The zeckendorf package

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

Package version: 0.9b (2025/10/07)
From source file zeckendorf.dtx of 07-10-2025 at 21:41:02 CEST

Part I. User manual

Part II. Commented source code

Part I. User manual

1. Mathematical background

Let us recall that the Fibonacci sequence starts with $F_0 = 0$, $F_1 = 1$, and obeys the recurrence $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. So $F_2 = 1$, $F_3 = 2$, $F_4 = 3$ and by a simple induction $F_k = k-1$. Ahem, not at all! Here are the first few, starting at $F_2 = 1$:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584 \ldots$$

Zeckendorf's Theorem says that any positive integer has a unique representation as a sum of the Fibonacci numbers F_n, $n \geq 2$, under the conditions that no two indices differ by one, and that no index is repeated. For example:

$$\begin{aligned} 10 &= 8 + 2 \\ 100 &= 89 + 8 + 3 \\ 1{,}000 &= 987 + 13 \end{aligned}$$

$$
\begin{aligned}
\texttt{10,000} &= 6765 + 2584 + 610 + 34 + 5 + 2\\
\texttt{100,000} &= 75025 + 17711 + 6765 + 377 + 89 + 21 + 8 + 3 + 1\\
\texttt{1,000,000} &= 832040 + 121393 + 46368 + 144 + 55\\
\texttt{10,000,000} &= 9227465 + 514229 + 196418 + 46368 + 10946 + 4181 + 377 + 13 + 3\\
\texttt{100,000,000} &= F_{39} + F_{37} + F_{35} + F_{32} + F_{30} + F_{28} + F_{23} + F_{21} + F_{15} + F_{13} + F_{11} + F_9 + F_4
\end{aligned}
$$

This is called the Zeckendorf representation, and it can be given either as above, or as the list of the indices (in decreasing or increasing order), or as a binary word which in the examples above are

$$
\begin{aligned}
\texttt{10} &= \texttt{10010}_{\texttt{Zeckendorf}}\\
\texttt{100} &= \texttt{1000010100}_{\texttt{Zeckendorf}}\\
\texttt{1,000} &= \texttt{100000000100000}_{\texttt{Zeckendorf}}\\
\texttt{10,000} &= \texttt{1010010000010001010}_{\texttt{Zeckendorf}}\\
\texttt{100,000} &= \texttt{100101000001001001010101}_{\texttt{Zeckendorf}}\\
\texttt{1,000,000} &= \texttt{10001010000000000010100000000}_{\texttt{Zeckendorf}}\\
\texttt{10,000,000} &= \texttt{1000001010010010100001000000100100}_{\texttt{Zeckendorf}}\\
\texttt{100,000,000} &= \texttt{10101001010100001010000010101010000100}_{\texttt{Zeckendorf}}\\
\texttt{1,000,000,000} &= \texttt{1010000100100001010101000001000101000101001}_{\texttt{Zeckendorf}}
\end{aligned}
$$

The least significant digit says whether the Zeckendorf representation uses F_2 and so on from right to left (one may prefer to put the binary digits in the reverse order, but doing as above is more reminiscent of binary, decimal, or other representations using a given radix). In the next-to-last example the word length is 39 - 2 + 1 = 38, and in general it is K - 1 where K is the largest index such that F_K is at most equal to the given positive integer. For 1,000,000,000 this maximal index is 44 and indeed the associated word has length 43.

In a Zeckendorf binary word the sub-word 11 never occurs, and this, combined wih the fact that the leading digit is 1, characterizes the Zeckendorf words.

Donald Knuth (whose name may ring some bells to TeX users) has shown that defining $a \circ b$ as $\sum_i \sum_j F_{a_i+b_j}$ where the a_i's and the b_j's are the indices involved in the respective Zeckendorf representations of a and b is an *associative* operation on positive integers (it is obviously commutative).

The Fibonacci recurrence can also be prolungated to negative n's, and it turns out that $F_{-n} = (-1)^{n-1}F_n$. **Donald Knuth** has shown that any relative integer has a unique representation as a sum of these ``NegaFibonacci'' numbers $F_{-n}$, $n \geq 1$, again with the condition that no index is repeated and no two indices differ by one. In the special case of zero, the representation is an empty sum. Here is the sequence of these ``NegaFibonacci'' numbers starting at $n = -1$:

1, -1, 2, -3, 5, -8, 13, -21, 34, -55, 89, -144, 233, -377, 610, -987...

2. Use on the command line

Open a command line window and execute:

```
etex zeckendorf
```

then follow the displayed instructions.

The (TeX Live) *tex executables are not linked with the readline library, and this makes interactive use quite painful. If you are on a decent system, launch the interactive session rather via

```
rlwrap etex zeckendorf
```

for a smoother experience.

3. Use as a LaTeX package

As expected, add to the preamble:

```
\usepackage{zeckendorf}
```

There are no options.

xintexpr is loaded, macros are defined to go from integers to Zeckendorf representations and back, and to compute the Knuth multiplication of positive integers.

\xintiieval is extended with the functions fib(), fibseq(), zeckindex() and zeck(). The $ is added to the syntax as infix operator (with same precedence as multiplication) doing the Knuth multiplication.

\ZeckTheFN This macro computes Fibonacci numbers.

```
\ZeckTheFN{100}, \ZeckTheFN{100 + 15}\newline
```

```
354224848179261915075, 483162952612010163284885
```

As shown, the argument can be an integer expression (only in the sense of \inteval, not in the one of \xinteval, for example you can not have powers only additions and multiplications). Negative arguments are allowed:

```
\ZeckTheFN{0}, \ZeckTheFN{-1}, \ZeckTheFN{-2}, \ZeckTheFN{-3},
\ZeckTheFN{-4}
```

```
0, 1, -1, 2, -3
```

The syntax of \xintiieval is extended via addition of a fib() function, which gives a convenient interface:

```
\xintiieval{seq(fib(n), n=-5..5, 10, 20, 100)}
```

```
5, -3, 2, -1, 1, 0, 1, 1, 2, 3, 5, 55, 6765, 354224848179261915075
```

```
\xintiieval{seq(fib(2^n), n=1..7)}
```

```
1, 3, 21, 987, 2178309, 10610209857723, 251728825683549488150424261
```

\ZeckTheFSeq This computes not only one but a whole contiguous series of Fibonacci numbers but its output format is a sequence of braced numbers, and tools such as those of xinttools are needed to manipulate its output. For this reason it is not further documented here.

The syntax of \xintiieval is extended via addition of a fibseq() function, which gives a convenient interface:

```
\xintiieval{fibseq(10,20)}\newline
[55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

Notice the square brackets used on output. In the terminology of xintexpr, the function produces a nutple. Use the * prefix to remove the brackets:

```
\xintiieval{reversed(*fibseq(-20,-10))}
-55, 89, -144, 233, -377, 610, -987, 1597, -2584, 4181, -6765
```

IMPORTANT: currently, fibseq(a,b) **falls into an infinite loop** if $a \geq b$. Use it only with $a < b$. Above we used the reversed() function to get the output in order from F_{-10} to F_{-20} and not from F_{-20} to F_{-10}.

\ZeckIndex This computes the largest index k such that $F_k \leq x$, where x is the input. The input is only *f*-expanded, if you need it to be an expression you must wrap it in \xintiieval.

The syntax of \xintiieval is extended via addition of a zeckindex() function, which gives a more convenient interface.

IMPORTANT: The input **must be positive** (for now). No check is made that this is the case.

Note: Input must not have more than a few thousand decimal digits.

