Work Your Core

bar runes: core strength *build your core*

NOCK 9		++ multi-arms	$ one arm	%= auto-fired	+6 besampled	wet	trans muted	
corp	|%	++						*produce a generic dry core. any number of arms in battery. all other dry bar hoons are macros of |%.*
trap	|.		$					*produce a corp with one arm named $ (eliminate the need for writing ++ $)*
loop	|-		$	%=				*produce a trap and then automatically "kick" it, i.e. compute its $ arm with %= (eliminates =< $)*
cork	|^	++		%=				*produce a corp (of any number of arms) with an automatically-"popped"* **$** *arm (so like a loop, but with additional non-auto-fired arms)*
gate	|=		$		+6			*produce a trap with a "sample" (eliminates need for three nested cores with =+s)*
gasp	|:		$		+6			*produce a dry gate with a custom sample* `|:([a=1 b=1] (mul a b))` *will be good; vs.* `|=([a=@ b=@] (mul a b))` *which will bunt to 0*
door	|_	++			+6			*produce a corp (of any number of arms) with a sample (so like a gate, but bigger – multiple arms)*
port	|~		$		+6		●	*produce an iron (contravariant – less specifically specced (superset) input is ok) gate*
mine	|?		$				●	*produce a trap (|.) that is cast (with ^?) to lead (bivariant – both superset and subset specs are ok)*
lava	|@	++				●		*produce a generic wet core. any number of arms in battery. all other wet bar hoons are macros of |@.*
gill	|*		$		+6	●		*produce a wet gate (a one-armed core with sample)*
ptri	|$		$		+6	●		*produce a gill specifically for producing a mold, whose body is parsed in pattern/spec mode.*
p'hep		--						*end a multi-arm core*

cen runes: just do it *fire your core*

%= == *pull a wing, with changes. the most important rune! everything that in any way references a "wing" (a portion of your tree mansion) turns into a* `%=`. *the first child you give it is the main wing: the location of the leg you wish to change or the arm you wish to change and then fire. this could be a face (name) or a tree address (e.g* `+>.$`*). next you give it a* `rig` *aka a* `tray`*: a list of wings that are located within that main wing (e.g. variables you wish to change) and changes to make.*

irregular: `wing(subwing1 hoon, subwing2 hoon)`

%_ == *resolve a wing with changes, then make sure it still fits the inferred spec the wing originally had by fencing it in with ^+.* `%=` *can change the type of a subwing, but* `%_` *will fail if you try to change type. (end tall: ==)*

%* == *evaluate a hoon expression, then using whatever noun that hoon expression produces, resolve a wing within that noun with changes.* `%*($ add a 2, b 3)` *(works b/c standard add gate has an a and a b defined)*

%- *fire a core. irregular for* `%:` *and thus whole family but* `%.` *is* `(gate arg(s))`
generally thought of as the rune for slamming a gate, but also used to fire any non-auto-fired arm.
`(|.(7)) (sqt 49) (add 5 2) (into num-descs 7 %luckiest) (qual @ ^ * _7)`

%. *reverse order* `%.(8 dec)` **%+** *w/ 2 (cell sample)* **%^** *w/ 3* **%:** == *w/ any # (generic version of %-)*

%~ *fire an arm in a door* `(~(arm door door-sample) arm-sample)` *sample placement mnemonic: a gate is outside, the door within it.*

lus runes: arm yourself *stock your battery with arms*

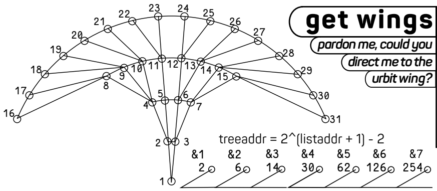
++ slus *begins a normal arm*

+$ jib *begins a structure-building arm. the arm will contain* `spec` *code, which means the trivial molds:* `* @ ^ ? ~` *, named molds (e.g.* `tape`, `list`, `tank`*), or buc rune expressions.*

+* pseu *make alas (alias) "pseudo-arm". common use as a pseudo-arm to give a face to a faceless core:* `+*  this  .`

+| chapter *begins a chapter (a group of arms). just for organization.*

-- p'hep *terminate your list of arms (and thus your core) in a multi-arm core.*



get wings

pardon me, could you direct me to the urbit wing?

