Noun Serialization

Brian Klatt ~ticlys-monlun,
N. E. Davis ~lagrev-nocfep*
Zorp Corp

Abstract

Noun serialization is commonly used for Nock communication, both between instances like Urbit ships and with the runtime and the Unix host operating system. This article describes the principal convention for representing a noun in compressed form as a byte array and its runtime affordance.

Contents

*We gratefully acknowledge the original architect of `+jam` and `+cue`, Curtis Yarvin `~sorreg-namtyv`. The earliest implementation, in `jam.watt`, dates to ~2011.6.11; Yarvin has indicated that an earlier draft no longer survives.

Address author correspondence to `~lagrev-nocfep`.

1 Introduction

The Nock combinator calculus deals in nouns and does not know about bit encodings or memory layouts. A Nock interpreter (runtime) must, however, deal with the practicalities. In other words, there must be a way of writing down an abstract binary tree (consisting of cells/pairs and atoms) as an actual, physical array of bits in memory. Every possible Nock noun can be represented as a finite sequence of bytes (an atom), and there are multiple ways to do so.[1]

A noun serialization strategy is rather like a Gödel numbering in that it systematically encodes a mathematical object (a noun) as a number (an atom). Unlike Gödel numbering, which classically serially encodes the symbols of mathematical statements, noun serialization encodes a binary tree structure.[2] Because a noun may be any atom—and atoms cannot have leading zeroes—both structure and value need to be unambiguously encoded and cannot be simply delimited (as by a `0` bit or similar). Run-length serialization, with or without references, forms the basis of practical strategies to encode a noun as an atom. This embodies a tradeoff between simplicity, size, and speed. We describe this strategy and some pedagogical variants which afford insight to the process. Likewise, deserialization is the inverse operation of converting a serialized atom back into the original noun, and must be unambiguously reversible. Noun serialization is essential for Nock communication, both between Nock execution layers such as Vere and NockVM, and between the runtime and the host operating system.

2 Naïve Serialization

A noun may be an atom or a cell, and so we first propose to label nodes by type, with `0` for atom and `1` for cell.

[1]The converse is not true: not every atom represents a valid deserialization or conversion from a graph encoding.

[2]This also echoes the way that S-expressions are encoded in Lisp.

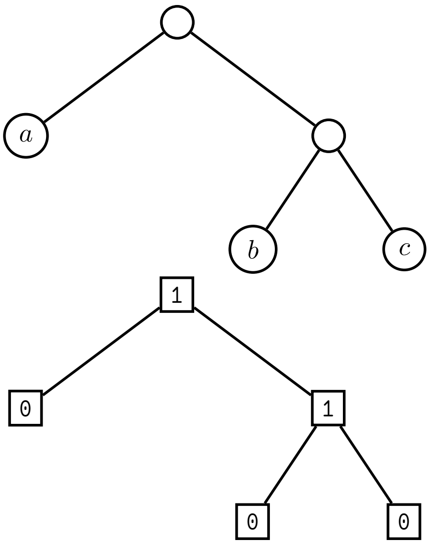

Figure 1: The noun `[a b c]` (the same as `[a [b c]]`) as a binary tree and its metadata labeling in the simplest scheme.

The noun in Figure 1 is represented as a binary tree, where each node is labeled with its type (atom or cell) and its value (if applicable). To serialize this noun, we can perform a pre-order traversal of the tree, recording the type and value of each node. The serialized form would be (in LSB order):

```
c 0 b 0 1 a 0 1
```

The issue, however, is that in decoding this bitstream we do not know where an atom value (such as `a`) ends. After all, the point of serialization is to be able to deserialize the bitstream back into the original noun. Thus we need a way to indicate the length of each atom.

2.1 Atom Length Encoding

Let's augment this scheme by providing length information for each atom. One basic idea is to include a string of `0` bits as the least-significant bits (LSB) of the atom which indicates the length of the atom in bits. Since the atom could be zero, we also delimit the length with a `1` bit. (See Table 1 for examples.)

$$a = a_{\ell-1}a_{\ell-2}\ldots a_1 a_0 \xrightarrow[\text{encode}]{} \bar{a} = a_{\ell-1}a_{\ell-2}\ldots a_1 a_0 \; \texttt{1} \; \texttt{0}^n,$$

where $\bar{a}$ is the encoded atom a and a has length ℓ bits.[3] This is a unary run-length encoding (RLE) using zeros.[4]

Table 1: Example atom encodings with length information appended as trailing zeros (unnamed encoding). Elided digits are shown in light gray for clarity.

