In this post, I work through one of the examples  
where We used nonstandard evaluation to write an interpreter for a simple  
machine.

**Puzzle description**

Each instruction consists of several parts: the register to modify, whether to  
increase or decrease that register’s value, the amount by which to increase or  
decrease it, and a condition. If the condition fails, skip the instruction  
without modifying the register. The registers all start at 0. The instructions  
look like this:

b inc 5 if a > 1

a inc 1 if b < 5

c dec -10 if a >= 1

c inc -20 if c == 10

[…]

You might also encounter <= (less than or equal to) or != (not equal to).  
However, the CPU doesn’t have the bandwidth to tell you what all the registers  
are named, and leaves that to you to determine.

**What is the largest value in any register** after completing the  
instructions in your puzzle input?

If I squint long enough at the register instructions, I can basically  
see R code.

# b inc 5 if a > 1

b <- if (a > 1) b + 5 else b

# a inc 1 if b < 5

a <- if (b < 5) a + 1 else a

# c dec -10 if a >= 1

c <- if (a >= 1) c - -10 else c

# c inc -20 if c == 10

c <- if (c == 10) c + -20 else c

If we can set up a way to convert the machine instructions into R code, R will  
handle the job of looking up values, modifying values and evaluating the logic  
and if statements. In other words, if we can convert register instructions  
into R code, **the problem simplifies into something like running an R script**.

And that’s a good strategy, because we have a lot of instructions to process.  
Each user receives some unique (I think) input for each problem, and my problem  
input contains 1,000 instructions.

library(magrittr)

full\_input <- "https://raw.githubusercontent.com" %>%

file.path("tjmahr", "adventofcode17", "master",

"inst", "input08.txt") %>%

readr::read\_lines()

length(full\_input)

#> [1] 1000

head(full\_input)

#> [1] "kd dec -37 if gm <= 9" "x dec -715 if kjn == 0"

#> [3] "ey inc 249 if x < 722" "n dec 970 if t > 3"

#> [5] "f dec -385 if msg > -3" "kd dec -456 if ic <= -8"

Our strategy for simulating the register machine will have the following steps:

* Parsing a register instruction
* Creating an R expression from an instruction
* Evaluating an R expression *inside* of a register machine
* Changing the evaluation rules to adapt to the quirks of this problem

**Parsing the register instructions with regular expressions**

The instructions have a very simple grammar. Here is how I would tag the first  
few lines of my problem input.

[target] [verb] [num1] if [s1] [op] [num2]

kd dec -37 if gm <= 9

x dec -715 if kjn == 0

ey inc 249 if x < 722

n dec 970 if t > 3

f dec -385 if msg > -3

We can parse these lines using regular expressions. Regular expressions are an  
incredibly powerful language for processing text using pattern-matching rules. I  
will walk through each of the regular expression patterns used to parse an  
instruction.

To match the verbs, we can use the or | operator, so  
(inc|dec) matches inc or dec. We can also match the six different  
comparison operators using | too. In the code below, I put the patterns in  
parentheses so that they will be treated as a single group.

re\_verb <- "(inc|dec)"

re\_op <- "(>|<|==|!=|>=|<=)"

A register name is just a sequence of letters. The special character \w  
matches any *word* character; that is, it matches uppercase/lowercase letters,  
digits and underscores. The (token)+ suffix matches 1 or more  
repetitions of a token. Putting these two together, \w+ will match 1 or  
more adjacent word characters. That pattern in principle could matches things  
beside register names (like numbers) but the instruction format here is so  
constrained that it’s not a problem.

# We have to double the backslashes when using them in R strings

re\_name <- "(\\w+)"

Numbers are just integers, and sometimes they are negative. A number here  
is an optional - plus some digits. The special character \d matches any  
digit from 0 to 9, so \d+ matches 1 or more successive digits. We can use  
the (token)? suffix to match an optional token. In our case,  
-?\d+ will match a sequence of digits and match leading hyphen if one is  
present.

re\_number <- "(-?\\d+)"

Each pattern in parentheses is a matching group, and the function str\_match()  
from the stringr package will return a matrix  
with a column for each matched group. I also include an extra set of  
parentheses around the condition in the if statement to also match the whole  
condition as well as its parts.

Library(Stringr)

# Combine the sub-patterns together

re <- sprintf("%s %s %s if (%s %s %s)", re\_name, re\_verb,

re\_number, re\_name, re\_op, re\_number)

re

#> [1] "(\\w+) (inc|dec) (-?\\d+) if ((\\w+) (>|<|==|!=|>=|<=) (-?\\d+))"

text <- "b inc 5 if a > 1"

stringr::str\_match(text, re)

#> [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]

#> [1,] "b inc 5 if a > 1" "b" "inc" "5" "a > 1" "a" ">" "1"

# Column 5 matches the subgroups in columns 6, 7 and 8 as a single

# group because of the extra grouping parentheses after the `if`.

We can package this step into a function that takes an instruction’s  
text and returns a list with the labelled parts of that instruction.

parse\_instruction <- function(text) {

stopifnot(length(text) == 1)

re <- "(\\w+) (inc|dec) (-?\\d+) if ((\\w+) (>|<|==|!=|>=|<=) (-?\\d+))"

text %>%

stringr::str\_match(re) %>%

as.list() %>%

setNames(c("instruction", "target", "verb", "num1",

"cond", "s1", "op", "num2"))

}

str(parse\_instruction(text))

#> List of 8

#> $ instruction: chr "b inc 5 if a > 1"

#> $ target : chr "b"

#> $ verb : chr "inc"

#> $ num1 : chr "5"

#> $ cond : chr "a > 1"

#> $ s1 : chr "a"

#> $ op : chr ">"

#> $ num2 : chr "1"

**Creating R code**

Next, we need to convert some strings into R code. We can do this with  
rlang::parse\_expr(). An  
expression captures magic words (code) allow us to manipulate or cast (evaluate)  
them.

