Hoon and You

A Functional Programming Perspective

J LeBlanc
WWT Asynchrony Labs

Introduction

The *Telos* of Hoon

- Urbit - clean-slate system software stack
- Martian software
- All assumptions revisited!

Kelvin Versioning

- Versions decrease towards 0K
- Typical software improvement is additive

CloudTron v4.0

- Now with Blockchain!
- Add a rich, smoky bacon scent
  to all your files stored in the cloud!

JSON

```
{
  "name": "J LeBlanc",
  "planets": [
    "~ribben-donnyl",
    "~mallet-rilmul"
  ],
  "favorite-number": 27,
  "city": "Saint Louis”
}
```

JSON

```
{
  "name": "J LeBlanc",
  "planets": [
    "~ribben-donnyl",
    "~mallet-rilmul",
  ],
  "favorite-number": 27,
  "city": "Saint Louis”,
}
```

Frozen Software

- Progress => reducing the number of things which have to change.
- OK => Platonic ideal

Simplicity

- Freezable software must be simple
- Ruthless about scope
- Minimal interaction with things out of scope

Simplicity

- A solution to the Law of Leaky Abstractions:
- *If there's no functionality to abstract away, there's nothing to leak.*
- Complexity is additive

Nock

```
A noun is an atom or a cell.  An atom is a natural number.
A cell is an ordered pair of nouns.

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
=a                  =a

/[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]
*[a 1 b]            b
*[a 2 b c]          *[*[a b] *[a c]]
*[a 3 b]            ?*[a b]
*[a 4 b]            +*[a b]
*[a 5 b]            =*[a b]

*[a 6 b c d]        *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
*[a 7 b c]          *[a 2 b 1 c]
*[a 8 b c]          *[a 7 [[7 [0 1] b] 0 1] c]
*[a 9 b c]          *[a 7 c 2 [0 1] 0 b]
*[a 10 [b c] d]     *[a 8 c 7 [0 3] d]
*[a 10 b c]         *[a c]
*[a 12 [b c] d]     #[b *[a c] *[a d]]

*a                  *a
```

Nock

A noun is an atom or a cell. An atom is a natural number.
A cell is an ordered pair of nouns.

```
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
=a                  =a

/[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
```

Nock

```
A noun is an atom or a cell.  An atom is a natural number.
A cell is an ordered pair of nouns.

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
=a                  =a

/[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
```

Nock

```
*[a [b c] d]        [*[a b c] *[a d]]

*[a 0 b]            /[b a]
*[a 1 b]            b
*[a 2 b c]          *[*[a b] *[a c]]
*[a 3 b]            ?*[a b]
*[a 4 b]            +*[a b]
*[a 5 b]            =*[a b]

*[a 6 b c d]        *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
*[a 7 b c]          *[a 2 b 1 c]
*[a 8 b c]          *[a 7 [[7 [0 1] b] 0 1] c]
*[a 9 b c]          *[a 7 c 2 [0 1] 0 b]
*[a 10 [b c] d]     *[a 8 c 7 [0 3] d]
*[a 10 b c]         *[a c]
*[a 12 [b c] d]     #[b *[a c] *[a d]]

*a                  *a
```

Nock

```
[8 [1 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0 6] 0 7] 9 2 0 1]
```

Nock

```
*[a [b c] d]        [*[a b c] *[a d]]

*[a 0 b]            /[b a]
*[a 1 b]            b
*[a 2 b c]          *[*[a b] *[a c]]
*[a 3 b]            ?*[a b]
*[a 4 b]            +*[a b]
*[a 5 b]            =*[a b]

*[a 6 b c d]        *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
*[a 7 b c]          *[a 2 b 1 c]
*[a 8 b c]          *[a 7 [[7 [0 1] b] 0 1] c]
*[a 9 b c]          *[a 7 c 2 [0 1] 0 b]
*[a 10 [b c] d]     *[a 8 c 7 [0 3] d]
*[a 10 b c]         *[a c]
*[a 12 [b c] d]     #[b *[a c] *[a d]]

*a                  *a
```

Nock

```
A noun is an atom or a cell.  An atom is a natural number.
A cell is an ordered pair of nouns.

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
=a                  =a

/[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]
*[a 1 b]            b
*[a 2 b c]          *[*[a b] *[a c]]
*[a 3 b]            ?*[a b]
*[a 4 b]            +*[a b]
*[a 5 b]            =*[a b]

*[a 6 b c d]        *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
*[a 7 b c]          *[a 2 b 1 c]
*[a 8 b c]          *[a 7 [[7 [0 1] b] 0 1] c]
*[a 9 b c]          *[a 7 c 2 [0 1] 0 b]
*[a 10 [b c] d]     *[a 8 c 7 [0 3] d]
*[a 10 b c]         *[a c]
*[a 12 [b c] d]     #[b *[a c] *[a d]]

*a                  *a
```

Nock Tradeoffs

- Simple but still useful
- Macros help further development
- The meaning of atoms TBD
- Binary tree => maximal power, minimal complexity

Nock Tradeoffs

- Also missing:
  - Variables
  - Functions
  - An environment
  - A syntax
  - Error handling of any kind

Hoon

Hoon

- Compiles down to Nock
- Designed for compiling, running, and reloading programs
- Can hot-load apps, kernel modules, or the whole OS by running a function on an incoming packet

Hoon

- Compiler written in Hoon
- Latest version: 143K
- Strives for simplicity

Hoon’s Terrifying Syntax - Runes

ASCII Pronunciation

ace	[1 space]
bar	|
bas	\
buc	$
cab	_
cen	%
col	:
com	,
doq	"
dot	.
fas	/
zap	!
gal	<
gap	[>1 space, nl]
gar	>
hax	#
hep	-
kel	{
ker	}
ket	^
lus	+
pam	&
pat	@
pal	(
par	)
sel	[
sem	;
ser	]
sig	~
soq	'
tar	*
tec	`
tis	=
wut	?

Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Fibonacci Numbers

```
|=  n/@ud                         :: 1
=/  a  0                          :: 2
=/  b  1                          :: 3
|-                                :: 4
^-  (list @ud)                    :: 5
:-                                :: 6
  a                               :: 7
?:  =(0 n)                        :: 8
  ~                               :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

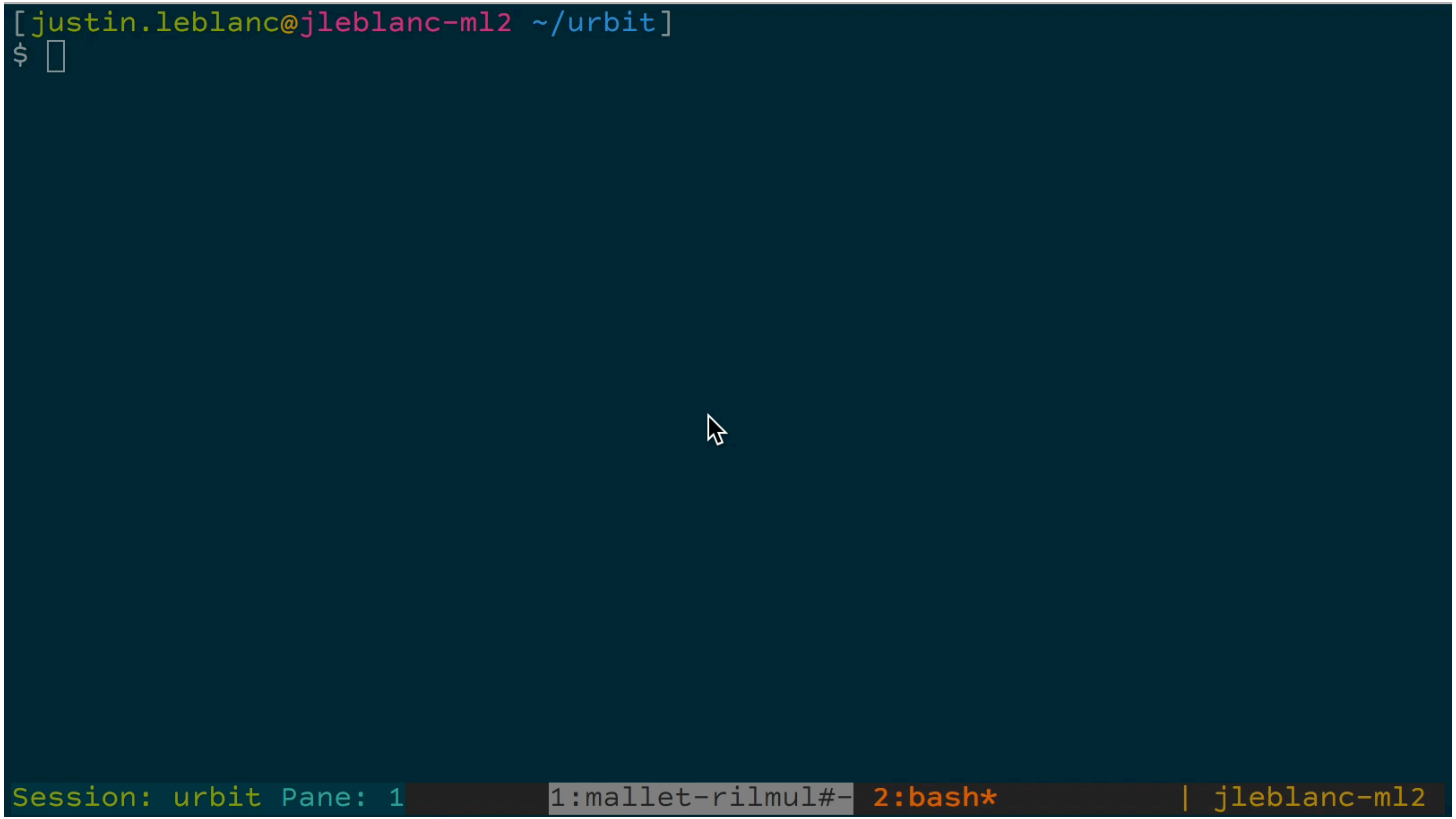
Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Fibonacci Numbers

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Demo



```
[justin.leblanc@jleblanc-ml2 ~/urbit]
$
```

```
Session: urbit Pane: 1            1:mallet-rilmul#-  2:bash*              | jleblanc-ml2
```

Hoons

- Runes expect subexpressions
- No “))))))))))”
- You have to know subexpressions
- Good style helps

Creating a Cell

- Create cells with `:-`
- Defined as:

```
{%clhp p=hoon q=hoon}
```

- Create a cell with two values:

```
:-  42  420
```

Creating a Cell

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Declare a Variable

- Declare variables with =/
- As seen in Fibonacci:

```
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
...
```

- Defined as:

```
{$tsfs p/toro q/hoon r/hoon}
```

Procedural Feel in a Functional Language

- Well-styled Hoon reads like a sequence of actions
- Use inverse runes => `?.` Instead of `?:`
- Style => “soft syntax”

The Subject

- Only one operand, the “Subject”
- Only one result, the “Product”
- No “environment” or “scope”. There is only ~~Zuul~~ the Subject.

Digression - Language Tradeoffs

Language Tradeoffs

- Syntax => “What does you have to type out?”
- Semantics => “What do the things mean?”
- Easier syntax means harder semantics

Static vs. Dynamic Types

- In JavaScript:

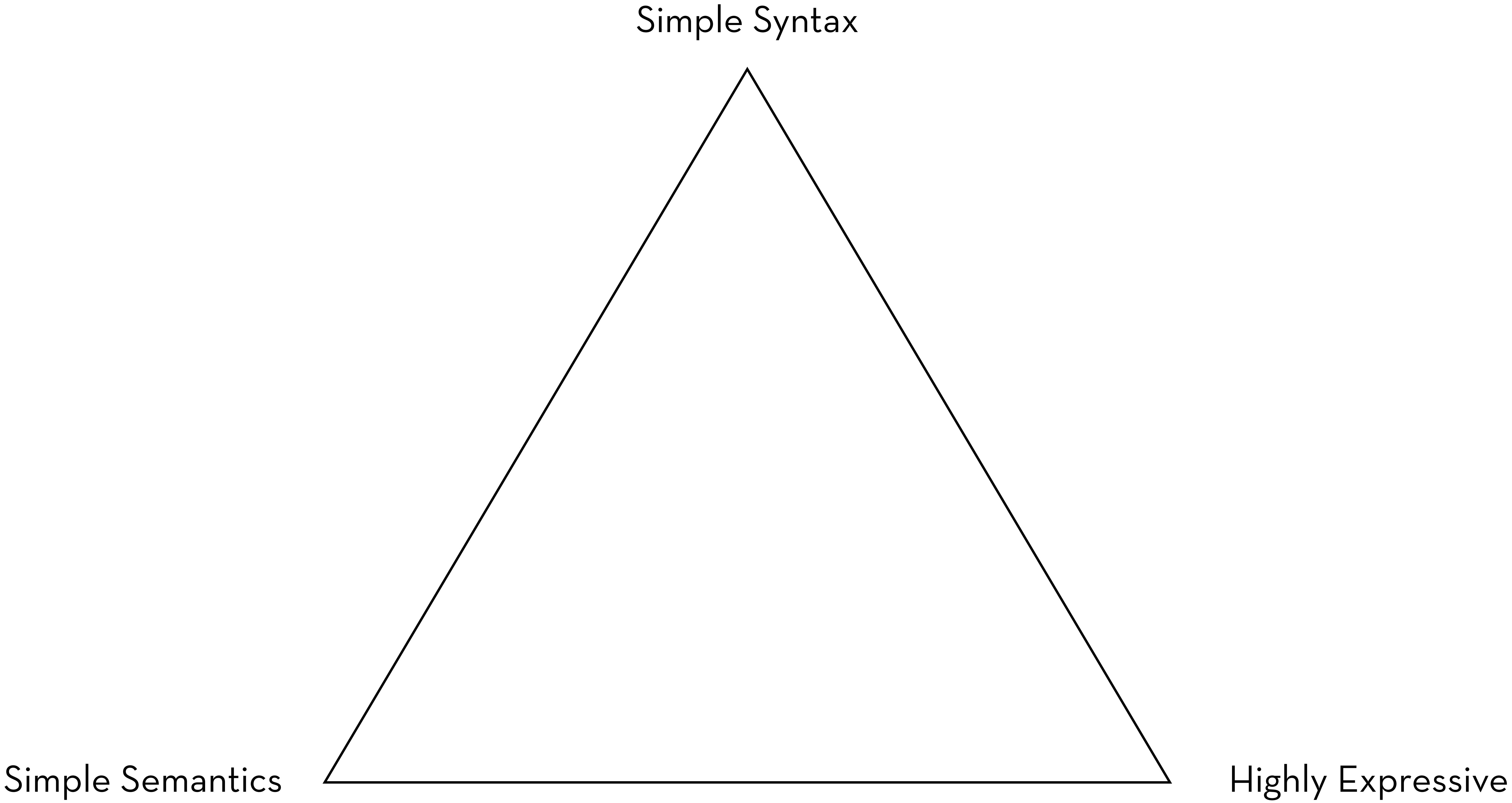
```
thing = 23
thing = "changed my mind"
thing = 12.3
```

- So, what is “`thing`” anyway?

Expressiveness

- “What concepts does the language make possible?”
- “How easy is it to put these ideas into code?”

LeBlanc’s Triangle



Types

Molds

- “type” => set of values
- “function” => domain -> do stuff -> range
- Define types via a function

Atoms

Atoms

- Can be represented as an atom:
  - Signed ints
  - Unsigned ints
  - Floats
  - Strings
  - IP Addresses
