%
    \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4}%
    \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5}%
    \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5##6}%
    \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5##6##7}%
    \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5##6##7##8}%
    \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5##6##7##8##9}%
    \fi
    #1\xintexprSafeCatcodes
    \XINT_expr_redefinemacros
    \XINT_NewExpr_a #1#2#3#4%
}%
```

1.2d's \xintNewExpr makes a local definition. In earlier releases, the definition was global. \the\toks0 inserts the \endgroup, but this will happen after \XINT_tmpa has already been expanded...

The %1 is \xint_firstofone for \xintNewExpr, \xint_gobble_i for \xintdeffunc.

Attention that at 1.4, there might be entire sub-xintexpressions embedded in detokenized form. They are re-tokenized and the main thing is that the parser should not mis-interpret catcode 11 characters as starting variable names. As some macros use : in their names, the retokenization must be done with : having catcode 11. To not break embedded non-evaluated sub-expressions, the \XINT_expr_getop was extended to intercept the : (alternative would have been to never inject any macro with : in its name... too late now). On the other hand the ! is not used in the macro names potentially kept as is non expanded by the \xintNewExpr/\xintdeffunc process; it can thus be retokenized with catcode 12. But the «hooks» of seq(), iter(), etc... if deciding they can't evaluate immediately will inject a full sub-expression (possibly arbitrarily complicated) and append to it for its delayed expansion a catcode 11 ! character (as well as possibly catcode 3 ~ and ? and catcode 11 caret ^ and even catcode 7 &). The macros \XINT_expr_tilde etc... below serve for this injection (there are *two* successive \scantokens using different catcode regimes and these macros remain detokenized during the first pass!) and as consequence the final meaning may have characters such as ! or and special catcodes depending on where they are located. It may thus not be possible to (easily) retokenize the meaning as printed in the log file if \xintverbosetrue was issued.

If a defined function is used in another expression it would thus break things if its meaning was included pre-expanded ; a mechanism exists which keeps only the name of the macro associated to the function (this name may contain digits by the way), when the macro can not be immediately fully expanded. Thus its meaning (with its possibly funny catcodes) is not exposed. And this gives opportunity to pre-expand its arguments before actually expanding the macro.

```
\catcode`~ 3 \catcode`? 3
\def\XINT_expr_tilde{~}\def\XINT_expr_qmark{?}% catcode 3
```

```
\def\XINT_expr_caret{^}\def\XINT_expr_exclam{!}% catcode 11
\def\XINT_expr_tab{&}% catcode 7
\def\XINT_expr_null{&&@}%
\catcode`~ 13 \catcode`@ 14 \catcode`\% 6 \catcode`# 12 \catcode`$ 11 @ $
\def\XINT_NewExpr_a %1%2%3%4%5@
{@
    \def\XINT_tmpa %%1%%2%%3%%4%%5%%6%%7%%8%%9{%5}@
    \def~{$noexpand$}@
    \catcode`: 11 \catcode`_ 11 \catcode`\@ 11
    \catcode`# 12 \catcode`~ 13 \escapechar 126
    \endlinechar -1 \everyeof {\noexpand }@
    \edef\XINT_tmpb
    {\scantokens\expandafter{\romannumeral`&&@\expandafter
     %2\XINT_tmpa{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}\relax}@
    }@
    \escapechar 92 \catcode`# 6 \catcode`$ 0 @ $
    \edef\XINT_tmpa %%1%%2%%3%%4%%5%%6%%7%%8%%9@
      {\scantokens\expandafter{\expandafter%3\meaning\XINT_tmpb}}@
    \the\toks0\expandafter
      {\XINT_tmpa{%%1}{%%2}{%%3}{%%4}{%%5}{%%6}{%%7}{%%8}{%%9}}@
    %1{\ifxintverbose
        \xintMessage{xintexpr}{Info}@
                    {\string%4\space now with @
                     \ifxintglobaldefs global \fi meaning \meaning%4}@
       \fi}@
}@
\catcode`% 14
\XINT_setcatcodes % clean up to avoid surprises if something changes
```

11.29.8 \ifxintexprsafecatcodes, \xintexprSafeCatcodes, \xintexprRestoreCatcodes

1.3c (2018/06/17).

Added \ifxintexprsafecatcodes to allow nesting

```
\newif\ifxintexprsafecatcodes
\let\xintexprRestoreCatcodes\empty
\def\xintexprSafeCatcodes
{%
  \unless\ifxintexprsafecatcodes
    \edef\xintexprRestoreCatcodes  {%
        \catcode59=\the\catcode59   % ;
        \catcode34=\the\catcode34   % "
        \catcode63=\the\catcode63   % ?
        \catcode124=\the\catcode124 % |
        \catcode38=\the\catcode38   % &
        \catcode33=\the\catcode33   % !
        \catcode93=\the\catcode93   % ]
        \catcode91=\the\catcode91   % [
        \catcode94=\the\catcode94   % ^
        \catcode95=\the\catcode95   % _
        \catcode47=\the\catcode47   % /
        \catcode41=\the\catcode41   % )
        \catcode40=\the\catcode40   % (
```