A Nonstandard Evaluation of Use Case

library(dplyr)

library(ggplot2)

library(rlang)

# the data is bundled with an R package I wrote

# devtools::install\_github("tjmahr/fillgaze")

df <- system.file("test-gaze.csv", package = "fillgaze") %>%

readr::read\_csv() %>%

mutate(Time = Time - min(Time)) %>%

select(Time:REyeCoordY) %>%

round(3) %>%

mutate\_at(vars(Time), round, 1) %>%

mutate\_at(vars(GazeX, GazeY), round, 0)

df

#> # A tibble: 14,823 x 8

#> Time Trial GazeX GazeY LEyeCoordX LEyeCoordY REyeCoordX REyeCoordY

#> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

#> 1 0 1 1176 643 0.659 0.589 0.566 0.602

#> 2 3.5 1 -1920 -1080 -1 -1 -1 -1

#> 3 20.2 1 -1920 -1080 -1 -1 -1 -1

#> 4 36.8 1 1184 648 0.664 0.593 0.57 0.606

#> 5 40 1 1225 617 0.685 0.564 0.591 0.579

#> 6 56.7 1 -1920 -1080 -1 -1 -1 -1

#> 7 73.4 1 1188 641 0.665 0.587 0.572 0.6

#> 8 76.6 1 1204 621 0.674 0.568 0.58 0.582

#> 9 93.3 1 -1920 -1080 -1 -1 -1 -1

#> 10 110. 1 1189 665 0.666 0.609 0.572 0.622

#> # ... with 14,813 more rows

In this particular eyetracking setup, offscreen looks are coded as negative gaze coordinates, and what’s extra weird here is that every second or third point is incorrectly placed offscreen. We see that in the frequent -1920 values in GazeX. Plotting the first few *x* and *y* pixel locations shows the pattern as well.

p <- ggplot(head(df, 40)) +

aes(x = Time) +

geom\_hline(yintercept = 0, size = 2, color = "white") +

geom\_point(aes(y = GazeX, color = "GazeX")) +

geom\_point(aes(y = GazeY, color = "GazeY")) +

labs(

x = "Time (ms)",

y = "Screen location (pixels)",

color = "Variable"

)

p +

annotate(

"text", x = 50, y = -200,

label = "offscreen", color = "grey20"