  - Bitcoin wallet
  - Animated gifs
  - The genome for the naked molerat
  - A cracked copy of Duke Nukem II

Atoms

- Specialize to the right:
  - @ => Any Atom
  - @t => UTF-8 text (a `cord`)
  - @ta => ASCII text
  - @tas => ASCII text symbol (lower-case, digits, hyphens)
- @ is similar to Scala’s general superclass `Any`

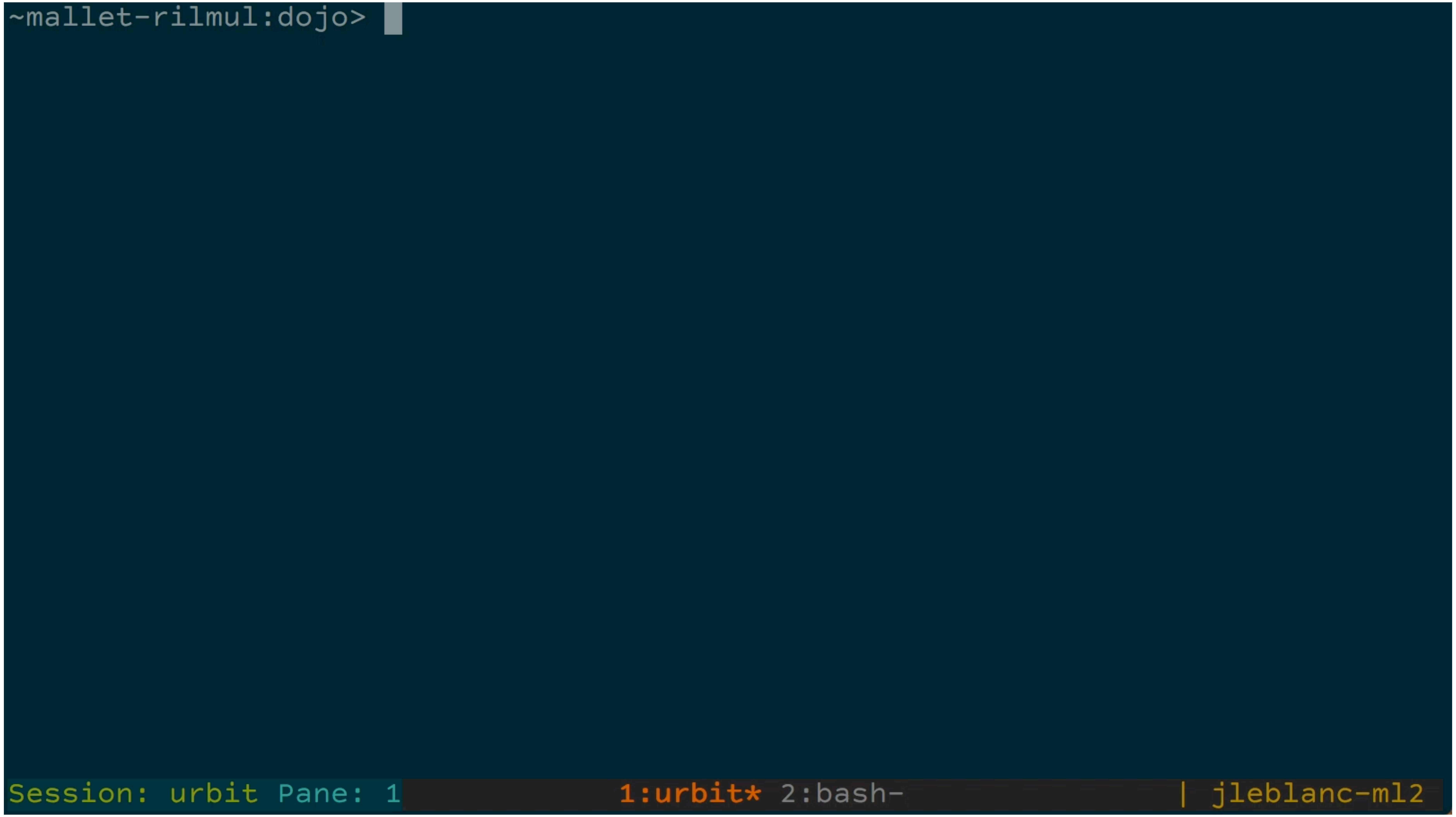
Atoms

@c	UTF-32	~-foobar	
@da	128-bit absolute date	~2016.4.23..20.09.26..f27b..dead..beef..babe	
		~2016.4.23	
@dr	128-bit relative date	~s17	(17 seconds)
		~m20	(20 minutes)
		~d42	(42 days)
@f	loobean	&	(0, yes)
		|	(1, no)
@p		~zod	(0)
@rs	32-bit IEEE float	.3.14	(pi)
		.-3.14	(negative pi)
@sd	signed decimal	--2	(2)
		-5	(-5)
@ub	unsigned binary	0b10	(2)
@uc	bitcoin address	0c1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa	
@ud	unsigned decimal	42	(42)
		1.420	(1420)
@ux	unsigned hexadecimal	0xcafe.babe	

Atoms

- @p is the Urbit phonemic base
- ~leb => 145
- ~samtul => 1066

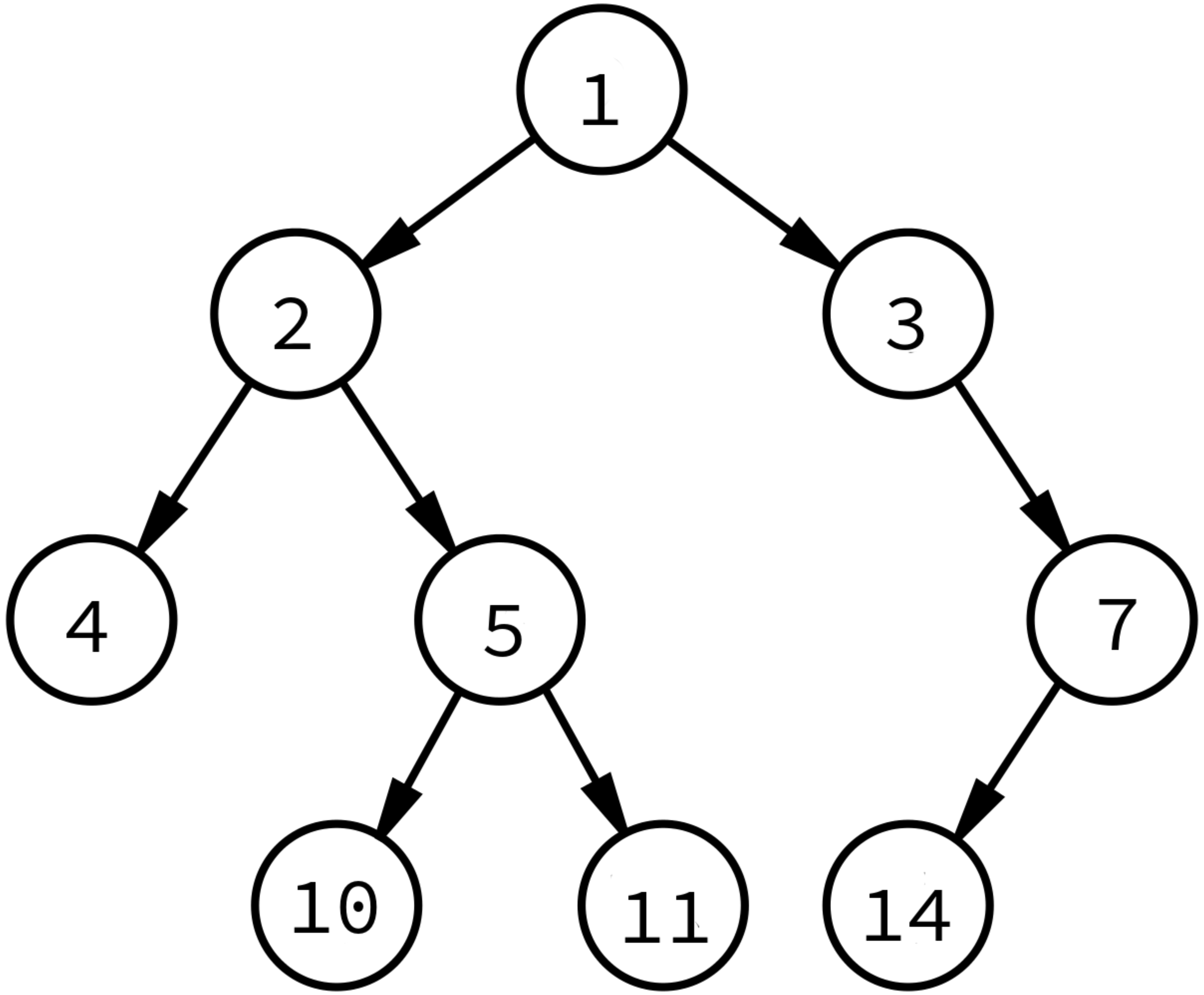
Demo



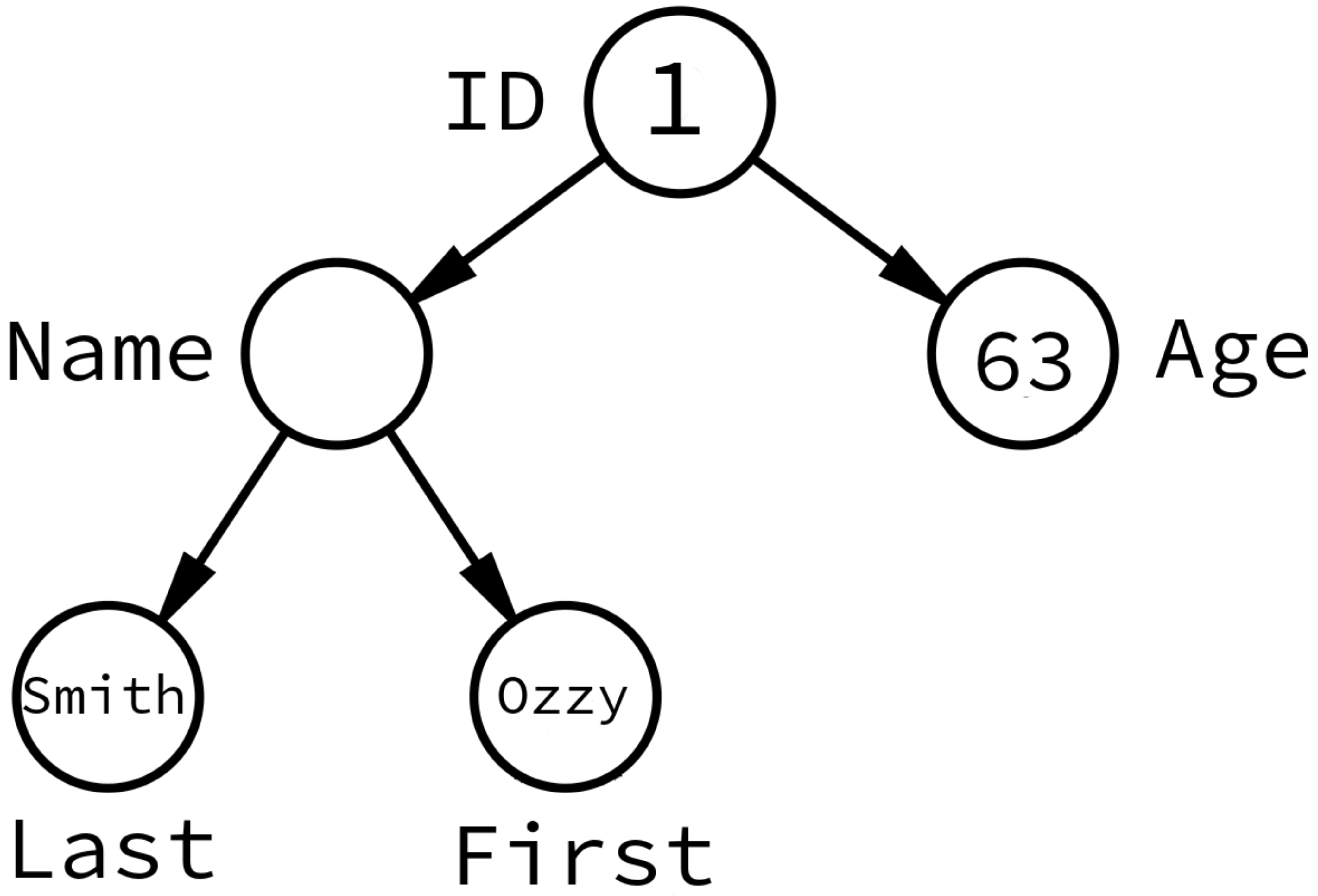
```
~mallet-rilmul:dojo>
```

