Hoon Cheat Sheets

Table of Contents

Arms

+ | lus

+|		Label a chapter (produces no arm)
+$	[term spec]	Produces a structure arm (type definition)
++	[term hoon]	Produces a (normal) arm
+*	[term term spec]	Define a deferred expression (within a door)

Cores

| | bar

|$	(lest term) spec	Produces a mold
|_	spec alas (map term tome)	Produces a door (a core with a sample)
|:	[hoon hoon]	Produces a gate with a custom sample
|%	(unit term) (map term tome)	Produces a core (battery and payload)
|.	hoon	Produces a trap (a core with one arm)
|^	hoon (map term tome)	Produces a core with a $ arm and computes the latter
|-	hoon	Produces a trap and evaluate it
|~	hoon [spec value]	Produces an iron gate
|*	hoon [spec value]	Produces a wet gate (a one-armed core with sample)
|=	hoon [spec value]	Produces a dry gate (a one-armed core with sample)
|@	hoon (unit term) (map term to me)	Produces a wet core (battery and payload)
|?	hoon	Produces a lead trap

Structures

$ | buc

$|	[spec hoon]	foo	Structure with verification
$_	hoon		Structure that normalizes to an example
$%	[list spec]		Structure that recognizes a union tagged by head atom
$:	[list spec]	[a=foo b=bar c=baz]	Form a cell type (tuple)
$<	[spec spec]		Structure from filter (excluding)
$>	[spec spec]		Structure from filter (requiring)
$-	[spec spec]		Structure that normalizes to an example gate
$^	hoon		Structure that normalizes a union tagged by head depth
$&	[spec hoon]		Repaired structure (using normalizing gate)
$~	[hoon spec]		Define a custom type default value
$@	[spec spec]		Structure that normalizes a union tagged by head atom
$=	[skin spec]		Structure that wraps a face around another structure
$?	[list spec]		Form a type from a union of other types

Calls

% | cen

%_	[wing (list (pair wing hoon))]		Resolve a wing with changes, preserving type
%:	[hoon (list hoon)]		Call a gate with many arguments
%.	[hoon hoon]		Call a gate, inverted
%-	[hoon hoon]	(gat smp)	Call a gate
%^	[hoon hoon hoon hoon]		Call a gate with triple sample
%+	[hoon hoon hoon]		Call a gate with a cell sample
%~	[wing hoon hoon]		Evaluate an arm in a door
%*	[wing hoon (list (pair wing hoon))]		Evaluate an expression, then resolve a wing with changes
%=	[wing (list (pair wing hoon))]	foo(bar 1, baz 2)	Resolve wing with changes

Cells

: | col

:_	[hoon hoon]		Construct a cell, inverted
:-	[hoon hoon]	[foo bar]. foo^bar	Construct a cell, 2-tuple
:^	[hoon hoon hoon hoon]	[foo bar baz quz]	Construct a cell, 2-tuple
:+	[hoon hoon hoon]		Construct a cell, 3-tuple
:~	(list hoon)	~[foo bar baz]	Construct a null-terminated list
:*	(list hoon)	[foo bar baz …]	Construct an n-tuple
::			Mark a comment (digraph, not rune)

Nock

. | dot

.^	[spec hoon]		Load from namespace using Nock 12 (scry or peek)
.+	atom	+foo	Increment an atom using Nock 4
.*	[hoon hoon]		Evaluate using Nock 2
.=	[hoon hoon]	=(foo bar)	Test for equality using Nock 5
.?	(Hoon)		Test for a cell or atom using Nock 5

Imports `(++ford arm of %clay)` / | fas

`/$`	`%from %to`	Import mark conversion gate from `/mar`
`/%`	`%mark`	import mark definition from `/mar`
`/-`	`foo, *bar, baz=qux`	import a file from `/sur` (`*` no face, = specified face)
`/+`	`foo, *bar, baz=qux`	import a file from `/lib` (`*` no face, = specified face)
`/=`	`clay-raw  /sys/vane/clay`	import results of user-specified path with face
`/*`	`myfile  %hoon  /gen/myfile/hoon`	Import the contents of a file in the desk converted to a mark (build-time static data)
`/~`	`face  type  /path`	Import contents of dir as `face=(map @ta type)`
`/?`		Pin version number (not enforced)

Casts ^ | ket

`^|`	hoon		Convert a gold core to an iron core (invariant)
`^:`	spec		Produce a 'factory' gate for a type (switch from regular parsing to spec/type parsing)
`^.`	[hoon hoon]		Typecast on value
`^-`	`[spec hoon]`	`` `foo`bar ``	Typecast by explicit type label
`^+`	[hoon hoon]		Typecast by inferred type (a fence)
`^&`	hoon		Convert a core to a zinc core (covariant)
`^~`	hoon		Fold constant at compile time
`^*`	spec	`*foo`	Bunt, produces default mold value
`^=`	`[skin hoon]`		Bind name to a value
`^?`	hoon		Convert a core to a lead core (bivariant)