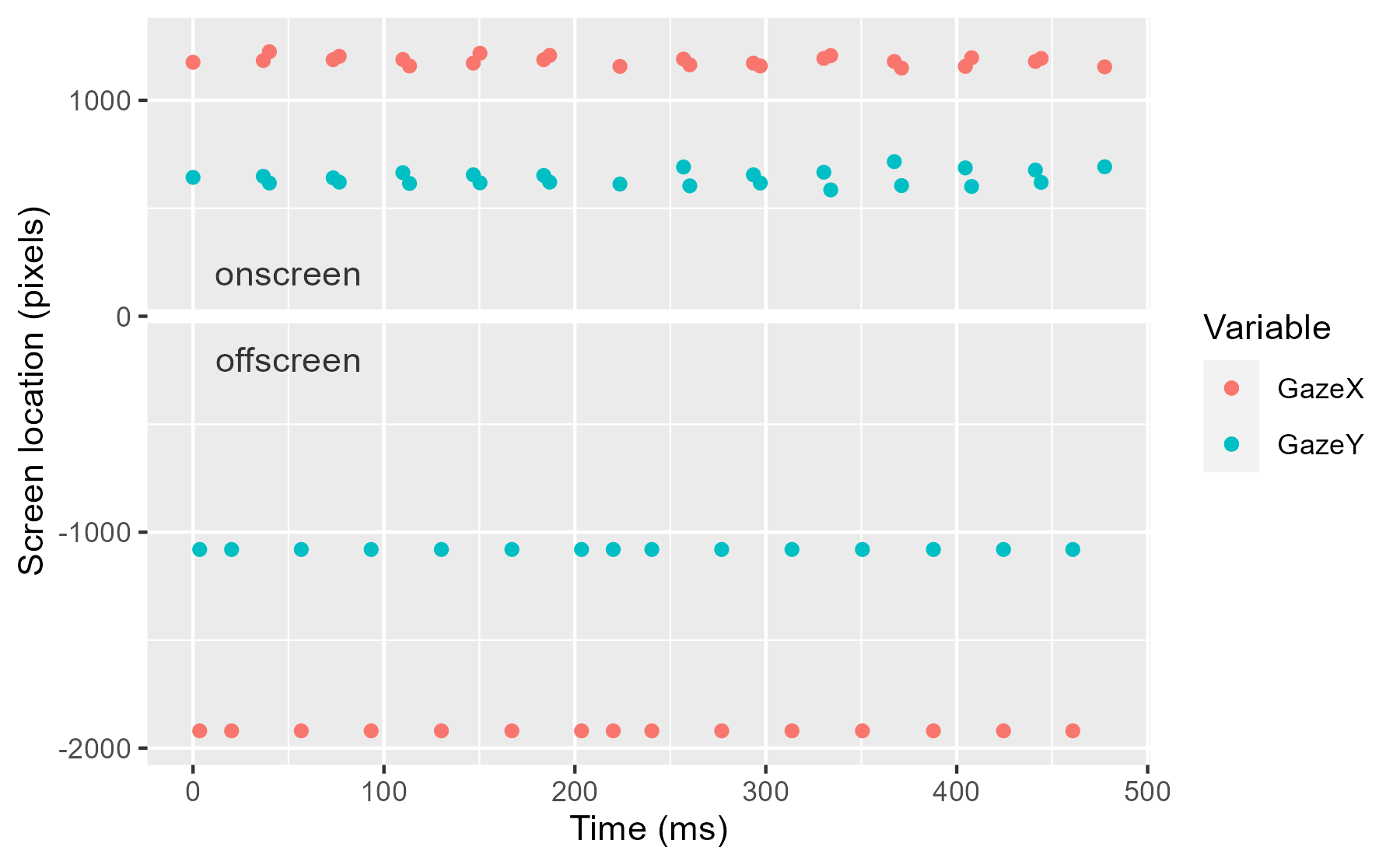
) +

annotate(

"text", x = 50, y = 200,

label = "onscreen", color = "grey20"

)



It is physiologically impossible for a person’s gaze to oscillate so quickly and with such magnitude (the gaze is tracked on a large screen display), so obviously something weird was going on with the experiment software.

This file motivated me to develop a [general purpose package for interpolating missing data in eyetracking experiments](https://github.com/tjmahr/fillgaze). This package was always something I wanted to do, and this file moved it from the *someday* list to the *today* list.

**A function to recode values in many columns as NA**[**Permalink**](https://www.tjmahr.com/set-na-where-nonstandard-evaluation-use-case/#a-function-to-recode-values-in-many-columns-as-na)

The first step in handling this problematic dataset is to convert the offscreen values into actual missing (NA) values). Because we have several columns of data, I wanted a succinct way to recode values in multiple columns into NA values.

First, we sketch out the *code we want to write* when we’re done.

set\_na\_where <- function(data, ...) {

# do things

}

set\_na\_where(

data = df,

GazeX = GazeX < -500 | 2200 < GazeX,

GazeY = GazeY < -200 | 1200 < GazeY

)

That is, after specifying the data, we list off an arbitrary number of column names, and with each name, we provide a rule to determine whether a value in that column is offscreen and should be set to NA. For example, we want every value in GazeX where GazeX < -500 or 2299 < GazeX is TRUE to be replaced with NA.

**Bottling up magic spells**[**Permalink**](https://www.tjmahr.com/set-na-where-nonstandard-evaluation-use-case/#bottling-up-magic-spells)

Lines of computer code are magic spells: We say the incantations and things happen around us. Put more formally, the code contains expressions that are evaluated in an environment.

hey <- "Hello!"

message(hey)

#> Hello!

exists("x")

#> [1] FALSE

x <- pi ^ 2

exists("x")

#> [1] TRUE

print(x)

#> [1] 9.869604

stop("what are you doing?")

#> Error in eval(expr, envir, enclos): what are you doing?

In our function signature, function(data, ...), the expressions are collected in the special “dots” argument (...). In normal circumstances, we can view the contents of the dots by storing them in a list. Consider:

hello\_dots <- function(...) {

str(list(...))

}

hello\_dots(x = pi, y = 1:10, z = NA)

#> List of 3

#> $ x: num 3.14

#> $ y: int [1:10] 1 2 3 4 5 6 7 8 9 10

#> $ z: logi NA

But we not passing in regular data, but expressions that need to be evaluated in a particular location. Below the magic words are uttered and we get an error because they mention things that do not exist in the current environment.

hello\_dots(GazeX = GazeX < -500 | 2200 < GazeX)

#> Error in str(list(...)): object 'GazeX' not found

What we need to do is prevent these words from being uttered until the time and place are right. **Nonstandard evaluation is a way of bottling up magic spells and changing how or where they are cast**—sometimes we even change the magic words themselves. We bottle up or *capture* the expressions given by the user by quoting them. quo() quotes a single expression, and quos() (plural) will quote a list of expressions. Below, we capture the expressions stored in the dots :speech\_balloon: and then make sure that their names match column names in the dataframe.

set\_na\_where <- function(data, ...) {

dots <- quos(...)

stopifnot(names(dots) %in% names(data), !anyDuplicated(names(dots)))

dots

# more to come

}

spells <- set\_na\_where(

data = df,

GazeX = GazeX < -500 | 2200 < GazeX,

GazeY = GazeY < -200 | 1200 < GazeY

)

spells

#> <list\_of<quosure>>

#>

#> $GazeX

#> <quosure>

#> expr: ^GazeX < -500 | 2200 < GazeX

#> env: 0000000017704B48

#>

#> $GazeY

#> <quosure>

#> expr: ^GazeY < -200 | 1200 < GazeY

#> env: 0000000017704B48

I call these results spells because it just contains the expressions stored as data. We can interrogate these results like data. We can query the names of the stored data, and we can extract values (the quoted expressions).

names(spells)

#> [1] "GazeX" "GazeY"

spells[[1]]

#> <quosure>

#> expr: ^GazeX < -500 | 2200 < GazeX

#> env: 0000000017704B48

**Casting spells**[**Permalink**](https://www.tjmahr.com/set-na-where-nonstandard-evaluation-use-case/#casting-spells)

We can cast a spell by evaluating an expression. To keep the incantation from fizzling out, we specify that we want to evaluate the expression *inside* of the dataframe. The function eval\_tidy(expr, data) lets us do just that: evaluate an expression expr inside of some data.