```
Session: urbit Pane: 1            1:urbit* 2:bash-                  | jleblanc-ml2
```

Trees – Slots



Trees — Faces



Union Types

- Hoon supports union types
- Arbitrary subtypes
- Different defaults for atoms or cells

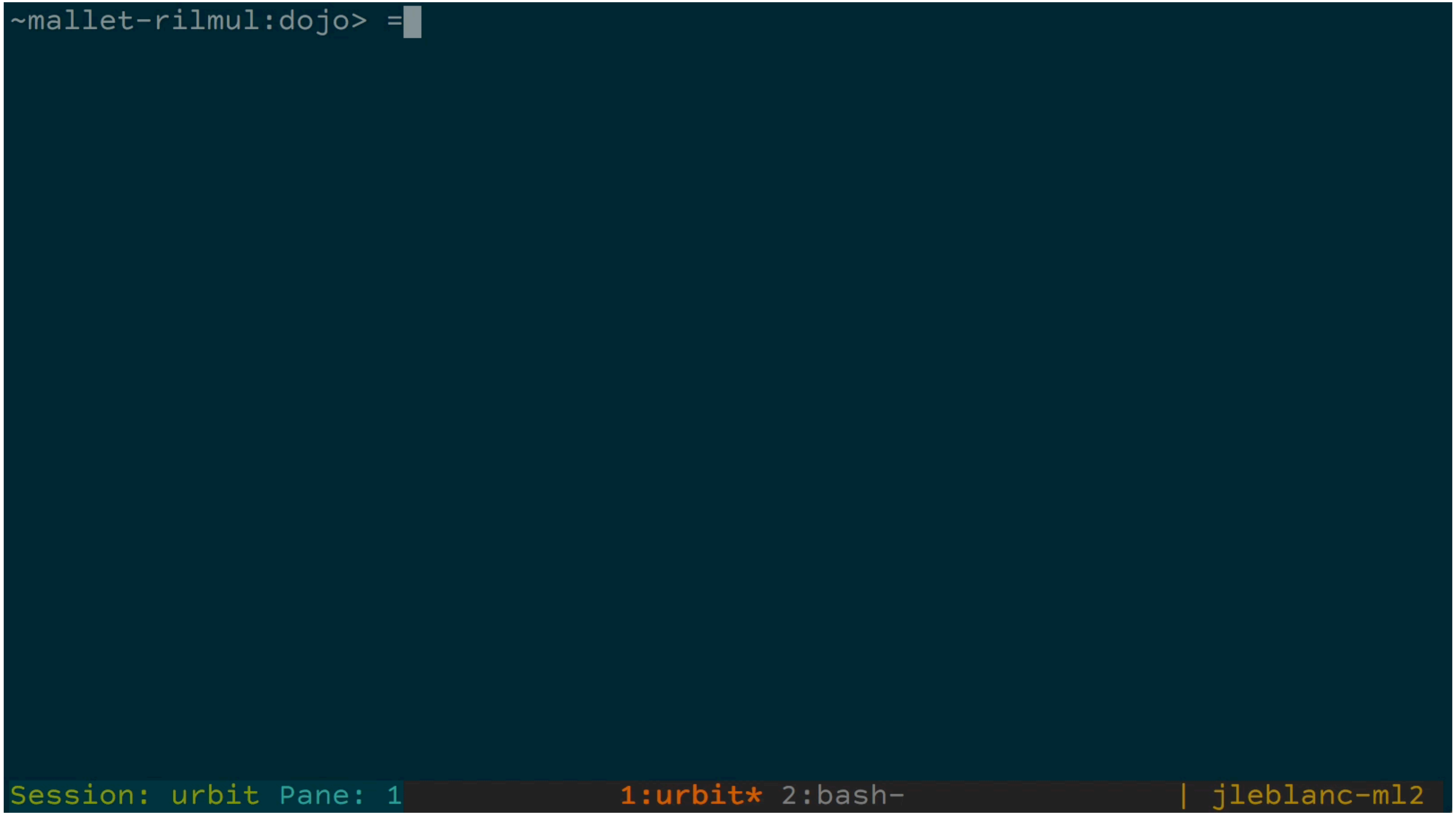
`unit`

- Hoon's version of `Maybe` or `Option` or `Possibly`
- Two values:
  - `$~` => `Nothing` or `None`
  - `[~ a]` => `Just a` or `Some a`
- Also provides `bind` and `just`

Typechecking

- A type a set of values
- Type checking => Does this type “nest” within some other type’s set of values?
- `unit => $~ | [~ a]`
- ~ “nests” within `unit`

Demo



