

Work Your Core

bar runes: core strength *build your core*

| | multi arms | one arm | auto fired | besam pled | wet | trans muted |
|-------|---------------|------------|---------------|---------------|-----|----------------|
| NOCK | % | ++ | | | | |
| corp | % | ++ | | | | |
| trap | . | | \$ | | | |
| loop | - | | \$ | %= | | |
| cork | ^ | ++ | | %= | | |
| gate | = | | \$ | | +6 | |
| gasp | : | | \$ | | +6 | |
| door | - | ++ | | | +6 | |
| port | ~ | | \$ | | +6 | |
| mine | ? | | \$ | | | |
| lava | @ | ++ | | | | |
| gill | * | | \$ | | +6 | |
| ptri | \$ | | \$ | | +6 | |
| p'hep | -- | | | | | |

produce a generic dry core. any number of arms in battery. all other dry bar hoonos are macros of %.

produce a corp with one arm named \$ (eliminate the need for writing ++ \$)

produce a trap and then automatically "kick" it, i.e. compute its \$ arm with %= (eliminates <= \$)

produce a corp (of any number of arms) with an automatically-"popped" \$ arm (so like a loop, but with additional non-auto-fired arms)

produce a trap with a "sample" (eliminates need for three nested cores with +=s)

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 hunt to 0

produce a corp (of any number of arms) with a sample (so like a gate, but bigger - multiple arms)

produce an iron (contravariant - less specifically specced (superset) input is ok) gate

produce a trap (.) that is cast (with ^?) to lead (bivariant - both superset and subset specs are ok)

produce a generic wet core. any number of arms in battery. all other wet bar hoonos are macros of @.

produce a wet gate (a one-armed core with sample)

produce a gill specifically for producing a mold, whose body is parsed in pattern/spec mode.

end a multi-arm core

cen runes: just do it *fire your core*

%= pull a wing, with changes. the most important rule: everything that in any way references a "wing" (a portion of your free mansion) turns into a %=. the first cell you give it is the main wing, the location of the arm you wish to change or the arm you wish to change and then for this could be a file (name) or a tree address (e.g. >> \$). next you give it a rig and 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 express the type of a subwing, but %_ will fail if you try to change type. (end tail: ==)

%%+ 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 like standard add gate has an a and a b defined)

%- fire a core. irregular for %; and thus whose family except %_. is [gate arg(s)] generally thought of as the rune for slamming a gate, but also used to give any non-auto-fired arm. (l.(7)) (sq4 49) (add 5 2) (into noun-descs 7 %luckiest) (qual % * @ 7)

%_ reverse order. %_ (s 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 | <i>begins a normal arm</i> |
| +\$ | jib | <i>begins a structure-building arm. the arm will contain spec code, which means the trivial molds: * @ ^ ? ~ ~ , named molds (e.g. tape, list, tank), or bus rule expressions.</i> |
| ++ | pseu | <i>make alias (alias) "pseudo-arm".</i> |
| + | chapter | <i>common use as a pseudo-arm to give a face to a faceless core: ++ this .</i> |
| -- | p'hep | <i>begins a chapter (a group of arms). just for organization.</i> |
| | | <i>terminate your list of arms (and thus your core) in a multi-arm core.</i> |

Catch Your Typos

buc runes: shape sifters

Now's the type system defines *data*, the point of it is for you, the programmer, to declare your intentions of what you're trying to do and then specifically what kind of data you're trying to generate and how around. This helps you because then if you make some mistake or error in your program, that will throw many times the type of the data and the type system will catch that and fail to compile.

These *bus* runs are the *model* builders. what's a *model*, you ask? a *model* is just a specific kind of gate, and a very simple one. it takes in as many any noun and as long as that noun is within its *spec*, it outputs that same noun (accept for *S* and *S*-).










































































































































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 *model* is like that.

at a few basic *models* 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-boolean |
| \$ _ anyhoon | | | | %spam totally discards the sample it's given and just produces the noun produced by the \$ _'s child noun no matter what. |
| \$ = skin=spec | | | | my-face=@ gives a face to a spec. |
| \$: [spec(s)] | | | | [a=@ b=@ c=@ d=@27] create a cell <i>mold</i> (to pass through, the noun must be the <i>cold</i>), the required shape of the cell further defined by specs within the \$ _: |
| \$? ?(spec(s)) | | | | ?[good %evil] the noun must fit any of the specs in the list (an "or" union) use only for a union of atoms. more complex nouns, use % , \$ *, and %\$. |
| \$ @ | | | | \$@(@*) 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 @] [xfizz tape] [buzz tape]) a whole list of cell molds, with the head a constant (cold atom), to be selected based on the head. |
| \$ > | | | | \$>(? [rule *]) → the head must be a flag (0 or 1) \$>[? fix a cell] → the head must fit a mold. \$>[? one cell] → select one mold from a % list, using above example: \$>[number bccn-exmpl] → [number @] takes a spec and further restricts it. \$>[(@ (curr lth 10))] creates a mold for atoms less than 10. |
| \$! | | | | [a-%mold combining-old-and-new-versions gate-normalizing-to-new] upgrades a structure from an old version to a new one. \$(%\$([xv1 !] [xv2 !]) [x=@ [a=@ b=@] [xv2 b.x]]) takes any spec and define a custom default value for it. \$(%\$@ \$(face @) \$(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 burn). |
| \$ = | | | | \$(@ ~ @) will take a cell of two atoms and output a cell. \$(@ %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 χ " means "cast away all info except χ ".