# Evaluate the first expression inside of the data

xs\_to\_set\_na <- eval\_tidy(spells[[1]], data = df)

# Just the first few bc there are 10000+ values

xs\_to\_set\_na[1:20]

#> [1] FALSE TRUE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE

#> [13] FALSE FALSE TRUE FALSE FALSE TRUE TRUE FALSE

In fact, we can evaluate them all at once with by applying eval\_tidy() on each listed expression.

to\_set\_na <- lapply(spells, eval\_tidy, data = df)

str(to\_set\_na)

#> List of 2

#> $ GazeX: logi [1:14823] FALSE TRUE TRUE FALSE FALSE TRUE ...

#> $ GazeY: logi [1:14823] FALSE TRUE TRUE FALSE FALSE TRUE ...

**Finishing touches**[**Permalink**](https://www.tjmahr.com/set-na-where-nonstandard-evaluation-use-case/#finishing-touches)

Now, the rest of the function is straightforward. Evaluate each NA-rule on the named columns, and then set each row where the rule is TRUE to NA.

set\_na\_where <- function(data, ...) {

dots <- quos(...)

stopifnot(names(dots) %in% names(data), !anyDuplicated(names(dots)))

set\_to\_na <- lapply(dots, eval\_tidy, data = data)

for (col in names(set\_to\_na)) {

data[set\_to\_na[[col]], col] <- NA

}

data

}

results <- set\_na\_where(

data = df,

GazeX = GazeX < -500 | 2200 < GazeX,

GazeY = GazeY < -200 | 1200 < GazeY

)

results

#> # A tibble: 14,823 x 8

#> Time Trial GazeX GazeY LEyeCoordX LEyeCoordY REyeCoordX REyeCoordY

#> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

#> 1 0 1 1176 643 0.659 0.589 0.566 0.602

#> 2 3.5 1 NA NA -1 -1 -1 -1

#> 3 20.2 1 NA NA -1 -1 -1 -1

#> 4 36.8 1 1184 648 0.664 0.593 0.57 0.606

#> 5 40 1 1225 617 0.685 0.564 0.591 0.579

#> 6 56.7 1 NA NA -1 -1 -1 -1

#> 7 73.4 1 1188 641 0.665 0.587 0.572 0.6

#> 8 76.6 1 1204 621 0.674 0.568 0.58 0.582

#> 9 93.3 1 NA NA -1 -1 -1 -1

#> 10 110. 1 1189 665 0.666 0.609 0.572 0.622

#> # ... with 14,813 more rows

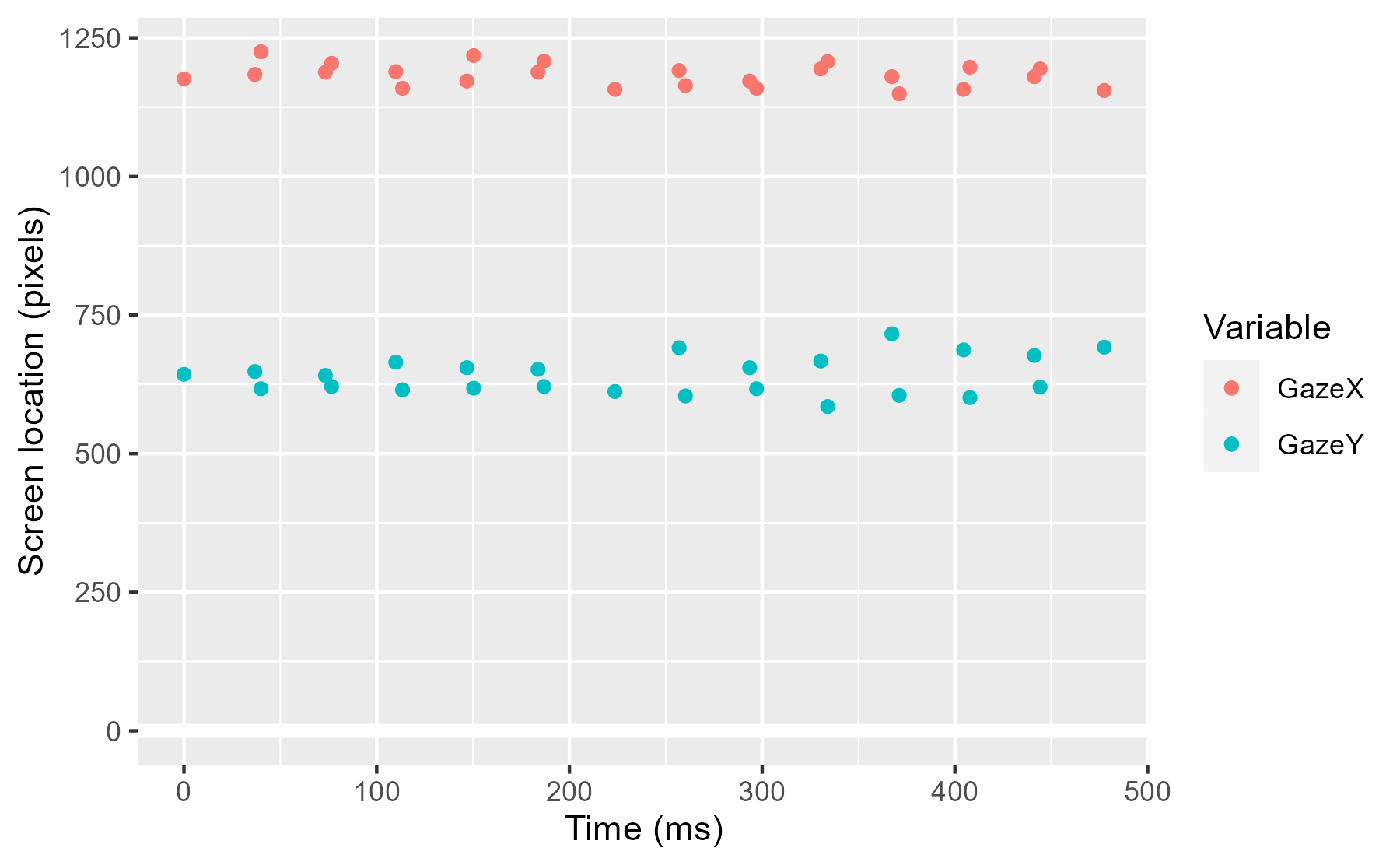
Visually, we can see that the offscreen values are no longer plotted. Plus, we are told that our data now has missing values.