ways to give directions to a wing. as an example, the subject of a simple gate: `++ door |=(sample=@ output)` `[battery sample context]`

term	*face or arm name*	*just state face/name. duplicate terms are no problem: prefix w/ ^s to skip the nearest instances.*	`$`	`sample`	`door` *($ and door are arm names. sample is a face.)*
lark	*(left-right)*	`.` *is whole current subject. – or < go left, + or > go right, alternating for legibility. start w/ – or +.*	`-`	`+<`	`+>`
tree	*address (number)*	*+ and then the numerical address. 1is whole current subject. splits: 2, 3. then: 4, 5, 6, 7. etc.*	`+2`	`+6`	`+7`
list	*address (smaller #)*	*treat the subject as a list.* `&n` *gives the nth item.* `|n` *gives the tail after the nth item.*	`&1`	`&2`	`|2` *(will only work if context is a cell)*
path	`sublimb.limb`	*combine any of the 4 syntaxes above into a path traversed right to left, w/ nodes separated by dots.*	`=>(..add +120.&1) → 8` *(try it in the dojo; change addresses)*		
=<	`sublimb:limb`	*you can then compose the above with* `=<`*. may look similar to a path, but is very different!*	`+120.&1:..add = +>+<-<.-:+1.add`		
,		*obtain faceless data. get just the data, not the face. often used with lists to obtain items w/o* `i` *and* `t` *faces.*	`+4 → our=sampel-palnet`	`,.+4 → ~sampel-palnet`	

Catch Your Typos *you gotta check yoself before you wreck yoself*

buc runes: shape sifters

hoon's type system defines sets. the point of it is for you, the programmer, to declare your intentions of what you're trying to do and specifically what kind of data you're trying to generate and pass around. this helps you because then if you make some mistake or typo in your program, that will many times alter the type of the data and the type system will catch that and fail to compile.

these buc runes are the mold builders. what is a mold, you ask? a mold is just a specific kind of gate, and a very simple one. it takes in as a sample any noun and as long as that noun is within its `spec`*, it outputs that same noun (except for $_ and $-). passing spec generally means the noun is the right shape of tree topologically, though sometimes specific atoms can be required at specific nodes. think of a shape-sorting toy with holes of different shapes and blocks to try to pass through them – a mold is like that. a few basic molds have been shortened to a single character:*

***** *noun* **^** *cell* **@** *atom* **?** *flag (an atom that's 0 or 1)* **~** *null (atom that's 0). also,* **!** *represents not-a-noun*

$_ `_anyhoon` — `_%spam` *totally discards the sample it's given and just produces the noun produced by the $_'s child hoon no matter what.*

$= `skin=spec` — `my-face=@` *gives a face to a spec.*

$: == `[spec(s)]` — `[a=@ b=* c=^ d=%27]` *create a cell mold (to pass through, the noun must be a cell). the required shape of the cell further defined by specs within the $:.*

$? == `?(spec(s))` — `?(%good %evil)` *the noun may fit any of the specs in the list (an "or" union). use only for a union of atoms. more complex nouns, use $@, $^, and/or $%.*

$@ — `$@(@ ^)` *a pass-through, allowing all atoms and cells through the mold.* `$@(null spec)` *allows only a particular kind of atom (a null) and cell (a spec).*

$^ — `$^([^ *] [@ *])` *like $@ except selects on whether the head of the noun is a cell or atom, rather than the whole noun. also the order is reversed: ^ @.*

$% == — `$%([%number @] [%fizz tape] [%buzz tape])` *a whole list of cell molds, with the head a constant (cold atom), to be selected based on the head.*

$> — `$>(? [* *]) →` *the head must be a flag (0 or 1) require the head of a cell mold to fit a mold. e.g. could select one mold from a $% list. using above example:* `$>(%number bccn-exmpl) → [%number @]`

$< — [head-filtering-mold cell-mold] *require the head of a cell mold to* ***not*** *fit a mold. e.g. filter a $% list, excluding one mold.* `$<(%number bccn-exmpl) → $%([%fizz tape] [%buzz tape])`

$| — *takes a spec and further restricts it.* `$|(@ (curr lth 10))` *creates a mold for atoms less than 10.*

$& — [spec hoon] *upgrades a structure from an old version to a new one.* `$&(a-$%-mold-combining-old-and-new-version gate-normalizing-to-new)`

$~ — *take any spec and define a custom default value for it.* `$~(%88 $=(face @))` *is* `face=@` *but with a default of 88*

$- — *input a noun fitting one mold to get the bunt of another as output.* `$-(@ ^)` *takes an atom, outputs a cell (it will just bunt).* `$-([@ @] ^)` *will take a cell of two atoms and output a cell bunt.* `$-(@ %ok)` *returns %ok when fed an atom*

ket runes: cast away *generate generalities*

when you cast, you are always throwing away type info, never adding. so you can only cast from more specific to more general. the most specific info is just exactly what the particular noun is, which the compiler already knows, obviously; so this should make sense. so the terminology "cast ***to*** *x" means "cast away all info* ***except*** *x".*

