Tidy Evaluation

Lionel Henry and Hadley Wickham | RStudio

RStudio®

Tidy evaluation

Our vision for dealing with a special class of R functions

- Usually called NSE but we prefer **quoting functions**
- Most interesting language feature of R

⟶ Refer to your data directly

```
x <- 1:32

mutate(starwars, height + x)
ggplot(starwars, aes(height, x))
lm(data = starwars, height ~ x)
```

Tidy evaluation

- Quoting works against you when you write functions
  - Easy to write the columns to which you refer
  - But hard to program them
- Tidy eval provides full programmability
  - Set of tools included in dplyr and soon tidyr, ggplot2, …
  - Challenging at first but rewarding!
  - Two important concepts: **quasiquotation** (unquoting) and **quosures**

Tidy evaluation

- What is "quoting"
- Why is **unquoting** necessary for programming
- How to use unquoting to create **wrapper functions**

Quoting

`quote()` is the most basic quoting function

```
letters[1:5]
#> [1] "a" "b" "c" "d" e"
```

Evaluate now!

```
quote(letters[1:5])
#> letters[1:5]
```

Quote expression

```
"letters[1:5]"
#> [1] "letters[1:5]"
```

Quote string

Delayed evaluation

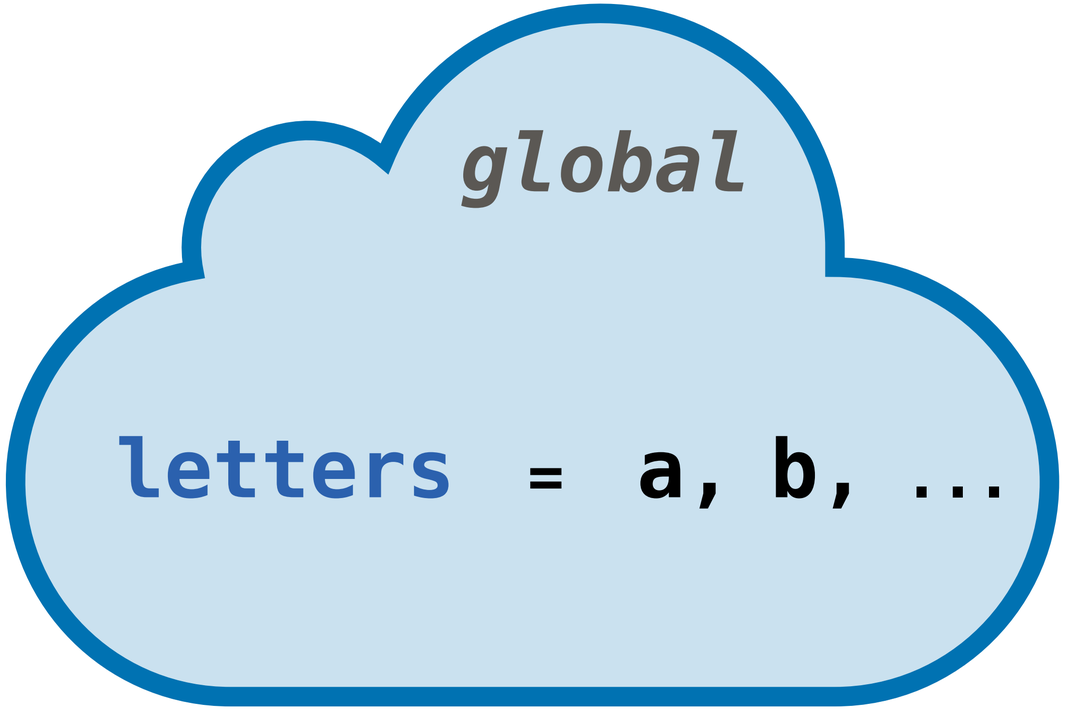
`eval()` gets the value of the expression

```
x <- quote(letters[1:5])
eval(x)
#> [1] "a" "b" "c" "d" "e"
```

Delayed evaluation

Killer feature: change the **context** of evaluation

```
x <- quote(letters[1:5])
eval(x)
#> [1] "a" "b" "c" "d" "e"
```