```
\ZeckIndex{123456789123456789123456789123456789}
169
```

```
\ZeckTheFN{\ZeckIndex{123456789123456789123456789123456789}}
93202207781383214849429075266681969
```

```
\ZeckTheFN{1+\ZeckIndex{123456789123456789123456789123456789}}
150804340016807970735635273952047185
```

```
\ZeckIndex{\xintiieval{2^100}}
145
```

```
\xintiieval{zeckindex(2^100)}
145
```

```
\xintiieval{fib(zeckindex(2^100))}
898923707008479989274290850145
```

```
\xintiieval{2^100}
1267650600228229401496703205376
```

```
\xintiieval{fib(1 + zeckindex(2^100))}
1454489111232772683678306641953
```

```
\xintiieval{seq(zeckindex(10^n), n = 0..40)}
2, 6, 11, 16, 20, 25, 30, 35, 39, 44, 49, 54, 59, 63, 68, 73, 78, 83, 87,
92, 97, 102, 106, 111, 116, 121, 126, 130, 135, 140, 145, 150, 154, 159,
164, 169, 173, 178, 183, 188, 193
```

\ZeckIndices This computes the Zeck representation as a comma separated list of indices. The input is only *f*-expanded, if you need it to be an expression you must wrap it in \xintiieval.

The macro is also known as \ZeckZeck.

The syntax of \xintiieval is extended via addition of a zeck() function, which gives a more convenient interface.

IMPORTANT: The input **must be positive**. No check is made that this is the case.

Note: Input must not have more than a few thousand decimal digits.

```
\ZeckZeck{123456789123456789123456789}
```

```
126, 123, 119, 117, 109, 104, 101, 95, 93, 90, 86, 84, 81, 76, 72, 69,
63, 61, 59, 55, 52, 50, 46, 41, 39, 37, 35, 33, 31, 29, 27, 25, 23, 20,
14, 11, 9, 6, 4, 2
```

Here is with zeck():

```
\xintiieval{zeck(123456789)}
```

```
[40, 36, 34, 28, 26, 24, 18, 16, 13, 7, 5, 2]
```

There are brackets, because the zeck() function produces a *nutple* (see xintexpr documentation). You can use the * prefix to unpack.

```
\xintiieval{*zeck(123456789123456789123456789)}
```

```
126, 123, 119, 117, 109, 104, 101, 95, 93, 90, 86, 84, 81, 76, 72, 69,
63, 61, 59, 55, 52, 50, 46, 41, 39, 37, 35, 33, 31, 29, 27, 25, 23, 20,
14, 11, 9, 6, 4, 2
```

It is easy with this syntax to manipulate the indices in various ways. Let's simple print them from smallest to largest:

```
\xintiieval{*reversed(zeck(123456789123456789123456789))}
```

```
2, 4, 6, 9, 11, 14, 20, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 46, 50,
52, 55, 59, 61, 63, 69, 72, 76, 81, 84, 86, 90, 93, 95, 101, 104, 109,
117, 119, 123, 126
```

The power of the \xintiieval, always eager to prove A=A, can be demonstrated:

```
\xintiieval{add(fib(n), n = *zeck(123456789))}
```

```
123456789
```

```
\xintiieval{add(fib(n), n = *zeck(123456789123456789123456789))}
```

```
123456789123456789123456789
```

TeX-nical note: There is also \ZeckBList which produces the indices as a sequence of braced items. To manipulate conveniently such outputs you need macros from xinttools or from LaTeX3. It is easier to use the powerful \xintiieval interface such as for example:

```
The first five indices are \xintiieval{*zeck(123456789123456789123456789)[:5]}.
```

The first five indices are 126, 123, 119, 117, 109.

```
The Zeckendorf representation of 123456789123456789123456789 uses
\xintiieval{len(zeck(123456789123456789123456789))} Fibonacci numbers.
```

The Zeckendorf representation of 123456789123456789123456789 uses 40 Fibonacci numbers.

\ZeckWord This computes the Zeck representation as a binary word. The input is only *f*-expanded, if you need it to be an expression you must wrap it in \xintiieval.

IMPORTANT: The input **must be positive**. No check is made that this is the case.

Note: Input must not have more than a few thousand decimal digits.

```
\ZeckWord{123456789}
```
```
1000101000001010100000101001000001010 01
```

```
\ZeckWord{\xintiieval{2^40}}
```
```
1000100000000100000010100000010101010101010001000101010100010
```

As TeX does not by default split long strings of digits at the line ends, we gave so far only some small examples. See xint or bnumexpr documentations for a \printnumber macro able to add linebreaks. Using such an auxiliary (a bit refined) we can for example obtain this:

```
\ZeckWord{\xintiieval{2^100}}
```
```
1010000010010010101010100100000000100100100101010001010010000100100⤶
1010010000000010100001001010101000000101001000100000000010010010001⤶
0010010100
```

Compare the above with the list of indices in the Zeckendorf representation: 145, 143, 137, 134, 131, 129, 127, 125, 123, 120, 111, 108, 105, 102, 100, 98, 94, 92, 89, 84, 81, 78, 76, 73, 64, 62, 57, 54, 52, 50, 48, 41, 39, 36, 32, 22, 19, 16, 12, 9, 6, 4.

\ZeckNFromIndices This computes an integer from a list of (comma separated) indices. These indices do not have to be positive, their order is indifferent and they can be repeated or differ by only one unit. The list is allowed to be empty. Contiguous commas (or commas separated only by space characters) act as a single one, a final comma is tolerated. A new *f*-expansion is done at each item, they can be (*f*-expandable) macros.

```
\ZeckNFromIndices{}\newline
\ZeckNFromIndices{100, ,,, 90, 80, 70, 60, 50, 40, 30 , , ,,,}
```
```
0
357128524055170099155
```

```
\ZeckIndices{357128524055170099155}
```
```
100, 90, 80, 70, 60, 50, 40, 30
```

```
\ZeckIndices{\ZeckNFromIndices{100, 90, 80, 70, 60, 50, 40, 30}}
```
```
100, 90, 80, 70, 60, 50, 40, 30
```

```
\ZeckNFromIndices{3,-1,4,-1,5,-9,2,-6,5,-3}
```
```
46
```

There is no associated \xintiieval function (currently) but it is a one-liner in its syntax:

```
\xintiieval{add(fib(i), i= 100, 90, 80, 70, 60, 50, 40, 30)}
```
```
357128524055170099155
```

```
\xintiieval{add(fib(i), i= 3, -1, 4, -1, 5, -9, 2, -6, 5, -3)}
46
```

```
\xintiieval{add(fib(i), i = *zeck(10^60))}
1000000000000000000000000000000000000000000000000000000000000
```

TeX-nical note: The first version of this documentation was using 1e60 in place of 10^60, which did not crash by sheer (mis-) luck (1+1e60 would have) and actually gave the correct result by some improbable combination of factors. The author had forgotten than the scientific notation 1e60 is not accepted (attow, using xintexpr 1.4o) in \xintiieval, contrarily to what happens with \xinteval. There is a shortcut for decimal powers which is a bit confidential: 1[⟨*exponent*⟩], so here we can use 1[60] or more generally n[E] as shortcut for E added trailing zeros. These things are, or should be, explained in some sections in fine print (or which should be in fine print) in the xintexpr PDF documentation.

\ZeckNFromWord This computes a positive integer from a binary word. The word can be arbitrary apart from not being empty.

```
\ZeckNFromWord{1}, \ZeckNFromWord{11}, \ZeckNFromWord{111},
\ZeckNFromWord{1111}, \ZeckNFromWord{11111}
1, 3, 6, 11, 19
```

```
\ZeckNFromWord{\xintReplicate{30}{10}}
4052739537880
```

```
\ZeckWord{4052739537880}
101010101010101010101010101010101010101010101010101010101010
```

\ZeckKMul This computes the Knuth multiplication of its two **positive** integer arguments. The two arguments are only *f*-expanded, you need to wrap each in an \xintiieval if it is an expression.

The syntax of \xintiieval is extended via addition of a $ infix operator, which gives a more convenient interface.

