The zeckendorf package

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

Package version: 0.9d (2025/11/16)
From source file zeckendorf.dtx of 16-11-2025 at 18:36:26 CET

Warning

This package is still in alpha stage. Any or all of the user interface may change in backwards incompatible ways at each release.

Suggestions for new features are most welcome!

Part I. User manual

Part II. Commented source code

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 $F_2 = 1$:

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

The ratios of consecutive Fibonacci numbers are the convergents of the golden ratio ϕ.

$$\phi = \frac{1 + \mathrm{sqrt}(5)}{2} \approx 1.618{,}033{,}988{,}749{,}894{,}848{,}204{,}586{,}834{,}37.$$

The Fibonacci recurrence can also be prolungated to negative n's, and it turns out that $F_{-n} = (-1)^{n-1}F_n$.

Let us a give a few equations which are constantly in use. The first one implies explicitly, in particular, that $\mathbf{Z}[\phi]$ (i.e. all polynomial expression in ϕ with integer coefficients) is $\mathbf{Z} + \mathbf{Z}\phi$.

$$\forall n \in \mathbf{Z} \quad \phi^n = F_{n-1} + F_n\phi\,. \tag{1}$$

Applying the $\phi \leftrightarrow \psi = -\phi^{-1} = 1 - \phi$ automorphism of the ring $\mathbf{Z}[\phi]$ and adding we obtain the **Lucas** numbers:

$$L_n = \phi^n + \psi^n = 2F_{n-1} + F_n = F_{n-1} + F_{n+1}\,. \tag{2}$$

If subtracting, we obtain the **Binet** formula:

$$F_n = \frac{\phi^n - \psi^n}{\phi - \psi}\,. \tag{3}$$

Of course one should always keep in mind that $-1 < \psi < 0$. And perhaps also that $\phi - \psi = \sqrt{5}$.

Finally, there is an important formula using 2×2-matrices, closely related with equation (1) and the recurrence relation of the Fibonacci numbers:

$$\forall n \in \mathbf{Z} \quad \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}. \tag{4}$$

Zeckendorf's Theorem (**Lekkerkerker**'s [1] in 1952 (preprint 1951) attributes the result to Zeckendorf; Zeckendorf, who was not in academia, published [2] only later in 1972) 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:

$$\begin{aligned}
10 &= 8 + 2 = F_6 + F_3 \\
100 &= 89 + 8 + 3 = F_{11} + F_6 + F_4 \\
1{,}000 &= 987 + 13 = F_{16} + F_7 \\
10{,}000 &= 6765 + 2584 + 610 + 34 + 5 + 2 = F_{20} + F_{18} + F_{15} + F_9 + F_5 + F_3
\end{aligned}$$

$$\begin{aligned}
100{,}000 &= 75025 + 17711 + 6765 + 377 + 89 + 21 + 8 + 3 + 1\\
&= F_{25} + F_{22} + F_{20} + F_{14} + F_{11} + F_{8} + F_{6} + F_{4} + F_{2}\\
1{,}000{,}000 &= 832040 + 121393 + 46368 + 144 + 55\\
&= F_{30} + F_{26} + F_{24} + F_{12} + F_{10}\\
10{,}000{,}000 &= 9227465 + 514229 + 196418 + 46368 + 10946 + 4181 + 377 + 13 + 3\\
&= F_{35} + F_{29} + F_{27} + F_{24} + F_{21} + F_{19} + F_{14} + F_{7} + F_{4}\\
100{,}000{,}000 &= F_{39} + F_{37} + F_{35} + F_{32} + F_{30} + F_{28} + F_{23} + F_{21} + F_{15} + F_{13} + F_{11} + F_{9} + F_{4}
\end{aligned}$$

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

$$\begin{aligned}
10 &= 10010_{\mathrm{zeck}}\\
100 &= 1000010100_{\mathrm{zeck}}\\
1{,}000 &= 100000000100000_{\mathrm{zeck}}\\
10{,}000 &= 10100100000010001010_{\mathrm{zeck}}\\
100{,}000 &= 100101000001001001010101_{\mathrm{zeck}}\\
1{,}000{,}000 &= 10001010000000000010100000000_{\mathrm{zeck}}\\
10{,}000{,}000 &= 1000001010010010100001000000100100_{\mathrm{zeck}}\\
100{,}000{,}000 &= 10101001010100001010000010101010000100_{\mathrm{zeck}}\\
1{,}000{,}000{,}000 &= 1010000100100001010101000001000101000101001_{\mathrm{zeck}}
\end{aligned}$$

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 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 defined in 1988 a **Fibonacci multiplication** ([3]) of positive integers via the formula

$$a \circ b = \sum_{i,j} F_{a_i + b_j}\,, \tag{5}$$

where $a = \sum F_{a_i}$ and $b = \sum F_{b_j}$ are the Zeckendorf representations of the positive integers a and b. Although it is sometimes true that formula (5) remains valid when using non-Zeckendorf expressions of a and/or b as sums of Fibonacci numbers, this is not a general rule. The next identity by Knuth, which applies whenever three positive integers a, b, c are expressed via their Zeckendorf representations, is thus non-trivial:

$$(a \circ b) \circ c = \sum_{i,j,k} F_{a_i + b_j + c_k}\,. \tag{6}$$

From it, the associativity of the Fibonacci multiplication follows immediately, the same as commutativity followed immediately from (5).

Knuth's proof is combinatorial in nature. **Pierre Arnoux** ([4]) obtained in 1989 a non-combinatorial proof of associativity based upon the identification of a certain subset (or subsets) of the ring $\mathbf{Z}[\phi]$, closed under multiplication, and indexed by the positive integers. The circle-product on the indices is mapped to the standard multiplication of these algebraic integers A_n: $A_n A_m = A_{n\circ m}$. As by-product of this, he obtained the following remarkable alternative formula for the Knuth product:

$$a \circ b = ab + a\sum_j F_{b_j - 1} + b\sum_i F_{a_i - 1}\ . \tag{7}$$

Again, here we use the Zeckendorf representations of the positive integers a and b. Clearly formula (7) is advantageous numerically compared to original definition (5). Arnoux also re-interpreted a ``star-product'' which had been defined by **Horacio Porta** and **Kenneth Stolarsky** ([5]).

Donald Knuth (see [6, 7.1.3]) has shown that any relative integer has a unique representation as a sum of the ``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 . . .

In 1957, the twelve-year-old **George Bergman** ([7]) introduced the notion of a ``base ϕ'' number system. This uses 0 and 1 as digits but with the ambiguity rule 011 ↔ 100 due to $\phi^2 = \phi + 1$. He proved that any positive integer can be represented this way finitely, i.e. is a *finite* sum of powers ϕ^k, with decreasing relative integers as exponents (i.e. each power occurring at most once and it is crucial that negative powers are allowed). For example:

$$100 = \phi^9 + \phi^6 + \phi^3 + \phi^1 + \phi^{-4} + \phi^{-7} + \phi^{-10} = 1001001010.0001001001_\phi\ .$$

Such a finite ``phi-ary'' representation (it seems ``phi-representation'' is the more commonly used term in academia) is unique if one adds the condition that no two exponents differ by one. This is equivalent to requiring that the number of terms is minimal. The real numbers which can be represented by such finite sums are exactly the positive numbers in $\mathbf{Z}[\phi]$, i.e. all combinations $p + q\phi$ with p and q relative integers which turn out to be strictly positive.

$$100 - 30\phi = \phi^8 + \phi^3 + \phi^{-3} + \phi^{-10} = 100001000.0010000001_\phi\ .$$

The naive approach to obtain the finite phi-representations, and actually prove that they do exist for all positive integers, is to show how to repeatedly add 1 (hence also powers of ϕ). One then only needs to explain how to subtract 1 (hence also powers of ϕ) to deduce that all $p + q\phi > 0$ are representable. This is actually what Bergman did. If one wants, as we do, to be able to obtain the representations for integers having say more than a few decimal digits, this theoretical approach is simply not feasible as is, one needs a bit more thinking.

A theoretical way, called the ``greedy'' algorithm, is based upon the fact that for any $x = p + q\phi > 0$, the maximal exponent $k \in \mathbf{Z}$ in its minimal representation is characterized by $\phi^k \leq x < \phi^{k+1}$. So one only needs to get k and then replace x by $x - \phi^k$. Doing this using floating point number calculations will only be able to handle integers with few enough digits to be exactly representable, and may lead at some point to a negative x, hence fail, due to rounding errors. So here again one has to think a bit.

This has been done by the author, and the resulting algorithm is implemented (expandably) here in ε-TEX.[1] Of course this is only elementary mathematics and it would be extremely surprising if the algorithm was not in the literature. Inputs of hundreds of digits are successfully handled. The same, implemented in C or other language with a library for big integers, would of course go way beyond and be a thousand times faster.

An ``integer-only'' algorithm (i.e. an algorithm which can be made to process only integers, but is in fact restricted to them; to compare, the approach described in the previous paragraph is in principle also implementable using integers only, but it applies to all $x = p + q\phi > 0$ not only to integers) to obtain the Bergman minimal ϕ-representation of a positive integer N is explained by **Donald Knuth** in the solution to Problem 35 of section 1.2.8 from [8] (there is a typographical error with a missing negative sign in an exponent there, on page 495; this has been reported to the author). It starts with the position of N with respect to Lucas numbers, the more subtle case being when N follows an odd indexed Lucas number.

One has to think a bit how to find efficiently the largest Lucas number at most equal to N, when N has hundreds of digits. This is about the same as identifying the maximal k such as $\phi^k \leq N$, as $\phi^k + (-1)^k\phi^{-k} = L_k$ is an integer. It is also very similar to finding the Zeckendorf maximal index which essentially means to locate $\sqrt{5}N$ with respect to powers of ϕ (as $\phi^k - (-1)^k\phi^{-k}$ for $k \geq 1$ belongs to $\sqrt{5}\,\mathbf{N}$).

For $x = N$ an **integer** (at least 2) it can be proven that the smallest contribution $\phi^{-\ell}$ to the minimal Bergman representation is with $\ell = k$ if k is even and $\ell = k + 1$ is k is odd. Otherwise stated ℓ is the smallest even integer at least

---

[1] As the intrepid reader will see if looking at the code, this uses a little bit floating point logarithms witn mantissas of eight decimal digits. This is because we have arbitrary precision logarithm available from xintexpr, with the fastest being with eight decimal digits precision, and after all we were not preparing a reference paper for *Mathematics of Computation* but simply aiming at computing for fun as efficiently as we could using tools at our disposal. This shortcut induces a theoretical upper bound on the size of the starting x: if it is an integer it must have less than say about one million decimal digits (see subsubsection 8.4.2 for details). As we can do computations (with TEXLive 2025 default memory settings) only up to about `13000` decimal digits (and in reasonable time up to less than `1000` digits), this is not a problem to us. And if we were to use logarithms with about 16 decimal digits of precision, the theoretical limit would raise to say inputs of less than about 10^{14} decimal digits. Each of our decimal digit occupies one word of computer memory, and even if we were using a programming language manipulating binary numbers, we would need more than 37 terabytes of computer memory to store the binary representation of (one less than) 10 to the power 10^{14}, so using double precision floats (which are close to having 16 decimal digits of precision) is largely enough to cover real-life cases. Nevertheless, in the `0.9d` code comments we briefly describe how we could proceed all the way using only integer arithmetic with no theoretical limit on input size. See subsubsection 8.4.2. Similar remarks apply to Zeckendorf representations.

equal to k. (So we can always find the location of the radix separator if we had lost it).

Christiane Frougny and **Jacques Sakarovitch** ([9]) showed that there exists a (non explicited) finite two-tape automaton which converts the Zeckendorf expansion of a positive integer into the Bergman representation (where the part with negative exponents is ``folded'' across the radix point to sit on top (or below) the part with positive exponents). Very recently **Jeffrey Shallit** ([10]) has revisited this topic and constructed explicitly a Frougny-Sakarovitch automaton.

References

[1] C. G. Lekkerkerker. Voorstelling van natuurlijke getallen door een som van getallen van Fibonacci. *Simon Stevin*, 29:190--195, 1951-1952.

[2] E. Zeckendorf. Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas. *Bull. Soc. Roy. Sci. Liège*, 41:179--182, 1972.

[3] Donald E. Knuth. Fibonacci multiplication. *Appl. Math. Lett.*, 1(1):57--60, 1988.

[4] Pierre Arnoux. Some remarks about Fibonacci multiplication. *Appl. Math. Lett.*, 2(4):319--320, 1989.

[5] H. Porta and K. B. Stolarsky. The edge of a golden semigroup. In *Number theory, Vol. I (Budapest, 1987)*, volume 51 of *Colloq. Math. Soc. János Bolyai*, pages 465--471. North-Holland, Amsterdam, 1990.

[6] Donald E. Knuth. *The art of computer programming. Vol. 4A. Combinatorial algorithms. Part 1*. Addison-Wesley, Upper Saddle River, NJ, 2011.

[7] George Bergman. A number system with an irrational base. *Math. Mag.*, 31:98--110, 1957/58.

[8] Donald E. Knuth. *The art of computer programming. Vol. 1*. Addison-Wesley, Reading, MA, third edition, 1997. Fundamental algorithms.

[9] Christiane Frougny and Jacques Sakarovitch. Automatic conversion from Fibonacci representation to representation in base ϕ, and a generalization. volume 9, pages 351--384. 1999. Dedicated to the memory of Marcel-Paul Schützenberger.

[10] Jeffrey Shallit. Proving properties of φ-representations with the Walnut theorem-prover. *Commun. Math.*, 33(2):Paper No. 3, 33, 2025.

Part I.

User manual

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. The core package features

3.1. Algebra in Q(ϕ), extensions to the `\xinteval` syntax

The `\xinteval` syntax is extended in the following manner:

1. Bracketed pairs `[a, b]` represent a+bϕ, where ϕ is the golden ratio, and one can operate on them with `+`, `-` (also as prefix unary operator), `*`, `/`, and `^` (or `**`) to do additions, subtractions, multiplications, divisions and powers with integer exponents.

   So `a` and `b` can be rational numbers and are not limited to integers for these computations.

   `phi` stands for `[0,1]` and its conjugate `psi = [1, -1]` is defined also. One can use on input `a + b phi`, which on output will be printed as `[a, b]`.

   DO NOT USE `\phi` OR `\psi`... except if redefined to expand to the letters `phi` and `psi` but this not recommended...!

