Representing Nouns as Byte Arrays

N. E. Davis & Sam Parker ~lagrev-nocfep & ~zod
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 two principal conventions for representing nouns in slightly compressed form as byte arrays, as well as introduces a new naïve run-length encoding for educational purposes.

Contents

Nock deals in nouns and does not know about bits or memory. However, a Nock interpreter (runtime) must deal with these practicalities. In other words, we must have a way of writing down an abstract binary tree (consisting of cells/pairs and atoms) as an actual, physical array of bits in memory.

There are three basic ways to encode a noun as an atom:

1. ++jel. A naïve encoding similar to ++jam but without references. Introduced in this article.

Urbit Systems Technical Journal II:1 (2025): 1–46.

Address author correspondence to ~lagrev-nocfep & ~zod.

2. `++jam`. A run-length encoding including references to repeated values. The standard serialization for Urbit/Arvo.

3. `++bulk`/`++silo`. A directed graph encoding with references. The standard serialization for Nockchain.

Each of these

1 `++jel`, the Naïve Encoding

The simplest way to encode a noun as a binary tree in an atom (byte array) without compression is to utilize bits to mark atom (`0`) vs. cell (`1`), the length of the atom in unary of `1`s terminated by `0`, and the actual value. (Since the leading zeros are stripped, they may need to be added back in.) This is in least-significant *byte* (LSB) order, so you should read the written atom starting from the *right*.

Thus the atom `0b0` would simply be encoded as:

```
:: 0x0 = 0 for atom; 1 for length (special case);
::       0 for end of length; 0 for value
> `@ub`(jel 0b0)
0b10
```

interpreted as (rightmost, LSB) `0` for atom, run length of `1`, and value `0` (stripped). I.e., `0b010` or

←

0 1 0

Other values include:

```
:: 0x1 = 0 for atom; 1 for length;
::       0 for end of length; 1 for value
> `@ub`(jel 0b1)
0b1010

:: [0x0 0x1] = 1 for cell; zero, then one
> `@ub`(jel [@0b0@ 0b1])
0b1.010@0.010@1

:: [0x1 0x0] = 1 for cell; one, then zero
> `@ub`(jel [@0b0@ 0b1])
```

```
0b10@1.010@1

:: 0x2 = 0 for atom; 2 for length;
::       0 for end of length; 2 for value
> `@ub`(jel 0b10)
0b10.0111

:: 0xff
> `@ub`(jel 0xff)
0b11.1111.1101.1111.1110

:: [[0x0 0x1] [0x2 0x3]]
> `@ub`(jel [@0b0@ 0b1])
0b10@1.010@1
```

Our code implementation for `++jel` is as follows:

```
!:  |%
++  jel
  =jel !:  |=  a=*
  ^-  @
  =+  l=0
  =+  b=0
  =<  -
  |-
  ?^  a
    =+  lv=$(a -.a)
    =+  rv=$(a +.a)
    =+  [c l]=(mash rv lv)
    [(con (lsh [0 1] c) 0b1) +(l)]
  ?:  =(0 a)  [0b10 4]
  :: need another mash in here for unary length
  =+  [c l]=(mash [a l] [b +((met 0 b))])
  [(con (lsh [0 1] c) 0b0) +(l)]
:: length of atom in unary
++  len
  |=  a=@
  ^-  @
  (fil 0 (met 0 a) 0b1)
:: mash two atoms together
++  mash
  |=  [a=[p=@ l=@] b=[p=@ l=@]]
  ^-  [c=@ l=@]
```

```
  :-  (con (lsh [0 1.b] p.a) p.b)
  (add l.a l.b)
--
```

Note that `l` is not the length of an atom in unary, but the length of the encoded noun in binary.

2 `++jam`, the Practical Serialization

The `++jel` algorithm has two flaws: it is subject to collisions XXX and it is verbose. `++jam` will improve on this by altering the RLE slightly and

special-cases 0

The new RLE calculation is to post the number plus one in binary rather than unary (e.g., for the length would be `0b010`).

(Since there is a value less significant than the length, the leading zero is not lost but serves as a divider.)

```
++  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]
```

2.1 `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 serialization format. A version number (currently as a single bit) precedes a serialization length followed by the `++jam` serialization of the noun. The version number is currently always `0b0`, but this may change in the future.

```
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 Unix affordance at the current time.

3 `+$bulk`, the Directed Graph Encoding