```
\ZeckKMul{100}{200}
44800
```

```
\ZeckKMul{\ZeckKMul{100}{200}}{300}
30079200
```

```
\ZeckKMul{100}{\ZeckKMul{200}{300}}
30079200
```

```
\xintiieval{100 $ 200, (100 $ 200) $ 300, 100 $ (200 $ 300)}
44800, 30079200, 30079200
```

The implementation is done via the Knuth definition: each operand is converted to a Zeckendorf representation, the indices are added and the sum of Fibonacci numbers is computed. Let us mention here that we could have defined a knuth() function easily using the powerful \xintiieval syntax:[1]

[1] We could not have used \xintdefiifunc here to define knuth(), so we used the \xintNewFunction interface. The sole inconvenient is that when using knuth() it is as if we injected by hand the replacement expression, which will have to be parsed by \xintiieval.

```
\xintNewFunction{knuth}[2]
   {add(fib(x), x = flat(ndmap(+, *zeck(#1); *zeck(#2);)))}
\xintiieval{knuth(100,200), knuth(knuth(100,200),300),
                            knuth(100,knuth(200,300))}
```

```
44800, 30079200, 30079200
```

The advantage of knowing this is that we can now check that our intuition about what happens when we compute `(a $ b) $ c`, which Knuth proved to be the same as `a $ (b $ c)`, is valid:

```
\xintNewFunction{knuththree}[3]
 {add(fib(x), x= flat(ndmap(+, *zeck(#1); *zeck(#2); *zeck(#3);)))}
\xintiieval{knuththree(100, 200, 300)}
\newline
\xintiieval{knuththree(1000, 2000, 3000), (1000 $ 2000) $ 3000,
                                          1000 $ (2000 $ 3000)}
```

```
30079200
29998632000, 29998632000, 29998632000
```

\ZeckSetAsKnuthOperator This takes as input a charcter, or multiple characters, and turns them (as a unit) work into an infix operator inside of `\xintiieval` computing the Knuth multiplication. The pre-defined use of `$` for this will not be canceled. You need to also do `\ZeckDeleteOperator{$}` if you want this meaning of `$` to be lost. In general repeated usage will only extend the list of operators doing the Knuth multiplication without removing the previously defined ones, except if `\ZeckDeleteOperator` is used for them.

IMPORTANT: There is NO WARNING if you override a pre-existing operator from the `\xintiieval` syntax (and not all such operators are user-documented because some exist for internal purposes only). But if done inside a group or environment, the former meaning will be recovered on exit.

A possible choice is to use `$$`. This may help avoiding syntax highlighting problems in your editor (or make them worse as I am currently experimenting while writing this). You can use `$ $` it is the same as `$$` to `\xintiieval`.

```
\ZeckSetAsKnuthOperator{$$}
\xintiieval{100 $$ 200, 200 $ $ 300, 100 $ $ 300}
```

```
44800, 134400, 67200
```

There are a few important points to be aware of:

- You can use a letter such as `o` as operator but it then must be used prefixed by `\string` which is not convenient:

  ```
  \ZeckSetAsKnuthOperator{o}
  \xintiieval{100 \string o 200 \string o 300}
  ```

  ```
  30079200
  ```
- With a Unicode engine, they are plenty of available characters that are already of catcode 12. For example:

```
\ZeckSetAsKnuthOperator{⊙}
\xintiieval{100 ⊙ 200 ⊙ 300}
```

30079200

You can also use letters from Greek or other scripts, but make sure they have catcode 12.

- It is not possible to use as operator a control sequence such as `\odot`. It has to be one or more characters. It can not be the period (full stop) which, although not being a predefined operator is recognized as decimal separator (even in `\xintiieval` due to some shared code with `\xinteval`).
- In case your document is compiled with `pdflatex` or `latex` and uses Babel, some characters may be catcode active. To use them as part of a name of an operator defined by `\ZeckSetAsKnuthOperator`, each such catcode active character has to be prefixed with `\string`. But `\string` is then *unneeded* inside `\xintiieval` (since `xintexpr 1.4n`).

\ZeckIndexedSum This is a utility which produces (expandably) `F_a + F_{a'} + ...` where `a, a', ...` are the Zeckendorf indices in decreasing order and the Fibonacci numbers are represented by the letter `F` and the index as subscript. Can only be used from inside math mode.

```
$\ZeckIndexedSum{100000000000000}$.
```

$F_{68} + F_{65} + F_{63} + F_{61} + F_{59} + F_{54} + F_{47} + F_{43} + F_{41} + F_{39} + F_{37} + F_{35} + F_{31} + F_{29} + F_{25} + F_{22} + F_{16} + F_{9} + F_{4} + F_{2}$.

The `+` is actually injected by `\ZeckIndexedSumSep` which defaults to mean `+\allowbreak`, so that as shown above a linebreak can be inserted by TeX.

\ZeckExplicitSum This is a utility which produces (expandably) `F_a + F_{a'} + ...` where `a, a', ...` are the Zeckendorf indices in decreasing order, and the Fibonacci numbers are written explicitly using decimal digits. May be used outside of math mode, but there will then be no spacing around the `+` signs.

```
$\ZeckExplicitSum{100000000000000}$.
```

$72723460248141+17167680177565+6557470319842+2504730781961+956722026041+ 86267571272 + 2971215073 + 433494437 + 165580141 + 63245986 + 24157817 + 9227465 + 1346269 + 514229 + 75025 + 17711 + 987 + 34 + 3 + 1$.

The `+` is actually injected by `\ZeckExplicitSumSep` which defaults to mean `+\allowbreak`, so that as shown above a linebreak can be inserted by TeX.

However, as one can see above and was already mentioned, TeX and LaTeX do not know out-of-the-box to split strings of digits at line endings. Hence the first line is squeezed, which is not pleasing, and a number extends nevertheless into the margin. The actual printing (and computation from the index) of the Fibonacci number is done via `\ZeckExplicitOne` whose default definition is to be an alias of `\ZeckTheFN`.

So if we redefine for example this way

```
\renewcommand\ZeckExplicitOne[1]{F_{#1}}
```

we will simply reconstruct what \ZeckIndexedSum does. Or, with the help of a xinttools utility we can inject breakpoints in between digits:

```
\renewcommand\ZeckExplicitOne[1]
   {\xintListWithSep{\allowbreak}{\ZeckTheFN{#1}}}
$\ZeckExplicitSum{100000000000000}$.
```

$72723460248141 + 17167680177565 + 6557470319842 + 2504730781961 + 956722026041 + 86267571272 + 2971215073 + 433494437 + 165580141 + 63245986 + 24157817 + 9227465 + 1346269 + 514229 + 75025 + 17711 + 987 + 34 + 3 + 1$.

Expert LaTeX users will know how to achieve a result such as this one, which pleasantly decorate the linebreaks:

$72723460248141 + 17167680177565 + 6557470319842 + 2504730781961 + 956722$↩
$026041 + 86267571272 + 2971215073 + 433494437 + 165580141 + 63245986 + 241$↩
$57817 + 9227465 + 1346269 + 514229 + 75025 + 17711 + 987 + 34 + 3 + 1$.

4. Use with Plain ε-TEX

You will need to input the core code using:

\input zeckendorfcore.tex

IMPORTANT: after this \input, the catcode regimen is a specific one (for example _, :, and ^ all have catcode letter). So, you will probably want to emit \ZECKrestorecatcodes immediately after this import, it will reset all modified catcodes to their values as prior to the import.

Then you can use the exact same interface as described in the previous section.

5. Changes

0.9b (2025/10/07)

Bug fixes:

- The instructions for interactive use mentioned 1e100 as possible input, but the author had forgotten that this syntax is not legitimate in \xintiieval (for example 1+1e10 crashes immediately).
- The code tries at some locations to be compatible with xintexpr versions earlier than 1.4n. But these versions did not load xintbinhex automatically and the needed \RequirePackage or \input for Plain TEX was lacking from the zeckendorf code.

Other changes: In the interactive interface, the input may now start with an \xintiieval function such as binomial whose first letter coincides with one of the letter commands without it being needed to for example add some \empty control sequence first. On the other hand, it was possible to use the full command names, now only their first letters (lower or uppercase) are recognized as such.

0.9alpha (2025/10/06) Initial release.

6. License

Part II.
Commented source code

7. Core code

A general remark is that expandable macros (usually) *f*-expand their arguments, and most are *f*-expandable. This *f*-expandability is achieved via \expanded triggers, diverging a bit from the overall style of the xint codebase (which predates availability of \expanded).

Extracts to zeckendorfcore.tex.

7.1. Loading xintexpr and setting catcodes

0.9alpha had a left-over \noexpand before the \endinput due to an oversight after replacing an \edef by a \def, embarrassing but unimportant. Also it made at a few places some effort to be compatible with older xint, but did not explicitly require xintbinhex, which is automatically loaded only since xintexpr 1.4n.