   ```
   \xinteval{phi^50, psi^50, phi^50 * psi^50}
   [7778742049, 12586269025], [20365011074, -12586269025], [1, 0]
   ```

   ```
   \xinteval{(1+phi)(10-7phi)(3+phi)/(2+phi)^3}
   [87/25, -59/25]
   ```

   ```
   \xinteval{add(phi^n, n = -4,-7,-10, 1, 3, 6, 9)}
   [100, 0]
   ```

   ```
   \xinteval{phi^20 / phi^10}
   [34, 55]
   ```

   TeX-nical note: When dividing, and except if both operands are scalars, the coefficients of the result are reduced to their smallest terms; but for scalar-only division, one needs to use the `reduce()` function explicitly.

   The `[0, 0]` acts as `0` in operations, but is not automatically replaced by it, if produced by a subtraction for example. It is not allowed as an exponent for powers.

2. The functions `phisign()`, `phiabs()`, `phinorm()`, `phiconj()` do what one expects.

> **Attention:** `\xinteval` functions are always used with parentheses, not with curly braces, contrarily to macros!

```
\xinteval{phisign(10000 - 6180 phi)}
1;
```

```
\xinteval{phisign(10000 - 6181 phi)}
-1;
```

```
\xinteval{phiabs(10000 - 6181 phi)}
[-10000, 6181]
```

```
\xinteval{phinorm(10000 - 6180 phi)}
7600
```

```
\xinteval{(10000 - 6180 phi) * phiconj(10000 - 6180 phi)}
[7600, 0]
```

3. The function `fib()` computes the Fibonacci numbers (also for negative indices), and `fibseq(a,b)` will compute a consecutive stretch of them from index `a` to index `b` (one may also have `b=a`, or `b<a`).

```
\xinteval{seq(fib(n), n=-5..5, 10, 20, 100)}
5, -3, 2, -1, 1, 0, 1, 1, 2, 3, 5, 55, 6765, 354224848179261915075
```

```
\xinteval{seq(fib(2^n), n=1..7)}
1, 3, 21, 987, 2178309, 10610209857723, 251728825683549488150424261
```

TeX-nical note: In the next example, `\xintFor` expands only once, but `\xinteval` needs two expansion steps so we use `\expanded` wrapper. We could have used `\xintFor*` but then we need `\xintCSVtoList` wrapper. We also could have used some `` \romannumeral-`0 `` prefix but I figured `\expanded` looked less scary. For details on `\xintFor`/`\xintFor*` check the `xinttools` documentation.

```
\xintFor #1 in {\expanded{\xinteval{*fibseq(100, 110)}}}%
    \do{#1\xintifForLast{.\par}{,\newline}}
354224848179261915075,
573147844013817084101,
927372692193078999176,
1500520536206896083277,
2427893228399975082453,
3928413764606871165730,
6356306993006846248183,
10284720757613717413913,
16641027750620563662096,
26925748508234281076009,
43566776258854844738105.
```

In the previous example, note the syntax `*fibseq(100,110)`. Indeed `fibseq(a,b)` produces a `nutple` (see `xintexpr` documentation), i.e. the output will display brackets `[...]` (even if `a=b`):

```
\xinteval{fibseq(20, 25)}
[6765, 10946, 17711, 28657, 46368, 75025]
```

With the `*` prefix the brackets are removed.

4. The `zeckindices()` function computes the indices needed for the Zeckendorf representation. The input must be an integer. If negative, it is replaced by its opposite. The zero input gives an empty output (i.e. is printed as `[]`).

   ```
   \xinteval{zeckindices(123456789)}
   [40, 36, 34, 28, 26, 24, 18, 16, 13, 7, 5, 2]
   ```

   We use the `*` prefix to not have brackets in the output.

   ```
   \xinteval{*zeckindices(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
   ```

   ```
   \xinteval{*zeckindices(1e40)}
   193, 186, 176, 174, 167, 163, 161, 159, 157, 153, 150, 147, 145, 143,
   141, 139, 136, 134, 130, 126, 119, 115, 113, 110, 108, 106, 103, 101,
   98, 95, 93, 91, 89, 86, 83, 78, 73, 67, 65, 63, 60, 57, 55, 52, 50, 47,
   45, 39, 37, 32, 28, 23, 21, 19, 16, 13, 5
   ```

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

   ```
   \xinteval{*reversed(zeckindices(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 `\xinteval`, always eager to prove `A=A`, can be demonstrated:

   ```
   \xinteval{add(fib(n), n = *zeckindices(123456789))}
   123456789
   ```

   ```
   \xinteval{add(fib(n), n = *zeckindices(123456789123456789123456))}
   123456789123456789123456
   ```

5. the `$` is added as infix operator on *positive* integers (it will error if used with non-positive integers), to compute the Knuth Fibonacci multiplication. It does it using the Arnoux formula (7). The `$$` does the same but using the original Knuth formula (5).

   For examples see subsubsection 3.4.1.

6. The `phiexponents()` function computes the exponents in the Bergman ϕ-representation of its input. This input must be either an integer or a bracketed pair `[a,b]` or equivalently `a + b phi`, standing for $a+b\phi$ with a and b relative integers. It $a+b\phi<0$ it is replaced by its opposite. The output is the empty nutple `[]` if input is zero. Non-integer input is truncated to integers.

   The `phiexponents()` function produces a bracketed list.

```
\xinteval{phiexponents(100)}\newline
\xinteval{phiexponents(100  - 50phi)}\newline
\xinteval{phiexponents(-100 + 50phi)}\newline
\xinteval{phiexponents(100  - 50psi)}
```

```
[9, 6, 3, 1, -4, -7, -10]
[6, 0, -4, -10]
[6, 0, -4, -10]
[10, 4, 0, -6]
```

We can use `*` prefix as already indicated if we prefer not to see the brackets:

```
\xinteval{*phiexponents(3141592653)}
```

```
45, 42, 31, 29, 27, 25, 21, 18, 6, 1, -2, -6, -19, -23, -32, -43, -46
```

The added `\xinteval` syntax elements are also sometimes examplified alongside their respective matching macros. Not all macros defined by the package are documented, because documentation takes incredible amount of times and induces costly maintenance. See the commented source code.

Important

The added syntax elements are only defined for `\xinteval`. It is possible though to access them inside of `\xintfloateval` using the lower-level `\xintexpr`. Here is an example:

```
\xintfloateval{\xintexpr fib(100) / fib(99)\relax}
```

```
1.618033988749895
```

The variables `phi` and `psi` can not be used for operations directly inside of `\xintfloateval`. And they should not be redefined as floating point variables, as this would break their usage in `\xinteval`. But one can transfer computations after having defined first an auxiliary `\xintfloateval` function:

```
\xintdeffloatfunc phi_to_fp(x):= x[0] + (1+sqrt(5))/2 * x[1];
```

Then one can use it this way:

```
\xintfloateval{phi_to_fp(\xintexpr (1+phi)(1+2phi)(1-3phi)\relax)}
```

```
-42.74264578624801
```

3.2. Fibonacci numbers

3.2.1. `\ZeckTheFN`

This macro computes Fibonacci numbers.

```
\ZeckTheFN{100}
```

```
354224848179261915075
```

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

```
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
```

fib()

The syntax of \xinteval is extended via addition of a fib() function, which gives a convenient interface. See its documentation in subsection 3.1.

3.2.2. \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.

fibseq()

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

```
\xinteval{fibseq(10,20)}
```

```
[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:

```
\xinteval{*fibseq(-10,-20)}
```

```
-55, 89, -144, 233, -377, 610, -987, 1597, -2584, 4181, -6765
```

3.3. Zeckendorf representation

3.3.1. \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 \xinteval. A negative input will be replaced by its absolute value. A vanishing input gives an empty output.

The macro is also known as \ZeckZeck.

```
\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
```

zeckindices()

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

3.3.2. \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 \xinteval.

A zero input gives an empty output and a negative input is replaced by its absolute value.

```
\ZeckWord{123456789}
```

```
1000101000000101010000001010010000010100 1
```

```
\ZeckWord{\xinteval{2^40}}
```

```
100010000000100000101000001010101010101000100010101010100010
```

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 obtain this:

```
\ZeckWord{\xinteval{2^100}}
```

```
  1010000010010010101010100100000000100100100101010001010010000100100101↩
0010000000010100001001010101000000101001000100000000010010010001001001 01↩
00
```

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.

3.3.3. \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
```

Emulation inside \xinteval

There is no associated \xinteval function but the functionality is a one-liner in its syntax:

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

```
357128524055170099155
```

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

Emulation inside \xinteval (cont)

```
46
```

It is even a one-liner to define by oneself the missing function:

```
\xintdeffunc Nfromlist(x):= add(fib(i), i = *x);
\xinteval{Nfromlist([100, 90, 80, 70, 60, 50, 40, 30])}\\
\xinteval{Nfromlist(zeckindices(1e60))}
```

```
35712852405517009155
1000000000000000000000000000000000000000000000000000000000000
```

3.3.4. \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
```

3.4. Knuth Fibonacci Multiplication

3.4.1. \ZeckKMul, \ZeckAMul

Both compute the Knuth multiplication of its two **positive** integer arguments. The former, using formula (5), the latter using (7). The two arguments are only *f*-expanded, you need to wrap each in an \xinteval if it is an expression.

```
\ZeckKMul{100}{200}, \ZeckAMul{100}{200}
```

```
44800, 44800
```

```
\ZeckKMul{\ZeckKMul{100}{200}}{300},
\ZeckAMul{\ZeckKMul{100}{200}}{300}
```

```
30079200, 30079200
```

```
\ZeckKMul{100}{\ZeckKMul{200}{300}},
\ZeckAMul{100}{\ZeckKMul{200}{300}}
```

```
30079200, 30079200
```

$, $$

The syntax of \xinteval is extended via addition of a $ infix operator computing according to the Arnoux formula (7), and $$ computing according to the Knuth formula (5).

```
\xinteval{(100 $ 200) $ 300, 100 $ (200 $ 300), 100 $$ 200 $$ 300}
```

```
30079200, 30079200, 30079200
```

Let us mention here that we could have defined a knuth() function easily using the powerful \xinteval syntax:

```
\xintNewFunction{knuth}[2]
```

```
   {add(fib(x), x = flat(ndmap(+, *zeckindices(#1); *zeckindices(#2);)))}
\xinteval{knuth(100,200), knuth(knuth(100,200),300),
                          knuth(100,knuth(200,300))}
```

```
44800, 30079200, 30079200
```

TEX-nical note: We could not have used `\xintdeffunc` 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 `\xinteval`.

About using `ndmap()` with `+` as first argument, it is related to xintexpr having defined a `` `+` `` function. So we can also use `*` here, but not `-` or `/`.

The advantage is that we have now the means to check the validity of Knuth's triple product formula (6):

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

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

3.4.2. \ZeckSetAsKnuthOp, \ZeckSetAsArnouxOp

This takes as input a character, or multiple characters, and turns them (as a unit) into an infix operator computing the Knuth multiplication, respectively according to the original Knuth definition (5) or to the Arnoux formula (7). The pre-defined meanings of `$` or `$$` for this will not be canceled. One may use `\ZeckDeleteOperator{⟨operator⟩}` to delete the existing meaning of an `\xinteval` operator.

IMPORTANT

There is NO WARNING if you override a pre-existing operator from the `\xinteval` 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.

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:

  ```
  \ZeckSetAsArnouxOp{o}
  \xinteval{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:

  ```
  \ZeckSetAsArnoux{⊙}
  \xinteval{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 non-letter characters. It can not be the period (full stop) which, although not being a predefined operator is recognized as decimal separator.
- In case your document is compiled with `pdflatex` or `latex` and uses Babel, some characters may be catcode active. To make them part of a name of an operator defined by `\ZeckSetAsKnuthOp`, each such catcode active character has to be prefixed with `\string` in the argument of `\ZeckSetAsKnuthOp`. But `\string` is then *unneeded* inside `\xinteval` (since xintexpr 1.4n).

3.5. Bergman phi-representation

3.5.1. \PhiExponents

It has a unique mandatory argument which can be (or expand too) either an integer a, or two braced integers {a}{b}.

It outputs the comma separated list of the exponents from the minimal Bergman representation of the absolute value of $a+b\phi$. If $a+b\phi<0$, this list will be prefixed by a period. If $a=b=0$, the output is empty.

phiexponents()

The syntax of `\xinteval` is extended via addition of a `phiexponents()` function, which gives a more convenient interface.

Contrarily to the macro, it loses the information about the sign of the input and tacitly replaces it with its absolute value.

See subsection 3.1 for examples.

```
\[100\rightarrow \PhiExponents{100}\]
```

$$100 \rightarrow 9, 6, 3, 1, -4, -7, -10$$

```
\[100\phi\rightarrow \PhiExponents{{0}{100}}\]
```

$$100\phi \rightarrow 10, 7, 4, 2, -3, -6, -9$$

```
\[1000000\rightarrow \PhiExponents{1000000}\]
```

$$1000000 \rightarrow 28, 26, 20, 16, 13, 8, 4, 0, -4, -9, -11, -14, -16, -20, -26, -28$$

```
$10^{20}\rightarrow{}$ \PhiExponents{\xinteval{10^20}}.
```

$10^{20} \rightarrow$ 95, 93, 86, 84, 67, 65, 62, 60, 45, 41, 38, 31, 28, 23, 21, 16, 12, 6, 4, -4, -6, -12, -17, -19, -24, -29, -32, -39, -43, -46, -60, -63, -68, -84, -87, -89, -91, -96.

We did not use math mode for the longer output, because TeX needs extra instructions to wrap the line. But the separator can be customized to this aim:

```
\renewcommand\PhiExponentsSep
     {,\allowbreak\hskip0pt plus 1pt\relax}
$10^{50}\rightarrow \PhiExponents{\xinteval{10^50}}$.
```

$10^{50} \rightarrow$ 239, 234, 232, 226, 223, 219, 217, 212, 205, 202, 200, 196, 192, 189, 186, 177, 173, 169, 165, 161, 159, 152, 149, 146, 144, 138, 131, 129, 127, 123, 120, 116, 114, 109, 107, 105, 103, 100, 98, 96, 94, 88, 86, 84, 82, 79, 76, 74, 72, 65, 63, 61, 57, 55, 53, 48, 41, 35, 33, 30, 28, 26, 22, 16, 14, 12, 9, 6, 4, 2, -2, -4, -7, -10, -12, -14, -16, -22, -26, -28, -31, -37, -39, -42, -49, -51, -59, -66, -72, -74, -77, -80, -82, -84, -86, -88, -94, -96, -98, -101, -110, -114, -116, -121, -125, -132, -138, -144, -147, -150, -153, -155, -157, -163, -167, -171, -175, -178, -187, -190, -192, -196, -200, -203, -206, -213, -215, -221, -224, -226, -232, -235, -237, -240.

