seplyr is an R package that supplies improved standard evaluation interfaces for many common data wrangling tasks.

The core of seplyr is a  
re-skinning of dplyr's  
functionality to seplyr conventions (similar to how stringr  
re-skins the implementing package stringi).

**Standard Evaluation and Non-Standard Evaluation**

"Standard evaluation" is the name we are using for the value oriented calling convention found in many programming languages. The idea is: functions are only allowed to look at the values of their arguments and not how those values arise (i.e., they can not look at source code or variable names). This evaluation principle allows one to transform, optimize, and reason about code.

It is what lets us say the following two snippets of code are equivalent.

* x <- 4; sqrt(x)
* x <- 4; sqrt(4)

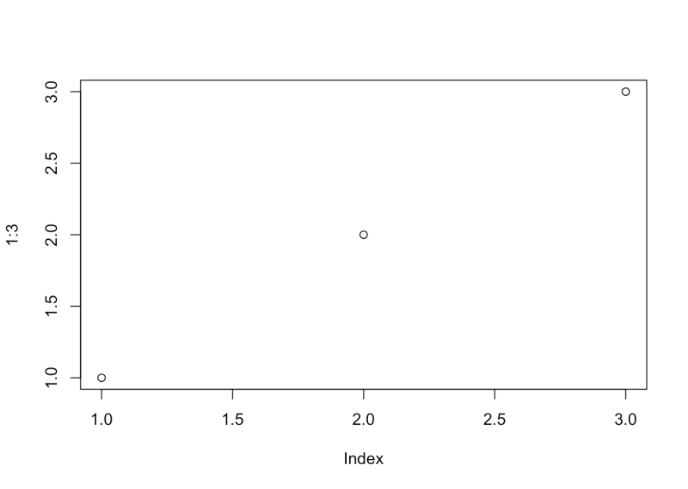
The mantra is:

"variables can be replaced with their values."

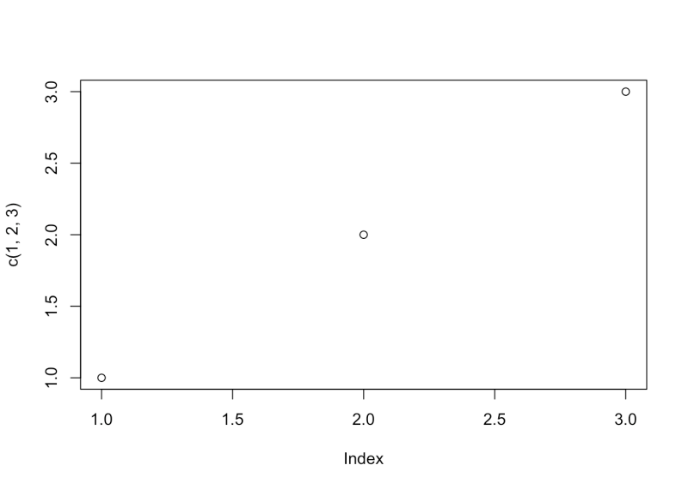
Which is called [referential transparency](https://en.wikipedia.org/wiki/Referential_transparency).

"Non-standard evaluation" is the name used for code that more aggressively inspects its environment. It is often used for harmless tasks such as conveniently setting axis labels on plots. For example, notice the following two plots have different y-axis labels (despite plotting identical values).

plot(x = 1:3)

[](http://revolution-computing.typepad.com/.a/6a010534b1db25970b01bb09df8498970d-pi)

plot(x = c(1,2,3))

[](http://revolution-computing.typepad.com/.a/6a010534b1db25970b01b8d2c6cc2a970c-pi)

**dplyr and seplyr**

The dplyr authors appear to *strongly* prefer a non-standard evaluation interface. Many in the dplyr community have come to *think* a package such as dplyr requires a non-standard interface. seplyr started as an experiment to show this is not actually the case.

Syntactically the packages are deliberately similar.

We can take a dplyr pipeline:

suppressPackageStartupMessages(library("dplyr"))

starwars %>%

select(name, height, mass) %>%

arrange(desc(height)) %>%

head()

## # A tibble: 6 x 3

## name height mass

##

## 1 Yarael Poof 264 NA

## 2 Tarfful 234 136

## 3 Lama Su 229 88

## 4 Chewbacca 228 112

## 5 Roos Tarpals 224 82

## 6 Grievous 216 159

And re-write it in seplyr notation:

library("seplyr")

starwars %.>%

select\_se(., c("name", "height", "mass")) %.>%

arrange\_se(., "desc(height)") %.>%

head(.)