# `plot %+% data`: replace the data in `plot` with `data`

p %+% head(results, 40)

#> Warning: Removed 15 rows containing missing values (geom\_point).

#> Warning: Removed 15 rows containing missing values (geom\_point).



One of the quirks about some eyetracking data is that during a blink, sometimes the device will record the *x* location but not the *y* location. (I think this happens because blinks move vertically so the horizontal detail can still be inferred in a half-closed eye.) This effect shows up in the data when there are more NA values for the *y* values than for the *x* values:

count\_na <- function(data, ...) {

subset <- select(data, ...)

lapply(subset, function(xs) sum(is.na(xs)))

}

count\_na(results, GazeX, GazeY)

#> $GazeX

#> [1] 2808

#>

#> $GazeY

#> [1] 3064

We can equalize these counts by running the function a second time with new rules.

df %>%

set\_na\_where(

GazeX = GazeX < -500 | 2200 < GazeX,

GazeY = GazeY < -200 | 1200 < GazeY

) %>%

set\_na\_where(

GazeX = is.na(GazeY),

GazeY = is.na(GazeX)

) %>%

count\_na(GazeX, GazeY)

#> $GazeX

#> [1] 3069

#>

#> $GazeY

#> [1] 3069

Alternatively, we can do this all at once by using the same NA-filtering rule on GazeX and GazeY.

df %>%

set\_na\_where(

GazeX = GazeX < -500 | 2200 < GazeX | GazeY < -200 | 1200 < GazeY,

GazeY = GazeX < -500 | 2200 < GazeX | GazeY < -200 | 1200 < GazeY

) %>%

count\_na(GazeX, GazeY)

#> $GazeX

#> [1] 3069

#>

#> $GazeY

#> [1] 3069

These last examples, where we compare different rules, showcases how nonstandard evaluation lets us write in a very succinct and convenient manner and quickly iterate over possible rules. Works like magic, indeed.

*Last knitted on 2021-11-16.*[*Source code on GitHub*](https://github.com/tjmahr/tjmahr.github.io/blob/master/_R/2017-08-15-set-na-where-nonstandard-evaluation-use-case.Rmd)*.*[1](https://www.tjmahr.com/set-na-where-nonstandard-evaluation-use-case/#fn:si)

