CSCSTI 50.05.17
UDC 004.627

Y. V. Shevchuk

Vbinary: variable length integer coding revisited

Abstract. The article introduces Vbinary, a parametrized variable-length prefix integer coding. The coding is considered by means of examples in comparison with existing codings, including Golomb/Rice and Elias codings. A naming schema is proposed that allows to specify the coding parameters concisely.

Vbinary coding utilizes unusual n-ary extension technique which makes the coding versatile, usable for both bit-based and byte-based data streams. By varying parameters, Vbinary coding can be made efficient for small numbers or large numbers, tailored to specific data distribution, tuned for efficient encoding and decoding. Potential uses for the coding are network protocols, on-disk and in-memory data representation, and final stages of data compression algorithms.

Key words and phrases: coding of integers, variable length coding, prefix code, parametrized coding, data compression.

2010 *Mathematics Subject Classification*: 94A45; 68P30, 94B60

Introduction

In programming practice we routinely define data fields and ponder on choosing the proper field width. This choice is often tough, especially in network protocol headers where you would have to upgrade all protocol speakers if you change your mind afterwards. RAM, disk space and network bandwidth are not free so the natural urge is to use as narrow field as possible, but then you start thinking about future extensions and in general "you shouldn't have arbitrary limits" [1]. The field width choice can take unreasonable efforts with unsatisfying result as any fixed length choice is a compromise. Thus programmers often resort to some kind of variable-length coding.

The research is a part of the Russian Academy of Sciences research project AAAA-A17-117040610378-6.

10.25209/2079-3316-2018-9-4-239-252

TABLE 1. Simple Vbinary codings

integer value	vbinary2x	vbinary2x2	vbinary2x3x
0	00	00	00
1	01	01	01
2	10	10	10
3	11 00	11 00	11 000
4	11 01	11 01	11 001
5	11 10	11 10	11 010
6	11 11 00	11 11	11 011
7	11 11 01	*none*	11 100
8	11 11 10	*none*	11 101
9	11 11 11 00	*none*	11 110
10	...	*none*	11 111 000
...	...	*none*	...

A number of variable-length coding schemas exist already: Unary coding, Golomb coding [2], Elias γ, δ, ω codings [3] are the most popular examples. In practice however you often find them unsuitable for the task in hand: bitwise coding not fitting your data structure, codeword length distribution too far from your data probability distribution, encoding and decoding cost unacceptable for your application. As a result, the most popular solution is the variable-byte coding (e.g. [14]) which is suboptimal in terms of coding efficiency and encoding cost but widely accepted.

Vbinary coding proposed in this paper is an attempt to improve the situation by providing a parametrized coding which can be tuned to fit the application needs in all three aspects: coding efficiency (number of bits necessary to represent a value), codeword length distribution and encoding/decoding cost.

1. Construction of codes

Let us start with a simple example called vbinary2x (Table 1). As the name says, this coding has the base length of 2 bits— this is the minimal space occupied by codewords in this coding. Of the four values provided by two bits in binary coding, three values are used to represent integers 0..2. The letter x in the name says the width should be extended when the base bit group capacity is exhausted, so the fourth value (binary 11) is used for width extension and says that (at least) two more bits follow. The two added bits are capable to encode three more integers, and then the next extension level is added.