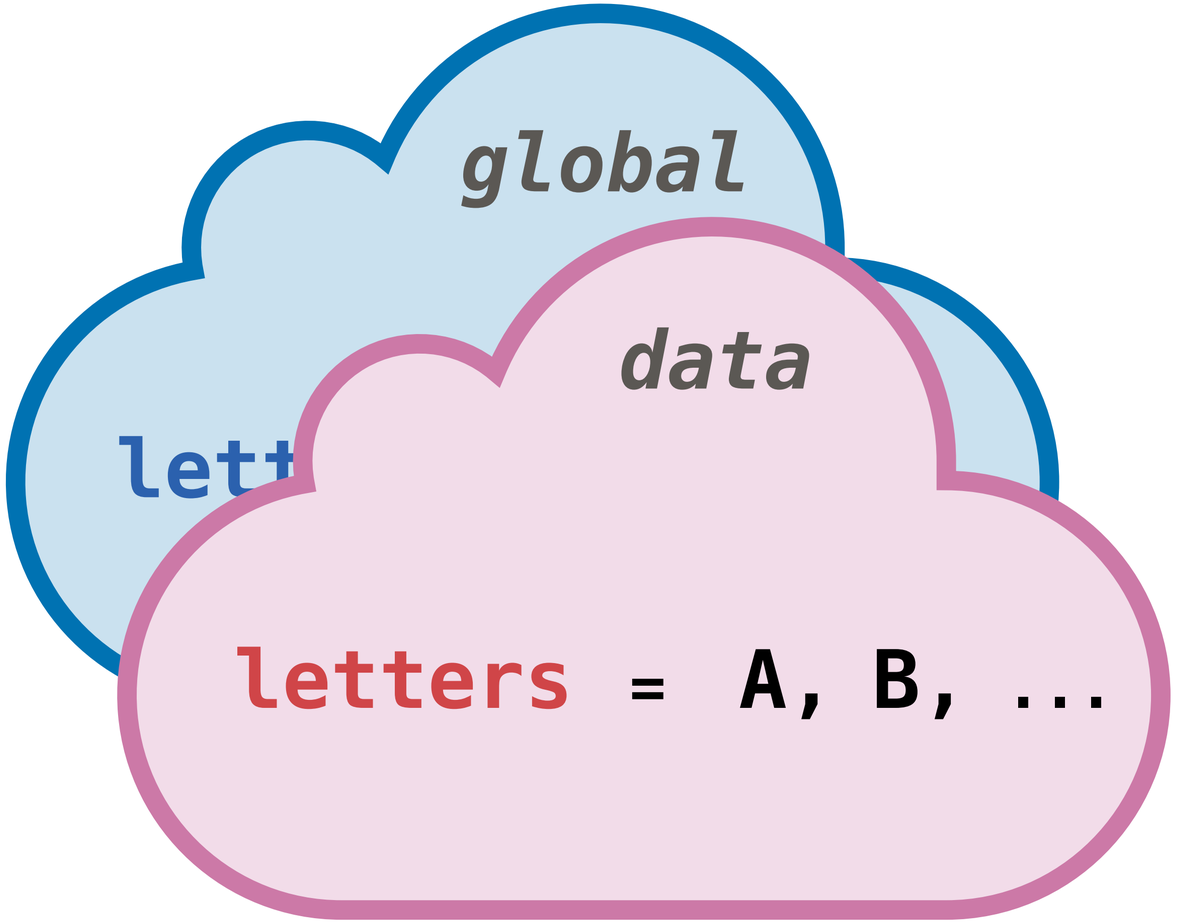


Delayed evaluation

Killer feature: change the **context** of evaluation

```
x <- quote(letters[1:5])
eval(x)
#> [1] "a" "b" "c" "d" "e"
```

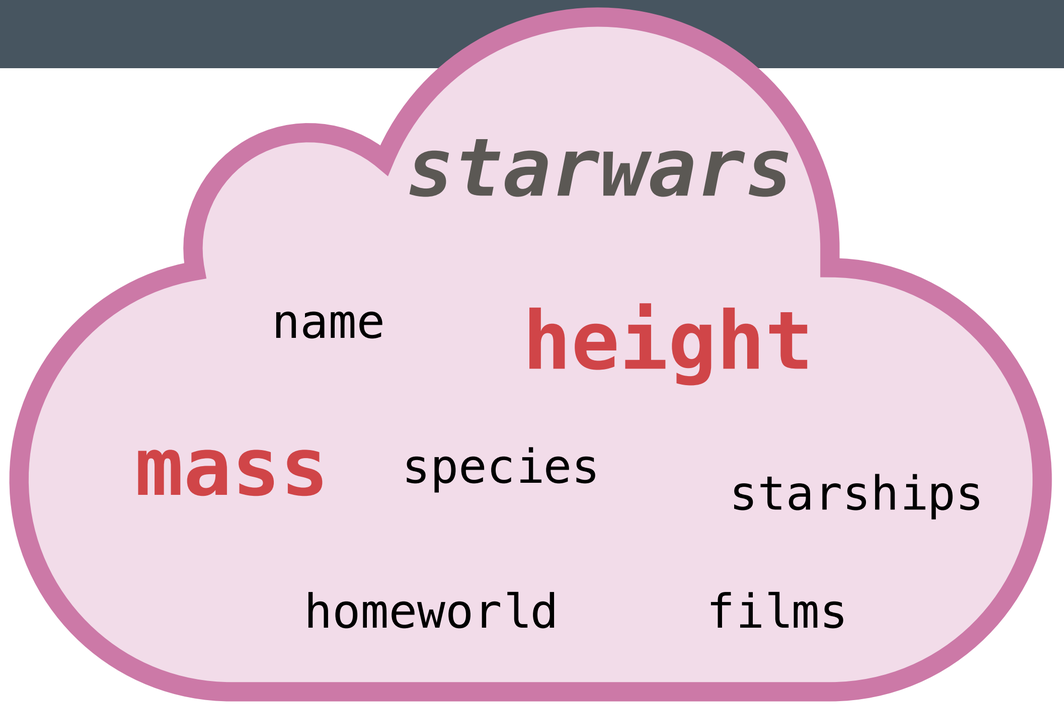
```
data <- list(letters = LETTERS)
eval(x, data)
#> [1] "A" "B" "C" "D" "E"
```



Tidyverse grammars

Arguments to tidyverse grammars are

- automatically quoted
- evaluated in a data context




```
mutate(starwars, bmi = mass / height^2)
```

⟶ *This is why grammar verbs are quoting functions*

Programming with grammars

- Programming is all about varying inputs
- How do we vary inputs of quoting functions?

Programming with grammars

```
stringly(nobody, calls, me, coward)
#> [1] "nobody calls me coward"
```

```
x <- "chicken"
stringly(nobody, calls, me, x)
#> [1] "nobody calls me x"
```

```
paste("nobody", "calls", "me", x)
#> [1] "nobody calls me chicken"
```

Programming with grammars

```
stringly(nobody, calls, me, coward)
#> [1] "nobody calls me coward"
```

```
x <- "chicken"
stringly(nobody, calls, me, x)
#> [1] "nobody calls me x"
```

```
paste("nobody", "calls", "me", x)
#> [1] "nobody calls me chicken"
```

