| bar core expressions

|_ `spec alas (map term tome)`
produce a door (a core with sample)

|% `(unit term) (map term tome)`
produce a core (battery and payload)

|@ `(unit term) (map term tome)`
produce a wet core (battery and payload)

|: `[hoon hoon]`
produce a gate with a custom sample

|. `hoon`
produce a trap (a core with one arm)

|- `hoon`
produce a trap (a core with one arm) and evaluates it

|^ `hoon (map term tome)`
produce a core whose battery includes a $ arm and computes the latter

|~ `[spec value]`
produce an iron gate

|* `[spec value]`
produce a wet gate (a one-armed core with sample)

|= `[spec value]`
produce a dry gate (a one-armed core with sample)

|? `hoon`
produce a lead trap

|$ `(lest term) spec`
produce a mold

$ buc structures

$@ `[spec spec]`
structure that normalizes a union tagged by head atom

$_ `hoon`
structure that normalizes to an example   `_foo`

$: `(list spec)`
form a cell type (tuple)   `[a=foo b=bar c=baz]`

$% `(list spec)`
structure that recognizes a union tagged by head atom (*e.g.,* a list of named parameters)

$< `[spec spec]`
structure from filter (excluding)

$> `[spec spec]`
structure from filter (requiring)

$| `[spec hoon]`
structure with verification

$& `[spec hoon]`
repaired structure

$^ `hoon`
structure that normalizes a union tagged by head depth (cell)

$~ `[hoon spec]`
define a custom type default value

$- `[spec spec]`
structure that normalizes to an example gate

$= `[skin spec]`
structure that wraps a face around another structure   `foo=bar`

$? `(list spec)`
form a type from a union of other types   `?($foo $bar $baz)`

$. `[spec (map term spec)]`
structure as read–write core
$* `hoon`   `*foo`
bunt a value (provide default "empty" value)
$; `hoon`
manual structure

% **cen calls & samples**

%_ `[wing (list (pair wing hoon))]`
resolve a wing with changes, preserving type
%. `[hoon hoon]`
call a gate, inverted
%^ `[hoon hoon hoon hoon]`
call a gate with triple sample
%+ `[hoon hoon hoon]`
call a gate with a cell sample
%- `[hoon hoon]`
call a gate   `(fun arg)`
%: `[hoon (list hoon)]`
call a gate with many arguments
%~ `[wing hoon hoon]`
evaluate an arm in a door   `~(arm core arg)`
%* `[wing hoon (list (pair winghoon))]`
evaluate an expression, then resolves a wing with changes
%= `[wing (list (pair wing hoon))]`
resolve a wing with changes   `foo(x 1, y 2, z 3)`

: **col cells**

:_ `[hoon hoon]`
construct a cell, inverted
:^ `[hoon hoon hoon hoon]`
construct a cell, 4-tuple   `[a b c d]`
:+ `[hoon hoon hoon]`
construct a cell, 3-tuple   `[a b c]`
:- `[hoon hoon]`
construct a cell, 2-tuple   `[a b], a^b (a^b^c)`
:~ `(list hoon)`
constructs a null-terminated list   `~[a b c]`
:* `(list hoon)`
construct an n-tuple   `[a b c d e …]`
::
mark a comment (digraph, not rune)

. **dot nock evaluations**

.+ `atom`
increment an atom using Nock 4   `+(42)`
.* `[hoon hoon]`
evaluate using Nock 2
.= `[hoon hoon]`
test for equality using Nock 5   `=(a b)`
.? `hoon`
test for cell or atom using Nock 3
.^ `[spec hoon]`
load from namespace using Nock 12 (scry)

-/=	**terminators**
`--`	terminate core expression (digraph, not rune)
`==`	terminate running series of Hoon expressions (digraph, not rune)

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

~	**sig interpreter hints**
`~|`	`[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
`~&`	`[@ud hoon hoon]`
	print (used for debugging)
`~?`	`[@ud hoon hoon hoon]`
	print conditionally (used for debugging)
`~=`	`[hoon hoon]`
	detect duplicate
`~!`	`[hoon hoon]`
	print type if compilation failure

;	**`mic` macros**	
;:	`[hoon (list hoon)]`	
	call a binary function as an n-ary function	`:(fun a b c d)`
;/	`hoon`	
	(Sail) yield tape as XML element	
;<	`[spec hoon hoon hoon]`	
	glue a pipeline together (monadic bind)	
;~	`[hoon (list hoon)]`	
	glue a pipeline together with a product-sample adapter (monadic bind)	
;;	`[spec hoon]`	
	normalize with a mold, asserting fixpoint	
;+		
	(Sail) make a single XML node	
;*		
	(Sail) make a list of XML nodes from Hoon expression	
;=	`marl:hoot`	
	(Sail) make a list of XML nodes	

=	**`tis` subject modifications**	
=|	`[spec hoon]`	
	combine default type value with the subject	
=.	`[wing hoon hoon]`	
	change one leg in the subject	
=?	`[wing hoon hoon hoon]`	
	change one leg in the subject conditionally	
=^	`[skin wing hoon hoon]`	
	pin the head of a pair; changes a leg with the tail	
=:	`[(list (pair wing hoon)) hoon]`	
	change multiple legs in the subject	
=/	`[skin hoon hoon]`	
	combine a named noun with the subject	
=;	`[skin hoon hoon]`	
	combine a named noun with the subject, inverted	
=<	`[hoon hoon]`	
	compose two expressions, inverted	`foo:bar`
=>	`[hoon hoon]`	
	compose two expressions	
=-	`[hoon hoon]`	
	combine a new noun with the subject	
=*	`[(pair term (unit spec)) hoon hoon]`	
	define an alias	
=,	`[hoon hoon]`	
	expose namespace (defines a bridge)	
=+	`[hoon hoon]`	
	combine a new noun with the subject	
=~	`(list hoon)`	
	compose many expressions	

