# Computing with `eval`

An exploration of Rebol

Brad Neimann

February 28, 2024

# Overview

1. Intro to Rebol
2. Rebol as a programming paradigm
3. The legacy of Rebol

# Intro to Rebol

(live demo)

# Basic syntax

```
>> 1 + 1
== 2

>> print "Hello World"
Hello World

>> ask "What is your name?  "
What is your name?  Brad
== "Brad"
```

# More basic syntax

```
>> name: ask "What is your name?  "
What is your name?  Brad
== "Brad"

>> name
== "Brad"

>> first name
== #"B"

>> pick name 2
== #"r"
```

# Control flow

```
>> foreach char name [probe char]
#"B"
#"r"
#"a"
#"d"
== #"d"

>> shifted: ""
== ""

>> foreach char name [append shifted (char + 1)]
== "Csbe"

>> shifted
== "Csbe"
```

# Function definition

```
>> shift: function [word n] [
[    shifted: copy ""
[    foreach char word [append shifted (char + n)]
[    return shifted
[    ]
== func [word n /local shifted char][
    shifted: copy ""
    foreach char word [append shifted ...

>> shift shifted -1
== "Brad"
```

# Non-obvious points

```
>> decode: function [word n] [shift word (negate n)]
== func [word n][shift word (negate n)]

>> decode: function [word n] [shift word negate n]
== func [word n][shift word negate n]

>> encode: :shift
== func [word n /local shifted char offset][
    shifted: copy ""
    foreach char word [append shifted ...
```

# Conditionals

```
>> shift: function [word n] [
[    shifted: copy ""
[    foreach char word [
[        either (char >= #"a")
[            [offset: #"a"] [offset: #"A"]
[        append shifted to-char
[            ((to-integer char) + n - offset
[            // 26 + offset)
[    ]
[    return shifted
[    ]
== func [word n /local shifted char offset] ...

>> shift "abcABC" -1
== "zabZAB"
```

# Blocks and other datatypes

```
>> my-block: [1 2.3 "foo" #"x"
[    28/2/2024 22:30 %/home/bradrn/Documents
[    100% AUD$20 192.168.1.1 some-word]
== [1 2.3 "foo" #"x" ...

>> index? my-block
== 1

>> next my-block
== [2.3 "foo" #"x" 28-Feb-2024 ...

>> index? next my-block
== 2
```

# Blocks as zippers

```
>> forall my-block [probe my-block]
[1 2.3 "foo" #"x" 28-Feb-2024 22:30:00 ...
[2.3 "foo" #"x" 28-Feb-2024 22:30:00 ...
["foo" #"x" 28-Feb-2024 22:30:00 ...
[#"x" 28-Feb-2024 22:30:00 %/home/bradrn/Documents ...
[28-Feb-2024 22:30:00 %/home/bradrn/Documents 100% ...
[22:30:00 %/home/bradrn/Documents 100% AUD$20.00 ...
[%/home/bradrn/Documents 100% AUD$20.00 ...
[100% AUD$20.00 192.168.1.1 some-word]
[AUD$20.00 192.168.1.1 some-word]
[192.168.1.1 some-word]
[some-word]
== [some-word]
```

# Manipulating blocks

```
>> forall my-block [
[    if #"x" == first my-block [print index? my-block]
[  ]
4
== none

>> my-find: function [item block] [
[    forall block [
[      if item == first block [return index? block]
[    ]
[  ]
== func [item block] ...

>> my-find #"x" my-block
== 4
```

# More datatypes

```
>> values: [a: 1 + 2]
== [a: 1 + 2]

>> first values
== a:

>> type? first values
== set-word!

>> third values
== +

>> type? third values
== word!
```

# Dialecting: `view`

```
>> view [text "Hello FP-Syd!"]
>> view [button "Click me" [print "Clicked!"]]
>> view [
[    message: text "not clicked"
[      italic red font-color white
[    button "Click me" [
[      message/text: "Clicked!"
[      message/font/color: black
[      message/color: white
[    ]
[    ]
```

# Dialecting: more `view`

```
>> view [
[    list: text-list data ["Alice" "Bob"] on-change [
[      current-value/text:
[        copy pick list/data list/selected
[    ]
[    below
[    current-value: field "<unselected>"
[    button "Add"
[      [append list/data copy current-value/text]
[    button "Change" [
[      poke list/data list/selected
[        copy current-value/text
[    ]
[    ]
```

# Dialecting: parse

```
>> digit: charset [#"0" - #"9"]
>> parse-rules: [
[    copy number some digit (number: load number)
[    any [
[        ["+" (op: :add) | "-" (op: :subtract)]
[        copy number2 some digit
[        (number: op number load number2)
[    ]
[    ]

>> parse "12+34" parse-rules print number
46
>> parse "12+34-47" parse-rules print number
-1
```

# Functions use blocks too!

```
>> add-2-body: [return a + b]
== [return a + b]

>> add-2: function [a b] add-2-body
== func [a b][return a + b]

>> add-2 10 20
== 30
```

# Functions use blocks too!

```
>> say-hello-body: [
[    prefix: "Hello "
[    return append prefix name
[  ]
== [prefix: "Hello " return append prefix name]

>> say-hello: function [name] say-hello-body
== func [name /local prefix] ...

>> say-hello "Brad"
== "Hello Brad"

>> say-hello-body
== [prefix: "Hello Brad" return append prefix name]
```

How does this work?
Why is it like this?