The attentive reader will have noticed though that our math mode does not differ much from our nice monospace text mode. Maybe look at some other LaTeX package by the author to find some clues explaining this top-quality typesetting.

3.5.2. \PhiBasePhi

It has a unique mandatory argument which can be (or expand two) either an integer `a`, or two braced integers `{a}{b}`.

It computes the Bergman ϕ-representation of $x = a + b\phi$ if x turns out to be positive, outputs `0` if both a and b vanish, and outputs a minus sign followed with the expansion of the opposite of x if $x < 0$.

The output for positive x is a sequence of 1's and 0's with possibly a period as radix separator (it can be customized, see next), which either starts with a leading `1` or with zero followed by the radix separator `0.`. It always ends with a `1`. No `1` is repeated.

The arguments are x-expanded, if you need them to be expressions you must wrap them using `\xinteval`.

A little stress-test:

```
\begin{multicols}{2}
\xintFor* #1 in {\xintSeq[-1]{20}{-21}}\do{%
    $\phi^{#1}=\PhiBasePhi{{\ZeckTheFN{#1-1}}%
                             {\ZeckTheFN{#1}}}_\phi$\par
}
\end{multicols}
```

$\phi^{20} = 100000000000000000000_\phi$

$\phi^{19} = 10000000000000000000_\phi$

$\phi^{18} = 1000000000000000000_\phi$

$\phi^{17} = 100000000000000000_\phi$

$\phi^{16} = 10000000000000000_\phi$

$\phi^{15} = 1000000000000000_\phi$

$\phi^{14} = 100000000000000_\phi$

$\phi^{13} = 10000000000000_\phi$

$$\phi^{12} = \mathtt{1000000000000}_\phi$$
$$\phi^{11} = \mathtt{100000000000}_\phi$$
$$\phi^{10} = \mathtt{10000000000}_\phi$$
$$\phi^{9} = \mathtt{1000000000}_\phi$$
$$\phi^{8} = \mathtt{100000000}_\phi$$
$$\phi^{7} = \mathtt{10000000}_\phi$$
$$\phi^{6} = \mathtt{1000000}_\phi$$
$$\phi^{5} = \mathtt{100000}_\phi$$
$$\phi^{4} = \mathtt{10000}_\phi$$
$$\phi^{3} = \mathtt{1000}_\phi$$
$$\phi^{2} = \mathtt{100}_\phi$$
$$\phi^{1} = \mathtt{10}_\phi$$
$$\phi^{0} = \mathtt{1}_\phi$$
$$\phi^{-1} = \mathtt{0.1}_\phi$$
$$\phi^{-2} = \mathtt{0.01}_\phi$$
$$\phi^{-3} = \mathtt{0.001}_\phi$$
$$\phi^{-4} = \mathtt{0.0001}_\phi$$

$$\phi^{-5} = \mathtt{0.00001}_\phi$$
$$\phi^{-6} = \mathtt{0.000001}_\phi$$
$$\phi^{-7} = \mathtt{0.0000001}_\phi$$
$$\phi^{-8} = \mathtt{0.00000001}_\phi$$
$$\phi^{-9} = \mathtt{0.000000001}_\phi$$
$$\phi^{-10} = \mathtt{0.0000000001}_\phi$$
$$\phi^{-11} = \mathtt{0.00000000001}_\phi$$
$$\phi^{-12} = \mathtt{0.000000000001}_\phi$$
$$\phi^{-13} = \mathtt{0.0000000000001}_\phi$$
$$\phi^{-14} = \mathtt{0.00000000000001}_\phi$$
$$\phi^{-15} = \mathtt{0.000000000000001}_\phi$$
$$\phi^{-16} = \mathtt{0.0000000000000001}_\phi$$
$$\phi^{-17} = \mathtt{0.00000000000000001}_\phi$$
$$\phi^{-18} = \mathtt{0.000000000000000001}_\phi$$
$$\phi^{-19} = \mathtt{0.0000000000000000001}_\phi$$
$$\phi^{-20} = \mathtt{0.00000000000000000001}_\phi$$
$$\phi^{-21} = \mathtt{0.000000000000000000001}_\phi$$

The radix separator is customizable as \PhiBasePhiSep. It defaults to a period `.`. If, for example, you use the package numprint, you could do

```
\makeatletter\renewommand{\PhiBasePhiSep}{\nprt@decimal}\makeatother
```

for the output to use the radix separator as set via \npdecimalsign command. It is easier to simply wrap usage of \PhiBasePhi inside of \numprint (or shortcut \np) command of that package.

TEX-nical note: For fancy set-ups, for example for a radix separator using color, it is recommended to use \RenewDocumentCommand. Indeed, it turns out that \PhiBasePhi will trigger an expansion context, and the radix separator has to be compatible with it.

```
% or \protected\def if not using LaTeX
\RenewDocumentCommand\PhiBasePhiSep{}{{\mathcolor{blue}{.}}}
\begin{gather*}
\xintFor #1 in {1, 10,  100, 1000, 10000, 100000, 1000000}\do{%
#1 = \PhiBasePhi{#1}_\phi \xintifForLast{}{\\}}
\end{gather*}
```

$$1 = \mathtt{1}_\phi$$
$$10 = \mathtt{10100.0101}_\phi$$
$$100 = \mathtt{1001001010.0001001001}_\phi$$
$$1000 = \mathtt{100010010001000.10001001010001}_\phi$$
$$10000 = \mathtt{100000100100000010000.00010000001000101001}_\phi$$
$$100000 = \mathtt{1010100010101000000100000.1010001010000000010000001}_\phi$$
$$1000000 = \mathtt{10100000100010010000100010001.00010000010100101000100000101}_\phi$$

Tip

As mentioned elsewhere, TeX and LaTeX will not per default split long sequences of zeros and ones at ends of lines. 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:

```
$10^{50}-10^{50}\phi = {}$\printnumber{%
   \PhiBasePhi{{\xinteval{10^50}}{\xinteval{-10^50}}}}${}_\phi$.
```

$10^{50} - 10^{50}\phi$ = -1000010100000010010001010000010000000100101000100010010010
01000000010001000100010001010000000100100101000000100000001010100010010 00
10100000101010100101010100000010101010010010101000000010101000101010000
100000010000010100101010001000001010100100101010.0010100100101010100
00010001010010000010100100000010100000001000000100000101001001010101
01000000101010010000000001000101000010001000000010000001000001001001 0010
1010000001000100010001001000000000100101000100010010010000000101000001 0
01010000000100101001$_\phi$.

The radix separator is somewhere in the middle.

TeX-nical note: Maybe this `10^{50}` gives opportunity to stress the following: in `\xinteval`, braces `{...}` are removed, so for example `2^{10-1}` is same as `2^10-1` and not at all `2^(10-1)`. It is easy to forget this when doing both TeX typesetting and `\xinteval` calculations at the same time. By the way, `2^-50` is accepted syntax in `\xinteval`, parentheses as in `2^(-50)` are not mandatory.

3.5.3. \PhiXfromExponents

This computes a ϕ-integer from an arbitrary list of (comma separated) exponents, not necessarily ordered. The output `{a}{b}` is not destined for direct typesetting, one needs for this to wrap usage of the macro inside of `\PhiTypesetX`.

The list input is allowed to be empty. A period upfront the input signals to change the sign of the output to its opposite.

TeX-nical note: When using the interactive interface, such a leading period in the list of exponents will be produced on output from an `a+b phi` if $a + b\phi < 0$, but when converting in the other direction, from a list of exponents to some `a+b phi`, a leading period will cause the first exponent to be replaced with zero, if it is non-negative, and will cause a crash if the first listed exponent is negative.

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.

```
$\PhiTypesetX{\PhiXfromExponents{}}$\newline
$\PhiTypesetX{\PhiXfromExponents{100, 49}}$\newline
$\PhiTypesetX{\PhiXfromExponents{15, 13, 10, 5, 3, 1, -6, -11, -16}}$\\
$\PhiTypesetX{\PhiXfromExponents{.16, 14, 11, 6, 4, 2, -5, -10, -15}}$
```

0

$218922995834555169026 + 354224848179261915075\phi$

2025

-2025ϕ

For these next two examples:

```
$\PhiTypesetX{\PhiXfromExponents{\PhiExponents{1000}}}$\newline
$\PhiTypesetX{\PhiXfromExponents{\PhiExponents{{1000}{-1000}}}}$
```

We have to be careful that we previously customized `\PhiExponentsSep` like this:

```
\renewcommand\PhiExponentsSep
     {,\allowbreak\hskip0pt plus 1pt\relax}
```

But this will break `\PhiXfromExponents` because it really needs its argument, after expansion, to be a genuine comma separated list (possibly with extra spaces, they do not matter). So we now reset `\PhiExponentsSep` to its default, and we can execute successfully the next instructions confirming this package is excellent at doing nothing:

```
\renewcommand\PhiExponentsSep{, }%
$\PhiTypesetX{\PhiXfromExponents{\PhiExponents{1000}}}$\newline
$\PhiTypesetX{\PhiXfromExponents{\PhiExponents{{1000}{-1000}}}}$
```

1000
$1000 - 1000\phi$

Emulation inside \xinteval

There is no associated `\xinteval` function but the functionality is a one-(or-two)-liner in its syntax:

```
\xinteval{[add(fib(i-1), i=100, 50, -40, -80),
           add(fib(i),   i=100, 50, -40, -80)]}
```

```
[218960884904872635122, 354201431463397382260]
```

Compare with:

```
$\PhiTypesetX{\PhiXfromExponents{100, 50, -40, -80}}$
```

$218960884904872635122 + 354201431463397382260\phi$

It is even a one-or-two-liner to define a function all by oneself!

```
\xintdeffunc ABfromlist(x):=
             [add(fib(i-1),i=*x), add(fib(i),i=*x)];
\xinteval{ABfromlist(phiexponents(10^30))}
```

```
[1000000000000000000000000000000, 0]
```

3.5.4. \PhiXfromBasePhi

This computes an element from $\mathbf{Z}[\phi]$ from a Bergman ϕ-representation. The input is allowed to be empty. If it contains a radix separator, it must be a period, and that period must be preceded by at least one `0` or `1`. It is allowed for `1`'s to be consecutive. A leading minus sign is allowed.

The output `{a}{b}` is not destined for direct typesetting one needs to wrap it inside of `\PhiTypesetX`.

```
\edef\x{\PhiXfromBasePhi{}, \PhiXfromBasePhi{0}, \PhiXfromBasePhi{-0}}
\meaning\x\newline
$\PhiTypesetX{\PhiXfromBasePhi{10.01}}$\newline
$\PhiTypesetX{\PhiXfromBasePhi{101000100010.001000100001}}$\newline
$\PhiTypesetX{\PhiXfromBasePhi{-101000100010.0010001000011}}$
```

```
macro:->{0}{0}, {0}{0}, {0}{0}
2
```

288
89 - 233ϕ

3.6. Typesetting

3.6.1. `\ZeckPrintIndexedSum`

This is a typesetting utility which produces (expandably), by default, `F_a + F_{a'} + ...` from `a, a', ...`.

```
$\ZeckPrintIndexedSum{\ZeckIndices{1000000000000000000}}$.
```

$F_{92} + F_{89} + F_{87} + F_{64} + F_{62} + F_{57} + F_{54} + F_{51} + F_{48} + F_{45} + F_{43} + F_{41} + F_{38} + F_{35} + F_{32} + F_{30} + F_{27} + F_{22} + F_{20} + F_{16} + F_{14} + F_9 + F_7$.

The `+` is injected by `\ZeckPrintIndexedSumSep` whose default definition is:

```
\def\ZeckPrintIndexedSumSep{+\allowbreak}
```

Each index from the input list is given as argument to `\ZeckPrintOne` whose default definition requires math mode:

```
\def\ZeckPrintOne#1{F_{#1}}
```

If one wants explicit Fibonacci numbers, one can do this:

```
$\def\ZeckPrintOne{\ZeckTheFN}
 \ZeckPrintIndexedSum{\ZeckIndices{1000000000000000000}}$.
```

```
7540113804746346429+1779979416004714189+679891637638612258+10610209857723+
4052739537881+365435296162+86267571272+20365011074+4807526976+1134903170+
433494437 + 165580141 + 39088169 + 9227465 + 2178309 + 832040 + 196418 + 17711 +
6765 + 987 + 377 + 34 + 13.
```

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 two lines are squeezed, but still overflow, which is not pleasing.

With the help of a xinttools utility we can redefine `\ZeckPrintOne` to inject breakpoints in-between consecutive digits:

```
\renewcommand\ZeckPrintOne[1]
   {\xintListWithSep{\allowbreak}{\ZeckTheFN{#1}}}
$\ZeckPrintIndexedSum{\ZeckIndices{1000000000000000000}}$.
```

```
7540113804746346429 + 1779979416004714189 + 679891637638612258 + 10610209857
723 + 4052739537881 + 365435296162 + 86267571272 + 20365011074 + 4807526976 + 1
134903170 + 433494437 + 165580141 + 39088169 + 9227465 + 2178309 + 832040 + 19641
8 + 17711 + 6765 + 987 + 377 + 34 + 13.
```

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

```
7540113804746346429 + 1779979416004714189 + 679891637638612258 + 1061020985↲
7723 + 4052739537881 + 365435296162 + 86267571272 + 20365011074 + 4807526976 + ↲
1134903170 + 433494437 + 165580141 + 39088169 + 9227465 + 2178309 + 832040 + 1964↲
18 + 17711 + 6765 + 987 + 377 + 34 + 13.
```

3.6.2. `\PhiPrintIndexedSum`

It is actually a clone of `\ZeckPrintIndexedSum` which only differs from it via separate configuration macros:

```
\def\PhiPrintIndexedSumSep{+\allowbreak}% same as \ZeckPrintIndexedSumSep
\def\PhiPrintOne#1{\phi^{#1}}% powers of phi rather than Fibonacci
```

```
                    % numbers.
```

As for `\ZeckPrintIndexedSum` the default configuration is thus math mode only.

```
\[2025 = \PhiPrintIndexedSum{\PhiExponents{2025}}\]
```

$$2025 = \phi^{15} + \phi^{13} + \phi^{10} + \phi^{5} + \phi^{3} + \phi^{1} + \phi^{-6} + \phi^{-11} + \phi^{-16}$$

It is important in the above example that `\PhiExponentsSep` has its default definition because `\PhiPrintIndexedSum` needs to see real commas.