Atom a	**Length (bits)**	**Encoded Form $\bar{a}$**
`0b0`	1	`0b01`
`0b1`	1	`0b110`
`0b10`	2	`0b10100`
`0b11`	2	`0b11100`
`0b101`	3	`0b1011000`
`0b111`	3	`0b1111000`
`0x70`	7	`0b111000010000000`
`0xff`	8	`0b11111111100000000`
`0x1234`	13	`0b100100011010010000000000000`

We can make one further optimization: since the most significant bit (MSB) of the atom is always `1` except in the case of zero (which is `0b1` with the leading zero trimmed), we can omit it from the encoded form. This saves one bit for every non-zero atom. Let's call this scheme `+fat` encoding. (See Table 2 for examples.)

`+fat` Serialization Algorithm

- **Encode**. Write[5] ℓ bits of `0`, where ℓ is the length of a in

[3]We will use the overbar notation $\bar{a}$ throughout to notionally indicate the encoded form of a, regardless of the method used.

[4]Zero is something of a "special case": it has no length and leading zeros are trimmed, so its encoding consists only of the delimiter, `0b1`.

[5]We always say this from the LSB, so from the right as written on the page.

unary, followed by `1`. The next ℓ bits are the atom a in binary without the leading `1`.

- **Decode**. Read the unary length ℓ by counting `0` bits until the first `1` bit. Read the next ℓ bits to get the atom a without the leading `1`. Prepend a `1` bit to a unless ℓ is zero, in which case the atom is zero.

Table 2: Example atom encodings (`+fat` encoding). Elided digits are shown in light gray for clarity.

Atom a	**Length (bits)**	**Encoded Form $\bar{a}$**
`0b0`	1	`0b1`
`0b1`	1	`0b110`
`0b10`	2	`0b10100`
`0b11`	2	`0b11100`
`0b101`	3	`0b1011000`
`0b111`	3	`0b1111000`
`0x70`	7	`0b1110000010000000`
`0xff`	8	`0b11111111100000000`
`0x1234`	13	`0b100100011010010000000000000`

The primary problem with the `+fat` scheme is that the encoding of an atom is about twice the bit length of the original atom. The next optimization is to use the bit length of the unary bit length, which is logarithmic in the bit length.

$$a = \underbrace{a_{\ell-1}a_{\ell-2}\ldots a_1 a_0}_{\ell = \underbrace{\ell_{h-1}\ell_{h-2}\ldots\ell_1\ell_0}_{h}}$$

$$a = a_{\ell-1}a_{\ell-2}\ldots a_1 a_0 \xrightarrow[\texttt{+mat}\text{ encode}]{}$$

$$\bar{a} = \underbrace{a_{\ell-1}a_{\ell-2}\ldots a_1 a_0}_{\ell}\underbrace{\ell_{h-2}\ell_{h-1}\ldots\ell_1\ell_0}_{h-1}\,\mathtt{1}\,\underbrace{\mathtt{0}\ldots\mathtt{0}}_{h},$$

where ℓ is the bit length of a and h is the "hyper bit length" of a. This yields a bit length encoding of $1 + 2\log_2(\ell)$ bits rather than 2ℓ bits. This scheme is called `+mat` encoding, and it is currently in use as an internal auxiliary format for other serializations.

`+mat` Serialization Algorithm

- **Encode**. Write the number h of `0` bits in unary, followed a leading `1` and then $h - 1$ bits for the low bits of ℓ. The next ℓ bits are the atom a in binary.
- **Decode**. Read the unary length of the length h by counting `0` bits (from the LSB/right) until the first `1` bit. Read the next $h-1$ bits to get $\ell-1$, the bit length of the atom. Finally, read the next ℓ bits (adding the leading `1`) to get the atom a.

Some worked examples of `+mat` encoding are included in Table 3.

The reference implementation of `+mat` encoding in Hoon is as follows:

```
++  mat
  ~/  %mat
  |=  a=@
  ^-  [p=@ q=@]
  ?:  =(0 a)
    [1 1]
  =+  b=(met 0 a)
  =+  c=(met 0 b)
  :-  (add (add c c) b)
  (cat 0 (bex c) (mix (end [0 (dec c)] b)
   ↳                   (lsh [0 (dec c)] a)))
```

Table 3: Example atom encodings (`+mat` encoding). The encoded form is the same result as `q:(mat a)` in Hoon.