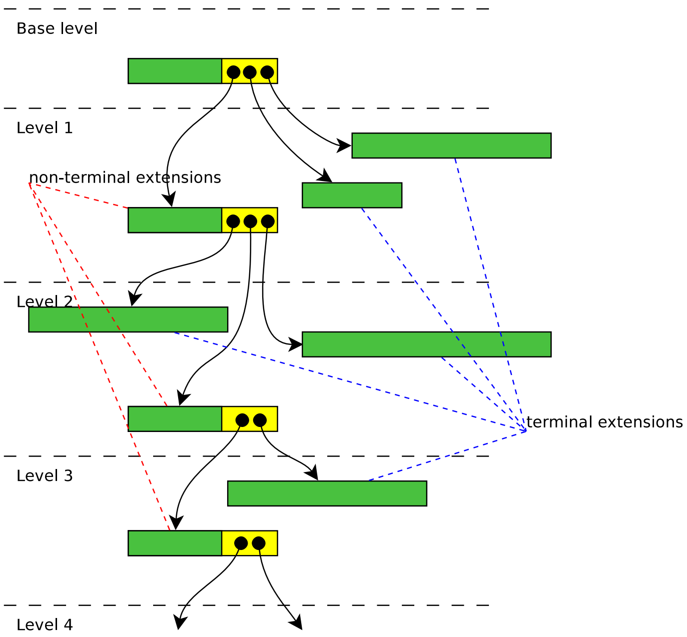

FIGURE 1. Levels of vbinary6x(6x,4,8)(8,6x,10)(6x,7)
Note: at most one non-terminal extension per level

The vbinary2x coding is infinite, adding bit pairs as needed to accomodate more integers (the last extension rule repeats indefinitely). The vbinary2x2 coding (Table 1) is a finite coding, which is signalled by the absence of x in the last extension level width present in the name. The vbinary2x2 coding is only capable of representing 7 integer values. In finite codings, the last codeword is spent to represent a value rather than extension: in this example, the last value (6) occupies the code 1111 which in vbinary2x was used for extension.

Vbinary coding name is basically a sequence of extension rules, although the rules can be more complex than those already shown. All codings in Table 1 have only one value reserved for extension at every level. Actually we can reserve up to 2^{w_i} values for extension, where w_i is the bit width of the current level. However, only one of the multiple extensions is permitted to be further extended (Figure 1) so as to keep the name structure linear. We will show the arity increase providing for more powerful codings. As we continue with more complex examples, please refer to the Vbinary name

TABLE 2. vbinary1x(8,1x) is the same as Golomb(256)

integer value	vbinary1x(8,1x)	bit count
0	0 00000000	9
...	...	9
255	0 11111111	9
256	1 0 00000000	10
...	...	10
511	1 0 11111111	10
512	1 1 0 00000000	11
...	...	11
767	1 1 0 11111111	11
...	...	≥12

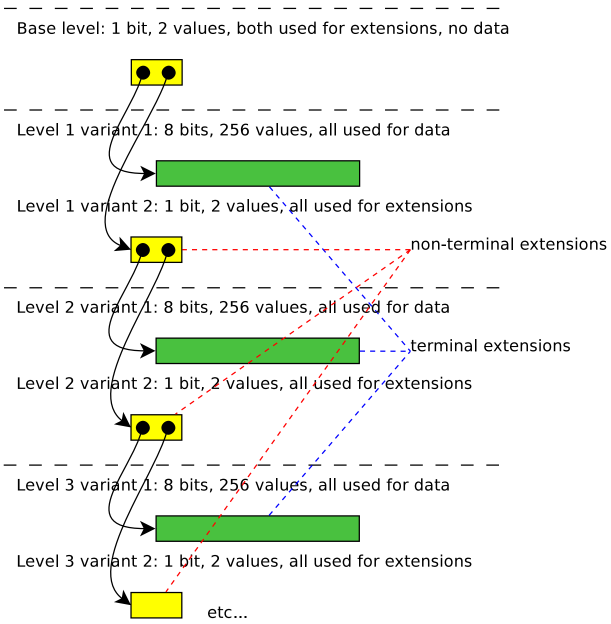
Base level: 1 bit, 2 values, both used for extensions, no data
Level 1 variant 1: 8 bits, 256 values, all used for data
Level 1 variant 2: 1 bit, 2 values, all used for extensions
non-terminal extensions
Level 2 variant 1: 8 bits, 256 values, all used for data
terminal extensions
Level 2 variant 2: 1 bit, 2 values, all used for extensions
Level 3 variant 1: 8 bits, 256 values, all used for data
Level 3 variant 2: 1 bit, 2 values, all used for extensions
etc...