Macros ; | mic

`;:`	`[hoon (list hoon)]`	`:(gat foo bar baz)`	Call a binary function as an *n*-ary function
`;/`	hoon		(Sail) yield tape as XML element
`;<`	`[spec hoon hoon hoon]`		Glue a pipeline together (monadic bind)
`;+`			(Sail) make a single XML node
`;;`	`[spec hoon]`		Normalize a mold, asserting fixpoint
`;~`	`[hoon (list hoon)]`		Glue a pipeline together with a product-sample adapter (monadic bind)
`:*`			(Sail) make a list of XML nodes from Hoon expression
`;=`	`marl:hoot`		(Sail) make a list of XML nodes

Hints ~ | sig

~|	[hoon hoon]	Print in stack trace if failure
~$	[term hoon]	Profiler hit counter
~_	[hoon hoon]	Print in stack trace, user-formatted
~%	[chum hoon tyre hoon]	Register jet
~/	[chum hoon]	Register jet with registered context
~<	$@(term [term hoon]) hoon]	Raw hint, applied to product (“backward”)
~>	[$@(term [term hoon]) hoon)	Raw hint, applied to computation (“forward”)
~+	[@ hoon]	Cache computation
~&	[@ hoon]	Cache computation
~&	[@ud hoon hoon]	Print (used for debugging)
~=	[hoon hoon]	Detect duplicate
~?	[@ud hoon hoon hoon]	Print conditionally (used for debugging)
~!	[hoon hoon]	Print type if compilation future

Subject = | tis

=|	[spec hoon]	Combine default type value with the subject
=:	[(list (pair wing hoon)) hoon]	Change multiple legs in the subject
=,	[hoon hoon]	Exposes namespace (defines a bridge)
=.	[wing hoon hoon]	Change one leg in the subject
=/	[skin hoon hoon]	Combine a named noun with the subject
=<	[hoon hoon]	Compose two expressions, inverted
=>	[hoon hoon]	Compose two expressions
=-	[hoon hoon]	Combine a new noun with the subject
=^	[skin wing hoon hoon]	Pin the head of a pair; changes a leg with the tail
=+	[hoon hoon]	Combine a new noun with the subject
=;	[skin hoon hoon]	Combine a named noun with the subject, inverted
=~	(List hoon)	Compose many expressions
=*	[(pair term (unit spec)) hoon hoon]	Define an alias
=?	[wing hoon hoon hoon]	Change one leg in the subject conditionally

Conditionals

? | wut

`?|`	`(list hoon)`	`|(foo bar baz …)`	Logical OR (loobean)
`?:`	`[hoon hoon hoon]`		Branch on a boolean test
`?.`	`[hoon hoon hoon]`		Branch on a boolean test, inverted
`?<`	`[hoon hoon]`		Assert false
`?>`	`[hoon hoon]`		Assert true
`?-`	`[wing` `(list (pair spec hoon))]`		Switch against type union, no default
`?^`	`[wing hoon hoon]`		Branch on whether a swing of the subject is a cell
`?+`	`[wing hoon` `(list (pair spec hoon))]`		Switch against a union, with default
`?&`	`(list hoon)`	`&(foo bar baz …)`	Logical AND (loobean)
`?@`	`[wing hoon hoon]`		Branch on whether a wing of the subject is an atom
`?~`	`[wing hoon hoon]`		Branch on whether a wing of the subect is null
`?=`	`[spec wing]`		Test pattern match
`?!`	`hoon`		logical NOT (loobean)

Terminators

`==`	Terminate running series of expressions (digraph, not rune)
`--`	Terminate core expression

Wild

! | zap

`!:`	`hoon`	Turn on stack trace
`!,`	`[*hoon hoon]`	Emit AST of expression, `!,(*hoon expression)`
`!.`	`hoon`	Turn off stack trace
`!<`	`hoon`	Lift dynamic value into static context
`!>`	`hoon`	Wrap a noun in its type
`!;`	`[hoon hoon]`	Emit the type for an expression using the type of type
`!@`	`[(list wing) hoon hoon]`	Evaluate conditional on existence of wing
`!=`	`hoon`	Make the Nock formula for a Hoon expression
`!?`	`[$@(@ {@ @}) hoon]`	Restrict Hoon Kelvin version
`!!`	`~`	Crash

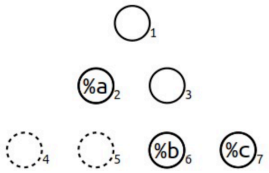
Nock 4K

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
[abc]                       [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 c]          /[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]                          Slot operator (tree address)
*[a 1 b]                    b                               Constant
*[a 2 b c]                  *[*[a b] *[a c]]                Evaluate
*[a 3 b]                    ?*[a b]                         Test for atom
*[a 4 b]                    +*[a b]                         Increment
*[a 5 b c]                  =[*[a b] *[a c]]                Distribution

*[a 6 b c d]                *[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]   If-then-else
*[a 7 b c]                  *[*[a b] c]                     Compose
*[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 noun

*[a11 [b c] d]              *[[*[a c] *[a d]] 0 3]          Hint
*[a11 b c]                  *[a c]