Atom a	**Length (bits)**	**Mathematics**	**Encoded Form $\bar{a}$**
`0b0`	$\ell = 1$	$\underbrace{\cancel{a_{\ell-1}} \cdots \cancel{a_0}}_{\ell=0} \quad \mathtt{1}\underbrace{\cancel{\mathtt{0}}\ldots\cancel{\mathtt{0}}}_{h=0}$	`0b1`
`0b1`	$\ell = 1$	$\underbrace{\mathtt{1}}_{\ell=1}\underbrace{\cancel{\ell_{h-2}} \cdots \cancel{\ell_0}}_{h-1=0} \quad \mathtt{1}\underbrace{\mathtt{0}}_{h=1}$	`0b110`
`0b10`	$\ell = 2$	$\underbrace{\mathtt{10}}_{a=2}\underbrace{\mathtt{0}}_{\text{low bits of }\ell} \quad \mathtt{1}\underbrace{\mathtt{00}}_{h=2}$	`0b100100`
`0b11`	$\ell = 2$	$\underbrace{\mathtt{11}}_{a=3}\underbrace{\mathtt{0}}_{\text{low bits of }\ell} \quad \mathtt{1}\underbrace{\mathtt{00}}_{h=2}$	`0b110100`
`0b100`	$\ell = 3$	$\underbrace{\mathtt{100}}_{a=4}\underbrace{\mathtt{1}}_{\text{low bits of }\ell} \quad \mathtt{1}\underbrace{\mathtt{00}}_{h=2}$	`0b1001100`
`0b101`	$\ell = 3$	$\underbrace{\mathtt{101}}_{a=5}\underbrace{\mathtt{1}}_{\text{low bits of }\ell} \quad \mathtt{1}\underbrace{\mathtt{00}}_{h=2}$	`0b1011100`
`0b1111`	$\ell = 4$	$\underbrace{\mathtt{1111}}_{a=15}\underbrace{\mathtt{00}}_{\text{low bits of }\ell} \quad \mathtt{1}\underbrace{\mathtt{000}}_{h=3}$	`0b1111001000`
`0x70`	$\ell = 7$	$\underbrace{\mathtt{1110000}}_{a=112}\underbrace{\mathtt{11}}_{\text{low bits of }\ell} \quad \mathtt{1}\underbrace{\mathtt{000}}_{h=3}$	`0b1110000111000`
`0x1234`	$\ell = 13$	$\underbrace{\mathtt{1001000110100}}_{a=4660}\underbrace{\mathtt{101}}_{\text{low bits of }\ell} \quad \mathtt{1}\underbrace{\mathtt{0000}}_{h=4}$	`0b10010001101001011 0000`

With `+mat` encoding in hand, we can produce a proper serialization scheme for nouns by combining the type bit with the encoded atom or recursively encoding cells. We call this scheme `+gel` encoding.

As before, we use `0` for atom and `1` for cell. If the noun is an atom, we follow the type bit with the `+mat` encoding of the atom. If the noun is a cell, we follow the type bit with the `+gel` encoding of the head and then the `+gel` encoding of the tail.

Consider the noun `[4 [0 1]]` (= `[4 0 1]`), represented as a binary tree in Figure 2. Reading across in a depth-first traversal, we produce an encoding order:

```
(mat 1)(0)(mat 0)(0)(1)(mat 4)(0)(1)
```

We already have the `+mat` encodings from Table 3; substituting these in yields:

```
[110](0)[1](0)(1)[1001100](0)(1)
0b110.0.1.0.1.1001100.0.1
0b1100101100110001
```

`+gel` Serialization Algorithm

- **Encode**.
  - If the noun is an atom, write `0` followed by the `+mat` encoding of the atom.
  - If the noun is a cell, write `1` followed by the `+gel` encoding of the head and then the `+gel` encoding of the tail.
- **Decode**. Read the first bit.
  - If it is `0`, read the next bits as a `+mat` encoded atom.
  - If it is `1`, recursively read the next bits as the head and then the tail.

`+gel` encoding is straightforward to read and write even by hand. Worked examples are included in Table 4.