FIGURE 2. Levels of vbinary1x(8,1x)

syntax provided in Appendix A.

Consider the vbinary1x(8,1x) coding (Table 2). The second extension level specification in the form (8,1x) means there are two variants of the second level and, respectively, two values reserved for extensions at the base level (Figure 2). As the base level is only one bit wide, there are

TABLE 3. vbinary1x2x(2,1x)(a1,a0) is equivalent to Elias γ

integer value	vbinary1x2x(2,1x)(a1,a0)	bit count
0	0	1
1	1 00	3
2	1 01	3
3	1 10 00	5
4	1 10 01	5
5	1 10 10	5
6	1 10 11	5
7	1 11 0 000	7
8	1 11 0 001	7
9	1 11 0 010	7
10	1 11 0 011	7
11	1 11 0 100	7
12	1 11 0 101	7
13	1 11 0 110	7
14	1 11 0 111	7
15	1 11 1 0 0000	9
...	...	≥9

no codewords assigned to data values at base level, both codewords are assigned to extensions.

The first extension is a terminal extension capable of representing 2^8 data values, and the second is a non-terminal extension one bit wide which behaves exactly as the base level. As the last extension specification contains x, the coding is infinite, the last extension (8,1x) is applied repeatedly.

The next example is vbinary1x2x(2,1x)(a1,a0). This infinite coding has three level specifications—base, 1st, 2nd— and features an additive repeater element (a1,a0). The repeater element is applied when generating levels 3rd, 4th, etc., adding 1 and 0 respectively to the last level widths to generate the new level. That is, the following codings are all exactly the same:

vbinary1x2x(2,1x)(a1,a0)

vbinary1x2x(2,1x)(3,1x)(a1,a0)

vbinary1x2x(2,1x)(3,1x)(4,1x)(a1,a0)

vbinary1x2x(2,1x)(3,1x)(4,1x)(5,1x)(a1,a0)

...

The effect of the additive repeater is doubling the number of values accomodated by every subsequent level (Table 3): $4, 8, 16, \ldots$

TABLE 4. vbinary1x is the same as Unary and as Golomb(1)

integer value	vbinary1x	bit count
0	0	1
1	1 0	2
2	1 1 0	3
3	1 1 1 0	4
4	1 1 1 1 0	5
5	1 1 1 1 1 0	6
...	...	...

The general repeater syntax (mKaC) allows for growing widths in a mix of geometric and arithmetic sequences: $w_{i+1} = Kw_i + C$, where the factor K and the additive component C can be integers or common fractions. Decimal fractions are not used as they are inconvenient to handle on small processors without FPU.

Slow width growth is supported as follows. The calculations are performed in fixed point arithmetics with a scaling factor[1] of 16. The integer part of the calculation result is used for width, but the whole result with the fractional part is kept behind the scenes and used as the input for the next growth step. This way, the repeater (a1/3) will result in width increase by 1 every three steps, while the repeater (m4/3) will provide exponential growth at a rate of $(4/3)^n$.

The order of value assignment at a level is as follows:

(1) use all widths defined for the level in the order written. In the width marked by x, skip the values reserved for extensions.

(2) use the values reserved for extensions while handling the widths of the next level according to step 1.

That is all to Vbinary name syntax and semantics. The syntax covers a vast family of codings which we will now consider in comparison with other variable length integer codings.

2. Relation to other variable length codings

Parametrized codings often have Unary coding as a special case [2, 5]. Vbinary is no exception: vbinary1x is the same as Unary coding (Table 4).

We have seen that vbinary1x(8,1x) is identical to Golomb(256) coding. Generalizing, vbinary1x(n,1x) is identical to Golomb(2^n).