*a                          *a                              Interpret
```

Syntax

`+1:[%a [%b %c]]`	`[%a [%b %c]]`	`[%a [%b %c]]`
`+3:[%a [%b %c]]`	`%a`	
`+4:[%a [%b %c]]`	`[%b %c]`	
`+5:[%a [%b %c]]`	`Invalid`	
`+6:[%a [%b %c]]`	`%b`	
`+7:[%a [%b %c]]`	`%c`	



`.:[%a [%b %c]]`	`[%a [%b %c]]`
`-:[%a [%b %c]]`	`%a`
`+:[%a [%b %c]]`	`[%b %c]`
`-<:[%a [%b %c]]`	`invalid`
`+<:[%a [%b %c]]`	`%b`
`+>:[%a [%b %c]]`	`%c`

`&n`	*n*th element
`|n`	tail after *n*th element
`<[1 2 3]>`	Render list as a tape
`>[1 2 3]<`	Renders list as a tank

Lark syntax equivalents

`+1`		`+5`	`->`
`+2`	`-`	`+6`	`+<`
`+3`	`+`	`+7`	`+>`
`+4`	`-<`	`+8`	`-< -`

`.`	Current subject
`+`	`+:.`
`-`	`-:.`
`+>`	`+>:.`
`a.b.c`	Limb search path

`^face`	face in outer core (`^^face`)
`..arm`	`Core in which ++arm is defined`
`, ,.`	`Strip the face`
`-:!>`	Type spear, use as `-:!>(.3.14)`

	`~`	0 (nil)	
`%.y`	`&`	yes/true/0	
`%.n`	`|`	no/false/1	
`+>`	`%a`	constant	
	`$`	empty term (`@tas`)	

`eny`	entropy
`now`	current time
`our`	ship

`` `a ``	`[~ a]`
`~[a b c]`	`[a b c ~]`
`[a b c]~`	`[[a b c] ~]`
`A/b`	`[%a b]`

`'urbit'`	Cord, atom @t
`“urbit”`	Tape or list of characters
`=wire`	Shadow type name (in defn)
`/path`	Path name
`%`	Current path

Elementary molds

`*`	noun
`2`	atom
`^`	cell
`?`	loobean
`~`	null

Aura Notation

Each aura has a characteristic pattern allowing unique identification in its representation.
Typically this is indicated by a combination of ~, ., and -.

`@`	Empty aura	
`@c`	Unicode dependent	~-~45fed.
`@d`	Date	
`@da`	Date, absolute	`~2020.12.25..7.15.0..1ef5`
`@dr`	Date, relative	`~d71.h19.m26.s24..9d55`
`@i`	Internet address	
`@if`	IPv4 address	
`@is`	IPv6 address	`.0.0.0.0.0.1c.c3c6.8f5a`
`@p`	Phonemic base	`~laszod-dozser-fosrum-fanbyr`
`@q`	Phonemic base, unscrambled (used with Urbit HD wallet)	`.~laszod-dozser-dalteb-hilsyn`
`@r`	IEEE-754 floating-point number	
`@rh`	Floating-point number, half-precision, 16-bit	`.~~3.14`
`@rs`	Floating-point number, single-precision, 32	`.3.141592653589793`
`@rd`	Floating-point number, double-precision, 64-bit	`.~3.141592653589793`
`@rq`	Floating-point number, quadruple-precision, 128-bit	`.~~~3.141592653589793`
`@s`	Integer, signed (sign bit low)	
`@sb`	Signed binary	`--0b10.0000`
`@sd`	Signed decimal	`--1.000`
`@sv`	Signed base-32 `0123456789abcdefghijklmnopqrstuv`	`--0v201.4gvml.245kc`
`@sw`	Signed base-64 `0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`	`--0w2.04AfS.G8xqc`
`@sx`	Signed hexadecimal `0123456789abcdef`	`--0x2004.90fd`
`@t`	UTF-8 text (cord)	`'urbit'`
`@ta`	ASCII text (knot)	`~.urbit`
`@tas`	ASCII text symbol (term)	`%urbit`
`@u`	Integer, unsigned	
`@ub`	Unsigned binary	`0b10.1011`
`@uc`	Bitcoin address `123456789abcdefghijklmnopqrstuvwxyzABCDEFGHJKLMNOPQRSTUVWXYZ`	`0c1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa`
`@ud`	Unsigned decimal	`8.675.309`
`@ui`	Unsigned decimal	`0i123456789`
`@uv`	Unsigned base-32 `0123456789abcdefghijklmnopqrstuv`	`0v88nvd`
`@uw`	Unsigned base-64 `0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`	`0wx5~J`
`@ux`	Unsigned hexadecimal `0123456789abcdef`	`0x84.5fed`

Each aura has a characteristic pattern allowing unique identification in its representation.

`@td`	8-bit ASCII text
`@rhE`	half-precision (16-bit) floating-point number
`@uxG`	unsigned 64-bit hexadecimal
`@uvJ`	unsigned 512-bit integer (frequently used for entropy)