```
\input xintexpr.sty
\input xintbinhex.sty
\wlog{Package: zeckendorfcore 2025/10/07 v0.9b (JFB)}%
\edef\ZECKrestorecatcodes{\XINTrestorecatcodes}%
\def\ZECKrestorecatcodesendinput{\ZECKrestorecatcodes\endinput}%
\XINTsetcatcodes%
```

Small helpers related to \expanded-based methods. But the package only has a few macros and these helpers are used only once or twice, some macros needing their own terminators due to various optimizations in the code organization.

```
\def\zeck_abort#1\xint:{{}}%
\def\zeck_done#1\xint:{\iffalse{\fi}}%
```

7.2. Support for computing Fibonacci numbers: \ZeckTheFN, \ZeckTheFSeq

The multiplicative algorithm is as in the bnumexpr manual (at 1.7b), but termination is different and simply leaves F_n;F_{n-1}; in input stream (in a form requiring \xintthe).

\Zeck@FPair and \Zeck@@FPair are not public interface. The former is a wrapper of the latter to handle negative or zero argument.

The public \ZeckTheFN uses the \Zeck@FPair which accepts a negative or zero argument. The non public \Zeck@@FN uses \Zeck@@FPair and is thus limited to positive argument, also it remains in \xintexpr encapsulated format requiring \xintthe for explicit digits.

```
\def\Zeck@FPair#1{\expandafter\zeck@fpair\the\numexpr #1.}%
\def\zeck@fpair #1{%
   \xint_UDzerominusfork
      #1-\zeck@fpair_n
      0#1\zeck@fpair_n
      0-\zeck@fpair_p
   \krof #1%
}%
\def\zeck@fpair_p #1.{\Zeck@@FPair{#1}}%
\def\zeck@fpair_n #1.{%
    \ifodd#1 \expandafter\zeck@fpair_ei\else\expandafter\zeck@fpair_eii\fi
    \romannumeral`&&@\Zeck@@FPair{1-#1}%
}%
\def\zeck@fpair_ei{\expandafter\zeck@fpair_fi}%
\def\zeck@fpair_eii{\expandafter\zeck@fpair_fii}%
\def\zeck@fpair_fi#1;#2;{%
    \romannumeral0\xintiiexpro #2\expandafter\relax\expandafter;%
    \romannumeral0\xintiiexpro -#1\relax;%
}%
\def\zeck@fpair_fii#1;#2;{%
    \romannumeral0\xintiiexpro -#2\expandafter\relax\expandafter;%
    #1;%
}%
\def\Zeck@@FPair#1{%
    \expandafter\Zeck@start
    \romannumeral0\xintdectobin{\the\numexpr#1\relax};%
}%
\def\Zeck@start 1#1{%
    \csname Zeck@#1\expandafter\endcsname
    \romannumeral0\xintiiexpro 1\expandafter\relax\expandafter;%
    \romannumeral0\xintiiexpro 0\relax;%
}%
\expandafter\def\csname Zeck@0\endcsname #1;#2;#3{%
    \csname Zeck@#3\expandafter\endcsname
    \romannumeral0\xintiiexpro (#1+2*#2)*#1\expandafter\relax\expandafter;%
    \romannumeral0\xintiiexpro #1*#1+#2*#2\relax;%
}%
\expandafter\def\csname Zeck@1\endcsname #1;#2;#3{%
    \csname Zeck@#3\expandafter\endcsname
    \romannumeral0\xintiiexpro 2*(#1+#2)*#1+#2*#2\expandafter\relax\expandafter;%
    \romannumeral0\xintiiexpro (#1+2*#2)*#1\relax;%
}%
\expandafter\let\csname Zeck@;\endcsname\empty
```

For individual Fibonacci numbers, we have non public \Zeck@@FN and public \ZeckTheFN.

```
\def\Zeck@@FN{\expandafter\zeck@@fn\romannumeral`&&@\Zeck@@FPair}%
\def\zeck@@fn#1;#2;{#1}%
\def\ZeckTheFN{\xintthe\expandafter\zeck@@fn\romannumeral`&&@\Zeck@FPair}%
```

The computation of stretches of Fibonacci numbers is not needed for the package, but is provided for user convenience. This is lifted from the development version of the \xintname user manual, which refactored a bit the code which has been there for over ten years. As we want to add a fibseq() function to \xintiieval, it is better to make it *f*-expandable.

Here we also handle negative arguments but still require the second argument to be larger (more positive) than the first.

```
\def\ZeckTheFSeq#1#2{%#1=starting index, #2>#1=ending index
    \expanded\bgroup\expandafter\ZeckTheF@Seq
    \the\numexpr #1\expandafter.\the\numexpr #2.%
}%
```

The #1+1 is because \Zeck@FPair{N} expands to F_{N};F_{N-1};, so here we will have F_{A+1},F_{A}; as starting point. We want up to F_B. If B=A+1 there will be nothing to do.

```
\def\ZeckTheF@Seq #1.#2.{%
    \expandafter\ZeckTheF@Seq@loop
    \the\numexpr #2-#1-1\expandafter.\romannumeral0\Zeck@FPair{#1+1}%
}%
```

Now leave in stream one coefficient, test if we have reached B and until then apply standard Fibonacci recursion. We insert \xintthe although not needed for typesetting but this is useful for matters of defining an associated fibseq() function.