Easy with regular function, quoting is done by the user

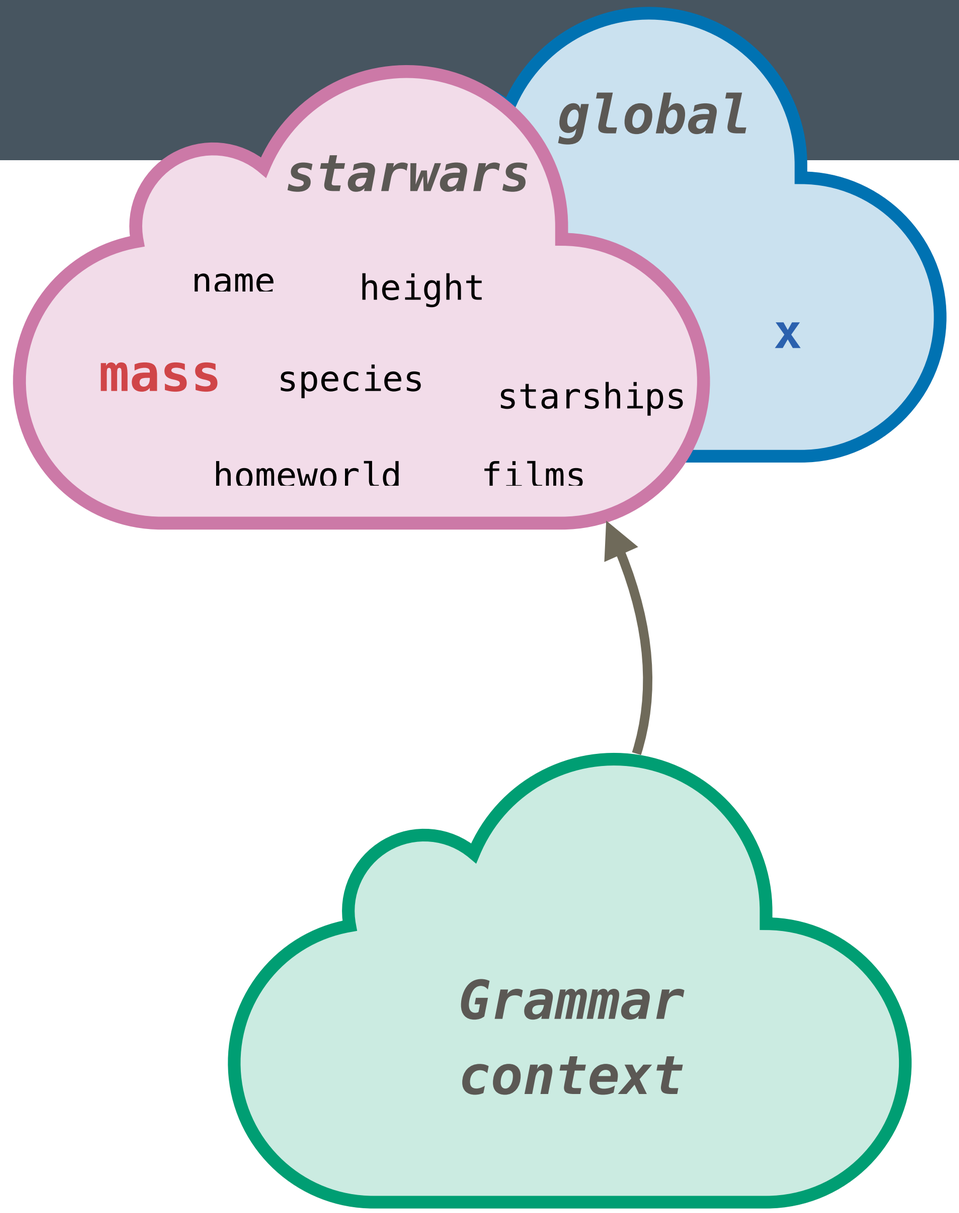
Programming with grammars

Varying inputs when quoting

```
x <- "height"
```

(quoting time)

```
mutate(starwars, bmi = mass / x^2)
```



Programming with grammars

Varying inputs when quoting

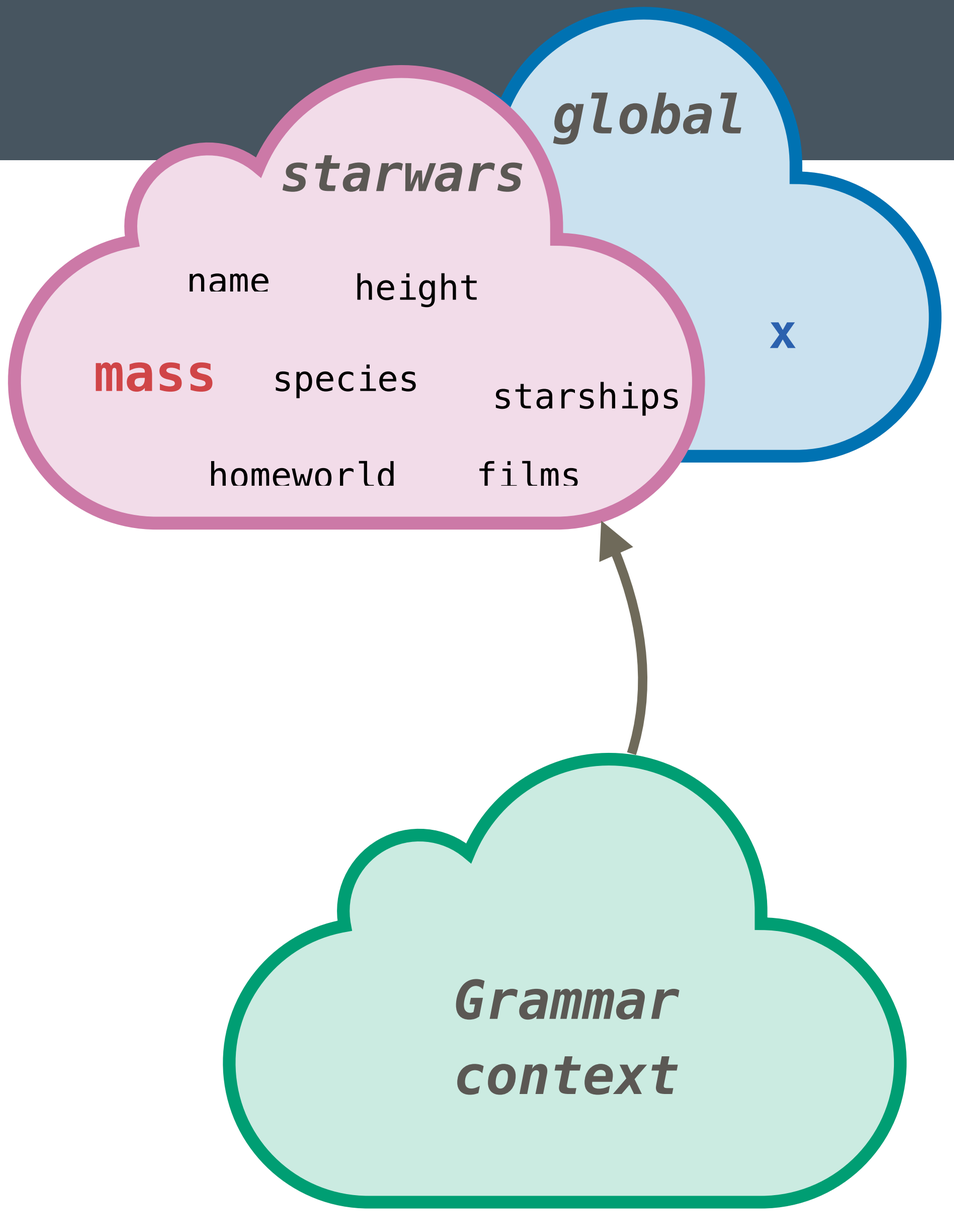
```
x <- "height"
```

(quoting time)

```
mutate(starwars, bmi = mass / x^2)
```

(evaluation time)

```
mutate(starwars, bmi = <dbl> / x^2)
```



Programming with grammars

Varying inputs when quoting

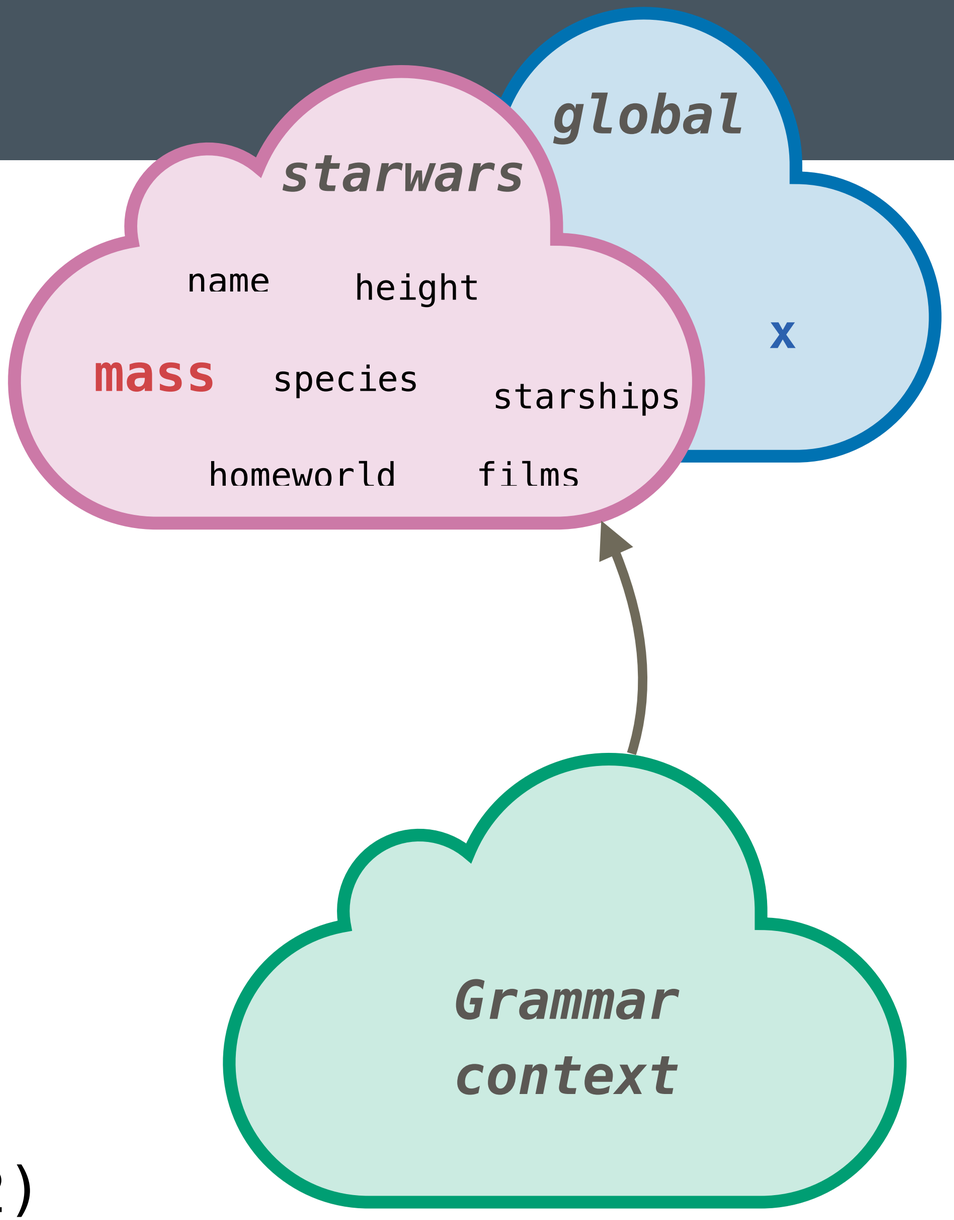
