Métaprogrammation en R

Sophie Baillargeon, Université Laval

2021-04-04

Table des matières

Note préliminaire : Lors de leur dernière mise à jour, ces notes ont été révisées en utilisant R version 4.0.3.

La métaprogrammation se définit par l'écriture d'un programme qui écrit lui-même un programme. En d'autres mots, faire de la métaprogrammation signifie de manipuler des éléments de langage sans les évaluer. En R, des exemples d'éléments du langage sont les noms, les formules, les expressions, les appels de fonctions, etc. Il est donc possible en R d'écrire un bout de code qui compose d'abord une ou des instructions sous forme d'expressions ou de chaînes de caractères, puis qui évalue ces instructions dans un deuxième temps. La métaprogrammation est parfois utile pour automatiser des calculs. Elle sert aussi à réaliser certaines tâches spécifiques, par exemple l'ajout d'annotations mathématiques à un graphique (abordé dans les notes sur les graphiques en R)

Les sections qui suivent décrivent quelques outils de métaprogrammation en R.

1 Assignation d'une valeur à un nom avec `assign`

Voici un exemple d'instruction R qui utilise la façon usuelle d'écrire une assignation en R :

```
obj <- head(letters)
```

Dans cette instruction, qui utilise l'opérateur d'assignation `<-`,

- `obj` est le nom référant à l'objet créé en mémoire ;
- `head(letters)` est l'expression dont la valeur, obtenue après évaluation, est enregistrée dans l'objet créé en mémoire.

Un nom est en fait un symbole relié à un objet.

Mais comment assigner une valeur à un nom stocké dans un objet sous forme de chaîne de caractères ? Par exemple, supposons que ce nom est stocké dans l'objet `nom`.

```
nom <- "nom_objet"
```

Nous voulons assigner une valeur au nom stocké dans `nom`, peu importe la chaîne de caractères que `nom` contient. Cette assignation peut être réalisée avec la fonction `assign` comme dans cet exemple :

```
assign(x = nom, value = head(letters))
```

Que contient notre environnement de travail maintenant ?

```
ls()
```

```
## [1] "nom"       "nom_objet" "obj"
```

Nous y trouvons un objet nommé `nom_objet`, ce qui correspond à la chaîne de caractères stockée dans `nom`. Cet objet contient la valeur (a, b, c, d, e, f), soit le vecteur des six premières lettres de l'alphabet latin créé par l'expression `head(letters)`.

```
nom_objet
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

Voici un autre exemple d'utilisation de la fonction `assign`. Supposons que nous souhaitons créer 5 objets, nommés `obj1` à `obj5`. Ces objets doivent contenir un vecteur d'entiers allant de 1 à `x` où `x` est le numéro de l'objet.

```
for (i in seq_len(5)) {
  assign(x = paste0("obj", i), value = 1:i)
}
```

Vérifions que ces objets ont bien été créés dans notre environnement de travail.

```
ls()
```

```
## [1] "i"         "nom"       "nom_objet" "obj"       "obj1"      "obj2"
## [7] "obj3"      "obj4"      "obj5"
```

```
obj1
```

```
## [1] 1
```

```
obj5
```

```
## [1] 1 2 3 4 5
```

2 Retour de la valeur assignée à un nom avec `get`

Pour atteindre la valeur assignée à un nom, nous sommes habitués à passer directement par ce nom, comme dans cet exemple.

```
str(nom_objet)
```

```
##  chr [1:6] "a" "b" "c" "d" "e" "f"
```

Mais comment procéder avec un nom stocké dans un objet sous forme de chaîne de caractères ? Il faut utiliser la fonction `get` comme suit.

```
str(get(nom))
```

```
##  chr [1:6] "a" "b" "c" "d" "e" "f"
```

Ce qui ne retourne pas la même chose que ceci.

```
str(nom)
```

```
# Équivalent à
str("nom_objet")
```

```
##  chr "nom_objet"
```

Par exemple, pour afficher le contenu des objets nommés `obj1` à `obj5`, nous pouvons procéder comme suit.