```
\def\ZeckTheF@Seq@loop #1.#2;#3;{% standard Fibonacci recursion
    {\xintthe#3}\ifnum #1=\z@ \expandafter\ZeckTheF@Seq@end\fi
    \expandafter\ZeckTheF@Seq@loop
    \the\numexpr #1-1\expandafter.%
    \romannumeral0\xintiiexpro #2+#3\relax;#2;%
}%
\def\ZeckTheF@Seq@end#1;#2;{{\xintthe#2}\iffalse{\fi}}%
```

7.3. \ZeckNearIndex, \ZeckIndex

If the ratio of logarithms was the exact mathematical value it would be certain (via rough estimates valid at least for say $x \geq 10$, and even smaller, but anyhow we can check manually it does work) that its integer rounding gives an integer K such that either K or K-1 is the largest index J with $F_J \leq x$. But the computation is done with only about

8 or 9 digits of precision. So certainly this assumption fails for x having more than one hundred million decimal digits, and would become a bit risky with an input having ten million digits.

But this is way beyond the reasonable range for usage of the package, as anyhow xint can handle multiplications only with operands of about up to 13000 digits, so there is no worry.

xintfrac's \xintiRound{0} is guaranteed to round correctly the input it has been given. This input is some approximation to an exact theoretical value involving ratio of logarithms (and square root of 5). Prior to rounding the computed numerical approximation, we are close to the exact theoretical value, where ``close'' means we expect to have about 8 leading digits in common (and we have already limited our scope so that we are talking about a value less than 10000 at any rate). If the computed rounding differs from the exact rounding of the exact value it must be that argument x is about mid-way (in log scale) between two consecutive Fibonacci numbers. The conclusion is that the integer we obtain after rounding can not be anything else than either J or J+1.

The argument is more subtle than it looks. The conclusion is important to us as it means we do not have to add extraneous checks involving computation of one or more additional Fibonacci numbers.

The formula with macros was obtained via an \xintdeffloatfunc and \xintverbosetrue after having set \xintDigits* to 8, and then we optimized a bit manually. The advantage here is that we don't have to set ourself \xintDigits and later restore it.

We can not use (except if only caring about interactive sessions where we control entirely the whole environment) \XINTinFloatDiv or \XINTinFloatMul if we don't set \xintDigits (which is user customizable) because they hardcode usage of \XINTdigits.

For the exact same reason 0.9b adds _raw postfix which had been forgotten at 0.9alpha. Indeed \PoorManLogBaseTen (without _raw) does an ``in-float'' conversion of its output, and this uses the current \XINTdigits and adds unnecessary overhead. The fix at 0.9b of this 0.9alpha oversight brought an efficiency gain of about 5% for this macro for inputs of 50 digits.

```
\def\ZeckNearIndex#1{\xintiRound{0}{%
    \xintFloatDiv[8]{\PoorManLogBaseTen_raw{\xintFloatMul[8]{2236068[-6]}{#1}}}%
                    {20898764[-8]}%
                          }%
}%
```

Now we compute the actual maximal index. This macro is only for user interface, because when obtaining the Zeckendorf representation via the greedy algorithm, we will want for efficienty to not discard the computed pair of Fibonacci numbers, but proceed using it.

```
\def\ZeckIndex{\expanded\zeckindex}%
\def\zeckindex#1{\expandafter\zeckindex_fork\romannumeral`&&@#1\xint:}%
\def\zeckindex_fork#1{%
  \xint_UDzerominusfork
    #1-\zeck_abort
    0#1\zeck_abort
    0-{\zeckindex_a#1}%
  \krof
}%
\def\zeckindex_a #1\xint:{%
    \expandafter\zeckindex_b
    \the\numexpr\ZeckNearIndex{#1}\xint:#1\xint:
}%
\def\zeckindex_b #1\xint:{%
    \expandafter\zeckindex_c
    \romannumeral`&&@\Zeck@@FPair{#1}#1\xint:
}%
\def\zeckindex_c #1;#2;#3\xint:#4\xint:{%
    \xintiiifGt{\xintthe#1}{#4}%
       {{\expandafter}\the\numexpr#3-1\relax}%
       {{}#3}%
}%
```

7.4. \ZeckIndices, \ZeckZeck

As explained at start of code comments, I decided when packaging the whole thing to make macros *f*-expandable via \expanded-trigger, not \romannumeral.

This and other macros start by computing the max index. It then subtracts the Fibonacci number from the input and loops.

```
\def\ZeckIndices{\expanded\zeckindices}%
\let\ZeckZeck\ZeckIndices
\def\zeckindices#1{\expandafter\zeckindices_fork\romannumeral`&&@#1\xint:}%
\def\zeckindices_fork#1{%
  \xint_UDzerominusfork
    #1-\zeck_abort
    0#1\zeck_abort
    0-{\bgroup\zeckindices_a#1}%
  \krof
}%
\def\zeckindices_a #1\xint:{%
    \expandafter\zeckindices_b
    \the\numexpr\ZeckNearIndex{#1}\xint:#1\xint:
}%
\def\zeckindices_b #1\xint:{%
    \expandafter\zeckindices_c
    \romannumeral`&&@\Zeck@@FPair{#1}#1\xint:
}%
\def\zeckindices_c #1;#2;#3\xint:#4\xint:{%
    \xintiiifGt{\xintthe#1}{#4}\zeckindices_A\zeckindices_B
    #1;#2;#3\xint:#4\xint:
}%
```

There is a slight annoyance here which is that we have to use the \xintthe... macros to have explicit digits so that we can test if the number is zero (if there is some macro for that in xintexpr it would do as us, look at the first digit, so we don't bother to check). But alas, the xintexpr manual has documented things such as \xintiiexprPrintOne as being customizable, so there is a potentiality here for user modifications causing a crash, if a custom \xintiiexprPrintOne prints Z or some other symbol in case of the zero value... We do have at our disposal \xintthebareiieval but it needs one more brace stripping step. So some \xinttheunbracedbareiieval is needed upstream and when this is done the code here will get updated.

```
\def\zeckindices_A#1;#2;#3\xint:#4\xint:{%
```

```
    \the\numexpr#3-1\relax
    \expandafter\zeckindices_loop
    \romannumeral`&&@\xinttheiiexpr #4-#2\relax\xint:
}%
\def\zeckindices_B#1;#2;#3\xint:#4\xint:{%
    #3%
    \expandafter\zeckindices_loop
    \romannumeral`&&@\xinttheiiexpr #4-#1\relax\xint:
}%
\def\zeckindices_loop#1{%
    \xint_UDzerofork#1\zeck_done 0{, \zeckindices_a#1}\krof
}%
```

7.5. \ZeckBList

This is the variant which produces the results as a sequence of braced indices. Useful as support for a zeck() function.

Originally in xint, xinttools, the term ``list'' is used for braced items. In the user manual of this package I have been using ``list'' more colloquially for comma separated values. Here I stick with xint conventions but use BList (short for ``list of Braced items'') and not only ``List'' in the name.

```
\def\ZeckBList{\expanded\zeckblist}%
\def\zeckblist#1{\expandafter\zeckblist_fork\romannumeral`&&@#1\xint:}%
\def\zeckblist_fork#1{%
  \xint_UDzerominusfork
    #1-\zeck_abort
    0#1\zeck_abort
    0-{\bgroup\zeckblist_a#1}%
  \krof
}%
\def\zeckblist_a #1\xint:{%
    \expandafter\zeckblist_b
    \the\numexpr\ZeckNearIndex{#1}\xint:#1\xint:
}%
\def\zeckblist_b #1\xint:{%
    \expandafter\zeckblist_c
    \romannumeral`&&@\Zeck@@FPair{#1}#1\xint:
}%
\def\zeckblist_c #1;#2;#3\xint:#4\xint:{%
    \xintiiifGt{\xintthe#1}{#4}\zeckblist_A\zeckblist_B
    #1;#2;#3\xint:#4\xint:
}%
\def\zeckblist_A#1;#2;#3\xint:#4\xint:{%
    {\the\numexpr#3-1\relax}%
    \expandafter\zeckblist_loop
    \romannumeral`&&@\xinttheiiexpr #4-#2\relax\xint:
}%
\def\zeckblist_B#1;#2;#3\xint:#4\xint:{%
    {#3}%
    \expandafter\zeckblist_loop
    \romannumeral`&&@\xinttheiiexpr #4-#1\relax\xint:
}%
\def\zeckblist_loop#1{\xint_UDzerofork#1\zeck_done 0{\zeckblist_a#1}\krof}%
```

7.6. \ZeckIndexedSum, \ZeckExplicitSum

They are expandable, but need *x*-expansion. The first one assumes it expands in math mode. We use \sb because the current catcode of _ is letter, and using \sb spares us some juggling.