```
        \catcode42=\the\catcode42   % *
        \catcode43=\the\catcode43   % +
        \catcode62=\the\catcode62   % >
        \catcode60=\the\catcode60   % <
        \catcode58=\the\catcode58   % :
        \catcode46=\the\catcode46   % .
        \catcode45=\the\catcode45   % -
        \catcode44=\the\catcode44   % ,
        \catcode61=\the\catcode61   % =
        \catcode96=\the\catcode96   % `
        \catcode32=\the\catcode32\relax % space
        \noexpand\xintexprsafecatcodesfalse
    }%
  \fi
  \xintexprsafecatcodestrue
        \catcode59=12  % ;
        \catcode34=12  % "
        \catcode63=12  % ?
        \catcode124=12 % |
        \catcode38=4   % &
        \catcode33=12  % !
        \catcode93=12  % ]
        \catcode91=12  % [
        \catcode94=7   % ^
        \catcode95=8   % _
        \catcode47=12  % /
        \catcode41=12  % )
        \catcode40=12  % (
        \catcode42=12  % *
        \catcode43=12  % +
        \catcode62=12  % >
        \catcode60=12  % <
        \catcode58=12  % :
        \catcode46=12  % .
        \catcode45=12  % -
        \catcode44=12  % ,
        \catcode61=12  % =
        \catcode96=12  % `
        \catcode32=10  % space
}%
\let\XINT_tmpa\undefined \let\XINT_tmpb\undefined \let\XINT_tmpc\undefined
\let\XINT_tmpd\undefined \let\XINT_tmpe\undefined
\ifdefined\RequirePackage\expandafter\xint_firstoftwo\else\expandafter\xint_secondoftwo\fi
{\RequirePackage{xinttrig}%
\RequirePackage{xintlog}}%
{\input xinttrig.sty
\input xintlog.sty
}%
\XINT_restorecatcodes_endinput%
```

12 Package xinttrig implementation

Contents

Comments under reconstruction.

The original was done in January 15 and 16, 2019. It provided asin() and acos() based on a Newton algorithm approach. Then during March 25-31 I revisited the code, adding more inverse trigonometrical functions (with a modified algorithm, quintically convergent), extending the precision range (so that the package reacts to the \xintDigits value at time of load, or reload), and replaced high level range reduction by some optimized lower level coding.

This led me next to improve upon the innards of \xintdeffunc and \xintNewExpr, and to add to xintexpr the \xintdefefunc macro (see user documentation).

Finally on April 5, 2019 I pushed further the idea of the algorithm for the arcsine function. The cost is at least the one of a combined sin()/cos() evaluation, surely this is not best approach for low precision, but I like the principle and its suitability to go into hundreds of digits if desired.

Almost all of the code remains written at high level, and in particular it is not easily feasible from this interface to execute computations with guard digits. Expect the last one or two digits to be systematically off.

Also, small floating-point inputs are handled quite sub-optimally both for the direct and inverse functions; substantial gains are possible. I added the ilog10() function too late to consider using it here with the high level interface.

12.1 Catcodes, ε-TEX and reload detection

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \catcode94=7    % ^
  \def\z{\endgroup}%
  \def\empty{}\def\space{ }\newlinechar10
  \expandafter\let\expandafter\w\csname ver@xintexpr.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info:^^J%
                 \space\space\space\space#2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xinttrig}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\w\relax % xintexpr.sty not yet loaded.
        \y{xinttrig}%
          {Loading should be via \ifx\x\empty\string\usepackage{xintexpr.sty}
                           \else\string\input\space xintexpr.sty \fi
           rather, aborting}%
         \aftergroup\endinput
    \fi
  \fi
\z%
\catcode`_ 11 \XINT_setcatcodes \catcode`? 12
```

12.2 Library identification

```
\ifcsname xintlibver@trig\endcsname
  \expandafter\xint_firstoftwo
\else
  \expandafter\xint_secondoftwo
\fi
{\immediate\write-1{Reloading xinttrig library using Digits=\xinttheDigits.}}%
{\expandafter\gdef\csname xintlibver@trig\endcsname{2021/03/29 v1.4d}%
\XINT_providespackage
\ProvidesPackage{xinttrig}%
[2021/03/29 v1.4d Trigonometrical functions for xintexpr (JFB)]%
}%
```

12.3 Ensure used letters are dummy letters

```
\xintFor* #1 in {iDTVtuwxyzX}\do{\xintensuredummy{#1}}%
```

12.4 \xintreloadxinttrig

```
\def\xintreloadxinttrig
   {\edef\XINT_restorecatcodes_now{\XINT_restorecatcodes}%
    \XINT_setcatcodes\catcode`? 12
    \input xinttrig.sty
    \XINT_restorecatcodes_now}%
```

12.5 Auxiliary variables (only temporarily needed, but left free to re-use)

These variables don't have really private names but this does not matter because only their actual values will be stored in the functions defined next. Nevertheless they are not unassigned, and are left free to use as is.

12.5.1 twoPi, threePiover2, Pi, Piover2

We take them with 60 digits and force conversion to \xintDigits setting via "0 + " syntax.

```
\xintdeffloatvar twoPi        := 0 +
    6.28318530717958647692528676655900576839433879875021164194989;%
\xintdeffloatvar threePiover2 := 0 +
    4.71238898038468985769396507491925432629575409906265873146242;%
\xintdeffloatvar Pi           := 0 +
    3.14159265358979323846264338327950288419716939937510582097494;%
\xintdeffloatvar Piover2      := 0 +
    1.57079632679489661923132169163975144209858469968755291048747;%
```

12.5.2 oneDegree, oneRadian

```
\xintdeffloatvar oneDegree := 0 +
    0.01745329251994329576923690768488612713442871888541725456097 19;% Pi/180
\xintdeffloatvar oneRadian := 0 +
    57.29577951308232087679815481410517033240547246656432154916 02;%   180/Pi
```

12.5.3 Inverse factorial coefficients: invfact2, ..., invfact44

Pre-compute 1/n! for n = 2, ..., 44
  We have to be careful that 1/i! in a float expression first evaluates i! as a floating point number then computes the inverse. Even if i! was computed exactly before being float-rounded, this process would not necessarily lead to the correct rounding of the exact fraction 1/i!.
  We could use \xintexpr1/i!\relax encapsulation but then the actual rounding is delayed to the time when functions are used... this is bad.
  We need to get now the correct rounding of the exact 1/i!.
  1.4 update: use \xintfloatexpr with optional argument for the rounding rather than «0+x» method. And there is no need now to hide within braces the inner semi-colon.

```
\xintdeffloatvar invfact\xintListWithSep{, invfact}{\xintSeq{2}{44}}%
     := \xintfloatexpr [\XINTdigits]  % force float rounding after exact evaluations
           \xintexpr rseq(1/2; @/i, i=3..44)\relax % no need to hide this inner ;
        \relax;%
```

12.6 The sine and cosine series

12.6.1 sin_aux(), cos_aux()

Should I rather use successive divisions by (2n+1)(2n), or rather multiplication by their precomputed inverses, in a modified Horner scheme ? The \ifnum tests are executed at time of definition.

Criteria for truncated series using $\pi/4$, actually 0.79.