```
x <- "height"
```

(quoting time)

```
mutate(starwars, bmi = mass / x^2)
```

(evaluation time)

```
mutate(starwars, bmi = <dbl> / "height"^2)
```



Quasiquotation

Root cause is that evaluation happens in two stages

- Quoting time
- Evaluation time

We need to vary the expression dplyr sees

⟶ Let's unquote during quotation

⟶ New friendly **!!** **unquoting operator**

Quasiquotation

!! evaluates *at quoting time*

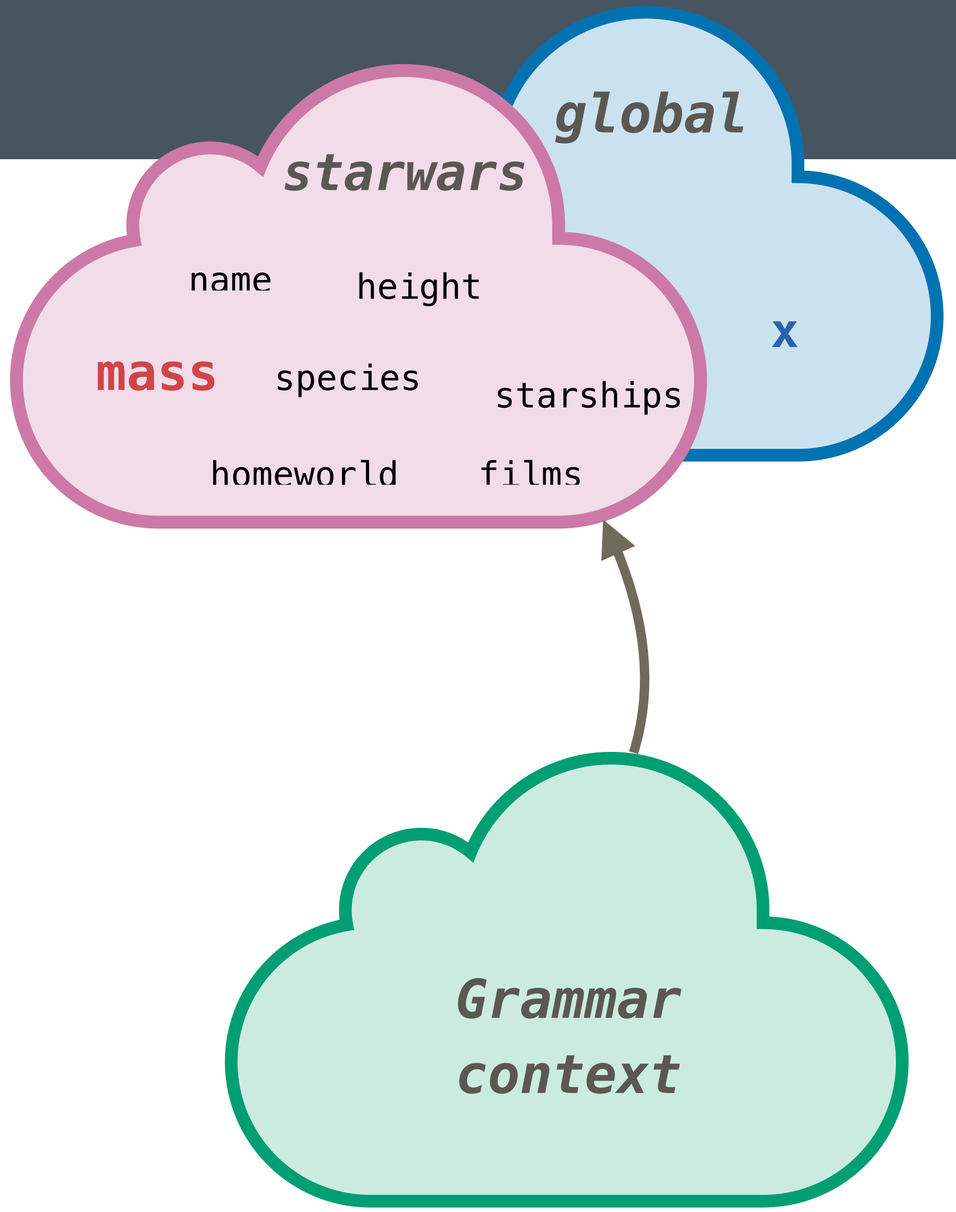
```
x <- "chicken"

stringly(nobody, calls, me, !! x)
#> [1] "nobody calls me chicken"
```

