Debugging Hoon

Error	Interpretation	Mitigation
HOON ERRORS		
`dojo: hoon expression failed`	input hoon failed to compute	correct expression
`find.` *`foo`*	failure to locate a limb in subject	check the wing (limb search path); make sure limb exists
`find.$`	failure to call item as gate	ensure that the code is calling a gate
`find-fork`	insufficient resolution in typechecker	use `?>` to assert type before use
`fish`	pattern matching	
`fish-core`	attempting to match a core as a mold	don't use a core with `?=` pattern-matching
`fish-loop`	recursive mold definition	don't use mold types like `list` with `?=` pattern matching
`mint/play`	conversion of AST to Nock	
`mint-lost`	a branch in a conditional can never be reached	make sure all branches are reachable
`mint-nice`	failure to cast	
`mint-vain`	hoon never executed; impossible match in `?-`, `?+`, `?~`, `?=`	make sure all branches are reachable
`mull`	type inference for wet cores	
`mull-bonk`	various pattern matching errors	
`mull-grow`	failure to compile at wet gate callsite	
`mull-nice`	type nesting errors	
`need/have`	expected mold & actual received mold	check the structure and type of molds; cast auras
`nest-fail`	failure to match call signature of gate	
`generator-build-fail`	Dojo unable to compile generator into valid program	check structure of Hoon in generator file
`syntax error`	malformed Hoon syntax	check your
RUNTIME ERRORS		
`bail: exit`	semantic failure	
`bail: evil`	bad crypto	
`bail: intr`	interrupt	
`bail: fail`	execution failure	
`bail: foul`	assertion of failure	
`bail: meme`	out-of-memory	
`bail: need`	network block	
`bail: oops`	assertion failure	
`bail: time`	operation timeout	
`loom: corrupt`	memory corruption	
`pier: serf unexpectedly shut down`	runtime crash	debug on basis of other error messages

COMMON BUGS

Aura mismatches	`mint-nice` is the characteristic error type.	Pass thru empty aura b/f final cast: `^-(@ud ^-(@ 'foo'))`
Generator issues		Check children of each rune to make sure they match. Check return types of expressions (or limit with `?>`/`^-`).
Shadowed faces	Variable names (such as `json`) covered in the subject by another limb name.	Use `^` ket to find the *n*th match or change limb name.

STRATEGIES

Stack debugging. Turn this on with `!:` zapcol; `!.` zapdot turns this off again. The output on a crash returns the stack and the current file/line number.

Employ `~&` sigpam `printf`-style debugging freely. This should have no effect on code execution as long as what you are printing isn't a complicated expression.

Bisection search. Stub out limbs you aren't currently testing with the crash rune `!!` zapzap. Use this to rapidly target where your code is going awry.

Build it again. Remove all of the complicated code from your program and add it in one line at a time. For instance, replace a complicated function with either a `~&` and `!!`, or return a known static hard-coded value instead. That way as you reintroduce lines of code or parts of expressions you can narrow down what went wrong and why.

Double-check the documentation and source for the gate in question. Make sure that each element of the sample (argument) does what you think it does. Make sure that you have a good grasp on any strange terminology employed.

DEBUGGING TOOLS

~ sig tools	`~&` sigpam emits printed messages as a side effect	`~_` sigcab produce a developer-formatted tracing message
	`~|` sigbar turns on a tracing message (for stack debugging)	`~!` sigzap print type on compilation failure
! zap tools	`!:` turn on stack debugging	`!.` turn off stack debugging
`%gall` `%dbug` app	`|start %dbug` Navigate to `http://localhost:8888/debug` (with the appropriate ship URL)	
Ship maintenance	`|pack` compact memory `|meld` unify memory (eliminate redundant subtrees) `:goad %force` force `%gall` to rebuild agents	
Profiling flags	`-j` create a JSON trace file in `.urb/put/trace` `-P` turn on profiling	
Debugging flags	Compile with `enableDebug = true` in `default.nix`. Run with `-g` flag to monitor memory behavior.	