```
\def\ZeckIndexedSumSep{+\allowbreak}%
\let\ZeckExplicitSumSep\ZeckIndexedSumSep
\def\ZeckExplicitOne{\xintthe\Zeck@@FN}%
\def\ZeckIndexedSum#1{%
    \expandafter\zeckindexedsum\expanded\zeckindices{#1},;%
}%
\def\zeckindexedsum#1{%
    \if,#1\expandafter\xint_gob_til_sc\fi \zeckindexedsum_a#1%
}%
\def\zeckindexedsum_a#1,{F\sb{#1}\zeckindexedsum_b}%
\def\zeckindexedsum_b #1{%
    \if;#1\expandafter\xint_gob_til_sc\fi
    \ZeckIndexedSumSep\zeckindexedsum_a#1%
}%
\def\ZeckExplicitSum#1{%
    \expandafter\zeckexplicitsum\expanded\zeckindices{#1},;%
}%
\def\zeckexplicitsum#1{%
    \if,#1\expandafter\xint_gob_til_sc\fi \zeckexplicitsum_a#1%
}%
\def\zeckexplicitsum_a#1,{\ZeckExplicitOne{#1}\zeckexplicitsum_b}%
\def\zeckexplicitsum_b #1{%
    \if;#1\expandafter\xint_gob_til_sc\fi
    \ZeckExplicitSumSep\zeckexplicitsum_a#1%
}%
```

7.7. \ZeckWord

This is slightly more complicated than \ZeckIndices and \ZeckBList because we have to keep track of the previous index to know how many zeros to inject.

```
\def\ZeckWord{\expanded\zeckword}%
\def\zeckword#1{\expandafter\zeckword_fork\romannumeral`&&@#1\xint:}%
\def\zeckword_fork#1{%
  \xint_UDzerominusfork
    #1-\zeck_abort
    0#1\zeck_abort
    0-{\bgroup\zeckword_a#1}%
  \krof
}%
\def\zeckword_a #1\xint:{%
    \expandafter\zeckword_b\the\numexpr\ZeckNearIndex{#1}\xint:
    #1\xint:
}%
```

```
\def\zeckword_b #1\xint:{%
    \expandafter\zeckword_c\romannumeral`&&@\Zeck@@FPair{#1}#1\xint:
}%
\def\zeckword_c #1;#2;#3\xint:#4\xint:{%
    \xintiiifGt{\xintthe#1}{#4}\zeckword_A\zeckword_B
    #1;#2;#3\xint:#4\xint:
}%
\def\zeckword_A#1;#2;#3\xint:#4\xint:{%
    \expandafter\zeckword_d
    \romannumeral`&&@\xinttheiiexpr#4-#2\expandafter\relax\expandafter\xint:
    \the\numexpr#3-1.%
}%
\def\zeckword_B#1;#2;#3\xint:#4\xint:{%
    \expandafter\zeckword_d
    \romannumeral`&&@\xinttheiiexpr#4-#1\relax\xint:
    #3.%
}%
\def\zeckword_d #1%
    {\xint_UDzerofork#1\zeckword_done0{1\zeckword_e}\krof #1}%
\def\zeckword_done#1\xint:#2.{1\xintReplicate{#2-2}{0}\iffalse{\fi}}%
\def\zeckword_e #1\xint:{%
    \expandafter\zeckword_f\the\numexpr\ZeckNearIndex{#1}\xint:
    #1\xint:
}%
\def\zeckword_f #1\xint:{%
    \expandafter\zeckword_g\romannumeral`&&@\Zeck@@FPair{#1}#1\xint:
}%
\def\zeckword_g #1;#2;#3\xint:#4\xint:{%
    \xintiiifGt{\xintthe#1}{#4}\zeckword_gA\zeckword_gB
    #1;#2;#3\xint:#4\xint:
}%
\def\zeckword_gA#1;#2;#3\xint:#4\xint:{%
    \expandafter\zeckword_h
    \the\numexpr#3-1\expandafter.%
    \romannumeral`&&@\xinttheiiexpr #4-#2\relax\xint:
}%
\def\zeckword_gB#1;#2;#3\xint:#4\xint:{%
    \expandafter\zeckword_h
    \the\numexpr#3\expandafter.%
    \romannumeral`&&@\xinttheiiexpr #4-#1\relax\xint:
}%
\def\zeckword_h #1.#2\xint:#3.{%
    \xintReplicate{#3-#1-1}{0}%
    \zeckword_d #2\xint:#1.%
}%
```

7.8. The Knuth Multiplication: \ZeckKMul

Here a \romannumeral0 trigger is used to match \xintiisum. Although it induces defining one more macro we obide by the general coding style of xint with a CamelCase then a lowercase macro, rather than having them merged as only one.

```
\def\ZeckKMul{\romannumeral0\zeckkmul}%
```

```
\def\zeckkmul#1#2{\expandafter\zeckkmul_a
                  \expanded{\ZeckIndices{#1}%
                            ,;%
                            \ZeckIndices{#2}%
                            ,,}%
}%
```

The space token at start of #2 after first one is not a problem as it ends up in a \numexpr anyhow.

```
\def\zeckkmul_a{\expandafter\xintiisum\expanded{{\iffalse}}\fi
                \zeckkmul_b}%
\def\zeckkmul_b#1;#2,{%
    \if\relax#2\relax\expandafter\zeckkmul_done\fi
    \zeckkmul_c{#2}#1,\zeckkmul_b#1;%
}%
\def\zeckkmul_c#1#2,{%
    \if\relax#2\relax\expandafter\xint_gobble_iii\fi
    {\xintthe\Zeck@@FN{#1+#2}}\zeckkmul_c{#1}%
}%
\def\zeckkmul_done#1;{\iffalse{{\fi}}}%
```

7.9. \ZeckNFromIndices

Spaces before commas are not a problem they disappear in \numexpr.

Each item is *f*-expanded to check not empty, but perhaps we could skip expanding, as they end up in \numexpr. Advantage of expansion of each item is that at any location we can generate multiple indices if desired.

```
\def\ZeckNFromIndices{\romannumeral0\zecknfromindices}%
\def\zecknfromindices#1{\expandafter\zecknfromindices_a\romannumeral`&&@#1,;}%
\def\zecknfromindices_a{\expandafter\xintiisum\expanded{{\iffalse}}\fi
                  \zecknfromindices_b
}%
\def\zecknfromindices_b#1{%
    \if;#1\xint_dothis\zecknfromindices_done\fi
    \if,#1\xint_dothis\zecknfromindices_skip\fi
    \xint_orthat\zecknfromindices_c #1%
}%
\def\zecknfromindices_c #1,{%
    {\ZeckTheFN{#1}}\expandafter\zecknfromindices_b\romannumeral`&&@%
}%
\def\zecknfromindices_skip,{\expandafter\zecknfromindices_b\romannumeral`&&@}%
\def\zecknfromindices_done;{\iffalse{{\fi}}}%
```

7.10. \ZeckNFromWord

The \xintreversedigits will *f*-expand its argument.

```
\def\ZeckNFromWord{\romannumeral0\zecknfromword}%
\def\zecknfromword#1{%
    \expandafter\zecknfromword_a\romannumeral0\xintreversedigits{#1};%
}%
\def\zecknfromword_a{%
    \expandafter\xintiisum\expanded{{\iffalse}}\fi\zecknfromword_b 2.%
}%
\def\zecknfromword_b#1.#2{%
    \if;#2\expandafter\zecknfromword_done\fi
    \if#21{\xintthe\Zeck@@FN{#1}}\fi
    \expandafter\zecknfromword_b\the\numexpr#1+1.%
}%
\def\zecknfromword_done#1.{\iffalse{{\fi}}}%
```

7.11. Extension of the \xintiieval syntax with fib(), fibseq(), zeck() and zeckindex() functions

fib() and fibseq() accept negative arguments, but fibseq(a,b) must be with b>a, else falls into an infinite loop. zeck() and zeckindex() require, but do not check, that their argument is positive.

We also add support for these functions to \xinteval and \xintfloateval. Arguments are then truncated (not rounded) to integers.