Quasiquotation

!! evaluates *at quoting time*

```
x <- "height"
mutate(starwars, bmi = mass / (!! x)^2)
```



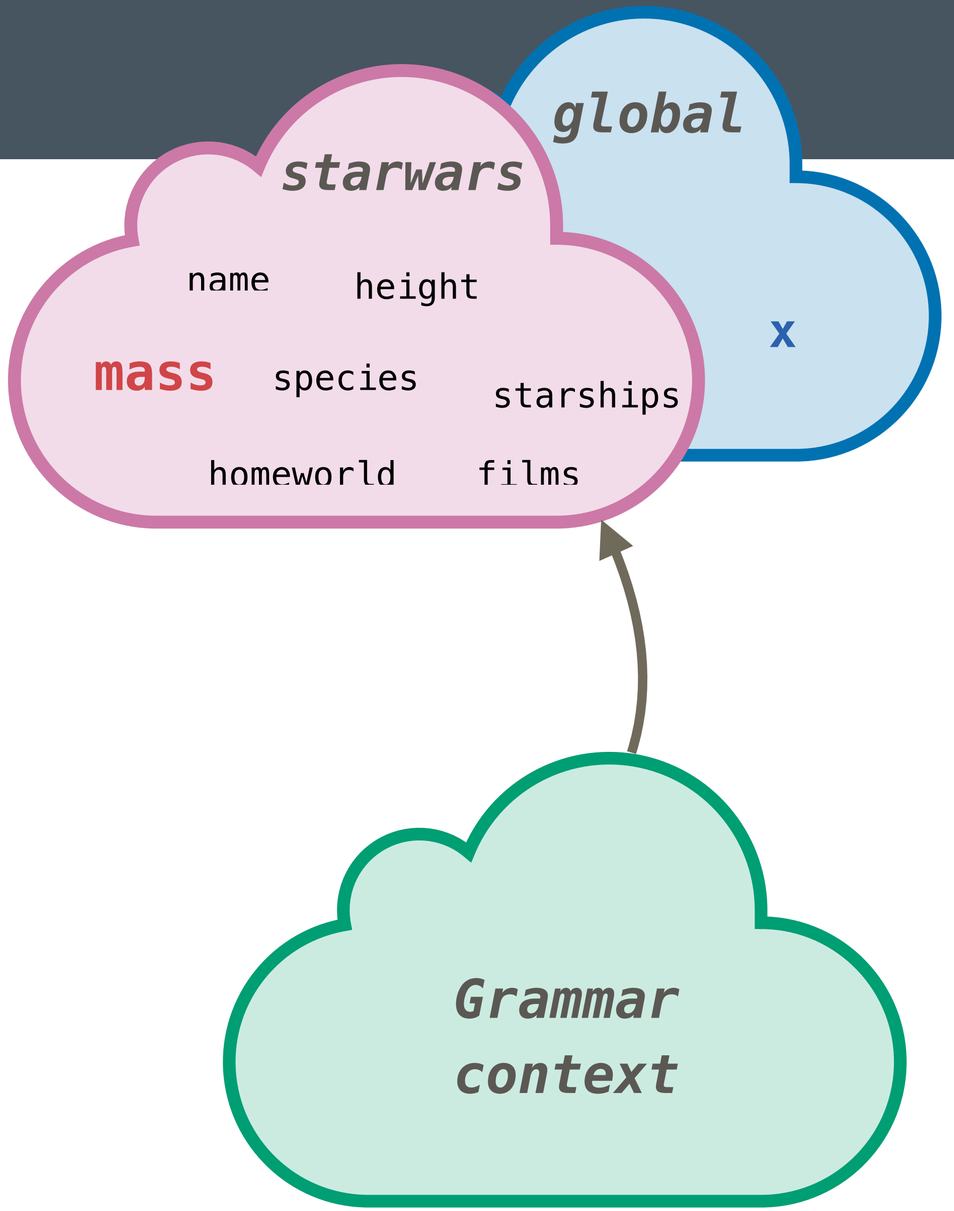
Quasiquotation

!! evaluates *at quoting time*

```
x <- "height"
mutate(starwars, bmi = mass / (!! x)^2)
```

(quoting time)