Small values of the variable X are very badly handled here because a much shorter truncation of the sine series should be used.

```
\xintdeffloatfunc sin_aux(X) := 1 - X(invfact3 - X(invfact5
\ifnum\XINTdigits>4
                           - X(invfact7
\ifnum\XINTdigits>6
                           - X(invfact9
\ifnum\XINTdigits>8
                           - X(invfact11
\ifnum\XINTdigits>10
                           - X(invfact13
\ifnum\XINTdigits>13
                           - X(invfact15
\ifnum\XINTdigits>15
                           - X(invfact17
\ifnum\XINTdigits>18
                           - X(invfact19
\ifnum\XINTdigits>21
                           - X(invfact21
\ifnum\XINTdigits>24
                           - X(invfact23
\ifnum\XINTdigits>27
                           - X(invfact25
\ifnum\XINTdigits>30
                           - X(invfact27
\ifnum\XINTdigits>33
                           - X(invfact29
\ifnum\XINTdigits>36
                           - X(invfact31
\ifnum\XINTdigits>39
                           - X(invfact33
\ifnum\XINTdigits>43
                           - X(invfact35
\ifnum\XINTdigits>46
                           - X(invfact37
\ifnum\XINTdigits>49
                           - X(invfact39
\ifnum\XINTdigits>53
                           - X(invfact41
\ifnum\XINTdigits>59
                           - X(invfact43
    )\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi));%
```

Criteria on basis of $\pi/4$, we actually used 0.79 to choose the transition values and this makes them a bit less favourable at 24, 26, 29...and some more probably. Again this is very bad for small X.

```
\xintdeffloatfunc cos_aux(X) := 1 - X(invfact2 - X(invfact4
\ifnum\XINTdigits>3
                            - X(invfact6
\ifnum\XINTdigits>5
                            - X(invfact8
\ifnum\XINTdigits>7
                            - X(invfact10
\ifnum\XINTdigits>9
                            - X(invfact12
\ifnum\XINTdigits>12
                            - X(invfact14
\ifnum\XINTdigits>14
                            - X(invfact16
\ifnum\XINTdigits>17
                            - X(invfact18
\ifnum\XINTdigits>20
                            - X(invfact20
\ifnum\XINTdigits>23
                            - X(invfact22
\ifnum\XINTdigits>25
                            - X(invfact24
\ifnum\XINTdigits>28
                            - X(invfact26
\ifnum\XINTdigits>32
                            - X(invfact28
\ifnum\XINTdigits>35
                            - X(invfact30
\ifnum\XINTdigits>38
                            - X(invfact32
\ifnum\XINTdigits>41
                            - X(invfact34
\ifnum\XINTdigits>44
                            - X(invfact36
\ifnum\XINTdigits>48
                            - X(invfact38
\ifnum\XINTdigits>51
                            - X(invfact40
\ifnum\XINTdigits>55
                            - X(invfact42
\ifnum\XINTdigits>58
                            - X(invfact44
    )\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi)\fi));%
```

12.6.2 Make sin_aux() and cos_aux() known to \xintexpr

We need them shortly for the asin() in an \xintexpr variant. We short-circuit the high level interface as it will not be needed to add some \xintFloat wrapper.

```
\expandafter\let\csname XINT_expr_func_sin_aux\expandafter\endcsname
                \csname XINT_flexpr_func_sin_aux\endcsname
\expandafter\let\csname XINT_expr_func_cos_aux\expandafter\endcsname
                \csname XINT_flexpr_func_cos_aux\endcsname
```

12.6.3 sin_(), cos_()

Use this only between -pi/4 and pi/4

```
\xintdeffloatfunc sin_(x) := x * sin_aux(sqr(x));%
```

Use this only between -pi/4 and pi/4

```
\xintdeffloatfunc cos_(x) := cos_aux(sqr(x));%
```

12.7 Range reduction for sine and cosine using degrees

Notice that even when handling radians it is much better to convert to degrees and then do range reduction there, because this can be done in the fixed point sense. I lost 1h puzzled about some mismatch of my results with those of Maple (at 16 digits) near $-\pi$. Turns out that Maple probably adds π in the floating point sense causing catastrophic loss of digits when one is near $-\pi$. On the other hand my sin(x) function will first convert to degrees then add 180 without any loss of floating point precision, even for a result near zero, then convert back to radians and use the sine series.

12.7.1 Core level macro \XINT_mod_ccclx_i

input: \the\numexpr\XINT_mod_ccclx_i k.N. (delimited by dots)
output: (N times 10^k) modulo 360. (with a final dot)
Attention N must be non-negative (I could make it accept negative but the fact that numexpr / is not periodical in numerator adds overhead).
360 divides 9000 hence 10^{k} is 280 for k at least 3 and the additive group generated by it modulo 360 is the set of multiples of 40.

```
\def\XINT_mod_ccclx_i #1.% input <k>.<N>. k is a non-negative exponent
{%
    \expandafter\XINT_mod_ccclx_e\the\numexpr
    \expandafter\XINT_mod_ccclx_j\the\numexpr1\ifcase#1 \or0\or00\else000\fi.%
}%
\def\XINT_mod_ccclx_j 1#1.#2.% #2=N is a non-negative mantissa
{%
    (\XINT_mod_ccclx_ja {++}#2#1\XINT_mod_ccclx_jb 0000000\relax
}%                                  1     2345678
\def\XINT_mod_ccclx_ja #1#2#3#4#5#6#7#8#9%
{%
    #9+#8+#7+#6+#5+#4+#3+#2\xint_firstoftwo{+\XINT_mod_ccclx_ja{+#9+#8+#7}}{#1}%
}%
\def\XINT_mod_ccclx_jb #1\xint_firstoftwo#2#3{#1+0)*280\XINT_mod_ccclx_jc #1#3}%
```

Attention that \XINT_cclcx_e wants non negative input because \numexpr division is not periodical ...

```
\def\XINT_mod_ccclx_jc  +#1+#2+#3#4\relax{+80*(#3+#2+#1)+#3#2#1.}%
\def\XINT_mod_ccclx_e#1.{\expandafter\XINT_mod_ccclx_z\the\numexpr(#1+180)/360-1.#1.}%
\def\XINT_mod_ccclx_z#1.#2.{#2-360*#1.}%
```

12.7.2 sind_(), cosd_(), and support macros \xintSind, \xintCosd

sind_() coded directly at macro level with a macro \xintSind (ATTENTION! it requires a positive argument) which will suitably use \XINT_flexpr_func_sin_ defined from \xintdeffloatfunc

```
\def\XINT_flexpr_func_sind_ #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one{\romannumeral`&&@\xintSind#3}}%
}%
```

Must be f-expandable for nesting macros from \xintNewExpr
ATTENTION ONLY FOR POSITIVE ARGUMENTS

```
\def\XINT_expr_unlock{\expandafter\xint_firstofone\romannumeral`&&@}%
\def\xintSind#1{\romannumeral`&&@\expandafter\xintsind
                \romannumeral0\XINTinfloatS[\XINTdigits]{#1}}%
\def\xintsind #1[#2#3]%
{%
    \xint_UDsignfork
      #2\XINT_sind
       -\XINT_sind_int
    \krof#2#3.#1..%<< attention extra dot
}%
\def\XINT_sind #1.#2.% NOT TO BE USED WITH VANISHING (OR NEGATIVE) #2.
{%
    \expandafter\XINT_sind_a
    \romannumeral0\xinttrunc{\XINTdigits}{#2[#1]}%
}%
\def\XINT_sind_a{\expandafter\XINT_sind_i\the\numexpr\XINT_mod_ccclx_i0.}%
\def\XINT_sind_int
{%
    \expandafter\XINT_sind_i\the\numexpr\expandafter\XINT_mod_ccclx_i
}%
\def\XINT_sind_i #1.% range reduction inside [0, 360[
{%
    \ifcase\numexpr#1/90\relax
       \expandafter\XINT_sind_A
    \or\expandafter\XINT_sind_B\the\numexpr-90+%
    \or\expandafter\XINT_sind_C\the\numexpr-180+%
    \or\expandafter\XINT_sind_D\the\numexpr-270+%
    \else\expandafter\XINT_sind_E\the\numexpr-360+%
    \fi#1.%
}%
```