TODO?

Maybe let it recognize an upfront period (as produced by `\PhiExponents` if $x = a + b\phi < 0$) in the input, and then use minus signs in the output?

3.6.3. \PhiTypesetX

This is supposed to receive as (single) argument two braced relative integers `{a}{b}`.

The default output is math-mode only, as it is of the type $a + b\phi$, with simplifications for zero or negative coefficients. This is decided by the two-arguments macro `\PhiTypesetXPrint` which can be redefined.

For examples, see the documentation of `\PhiXfromExponents` and `\PhiXfromBasePhi`.

4. Use as a LaTeX package

As expected, add to the preamble:

```
\usepackage{zeckendorf}
```

There are no options.

5. 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.

6. Changes

0.9d (2025/11/16) **Breaking changes:**

- The radix separator used by default by `\PhiBasePhi` on output and expected by `\PhiXfromBasePhi` on input is now a period, not a comma (the comma was used by accident by the author due to patriotism).
- `\ZeckIndices` and `\ZeckWord` both replace a negative argument by its absolute value, rather than returning an empty output as so far.

New feature: Macros to support Zeckendorf and Bergman representations using hexadecimal digits. For lack of time, the PDF doc is not updated, please use the interactive interface or check the source code for the macro names. Thanks to **Laurent Barme** for the feature request.

Bug fix: After the `0.9c` transition from `\xintiieval` to `\xinteval`, the `$` and `$$` infix operators were broken with operands such as `(2+3)` in place of lone integers.

Improvements:

- The `fibseq(a,b)` function can now be used also with `a<b` and `a=b`.
- Under-the-hood polishing for efficiency, improved code comments and documentation of the main algorithms.

0.9c (2025/10/17) This adds many new features and has some breaking changes due to renamings, not listed here.

- It is not `\xintiieval` but `\xinteval`'s syntax which is now extended.
- Variables `phi` and `psi` are defined and one can do algebra with `+`, `-`, `*`, `/` and `^` on them in $\mathbf{Q}(\phi)$.
- The Bergman ϕ-representation is added for elements of $\mathbf{Z}[\phi]$ in particular for integers.
- The `$` character doing the Knuth Fibonacci multiplication on positive integers now uses the (more efficient) Arnoux formula. The `$$` computes out of deference according to the original Knuth definition.
- The PDF documentation section on the mathematical background has been extended and includes bibliographical references.
- The interactive interface integrates all novelties.

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). *This remark is obsolete as of 0.9c because the interactive mode now uses `\xinteval`, not `\xintiieval`.*

- 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.

7. License

```
Copyright (c) 2025 Jean-François Burnol

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

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

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

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

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

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

    zeckendorf.dtx,
    zeckendorfcore.tex, zeckendorf.tex, zeckendorf.sty,
    README.md, zeckendorf-doc.tex, zeckendorf-doc.pdf
```

Part II.
Commented source code

8. Core code

Extracts to zeckendorfcore.tex.

A general remark is that expandable macros (usually) *f*-expand their arguments, and most are *f*-expandable. Usually, the CamelCase macro (with neither @ nor _ in their names) expands to either \romannumeral0 or \expanded followed with a lowercase macro. Macros destined to be used in typesetting context usually omit any such construct and may require *x*-expansion. They remain fully expandable as long as some user level customization (for example for the radix separator) has not injected things not compatible with an \edef.

For variety we use here sometimes @ in macro names, whereas xint uses only _ (and sometimes some other a priori non-letter characters).

8.1. Loading xintexpr and setting catcodes

```
\input xintexpr.sty
\input xintbinhex.sty
\wlog{Package: zeckendorfcore 2025/11/16 v0.9d (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, most macros using own custom terminators adapted to their own optimizations.

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

8.2. Fibonacci numbers

8.2.1. \ZeckTheFN

The multiplicative algorithm is as in the bnumexpr manual (at 1.7b), or since about ten years in the xint manual (at 1.4o or earlier) but termination is different and simply leaves {F_n}{F_{n-1}} in input stream. We do not use \csname...\endcsname branching here, for variety. Also, we replaced usage of chained expressions handled via \xintiiexpro with direct usage of the xintcore macros, for optimized efficiency, and taking into account that \expanded now helps doing this without intermediate step.

\Zeck@FPair and \Zeck@@FPair are not public interface. The former is allows a negative or zero argument, the latter is positive only.

```
\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{\expanded{{#2}{\XINT_Opp#1}}}%
\def\zeck@fpair_fii#1#2{\expanded{{\XINT_Opp#2}{#1}}}%
\def\Zeck@@FPair#1{%
    \expandafter\zeck@@fpair@start
    \romannumeral0\xintdectobin{\the\numexpr#1\relax};%
}%
```

Inlining here at start the \zeck@@fpair@again because we don't want the \expandafter's here, due to current \XINTfstop definition.

```
\def\zeck@@fpair@start 1#1{%
    \xint_gob_til_sc#1\zeck@@fpair@done;%
    \xint_UDzerofork
        #1\zeck@@fpair@zero
         0\zeck@@fpair@one
    \krof
    {1}{0}%
}%
```

Prior to 0.9d we were using coding like this, as it has been easier to use expressions (the xint documentation had such code for more than ten years, precisely to illustrate chaining of expressions, and at a time when \expanded was not available, and of course we simply took it over initially).

```
\romannumeral0\xintiiexpro (#1+2*#2)*#1\expandafter\relax\expandafter;%
\romannumeral0\xintiiexpro #1*#1+#2*#2\relax;%
```

or

```
\romannumeral0\xintiiexpro 2*(#1+#2)*#1+#2*#2\expandafter\relax\expandafter;%
\romannumeral0\xintiiexpro (#1+2*#2)*#1\relax;%
```

At 0.9d we go directly to the xintcore core macros for optimal efficiency. This required a few adjustments elsewhere and the removal from code comments of some technical discusions about \xintthe. The also dropped semi-colons as we are already using braces.

```
\def\zeck@@fpair@zero #1#2#3{%
    \zeck@@fpair@again#3%
    \expanded{%
      {\xintiiMul{#1}{\xintiiAdd{#1}{\xintDouble{#2}}}}%
      {\xintiiAdd{\xintiiSqr{#1}}{\xintiiSqr{#2}}}%
    }%
}%
\def\zeck@@fpair@one #1#2#3{%
    \zeck@@fpair@again#3%
    \expanded{%
      {\xintiiAdd{\xintDouble{\xintiiMul{\xintiiAdd{#1}{#2}}{#1}}}%
                  {\xintiiSqr{#2}}}%
      {\xintiiMul{#1}{\xintiiAdd{#1}{\xintDouble{#2}}}}%
    }%
}%
\def\zeck@@fpair@again#1{%
    \xint_gob_til_sc#1\zeck@@fpair@done;%
    \xint_UDzerofork
        #1{\expandafter\zeck@@fpair@zero}%
         0{\expandafter\zeck@@fpair@one}%
    \krof
}%
\def\zeck@@fpair@done#1\krof{}%
```

For individual Fibonacci numbers, we have non public \Zeck@@FN which only works on positive input and has a braced output. We also have non-public \Zeck@FN and \Zeck@FNminusOne which accept negative input, and whose output is also braced. And we have public \ZeckTheFN which accepts negative input and whose output is not braced.

The reason for strange name \ZeckTheFN is that originally \Zeck@FPair produced its output using a special xintexpr format, which needs to be prefixed with \xintthe to get resolved into only digits. We have now modified the structure by-pass this but the name sticks.

```
\def\zeck@bracedfirstoftwo  #1#2{{#1}}%
\def\zeck@bracedsecondoftwo #1#2{{#2}}%
\def\Zeck@FN{%
    \expandafter\zeck@bracedfirstoftwo\romannumeral`&&@\Zeck@FPair
}%
\def\Zeck@FNminusOne{%
    \expandafter\zeck@bracedsecondoftwo\romannumeral`&&@\Zeck@FPair
}%
\def\ZeckTheFN{\expandafter\xint_firstoftwo\romannumeral`&&@\Zeck@FPair}%
\def\Zeck@@FN{\expandafter\zeck@bracedfirstoftwo\romannumeral`&&@\Zeck@@FPair}%
```

8.2.2. \ZeckTheFSeq

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.

The two arguments may be negative, and since 0.9d they do not have to be ordered.

```
\def\ZeckTheFSeq#1#2{%
    \expanded\bgroup\expandafter\zeckthefseq_a
    \the\numexpr #1\expandafter.\the\numexpr #2.%
}%
\def\zeckthefseq_a#1.#2.{\expandafter\zeckthefseq_b\the\numexpr#2-#1.#1.}%
\def\zeckthefseq_b #1{%
    \xint_UDzerominusfork
     0#1\zeckthefseq_n
     #1-\zeckthefseq_one
     0-\zeckthefseq_p
    \krof #1%
}%
\def\zeckthefseq_one0.#1.{{\ZeckTheFN{#1}}\iffalse{\fi}}%
```

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 more to do.

```
\def\zeckthefseq_p #1.#2.{%
    \expandafter\zeckthefseq_loop
    \the\numexpr #1-1\expandafter.%
    \romannumeral`&&@\expandafter\zeck@sep@with@sc
    \romannumeral`&&@\Zeck@FPair{#2+1}\xintiiadd
}%
\def\zeck@sep@with@sc #1#2{#1;#2;}%
```

We will have F_{A-1};F_{A}; as starting point. We want down to F_B. If B=A-1 there will be nothing more to do.

```
\def\zeckthefseq_n -#1.#2.{%
    \expandafter\zeckthefseq_loop
    \the\numexpr #1-1\expandafter.%
    \romannumeral`&&@\expandafter\zeck@exch@with@sc
    \romannumeral`&&@\Zeck@FPair{#2}\xintiisub
}%
\def\zeck@exch@with@sc #1#2{#2;#1;}%
```

Now leave in stream one (braced) number, and test if we have reached B and until then apply standard Fibonacci recursion. This is all done using a single looping macro, only termination branches to another one.

We add a bit sub-optimality in having one single macro handling both increasing and decreasing indices.

```
\def\zeckthefseq_loop #1.#2;#3;#4{%
    {#3}\ifnum #1=\z@ \expandafter\zeckthefseq_end\fi
    \expandafter\zeckthefseq_loop\the\numexpr #1-1\expandafter.%
    \romannumeral0#4{#3}{#2};#2;#4%
}%
\def\zeckthefseq_end#1;#2;#3{{#2}\iffalse{\fi}}%
```

8.3. Zeckendorf representation

8.3.1. \ZeckNearIndex, \ZeckMaxK

If the ratio of logarithms $\log(\sqrt{5}x)/\log\phi$ 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 decimal digits of precision. So this assumption fails certainly for x having more than one hundred million decimal digits, and quite probably with an input having ten million digits, as we do not want to exceed $\phi^{10^7}/\sqrt{5} \approx 1.13 \times 10^{2,089,876}$. But with one million decimal digits we are safe (see subsubsection 8.4.2 for related comments).

As anyhow xint can handle multiplications only with operands of about up to 13000 digits (with TeXLive 2025 default memory settings), and computation times limit reasonable inputs to less than 1000 digits, there is no worry for us.

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 quite less than 100000 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. This is e.g. why we use \PoorManLogBaseTen_raw and not \PoorManLogBaseTen.

```
\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 now only for user interface, as we dropped at 0.9c adding a maxk() function to the \xinteval interface.

With 0.9d replace negative input by its absolute value.

```
\def\ZeckMaxK{\expanded\zeckmaxk}%
\def\zeckmaxk#1{\expandafter\zeckmaxk_fork\romannumeral`&&@#1\xint:}%
\def\zeckmaxk_fork#1{%
  \xint_UDzerominusfork
    #1-{\bgroup\zeck_abort}%
    0#1\zeckmaxk_a
    0-{\zeckmaxk_a#1}%
  \krof
}%
\def\zeckmaxk_a #1\xint:{%
    \expandafter\zeckmaxk_b
    \the\numexpr\ZeckNearIndex{#1}\xint:#1\xint:
}%
\def\zeckmaxk_b #1\xint:{%
    \expandafter\zeckmaxk_c
    \romannumeral`&&@\Zeck@@FPair{#1}#1\xint:
}%
\def\zeckmaxk_c #1#2#3\xint:#4\xint:{%
    \xintiiifGt{#1}{#4}%
       {{\expandafter}\the\numexpr#3-1\relax}%
       {{}#3}%
}%
```

8.3.2. \ZeckIndices

This starts by computing the maximal index. It then subtracts the Fibonacci number from the input and loops.

At 0.9d let it (rather than returning empty output) accept a negative argument (silently replaced by its absolute value).

```
\def\ZeckIndices{\expanded\zeckindices}%
\let\ZeckZeck\ZeckIndices
\def\zeckindices#1{\bgroup\expandafter\zeckindices_fork\romannumeral`&&@#1\xint:}%
\def\zeckindices_fork#1{%
  \xint_UDzerominusfork
    #1-\zeck_abort
    0#1\zeckindices_a
    0-{\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{#1}{#4}\zeckindices_A\zeckindices_B
    #1;#2;#3\xint:#4\xint:
}%
\def\zeckindices_A#1;#2;#3\xint:{%
    \the\numexpr#3-1\relax\zeckindices_loop{#2}%
}%
\def\zeckindices_B#1;#2;#3\xint:{%
    #3\zeckindices_loop{#1}%
}%
\def\zeckindices_loop #1#2\xint:{%
    \expandafter\zeckindices_loop_i
    \romannumeral0\xintiisub{#2}{#1}\xint:
}%
\def\zeckindices_loop_i#1{%
    \xint_UDzerofork#1\zeck_done 0{, \zeckindices_a#1}\krof
}%
```

8.3.3. \ZeckBList

This is the variant which produces the results as a sequence of braced indices.

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.