```
\def\XINT_iiexpr_func_fib #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\ZeckTheFN#3}}%
}%
\def\ZeckTheFNnum#1{\ZeckTheFN{\xintNum{#1}}}%
\def\XINT_expr_func_fib #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\ZeckTheFNnum#3}}%
}%
\let\XINT_flexpr_func_fib\XINT_expr_func_fib
\def\XINT_iiexpr_func_fibseq #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\ZeckTheFSeq#3}}%
}%
\def\ZeckTheFSeqnum#1#2{\ZeckTheFSeq{\xintNum{#1}}{\xintNum{#2}}}%
\def\XINT_expr_func_fibseq #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:two
    {\romannumeral`&&@\ZeckTheFSeqnum#3}}%
}%
\let\XINT_flexpr_func_fibseq\XINT_expr_func_fibseq
\def\XINT_iiexpr_func_zeckindex #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\ZeckIndex#3}}%
}%
\def\ZeckIndexnum#1{\ZeckIndex{\xintNum{#1}}}%
```

```
\def\XINT_expr_func_zeckindex #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\ZeckIndexnum#3}}%
}%
\let\XINT_flexpr_func_zeckindex\XINT_expr_func_zeckindex
\def\XINT_iiexpr_func_zeck #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\ZeckBList#3}}%
}%
\def\ZeckBListnum#1{\ZeckBList{\xintNum{#1}}}%
\def\XINT_expr_func_zeck #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\ZeckBListnum#3}}%
}%
\let\XINT_flexpr_func_zeck\XINT_expr_func_zeck
```

7.12. Extension of the \xintiieval syntax with $ as infix operator for the Knuth multiplication

Unfortunately, contrarily to bnumexpr, xintexpr (at 1.4o) has no public interface to define an infix operator, and the macros it defines to that end have acquired another meaning at end of loading xintexpr.sty, so we have to copy quite a few lines of code. This is provisory and will be removed when xintexpr.sty will have been udpated. We also copy/adapt \bnumdefinfix.

We test for existence of \xintdefinfix so as to be able to update upstream and not have to sync it immediately. But perhaps upstream will choose some other name than \xintdefinfix...

```
\ifdefined\xintdefinfix
  \def\zeckdefinfix{\xintdefinfix {iiexpr}}%
\else
\ifdefined\xint_noxpd\else\let\xint_noxpd\unexpanded\fi % support old xint
\def\ZECK_defbin_c #1#2#3#4#5#6#7#8%
{%
  \XINT_global\def #1##1% \XINT_#8_op_<op>
  {%
    \expanded{\xint_noxpd{#2{##1}}\expandafter}%
    \romannumeral`&&@\expandafter#3\romannumeral`&&@\XINT_expr_getnext
  }%
  \XINT_global\def #2##1##2##3##4% \XINT_#8_exec_<op>
  {%
    \expandafter##2\expandafter##3\expandafter
      {\romannumeral`&&@\XINT:NEhook:f:one:from:two{\romannumeral`&&@#7##1##4}}%
  }%
  \XINT_global\def #3##1% \XINT_#8_check-_<op>
  {%
```

```
    \xint_UDsignfork
      ##1{\expandafter#4\romannumeral`&&@#5}%
        -{#4##1}%
    \krof
  }%
  \XINT_global\def #4##1##2% \XINT_#8_checkp_<op>
  {%
    \ifnum ##1>#6%
      \expandafter#4%
      \romannumeral`&&@\csname XINT_#8_op_##2\expandafter\endcsname
    \else
      \expandafter ##1\expandafter ##2%
    \fi
  }%
}%
```

ATTENTION there is lacking at end here compared to the bnumexpr version an adjustment for updating minus operator, if some other right precedence than 12, 14, 17 is used. Doing this would requiring copying still more, so is not done.

```
\def\ZECK_defbin_b #1#2#3#4#5%
{%
  \expandafter\ZECK_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_-\romannumeral\ifnum#4>12 #4\else12\fi\expandafter\endcsname
  \csname xint_c_\romannumeral#4\endcsname
  #5%
  {#1}% #8 for \ZECK_defbin_c
  \XINT_global
  \expandafter
  \let\csname XINT_expr_precedence_#2\expandafter\endcsname
      \csname xint_c_\romannumeral#3\endcsname
}%
```

These next two currently lifted from bnumexpr with some adaptations, see previous comment about precedences.

```
\def\zeckdefinfix #1#2#3#4%
{%
    \edef\ZECK_tmpa{#1}%
    \edef\ZECK_tmpa{\xint_zapspaces_o\ZECK_tmpa}%
    \edef\ZECK_tmpL{\the\numexpr#3\relax}%
    \edef\ZECK_tmpL
         {\ifnum\ZECK_tmpL<4 4\else\ifnum\ZECK_tmpL<23 \ZECK_tmpL\else 22\fi\fi}%
    \edef\ZECK_tmpR{\the\numexpr#4\relax}%
    \edef\ZECK_tmpR
         {\ifnum\ZECK_tmpR<4 4\else\ifnum\ZECK_tmpR<23 \ZECK_tmpR\else 22\fi\fi}%
    \ZECK_defbin_b {iiexpr}\ZECK_tmpa\ZECK_tmpL\ZECK_tmpR #2%
    \expandafter\ZECK_dotheitselves\ZECK_tmpa\relax
  \unless\ifcsname
    XINT_iiexpr_exec_-\romannumeral\ifnum\ZECK_tmpR>12 \ZECK_tmpR\else 12\fi
  \endcsname
  \xintMessage{zeckendorf}{Error}{Right precedence not among accepted values.}%
  \errhelp{Accepted values include 12, 14, and 17.}%
  \errmessage{Sorry, you can not use \ZECK_tmpR\space as right precedence.}%
 \fi
 \ifxintverbose
  \xintMessage{zeckendorf}{info}{infix operator \ZECK_tmpa\space
  \ifxintglobaldefs globally \fi
    does
    \xint_noxpd{#2}\MessageBreak with precedences \ZECK_tmpL, \ZECK_tmpR;}%
 \fi
}%
\def\ZECK_dotheitselves#1#2%
{%
  \if#2\relax\expandafter\xint_gobble_ii
  \else
 \XINT_global
   \expandafter\edef\csname XINT_expr_itself_#1#2\endcsname{#1#2}%
   \unless\ifcsname XINT_expr_precedence_#1\endcsname
 \XINT_global
    \expandafter\edef\csname XINT_expr_precedence_#1\endcsname
            {\csname XINT_expr_precedence_\ZECK_tmpa\endcsname}%
 \XINT_global
    \expandafter\odef\csname XINT_iiexpr_op_#1\endcsname
            {\csname XINT_iiexpr_op_\ZECK_tmpa\endcsname}%
   \fi
  \fi
  \ZECK_dotheitselves{#1#2}%
}%
\fi
```

There is no ``undefine operator'' in bnumexpr currently. Experimental, I don't want to spend too much time. I sense there is a potential problem with multi-character operators related to ``undoing the itselves'', because of the mechanism which allows to use for example ;; as short-cut for ;;; if ;; was not pre-defined when ;;; got defined. To undefine ;;, I would need to check if it really has been aliased to ;;;, and I don't do the effort here.