#2 will be empty in the "integer branch". Notice that a single dot "." is valid as input to the xintfrac macros. During developing phase I did many silly mistakes due to wanting to use too low-level interface, e.g. I would use something like #2[-\XINTdigits] with #2 the fractional digits, but there maybe some leading zero and then xintfrac.sty will think the whole thing is zero due to the requirements of my own core format A[N]....

Multiplication is done exactly but anyway currently float multiplication goes via exact multiplication after rounding arguments ; as here integer part has at most three digits, doing exact multiplication will prove not only more accurate but probably faster.

```
\def\XINT_sind_A#1{%
\def\XINT_sind_A##1.##2.%
{%
    \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_sin_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{##1.##2}#1}}%
}%
}\expandafter
\XINT_sind_A\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
\def\XINT_sind_B#1{\xint_UDsignfork#1\XINT_sind_B_n-\XINT_sind_B_p\krof #1}%
\def\XINT_tmpa#1{%
\def\XINT_sind_B_n-##1.##2.%
{%
    \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_cos_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{\xintSub{##1[0]}{.##2}}#1}}%
}%
\def\XINT_sind_B_p##1.##2.%
{%
    \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_cos_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{##1.##2}#1}}%
}%
}\expandafter
\XINT_tmpa\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
\def\XINT_sind_C#1{\xint_UDsignfork#1\XINT_sind_C_n-\XINT_sind_C_p\krof #1}%
\def\XINT_tmpa#1{%
\def\XINT_sind_C_n-##1.##2.%
{%
    \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_sin_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{\xintSub{##1[0]}{.##2}}#1}}%
}%
\def\XINT_sind_C_p##1.##2.%
{%
    \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_sin_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{##1.##2}#1}}%
}%
}\expandafter
\XINT_tmpa\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
\def\XINT_sind_D#1{\xint_UDsignfork#1\XINT_sind_D_n-\XINT_sind_D_p\krof #1}%
\def\XINT_tmpa#1{%
\def\XINT_sind_D_n-##1.##2.%
{%
    \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_cos_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{\xintSub{##1[0]}{.##2}}#1}}%
}%
\def\XINT_sind_D_p##1.##2.%
{%
    \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_cos_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{##1.##2}#1}}%
}%
}\expandafter
\XINT_tmpa\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
\def\XINT_sind_E#1{%
\def\XINT_sind_E-##1.##2.%
```

```
{%
    \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_sin_\expandafter
         {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{\xintSub{##1[0]}{.##2}}#1}}%
}%
}\expandafter
\XINT_sind_E\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
```

The cosd_ auxiliary function

```
\def\XINT_flexpr_func_cosd_ #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one{\romannumeral`&&@\xintCosd#3}}%
}%
```

ATTENTION ONLY FOR POSITIVE ARGUMENTS

```
\def\xintCosd#1{\romannumeral`&&@\expandafter\xintcosd
                \romannumeral0\XINTinfloatS[\XINTdigits]{#1}}%
\def\xintcosd #1[#2#3]%
{%
    \xint_UDsignfork
      #2\XINT_cosd
       -\XINT_cosd_int
    \krof#2#3.#1..%<< attention extra dot
}%
\def\XINT_cosd #1.#2.% NOT TO BE USED WITH VANISHING (OR NEGATIVE) #2.
{%
    \expandafter\XINT_cosd_a
    \romannumeral0\xinttrunc{\XINTdigits}{#2[#1]}%
}%
\def\XINT_cosd_a{\expandafter\XINT_cosd_i\the\numexpr\XINT_mod_ccclx_i0.}%
\def\XINT_cosd_int
{%
    \expandafter\XINT_cosd_i\the\numexpr\expandafter\XINT_mod_ccclx_i
}%
\def\XINT_cosd_i #1.%
{%
    \ifcase\numexpr#1/90\relax
       \expandafter\XINT_cosd_A
    \or\expandafter\XINT_cosd_B\the\numexpr-90+%
    \or\expandafter\XINT_cosd_C\the\numexpr-180+%
    \or\expandafter\XINT_cosd_D\the\numexpr-270+%
    \else\expandafter\XINT_cosd_E\the\numexpr-360+%
    \fi#1.%
}%
```

#2 will be empty in the "integer" branch, but attention in general branch to handling of negative integer part after the subtraction of 90, 180, 270, or 360, and avoid abusing A[N] notation which yes speeds up xintfrac parsing but has its pitfalls.

```
\def\XINT_cosd_A#1{%
\def\XINT_cosd_A##1.##2.%
{%
```

```
    \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_cos_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{##1.##2}#1}}%
}%
}\expandafter
\XINT_cosd_A\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
\def\XINT_cosd_B#1{\xint_UDsignfork#1\XINT_cosd_B_n-\XINT_cosd_B_p\krof #1}%
\def\XINT_tmpa#1{%
\def\XINT_cosd_B_n-##1.##2.%
{%
    \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_sin_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{\xintSub{##1[0]}{.##2}}#1}}%
}%
\def\XINT_cosd_B_p##1.##2.%
{%
    \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_sin_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{##1.##2}#1}}%
}%
}\expandafter
\XINT_tmpa\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
\def\XINT_cosd_C#1{\xint_UDsignfork#1\XINT_cosd_C_n-\XINT_cosd_C_p\krof #1}%
\def\XINT_tmpa#1{%
\def\XINT_cosd_C_n-##1.##2.%
{%
    \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_cos_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{\xintSub{##1[0]}{.##2}}#1}}%
}%
\def\XINT_cosd_C_p##1.##2.%
{%
    \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_cos_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{##1.##2}#1}}%
}%
}\expandafter
\XINT_tmpa\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
\def\XINT_cosd_D#1{\xint_UDsignfork#1\XINT_cosd_D_n-\XINT_cosd_D_p\krof #1}%
\def\XINT_tmpa#1{%
\def\XINT_cosd_D_n-##1.##2.%
{%
    \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_sin_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{\xintSub{##1[0]}{.##2}}#1}}%
}%
\def\XINT_cosd_D_p##1.##2.%
{%
    \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_sin_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{##1.##2}#1}}%
}%
}\expandafter
\XINT_tmpa\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
\def\XINT_cosd_E#1{%
\def\XINT_cosd_E-##1.##2.%
{%
    \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_cos_\expandafter
        {\romannumeral0\XINTinfloat[\XINTdigits]{\xintMul{\xintSub{##1[0]}{.##2}}#1}}%
```

```
}%
}\expandafter
\XINT_cosd_E\expandafter{\romannumeral`&&@\xintthebarefloateval oneDegree\relax}%
```