At 0.9d let it (rather than returning empty output) accept a negative argument (silently replaced by its absolute value).

```
\def\ZeckBList{\expanded\zeckblist}%
\def\zeckblist#1{\bgroup\expandafter\zeckblist_fork\romannumeral`&&@#1\xint:}%
\def\zeckblist_fork#1{%
  \xint_UDzerominusfork
    #1-\zeck_abort
    0#1\zeckblist_a
    0-{\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{#1}{#4}\zeckblist_A\zeckblist_B
    #1;#2;#3\xint:#4\xint:
}%
\def\zeckblist_A#1;#2;#3\xint:{%
    {\the\numexpr#3-1\relax}\zeckblist_loop{#2}%
}%
```

```
\def\zeckblist_B#1;#2;#3\xint:{%
    {#3}\zeckblist_loop{#1}%
}%
\def\zeckblist_loop#1#2\xint:{%
    \expandafter\zeckblist_loop_i
    \romannumeral0\xintiisub{#2}{#1}\xint:
}%
\def\zeckblist_loop_i#1{\xint_UDzerofork#1\zeck_done 0{\zeckblist_a#1}\krof}%
```

8.3.4. \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{\bgroup\expandafter\zeckword_fork\romannumeral`&&@#1\xint:}%
\def\zeckword_fork#1{%
  \xint_UDzerominusfork
    #1-\zeck_abort
    0#1\zeckword_a
    0-{\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{#1}{#4}\zeckword_A\zeckword_B
    #1;#2;#3\xint:#4\xint:
}%
\def\zeckword_A#1;#2;#3\xint:#4\xint:{%
    \expandafter\zeckword_d
    \romannumeral0\xintiisub{#4}{#2}\xint:
    \the\numexpr#3-1.%
}%
\def\zeckword_B#1;#2;#3\xint:#4\xint:{%
    \expandafter\zeckword_d
    \romannumeral0\xintiisub{#4}{#1}\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{#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.%
    \romannumeral0\xintiisub{#4}{#2}%
    \xint:
}%
\def\zeckword_gB#1;#2;#3\xint:#4\xint:{%
    \expandafter\zeckword_h
    \the\numexpr#3\expandafter.%
    \romannumeral0\xintiisub{#4}{#1}%
    \xint:
}%
\def\zeckword_h #1.#2\xint:#3.{%
    \xintReplicate{#3-#1-1}{0}%
    \zeckword_d #2\xint:#1.%
}%
```

8.3.5. \ZeckHexWord

```
\def\ZeckHexWord{\romannumeral0\zeckhexword}%
\def\zeckhexword#1{%
  \xintbintohex{%
    \expanded\bgroup\expandafter\zeckword_fork\romannumeral`&&@#1\xint:
  }%
}%
```

8.3.6. \ZeckNFromIndices

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

Extraneous commas are skipped, in particular a final comma is allowed.

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 is that can generate multiple indices from some macro expansion inserting commas dynamically.

```
\def\ZeckNFromIndices{\romannumeral0\zecknfromindices}%
\def\zecknfromindices{\zeck@applyandiisum\Zeck@FN}%
\def\zeck@applyandiisum {%
    \expandafter\xintiisum\expanded\zeck@applytocsv
}%
```

Macro #1 is assumed to output something within braces.

```
\def\zeck@applytocsv #1#2{%
    {{\expandafter\zeck@applytocsv_a\expandafter#1%
      \romannumeral`&&@#2,;}}%
}%
\def\zeck@applytocsv_a #1#2{%
    \if;#2\expandafter\zeck@applytocsv_done\fi
    \if,#2\expandafter\zeck@applytocsv_skip\fi
    \zeck@applytocsv_b #1#2%
}%
\def\zeck@applytocsv_b #1#2,{%
    #1{#2}%
    \expandafter\zeck@applytocsv_a\expandafter#1\romannumeral`&&@%
}%
\def\zeck@applytocsv_done#1\zeck@applytocsv_b#2;{}%
\def\zeck@applytocsv_skip #1#2,{%
    \expandafter\zeck@applytocsv_a\expandafter#2\romannumeral`&&@%
}%
```

8.3.7. \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\Zeck@@FN{#1}\fi
    \expandafter\zecknfromword_b\the\numexpr#1+1.%
}%
\def\zecknfromword_done#1.{\iffalse{{\fi}}}%
```

8.3.8. \ZeckNfromHexWord

Added at 0.9d.

```
\def\ZeckNfromHexWord{\romannumeral0\zecknfromhexword}%
\def\zecknfromhexword#1{%
    \expandafter\zecknfromword_a\romannumeral0\xintreversedigits{\xintHexToBin{#1}};%
}%
```

8.4. Bergman representation

8.4.1. \PhiIISign_ab

\PhiIISign_ab is for use with two already expanded arguments {a}{b} and which are strict integers.

The general macro which accepts both one (unbraced) integer or two (braced) integers is defined later for support of the phisign() function.

```
\def\PhiIISign_ab {\romannumeral0\phiiisign_ab}%
\def\phiiisign_ab #1#2{%
    \xintiiifsgn{#1}%
    {% a < 0
     \xintiiifSgn{#2}%
           {-1}%
           {-1}%
           {% a < 0, b > 0, return 1 iff a^2+ab<b^2
            \xintiiifLt{\xintiiMul{#1}{\xintiiAdd{#1}{#2}}}{\xintiiSqr{#2}}%
            {1}%
            {-1}%
           }%
    }%
    {\xintiiSgn{#2}}%
    {% a > 0
     \xintiiifSgn{#2}%
           {% a > 0, b < 0, return 1 iff a^2+ab>b^2
            \xintiiifGt{\xintiiMul{#1}{\xintiiAdd{#1}{#2}}}{\xintiiSqr{#2}}%
            {1}%
            {-1}%
           }%
           {1}%
           {1}%
    }%
}%
```

8.4.2. `\PhiMaxE`

We want the greatest k with $\phi^k \leq a+b\phi$, assuming that the latter is strictly positive. We will do this using a careful computation (i.e. avoiding ``catastrophic cancellations'' if a and b are of opposite signs) of the base-ten logarithm of $a+b\phi$, with about 8 decimal digits of precision. Rounding the quotient by $\log_{10}\phi$ to the nearest integer we obtain a candidate K. We compute ϕ^K using Fibonacci numbers and compare (using integer-only arithmetic). If this is larger than $a + b\phi$ the seeked k is K-1 else it is K. For why, see the explanations relative to the computation of the Zeckendorf representation and the next paragraph about theoretical limitation.

The rounding to an integer of $\log(a + b\phi)/\log(\phi)$ obtained with 8 decimal digits of precision will not error by 2 units or more if the input was less than ϕ^{10^7}, so for an x which is an integer having less than two million decimal digits, or say one million, this is safe. And xint can only do computations with operands having less than about 13000 digits (with TeXLive 2025 default memory settings). If using another programming context not having such limitations, and using rather double precision floats, which gives slightly less than 16 decimal digits of floating point precision, the upper bound would raise to about $\phi^{10^{15}}$, and inputs with at most 10^{14} decimal digits are safe, i.e. all real life inputs are safe.

Let us nevertheless explain how we could do without logarithms and without any upper bound on size of the input. Let $x = a + b\phi$, assumed to be positive. First, we test if $x < 1$. We can do this using integers only. If true, we multiply x by ϕ, ϕ^2, ϕ^4, using algebra in $\mathbf{Z}[\phi] = \mathbf{Z}+\mathbf{Z}\phi$, until finding the smallest power of 2 such that $\phi^{2^n}x = x' \geq 1$. The powers of ϕ are computed by repeated squarings (and they can be pre-stored up to certain reasonable maximal n, but we are discussing here a ``no prior bound'' situation). The searched-for exponent $k(x)$ is $k(x') - 2^n$. So we are reduced to the $x \geq 1$ case. If $x < \phi$, then $k(x) = 0$. If $x \geq \phi = \phi^{2^0}$, repeated squaring of ϕ and comparisons with x using $u + v\phi$ representations will give us the smallest $n \geq 0$ with $\phi^{2^{n+1}} > x$. Then divide x by ϕ^{2^n} (or rather multiply with ϕ^{-2^n}), again using $\mathbf{Z}[\phi]$ algebra, obtaining some x' with $1 \leq x' < \phi^{2^n}$ and $k(x) = k(x') + 2^n$ with $0 \leq k(x') < 2^n$. After finitely many steps we will have reduced to $[1, \phi)$ and the algorithm ends.

The macro helpers handling the computation of the ratio of logarithms should not make assumptions about \xintDigits. Here is the original source as it was used to create the code (not any AI would be able to do that... but xintexpr succeeds!). In particular note the impressive nesting of \xintiiexpr inside \xintfloatexpr. The log output (thanks to \xintverbosetrue) was then edited by hand to not use macros using tacitly \XINTdigits, and to reduce to 8 digits of precision as this is enough.

```
\xintverbosetrue
\xintdeffloatvar Phi := (1 + sqrt(5))/2;
\xintdeffloatvar Psi := (1 - sqrt(5))/2;
\xintdeffloatvar logPhi := log10(Phi);% would have been precomputed anyhow
\xintdefiifunc greedyA(a,b):= \xintfloatexpr
        round(log10(a+b*Phi) / logPhi)\relax;
\xintdefiifunc greedyB(a,b):= \xintfloatexpr
  round(log10(\xintiiexpr (a*(a+b) -sqr(b))\relax/(a+b*Psi))
        / logPhi)\relax;
```

```
\def\bergman_nearindex_A#1;#2;{%
  \xintiRound {0}{%
   \xintFloatDiv[8]{%
    \PoorManLogBaseTen_raw
     {\xintFloatAdd[8]{#1}%
                      {\xintFloatMul[8]{#2}{1618034[-6]}}}%
    }%
    {20898764[-8]}%
  }%
}%
\def\bergman_nearindex_B#1;#2;{%
  \xintiRound {0}{%
   \xintFloatDiv[8]{%
     \PoorManLogBaseTen_raw
     {\xintFloatDiv[8]%
          {\xintiiSub
             {\xintiiMul {#1}{\xintiiAdd{#1}{#2}}}%
             {\xintiiSqr {#2}}%
          }%
          {\xintFloatAdd[8]%
             {#1}{\xintFloatMul[8]{#2}{-618034[-6]}}%
          }%
     }%
    }%
    {20898764[-8]}%
  }%
}%
\def\PhiMaxE{\the\numexpr\phimaxe}%
\def\phimaxe #1{%
    \expandafter\phimaxe_a\expanded{{#1}}%
}%
\def\phimaxe_a #1{\expandafter\phimaxe_b\string#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
```

```
\def\phimaxe_b #1[\if#1{\expandafter\phimaxe_X % }
                   \else\expandafter\phimaxe_N\fi #1]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phimaxe_N #1;{\phimaxe_ab {#1}{0};}%
\def\phimaxe_X #1{\expandafter\phimaxe_ab\expandafter{\iffalse}\fi}%
\def\PhiMaxE_ab {\romannumeral0\phimaxe_ab}%
\def\phimaxe_ab #1#2;{%
    \expandafter\phimaxe_i\romannumeral`&&@%
    \ifnum\numexpr\xintiiSgn{#1}*\xintiiSgn{#2}\relax=-1
          \expandafter\bergman_nearindex_B
    \else \expandafter\bergman_nearindex_A
    \fi #1;#2;;#1;#2;%
}%
\def\phimaxe_i #1;{%
    \expandafter\phimaxe_j\romannumeral`&&@\zeck@fpair #1.#1;%
}%
\def\phimaxe_j #1#2#3;#4;#5;{%
    #3\ifnum
        \expandafter\PhiIISign_ab
        \expanded{{\xintiiSub{#2}{#4}}{\xintiiSub{#1}{#5}}}>\xint_c_
    -1\fi\relax
}%
```

8.4.3. \PhiBList

Will serve (or rather a close derivative) as support for the phiexponents() function in \xinteval, and is used both by \PhiExponents and \PhiBasePhi.

```
\def\PhiBList{\expanded\phiblist}%
\def\phiblist #1{%
    \expandafter\phiblist_a\expanded{{#1}}%
}%
\def\phiblist_a #1{\expandafter\phiblist_b\string#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phiblist_b #1[\if#1{\expandafter\phiblist_X % }
                   \else\expandafter\phiblist_N\fi #1]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phiblist_N #1;{\phiblist_ab {#1}{0};}%
\def\phiblist_X #1{\expandafter\phiblist_ab\expandafter{\iffalse}\fi}%
\def\PhiBlist_ab {\expanded\phiblist_ab}%
\def\phiblist_ab #1#2;{{%
    \ifcase\PhiIISign_ab{#1}{#2} %
     0\expandafter\phiblist_stop
    \or
     +\expandafter\phiblist_i
    \else
     -\expandafter\phiblist_neg
    \fi
    #1;#2;%
}}%
\def\phiblist_stop#1;#2;{}%
```

Attention that adding minus signs here would fool \xintiiSgn which makes no normalization and looks only at first token.

TODO: check if it is more efficient to do `\expanded{\noexpand\foo...}` rather than `\expandafter\foo\expanded{...}`.

```
\def\phiblist_neg#1;#2;{%
    \expandafter\phiblist_i\expanded{\XINT_Opp#1;\XINT_Opp#2;}%
}%
\def\phiblist_i#1;#2;{%
    \expandafter\phiblist_j\romannumeral`&&@%
    \ifnum\numexpr\xintiiSgn{#1}*\xintiiSgn{#2}\relax=-1
          \expandafter\bergman_nearindex_B
    \else \expandafter\bergman_nearindex_A
    \fi #1;#2;;#1;#2;%
}%
\def\phiblist_j #1;{%
    \expandafter\phiblist_k\romannumeral`&&@\zeck@fpair #1.#1;%
}%
\def\phiblist_k #1#2#3;#4;#5;{%
    \if1\expandafter\PhiIISign_ab
        \expanded{{\xintiiSub{#2}{#4}}{\xintiiSub{#1}{#5}}}%
      \expandafter\phiblist_ci
    \else
      \expandafter\phiblist_cii
    \fi
    #1;#2;#3;#4;#5;%
}%
\def\phiblist_ci #1;#2;#3;#4;#5;{%
    {\the\numexpr#3-1\relax}%
    \expandafter\phiblist_again\expanded{%
      {\xintiiSub{\xintiiAdd{#2}{#4}}{#1}}%
      {\xintiiSub{#5}{#2}}%
    }%
}%
\def\phiblist_cii #1;#2;#3;#4;#5;{%
    {#3}%
    \expandafter\phiblist_again
      {\xintiiSub{#4}{#2}}%
      {\xintiiSub{#5}{#1}}%
}%
\def\phiblist_again #1#2{%
    \if0\PhiIISign_ab{#1}{#2}%
        \expandafter\phiblist_stop
    \else
        \expandafter\phiblist_i
    \fi
    #1;#2;%
}%
```