```
for (i in 1:5) {
  cat(paste0("obj", i), "=", get(paste0("obj", i)), "\n")
}
```

```
## obj1 = 1
## obj2 = 1 2
## obj3 = 1 2 3
## obj4 = 1 2 3 4
## obj5 = 1 2 3 4 5
```

3 Appel d'une fonction avec `do.call`

Nous venons d'apprendre comment manipuler un nom que nous possédons sous forme de chaîne de caractères. Comment procéder lorsque ce nom est celui d'une fonction que nous souhaitons appeler ?

Il est alors possible d'utiliser `get`, mais une autre fonction peut aussi nous être utile : `do.call`.

Par exemple, les trois instructions suivantes provoquent toute l'évaluation du même appel à la fonction `median`.

```
median(x = 1:10)
```

```
## [1] 5.5
```

```
get("median")(x = 1:10)
```

```
## [1] 5.5
```

```
do.call("median", args = list(x = 1:10))
```

```
## [1] 5.5
```

En fait, l'avantage de `do.call` n'est pas qu'il soit capable de manipuler une fonction dont le nom est sous forme de chaîne de caractères. D'ailleurs, `do.call` accepte comme premier argument la fonction directement.

Sa principale utilité est plutôt d'accepter sous forme de liste les arguments à inclure dans l'appel à une fonction. Cette liste peut être construite par étapes, potentiellement conditionnelles.

Voici un exemple de fonction qui exploite le potentiel de la fonction `do.call`.

```
#' @title Calcul de statistiques descriptives
#' @description Cette fonction permet de calculer des statistiques descriptives au choix
#' @param x vecteur d'observations
#' @param choix une chaîne de caractères spécifiant le nom de la fonction à appeler pour
#'              le calcul, soit "table" (par défaut), "mean", "median", "sd" ou "mad"
#' @param retirer_NA un logique spécifiant si les valeurs manquantes (NA)
#'                   doivent être retirées avant le calcul (par défaut TRUE)
#' @return le résultat de l'appel à la fonction choisie
choixstat <- function(x, choix = c("table", "mean", "median", "sd", "mad"),
                      retirer_NA = TRUE) {
  choix <- match.arg(choix)
  arguments <- list(x)
  if (choix == "table") {
    arguments$useNA <- if (retirer_NA) "no" else "ifany"
  } else {
    arguments$na.rm <- retirer_NA
  }
  return(do.call(what = choix, args = arguments))
}
```

```
choixstat(x = c(2, 3, 2, 3, 3, 4, NA, 3), choix = "median", retirer_NA = TRUE)
```

```
# Équivalent
median(c(2, 3, 2, 3, 3, 4, NA, 3), na.rm = TRUE)
```

```
## [1] 3
```

```
choixstat(x = c(2, 3, 2, 3, 3, 4, NA, 3), choix = "median", retirer_NA = FALSE)
```

```
# Équivalent à
median(c(2, 3, 2, 3, 3, 4, NA, 3), na.rm = FALSE)
```

```
## [1] NA
```

```
choixstat(x = c(2, 3, 2, 3, 3, 4, NA, 3), choix = "table", retirer_NA = TRUE)
```

```
# Équivalent à
table(c(2, 3, 2, 3, 3, 4, NA, 3), useNA = "no")
```

```
##
## 2 3 4
## 2 4 1
```

```
choixstat(x = c(2, 3, 2, 3, 3, 4, NA, 3), choix = "table", retirer_NA = FALSE)
```

```
# Équivalent à
table(c(2, 3, 2, 3, 3, 4, NA, 3), useNA = "ifany")
```

```
##
##    2    3    4 <NA>
##    2    4    1    1
```

4 Manipulation de formules

Les formules sont des éléments de langage R particuliers. Ils servent à spécifier des modèles statistiques. L'instruction suivante est un exemple de création d'une formule en R.

```
f1 <- y ~ x1 + x2
```

Une formule contient un opérateur `~`, possiblement avec un argument à gauche pour spécifier la ou les variables réponses du modèle et un argument à droite pour spécifier la ou les variables explicatives du modèle. Des informations sur les formules ont été fournies dans le cours sur les calculs statistiques en R.

R reconnaît que `f1` est bien une formule. Il la décompose même en trois parties : l'opérateur `~`, la partie de gauche du modèle (en anglais *LHS* pour *left hand side*) et la partie de droite du modèle (en anglais *RHS* pour *right hand side*).