## # A tibble: 6 x 3

## name height mass

##

## 1 Yarael Poof 264 NA

## 2 Tarfful 234 136

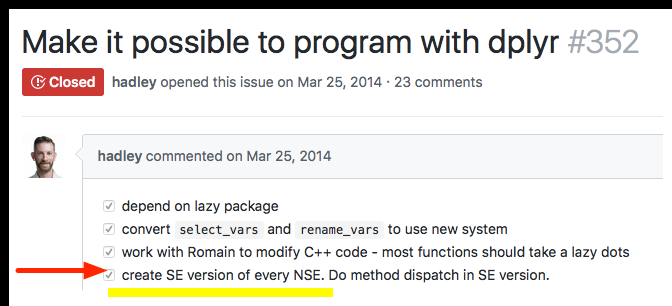
## 3 Lama Su 229 88

## 4 Chewbacca 228 112

## 5 Roos Tarpals 224 82

## 6 Grievous 216 159

We are presumably not the only ones who considered this a limitation:

[](http://revolution-computing.typepad.com/.a/6a010534b1db25970b01b7c93c77a7970b-pi)

seplyr is an attempt to make programming a primary concern by  
making the value-oriented (standard) interfaces the primary interfaces.

**mutate()**

The earlier "standard evaluation costs just a few quotes" becomes a bit strained when we talk about the dplyr::mutate() operator. It doesn't seem worth the effort unless you get something more in return. In seplyr 0.5.0 we introduced "the something more": planning over and optimizing dplyr::mutate() sequences.

A seplyr mutate looks like the following:

select\_se(., c("name", "height", "mass")) %.>%

mutate\_se(., c(

"height" := "height + 1",

"mass" := "mass + 1",

"height" := "height + 2",

"mass" := "mass + 2",

"height" := "height + 3",

"mass" := "mass + 3"

)) %.>%

arrange\_se(., "name") %.>%

head(.)

## # A tibble: 6 x 3

## name height mass

##

## 1 Ackbar 186 89

## 2 Adi Gallia 190 56

## 3 Anakin Skywalker 194 90

## 4 Arvel Crynyd NA NA

## 5 Ayla Secura 184 61

## 6 Bail Prestor Organa 197 NA

seplyr::mutate\_se() always uses ":=" to denote assignment (dplyr::mutate() prefers "=" for assignment, except in cases where ":=" is required).

The advantage is: once we are go to the trouble to capture the mutate expressions we can treat them *as data* and apply procedures to *them*. For example we can re-group and optimize the mutate assignments.

plan <- partition\_mutate\_se(

c("name" := "tolower(name)",

"height" := "height + 0.5",

"height" := "floor(height)",

"mass" := "mass + 0.5",

"mass" := "floor(mass)"))

print(plan)

## $group00001

## name height mass

## "tolower(name)" "height + 0.5" "mass + 0.5"

##

## $group00002

## height mass

## "floor(height)" "floor(mass)"

Notice seplyr::partition\_mutate\_se() re-ordered and re-grouped the assignments so that:

* In each group each value used is independent of values produced in other assignments.
* All dependencies between assignments are respected by the group order.

The "safe block" assignments can then be used in a pipeline:

starwars %.>%

select\_se(., c("name", "height", "mass")) %.>%

mutate\_seb(., plan) %.>%

arrange\_se(., "name") %.>%

head(.)

## # A tibble: 6 x 3

## name height mass

##

## 1 ackbar 180 83

## 2 adi gallia 184 50

## 3 anakin skywalker 188 84

## 4 arvel crynyd NA NA

## 5 ayla secura 178 55

## 6 bail prestor organa 191 NA

This may not seem like much. However, when using dplyr with a SQL database (such as PostgreSQL or even Sparklyr) keeping the number of dependencies in a block low is critical for correct calculation. Furthermore, on Sparklyr sequences of mutates are simulated by nesting of SQL statements, so you must also keep the number of mutates at a moderate level (i.e., you want a minimal number of blocks or groups).

**Machine Generated Code**

Because we are representing mutate assignments as user manipulable data we can also enjoy the benefit of machine generated code. seplyr 0.5.\* uses this opportunity to introduce a simple function named if\_else\_device(). This device uses R's ifelse() statement (which conditionally chooses values in a vectorized form) to implement a more powerful block-if/else statement.