```
~mallet-rilmul:dojo> =









Session: urbit Pane: 1             1:urbit* 2:bash-                   | jleblanc-ml2
```

Cores

- Cores => cell with code and data
- Arms => compiled attributes in cores
- Gates => core with one nameless arm
- Cores ≈ objects in most OO languages

Mint

- Types only matter in functions
- Types in Hoon are inferred by the compiler
- Types are not declared!

Mint

- Only forward type inference => compiler infers the return value
- No backwards inference
- Some types must be annotated

Mint

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

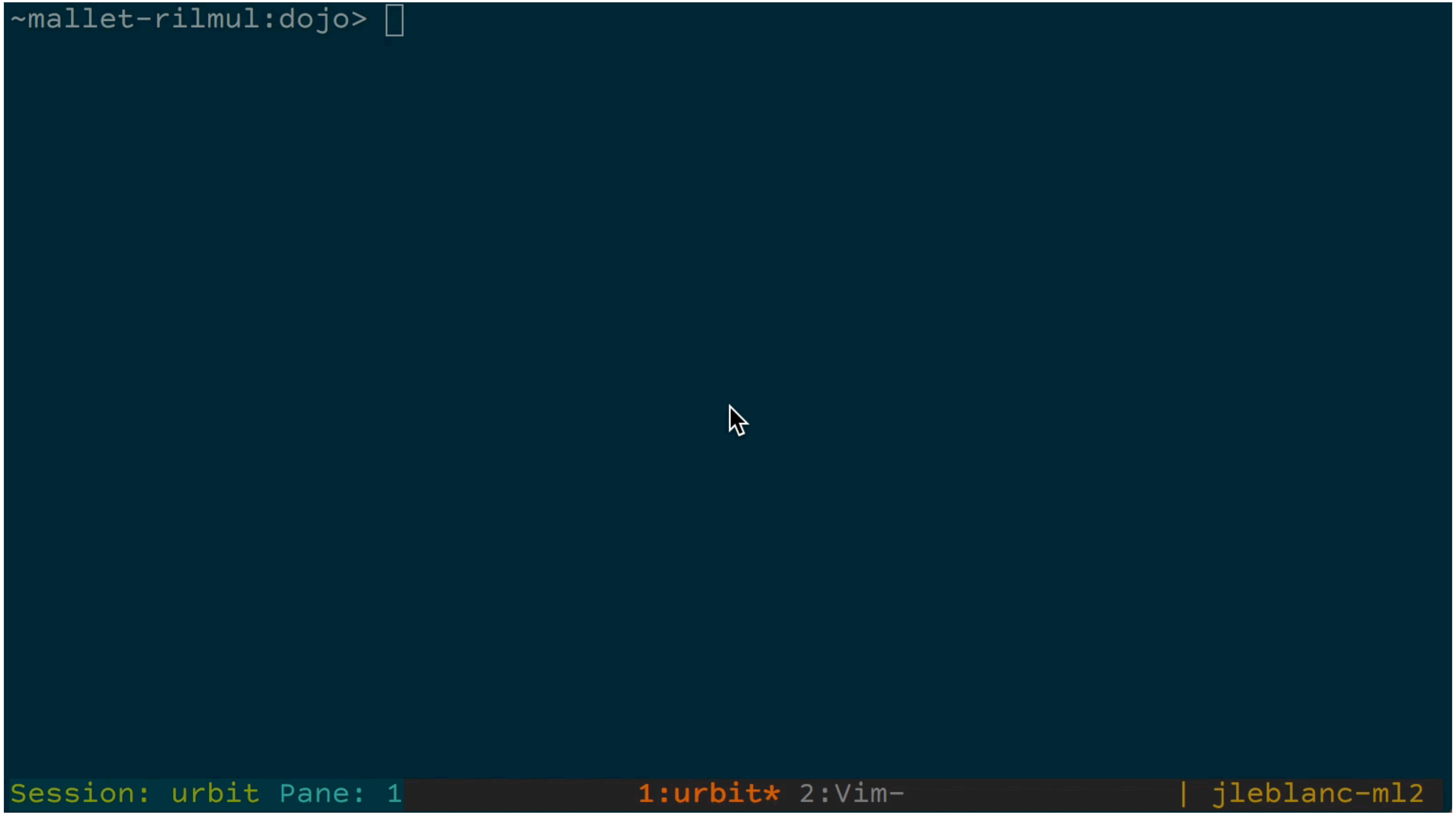
Mint

