We recently completed all 25 days of [Advent of Code 2017](http://adventofcode.com/2017), an annual  
series of recreational programming puzzles. Each day describes a programming  
puzzle and illustrates a handful of simple examples of the problem. The puzzle  
then requires the participant to solve a much, much larger form of the problem.

For five or so of the puzzles, I used *nonstandard evaluation* to implement my  
solution. [As I previously wrote](https://tjmahr.github.io/set-na-where-nonstandard-evaluation-use-case/), nonstandard evaluation is a way of  
bottling up magic spells (lines of R code) and changing how or where they are  
cast (evaluated). I chose to use special evaluation not because it was the  
easiest or most obvious solution but because I wanted to develop my skills with  
computing on the language. 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**

[Day 8](http://adventofcode.com/2017/day/8) requires us to simulate the state  
of a register machine as it receives a series of instructions.

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](https://www.regular-expressions.info/alternation.html), 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](https://www.regular-expressions.info/repeat.html) 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](https://www.regular-expressions.info/optional.html) 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](http://stringr.tidyverse.org/) 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.

# 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(). It takes a string and creates an R expression,  
something I’ve described as a kind of [bottled up magic spell](https://tjmahr.github.io/set-na-where-nonstandard-evaluation-use-case/): An  
expression captures magic words (code) allow us to manipulate or cast (evaluate)  
them.

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](http://rlang.tidyverse.org/articles/tidy-evaluation.html); 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)](http://adv-r.had.co.nz/Expressions.html#structure-of-expressions).  
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. But  
it’s also a viable problem-solving strategy when the “machine” or the  
“instructions” are subtler, as in  
[this problem about simulating “dance” moves](http://adventofcode.com/2017/day/16).

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. With no special evaluation  
   trickery, this approach is closer to the idea of “just running R code”. [](https://tjmahr.github.io/nonstandard-eval-register-machines/#fnref:starthere)
2. If all I did for a living was write code to simulate machines  
   or toy languages, I might try to formalize this custom evaluation process  
   with pre-evaluation and post-evaluations “hooks” that could be arguments  
   to a custom evaluation function. I’m just brainstorming though. [](https://tjmahr.github.io/nonstandard-eval-register-machines/#fnref:brainstorm)