[1]small scaling factor is chosen to keep computations efficient on 8-bit CPUs

Table 5. vbinary1x(7,1x)(a7,a0) is close to variable-byte coding

integer value	vbinary1x(7,1x)(a7,a0)	bit count
0	0 0000000	8
1	0 0000001	8
...	...	...
127	0 1111111	8
128	1 0 0000000 0000000	8
...	...	...
255	1 0 0000000 1111111	16
256	1 0 0000001 0000000	16
...	...	...
16511	1 0 1111111 1111111	16
16512	1 1 0 0000000 0000000 0000000	24
...	...	...
2113663	1 1 0 1111111 1111111 1111111	24
...	...	...

We have also considered vbinary1x2x(2,1x)(a1,a0) coding (Table 3), which is equivalent to Elias γ coding [3]. Elias γ differs by inversion of the unary part and value range shift by 1.

Another relative is the variable-byte coding which does not seem to have a known author but is widely used in software. The variety of the coding implemented in Git [14] called *varint* is equivalent to vbinary1x(7,1x)(a7,a0) (Table 5), though with different bit order: Vbinary puts Unary prefix at the beginning of codewords, while *varint* has the same unary prefix spread over MSBs of the bytes comprising a codeword.

The *start-step-stop* coding [4] is a finite parametrized prefix code organized like Elias γ coding, but with binary part growing not by 1-bit steps but by steps specified in the parameters. This coding has equivalents in Vbinary family: for example, the start-step-stop(3,2,9) coding coincides with vbinary1x(3,1x)(5,1x)(7,9). We cannot a repeater in the definition because Vbinary repeaters lack stop conditions. vbinary1x(3,1x)(a2,a0) is close to start-step-stop(3,2,9) but is an infinite coding and so its codewords are longer by one bit in the number range 168..679.

Start/Stop coding [5] is a finite parametrized prefix code. The coding is parametrized by a vector of widths of binary fields which provides for matching the coding to specific data distribution. Variable length prefix code property is achieved using a spread-out equivalent of unary coding. Start/Stop coding is not exactly expressable in Vbinary, but is ideologically closer to Vbinary than other codes considered so far.

All codings we compared so far employ Unary coding to implement the prefix code property. Unary coding can be expressed in Vbinary, so these codings mostly have Vbinary equivalents. However, the n-ary extension technique used in Vbinary can do much more than just implement Unary coding.

Codings that utilize something different from Unary to implement variable length prefix, e.g Levenshtein $W2'$ coding [11], Elias δ and ω codings [3], or their successors [7] [8], have no equivalents in Vbinary family. Citing [4], "these encodings have nice asymptotic properties for very large integers", but they are not particularly efficient with small integers.

There was a substantial work on variable-length codings for small integers summarized in [9]. These codes have no Vbinary equivalents. They demonstrate an improvement over Elias γ coding for certain data distributions but lack the flexibility achievable in parametrized codings.

Small integer representation is where Vbinary shine, as we will show in the next section, though large integers can be encoded efficiently in Vbinary as well.

3. Small numbers example

Suppose we are designing a network protocol and define a header field that distinguishes between 17 types of records. We would also like to reserve room for future protocol extensions. The straightforward solution is to define a field 5 bits wide with fixed-width binary coding. We will be able to add 15 more records in the future, but not more. A better solution is to use a variable-length coding: we will get unlimited extensibility and can also take advantage of non-uniform record frequency distribution, assigning shortest codewords to the most frequent records. Table 6 summarizes performance of a number of variable-length codings in this scenario. For any third-party variable-length coding shown in the table there is a Vbinary coding that is better by at least one characteristic. Looking at Vbinary codeword table (Table 7) one can see the codeword length distribution and choose the coding that fits the record frequency distribution best. If necessary, more ad-hoc Vbinary codings can be devised and evaluated using the freely available tool [15].