8.4.4. \PhiExponents

As this depends upon `\PhiBList` it will have to unbrace at each step to check sign of the exponent.

```
\def\PhiExponents{\expanded\phiexponents}%
\def\phiexponents#1{{%
```

```
   \expandafter\phiexponents_a
   \expanded\expandafter\phiblist_a\expanded{{#1}}%
   ;%
}}%
\def\phiexponents_a #1{\if-#1.\fi\phiexponents_b}%
\def\phiexponents_b #1{%
    \if;#1\expandafter\phiexponents_done\fi
    #1\phiexponents_c
}%
\def\phiexponents_c #1{%
    \if;#1\expandafter\phiexponents_done\fi
    \PhiExponentsSep#1\phiexponents_c
}%
\def\phiexponents_done#1\phiexponents_c{}%
\def\PhiExponentsSep{, }%
```

8.4.5. \PhiBasePhi

As this depends upon \PhiBList it will have to unbrace at each step to check sign of the exponent.

```
\def\PhiBasePhi{\expanded\phibasephi}%
\def\phibasephi#1{{%
   \expandafter\phibasephi_a
   \expanded\expandafter\phiblist_a\expanded{{#1}}%
   ;%
}}%
\def\phibasephi_a #1{%
    \if-#1-\fi
    \if0#1\expandafter\xint_gob_til_sc\fi
    \phibasephi_b
}%
\def\phibasephi_b #1{\phibasephi_c #1.}%
\def\phibasephi_c #1#2.{%
  \if-#1%
     0\PhiBasePhiSep\xintReplicate{#2-1}{0}%
     1\expandafter\phibasephi_n
  \else
     1\expandafter\phibasephi_p
  \fi
  #1#2.%
}%
\def\phibasephi_p #1.#2{\phibasephi_pa #1.#2\xint:}%
\def\phibasephi_pa #1.#2{%
  \if;#2\xintReplicate{#1}{0}\expandafter\xint_gob_til_xint:\fi
  \phibasephi_pb #1.#2%
}%
\def\phibasephi_pb #1.#2#3\xint:{%
  \if-#2%
    \xintReplicate{#1}{0}\PhiBasePhiSep\xintReplicate{#3-1}{0}%
    1\expandafter\phibasephi_n
  \else
    \xintReplicate{#1-#2#3-1}{0}%
    1\expandafter\phibasephi_p
  \fi
  #2#3.%
}%
\def\phibasephi_n #1.#2{\phibasephi_na #1.#2\xint:}%
\def\phibasephi_na #1.#2{%
  \if;#2\expandafter\xint_gob_til_xint:\fi
  \phibasephi_nb #1.#2%
}%
\def\phibasephi_nb #1.#2#3\xint:{%
  \xintReplicate{#1+#3-1}{0}%
  1\phibasephi_n #2#3.%
}%
\def\PhiBasePhiSep{.}%
```

8.4.6. \PhiBaseHexPhi

Broken if \PhiBasePhiSep is not default.

Attention for fractional part that we must first extend with trailing zeros if needed to make it have 4N digits. (Expansion of integers will have an even number of fractional digits, but of course this is not true of general a + b phi.

```
\def\PhiBaseHexPhi{\expanded\phibasehexphi}%
\def\phibasehexphi#1{\expandafter\phibasehexphi_a\expanded{%
   \expandafter\phibasephi_a
   \expanded\expandafter\phiblist_a\expanded{{#1}}%
   ;}..,;%
}%
```

MEMO: fortunately the second \xintBinToHex will not trim leading zeros which were originally zeros after the radix separator. But we also must make sure to apply it to an input having a multiple of four number of binary digits.

```
\def\phibasehexphi_a #1.#2.#3#4;{{%
    \xintBinToHex{#1}%
    \if.#3.\xintBinToHex{\expanded{#2\expandafter\phibasehexphi_aux
         \romannumeral0\xintlength{#2}.}}\fi
}}%
%     \end{macrocode}
% Here |#1| is at least one.
%     \begin{macrocode}
\def\phibasehexphi_aux #1.{\xintReplicate{4*((#1+1)/4) - #1}{0}}%
```

8.4.7. \PhiXfromExponents

If the list starts with period, it means it represents a negative number (or perhaps zero is there is nothing else).

```
\def\PhiXfromExponents{\expanded\phixfromexponents}%
\def\phixfromexponents#1{%
    \expandafter\phixfromexponents_a\romannumeral`&&@#1,;%
}%
\def\phixfromexponents_a #1{%
    \if.#1\expandafter\phixfromexponents_n
```

```
     \else\expandafter\phixfromexponents_p
    \fi #1%
}%
\def\phixfromexponents_p #1;{{%
    {\xintiiSum{\expanded{\zeck@applytocsv_a\Zeck@FNminusOne#1;}}}%
    {\xintiiSum{\expanded{\zeck@applytocsv_a\Zeck@FN#1;}}}%
}}%
\def\phixfromexponents_n .#1;{{%
    {\xintiiOpp{\xintiiSum{\expanded{\zeck@applytocsv_a\Zeck@FNminusOne#1;}}}}%
    {\xintiiOpp{\xintiiSum{\expanded{\zeck@applytocsv_a\Zeck@FN#1;}}}}%
}}%
```

8.4.8. \PhiXfromBasePhi

The radix separator must be an explicit period. There must be at least one digit before the period, if the latter is there. Empty input is allowed. Input must *f*-expand completely thus input such as \macroA.\macroB is not allowed.

Coded the lazy way by first converting to comma separated list of exponents. The \phiexponentsfromrep's output has a final comma but this is ok.

```
\def\PhiXfromBasePhi{\expanded\phixfrombasephi}%
\def\phixfrombasephi{\expandafter\phixfromexponents\expanded\phiexponentsfromrep}%
\def\phiexponentsfromrep#1{%
    {{\iffalse}\fi\expandafter\phiexponentsfromrep_a\romannumeral`&&@#1.;\xint:}%
}%
\def\phiexponentsfromrep_a #1{%
    \if-#1.\xint_dothis\phiexponentsfromrep_a\fi
    \if.#1\xint_dothis\zeck_done\fi
    \xint_orthat{\phiexponentsfromrep_b #1}%
}%
\def\phiexponentsfromrep_b #1.#2{%
    \expandafter\phiexponentsfromrep_c\romannumeral0\xintreversedigits{#1};%
    \if;#2\expandafter\zeck_done\else
          \expandafter\phiexponentsfromrep_i\fi #2%
}%
\def\phiexponentsfromrep_c{\phiexponentsfromrep_d 0.}%
\def\phiexponentsfromrep_d#1.#2{%
    \if;#2\expandafter\xint_gob_til_dot\fi
    \if#21#1,\fi
    \expandafter\phiexponentsfromrep_d\the\numexpr#1+1.%
}%
\def\phiexponentsfromrep_i{\phiexponentsfromrep_j -1.}%
\def\phiexponentsfromrep_j#1.#2{%
    \if;#2\expandafter\zeck_done\fi
    \if#21#1,\fi
    \expandafter\phiexponentsfromrep_j\the\numexpr#1-1.%
}%
```

8.4.9. \PhiXfromBaseHexPhi

Added at 0.9d.

```
\def\PhiXfromBaseHexPhi{\expanded\phixfrombasehexphi}%
```

```
\def\phixfrombasehexphi{\expandafter\phixfromexponents
                        \expanded\phiexponentsfromhexrep}%
\def\phiexponentsfromhexrep#1{%
    {{\iffalse}\fi\expandafter\phiexponentsfromhexrep_a\romannumeral`&&@#1.;\xint:}%
}%
\def\phiexponentsfromhexrep_a #1{%
    \if-#1.\xint_dothis\phiexponentsfromhexrep_a\fi
    \if.#1\xint_dothis\zeck_done\fi
    \xint_orthat{\phiexponentsfromhexrep_b #1}%
}%
\def\phiexponentsfromhexrep_b #1.#2{%
    \expandafter\phiexponentsfromrep_c
        \romannumeral0\xintreversedigits{\xintHexToBin{#1}};%
    \if;#2\expandafter\zeck_done\else
          \expandafter\phiexponentsfromhexrep_i\fi #2%
}%
```

Attention that conversion from hexadecimal to binary must preserve leading zeros!

```
\def\phiexponentsfromhexrep_i#1;{%
    \expanded{\noexpand\phiexponentsfromrep_j -1.\xintCHexToBin{#1}};%
}%
```

8.5. The Knuth Fibonacci multiplication

8.5.1. \ZeckKMul: Knuth definition

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.

For the \xinteval we need a variant applying \xintNum to its arguments.

```
\def\ZeckKMul{\romannumeral0\zeckkmul}%
\def\zeckkmul#1#2{\expandafter\zeckkmul_a
                  \expanded{\ZeckIndices{#1}%
                            ,;%
                            \ZeckIndices{#2}%
                            ,,}%
}%
\def\ZeckKMulNum#1#2{\romannumeral0\zeckkmul{\xintNum{#1}}{\xintNum{#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_iv\fi
    \Zeck@@FN{#1+#2}\zeckkmul_c{#1}%
}%
\def\zeckkmul_done#1;{\iffalse{{\fi}}}%
```

8.5.2. \ZeckAMul: Arnoux formula

Here a `\romannumeral0` trigger is used to match `\xintiisum`.

```
\def\ZeckAMul{\romannumeral0\zeckamul}%
\def\ZeckAMulNum#1#2{\romannumeral0\zeckamul{\xintNum{#1}}{\xintNum{#2}}}%
\def\zeckamul#1{\expandafter\zeckamul_in\romannumeral`&&@#1;}%
\def\zeckamul_in#1;#2{\expandafter\zeckamul_a\romannumeral`&&@#2;#1;}%
\def\zeckamul_a #1;#2;{\xintiiadd
    {\xintiiMul{#1}{#2}}
    {\xintiiAdd{\xintiiMul{#1}{\ZeckB{#2}}}{\xintiiMul{\ZeckB{#1}}{#2}}}%
}%
```

8.5.3. \ZeckB: B operator

Here a `\romannumeral0` trigger is used to match `\xintiisum`. It is a fact of life that `\xintiiSum` needs to grab something at each item before expanding it, rather than expanding prior to grabbing. So we use an `\expanded` wrapper.

```
\def\ZeckB{\romannumeral0\zeckb}%
\def\zeckb#1{\xintiisum{\expanded{\iffalse}\fi
             \expandafter\zeckb_a\expanded\zeckindices{#1},,}}%
```

`#1-1` is always positive.

```
\def\zeckb_a#1,{%
    \if\relax#1\relax\expandafter\zeckb_done\fi
    \Zeck@@FN{#1-1}\zeckb_a
}%
\def\zeckb_done#1\zeckb_a{\iffalse{\fi}}%
```

8.6. Typesetting

8.6.1. \ZeckPrintIndexedSum

Expandable, but needs x-expansion. The default requires math mode, at it uses `\sb`. We do not use `_` here due to its current catcode. It only f-expands its argument. No repeated or final comma is allowed.

```
\def\ZeckPrintIndexedSumSep{+\allowbreak}%
\def\ZeckPrintOne#1{F\sb{#1}}%
\def\ZeckPrintIndexedSum#1{%
    \expandafter\zeckprintindexedsum\romannumeral`&&@#1,;%
}%
\def\zeckprintindexedsum#1{%
    \if,#1\expandafter\xint_gob_til_sc\fi \zeckprintindexedsum_a#1%
}%
\def\zeckprintindexedsum_a#1,{\ZeckPrintOne{#1}\zeckprintindexedsum_b}%
\def\zeckprintindexedsum_b #1{%
    \if;#1\expandafter\xint_gob_til_sc\fi
    \ZeckPrintIndexedSumSep\zeckprintindexedsum_a#1%
}%
```

8.6.2. \PhiPrintIndexedSum

A clone of `\ZeckPrintIndexedSum` with its own namespace.

```
\let\PhiPrintIndexedSumSep\ZeckPrintIndexedSumSep
\def\PhiPrintOne#1{\phi\sp{#1}}%
\def\PhiPrintIndexedSum#1{%
    \expandafter\phiprintindexedsum\romannumeral`&&@#1,;%
}%
\def\phiprintindexedsum#1{%
    \if,#1\expandafter\xint_gob_til_sc\fi \phiprintindexedsum_a#1%
}%
\def\phiprintindexedsum_a#1,{\PhiPrintOne{#1}\phiprintindexedsum_b}%
\def\phiprintindexedsum_b #1{%
    \if;#1\expandafter\xint_gob_til_sc\fi
    \PhiPrintIndexedSumSep\phiprintindexedsum_a#1%
}%
```

8.6.3. \PhiTypesetX

```
\def\PhiTypesetX #1{%
    \expandafter\PhiTypesetXPrint\expanded{#1}%
}%
\def\PhiTypesetXPrint #1#2{%
    \xintiiifSgn{#1}%
      {#1\xintiiifSgn{#2}{#2\phi}{}{+#2\phi}}%
      {\xintiiifSgn{#2}{#2\phi}{0}{#2\phi}}%
      {#1\xintiiifSgn{#2}{#2\phi}{}{+#2\phi}}%
}%
```

8.7. Extensions of the \xinteval syntax

Initially functions and Knuth operator were added to \xintiieval only, but when it was decided to overload the infix operators to handle inputs from $\mathbf{Z}[\phi]$, it felt awkward not to include the division so finally we support $\mathbf{Q}(\phi)$ algebra, and for this had to switch to \xinteval.

8.7.1. Provisory ad hoc support for adding xintexpr operators

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 {expr}}%
\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.