12.8 sind(), cosd()

```
\xintdeffloatfunc sind(x) := (x)??
                             {(x>=-45)?
                               {sin_(x*oneDegree)}
                               {-sind_(-x)}
                             }
                             {0}
                             {(x<=45)?
                               {sin_(x*oneDegree)}
                               {sind_(x)}
                             }
                             ;%
\xintdeffloatfunc cosd(x) := (x)??
                             {(x>=-45)?
                               {cos_(x*oneDegree)}
                               {cosd_(-x)}
                             }
                             {1}
                             {(x<=45)?
                               {cos_(x*oneDegree)}
                               {cosd_(x)}
                             }
                             ;%
```

12.9 sin(), cos()

For some reason I did not define sin() and cos() in January 2019 ??

```
\xintdeffloatfunc sin(x):= (abs(x)<0.79)?
                            {sin_(x)}
                            {(x)??
                              {-sind_(-x*oneRadian)}
                              {0}
                              {sind_(x*oneRadian)}
                            }
                            ;%
\xintdeffloatfunc cos(x):= (abs(x)<0.79)?
                             {cos_(x)}
                             {cosd_(abs(x*oneRadian))}
                             ;%
```

12.10 sinc()

Should I also consider adding (1-cos(x))/(x^2/2) ? it is sinc^2(x/2) but avoids a square.

```
\xintdeffloatfunc sinc(x):= (abs(x)<0.79) ?
                             {sin_aux(sqr(x))}
                             {sind_(abs(x)*oneRadian)/abs(x)}
                             ;%
```

12.11 tan(), tand(), cot(), cotd()

The 0 in cot(x) is a dummy place holder, 1/0 would raise an error at time of definition...

```
\xintdeffloatfunc tand(x):= sind(x)/cosd(x);%
\xintdeffloatfunc cotd(x):= cosd(x)/sind(x);%
\xintdeffloatfunc tan(x) := (x)??
                             {(x>-0.79)?
                               {sin(x)/cos(x)}
                               {-cotd(90+x*oneRadian)
                               }
                             }
                             {0}
                             {(x<0.79)?
                               {sin(x)/cos(x)}
                               {cotd(90-x*oneRadian)}
                             }
                             ;%
\xintdeffloatfunc cot(x) := (abs(x)<0.79)?
                              {cos(x)/sin(x)}
                              {(x)??
                                {-tand(90+x*oneRadian)}
                                {0}
                                {tand(90-x*oneRadian)}
                              };%
```

12.12 sec(), secd(), csc(), cscd()

```
\xintdeffloatfunc sec(x) := inv(cos(x));%
\xintdeffloatfunc csc(x) := inv(sin(x));%
\xintdeffloatfunc secd(x):= inv(cosd(x));%
\xintdeffloatfunc cscd(x):= inv(sind(x));%
```

12.13 Core routine for inverse trigonometry

Compute asin(x)

The approach I shall first describe (which is only a first step towards our final approach) converges quintically but requires an initial square root computation. For atan(x), we do not have to do any such square root extraction. See code next.

The algorithm (for this first approach): we have 0 <= t < 0.72, let t1 = t*(1+t^2/6). We also have u = sqrt(1 - t^2). We seek a = Arcsin t with t = sin(a).

Then t1 < Arcsin t and the difference (we don't know it!) δ_1 is < 0.02. We compute D = t*cos(t1)-u*sin(t1). This computation is done "exactly" via the \xintexpr encapsulation. In other terms we use doubled precision. Anyhow, currently (1.3e) the Float macros of xintfrac.sty for multiplication do go via such exact multiplication when the mantissas have the expected sizes. So we can't gain but only lose due to catastrophic subtraction in using float operations here.

Thus D is sin(a-t1) = sin(δ_1). And δ_1 = Arcsin D, but D is small! We then use again two terms of the Arcsin series and define t2 = t1 + D * (1 + D^2/6). Let δ_2 = a - t2. Then δ_2 is of the order of the neglected term 3*(δ_1)^5/40.

©copyright J.F. Burnol, March 30, 2019. This surely has a name.

The algorithm is quintically convergent. One can do the same to go from exp to log. Basically the idea is that we can improve the Newton Method for any function f for which knowing target value of f implies one also knows target value of its derivative. In fact I obtained the quintic algorithm by

combining the Newton formula with the one from using f(x)/f'(a) and not f(x)/f'(x) in the update to cancel the two quadratic errors.

One iteration (t2) gives about 9 digits, two iterations (t3) 49 digits ! And if we want hepta-convergence we only need to use one more term of the Arcsin series in the update of the t_n... really this is very nice.

And actually (t2) already gives 30 digits of floating point precision for input t<0.1. Let's confirm this:

```
> Digits := 60:
> t0 := 0.1; t1 := t0*(1+t0^2/6); u0 := sqrt(1-t0^2); D1 :=t0*cos(t1)-u0*sin(t1);
                                        t0 := 0.1
          t1 := 0.100166666666666666666666666666666666666666666666666666666667
          u0 := 0.994987437106619954734479821001206005178126563676806079117605
                                                                            -6
          D1 := 0.754494893129607272232433362202120141404083765295901166 8 10
> t2 := t1 + D1*(1+D1^2/6);
          t2 := 0.100167421161559796345523179452674980956388959919827205633117
> a := arcsin(0.1);
          a := 0.100167421161559796345523179452693318568675972229629541391024
> t2/a;
             0.999999999999999999999999999998169303744234802813068128141 73
```

Each iteration costs a computation of one cos and one sine done at the full final precision. This is stupid because we should compute at an evolving precision, but anyhow this is not our problem anymore as our final algorithm is not a loop but it does exactly one iteration for all inputs. As examplified above it remains true that we could improve its speed for small inputs by using shorter auxiliary series (see below).

