Nocksche and Nocko

[[[0 1] [[0 2] [0 3]]] [[0 1] [[0 2] [0 3]]]]

James P. Torre, IV

jpt4@proton.me

Nock (4K)

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

Reduce by the first matching pattern; variables match any noun.

```
nock(a)             *a
[a b c]             [a [b c]]

?[a b]              0
?a                  1
+[a b]              +[a b]
+a                  1 + a
=[a a]              0
=[a b]              1

/[1 a]              a
/[2 a b]            a
/[3 a b]            b
/[(a + a) b]        /[2 /[a b]]
/[(a + a + 1) b]    /[3 /[a b]]
/a                  /a

#[1 a b]            a
#[(a + a) b c]      #[a [b /[(a + a + 1) c]] c]
#[(a + a + 1) b c]  #[a [/[(a + a) c] b] c]
#a                  #a
```

```
*[a [b c] d]        [*[a b c] *[a d]]

*[a 0 b]            /[b a]
*[a 1 b]            b
*[a 2 b c]          *[*[a b] *[a c]]
*[a 3 b]            ?*[a b]
*[a 4 b]            +*[a b]
*[a 5 b c]          =[*[a b] *[a c]]

*[a 6 b c d]        *[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]
*[a 7 b c]          *[*[a b] c]
*[a 8 b c]          *[[*[a b] a] c]
*[a 9 b c]          *[*[a c] 2 [0 1] 0 b]
*[a 10 [b c] d]     #[b *[a c] *[a d]]

*[a 11 [b c] d]     *[[*[a c] *[a d]] 0 3]
*[a 11 b c]         *[a c]

*a                  *a
```

Use cases

Urbit, Plunder	Network-centric operating system
Zorp	Zero-Knowledge virtual machine, Subject Knowledge Analysis
Simplicity	Functional language for blockchain applications
Awelon Blue	Reactive Demand Programming
knock	K Framework implementation

Claim: Capable of supporting rigorous, practical, functional systems programming.

Nock (4K)

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

Reduce by the first matching pattern; variables match any noun.

nock(a)	*a
[a b c]	[a [b c]]
?[a b]	0
?a	1
+[a b]	+[a b]
+a	1 + a
=[a a]	0
=[a b]	1
/[1 a]	a
/[2 a b]	a
/[3 a b]	b
/[(a + a) b]	/[2 /[a b]]
/[(a + a + 1) b]	/[3 /[a b]]
/a	/a
#[1 a b]	a
#[(a + a) b c]	#[a [b /[(a + a + 1) c]] c]
#[(a + a + 1) b c]	#[a [/[(a + a) c] b] c]
#a	#a

*[a [b c] d]	[*[a b c] *[a d]]
*[a 0 b]	/[b a]
*[a 1 b]	b
*[a 2 b c]	*[*[a b] *[a c]]
*[a 3 b]	?*[a b]
*[a 4 b]	+*[a b]
*[a 5 b c]	=[*[a b] *[a c]]
*[a 6 b c d]	*[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]
*[a 7 b c]	*[*[a b] c]
*[a 8 b c]	*[[*[a b] a] c]
*[a 9 b c]	*[*[a c] 2 [0 1] 0 b]
*[a 10 [b c] d]	#[b *[a c] *[a d]]
*[a 11 [b c] d]	*[[*[a c] *[a d]] 0 3]
*[a 11 b c]	*[a c]
*a	*a

Valid NEXPs

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

Reduce by the first matching pattern; variables match any noun.

```
nocksche: [[[0 1] [[0 2] [0 3]]] [[0 1] [[0 2] [0 3]]]]

nocko: '[[[(nat 0) (nat 1)] [[(nat 0) (nat 0 1)] [(nat 0) (nat 1 1)]]]
         [[(nat 0) (nat 1)] [[(nat 0) (nat 0 1)] [(nat 0) (nat 1 1)]]]]
```

neval vs nock

[a [b c] d]   [[a b c] *[a d]]

```
(define (neval n)
  (match n
   ...
    [ `(tar [,a [[,b ,c] ,d]])      `[,(neval `(tar [,a [,b ,c]])) ,(neval `(tar [,a ,d]))] ]
   ...)

(define (tar a)
  (match a
    [ `[,a [[,b ,c] ,d]]              `[,(tar `[,a [,b ,c]]) ,(tar `[,a ,d])] ]
   ...)

(define (taro i o)
  (fresh (a b c d resa resb resc)
        (conde
         [ (== `[,a [[,b ,c] ,d]] i)
           (taro `[,a [,b ,c]] resa)
           (taro `[,a ,d] resb)                              (== `[,resa ,resb] o) ]
    ...)
```

Strictness

```
*[a 11 [b c] d]     *[[*[a c] *[a d]] 0 3]
*[a 11 b c]          *[a c]

*a                  *a
```

Right Associativity

[a b c]   [a [b c]]

Shallow or deep?

Early or late?

Obligatory?