1. sessioninfo::session\_info()
2. #> - Session info --------------------------------------------------------------
3. #> hash: carrot, flag: Niue, person in bed: medium-light skin tone
4. #>
5. #> setting value
6. #> version R version 4.1.2 (2021-11-01)
7. #> os Windows 10 x64 (build 22000)
8. #> system x86\_64, mingw32
9. #> ui RTerm
10. #> language (EN)
11. #> collate English\_United States.1252
12. #> ctype English\_United States.1252
13. #> tz America/Chicago
14. #> date 2021-11-16
15. #> pandoc NA
16. #>
17. #> - Packages -------------------------------------------------------------------
18. #> ! package \* version date (UTC) lib source
19. #> D archive 1.1.2 2021-10-25 [1] CRAN (R 4.1.1)
20. #> assertthat 0.2.1 2019-03-21 [1] CRAN (R 4.1.0)
21. #> bit 4.0.4 2020-08-04 [1] CRAN (R 4.1.0)
22. #> bit64 4.0.5 2020-08-30 [1] CRAN (R 4.1.0)
23. #> cli 3.1.0 2021-10-27 [1] CRAN (R 4.1.1)
24. #> colorspace 2.0-2 2021-06-24 [1] CRAN (R 4.1.0)
25. #> crayon 1.4.2 2021-10-29 [1] CRAN (R 4.1.1)
26. #> DBI 1.1.1 2021-01-15 [1] CRAN (R 4.1.0)
27. #> digest 0.6.28 2021-09-23 [1] CRAN (R 4.1.1)
28. #> dplyr \* 1.0.7 2021-06-18 [1] CRAN (R 4.1.0)
29. #> ellipsis 0.3.2 2021-04-29 [1] CRAN (R 4.1.0)
30. #> emo 0.0.0.9000 2021-10-14 [1] Github (hadley/emo@3f03b11)
31. #> evaluate 0.14 2019-05-28 [1] CRAN (R 4.1.0)
32. #> fansi 0.5.0 2021-05-25 [1] CRAN (R 4.1.0)
33. #> farver 2.1.0 2021-02-28 [1] CRAN (R 4.1.0)
34. #> generics 0.1.1 2021-10-25 [1] CRAN (R 4.1.1)
35. #> ggplot2 \* 3.3.5 2021-06-25 [1] CRAN (R 4.1.0)
36. #> git2r 0.28.0 2021-01-10 [1] CRAN (R 4.1.1)
37. #> glue 1.4.2 2020-08-27 [1] CRAN (R 4.1.1)
38. #> gtable 0.3.0 2019-03-25 [1] CRAN (R 4.1.0)
39. #> here 1.0.1 2020-12-13 [1] CRAN (R 4.1.0)
40. #> highr 0.9 2021-04-16 [1] CRAN (R 4.1.0)
41. #> hms 1.1.1 2021-09-26 [1] CRAN (R 4.1.1)
42. #> knitr \* 1.36 2021-09-29 [1] CRAN (R 4.1.1)
43. #> labeling 0.4.2 2020-10-20 [1] CRAN (R 4.1.0)
44. #> lifecycle 1.0.1 2021-09-24 [1] CRAN (R 4.1.1)
45. #> lubridate 1.8.0 2021-10-07 [1] CRAN (R 4.1.1)
46. #> magrittr 2.0.1 2020-11-17 [1] CRAN (R 4.1.0)
47. #> munsell 0.5.0 2018-06-12 [1] CRAN (R 4.1.0)
48. #> pillar 1.6.4 2021-10-18 [1] CRAN (R 4.1.1)
49. #> pkgconfig 2.0.3 2019-09-22 [1] CRAN (R 4.1.0)
50. #> purrr 0.3.4 2020-04-17 [1] CRAN (R 4.1.0)
51. #> R6 2.5.1 2021-08-19 [1] CRAN (R 4.1.1)
52. #> ragg 1.2.0 2021-10-30 [1] CRAN (R 4.1.1)
53. #> readr 2.0.2 2021-09-27 [1] CRAN (R 4.1.1)
54. #> rlang \* 0.4.12 2021-10-18 [1] CRAN (R 4.1.1)
55. #> rprojroot 2.0.2 2020-11-15 [1] CRAN (R 4.1.0)
56. #> rstudioapi 0.13 2020-11-12 [1] CRAN (R 4.1.0)
57. #> scales 1.1.1 2020-05-11 [1] CRAN (R 4.1.0)
58. #> sessioninfo 1.2.1 2021-11-02 [1] CRAN (R 4.1.2)
59. #> stringi 1.7.5 2021-10-04 [1] CRAN (R 4.1.1)
60. #> stringr 1.4.0 2019-02-10 [1] CRAN (R 4.1.0)
61. #> systemfonts 1.0.3 2021-10-13 [1] CRAN (R 4.1.1)
62. #> textshaping 0.3.6 2021-10-13 [1] CRAN (R 4.1.1)
63. #> tibble 3.1.5 2021-09-30 [1] CRAN (R 4.1.1)
64. #> tidyselect 1.1.1 2021-04-30 [1] CRAN (R 4.1.0)
65. #> tzdb 0.2.0 2021-10-27 [1] CRAN (R 4.1.1)
66. #> utf8 1.2.2 2021-07-24 [1] CRAN (R 4.1.0)
67. #> vctrs 0.3.8 2021-04-29 [1] CRAN (R 4.1.0)
68. #> vroom 1.5.5 2021-09-14 [1] CRAN (R 4.1.1)
69. #> withr 2.4.2 2021-04-18 [1] CRAN (R 4.1.0)
70. #> xfun 0.27 2021-10-18 [1] CRAN (R 4.1.1)
71. #>
72. #> [1] C:/Users/trist/Documents/R/win-library/4.1
73. #> [2] C:/Program Files/R/R-4.1.2/library
74. #>
75. #> D -- DLL MD5 mismatch, broken installation.
76. #>
77. #> ------------------------------------------------------------------------------

code <- rlang::parse\_expr("print('hello')")

code

#> print("hello")

code <- rlang::parse\_expr("if (a > 1) b + 5 else b")

code

#> if (a > 1) b + 5 else b

The format of the instructions is relatively straightforward. We can fill  
in a template with the parts of the parsed line. Because inc/dec are just  
addition and subtraction, we replace them with the appropriate math operations.

create\_r\_instruction <- function(parsed) {

parsed$math <- if (parsed$verb == "inc") "+" else "-"

code <- sprintf("if (%s) %s %s %s else %s", parsed$cond,

parsed$target, parsed$math, parsed$num1,

parsed$target)

rlang::parse\_expr(code)

}

r\_code <- "b inc 5 if a > 1" %>%

parse\_instruction() %>%

create\_r\_instruction()

r\_code

#> if (a > 1) b + 5 else b

**Create the register machine**

We have to figure out *where* we want to evaluate the generated R code. We  
create a register object to hold the values. The object will just be a list()  
with some extra metadata. This object will be the location where the R code  
is evaluated.

create\_register\_machine <- function(...) {

initial <- list(...)

data <- c(initial, list(.metadata = list()))

structure(data, class = c("register\_machine", "list"))

}

# Give the machines a pretty print method

print.register\_machine <- function(x, ...) {

utils::str(x, ...)

invisible(x)

}

create\_register\_machine()

#> List of 1

#> $ .metadata: list()

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

For now, we can initialize registers by using named arguments to the function.

create\_register\_machine(a = 0, b = 0)

#> List of 3

#> $ a : num 0

#> $ b : num 0

#> $ .metadata: list()

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

**Evaluating code inside of the machine**

So far, we have:

* A way to analyze register instructions and convert them into R code
* An object that holds register values

Now, we need to evaluate an expression *inside* of the register. We will use  
tidy evaluation; the function eval\_tidy() lets us evaluate an  
R expression inside of a data object.

r\_code

#> if (a > 1) b + 5 else b

# b + 5

r <- create\_register\_machine(a = 4, b = 7)

rlang::eval\_tidy(r\_code, data = r)

#> [1] 12

# just b

r <- create\_register\_machine(a = 0, b = 7)

rlang::eval\_tidy(r\_code, data = r)

#> [1] 7

Now, we need to actually do something. We need to update the register machine  
using the value from the evaluated instruction. Otherwise, the machine will just  
read expressions and forget everything it’s read.