In January I used a loop via an iter() construct, with some subs() to avoid repeating computations. This can only be done in an \xintNewFunction. Here is how it looked after some optimization for the stopping criteria, after replacing generic Newton algorithm by a specific quintic one for arcsine:

```
\begingroup
\edef\x{\endgroup
\noexpand\xintNewFunction{asin_l}[2]{
   iter(##1*(1+sqr(##1)/6);
% FIXME : réfléchir au critère d'arrêt.
%
% Je n'utilise pas abs(D) pour un micro-gain est-ce que le risque en vaut la
% chandelle ? (avec abs(D) on pourrait utiliser la fonction avec un #1 négatif)
%
% Am I sure rounding errors could not cause neverending loop?
% Such things should be done with increased precision and rounded at end.
              subs((D<\ifcase\numexpr2+\XINTdigits-5*(\XINTdigits/5)\relax
                      3.68\or2.32\or1.47\or0.923\or0.582\fi
                      e-\the\numexpr\XINTdigits/5\relax)
                   ?{break(@+D*(1+sqr(D)/6))}{@+D*(1+sqr(D)/6)},
                   D=\noexpand\xintexpr
                       subs(##1*cos_aux(X) - ##2*@*sin_aux(X), X=sqr(@))
                     \relax
                   ),
         i=1++)dummy iteration index, not used but needed by iter()
}}\x
```

I don't have time to explain the final algorithm below and how the transition values were chosen or why (the series below is enough up to 59 digits of precision). It does only one iteration, in

all cases. Using it for arcsine requires a preliminary square root extraction, but for arctangent one arranges things to avoid having to compute a square root.

©copyright J.F. Burnol, April 5, 2019. This surely has a name.

Certainly I can do similar things to compute logarithms.

```
\xintdeffloatfunc asin_aux(X) := 1
\ifnum\XINTdigits>3 % actually 4 would achieve 1ulp in place of <0.5ulp
                          + X(1/6
\ifnum\XINTdigits>9
                          + X(3/40
\ifnum\XINTdigits>16
                          + X(5/112
\ifnum\XINTdigits>25
                          + X(35/1152
\ifnum\XINTdigits>35
                          + X(63/2816
\ifnum\XINTdigits>46
                          + X(231/13312
                          )\fi)\fi)\fi)\fi)\fi)\fi;%
\xintdeffloatfunc asin_o(D, T) := T + D*asin_aux(sqr(D));%
\xintdeffloatfunc asin_n(V, T, t, u) :=% V is square of T
                    asin_o (\xintexpr t*cos_aux(V) - u*T*sin_aux(V)\relax, T);%
\xintdeffloatfunc asin_m(T, t, u) := asin_n(sqr(T), T, t, u);%
\xintdeffloatfunc asin_l(t, u)    := asin_m(t*asin_aux(sqr(t)), t, u);%
```

12.14 asin(), asind()

Only non-negative arguments t and u for asin_a(t,u), and asind_a(t,u).

At 1.4 usage of sqrt_() which has only one argument, whereas currently sqrt() admits a second optional argument hence sub-optimality here if we use sqrt(), especially since 1.4 handles more fully such functions with optional argument in \xintdeffunc.

Actually thinking of making sqrt() a one argument only function and sqrt_() will be the one with two arguments. But I worked hard on the \xintdeffunc hooks, thus some reticence, because why then not do that for all others?

```
\xintdeffloatfunc asin_a(t, u) := (t<u)?
                                  {asin_l(t, u)}
                                  {Piover2 - asin_l(u, t)}
                                  ;%
\xintdeffloatfunc asind_a(t, u):= (t<u)?
                                  {asin_l(t, u) * oneRadian}
                                  {90 - asin_l(u, t) * oneRadian}
                                  ;%
\xintdeffloatfunc asin(t) := (t)??
                              {-asin_a(-t, sqrt_(1-sqr(t)))}
                              {0}
                              {asin_a(t, sqrt_(1-sqr(t)))}
                              ;%
\xintdeffloatfunc asind(t) := (t)??
                               {-asind_a(-t, sqrt_(1-sqr(t)))}
                               {0}
                               {asind_a(t, sqrt_(1-sqr(t)))}
                               ;%
```

12.15 acos(), acosd()

```
\xintdeffloatfunc acos(t) := Piover2 - asin(t);%
\xintdeffloatfunc acosd(t):= 90 - asind(t);%
```

12.16 atan(), atand()

This involves no square root!

TeX hackers note 1:

The subs(, x = ..) mechanism has no utility in a function definition, there is no parallel mechanism at the underlying macros, so in fact the substituted things will remain unevaluated if they involve indeterminates, so this is exactly like not trying to make things more efficient at all.

Currently, the only way is thus to employ auxiliary functions like is done next. Contrarily to TeX macros, we must define the functions one after the other in the correct order, so the auxiliaries come first.

TeX hackers note 2:

At 1.4, the way to inject lazy conditionals in function definitions has changed. Prior one used if(,,) and ifsgn(,,,) which was counter-intuitive because in pure numeric context they evaluate all branches. Now one must use ? and ?? which the are the lazy conditionals from the numeric context.

radians

```
\xintdeffloatfunc atan_b(t, w, z):= 0.5 * (w< 0)?
                                          {Pi - asin_a(2z * t, -w*z)}
                                          {asin_a(2z * t, w*z)}
                                          ;%
\xintdeffloatfunc atan_a(t, T) := atan_b(t, 1-T, inv(1+T));%
\xintdeffloatfunc atan(t):= (t)??
                             {-atan_a(-t, sqr(t))}
                             {0}
                             {atan_a(t, sqr(t))}
                             ;%
```

degrees

```
\xintdeffloatfunc atand_b(t, w, z) := 0.5 * (w< 0)?
                                            {180 - asind_a(2z * t, -w*z)}
                                            {asind_a(2z * t, w*z)}
                                            ;%
\xintdeffloatfunc atand_a(t, T) := atand_b(t, 1-T, inv(1+T));%
\xintdeffloatfunc atand(t) := (t)??
                               {-atand_a(-t, sqr(t))}
                               {0}
                               {atand_a(t, sqr(t))}
                               ;%
```

12.17 Arg(), atan2(), Argd(), atan2d(), pArg(), pArgd()

Arg(x,y) function from $-\pi$ (excluded) to $+\pi$ (included)

```
\xintdeffloatfunc Arg(x, y):= (y>x)?
                               {(y>-x)?
                                 {Piover2 - atan(x/y)}
```

```
                                  {(y<0)?
                                    {-Pi + atan(y/x)}
                                    {Pi + atan(y/x)}
                                  }
                                }
                                {(y>-x)?
                                  {atan(y/x)}
                                  {-Piover2 + atan(x/-y)}
                                }
                                ;%
```

atan2(y,x) = Arg(x,y) ... (some people have atan2 with arguments reversed but the convention here seems the most often encountered)

```
\xintdeffloatfunc atan2(y,x) := Arg(x, y);%
```