4. Byte-based examples

There are applications where byte-aligned codings are considered preferable to bit-aligned for speed reasons [12]. One can construct byte-

TABLE 6. Performance of codings ($n \leqslant 16$) in Small Numbers Example

Encoding	min	avg	median	max	limit
binary	5	5	5	5	32
Unary	1	8.5	8	17	unlimited
Elias γ	1	5.94	7	9	unlimited
Elias δ	1	6.53	8	9	unlimited
Elias ω	1	5.76	7	11	unlimited
Golomb(16)	5	5.06	5	6	unlimited
Golomb(8)	4	4.59	5	6	unlimited
Golomb(4)	3	4.65	5	7	unlimited
vbinary4x1x	4	4.1	4	6	unlimited
vbinary4x2x	4	4.24	4	6	unlimited
vbinary3x(3,3x)	3	4.94	6	6	unlimited
vbinary3x(2,2,2)	3	4.41	5	5	17
vbinary2x(3,4x)	2	5.06	5	6	unlimited
vbinary1x2x(2,3x)	1	5.59	6	8	unlimited
vbinary1x2x(2,2,3x)3x	1	5.06	5	6	unlimited

TABLE 7. Vbinary codings used in Small Numbers Example

integer value	vbinary 4x1x	vbinary 4x2x	vbinary 3x(3,3x)	vbinary 3x(2,2,2)	vbinary 2x(3,4x)	vbinary 1x2x(2,3x)	vbinary 1x2x(2,2,3x)3x
0	0000	0000	000	000	00	0	0
1	0001	0001	001	001	01	1 00	1 00
2	0010	0010	010	010	10 000	1 01	1 01 00
3	0011	0011	011	011	10 001	1 10 00	1 01 01
4	0100	0100	100	100	10 010	1 10 01	1 01 10
5	0101	0101	101	101 00	10 011	1 10 10	1 01 11
6	0110	0110	110 000	101 01	10 100	1 10 11	1 10 00
7	0111	0111	110 001	101 10	10 101	1 11 000	1 10 01
8	1000	1000	110 010	101 11	10 110	1 11 001	1 10 10
9	1001	1001	110 100	110 00	10 111	1 11 010	1 10 11
10	1010	1010	110 101	110 01	11 0000	1 11 011	1 11 000
11	1011	1011	110 110	110 10	11 0001	1 11 100	1 11 001
12	1100	1100	110 111	110 11	11 0010	1 11 101	1 11 010
13	1101	1101	111 000	111 00	11 0011	1 11 110 00	1 11 011
14	1110	1110	111 001	111 01	11 0100	1 11 110 01	1 11 100
15	1111 0	1111 00	111 010	111 10	11 0101	1 11 110 10	1 11 101
16	1111 1 0	1111 01	111 011	111 11	11 0110	1 11 110 11	1 11 110
17	1111 1 1 0	1111 10	111 100	*none*	11 0111	1 11 111 000	1 11 111 000
18	1111 1 1 1 0	1111 11 00	111 101	*none*	11 1000	1 11 111 001	1 11 111 001

TABLE 8. vbinary1x(7,15) is a finite byte aligned coding

integer value	vbinary1x(7,15)	bit count
0	0 0000000	8
...	...	...
127	0 1111111	8
128	1 000000000000000	16
...	...	...
32895	1 111111111111111	16

TABLE 9. vbinary8x(8,16,24,56) is a finite byte aligned coding

integer value	vbinary8x(8,16,24,56)	bit count
0	00000000	8
1	00000001	8
...	...	...
251	11111011	8
252	11111100 00000000	16
...	...	...
507	11111100 11111111	16
508	11111101 0000000000000000	24
...	...	...
$508 + 2^{16} - 1$	11111101 1111111111111111	24
$508 + 2^{16}$	11111110 000000000000000000000000	32
...	...	...
$508 + 2^{16} + 2^{24} - 1$	11111110 111111111111111111111111	32
$508 + 2^{16} + 2^{24}$	11111111 000...(56 zero bits)...	64
...	...	...
$508 + 2^{16} + 2^{24} + 2^{56} - 1$	11111111 111...(56 one bits)...	64

aligned codings in Vbinary family. One example is vbinary1x(7,1x)(a7,a0) coding considered above (Table 5), but Vbinary allows to construct codings that accomodate more values in the same number of bytes (Table 8) or allow faster encoding/decoding by avoiding byte-by-byte processing (Table 9). More variants can be invented with specific purpose for the coding in mind.