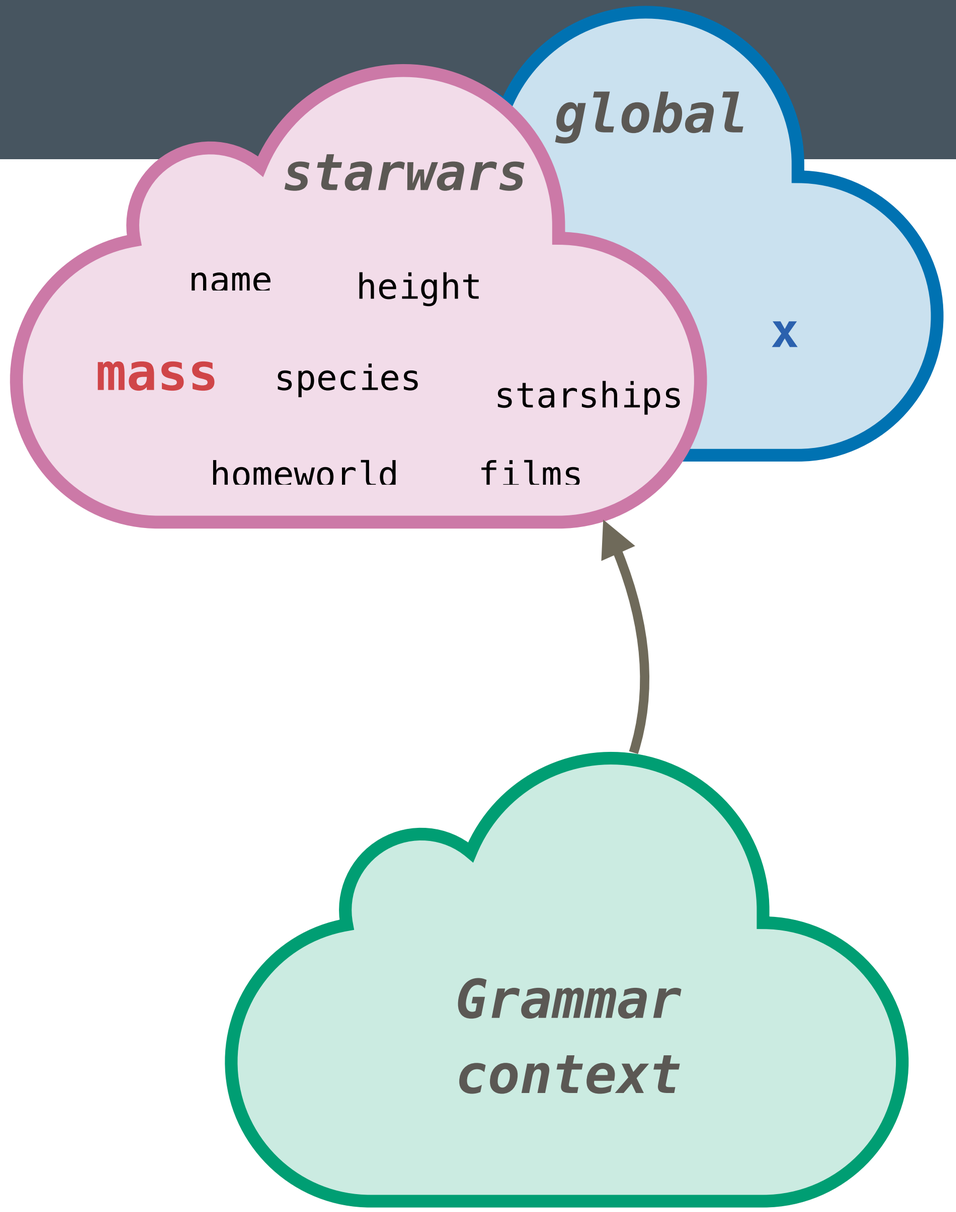
```
mutate(starwars, bmi = mass / "height"^2)
```



Quasiquotation

!! evaluates *at quoting time*

```
x <- "height"
mutate(starwars, bmi = mass / (!! x)^2)
```