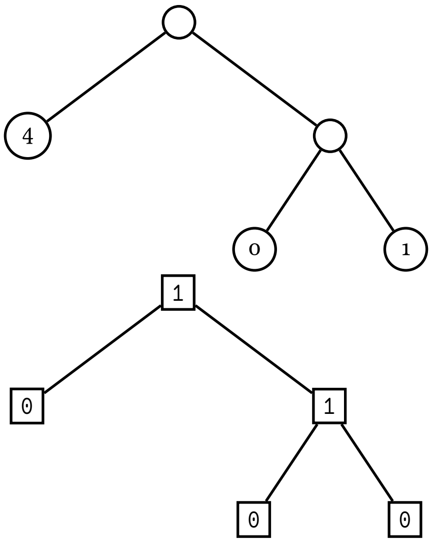

Figure 2: The noun `[4 0 1]` as a binary tree and its metadata labeling in the simplest scheme. Compare Figure 1.

Our code implementation for `+gel` encoding is as follows:

```
++  gel
  |=  a=*
  ^-  @
  =+  b=0
  =<  q
  |-  ^-  [p=@ q=@]
  ?:  ?=(@ a)
    =+  d=(mat a)
    [(add 1 p.d) (lsh 0 q.d)]
  =>  .(b (add 1 b))
  =+  d=$(a -.a)
  =+  e=$(a +.a, b (add b p.d))
  :-  (add 1 (add p.d p.e))
  (mix 1 (lsh [0 1] (cat 0 q.d q.e)))
```

Decoding is left as an exercise to the motivated reader.

Table 4: Example noun encodings (`+gel` encoding). `.` dot delimits bit fields for readability.

Noun	Encoded Form
`0`	`0b0.1.0`
`[0 0]`	`0b1.0.1.0.1`
`[1 0]`	`0b1.0.110.0.1`
`[2 1 0]`	`0b1.0.110.0.1.100100.0.1`
`[[2 3] 1 0]`	`0b1.0.110.0.1.110100.0.100100.0.1.1`

3 Practical Serialization

Whatever the pedagogical advantages of `+gel`, the algorithm is still relatively verbose. Nock-based operating functions have preferred in practice to use `+jam` encoding, which is a more compact run-length encoding with subtree deduplication.[6] `+jam` improves the basic strategy by altering the RLE algorithm slightly and supporting internal references for noun subtrees that have already been encoded.

`+jam` converts a noun into a buffer and deduplicates repeated subtrees. It walks subtrees and encodes each in a way that allows for efficient storage and retrieval, while also permitting references to previously encoded values. Basically, it takes `+gel` and adds a "caching mode" flag; instead of repeating subtrees, encode the index in the full jammed noun to where the current node's jammed subnoun was first written. Thus the code changes from `0` for atom and `1` for cell to:

- `0` for atom.
- `01` for cell.
- `11` for cached node.

Following the metadata labels given in Figure 3, the encoding order for `[[0 0] 0 0]` is:

[6]Note the calculation of `~dozreg-toplud`, p. 57 of this issue, that an operational Arvo instance may have up to 1.66×10^{21} nouns, greatly reduced by structural sharing.

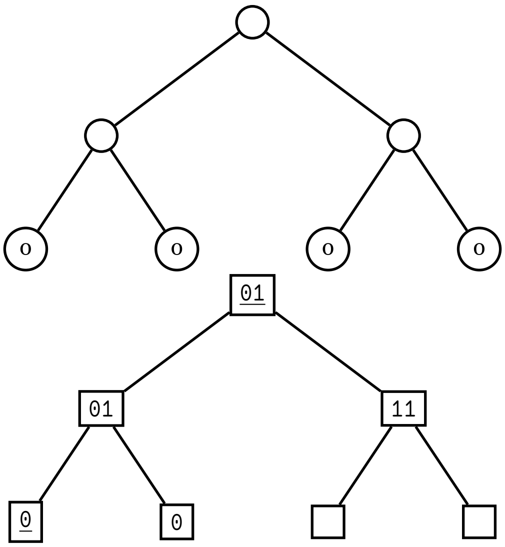

Figure 3: The noun `[[0 0] 0 0]` as a binary tree and its metadata labeling for the `+jam` scheme.

```
(mat 2)(11)(mat 0)(0)(mat 0)(0)(01)(01)
0b100100.11.1.0.1.0.01.01
0b1001001110100101
```

The cached entry `(mat 2)` indicates that the cached subtree is a copy of the subtree located at tree address 2 in the overall jammed noun.