```
\ifdefined\xintdefinfix
\else
\ifdefined\xint_noxpd\else\let\xint_noxpd\unexpanded\fi % support old xint
\def\ZECK_undefbin_b #1#2%
{%
 \XINT_global\expandafter\let
  \csname XINT_#1_op_#2\endcsname\xint_undefined
 \XINT_global\expandafter\let
  \csname XINT_#1_exec_#2\endcsname\xint_undefined
 \XINT_global\expandafter\let
  \csname XINT_#1_check-_#2\endcsname\xint_undefined
 \XINT_global\expandafter\let
  \csname XINT_#1_checkp_#2\endcsname\xint_undefined
 \XINT_global\expandafter\let
  \csname XINT_expr_precedence_#2\endcsname\xint_undefined
 \XINT_global\expandafter\let
```

```
    \csname XINT_expr_itself_#2\endcsname\xint_undefined
}%
\def\zeckundefinfix #1%
{%
    \edef\ZECK_tmpa{#1}%
    \edef\ZECK_tmpa{\xint_zapspaces_o\ZECK_tmpa}%
    \ZECK_undefbin_b {iiexpr}\ZECK_tmpa
%%  \ifxintverbose
    \xintMessage{zeckendorf}{Warning}{infix operator \ZECK_tmpa\space
        has been DELETED!}%
%%  \fi
}%
\fi
```

We do not define the extra \chardef's as does bnumexpr to allow more user choices of precedences, not only because nobody will ever use the feature, but also because it needs extra configuration for minus unary operator. (as mentioned above)

Attention, this is like \bnumdefinfix and thus does not have same order of arguments as the \ZECK_defbin_b above.

```
\zeckdefinfix{$}{\ZeckKMul}{14}{14}% $ (<-only to tame Emacs/AUCTeX highlighting)
\def\ZeckSetAsKnuthOperator#1{\zeckdefinfix{#1}{\ZeckKMul}{14}{14}}%
\def\ZeckDeleteOperator#1{\zeckundefinfix{#1}}%
```

ATTENTION! we leave the modified catcodes in place! (the question mark has regained its catcode other though).

8. Interactive code

Extracts to zeckendorf.tex.

```
\input zeckendorfcore.tex
\xintexprSafeCatcodes
```

First release used some trick, but the nesting of conditionals in the main loop has become more involved, so let's do something more straightforward with a TeX boolean.

```
\let\ZeckShouldISayOrShouldIGo\iftrue
\def\ZeckCmdQ{\let\ZeckShouldISayOrShouldIGo\iffalse}
\let\ZeckCmdX\ZeckCmdQ
\let\ZeckCmdx\ZeckCmdQ
\let\ZeckCmdq\ZeckCmdQ

\newif\ifzeckindices
\def\ZeckCmdL{\zeckindicestrue
              \def\ZeckFromN{\ZeckIndices}\def\ZeckToN{\ZeckNFromIndices}}
\let\ZeckCmdl\ZeckCmdL

\def\ZeckCmdB{\zeckindicesfalse
              \def\ZeckFromN{\ZeckWord}\def\ZeckToN{\ZeckNFromWord}}
\let\ZeckCmdW\ZeckCmdB
\let\ZeckCmdb\ZeckCmdB
\let\ZeckCmdw\ZeckCmdB

\newif\ifzeckfromN
```

```
\zeckfromNtrue
\def\ZeckConvert{\csname Zeck\ifzeckfromN From\else To\fi N\endcsname}
\def\ZeckCmdT{\ifzeckfromN\zeckfromNfalse\else\zeckfromNtrue\fi}
\let\ZeckCmdt\ZeckCmdT

\newif\ifzeckmeasuretimes
\expandafter\def\csname ZeckCmd@\endcsname{%
  \ifdefined\xinttheseconds
      \ifzeckmeasuretimes\zeckmeasuretimesfalse\else\zeckmeasuretimestrue\fi
  \else
      \immediate\write128{Sorry, this requires xintexpr 1.4n or later.}%
  \fi
}

\newif\ifzeckevalonly
\def\ZeckCmdE{\ifzeckevalonly\zeckevalonlyfalse\else\zeckevalonlytrue\fi}
\let\ZeckCmde\ZeckCmdE

\ZeckCmdL

\def\ZeckInviteA{Commands are Q(uit), L(ist), W(ord), T(oggle), E(val-only) or @.}

\newlinechar10
\immediate\write128{}
\immediate\write128{Welcome to Zeckendorf 0.9b (2025/10/07, JFB).}

\immediate\write128{Command summary (lowercase also):^^J
 Q to quit. Also X.^^J
 L for Zeckendorf representations as lists of indices.^^J
 W for Zeckendorf representations as binary words. Also B.^^J
 T to toggle the direction of conversions.^^J
 E to toggle to and from  \string\xintiieval-only mode.^^J
 @ to toggle measurement of execution times.}
\immediate\write128{}
\immediate\write128{%
The input, except for "Word -> Integer", is parsed in \string\xintiieval.^^J%
So for example 2^100, 100!, or binomial(100,50) are legitimate.^^J%
\space\space The fib() function computes Fibonacci numbers.^^J%
\space\space The character $ serves as symbol for Knuth multiplication.^^J%$
List input can use negative integers, and order does not matter.^^J%
Binary word can be arbitrary, except empty.^^J}
\immediate\write128{**** empty input is not supported! no linebreaks in input! ****}

\def\zeckpar{\par}
\long\def\xintbye#1\xintbye{}
\long\def\zeckgobbleii#1#2{}
\long\def\zeckfirstoftwo#1#2{#1}
\def\zeckonlyonehelper #1#2#3\zeckonlyonehelper{\xintbye#2\zeckgobbleii\xintbye0}

\xintloop
\immediate\write128{\ZeckInviteA}
\message{\ifzeckevalonly (eval only mode, hit E to exit it)\else
```

```
         \ifzeckfromN Integer -> \ifzeckindices indices\else binary word\fi
         \else
           \ifzeckindices Indices \else Binary word \fi
         -> integer\fi\fi
         : }
\read-1to\zeckbuf
\ifx\zeckbuf\zeckpar
  \immediate\write128{**** empty input is not supported, please try again.}
\else
  \edef\zeckbuf{\zeckbuf}
```

Space token at end of \zeckbuf is annoying. We could have used \xintLength which does not count space tokens.

```
  \if 1\expandafter\zeckonlyonehelper\zeckbuf\xintbye\zeckonlyonehelper1%
   \ifcsname ZeckCmd\expandafter\zeckfirstoftwo\zeckbuf\relax\endcsname
     \csname ZeckCmd\expandafter\zeckfirstoftwo\zeckbuf\relax\endcsname
   \else
     \immediate\write128{%
     **** Unrecognized command letter
          \expandafter\zeckfirstoftwo\zeckbuf\relax. Try again.^^J}
   \fi
  \else
   \ifzeckfromN\edef\ZeckIn{\xintiieval{\zeckbuf}}\else
      \ifzeckindices\edef\ZeckIn{\xintiieval{\zeckbuf}}\else
         \def\ZeckIn{\zeckbuf}%
      \fi
   \fi
```

Using the conditional so that this can also be used by default with older xint.

```
   \ifzeckmeasuretimes\xintresettimer\fi
   \immediate\write128{\ifzeckevalonly\ZeckIn\else\ZeckConvert{\ZeckIn}\fi}%
   \immediate\write128{\ifzeckmeasuretimes
                       \ifzeckevalonly Evaluation \else Conversion \fi
                        took \xinttheseconds s^^J\fi}
  \fi
\fi
\ZeckShouldISayOrShouldIGo
\repeat

\immediate\write128{Bye. Results are also in log file (hard-wrapped too, alas).}
\bye
```

9. LaTeX code

Extracts to zeckendorf.sty.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{zeckendorf}
   [2025/10/07 v0.9b Zeckendorf representations of big integers (JFB)]%
\RequirePackage{xintexpr}
\RequirePackage{xintbinhex}% superfluous if with xint 1.4n or later
\input zeckendorfcore.tex
\ZECKrestorecatcodesendinput%
```