```
|=  n/@ud                       :: 1
=/  a  0                        :: 2
=/  b  1                        :: 3
|-                              :: 4
^-  (list @ud)                  :: 5
:-                              :: 6
  a                             :: 7
?:  =(0 n)                      :: 8
  ~                             :: 9
$(n (dec n), a b, b (add a b))  :: 10
```

Mint

1. Document intentions
2. Localized type error
3. Avoid potential bugs

Demo



```
~mallet-rilmul:dojo> 
```

```
Session: urbit Pane: 1          1:urbit* 2:Vim-          | jleblanc-ml2
```

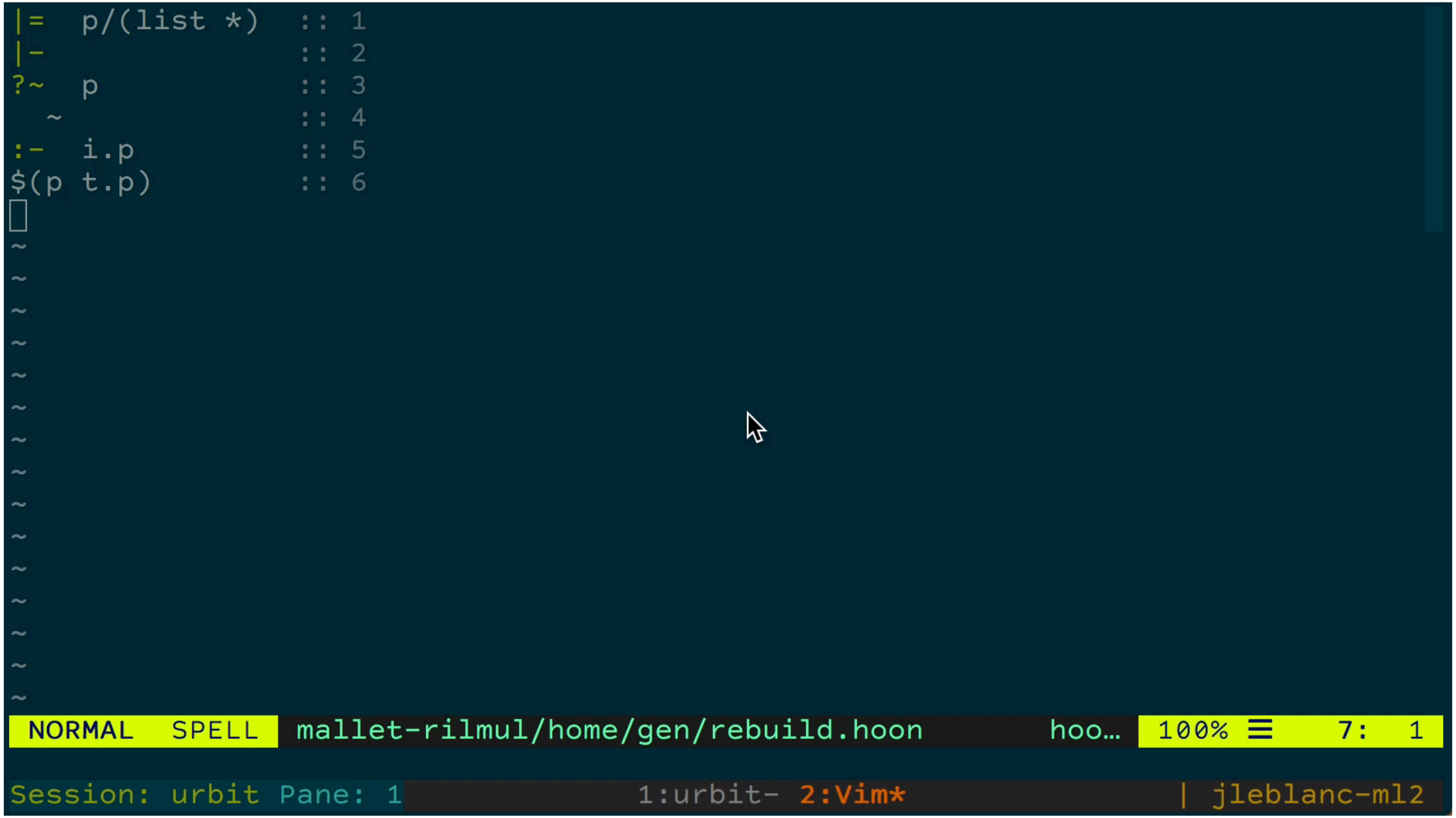
Polymorphism

- “Dry Polymorphism” == “Variance” == Liskov Substitution Principle
- “Wet Polymorphism” == “Genericity”
- Defer type inference until we use the function
- The function acts something like a macro

Polymorphism