5. Some properties of the Vbinary coding

5.1. Lexicographic order

Vbinary codeword sequence is lexicografically ordered if and only if all extensions are made from the last binary group of the level. For example, both vbinary8x(8,16,24,56x)64 and vbinary8x(8x,16,24,56)112 are finite codings with maximum length of 128 bits. The first is lexicografically ordered and has maximum value around 2^{64}, while the second is not lexicografically ordered and has maximum value around 2^{112}. So, lexicographic order is achieved at the cost of coding efficiency. If you trade in lexicographic order for coding efficiency you will not be able to compare Vbinary-encoded numbers without decoding them to binary first, which is not a huge loss if the decoding cost is small.

5.2. Codeword length monotonicity

In all variable-length codings known to the author the codeword length monotonically increases with the value of encoded integer: $L(v(n)) \leq L(v(n+1))$ for any n, where L is the bit length function, $v(n)$ is a variable-length encoding function, n is a non-negative integer. In Vbinary you have the freedom to define a coding such that $L(v(n))$ is not monotonic. For example, vbinary8x(8,16,24,56,2) has four 16-bit codewords assigned to the higher end of the range, while the maximum codeword length is 64 bits.

6. Conslusion

Vbinary is a highly parametrized variable length integer coding with prefix property. Vbinary coding uses unusual n-ary extension technique which makes the coding versatile, usable for both bit-based and byte-based data streams. By varying parameters, Vbanary coding can be made efficient for small numbers or large numbers, tailored to specific data distribution, tuned for efficient encoding and decoding. We compared Vbinary to other variable length codings and got the impression that Vbinary with parameters chosen for the specific task will outperform any general-purpose variable-length coding. No attempt to prove that has been made though.

Potential uses for Vbinary coding are network protocols, on-disk and in-memory data representation. Probably the coding can be useful in data compression applications where Golomb/Rice coding or other variable-length integer codings are used today [13] [10].

We have provided a naming schema which makes it convenient for programmers to use Vbinary coding instead of inventing ad hoc variable length codings. A software tool for experimentation with Vbinary coding parameters is available [15].

References

[1] R. M. Stallman, D. Betz, J. Edwards. *BYTE Interview with Richard Stallman*, July, 1986. URL ↑239

[2] S. W. Golomb. "Run-length encodings", *IEEE Transactions on Information Theory*, **12**:3 (1966), pp. 399–401. URL ↑240,244

[3] P. Elias. "Universal codeword sets and representations of the integers", *IEEE Transactions on Information Theory*, **21**:2 (1975), pp. 194–203. doi ↑240,245,246

[4] E. R. Fiala, D. H. Greene. "Data compression with finite windows", *CACM*, **32**:4 (1989), pp. 490–505. doi ↑245,246

[5] S. Pigeon. "Start/stop codes", Data Compression Conference (Snowbird, Utah, 2001), 2001, 0511. doi URL ↑244,245

[6] A. S. Fraenkel, S. T. Klein. "Robust universal complete codes for transmission and compression", *Discrete Applied Mathematics*, **64**:1 (1996), pp. 31–55. doi ↑

[7] M. Nangir, H. Behroozi, M. R. Aref. "A new recursive algorithm for universal coding of integers", *Journal of Information Systems and Telecommunication*, **3**:1 (9) (2015), pp. 1–6 10.7508/jist.2015.01.001. ↑246