# A bit of history...

Rebol was released in 1997 by Carl Sassenrath.
(better known as the architect of AmigaOS)

> “ The Relative Expression-Based Object Language (REBOL) was designed to make it easier to communicate between computers, or between people and computers, using context-dependent sublanguages. ”
>
> — https://drdobbs.com/embedded-systems/the-rebol-scripting-language/184404172

Rebol is about exchanging and interpreting data.

# Everything is a datatype:

```
action!
binary!
bitset!
block!
char!
datatype!
date!
email!
error!
event!
file!
float!
function!
get-path!
get-word!
handle!
hash!
image!
integer!
issue!
lit-path!
lit-word!
logic!
map!
money!
native!
none!
object!
op!
pair!
paren!
path!
percent!
point!
port!
ref!
refinement!
routine!
set-path!
set-word!
string!
tag!
time!
tuple!
typeset!
unset!
url!
vector!
word!
```

# Data exchange made easy

```
>> response
== {make hash! [foo123 make hash! [item "milk" price
 AUD$5.00] bar456 make hash! [item "bread" price AUD
$4.00] baz789 make hash! [item "egg" price AUD$8.00]]}

>> transcode response
== [make hash! [foo123 make hash! [item "milk" ...

>> do transcode response
== make hash! [foo123 make hash! [item "milk" ...
```

# Macros made easy, too!

```
>> my-and: function [val1 val2] [
[    either (do val1) val2 [return false]
[    ]
== func [val1 val2][either do val1 val2 [return false]]

>> my-and [print "first" 1 = 2] [print "second" 3 = 3]
first
== false
>> my-and [print "first" 1 = 1] [print "second" 3 = 3]
first
second
== true
```

Evaluation (of parsed Rebol values)
is core to Rebol.

# ...wait, how does this work?

```
>> my-and: function [val1 val2] [
[    either do val1 val2 [return false]
[    ]

>> new-value: 2
== 2

>> my-and [1 = 1] [new-value = 2]
== true
```

# ...wait, how does this work?

```
>> in-context: context [
[    new-value: 10
[    my-and: function [val1 val2] [
[      print new-value
[      either do val1 val2 [return false]
[    ]
[  ]

>> new-value: 2
== 2

>> in-context/my-and [1 = 1] [new-value = 2]
10
== true
```

Key idea: words have an ‘invisible pointer’ binding to their values.

# ‘Definitional scoping’

Revisiting an earlier example:

```
>> add-2-body: [return a + b]
== [return a + b]

>> add-2: function [a b] add-2-body
== func [a b][return a + b]

>> add-2 10 20
== 30
```

Putting it all together with dialecting...

# Simpler data transfer

Non-dialected:

```
make hash! [
    foo123 make hash! [
        item "milk"
        price AUD$5.00
    ]
    bar456 make hash! [
        item "bread"
        price AUD$4.00
    ]
    baz789 make hash! [
        item "egg" price AUD$8.00
    ]
]
```

# Simpler data transfer

Dialected:

```
[
    #foo123 "milk" AUD$5.00
    #bar456 "bread" AUD$4.00
    #baz789 "egg" AUD$8.00
]
```

# Simpler programming

```
>> digit: charset [#"0" - #"9"]
>> parse-rules: [
[    copy number some digit (number: load number)
[    any [
[        ["+" (op: :add) | "-" (op: :subtract)]
[        copy number2 some digit
[        (number: op number load number2)
[    ]
[    ]

>> parse "12+34" parse-rules print number
46
>> parse "12+34-47" parse-rules print number
-1
```

# Dialects are everywhere!

List creation:

```
>> compose ["element 1" (1 + 1) "element 3" (2 * 2)]
== ["element 1" 2 "element 3" 4]
```

Function specs:

```
>> func [
[    arg1 [string!]
[    /refinement optional-arg [word! logic!]
[    /local local-var-name
[  ] […]
```

And more...

Other languages:

# The Rebol family

- Rebol 1
- Rebol 2
- Rebol 3 Alpha
- Red
- Ren-C
- AltScript
- Boron
- Meta
- (I think this is it?)

# JSON

> “I discovered JSON. I do not claim to have invented JSON, because it already existed in nature [...]
>
> [Rebol is] all built upon a representation of data which is then executable as programs [...]”

—Douglas Crockford, https://www.youtube.com/watch?v=-C-JoyNuQJs

# R

R (1993) predates Rebol... but uses many of the same ideas!

```
> data[,
+   date_time_num := as.numeric(local_date_time_full)]
> data$date_time_num
  [1] 2.024023e+13 2.024023e+13 2.024023e+13 ...

> lm(apparent_t ~ air_temp, data = data)
[...]
(Intercept)     air_temp
    -0.8833       1.0815

> translate_sql(if (x > mean(x)) "big" else "small")
<SQL> CASE WHEN (`x` > AVG(`x`) OVER ()) THEN 'big'
      WHEN NOT (`x` > AVG(`x`) OVER ()) THEN 'small'
      END
```

```
>> quit
```