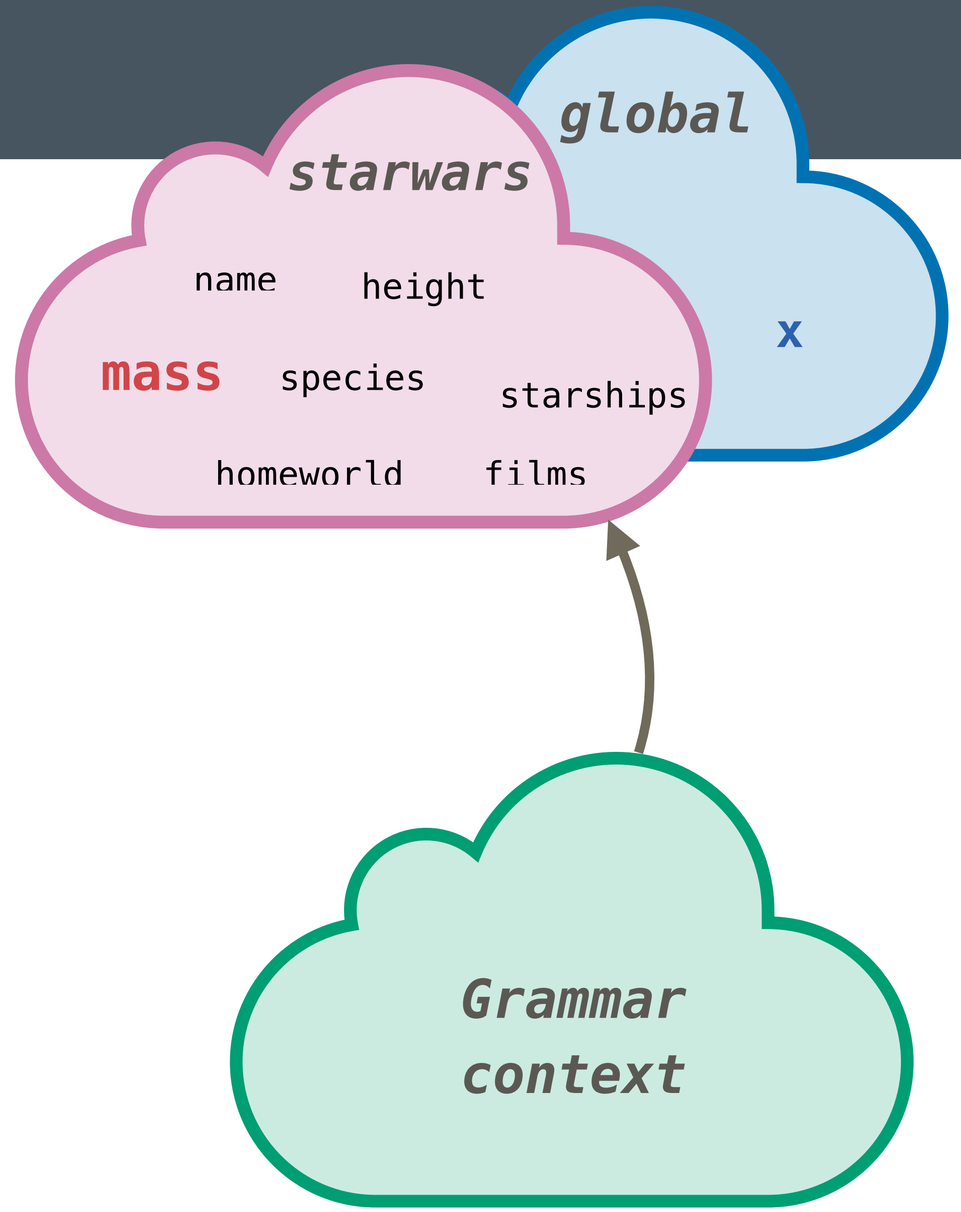


We need something that looks like code…
A symbol!

Quasiquotation

!! evaluates *at quoting time*

```
x <- sym("height")
mutate(starwars, bmi = mass / (!! x)^2)
```



We need something that looks like code…
A symbol!

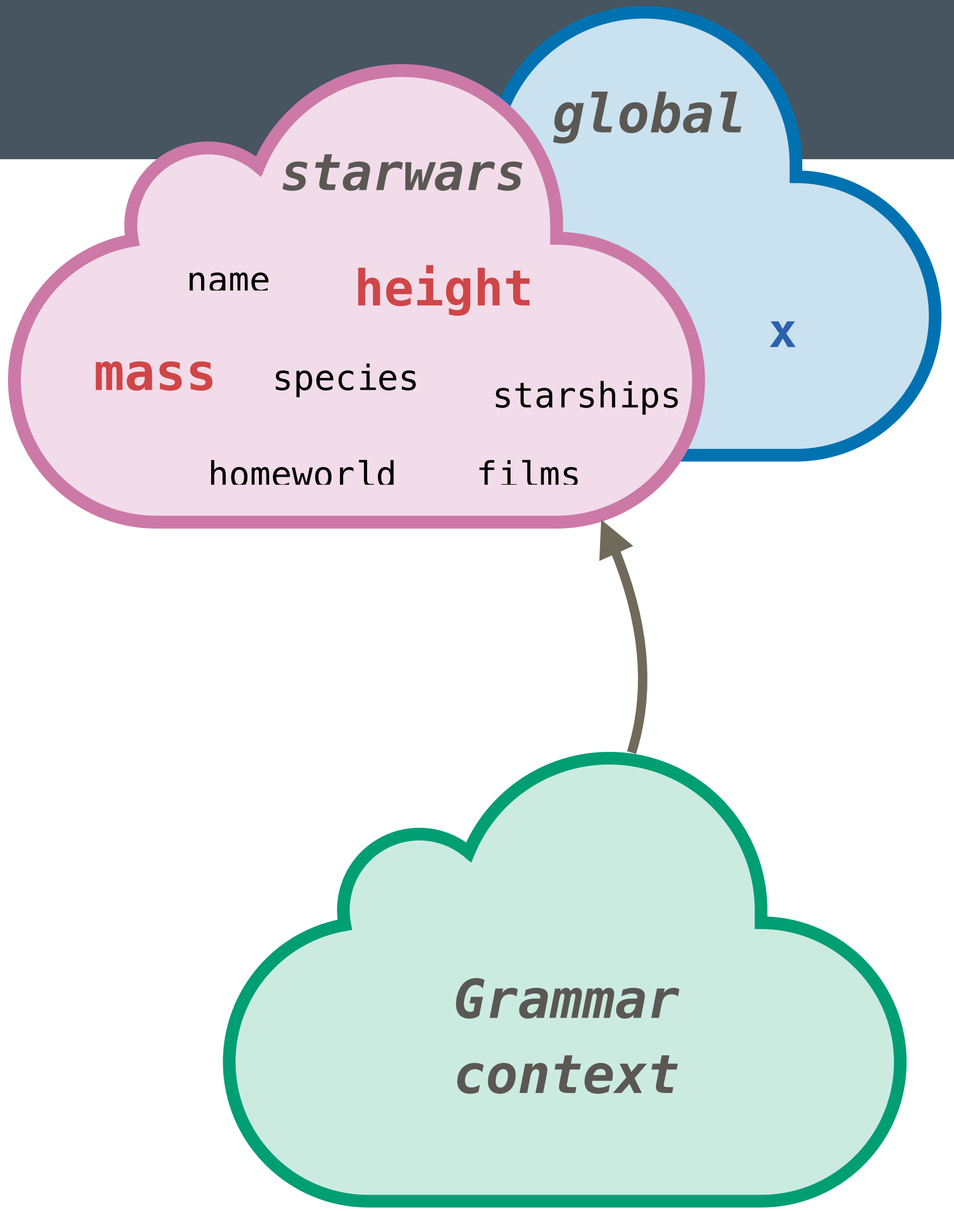
Quasiquotation

!! evaluates *at quoting time*

```
x <- sym("height")
mutate(starwars, bmi = mass / (!! x)^2)
```