```
str(f1)
```

```
## Class 'formula'  language y ~ x1 + x2
##   ..- attr(*, ".Environment")=<environment: 0x000001f2cef62448>
```

```
f1[1]
```

```
## `~`()
## <environment: 0x000001f2cef62448>
```

```
f1[2]
```

```
## y()
```

```
f1[3]
```

```
## (x1 + x2)()
```

4.1 Fonction `as.formula`

La fonction `as.formula` permet de créer une formule à partir d'une chaîne de caractères.

```
str("y ~ x1 + x2")
```

```
##  chr "y ~ x1 + x2"
```

```
f2 <- as.formula("y ~ x1 + x2")
str(f2)
```

```
## Class 'formula'  language y ~ x1 + x2
##   ..- attr(*, ".Environment")=<environment: 0x000001f2cef62448>
```

Cette possibilité de transformer une chaîne de caractères en formule s'avère pratique, par exemple, lorsque nous avons besoin de construire une formule comprenant un grand nombre de termes identifiables de façon automatique.

Voici un exemple de fonction qui utilise `as.formula` pour construire un modèle de régression polynomial.

```
#' @tital Régression polynomiale
#' @description Ajustement d'un modèle de régression polynomial entre deux variables
#' @param y vecteur des observations de la variable réponse
#' @param x vecteur des observations de la variable explicative
#' @param dg degré du modèle polynomial à ajuster
#' @return sortie de la fonction lm pour le modèle demandé
reg_poly <- function(y, x, dg){
  formule <- paste(
    "y ~",
```

```
    paste(paste0("I(x^", 1:dg, ")"), collapse = " + ")
  )
  return(lm(as.formula(formule)))
}
```

```
reg_poly(y = cars$dist, x = cars$speed, dg = 3)
```

```
##
## Call:
## lm(formula = as.formula(formule))
##
## Coefficients:
## (Intercept)       I(x^1)       I(x^2)       I(x^3)
##   -19.50505      6.80111     -0.34966      0.01025
```

```
reg_poly(y = cars$dist, x = cars$speed, dg = 5)
```

```
##
## Call:
## lm(formula = as.formula(formule))
##
## Coefficients:
## (Intercept)       I(x^1)       I(x^2)       I(x^3)       I(x^4)       I(x^5)
##   -2.650053     5.484259    -1.426123     0.194049    -0.010040     0.000179
```

4.2 Méthode `update.formula`

La méthode `update.formula` permet de partir d'un modèle et de le modifier. Par exemple, reprenons la formule `f1`.

```
f1
```

```
## y ~ x1 + x2
## <environment: 0x000001f2cef62448>
```

Ajoutons-y une variable explicative.

```
update(f1, . ~ . + x3)
```

```
## y ~ x1 + x2 + x3
## <environment: 0x000001f2cef62448>
```

Ou encore, retirons une variable.

```
update(f1, . ~ . - x2)
```

```
## y ~ x1
## <environment: 0x000001f2cef62448>
```

Nous pourrions aussi transformer une variable.

```
update(f1, sqrt(.) ~ .)
```

```
## sqrt(y) ~ x1 + x2
## <environment: 0x000001f2cef62448>
```

La fonction `update` est générique. Si le premier argument qu'elle reçoit est une formule, elle appelle la méthode `update.formula`. Dans un appel à `update.formula`, un point (symbole `.`) représente une partie de la formule originale. Le `.` à gauche du `~` représente le LHS de la formule fournie comme premier argument et le `.` à droite du `~` le RHS la formule fournie comme premier argument.

5 Manipulation d'instructions

Il est possible d'écrire des instructions R complètes, sous forme d'expression R ou de chaîne de caractères, sans les évaluer.

Prenons par exemple l'instruction R suivante : `median(x = 1:10)`. Il s'agit d'un appel à la fonction `median`. Si nous soumettons cette instruction dans la console, elle est évaluée et sa valeur, ici l'objet retourné en sortie par la fonction `median`, est imprimée.

```
median(x = 1:10)
```

```
## [1] 5.5
```

Si nous ajoutons une assignation au début de l'instruction comme suit :

```
out <- median(x = 1:10)
```

la sortie retournée par l'appel à la fonction `median` n'est pas imprimée. Elle est plutôt assignée au nom `out`, qui pointe donc maintenant vers un objet contenant la sortie produite.