Argd(x,y) function from -180 (excluded) to +180 (included)

```
\xintdeffloatfunc Argd(x, y):= (y>x)?
                                {(y>-x)?
                                  {90 - atand(x/y)}
                                  {(y<0)?
                                    {-180 + atand(y/x)}
                                    {180 + atand(y/x)}
                                  }
                                }
                                {(y>-x)?
                                  {atand(y/x)}
                                  {-90 + atand(x/-y)}
                                }
                                ;%
```

atan2d(y,x) = Argd(x,y)

```
\xintdeffloatfunc atan2d(y,x) := Argd(x, y);%
```

pArg(x,y) function from 0 (included) to 2π (excluded) I hesitated between pArg, Argpos, and Argplus. Opting for pArg in the end.

```
\xintdeffloatfunc pArg(x, y):= (y>x)?
                                {(y>-x)?
                                  {Piover2 - atan(x/y)}
                                  {Pi + atan(y/x)}
                                }
                                {(y>-x)?
                                  {(y<0)?
                                    {twoPi + atan(y/x)}
                                    {atan(y/x)}
                                  }
                                  {threePiover2 + atan(x/-y)}
                                }
                                ;%
```

pArgd(x,y) function from 0 (included) to 360 (excluded)

```
\xintdeffloatfunc pArgd(x, y):=(y>x)?
                                {(y>-x)?
                                  {90 - atan(x/y)*oneRadian}
                                  {180 + atan(y/x)*oneRadian}
                                }
                                {(y>-x)?
                                  {(y<0)?
                                    {360 + atan(y/x)*oneRadian}
                                    {atan(y/x)*oneRadian}
                                  }
                                  {270 + atan(x/-y)*oneRadian}
                                }
                                ;%
```

12.18 Synonyms: tg(), cotg()

These are my childhood notations and I am attached to them. In radians only. We skip some overhead here by using a \let at core level.

```
\expandafter\let\csname XINT_flexpr_func_tg\expandafter\endcsname
                \csname XINT_flexpr_func_tan\endcsname
\expandafter\let\csname XINT_flexpr_func_cotg\expandafter\endcsname
                \csname XINT_flexpr_func_cot\endcsname
```

12.19 Let the functions be known to the \xintexpr parser

See xint.pdf for some explanations (as well as code comments in xintexpr.sty). In fact it is this context which led to my addition at 1.3e of \xintdefefunc to the \xintexpr syntax.

```
\xintFor #1 in {sin, cos, tan, sec, csc, cot,
                asin, acos, atan}\do
{%
    \xintdeffunc #1(x) := \xintfloatexpr #1(sfloat(x))\relax;%
    \xintdeffunc #1d(x):= \xintfloatexpr #1d(sfloat(x))\relax;%
}%
\xintFor #1 in {Arg, pArg, atan2}\do
{%
    \xintdeffunc #1(x, y) := \xintfloatexpr #1(sfloat(x), sfloat(y))\relax;%
    \xintdeffunc #1d(x, y):= \xintfloatexpr #1d(sfloat(x), sfloat(y))\relax;%
}%
\xintdeffunc tg(x)  := \xintfloatexpr tg(sfloat(x))\relax;%
\xintdeffunc cotg(x):= \xintfloatexpr cotg(sfloat(x))\relax;%
\xintdeffunc sinc(x):= \xintfloatexpr sinc(sfloat(x))\relax;%
```

Restore used dummy variables to their status prior to the package reloading. On first loading this is not needed naturally, because this is done immediately at end of xintexpr.sty.

```
\xintFor* #1 in {iDTVtuwxyzX}\do{\xintrestorevariable{#1}}%
```

13 Package xintlog implementation

Contents

I almost included extended precision implementation for 1.3e but was a bit short on time; besides I hesitated between using poormanlog at starting point or not. For up to 50 digits, it would help reduce considerably the needed series for the logarithm. For more digits I should rather apply my copyrighted method of the arcsine (it must be in literature).

13.1 Catcodes, ε-TEX and reload detection

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \catcode94=7    % ^
  \def\z{\endgroup}%
  \def\empty{}\def\space{ }\newlinechar10
  \expandafter\let\expandafter\w\csname ver@xintexpr.sty\endcsname
  \expandafter\let\expandafter\x\csname ver@xintlog.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info:^^J%
                 \space\space\space\space#2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintlog}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\w\relax % xintexpr.sty not yet loaded.
        \y{xintlog}%
          {Loading should be via \ifx\x\empty\string\usepackage{xintexpr.sty}
                                 \else\string\input\space xintexpr.sty \fi
           rather, aborting}%
        \aftergroup\endinput
    \else
```

```
      \ifx\x\relax % first loading (initiated from xintexpr.sty)
      \else
        \ifx\x\empty % LaTeX first loading, \ProvidesPackage not yet seen
        \else
          \y{xintlog}{Already loaded, aborting}%
          \aftergroup\endinput
        \fi
      \fi
    \fi
  \fi
\z%
```

Attention to catcode regime when loading below poormanlog. It (v0.04) uses ^ with its normal catcode but \XINT_setcatcodes would set it to letter.

This file can only be loaded from xintexpr.sty and it restores catcodes near its end. To play it safe and be hopefully immune to whatever is done in poormanlog or in xinttrig.sty which is loaded before, we will switch to standard catcode regime here.

As I learned the hard way (I never use my user macros), at the worst moment when wrapping up the final things for 1.3e release, \xintexprSafeCatcodes MUST be followed by some \xintexprRestoreCatcodes quickly, else next time it is used (for example by \xintdefvar) the \xintexprRestoreCatcodes will restore an obsolete catcode regime...

13.2 Library identification

```
\xintexprSafeCatcodes\catcode`_ 11
\XINT_providespackage
\ProvidesPackage{xintlog}%
[2021/03/29 v1.4d Logarithms and exponentials for xintexpr (JFB)]%
```

13.3 Loading of poormanlog package

Attention to catcode regime when loading poormanlog. It matters less now for 1.3f as those chunks of code from poormanlog.tex v0.04 which needed specific xintexpr like catcodes got transferred here anyway.