?	**`wut` conditionals**	
?|	`(list hoon)`	
	logical OR (loobean)	`|(foo bar baz)`
?:	`[hoon hoon hoon]`	
	branch on a boolean test	
?.	`[hoon hoon hoon]`	

branch on a boolean test, inverted
?<   `[hoon hoon]`
negative assertion
?>   `[hoon hoon]`
positive assertion
?-   `[wing (list (pair spec hoon))]`
switch against a union, no default
?^   `[wing hoon hoon]`
branch on whether a wing of the subject is a cell
?=   `[spec wing]`
test pattern match
?#   `[skin wing]`
test pattern match
?+   `[wing hoon (list (pair spec hoon))]`
switch against a union, with default
?&   `(list hoon)`
logical AND (loobean)   `&(foo bar baz)`
?@   `[wing hoon hoon]`
branch on whether a wing of the subject is an atom
?~   `[wing hoon hoon]`
branch on whether a wing of the subject is null
?!   `hoon`
logical NOT (loobean)   `!foo`

!   `zap` **wildcards**

!:   `hoon`
turn on stack trace
!.   `hoon`
turn off stack trace
!,   `[*hoon hoon]`
emit AST of expression (use as `!,(*hoon expression)`)
!;   `[hoon hoon]`
emit the type for an expression using the type of type (raw `!>`)
!>   `hoon`
wrap a noun in its type
!<   `hoon`
lift dynamic value into static context
!@   `[(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

/   `fas` **build operations (++ford arm of %clay)**

/?   `foo`
pin a version number
/-   `foo, *bar, baz=qux`
import a file from the `sur` directory (* pinned with no face, = with specified face)
/+   `foo, *bar, baz=qux`
import a file from the `lib` directory (* pinned with no face, = with specified face)
/=   `clay-raw  /sys/vane/clay`

       import results of user-specified path wrapped in face

/%   `%mark`

       import mark definition from `mar/`

/$ `%from %to`

       import mark conversion gate from `mar/`

/*   `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 a directory under `face=(map @ta type)`

   +   **`lus` arm definitions**

+|

       label a chapter (produces no arm)

+$ `[term spec]`

       produce a structure arm (type definition)

++   `[term hoon]`

       produce a (normal) arm

+*   `[term term spec]`

       produce a type constructor arm

syntax

```
+1:[%a [%b %c]] [%a [%b %c]]
+2:[%a [%b %c]] %a
+3:[%a [%b %c]] [%b %c]
+4:[%a [%b %c]] %ride failed
+6:[%a [%b %c]] %b
+7:[%a [%b %c]] %c
```

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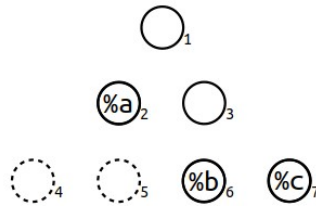
```
.:[%a [%b %c]] [%a [%b %c]]
-:[%a [%b %c]] %a
+:[%a [%b %c]] [%b %c]
-<:[%a [%b %c]] %ride failed
+<:[%a [%b %c]] %b
+>:[%a [%b %c]] %c
```

```
&n  nth element
|n  tail after nth element

<[1 2 3]>  renders list as a tape
>[1 2 3]<  renders list as a tank

     ·  current subject
     +  +:.
     -  -:.
    +>  +>:.
 a.b.c  limb search path
```

lark syntax equivalents

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

```
^face  face in outer core (^^face)
..arm  core in which ++arm is defined
,  ,.  strip the face

-:!>  type spear, use as -:!>(.3.14)
```

```
        ~  0 (nil)                eny  entropy              `a  [~ a]
 %.y    &  yes/true/0             now  current time    ~[a b c]  [a b c ~]
 %.n    |  no/false/1             our  ship            [a b c]~  [[a b c] ~]
       %a  constant                                        a/b  [%a b]
        $  empty term (@tas)
```

elementary molds

```
'urbit' cord, atom @t           * noun
"urbit" tape or list of characters   @ atom (atom)
 =wire  shadow type name (in defn)   ^ cell
 /path  path name                    ? loobean
     %  current path                 ~ 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 codepoint*	`~-~45fed.`
`@d`	*Date*	
`@da`	*Date, absolute*	`~2020.12.25..7.15.0..1ef5`
`@dr`	*Date, relative*	`~d71.h19.m26.s24..9d55`
`@f`	*Loobean (for compiler, not castable)*	`&`
`@i`	*Internet address*	
`@if`	*IPv4 address*	`.195.198.143.90`
`@is`	*IPv6 address*	`.0.0.0.0.0.1c.c3c6.8f5a`
`@n`	*Nil (for compiler, not castable)*	`~`
`@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-bit*	`.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*	`0c1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa`
	from set `123456789abcdefghijklmnopqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ`	
`@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`

Capital letters at the end of auras indicate the bitwidth in binary powers of two, starting from `A`.

`@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)	

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]                                  slot operator (noun at 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

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

*a                  *a                                      interpret
```