To update the machine, we have to determine the register to update. Fortunately,  
our generated code always ends with an else branch that has the target  
register.

r\_code

#> if (a > 1) b + 5 else b

If we can pull out that symbol after the else, we will have the name of  
register to update in the machine. Because the code is so formulaic, we can just  
extract the symbol directly using the code’s abstract syntax tree (AST).  
pryr::call\_tree() shows the AST for an expression.

pryr::call\_tree(r\_code)

#> \- ()

#> \- `if

#> \- ()

#> \- `>

#> \- `a

#> \- 1

#> \- ()

#> \- `+

#> \- `b

#> \- 5

#> \- `b

We can extract elements from the tree like elements in a list by selecting  
indices.

# The numbers match one of the slashs at the first level of indentation

r\_code[[1]]

#> `if`

r\_code[[2]]

#> a > 1

# We can crawl down subtrees too

r\_code[[2]][[2]]

#> a

# But what we want is the last branch from the `if` level

r\_code[[4]]

#> b

If we convert the symbol into a string, we can look it up in the register using  
the usual list lookup syntax.

r <- create\_register\_machine(a = 4, b = 7)

target <- rlang::as\_string(r\_code[[4]])

r[[target]]

#> [1] 7

We can also use list lookup syntax with assignment to *modify* the register.

r[[target]] <- rlang::eval\_tidy(r\_code, data = r)

r

#> List of 3

#> $ a : num 4

#> $ b : num 12

#> $ .metadata: list()

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

Let’s wrap these steps into a function.

eval\_instruction <- function(register\_machine, instruction) {

target <- rlang::as\_string(instruction[[4]])

register\_machine[[target]] <- rlang::eval\_tidy(

expr = instruction,

data = register\_machine)

register\_machine

}

create\_register\_machine(a = 2, b = 0) %>%

eval\_instruction(r\_code)

#> List of 3

#> $ a : num 2

#> $ b : num 5

#> $ .metadata: list()

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

create\_register\_machine(a = 2, b = 0) %>%

# For quick testing, we pass in quoted expressions

eval\_instruction(quote(if (a > 1) b - 100 else b)) %>%

# Should not run

eval\_instruction(quote(if (a < 1) b + 5 else b)) %>%

# Should run

eval\_instruction(quote(if (a > 1) a + 10 else a))

#> List of 3

#> $ a : num 12

#> $ b : num -100

#> $ .metadata: list()

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

**Time for some *extra* nonstandard evaluation**

The code so far only works if the machine already has registers that match the  
registers in an instruction. Otherwise, we raise an error.

create\_register\_machine() %>%

eval\_instruction(quote(if (a > 1) b - 100 else b))

#> Error in overscope\_eval\_next(overscope, expr): object 'a' not found

# "Overscope" is the tidy evaluation term for the data context, so failing to

# find the name in the data is failing to find the name in the overscope.

To solve the problem, we could study the 1,000 lines of input beforehand,  
extract the register names, initialize them to 0 and then evaluate the  
instructions.[1](https://tjmahr.github.io/nonstandard-eval-register-machines/#fn:starthere) Or… or… we could procrastinate and only  
initialize a register name to 0 when the machine encounters a name it doesn’t  
recognize. If, for some reason, our machine received instructions  
one at a time, like over a network connection, then the procrastinated approach  
seems even more reasonable.

This latter strategy will involve some *very* nonstandard evaluation. I  
emphasize the “very” because **we are changing one of the fundamental rules of R  
evaluation** :smiling\_imp:. R throws an error if you ask it to evaluate the name  
of a variable that doesn’t exist. But here we are going to detect those missing  
variables and set them to 0 before they get evaluated.

To find the brand-new register names, we can inspect the call tree and find the  
names of the registers. We already know where the target is. The other place  
where names show up is in the condition of the if statement.

pryr::call\_tree(r\_code)

#> \- ()

#> \- `if

#> \- ()

#> \- `>

#> \- `a

#> \- 1

#> \- ()

#> \- `+

#> \- `b

#> \- 5

#> \- `b

extract\_register\_names <- function(instruction) {

reg\_target <- rlang::as\_string(instruction[[4]])

reg\_condition <- rlang::as\_string(instruction[[2]][[2]])

list(target = reg\_target,

registers = unique(c(reg\_target, reg\_condition))

)

}

extract\_register\_names(quote(if (a > 1) b - 100 else b)) %>% str()

#> List of 2

#> $ target : chr "b"

#> $ registers: chr [1:2] "b" "a"

# Just returns unique names

extract\_register\_names(quote(if (b > 1) b - 100 else b)) %>% str()

#> List of 2

#> $ target : chr "b"

#> $ registers: chr "b"

We can define a helper function which checks for missing names—names that  
yield NULL values when we try to retrieve them—and initializes them to 0.

initialize\_new\_registers <- function(register\_machine, registers) {

for (each\_register in registers) {

if (is.null(register\_machine[[each\_register]])) {

register\_machine[[each\_register]] <- 0

}

}

register\_machine

}

# Before

r

#> List of 3

#> $ a : num 4

#> $ b : num 12

#> $ .metadata: list()

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

initialize\_new\_registers(r, c("a", "b", "w", "a", "s", "j"))

#> List of 6

#> $ a : num 4

#> $ b : num 12

#> $ .metadata: list()

#> $ w : num 0

#> $ s : num 0

#> $ j : num 0

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

Finally, we update our evaluation function to do this step automatically. I’m  
also going to add some code to record the value of the maximum register whenever  
an instruction is evaluated because, ummm, that’s the whole point of puzzle.