```
out
```

```
## [1] 5.5
```

Mais comment stocker dans un objet l'instruction elle-même ?

5.1 Fonctions `quote` et `expression`

La fonction `quote` retourne une instruction R non évaluée, que nous pouvons appeler « expression ».

```
out_quote <- quote(median(x = 1:10))
str(out_quote)
```

```
##  language median(x = 1:10)
```

C'est un « élément de langage ».

La fonction `expression`, mentionnée dans les notes sur les graphiques en R pour l'ajout d'annotations mathématiques à un graphique, est similaire à la fonction `quote`, mais retourne un objet de langage un peu plus complexe.

```
out_exp <- expression(median(x = 1:10))
str(out_exp)
```

```
##   expression(median(x = 1:10))
```

Nous n'entrerons pas ici dans les détails techniques de la distinction entre ces deux types d'objets.

5.2 Fonction `eval`

Lorsque nous désirons évaluer une expression R, nous pouvons la fournir en entrée à la fonction `eval`. Les objets retournés par les fonctions `quote` et `expression` s'évaluent de la même façon.

```
eval(out_quote)
```

```
## [1] 5.5
```

```
eval(out_exp)
```

```
## [1] 5.5
```

5.3 Fonction `call`

Si l'instruction que nous souhaitons manipuler est un appel à une fonction, nous pouvons aussi créer l'expression non évaluée de l'instruction avec la fonction `call`.

```
out_call <- call("median", x = 1:10)
str(out_call)
```

```
##  language median(x = 1:10)
```

Les objets `out_call` et `out_quote` sont équivalents.

```
all.equal(out_quote, out_call)
```

```
## [1] TRUE
```

Évaluer `out_call` se réalise de la même façon qu'évaluer `out_quote` ou `out_exp`.

```
eval(out_call)
```

```
## [1] 5.5
```

5.4 Fonction `parse`

Étant donné que les chaînes de caractères se manipulent facilement, il serait utile de pouvoir transformer une chaîne de caractères en expression. C'est ce que nous permet de faire la fonction `parse`.

Par exemple, si nous avons la chaîne de caractère suivante :

```
instruc_car <- "median(x = 1:10)"
str(instruc_car)
```

```
##  chr "median(x = 1:10)"
```

nous pouvons la transformer en expression non évaluée avec `parse` comme suit.

```
out_parse <- parse(text = instruc_car)
str(out_parse)
```

```
## length 1 expression(median(x = 1:10))
##  - attr(*, "srcref")=List of 1
##   ..$ : 'srcref' int [1:8] 1 1 1 16 1 16 1 1
##   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x000001f2cecd8188>
##  - attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x000001f2cecd8188>
##  - attr(*, "wholeSrcref")= 'srcref' int [1:8] 1 0 2 0 0 0 1 2
##   ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x000001f2cecd8188>
```

L'objet `out_parse` ressemble à l'objet `out_exp` et s'évalue de la même façon, avec la fonction `eval`.

```
eval(out_parse)
```

```
## [1] 5.5
```

Nous avons donc parcouru le chemin de transformation suivant :

> instruction sous forme de chaîne de caractères,
> → expression non évaluée avec `parse`,
> → évaluation de l'expression avec `eval`.

Grâce à ces outils, nous pourrions améliorer la fonction `reg_poly`. Vous aurez peut-être remarqué que dans l'impression de la sortie de cette fonction, le `Call` a toujours la même allure : `lm(formula = as.formula(formule))`. Ce n'est pas très informatif. Voici une deuxième version de cette fonction, utilisant `parse` et `eval` plutôt que `as.formula`, qui produit un affichage amélioré.

