Representing Nouns as Byte Arrays

N. E. Davis `~lagrev-nocfep`
Zorp Corp

Abstract

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.

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

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

Address author correspondence to `~lagrev-nocfep`.

0.1 `++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

←

$\boxed{0}\,\boxed{1}\,\boxed{0}$

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 l.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.