Nock 4K

A noun is an atom or a cell. An atom is a natural number. A cell is an ordered pair of nouns.

Axiomatic Operators. Reduce by first matching pattern; vars match any noun.

```
[a b c]             [a [b c]]                            # Cell syntax

?[a b]              0                                    # True
?a                  1                                    # False
+[a b]              +[a b]                               #
+a                  1+a                                  # Increment
=[a a]              0                                    # Equality
=[a b]              1                                    # Inequality

/[1 a]              a                                    # Slot (self)
/[2 a b]            a                                    # Slot (left/head)
/[3 a b]            b                                    # Slot (right/tail)
/[(a+a) b]          /[2 /[a b]]                          # Slot (any even)
/[(a+a+1) b]        /[3 /[a b]]                          # Slot (any odd)
/a                  /a                                   #

#[1 a b]            a                                    # Edit (self)
#[(a+a) b c]        #[a [b /[(a+a+1) c]] c]              # Edit (left/head)
#[(a+a+1) b c]      #[a [/[(a+a) c] b] c]                # Edit (right/head)
#a                  #a                                   #

*[a [b c] d]        [*[a b c] *[a d]]                    # Cell distribution
```

Opcodes. Analogous combinators are in some cases indicated.

```
*[a 0 b]            /[b a]                               # Slot/axis (I)
*[a 1 b]            b                                    # Constant (K)
*[a 2 b c]          *[*[a b] *[a c]]                     # Evaluate (S)
*[a 3 b]            ?*[a b]                              # Cell test
*[a 4 b]            +*[a b]                              # Increment
*[a 5 b c]          =[*[a b] *[a c]]                     # Equality test

*[a 6 b c d]        *[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]# Branch
*[a 7 b c]          *[*[a b] c]                          # Compose (B)
*[a 8 b c]          *[[*[a b] a] c]                      # Extend
*[a 9 b c]          *[*[a c] 2 [0 1] 0 b]                # Invoke
*[a 10 [b c] d]     #[b *[a c] *[a d]]                   # Edit

*[a 11 [b c] d]     *[[*[a c] *[a d]] 0 3]               # Hint, dynamic
*[a 11 b c]         *[a c]                               # Hint, static

*a                  *a                                   # Operate until done
```

Key Points

- Turing-complete: implements minimum reqs of μ-recursive functions.
- Functional ↦◊: uniquely maps [subject formula] to product.
- Subject-oriented: subject entirely determines scope for formula.
- Homoiconic: uses same representation for data and control.
- Untyped: only knows about nouns.
- Solid-state: maintains no transient state in interpreter.

Topics

- **Basic Nock rules 0-5**
  - Sufficient for Turing completeness
  - Example: Increment

  - Example: Addition

  - Example: Decrement

- **Extended Nock rules 6-11**
  - Convenient for compilation and interpretation
  - Example:

  - Example: Addition (Redux)

- **Interpreter**
  - Virtualization
  - Tree-walking interpreter
  - Bytecode
- **Nock ISA Languages**
  - Hoon: mature kernel language, terse "runes" (developed by ~Urbit)
  - Jock: alpha scripting language (developed by ZORP/NOCKCHAIN)

```
::  Hoon 140K
=>
  ^=  point
  |_  [x=@ y=@]
  ++  adder
    |=  p=[x=@ y=@]
    (add x x.p)
  ++  inc
    |=  q=@
    +(q)
  --
=/  vector
  ~(. point [12 23])
(adder:vector [30 19])
```

```
// Jock alpha
compose
  class Point(x:Atom y:Atom) {
    adder(p:(x:Atom y:Atom)) -> Point {
      (x + p.x
       y + p.y)
    }
    inc(q:Atom) -> Atom {
      +(q)
    }
  };

let vector = Point(12 23);
vector.adder(30 19)
```

```
[7 [8 [1 0 0] [1 [8 [1 0 0] [1 8 [7 [0 31] 9 348 0 1] 9 2 10 [6 [0 62] 0 28] 0 2] 0
1] 8 [1 0] [1 4 0 6] 0 1] 0 1] 8 [8 [0 1] 10 [6 7 [0 3] 1 12 23] 0 2] 8 [7 [0 2] 9 4
0 1] 9 2 10 [6 7 [0 3] 1 30 19] 0 2]
```