(quoting time)

```
mutate(starwars, bmi = mass / height^2)
```



Quasiquotation

!! evaluates *at quoting time*

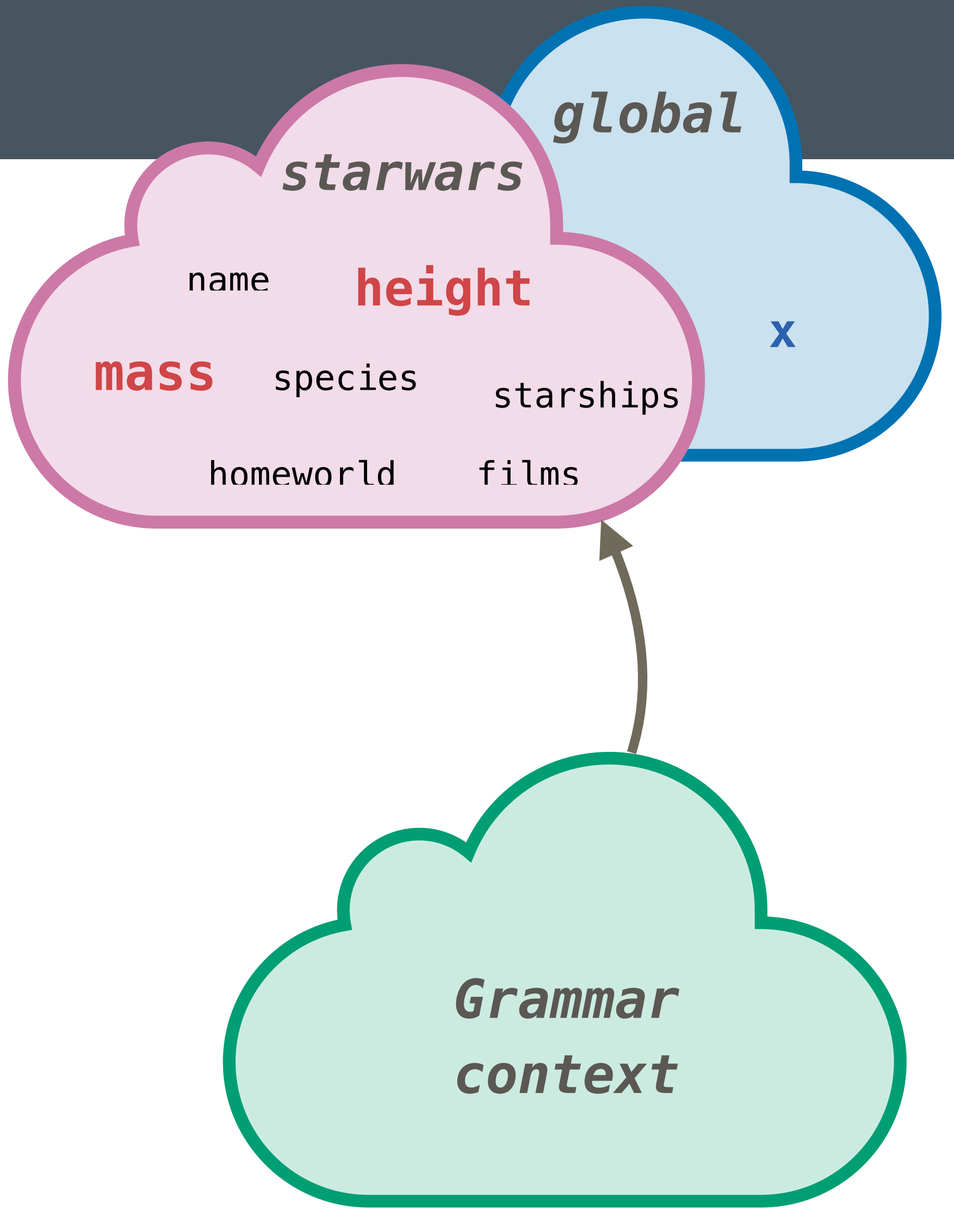
```
x <- sym("height")
mutate(starwars, bmi = mass / (!! x)^2)
```

(quoting time)

```
mutate(starwars, bmi = mass / height^2)
```

(evaluation time)

```
mutate(starwars, bmi = <dbl> / <dbl>^2)
```



Quasiquotation

Use expr() to debug quoting time

- Soon in dplyr (currently in rlang)
- Basic quoting like `quote()` but supports unquoting

```
x <- sym("height")

expr((!! x)^2)
#> (height)^2

expr(mutate(starwars, bmi = mass / (!! x)^2))
#> mutate(starwars, bmi = mass / (height)^2)
```

Creating a grammar wrapper

Let's use !! to create a dplyr **wrapper function**

Creating a grammar wrapper

We often take a grouped average?

Let's reduce duplication with a function

```
starwars %>%
  group_by(species) %>%
  summarise(avg = mean(height))
```

Creating a grammar wrapper

```
summarise_by <- function(df, group, var) {



  df %>%
    group_by(group) %>%
    summarise(avg = mean(var))
}
```

Wrapper takes strings

```
summarise_by(starwars, "species", "height")
```

Creating a grammar wrapper

```
summarise_by <- function(df, group, var) {
  group <- sym(group)
  var <- sym(var)

  df %>%
    group_by(!! group) %>%
    summarise(avg = mean(!! var))
}
```

Wrapper takes strings

```
summarise_by(starwars, "species", "height")
```

Creating a grammar wrapper

```
summarise_by <- function(df, group, var) {
  group <- sym(group)
  var <- sym(var)

  df %>%
    group_by(!! group) %>%
    summarise(avg = mean(!! var))
}
```

Wrapper takes strings

```
summarise_by(starwars, "species", "height")
#> # A tibble: 38 x 2
#>       species      avg
#>         <chr>    <dbl>
#>  1    Aleena  79.0000
#>  2  Besalisk 198.0000
```

Creating a grammar wrapper

Not bad but we can do better!

- Let's use **`enquo()`** to create a tidyverse-like wrapper
- Makes your function **quote its arguments** with !! support

Creating a grammar wrapper

```
summarise_by <- function(df, group, var) {
  group <- sym(group)
  var <- sym(var)

  df %>%
    group_by(!! group) %>%
    summarise(avg = mean(!! var))
}
```

Wrapper takes strings

```
summarise_by(starwars, "species", "height")
```

Creating a grammar wrapper

```
summarise_by <- function(df, group, var) {
  group <- enquo(group)
  var <- enquo(var)

  df %>%
    group_by(!! group) %>%
    summarise(avg = mean(!! var))
}
```

Quoting wrapper

```
summarise_by(starwars, species, height)
```

Creating a grammar wrapper

```
summarise_by <- function(df, group, var) {
  group <- enquo(group)
  var <- enquo(var)

  df %>%
    group_by(!! group) %>%
    summarise(avg = mean(!! var))
}
```

Quoting wrapper

```
summarise_by(starwars, species, height)
#> # A tibble: 38 x 2
#>       species      avg
#>         <chr>    <dbl>
#>  1    Aleena  79.0000
#>  2  Besalisk 198.0000
```

Summary

- Quoting functions are not like regular functions
  - We'll work on making quoting functions more obvious in IDE
  - Quoting makes it harder to program and create functions
- From quotation to quasiquotation
  - We need **unquoting** ⟶ tidy eval provides `!!`
  - Allows to program at quoting time
  - Use `expr()` to debug quoting time
- Use `enquo()` to create your own quoting functions

Summary

- What to learn next?
  - syms() and quos() for variable number of arguments
  - The concept of quosure is useful
- Better documentation is coming up
  - Tutorial in the dplyr programming vignette