```
|=  p/(list @)  :: 1
|-              :: 2
?~  p           :: 3
  ~             :: 4
:-  i.p         :: 5
$(p t.p)        :: 6
```

Demo



```
|=  p/(list *)  :: 1
|-              :: 2
?~  p           :: 3
  ~             :: 4
:-  i.p         :: 5
$(p t.p)        :: 6
```

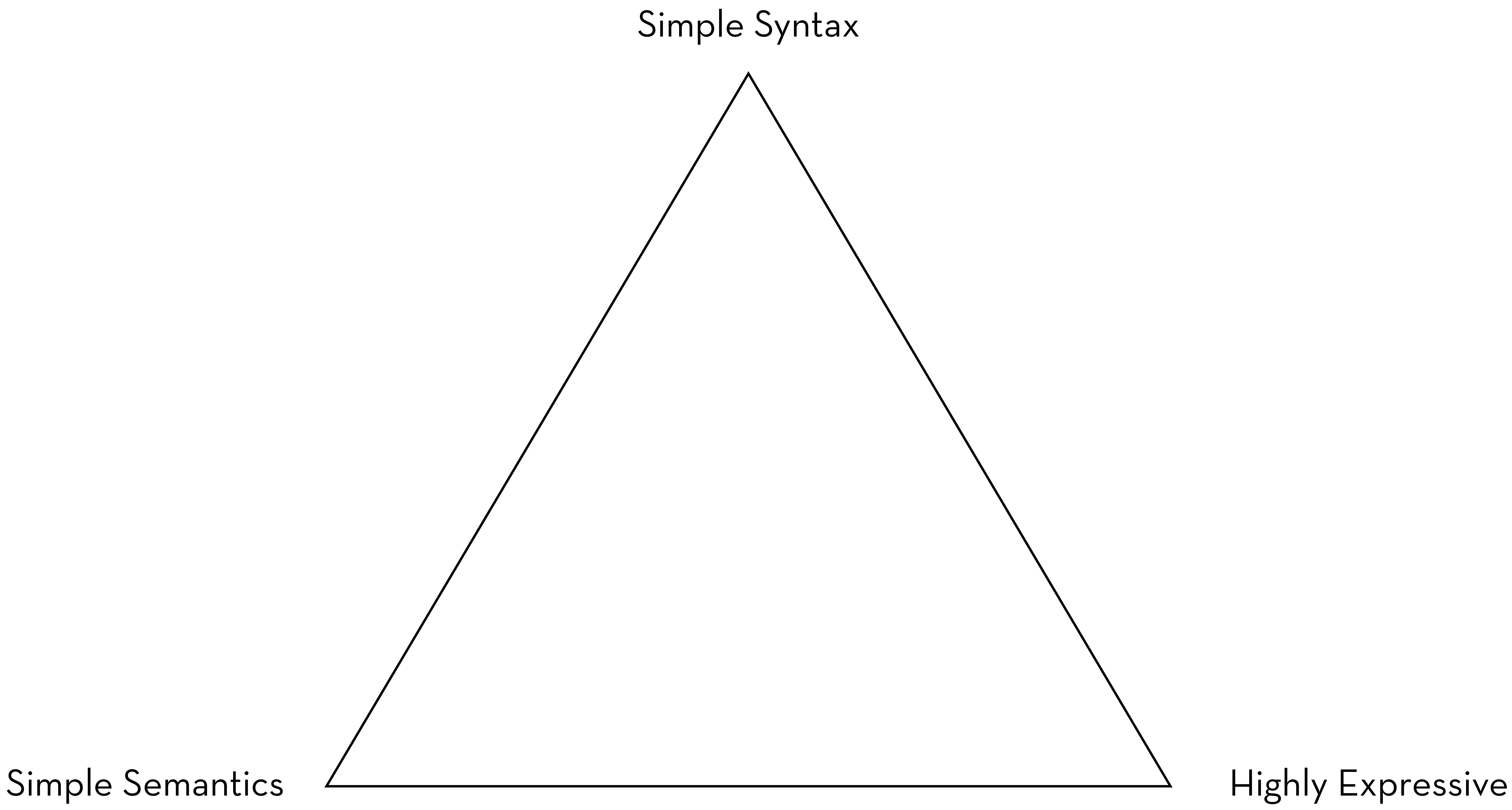
```
NORMAL  SPELL  mallet-rilmul/home/gen/rebuild.hoon      hoo…  100% ☰   7:  1

Session: urbit Pane: 1          1:urbit- 2:Vim*             | jleblanc-ml2
```

That’s All… For Now

Conclusions

- Simplicity
- ASCII Pronunciation
- Kelvin Versioning

LeBlanc’s Triangle



Conclusions

“Urbit is cool and you should check it out.”

Acknowledgments

- Ted, Mark, everyone at Tlon
- Josh Reagan
- Worldwide Technology - Asynchrony Labs
    - *Now hiring in St. Louis and Denver!*

Contact Info

leblanc@wustl.edu

@famousj

~ribben-donnyl

https://urbit.threadless.com/

One More Thing…