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)

```
\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 {expr}\ZECK_tmpa\ZECK_tmpL\ZECK_tmpR #2%
    \expandafter\ZECK_dotheitselves\ZECK_tmpa\relax
  \unless\ifcsname
    XINT_expr_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_expr_op_#1\endcsname
                        {\csname XINT_expr_op_\ZECK_tmpa\endcsname}%
      \fi
    \fi
    \ZECK_dotheitselves{#1#2}%
}%
```

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.

```
\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 {expr}\ZECK_tmpa
%%  \ifxintverbose
    \xintMessage{zeckendorf}{Warning}{infix operator \ZECK_tmpa\space
        has been DELETED!}%
%%  \fi
}%
\fi
\def\ZeckDeleteOperator#1{\zeckundefinfix{#1}}%
```

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

```
\def\ZeckSetAsKnuthOp#1{\zeckdefinfix{#1}{\ZeckKMulNum}{14}{14}}%
\def\ZeckSetAsArnouxOp#1{\zeckdefinfix{#1}{\ZeckAMulNum}{14}{14}}%
```

8.7.2. The $ and $$ as infix operator for the Knuth multiplication

```
\ZeckSetAsArnouxOp{$}% $ (<-only to tame Emacs/AUCTeX highlighting)
\ZeckSetAsKnuthOp{$$}% $$
```

8.7.3. Support macros for Q(ϕ) algebra

\PhiSgn

```
\def\PhiSign{\romannumeral0\phisign}%
\def\phisign #1{%
    \expandafter\phisign_a\expanded{{#1}}%
}%
\def\phisign_a #1{\expandafter\phisign_b\string#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phisign_b #1[\if#1{\expandafter\phisign_X % }
                     \else\expandafter\phisign_N\fi #1]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phisign_N #1;{\XINT_sgn #1\xint:}%
\def\phisign_X #1{\expandafter\phisign_ab\expandafter{\iffalse}\fi}%
\def\PhiSign_ab {\romannumeral0\phisign_ab}%
\def\phisign_ab #1#2{%
    \xintiiifsgn{#1}%
    {% a < 0
```

```
     \xintiiifSgn{#2}%
           {-1}%
           {-1}%
           {% a < 0, b > 0, return 1 iff a^2+ab<b^2
            \xintifLt{\xintMul{#1}{\xintAdd{#1}{#2}}}{\xintSqr{#2}}%
            {1}%
            {-1}%
           }%
    }%
    {\xintiiSgn{#2}}%
    {% a > 0
     \xintiiifSgn{#2}%
           {% a > 0, b < 0, return 1 iff a^2+ab>b^2
            \xintifGt{\xintMul{#1}{\xintAdd{#1}{#2}}}{\xintSqr{#2}}%
            {1}%
            {-1}%
           }%
           {1}%
           {1}%
    }%
}%
```

\PhiAbs

```
\def\PhiAbs{\romannumeral0\phiabs}%
\def\phiabs #1{%
     \expandafter\phiabs_a\expanded{{#1}}%
}%
\def\phiabs_a #1{\expandafter\phiabs_b\string#1}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phiabs_b #1[\if#1{\expandafter\phiabs_X % }
                  \else\expandafter\phiabs_N\fi #1]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\let\phiabs_N \XINT_abs
\def\phiabs_X #1{\expandafter\phiabs_x\expandafter{\iffalse}\fi}%
\def\phiabs_x #1#2{\expanded{%
   \ifnum\PhiSign_ab {#1}{#2}<\xint_c_
    \expandafter\xint_firstoftwo\else\expandafter\xint_secondoftwo\fi
    {{\XINT_Opp#1}{\XINT_Opp#2}}{{#1}{#2}}%
   }%
}%
```

\PhiNorm

```
\def\PhiNorm{\romannumeral0\phinorm}%
\def\phinorm #1{%
    \expandafter\phinorm_a\expanded{{#1}}%
}%
\def\phinorm_a #1{\expandafter\phinorm_b\detokenize{#1;}{#1}}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phinorm_b #1#2;[\if#1{\expandafter\phinorm_X % }
                     \else\expandafter\phinorm_N\fi]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
```

```
\let\phinorm_N\xintsqr
\def\phinorm_X #1{\phinorm_x #1}%
\def\phinorm_x #1#2{\xintsub
  {\xintMul{#1}{\xintAdd{#1}{#2}}}%
  {\xintSqr{#2}}%
}%
```

\PhiConj

```
\def\PhiConj{\romannumeral0\phiconj}%
\def\phiconj #1{%
    \expandafter\phiconj_a\expanded{{#1}}%
}%
\def\phiconj_a #1{\expandafter\phiconj_b\detokenize{#1;}#1}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phiconj_b #1#2;[\if#1{\expandafter\phiconj_X % }
                     \else\expandafter\phiconj_N\fi]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\let\phiconj_N\space
\def\phiconj_X #1#2{\expanded{%
  {\xintAdd{#1}{#2}}{\XINT_Opp #2}%
}}%
```

\PhiOpp

```
\def\PhiOpp{\romannumeral0\phiopp}%
\def\phiopp #1{%
    \expandafter\phiopp_a\expanded{{#1}}%
}%
\def\phiopp_a #1{\expandafter\phiopp_b\string#1}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phiopp_b #1[\if#1{\expandafter\phiopp_X % }
                  \else\expandafter\phiopp_N\fi #1]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\let\phiopp_N \XINT_opp
\def\phiopp_X #1{\expandafter\phiopp_x\expandafter{\iffalse}\fi}%
\def\phiopp_x #1#2{\expanded{%
    {\XINT_Opp #1}{\XINT_Opp #2}%
}}%
```

\PhiAdd

```
\def\PhiAdd{\romannumeral0\phiadd}%
\def\phiadd #1#2{%
    \expandafter\phiadd_a\expanded{{#1}{#2}}%
}%
\def\phiadd_a #1{\expandafter\phiadd_b\string#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phiadd_b #1[\if#1{\expandafter\phiadd_X % }
                  \else\expandafter\phiadd_N\fi #1]%
\def\phiadd_N #1;#2[\expandafter\phiadd_n\string#2;[#1]]%
\def\phiadd_n #1[\if#1{\expandafter\phiadd_nX % }
                  \else\expandafter\phiadd_nn\fi #1]%
\def\phiadd_nX #1[\expandafter\phiadd_nx\expandafter[\iffalse]\fi]%
```

```
\def\phiadd_X #1[\expandafter\phiadd_x\expandafter[\iffalse]\fi]%
```

#1={a}{b}.

```
\def\phiadd_x #1;#2[\expandafter\phiadd_xa\string#2;#1]%
\def\phiadd_xa#1[\if#1{\expandafter\phiadd_XX % }
                  \else\expandafter\phiadd_xn\fi #1]%
\def\phiadd_XX #1[\expandafter\phiadd_xx\expandafter[\iffalse]\fi]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phiadd_nn #1;{\xintadd{#1}}%
\def\phiadd_nx #1#2;#3{\expandafter
   {\romannumeral0\xintadd{#1}{#3}}{#2}%
}%
\def\phiadd_xn #1;#2{\expandafter
   {\romannumeral0\xintadd{#1}{#2}}%
}%
\def\phiadd_xx #1#2;#3#4{\expanded{%
    {\xintAdd{#1}{#3}}%
    {\xintAdd{#2}{#4}}%
}}%
```

\PhiSub

```
\def\PhiSub{\romannumeral0\phisub}%
\def\phisub #1#2{%
    \expandafter\phisub_a\expanded{{#1}{#2}}%
}%
\def\phisub_a #1{\expandafter\phisub_b\string#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phisub_b #1[\if#1{\expandafter\phisub_X % }
                  \else\expandafter\phisub_N\fi #1]%
\def\phisub_N #1;#2[\expandafter\phisub_n\string#2;[#1]]%
\def\phisub_n #1[\if#1{\expandafter\phisub_nX % }
                  \else\expandafter\phisub_nn\fi #1]%
\def\phisub_nX #1[\expandafter\phisub_nx\expandafter[\iffalse]\fi]%
\def\phisub_X #1[\expandafter\phisub_x\expandafter[\iffalse]\fi]%
```

#1={a}{b}.

```
\def\phisub_x #1;#2[\expandafter\phisub_xa\string#2;#1]%
\def\phisub_xa#1[\if#1{\expandafter\phisub_XX % }
                  \else\expandafter\phisub_xn\fi #1]%
\def\phisub_XX #1[\expandafter\phisub_xx\expandafter[\iffalse]\fi]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phisub_nn #1;#2{\xintsub{#2}{#1}}%
\def\phisub_nx #1#2;#3{\expanded{%
   {\xintSub{#3}{#1}}{\XINT_Opp#2}%
}}%
\def\phisub_xn #1;#2{\expandafter
   {\romannumeral0\xintsub{#2}{#1}}%
}%
\def\phisub_xx #1#2;#3#4{\expanded{%
    {\xintSub{#3}{#1}}%
    {\xintSub{#4}{#2}}%
}}%
```

\PhiMul

```
\def\PhiMul{\romannumeral0\phimul}%
\def\phimul #1#2{%
    \expandafter\phimul_a\expanded{{#1}{#2}}%
}%
\def\phimul_a #1{\expandafter\phimul_b\string#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phimul_b #1[\if#1{\expandafter\phimul_X % }
                  \else\expandafter\phimul_N\fi #1]%
\def\phimul_N #1;#2[\expandafter\phimul_n\string#2;[#1]]%
\def\phimul_n #1[\if#1{\expandafter\phimul_nX % }
                  \else\expandafter\phimul_nn\fi #1]%
\def\phimul_nX #1[\expandafter\phimul_nx\expandafter[\iffalse]\fi]%
\def\phimul_X #1[\expandafter\phimul_x\expandafter[\iffalse]\fi]%
```

#1={a}{b}.

```
\def\phimul_x #1;#2[\expandafter\phimul_xa\string#2;#1]%
\def\phimul_xa#1[\if#1{\expandafter\phimul_XX % }
                  \else\expandafter\phimul_xn\fi #1]%
\def\phimul_XX #1[\expandafter\phimul_xx\expandafter[\iffalse]\fi]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phimul_nn#1;{\xintmul{#1}}%
\def\phimul_nx#1#2;#3{\expanded{%
    {\xintMul{#1}{#3}}{\xintMul{#2}{#3}}%
}}%
\def\phimul_xn#1;#2#3{\expanded{%
    {\xintMul{#1}{#2}}{\xintMul{#1}{#3}}%
}}%
\def\phimul_xx #1#2;#3#4{%
    \expandafter\phimul_xx_a\expanded{%
      \xintMul{#1}{#3};% ca
      \xintMul{#2}{#4};% db
      \xintMul{#1}{#4};% da
      \xintMul{#2}{#3};% cb
    }%
}%
\def\phimul_xx_a #1;#2;#3;#4;{\expanded{%
    {\xintAdd{#1}{#2}}%
    {\xintAdd{#3}{\xintAdd{#4}{#2}}}%
}}%
```

\PhiDiv

```
\def\PhiDiv{\romannumeral0\phidiv}%
\def\phidiv #1#2{%
    \expandafter\phidiv_a\expanded{{#1}{#2}}%
}%
\def\phidiv_a #1{\expandafter\phidiv_b\string#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phidiv_b #1[\if#1{\expandafter\phidiv_X % }
                  \else\expandafter\phidiv_N\fi #1]%
\def\phidiv_N #1;#2[\expandafter\phidiv_n\string#2;[#1]]%
\def\phidiv_n #1[\if#1{\expandafter\phidiv_nX % }
```

```
                  \else\expandafter\phidiv_nn\fi #1]%
\def\phidiv_nX #1[\expandafter\phidiv_nx\expandafter[\iffalse]\fi]%
\def\phidiv_X #1[\expandafter\phidiv_x\expandafter[\iffalse]\fi]%
```

#1={a}{b}.

```
\def\phidiv_x #1;#2[\expandafter\phidiv_xa\string#2;#1]%
\def\phidiv_xa#1[\if#1{\expandafter\phidiv_XX % }
                  \else\expandafter\phidiv_xn\fi #1]%
\def\phidiv_XX #1[\expandafter\phidiv_xx\expandafter[\iffalse]\fi]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phidiv_nn#1;#2{\xintdiv{#2}{#1}}%
\def\phidiv_nx#1#2{%
    \expandafter\phidiv_nx_a\expanded{%
     {\xintSub{\xintMul{#1}{\xintAdd{#1}{#2}}}{\xintSqr{#2}}}%
    }{#1}{#2}%
}%
\def\phidiv_nx_a#1#2#3;#4{\expanded{%
    {\xintIrr{\xintDiv{\xintMul{#4}{\xintAdd{#2}{#3}}}{#1}}[0]}%
    {\xintIrr{\xintOpp{\xintDiv{\xintMul{#4}{#3}}{#1}}}[0]}%
}}%
\def\phidiv_xn #1;#2#3{\expanded{%
    {\xintIrr{\xintDiv{#2}{#1}}[0]}%
    {\xintIrr{\xintDiv{#3}{#1}}[0]}%
}}%
\def\phidiv_xx #1#2;#3#4{%
    \expandafter\phidiv_xx_a\expanded{%
     \expandafter\phimul_xx
       \expanded{{\xintAdd{#1}{#2}}{\XINT_Opp #2}};{#3}{#4}%
     {\xintSub{\xintMul{#1}{\xintAdd{#1}{#2}}}{\xintSqr{#2}}}%
    }%
}%
\def\phidiv_xx_a #1#2#3{%
    \expanded{%
    {\xintIrr{\xintDiv{#1}{#3}}[0]}%
    {\xintIrr{\xintDiv{#2}{#3}}[0]}%
   }%
}%
```

\PhiPow

```
\def\PhiPow{\romannumeral0\phipow}%
\def\phipow #1#2{%
    \expandafter\phipow_a\expanded
    {{#1}{\xintNum{#2}}}%
}%
\def\phipow_a #1{\expandafter\phipow_b\string#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phipow_b #1[\if#1{\expandafter\phipow_X % }
                  \else\expandafter\phipow_N\fi #1]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phipow_N #1;{\xintpow{#1}}%
\def\phipow_X #1{\expandafter\phipow_x\expandafter{\iffalse}\fi}%
```

Let's handle negative exponents too, now that we use \xinteval.

```
\def\phipow_x #1;#2{\phipow_fork #2;#1}%
\def\phipow_fork #1{%
    \xint_UDzerominusfork
       0#1\phipow_neg
       #1-\phipow_zero
        0-\phipow_pos
    \krof #1%
}%
\def\phipow_zero 0;#1#2{{1}{0}}%
\def\phipow_neg -{%
    \expandafter\phiinv_ab\romannumeral0\phipow_pos
}%
\def\phiinv_ab #1#2{%
    \expandafter\phiinv_c\expanded{%
     {\xintSub{\xintMul{#1}{\xintAdd{#1}{#2}}}{\xintSqr{#2}}}%
    }{#1}{#2}%
}%
\def\phiinv_c #1#2#3{\expanded{%
    {\xintIrr{\xintDiv{\xintAdd{#2}{#3}}{#1}}[0]}%
    {\xintIrr{\xintOpp{\xintDiv{#3}{#1}}}[0]}%
}}%
\def\phipow_pos #1;{%
    \expandafter\phipow_xa
    \expanded{10\xintDecToBin{#1}},;%
}%
\def\phipow_xa #1#2#3#4;{%
  \if#3,\expandafter\phipow_done\fi
  \if#31\expandafter\phipow_xo
   \else\expandafter\phipow_xe\fi
  {#1}{#2}#4;%
}%
\def\phipow_done \if#1\fi #2#3;#4#5{{#2}{#3}}%
\def\phipow_xo #1#2{%
  \expandafter\phipow_xo_a\expanded{%
  \xintSqr{#1};\xintMul{#1}{#2};\xintSqr{#2};%
  }%
}%
\def\phipow_xo_a #1;#2;#3;{%
  \expandafter\phipow_xo_b\expanded{%
  \xintAdd{#1}{#3};\xintAdd{#2}{\xintAdd{#2}{#3}};%
  }%
}%
\def\phipow_xo_b#1;#2;#3;#4#5{%
  \expandafter\phipow_xa\romannumeral0%
  \phimul_xx {#1}{#2};{#4}{#5}#3;{#4}{#5}%
}%
\def\phipow_xe #1#2{%
  \expandafter\phipow_xe_a\expanded{%
  \xintSqr{#1};\xintMul{#1}{#2};\xintSqr{#2};%
  }%
}%
\def\phipow_xe_a #1;#2;#3;{%
```

```
  \expandafter\phipow_xa\expanded{%
  {\xintAdd{#1}{#3}}{\xintAdd{#2}{\xintAdd{#2}{#3}}}%
  }%
}%
```

8.7.4. Overloading +, -, *, /, ^, and **

The ** is pre-aliased to ^ at xintexpr level via \XINT_expr_itself_**, so nothing to do here once ^ is handled.