For example: suppose we want to NA-out one of height or mass for each row of the starwars data uniformly at random. This can be written naturally using the if\_else\_device.

if\_else\_device(

testexpr = "runif(n())>=0.5",

thenexprs = "height" := "NA",

elseexprs = "mass" := "NA")

## ifebtest\_30etsitqqutk

## "runif(n())>=0.5"

## height

## "ifelse( ifebtest\_30etsitqqutk, NA, height)"

## mass

## "ifelse( !( ifebtest\_30etsitqqutk ), NA, mass)"

Notice the if\_else\_device translates the user code into a sequence of dplyr::mutate() expressions (using only the weaker operator ifelse()). Obviously the user could perform this translation, but if\_else\_device automates the record keeping. Also many such steps can be chained together and broken into a minimal sequence of blocks by partition\_mutate\_se() (not forcing a new dplyr::mutate() step for each if-block encountered).

When we combine the device with the partitioned we get performant database-safe code where the number of blocks is only the level of variable dependence (and not the possibly much larger number of initial value uses that a straightforward non-reordering split would give; note: seplyr::mutate\_se() 0.5.1 and later incorporate the partition\_mutate\_se() in mutate\_se()).

starwars %.>%

select\_se(., c("name", "height", "mass")) %.>%

mutate\_se(., if\_else\_device(

testexpr = "runif(n())>=0.5",

thenexprs = "height" := "NA",

elseexprs = "mass" := "NA")) %.>%

arrange\_se(., "name") %.>%

head(.)

## # A tibble: 6 x 4

## name height mass ifebtest\_wwr9k0bq4v04

##

## 1 Ackbar NA 83 TRUE

## 2 Adi Gallia 184 NA FALSE

## 3 Anakin Skywalker NA 84 TRUE

## 4 Arvel Crynyd NA NA TRUE

## 5 Ayla Secura 178 NA FALSE

## 6 Bail Prestor Organa 191 NA FALSE

# Example: clear one of a or b in any row where both are set.

d <- data.frame(a = c(0, 0, 1, 1, 1, 1, 1, 1, 1, 1),

b = c(0, 1, 0, 1, 1, 1, 1, 1, 1, 1),

edited = FALSE)

program <- if\_else\_device( # detect rows with both a and b set

testexpr = qe((a+b)>1),

thenexprs = c(

if\_else\_device( # randomly clear one of a or b

testexpr = qe(runif(dplyr::n()) >= 0.5),

thenexprs = qae(a := 0),

elseexprs = qae(b := 0)),

qae(edited := TRUE)))

print(program)

#> ifebtest\_u7fd78yvg5fi

#> "(a + b) > 1"

#> ifebtest\_wl4dieexombe

#> "runif(dplyr::n()) >= 0.5"

#> a

#> "ifelse( ifebtest\_u7fd78yvg5fi, ifelse( ifebtest\_wl4dieexombe, 0, a), a)"

#> b

#> "ifelse( ifebtest\_u7fd78yvg5fi, ifelse( !( ifebtest\_wl4dieexombe ), 0, b), b)"

#> edited

#> "ifelse( ifebtest\_u7fd78yvg5fi, TRUE, edited)"

plan <- partition\_mutate\_se(program)

print(plan)

#> $group00001

#> ifebtest\_u7fd78yvg5fi ifebtest\_wl4dieexombe

#> "(a + b) > 1" "runif(dplyr::n()) >= 0.5"

#>

#> $group00002

#> a

#> "ifelse( ifebtest\_u7fd78yvg5fi, ifelse( ifebtest\_wl4dieexombe, 0, a), a)"

#> b

#> "ifelse( ifebtest\_u7fd78yvg5fi, ifelse( !( ifebtest\_wl4dieexombe ), 0, b), b)"

#> edited

#> "ifelse( ifebtest\_u7fd78yvg5fi, TRUE, edited)"

#>

res <- d %.>%

mutate\_seb(., plan) %.>%

select\_se(., grepdf('^ifebtest\_.\*', ., invert=TRUE))

print(res)

#> a b edited

#> 1 0 0 FALSE

#> 2 0 1 FALSE

#> 3 1 0 FALSE

#> 4 0 1 TRUE

#> 5 0 1 TRUE

#> 6 1 0 TRUE

#> 7 1 0 TRUE

#> 8 1 0 TRUE

#> 9 1 0 TRUE

#> 10 0 1 TRUE

**Conclusion**

The value oriented notation is a bit clunkier, but this is offset by it's greater  
flexibility in terms of composition and working parametrically.

Our group has been using seplyr::if\_else\_device() and seplyr::partition\_mutate\_se() to greatly simplify porting powerful SAS procedures to R/Sparklyr/Apache Spark clusters.