However, this bit of cleverness isn't the end: `+jam` also decides *when* to cache subtrees based on the size of the nouns involved. For instance, `[3 3 3]` encodes as:

```
(mat 3)(0)(mat 3)(0)(01)(mat 3)(0)(01)
0b110100.0.110100.0.01.110100.0.01
0b11010001101000011101000001
```

wherein no caching is used. When 3 is first encountered at index 2, 2 and 3 have the same bit lengths; since `(mat 2)` and `(mat 3)` are the same length, caching would not save any space and would add decoding overhead due to dereferencing.

In contrast, `[4 4 4]` encodes as:

```
(mat 2)(11)(mat 2)(11)(01)(1001100)(0)(01)
0b100100.11.100100.11.01.1001100.0.01
0b10010011100100110110011000001
```

where caching is used for the second and third occurrences of 4. Here, 4 has bit length 3 which is greater than the bit length of 2 which is 2, and caching saves space overall in the jammed noun.

`+jam` Serialization Algorithm

- **Encode**.
  - If the noun is an atom, write `0` followed by the `+mat` encoding of the atom.
  - If the noun is a cell, check if it has been previously encoded.
    * If it has, write `11` followed by the `+mat` encoding of the index where it was first encoded.
    * If it has not, write `01` followed by the `+jam` encoding of the head and then the `+jam` encoding of the tail. Store the current index for future reference.
- **Decode**. Read the first bit.
  - If it is `0`, read the next bits as a `+mat` encoded atom.
  - If it is `1`, read the next bit.
    * If it is `0`, recursively read the next bits as the head and then the tail.
    * If it is `1`, read the next bits as a `+mat` encoded index and retrieve the cached subtree.

Here is the reference implementation of `+jam` in Hoon:

```
++  jam
  ~/  %jam
  |=  a=*
  ^-  @
  =+  b=0
  =+  m=`(map * @)`~
  =<  q
  |-  ^-  [p=@ q=@ r=(map * @)]
  =+  c=(~(get by m) a)
  ?~  c
    =>  .(m (~(put by m) a b))
    ?:  ?=(@ a)
      =+  d=(mat a)
      [(add 1 p.d) (lsh 0 q.d) m]
    =>  .(b (add 2 b))
    =+  d=$(a -.a)
    =+  e=$(a +.a, b (add b p.d), m r.d)
    :+  (add 2 (add p.d p.e))
      (mix 1 (lsh [0 2] (cat 0 q.d q.e)))
    r.e
  ?:  ?&(?=(@ a) (lte (met 0 a) (met 0 u.c)))
    =+  d=(mat a)
    [(add 1 p.d) (lsh 0 q.d) m]
  =+  d=(mat u.c)
  [(add 2 p.d) (mix 3 (lsh [0 2] q.d)) m]
```

A Python implementation of `+jam` is included in the appendix.

Here is the reference implementation of `+cue` in Hoon:

```
++  cue
  ~/  %cue
  |=  a=@
  ^-  *
  =+  b=0
  =+  m=`(map @ *)`~
  =<  q
  |-  ^-  [p=@ q=* r=(map @ *)]
  ?:  =(0 (cut 0 [b 1] a))
    =+  c=(rub +(b) a)
    [+(p.c) q.c (~(put by m) b q.c)]
  =+  c=(add 2 b)
  ?:  =(0 (cut 0 [+(b) 1] a))
    =+  u=$(b c)
    =+  v=$(b (add p.u c), m r.u)
    =+  w=[q.u q.v]
    [(add 2 (add p.u p.v)) w (~(put by r.v) b w)]
  =+  d=(rub c a)
  [(add 2 p.d) (need (~(get by m) q.d)) m]
::  length decode
++  rub
  ~/  %rub
  |=  [a=@ b=@]
  ^-  [p=@ q=@]
  =+  ^=  c
      =+  [c=0 m=(met 0 b)]
      |-  ?<  (gth c m)
      ?.  =(0 (cut 0 [(add a c) 1] b))
        c
      $(c +(c))
  ?:  =(0 c)
    [1 0]
  =+  d=(add a +(c))
  =+  e=(add (bex (dec c)) (cut 0 [d (dec c)] b))
  [(add (add c c) e) (cut 0 [(add d (dec c)) e] b)]
```

Likewise, a Python implementation of `+cue` is included in the appendix.

3.1 `newt` Encoding

"Newt" encoding is a runtime-oriented extension of `+jam`based noun serialization which adds a short identifying header in case of future changes to the runtime's serialization format. A version number (currently a single bit) precedes a RLE serialization length followed by the `+jam` serialization of the noun. The version number is currently `0b0`.

```
V.LLLL.JJJJ.JJJJ.JJJJ.JJJJ.JJJJ.JJJJ
```

where `V` is the version number, `L` is the total length of the noun in bytes, and `J` is the `+jam` serialization of the noun.