[8] J. Nelson Raja, P. Jaganathan, S. Domnic. "A new variable-length integer code for integer representation and its application to text compression", *Indian Journal of Science and Technology*, **8**:24 (September 2015). doi ↑246

[9] P. Fenwick. "A note on variable-length codes with constant Hamming weights", *Journal of Universal Computer Science*, **21**:9 (2015), pp. 1136–1142. doi ↑246

[10] P. Fenwick. "Burrows-Wheeler compression with variable length integer codes", *Software Practice and Experience*, **32**:13 (2002), pp. 1307–1316. doi ↑249

[11] V. I. Levenshtein. "On the redundancy and delay of decodable coding of natural numbers", *Probl. Cybern.*, **20** (1968) (in Russian). ↑246

[12] F. Scholer, H. E. Williams, J. Yiannis, J. Zobel. "Compression of inverted indexes For fast query evaluation", *SIGIR '02 Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (Tampere, Finland, August 11–15, 2002), ACM, 2002, pp. 222–229. doi ↑246

[13] Josh Coalson. *FLAC documentation: format overview*. URL ↑249

[14] J. C. Hamano. *git/varint.c*, 2012. URL ↑240,245

[15] Yu. V. Shevchuk. *vbinary/vbinary-eval.pl*, 2018. URL ↑246,249

[16] D. Crocker, P. Overell, *Augmented BNF for Syntax Specifications: ABNF*, RFC 5234, 2008. URL ↑251

Appendix A. Vbinary name syntax in Augmented BNF [16]

```
vbinary-name  = "vbinary" basewidth
vbinary-name /= "vbinary" basewidth repeat-rule
vbinary-name /= "vbinary" basewidth "x" [*midlevel lastlevelx]
vbinary-name /= "vbinary" basewidth "x" [*midlevel] lastlevel repeat-rule

midlevel      = width "x"
      ; 1-ary non-terminal extension
midlevel     /= "(" 1*(width ",") width "x" ")"
midlevel     /= "(" *(width ",") width "x," *(width ",") width ")"
      ; n-ary non-terminal extensions

lastlevelx    = width "x"
      ; 1-ary non-terminal extension

lastlevel     = width
      ; 1-ary terminal extension
lastlevel    /= "(" 1*(width ",") width ")"
      ; n-ary terminal extension
lastlevel    /= "(" 1*(width ",") width "x" ")"
lastlevel    /= "(" *(width ",") width "x," *(width ",") width ")"
      ; n-ary non-terminal extensions

repeat-rule   = addmul
      ; for 1-ary lastlevel
repeat-rule  /= "(" 1*(addmul ",") addmul ")"
      ; for n-ary lastlevel

addmul        = add / mul / mul add
add           = "a" increment
mul           = "m" factor

rational      = 1*DIGIT
rational     /= 1*DIGIT "/" 1*DIGIT

basewidth     = width
width         = 1*DIGIT
increment     = rational
factor        = rational
```

Received 09.11.2018
Revised 11.11.2018
Published 30.11.2018

Recommended by *prof. S. V. Znamenskij*

About the author:

Yury Vladimirovich Shevchuk

Head of telecommunication laboratory Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Ph.D. Interest areas: system programming, digital electronics, computer networks, sensor networks, wide area monitoring and control, distributed programming

iD 0000-0002-2327-0869

e-mail: sizif@botik.ru

Sample citation of this publication:

Yury Shevchuk. "Vbinary: variable length integer coding revisited". *Program Systems: Theory and Applications*, 2018, **9**:4(39), pp. 239–252.

doi 10.25209/2079-3316-2018-9-4-239-252

URL `http://psta.psiras.ru/read/psta2018_4_239-252.pdf`

The same article in Russian: doi 10.25209/2079-3316-2018-9-4-477-491

Эта же статья по-русски: doi 10.25209/2079-3316-2018-9-4-477-491