Right Associativity

[a b c]   [a [b c]]

```
(define (ras a)
  (match a
   [(? noun) a]
   [`(,x ,y . ,z)
    #:when (not (null? z))
    `[,(ras (car a)) ,(ras (cdr a))]
    ]
   [`(,x ,y . ,z)
    #:when (null? z)
    `[,(ras (car a)) ,(ras (cadr a))]
    ]
   [_ 'error-not-a-noun]
   ))
```

Right Associativity

```
(define (raso i o)
  (fresh (a b c d e resa resb resc resd)
         (conde
          [ (atomo i) (== i o) ]
          [ (== `[,a ,b . ,c] i)
            (== `(,d . ,e) c) (=/= '() e)
           (=/= 'nat d)
            (raso a resa) (raso b resb) (raso c resc)
            (== `[,resa [,resb ,resc]] o)
            ]
          [ (== `[,a ,b . ,c] i)
            (== `(,d . ()) c)
           (raso a resa) (raso b resb) (raso d resd)
            (== `[,resa [,resb ,resd]] o)
            ]
          [ (== `[,a ,b . ,c] i)
            (== '() c)
            (raso a resa) (raso b resb)
            (== `[,resa ,resb] o) ]
          )))
```

“Reverse Tree Forth”

Ix = x		
Kxy = x	*[a 1 b]	b
Sxyz = xz(yz)	*[a 2 b c]	*[*[a b] *[a c]]
Zgv = g(Zg)v	*[a 8 b c]	*[[*[a b] a] c]

“Reverse Tree Forth”

```
(define Is '[0 1])

(define Kc '[8 [[1 1] [0 1]]])

(define Sc '[[1 [1 2]] [[1 [0 1]] [[1 1] [0 1]]]])
```

```
Ix    x=12    (nock `[12 ,Is]) => 12

Kxy    x=7 y=6    (nock `[6 ,(tar `[7 ,Kc])]) = > 7

SKxy    x=7 y=6    (nock `[6 ,(tar `[7 ,(tar `[,Kc ,Sc])])]) => 6
```

Unlambda: ```sxyz → ``xz`yz

quine and quineo

*a   *a

nocksche: [[[0 1] [[0 2] [0 3]]] [[0 1] [[0 2] [0 3]]]]

nocko: '[[[(nat 0) (nat 1)] [[(nat 0) (nat 0 1)] [(nat 0) (nat 1 1)]]]
[[(nat 0) (nat 1)] [[(nat 0) (nat 0 1)] [(nat 0) (nat 1 1)]]]]

Future Work

More combinators – e.g. B, C, K, W

Reduce Nock 6-11 to 0-5

clojure.core.logic.fd – explore improved reverse synthesis

Nock-in-Nock

Additional formal verification – e.g. https://github.com/nvasilakis/Noq

References

[0] https://docs.urbit.org/language/nock

[1] https://sr.ht/~plan/plunder/

[2] https://zorp.io/

[3] https://www.youtube.com/watch?v=8vtnmiEN-r4

[4] https://blockstream.com/simplicity.pdf

[5] https://github.com/dmbarbour/wikilon/blob/master/docs/AwelonLang.md

[6] https://github.com/runtimeverification/knock/tree/master

[7] https://esolangs.org/wiki/Unlambda

[8] https://en.wikipedia.org/wiki/Fixed-point_combinator#Strict_fixed-point_combinator

Appendix – Quine Derivation

```
*[a [b c] d]
[*[a b c] *[a d]]
[b c]=[0 1]
[*[a 0 1] *[a d]]
[/[1 a] *[a d]]
[a *[a d]]
d=[[e f] g]
[a *[a [[e f] g]]]
[a [*[a e f] *[a g]]]
[e f]=[0 bcindex]
[a [*[a [0 bcindex]] *[a g]]]
[a [/[bcindex a] *[a g]]]
[bcindex a]=[2 [[b c] h]]
bcindex=2
a=[[b c] h]
```

```
[a [/[2 [[b c] h]] *[[[b c] h] g]]]
[a [[b c] *[[[b c] h] g]]]
g=[0 dindex]
[a [[b c] *[[[b c] h] [0 dindex]]]]
[a [[b c] /[dindex [[b c] h]]]]
dindex=3
[a [[b c] /[3 [[b c] h]]]]
[a [[b c] h]]
h=d=[[e f] g]=[[0 bcindex] g]=[[0 2] g]
  =[[0 2] [0 dindex]]=[[0 2] [0 3]]
h=d=[[0 2] [0 3]]
[b c]=[0 1]
a=[[b c] h]=[[0 1] h]=[[0 1] [[0 2] [0 3]]]
a=[[0 1] [[0 2] [0 3]]]

[a [b c] d]=
[[[0 1] [[0 2] [0 3]]] [0 1] [[0 2] [0 3]]]
```

Overview of Nock
use and goals
examination of informal spec

Nocksche
monolithic term rewriting vs modular functions
strict vs non-strict
external versus internal syntax
right associative normalization

Nocko
partitioning of pattern space

Combinators, Quine

Future work