| | |
|---|---|
| <p>             </p> | <p> *spechoon <i>cast to a spec by simply explicitly writing the spec to which you wish to cast the noun.</i> <i>assign a face to a noun.</i> </p> |
| <p> *skinhoon <i>cast to "whatever this is." cast by writing a noun and letting the system infer its spec.</i> $+ (>? \text{c} \text{fn}z) \rightarrow \text{'fn}z$ $\sim (-\text{f}y\text{n} \text{'e} @ 255) \rightarrow -\text{f}es$ $\text{*mul} \rightarrow 1$ $\text{*mul:rs} \rightarrow 0$ (a poorly-chosen hint) </p> | <p> *spec <i>bunt</i> – i.e. produce the default value of the spec you explicitly write. <i>[gate hoon] cast based on the noun produced by passing q to p – the hoon (any expression) to the gate (e.g. $\text{V}mo$ which makes it a list).</i> </p> |
| <p> fold constant </p> | <p>             </p> |
| <p>             </p> | <p>             </p> |
| <p>             </p> | <p>             </p> |
| <p>             </p> | <p>             </p> |
| <p>             </p> | <p>             </p> |
| <p>             </p> | <p>      </p> |

Auras

interpret atoms in different ways

[illegible]

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 | NOCK 8 | NOCK 10 |
|---|---|--|
| => compose 2 <i>hoons</i> , erases old subject | =+ add a new noun to the subject | =. change a leg |
| = compose 2 <i>hoons</i> in reverse order | =- =+ in reverse order | =: change multiple legs |
| (sub +2 1)B(mu1 7 4) . + 27 | =/ pin add <i>fo-hoon</i> noun to subject | =: = & -. pin a noun & change leg |
| = compose many <i>hoons</i> | =; =/ in reverse order | =? change leg if a statement is true |
| =- (cat; yz - 124) . | =! add a typed <i>bunt</i> to the subject | =* <i>alaf</i> define alias |
| (dtw -100) . | =! import from /lib directory | =* <i>alaf</i> define a bridge / expose a namespace |
| (mul -62) . | =/ import converted to a mark | |
| (mul -4) . | | |
| (sub -4) . | | |
| + 27 | | |

Do Tricks

| | | | | |
|--|---|---|--|--|
| <p>cell-making notes and syntaxes</p> <p>(head tail) → :-(head tail) [1 2 3 4] = [1 [2 [3 4]]] group right by default [2] → [1 2] [2] → [1 2] 1st item made cell [1 2] → [1 -] 1st prog/ u/n! (a unit, once cast) [1 2] → [1 - 2] suffix u/n! (a list, once cast) [1 2] = - [1 [2]] → [1 [2 -]] as above but whole cell becomes a single list item [1 2] → [1 2 3] ur-lf-suff syntax</p> <p>Misc: tapeifu, tankifu, lose face, spear type [1 2] renders list as a tag >[1 2] obtain faceless data; eg. 27 not i=27 -!> (noun) the type spear: shows the head of the vase of a noun, which will be that noun's type.</p> | <p>cell-making regular runes</p> <p>1 make a cell - a pair of 2 nouns, order matters! 2nd called the head, 2nd tail. reverse order 2 any # of elements, adds a - (not a proper list until cast). 2 1</p> <p>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.</p> | <p>3 run code (core or u/n! Nock 2 is it a cell? Nock 3 increment Nock 4 is it equal? Nock 5 scry just read, no even, not logged. "nock 12"</p> <p>4 if-then-else: branch on true or false, do 2nd child if the 1st is true, 3rd if false. 5 must be: true or else fals (cashes with !) 6 or 7 a cell? 8 whole menu of if-then cases, no defaults. eg. 5, 2-u/n! meat 20, megs 5, 2-u/n! 1</p> <p>comment ends code parsing on that line to insert a comment. code parsing resumes on the new line. you cannot end a noun file with a comment.</p> | <p>9 ifn't then else: reverse order: false or true 10 mustn't be: false or fals 11 noon not 12 a two-argument gate repeat in a chain. E.g.: :(mul 3 3 3) (dec := =) 13 print as program progresses (use for debugging) w/ up to 3 optional args for color-coded log levels.</p> <p>fish test whether a noun fish a. a natural rune aka 181s promised upcoming rune that will fish using wet molds 14 make a two-argument gate repeat in a chain. E.g.: :(mul 3 3 3) (dec := =) 15 print as program progresses (use for debugging) w/ up to 3 optional args for color-coded log levels.</p> <p>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 noun command - they'll crash on their own once all the nock formulas they through have finished reducing NOCK OF A → A when A doesn't begin with a nock formula code (qoms & through 111)</p> | <p>16 print in stack trace, user-formatters turn on comment show AST! ,(*noon (dec 28)) 17 Vase a cell of a spec's AST and a noun matching the spec. [1](<h1> → [0]/@t q=26.984 [1](<h1> → [0]/@t q=[104 105 0] [1](<X2 2) → [0]/@X2 @ud q=[2 2]</p> |
|--|---|---|--|--|