^- `` `spec`hoon `` *cast to a spec by simply explicitly writing the spec to which you wish to cast the noun.*

^= `skin=hoon` *assign a face to a noun.*

^+ `fence` *cast to "whatever this is." cast by writing a noun and letting the system infer its spec.*

^* `*spec` `*mul → 1  *mul:rs → 0` *(a poorly-chosen bunt) bunt – i.e. produce the default value of the spec you explicitly write.*

^. [gate hoon] *cast based on the noun produced by passing q to p – the hoon (any expression) to the gate (e.g.* `limo` *which makes it a list).*

^~ *fold constant* **^&** *transmute to zinc* **^|** *transmute to iron* **^?** *transmute to lead*

Nock

- IS INITIAL SUBJECT, A NOUN (A TREE OF NUMBERS)

P, Q, AND R ARE NOUNS

⟦ ⟧ IS A CELL (A NON-SINGLE-NODE TREE)

P%(Q) IS KNOCK THE NOCK FORMULA Q WITH THE SUBJECT P

0 TREE ADDRESS: •%(0 P) -> +P•

1 JUST PRODUCE P: •%(1 P) -> P

2 NEW SUBJECT, NEW FORMULA, KNOCK THEM: •%(2 P Q) -> •%(P)%(•%(Q))
KNOCK P AND KNOCK Q, THEN KNOCK THE Q-PRODUCT WITH THE P-PRODUCT AS SUBJECT

3 IS P A CELL? •%(3 P) -> .Y OR .N

4 ADD ONE TO P ("INCREMENT"): •%(4 P) -> P PLUS ONE

5 IS P IDENTICAL TO Q?: •%(5 P Q) -> .Y OR .N

6 IF-THEN-ELSE: •%(6 P Q R) -> Q IF P PRODUCES .Y, R IF .N

7 P MAKES A NEW SUBJECT FOR Q (CHAIN OR "COMPOSE")
•%(7 P Q) -> •%(P)%(Q)
FIRST KNOCK P, THEN WITH THAT AS THE NEW SUBJECT KNOCK Q

8 P ADD (AS HEAD) TO SUBJECT FOR Q •%(8 P Q) -> ⟦%(P) •⟧%(Q)

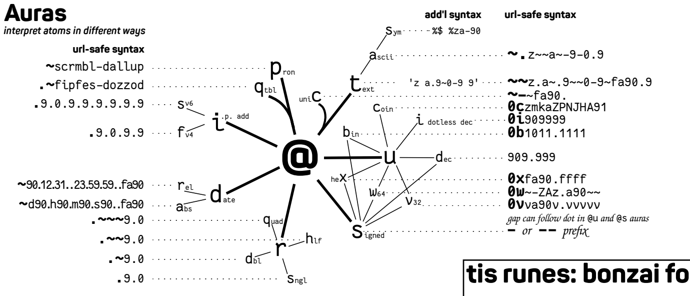
9 BUILD A CORE & FIRE AN ARM •%(9 P Q) -> %(Q)%(+P%(Q))
P IS A TREE ADDRESS, Q IS A FORMULA THAT BUILDS A CORE.

10 REPLACE PART OF SUBJECT: •%(10 P) -> +P•

11 HINT, E.G. JET: •%(11 P Q) -> •%(Q)

Auras

interpret atoms in different ways



Mod Your Tree

tis runes: bonzai forestry

prune off, graft on, and swap out. a couple of oft-used fas runes, too. 7 replaces the subject, 8 prepends to subject, 10 changes the subject.

NOCK 7

=> *compose 2 hoons. erases old subject!*

=< *compose 2 hoons in reverse order*

`(sub +2 1):[(add 7 4) .] → 27`

=~ == *compose many hoons*

```
=~  [cet:yo .]
    [(sub - 124) .]
    [(div - 100) .]
    [(div - 52) .]
    [(mul - 4) .]
    [(sub - 1) .]
    -
  ==
→ 27
```

NOCK 8

=+ *add a new noun to the subject*

=- *=+ in reverse order*

=/ `pin` *add be-faced noun to subject*

=; *=/ in reverse order*

=| *a typed bunt to the subject*

/+ *import from /lib directory*

/* *import converted to a mark*

NOCK 10

=. *change a leg*

=: == *change multiple legs*

=^ *=/ & =. pin a noun & change leg*

=? *change leg if a statement is true*

=* `alas` *define alias*

=, *define a bridge / expose a namespace*

Leftovers

/? *pin a hoon kelvin version number*
/- *import from /sur*
/= *import custom path w/ a face*
/% *import mark def. from /mar*
/$ *import mark conversion gate from /mar*
/* *import contents of a file converted to a mark (build-time static data)*
/~ *import contents of a directory under* `face=(map @ta type)`
!; *raw vase.* `!;(*type 'mum')`
!< *de-vase & pass through a mold.* `!<(@ta !>('mum')) → ~.mum`
!= *make the nock formula for a hoon expression*
!@ *evaluate conditional on existence of wing*
!. *turn off stack trace*
!? *restrict hoon kelvin version*
^: *now redundant*