eval\_instruction <- function(register\_machine, instruction) {

# Set any new registers to 0

registers <- extract\_register\_names(instruction)

register\_machine <- initialize\_new\_registers(

register\_machine, registers$registers)

# Evaluate instruction

register\_machine[[registers$target]] <- rlang::eval\_tidy(

expr = instruction,

data = register\_machine)

# Find the maximum value

register\_names <- setdiff(names(register\_machine), ".metadata")

current\_max <- max(unlist(register\_machine[register\_names]))

register\_machine$.metadata$max <- current\_max

register\_machine

}

Let’s try four instructions from a clean slate.

create\_register\_machine() %>%

# b gets 5

eval\_instruction(quote(if (d < 1) b + 5 else b)) %>%

# c gets 10

eval\_instruction(quote(if (b > 1) c + 10 else c)) %>%

# b gets 5 more

eval\_instruction(quote(if (a < 1) b + 5 else b))

#> List of 5

#> $ .metadata:List of 1

#> ..$ max: num 10

#> $ b : num 10

#> $ d : num 0

#> $ c : num 10

#> $ a : num 0

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

Now, for the moment of truth… Let’s process all 1,000 instructions.

r <- create\_register\_machine()

for (each\_instruction in full\_input) {

parsed <- each\_instruction %>%

parse\_instruction() %>%

create\_r\_instruction()

r <- eval\_instruction(r, parsed)

}

r

#> List of 27

#> $ .metadata:List of 1

#> ..$ max: num 4832

#> $ kd : num -2334

#> $ gm : num -4239

#> $ x : num -345

#> $ kjn : num -1813

#> $ ey : num 209

#> $ n : num -764

#> $ t : num 2997

#> $ f : num 4468

#> $ msg : num -3906

#> $ ic : num -263

#> $ zv : num -599

#> $ gub : num 2025

#> $ yp : num -2530

#> $ lyr : num -2065

#> $ j : num 3619

#> $ e : num -4230

#> $ riz : num 863

#> $ lzd : num 4832

#> $ ucy : num -3947

#> $ i : num 3448

#> $ omz : num -3365

#> $ djq : num 392

#> $ bxy : num 1574

#> $ tj : num 1278

#> $ y : num 1521

#> $ m : num 2571

#> - attr(\*, "class")= chr [1:2] "register\_machine" "list"

:star: Ta-da! The maximum register value is 4,832. **Problem solved!**

**And then the rules change**

Advent of Code problems come in two parts, and we don’t learn the question  
behind Part 2 until we complete Part 1. In this case, after submitting our  
solution for Part 1, we receive the following requirement:

To be safe, the CPU also needs to know **the highest value held in any  
register during this process** so that it can decide how much memory to allocate  
to these operations.

Accounting for this twist requires a small change to the evaluation code. We  
add another metadata variable to track the highest value ever stored in a  
register.

eval\_instruction <- function(register\_machine, instruction) {

# Set any new registers to 0

registers <- extract\_register\_names(instruction)

register\_machine <- initialize\_new\_registers(

register\_machine, registers$registers)

# Evaluate instruction

register\_machine[[registers$target]] <- rlang::eval\_tidy(

expr = instruction,

data = register\_machine)

# Find the maximum value

register\_names <- setdiff(names(register\_machine), ".metadata")

current\_max <- max(unlist(register\_machine[register\_names]))

register\_machine$.metadata$max <- current\_max

# Create the max-ever value if necessary

if (is.null(register\_machine$.metadata$max\_ever)) {

register\_machine$.metadata$max\_ever <- 0

}

# Maybe update the max-ever value

if (register\_machine$.metadata$max\_ever < current\_max) {

register\_machine$.metadata$max\_ever <- current\_max

}

register\_machine

}

Admittedly, eval\_instruction() is starting to get bloated. Conceptually, we  
could probably the break this function down into three functions: pre-evaluation  
steps, evaluation, and post-evaluation steps.[2](https://tjmahr.github.io/nonstandard-eval-register-machines/#fn:brainstorm) But this is good  
enough for now.

We run the instructions again to get the updated metadata.

r <- create\_register\_machine()

for (each\_instruction in full\_input) {

parsed <- each\_instruction %>%

parse\_instruction() %>%

create\_r\_instruction()

r <- eval\_instruction(r, parsed)

}

r$.metadata

#> $max

#> [1] 4832

#>

#> $max\_ever

#> [1] 5443

:star2: And boom! **Another problem solved.**

**eval(thoughts, envir = this\_problem)**

I like this kind of nonstandard evaluation approach for converting problems into  
R code, but it’s mostly useful when the problem describes a series of  
instructions that can be parsed and evaluated. For problems like this register  
machine simulation, the nonstandard evaluation route is straightforward.

Odds are, you’ll never have to write an interpreter for a toy machine or  
language. Nevertheless, here are some R functions that we used for this  
puzzle that are helpful in other contexts:

* stringr::str\_match() to extract all the groups in a regular  
  expression at once.
* rlang::parse\_expr() to convert a string of text into an R expression.
* pryr::call\_tree() to visualize an expression’s syntax tree and  
  expression[[i]][[j]] to pluck out symbols from locations in a tree.
* rlang::as\_string() to convert a symbol into a string.
* rlang::eval\_tidy() to evaluate an expression inside of a data  
  context.

1. That actually would be pretty easy. Get a dataframe with  
   purrr::map\_df(full\_input, parse\_instruction). Find the unique register  
   names. Create a list of 0’s with those names. Use do.call() to call  
   create\_register\_machine() with that list.