The unary - requires extra care.

```
\zeckdefinfix{+}{\PhiAdd}{12}{12}%
\zeckdefinfix{-}{\PhiSub}{12}{12}%
\xintFor #1 in {xii,xiv,xvii}\do{%
    \expandafter\def\csname XINT_expr_exec_-#1\endcsname
    ##1##2##3%
    {%
      \expandafter ##1\expandafter ##2\expandafter
       {%
        \romannumeral`&&@\XINT:NEhook:f:one:from:one
        {\romannumeral`&&@\PhiOpp##3}%
       }%
    }%
}%
\zeckdefinfix{*}{\PhiMul}{14}{14}%
\zeckdefinfix{/}{\PhiDiv}{14}{14}%
\zeckdefinfix{^}{\PhiPow}{18}{17}%
```

8.7.5. Variables and functions for \xinteval

The macros computing Fibonacci numbers, Zeckendorf indices, and Bergman exponents, were done originally assuming to be used with arguments in strict integer format. But when operations are executed in \xinteval the intermediate results will use the ``raw'' format described in the xintexpr manual, not the ``strict integer format''. We thus need wrappers to apply \xintNum for normalization, even though this adds annoying overhead. These wrappers can assume that the argument is already expanded.

For macros handling input being either one unbraced integer or a pair of braced integers this is more complicated. We separated \PhiIISign_ab from \PhiSign to this aim. The former for optimized internal usage, only using integer algebra. The latter uses the xintfrac macros, so there is no problem and we do not want to truncate arguments to integers. Similarly for \PhiAbs no need to do something special.

\PhiMaxE is integer-only, but in the end I decided to not provide an \xinteval interface and to remove the one for \ZeckMaxK.

For the support for phiexponents(), which is also integer only we have to use \xintNum, the problem is that we can't do that prior to know if used with an integer or a nutple. So \Phi@BList was done to handle that.

```
\xintdefvar phi:=[0,1];%
\xintdefvar psi:=[1,-1];%
\def\XINT_expr_func_phinorm #1#2#3%
{%
```

```
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\PhiNorm#3}}%
}%
\def\XINT_expr_func_phiconj #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\PhiConj#3}}%
}%
\def\XINT_expr_func_phisign #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\PhiSign#3}}%
}%
\def\XINT_expr_func_phiabs #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\PhiAbs#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}}%
}%
\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}}%
}%
\def\ZeckBListNum #1{%
  \expanded\bgroup\expandafter\zeckblist_fork\romannumeral0\xintnum{#1}\xint:
}%
\def\XINT_expr_func_zeckindices #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\ZeckBListNum#3}}%
}%
```

TODO: I have forgotten now but I vaguely remember if compatibility with usage of the defined function in \xintdeffunc is hoped for that it should first expand its argument even though in our context if purely numerical this is unneeded (and f-expansion will end up hitting a brace if the input is a nutple). Adding anyhow. I have other things in mind currently, to examine later, already quite enough hours on this package.

```
\def\Phi@BList#1{\expandafter\expandafter\expandafter
```

```
     \phi@blist_b\expandafter\string\romannumeral`&&@#1;}%
\catcode`[=1 \catcode`]=2 \catcode`\{=12 % }
\def\phi@blist_b #1[\if#1{\expandafter\phi@blist_X % }
                    \else\expandafter\phi@blist_N\fi #1]%
\catcode`[=12 \catcode`]=12 \catcode`\{=1 % }
\def\phi@blist_N #1;{%
    \expandafter\xint_gobble_i\expanded
    \expandafter\phiblist_ab \expanded{{\xintNum{#1}}{0};%
}%
\def\phi@blist_X #1{%
    \expandafter\phi@blist_x\expandafter{\iffalse}\fi
}%
\def\phi@blist_x #1#2;{%
    \expandafter\xint_gobble_i\expanded
    \expandafter\phiblist_ab \expanded{{\xintNum{#1}}{\xintNum{#2}}};%
}%
\def\XINT_expr_func_phiexponents #1#2#3%
{%
    \expandafter #1\expandafter #2\expandafter{%
    \romannumeral`&&@\XINT:NEhook:f:one:from:one
    {\romannumeral`&&@\Phi@BList#3}}%
}%
```

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

9. Interactive code

Extracts to zeckendorf.tex.

```
\input zeckendorfcore.tex
\let\xintfirstoftwo\xint_firstoftwo
\let\xintsecondoftwo\xint_secondoftw
\let\zeckexprmapwithin\XINT:expr:mapwithin
\def\zeckNumbraced#1{{\xintNum{#1}}}
\xintexprSafeCatcodes

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

\newif\ifzeckphimode
\newif\ifzeckindices
\zeckindicestrue
\newif\ifzeckfromN
\zeckfromNtrue
\newif\ifzeckmeasuretimes
\newif\ifzeckevalonly
\newif\ifzeckhex

\def\ZeckCmdP{%
```

```
    \zeckphimodetrue
    \ifzeckindices\ZeckCmdL\else\Zeck@CmdB\fi
}
\let\ZeckCmdp\ZeckCmdP
\def\ZeckCmdZ{%
    \zeckphimodefalse
    \ifzeckindices\ZeckCmdL\else\Zeck@CmdB\fi
}
\let\ZeckCmdz\ZeckCmdZ

\def\PhiTypesetXPrint #1#2{a=#1, b=#2}
\def\ZeckCmdL{%
    \zeckindicestrue
    \ifzeckphimode
      \def\ZeckFromN{\PhiExponents}%
      \def\ZeckToN##1{\PhiTypesetX{\PhiXfromExponents{##1}}}%
    \else
      \def\ZeckFromN{\ZeckIndices}%
      \def\ZeckToN{\ZeckNFromIndices}%
    \fi
}
\let\ZeckCmdl\ZeckCmdL

\def\ZeckCmdB{%
    \zeckindicesfalse
    \zeckhexfalse
    \Zeck@CmdB
}
\def\Zeck@CmdB{%
    \ifzeckphimode
      \ifzeckhex
        \def\ZeckFromN{\PhiBaseHexPhi}%
        \def\ZeckToN##1{\PhiTypesetX{\PhiXfromBaseHexPhi{##1}}}%
      \else
        \def\ZeckFromN{\PhiBasePhi}%
        \def\ZeckToN##1{\PhiTypesetX{\PhiXfromBasePhi{##1}}}%
      \fi
    \else
      \ifzeckhex
        \def\ZeckFromN{\ZeckHexWord}%
        \def\ZeckToN{\ZeckNfromHexWord}%
      \else
        \def\ZeckFromN{\ZeckWord}%
        \def\ZeckToN{\ZeckNfromWord}%
      \fi
    \fi
}
\let\ZeckCmdW\ZeckCmdB
\let\ZeckCmdb\ZeckCmdB
\let\ZeckCmdw\ZeckCmdB

\def\ZeckCmdC{%
```

```
    \zeckindicesfalse
    \zeckhextrue
    \Zeck@CmdB
}
\let\ZeckCmdc\ZeckCmdC

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

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

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

\def\ZeckCmdH{\immediate\write128{\ZeckHelpPanel}}
\let\ZeckCmdh\ZeckCmdH

\ZeckCmdL

\def\ZeckCommands{Enter input or command
                  (q, z, p, l, w, b, c, t, e, @, or h for help).}
\def\ZeckPrompt{%
  \ifzeckevalonly
    <<<Eval-only (e to quit)>>>^^J%
    [IN] expression =
  \else
   \ifzeckfromN
    \ifzeckphimode
      \ifzeckindices <convert to Bergman phi-exponents>^^J%
      \else
        \ifzeckhex
                   <convert to Bergman hexphi-representation>^^J%
        \else
                   <convert to Bergman phi-representation>^^J%
        \fi
      \fi
      \ZeckCommands^^J
      [IN] a + b phi =
    \else
      \ifzeckindices <convert to Zeckendorf indices>^^J%
      \else
```

```
        \ifzeckhex
                     <convert to Zeckendorf hex-word>^^J%
        \else
                     <convert to Zeckendorf word>^^J%
        \fi
      \fi
      \ZeckCommands^^J
      [IN] N =
    \fi
   \else
    \ifzeckphimode <convert to a + b phi>^^J
      \ZeckCommands^^J
      [IN]
       \ifzeckindices phi exponents =
       \else
         \ifzeckhex
                   hexphi-representation =
         \else
                      phi-representation =
         \fi
       \fi
    \else          <convert to integer>^^J
      \ZeckCommands^^J
      [IN]
       \ifzeckindices indices =
       \else
         \ifzeckhex
                      hex word =
         \else
                      binary word =
         \fi
       \fi
    \fi
   \fi
  \fi
}
\newlinechar10
\immediate\write128{}
\immediate\write128{Welcome to Zeckendorf 0.9d (2025/11/16, JFB).}

\def\ZeckHelpPanel{Commands (lowercase also):^^J
 Q to quit. Also X.^^J
 H for this help.^^J
 Z to switch to Zeckendorf-mode (starting default).^^J
 P to switch to phi-mode.^^J
 L for indices or exponents.^^J
 W for binary words or reps. Also B.^^J
 C for hexadecimal words or reps.^^J
 T to toggle the direction of conversions.^^J
 E to toggle to and from  \string\xinteval-only mode.^^J
 @ to toggle measurement of execution times.^^J
^^J
```

```
 - binary words, phi-representations, are parsed only by \string\edef.^^J
 - all other inputs are handled by \noexpand\xinteval so for example one^^J
\space\space
   can use 2^100 or 100! or binomial(100,50).  And a list of indices^^J
\space\space
   or exponents can be for example seq(3*a+1, a=0..10).^^J
^^J
\space\space The fib() function computes Fibonacci numbers.^^J
\space\space The character $ serves as symbol for Knuth multiplication.^^J%
**** empty input is not supported!
     no linebreaks in input! ****}

\immediate\write128{\ZeckHelpPanel}

\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}

\xintFor*#1 in {0123456789}\do{%
   \expandafter\def\csname ZeckCmd#1\endcsname{%
      \immediate\write128{%
** Due to under-funding, a lone #1 is not accepted. Inputs must have^^J%
** two characters at least.  Think about a donation? Try 0#1.}}
}%
\xintloop
\message{\ZeckPrompt}
\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
```

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

With 0.9b the time needed for parsing the input was not counted, but this meant that measuring in the evaluation-only mode always printed 0.0s.

0.9c has refactored here entirely.

```
   \ifzeckmeasuretimes\xintresettimer\fi
   \if1\ifzeckevalonly0\fi\ifzeckfromN0\fi\ifzeckindices0\fi1%
```

```
      \edef\ZeckIn{{\zeckbuf}}%
   \else
      \expandafter\def\expandafter\ZeckIn\expandafter{%
      \romannumeral0\xintbareeval\zeckbuf\relax}%
```

0.9c uses `\xinteval`. It adds phi-mode to the interactive interface, but as `1/phi` or anything doing an operation will inject ``raw xintfrac format'', we have to be careful about that, because we use `\PhiExponents` and `\PhiBasePhi` which are assuming being used with either an integer `a` or a pair `{a}{b}`. Using here some core level auxiliary from xintexpr to avoid a dozen lines like what was done for `\Phi@BList`. For this to work we need a variant of `\xintNum` which outputs with extra braces. This was for the author a refreshing journey to revisit forgotten deep code written years ago for xintexpr. But it would be more efficient to do something akin to the `\Phi@BList` business.

By the way we have to do this not only for phi-mode, but also for integer-mode, because some input such as `1e40` will be internally `1[40]` which `\ZeckIndices` does not understand as it does not apply `\xintNum`. In fact any input doing an operation such as an addition will be in ``raw xintfrac format'' internally. So we have to do a normalization also for lists of exponents or indices.

```
      \ifzeckevalonly\else
         \expandafter\def\expandafter\ZeckIn
           \expanded
           \expandafter\zeckexprmapwithin
           \expandafter\zeckNumbraced\ZeckIn
```

For lists of exponents and indices the predefined macros expect comma separated lists. We can either "print" using (full) `\xinteval`, or use `\xintListwithSep`, or write a little helper requiring only `\edef` expansion. We add one level of bracing removed later.

```
         \ifzeckfromN\else
            \expandafter\def\expandafter\ZeckIn\expandafter{%
              \expandafter{\romannumeral0\xintlistwithsep,\ZeckIn}%
            }%
         \fi
      \fi
   \fi
   \immediate\write128{%
     [OUT] \ifzeckevalonly
       \expanded\expandafter\XINTexprprint\expandafter.\ZeckIn
     \else
       \expandafter\ZeckConvert\ZeckIn
     \fi
   }%
   \ifzeckmeasuretimes
     \edef\tmp{\xinttheseconds}%
     \immediate\write128{%
        \ifzeckevalonly Evaluation \else Conversion \fi
        took \tmp s%
     }%
   \fi
  \fi
\fi
\ZeckShouldISayOrShouldIGo
\repeat
```

```

\immediate\write128{Bye. Session was saved to log file (hard-wrapped too, alas).}
\bye
```

10. LaTeX code

Extracts to zeckendorf.sty.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{zeckendorf}
   [2025/11/16 v0.9d Zeckendorf and base-phi representations of big integers (JFB)]%
\RequirePackage{xintexpr}
\RequirePackage{xintbinhex}% superfluous if with xint 1.4n or later
\input zeckendorfcore.tex
\ZECKrestorecatcodesendinput%
```