```
\ifdefined\RequirePackage
  \RequirePackage{poormanlog}%
\else
  \input poormanlog.tex
\fi
```

\XINT_setcatcodes switches to the standard catcode regime of xint*.sty files. Formerly we needed here the ! of catcode 11 as in xintexpr.sty, which is set by \XINT_setcatcodes but does not apply now.

See the remark above about importance of doing \xintexprRestoreCatcodes if \xintexprSafeCatcodes has been used...

```
\xintexprRestoreCatcodes\csname XINT_setcatcodes\endcsname
```

13.4 The log10() and pow10() functions

The support macros from poormanlog v0.04 \PoorManLogBaseTen, \PoorManLogPowerOfTen, \PoorManPower got transferred into xintfrac.sty at 1.3f.

```
\expandafter\def\csname XINT_expr_func_log10\endcsname#1#2#3%
```

```
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\PoorManLogBaseTen#3}}%
}%
\expandafter\let\csname XINT_flexpr_func_log10\expandafter\endcsname
                \csname XINT_expr_func_log10\endcsname
\expandafter\def\csname XINT_expr_func_pow10\endcsname#1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\PoorManPowerOfTen#3}}%
}%
\expandafter\let\csname XINT_flexpr_func_pow10\expandafter\endcsname
                \csname XINT_expr_func_pow10\endcsname
```

13.5 The log(), exp(), and pow() functions

The log10() and pow10() were defined by poormanlog v0.04 but have been moved here at xint 1.3f. The support macros are defined in xintfrac.sty.

```
\def\XINT_expr_func_log #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintLog#3}}%
}%
\def\XINT_flexpr_func_log #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\XINTinFloatLog#3}}%
}%
\def\XINT_expr_func_exp #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\xintExp#3}}%
}%
\def\XINT_flexpr_func_exp #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\XINTinFloatExp#3}}%
}%
\def\XINT_expr_func_pow #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\PoorManPower#3}}%
}%
\let\XINT_flexpr_func_pow\XINT_expr_func_pow
```

13.6 \poormanloghack

With \poormanloghack{**}, the ** operator will use pow10(y*log10(x)). Same for ^. Sync'd with xintexpr 1.4.

MEMO: the reason why I need to redefine a lot of stuff is that xintexpr.sty does the job only for ^ and then does a \let for exec_** only. So if now ^ and ** possibly act differently all must be duplicated.

```
\catcode`\* 11
\def\poormanloghack**
{%
 \def\XINT_tmpa ##1##2##3##4##5##6%
 {%
  \def ##3####1% \XINT_expr_op_<op>
  {%
    \expanded{\unexpanded{##4{####1}}\expandafter}%
    \romannumeral`&&@\expandafter##2\romannumeral`&&@\XINT_expr_getnext
  }%
  \def##2####1% \XINT_expr_check-_<op>
  {%
    \xint_UDsignfork
      ####1{\expandafter##2\romannumeral`&&@##1}%
        -{##5####1}%
    \krof
  }%
  \def##5####1####2% \XINT_expr_checkp_<op>
  {%
    \ifnum ####1>\XINT_expr_precedence_**
      \expandafter##5%
      \romannumeral`&&@\csname XINT_##6_op_####2\expandafter\endcsname
    \else
      \expandafter ####1\expandafter ####2%
    \fi
  }%
 }%
 \expandafter\XINT_tmpa
   \csname XINT_expr_op_-ix\expandafter\endcsname
   \csname XINT_expr_check-_**\endcsname
   \XINT_expr_op_**
   \XINT_expr_exec_**
   \XINT_expr_checkp_** {expr}%
 \expandafter\XINT_tmpa
   \csname XINT_flexpr_op_-ix\expandafter\endcsname
   \csname XINT_flexpr_check-_**\endcsname
   \XINT_flexpr_op_**
   \XINT_flexpr_exec_**
   \XINT_flexpr_checkp_** {flexpr}%
 \def\XINT_expr_exec_** ##1##2##3##4% \XINT_expr_exec_<op>
  {%
    \expandafter##2\expandafter##3\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\PoorManPower##1##4}}%
  }%
```

```
  \let\XINT_flexpr_exec_**\XINT_expr_exec_**
}%
\def\poormanloghack^
{%
 \def\XINT_tmpa ##1##2##3##4##5##6%
 {%
  \def ##3####1% \XINT_expr_op_<op>
  {%
    \expanded{\unexpanded{##4{####1}}\expandafter}%
    \romannumeral`&&@\expandafter##2\romannumeral`&&@\XINT_expr_getnext
  }%
  \def##2####1% \XINT_expr_check-_<op>
  {%
    \xint_UDsignfork
      ####1{\expandafter##2\romannumeral`&&@##1}%
        -{##5####1}%
    \krof
  }%
  \def##5####1####2% \XINT_expr_checkp_<op>
  {%
    \ifnum ####1>\XINT_expr_precedence_^
      \expandafter##5%
      \romannumeral`&&@\csname XINT_##6_op_####2\expandafter\endcsname
    \else
      \expandafter ####1\expandafter ####2%
    \fi
  }%
 }%
 \expandafter\XINT_tmpa
   \csname XINT_expr_op_-ix\expandafter\endcsname
   \csname XINT_expr_check-_^\endcsname
   \XINT_expr_op_^
   \XINT_expr_exec_^
   \XINT_expr_checkp_^ {expr}%
 \expandafter\XINT_tmpa
   \csname XINT_flexpr_op_-ix\expandafter\endcsname
   \csname XINT_flexpr_check-_^\endcsname
   \XINT_flexpr_op_^
   \XINT_flexpr_exec_^
   \XINT_flexpr_checkp_^ {flexpr}%
 \def\XINT_expr_exec_^ ##1##2##3##4% \XINT_expr_exec_<op>
  {%
    \expandafter##2\expandafter##3\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\PoorManPower##1##4}}%
  }%
  \let\XINT_flexpr_exec_^\XINT_expr_exec_^
}%
\def\poormanloghack#1{\csname poormanloghack#1\endcsname}%
```

IMPORTANT: We don't worry about resetting catcodes now as this file is theoretically only loadable from xintexpr.sty itself which will take care of the needed restore.

14 Cumulative line count

```
xintkernel: 598.
 xinttools:1629.
  xintcore:2172.
      xint:1623.
xintbinhex: 472.
   xintgcd: 368.
  xintfrac:3571.
xintseries: 386.
 xintcfrac:1029.
  xintexpr:4341.
  xinttrig: 555.
   xintlog: 197.
```

Total number of code lines: 16941. (but 3887 lines among them start either with {% or with }%.)
Each package starts with circa 50 lines dealing with catcodes, package identification and reloading management, also for Plain TEX. Version 1.4d of 2021/03/29.