Runtime communications vanes like `%khan` and `%lick` utilize this encoding locally. It is exclusively used as a host OS runtime affordance at the current time.

4 Conclusion

Noun serialization is a fundamental aspect of Urbit's architecture, enabling efficient communication of untyped data structures. Starting from basic run-length encoding, we have explored progressively more sophisticated schemes culminating in `+jam`, which balances compactness and efficiency through subtree deduplication. Further extensions like `newt` illustrate the adaptability of serialization strategies to meet evolving runtime requirements. As Nock-based solid-state computing continues to evolve, noun serialization will remain an important foundation for data interchange and persistence.

Appendix: Python `+jam`/`+cue`

Paul Driver `~fodwyt-ragful`

The following is a simple Python implementation of `+jam` serialization and `+cue` deserialization drawn from Urbit's auxiliary `pynoun` library.

```
from bitstring import BitArray
noun = int | Cell
# The Cell class represents an ordered pair of two nouns.
def jam_to_stream(n: noun, out: BitArray):
    """jam but put the bits into a stream
    >>> s = BitArray()
    >>> jam_to_stream(Cell(0,0), s)
    >>> s
    BitArray('0b100101')
    """

    cur = 0
    refs = {}

    def bit(b: bool):
        nonlocal cur
        out.append([b])
        cur += 1

    def zero():
        bit(False)

    def one():
        bit(True)

    def bits(num: int, count: int):
        nonlocal cur
        for i in range(0, count):
            out.append([(num & (1 << i)) != 0])
        cur += count

    def save(a: noun):
        refs[a] = cur
```

```
def mat(i: int):
    if 0 == i:
        one()
    else:
        a = i.bit_length()
        b = a.bit_length()
        above = b + 1
        below = b - 1
        bits(1 << b, above)
        bits(a & ((1 << below) - 1), below)
        bits(i, a)

def back(ref: int):
    one()
    one()
    mat(ref)

def r(a: noun):
    dupe = refs.get(a)
    if deep(a):
        if dupe:
            back(dupe)
        else:
            save(a)
            one()
            zero()
            r(a.head)
            r(a.tail)
    elif dupe:
        isize = a.bit_length()
        dsize = dupe.bit_length()
        if isize < dsize:
            zero()
            mat(a)
        else:
            back(dupe)
    else:
        save(a)
        zero()
        mat(a)
r(n)
```

```
def jam(n: noun):
    """urbit serialization: * -> @

    >>> jam(0)
    2
    >>> jam(Cell(0,0))
    41
    >>> jam(Cell(Cell(1234567890987654321, \\
    ...               1234567890987654321), \\
    ...          Cell(1234567890987654321, \\
    ...               1234567890987654321)))
    2284009509580689287425738957 3
    """

    out = BitArray()
    jam_to_stream(n, out)
    return read_int(len(out), out)

def cue_from_stream(s: BitArray):
    """cue but read the bits from a stream
    >>> s = BitArray('0b01')
    >>> cue_from_stream(s)
    0
    """

    refs = {}
    cur = 0
    position = 0

    def bits(n: int):
        nonlocal cur, position
        cur += n
        result = 0
        for i in range(n):
            result |= (1 if s[position] else 0) << i
            position += 1
        return result

    def one():
        nonlocal cur, position
        cur += 1
        bit = s[position]
```

```
            position += 1
            return bit

        def rub():
            z = 0
            while not one():
                z += 1
            if 0 == z:
                return 0
            below = z - 1
            lbits = bits(below)
            bex = 1 << below
            return bits(bex ^ lbits)

        def r(start: int):
            ret = None
            if one():
                if one():
                    ret = refs[rub()]
                else:
                    hed = r(cur)
                    tal = r(cur)
                    ret = Cell(hed, tal)
            else:
                ret = rub()
            refs[start] = ret
            return ret
        return r(cur)

    def cue(i: int):
        """urbit deserialization: @ -> *

        >>> str(cue(2284009509580689287425738957 3))
        '[[1234567890987654321 1234567890987654321]
          1234567890987654321 1234567890987654321]'
        """

        bits = BitArray()
        while i > 0:
            bits.append([i & 1 == 1])
            i >>= 1
        return cue_from_stream(bits)
```