~| *print in stack trace if failure*
~! *print type if failure*
~? *print conditionally (use for debugging)*
~% *register a jet*
~/ *register a jet with registered context*
~+ *cache a computation*
~< *raw hint, applied to product backward*
~> *raw hint, applied to product forward*
~= *detect duplicate*
~$ *profiler hit counter*
;< *glue a pipeline together (monadic bind)*
;~ == *glue pipeline together w/ product-sample adapter*
;; *normalize w/ a mold, assert fixpoint*
;+ *make a single xml node*
;= == *make a list of xml nodes*
;* *make xml nodes w/ marl-making hoon*
;/ *turn a tape into an xml element*

Do Tricks *with enough logic, you can do anything logical*

cell-making notes and syntaxes

`[head tail] → :-(head tail)`
`[1 2 3 4] = [1 [2 [3 4]]]` *group right by default*
`1^2 → [1 2]`
`1/2 → [%1 2]` *1st item made cold*
`` `1 → [~ 1] `` *prefix w/ null (a unit, once cast)*
`~[1 2] → [1 2 ~]` *suffix w/ null (a list, once cast)*
`[1 2]~ = ~[[1 2]] → [[1 2] ~]`
as above but whole cell becomes a single list item.
`._1_2_3__ → [1 2 3]` *url-safe syntax*

Misc.: tapeify, tankify, lose face, spear type

`<[1 2]>` *renders list as a tape.*
`>[1 2]<` *renders list as a tank.*
`,` *obtain faceless data. e.g.* `27` *not* `i=27`
`-:!>(noun)` *the* `type spear:` *shows the head of the vase of a noun, which will be that noun's type.*

cell-making regular runes

:- *make a cell – a pair of 2 nouns. order matters. 1st called the head, 2nd tail.*

:_ *reverse order*

:+ *3* **:^** *4* **:*** == *any*

:~ == *any # of elements, adds a ~ (not a proper list until cast).* `~[1 2]`

.* *run code (core or not) w/ NOCK 2*

.? *is it a cell? NOCK 3*

.+ *increment NOCK 4*

.= *is it equal? NOCK 5*

.^ `scry` *just read, no event, not logged. "nock 12"*

?: `if`*-then-else: branch on true or false. do 2nd child if the 1st is true, 3rd if false.*

?> *must be: true or else fails (crashes with !!)*

?| `|()` *or* **?&** `&()` *and*

?^ *a cell?* **?@** *an atom?*

?- == *whole menu of if-then cases, no default. ifs are specs.* `?-(meal %meat 20, %eggs 5, %pulse 1)`

?. `ifn't` *then else: reverse order: false or true*

?< *mustn't be: false or fails*

?! `!hoon` *not*

?~ *null?*

?+ == `?-` *w/ a default* `?+(x adult to4 prek, %5 krgn, sx-11 elem, teen hs)`

?= `fish` *test whether a noun fits a mold. a natural rune aka %fits*

?# *a promised upcoming rune that will fish using wet molds*

;: == `:()` *make a two-argument gate repeat in a chain. E.g.:* `:(mul 3 3 3)` *(end tall: ==)*

~& *print as program progresses (use for debugging) w/ up to 3 optional* **>***s for colorcoded log levels.*

~_ *print in stack trace, user-formatted*

!: *turn on stack trace*

!, *show AST* `!,(*hoon 'mum')`

!> `vase` *a cell of a spec's AST and a noun matching the spec.*

```
!>('hi')    →  [#t/@t q=26.984]
!>("hi")    →  [#t/"" q=[104 105 0]]
!>([%2 2]) →  [#t/[%2 @ud] q=[2 2]]
```

== `stet` *ends a rune with an indefinite number of children. most runes have a fixed number of children, so for them no terminator at all is needed.*

:: `comment` *ends code parsing on that line to insert a comment. code parsing resumes on the new line. you cannot end a hoon file with a comment.*

!! `crash` *not a bad thing: means to just stop computing, be done. your urbit will now sit idle, awaiting your next command. all programs need not end with this hoon command – they'll crash on their own once all the nock formulas they fired have finished reducing (*NOCK OF A → A *when A doesn't begin with a nock formula code (atoms 0 through 11))*