```r
#' @inherit reg_poly title description params return
reg_poly_2 <- function(y, x, dg){
  instruc <- paste0(
    "lm(y ~ ",
    paste(paste0("I(x^", 1:dg, ")"), collapse = " + "),
    ")"
  )
  return(eval(parse(text = instruc)))
}
```

```r
reg_poly_2(y = cars$dist, x = cars$speed, dg = 3)
```

```
##
## Call:
## lm(formula = y ~ I(x^1) + I(x^2) + I(x^3))
##
## Coefficients:
## (Intercept)       I(x^1)       I(x^2)       I(x^3)
##   -19.50505      6.80111     -0.34966      0.01025
```

```r
reg_poly_2(y = cars$dist, x = cars$speed, dg = 5)
```

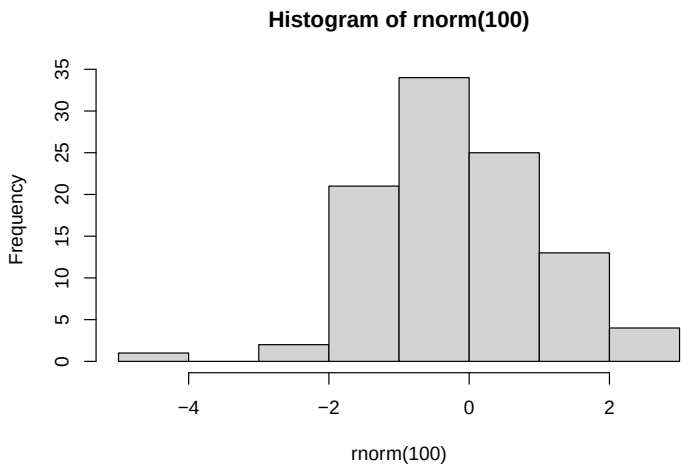
```
##
## Call:
## lm(formula = y ~ I(x^1) + I(x^2) + I(x^3) + I(x^4) + I(x^5))
##
## Coefficients:
## (Intercept)       I(x^1)       I(x^2)       I(x^3)       I(x^4)       I(x^5)
##   -2.650053     5.484259    -1.426123     0.194049    -0.010040     0.000179
```

6 Manipulation de l'appel d'une fonction

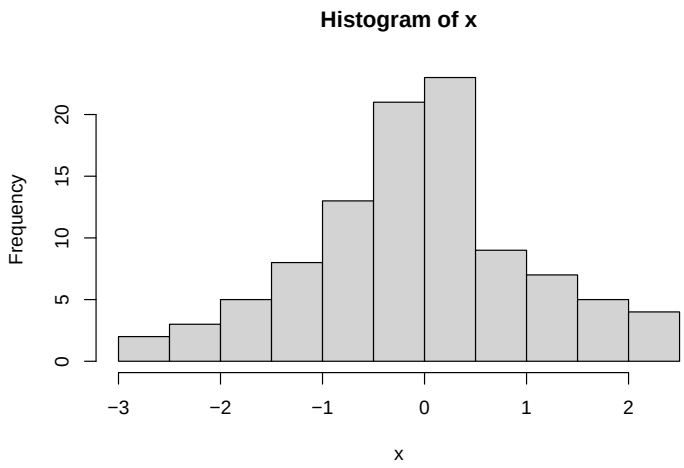
Dans le corps d'une fonction, il est parfois utile de pouvoir manipuler l'appel de la fonction.

Par exemple, la fonction `hist` utilise la façon dont la fonction a été appelée pour écrire le titre du graphique et le nom de l'axe des x.

```r
hist(rnorm(100))
```



```r
x <- rnorm(100)
hist(x)
```



Comment arriver à faire pareil ?

6.1 Fonctions `substitute` et `deparse`

Essayons de trouver dans le corps de la fonction `hist` comment elle procède. En fait, la fonction `hist` étant générique, allons voir dans le corps de sa méthode par défaut.

```r
View(hist.default)
# résultat non affiché ici, soumettre la commande dans la console pour le voir
```

Les fonctions `substitute` et `deparse` sont utilisées pour créer l'objet `xname`, qui est ensuite utilisé dans les valeurs par défaut des arguments `main` et `xlab`.

Exemples pour mieux comprendre `substitute` et `deparse` :

```r
fct1 <- function(x){
  return(x)
}
test <- fct1(1:10)
str(test)
```

```
##  int [1:10] 1 2 3 4 5 6 7 8 9 10
```

La fonction `fct1` retourne ici un vecteur, soit le résultat de l'évaluation de l'expression fournie en entrée à l'argument `x`.

```r
fct2 <- function(x){
  return(substitute(x))
}
test <- fct2(1:10)
str(test)
```

```
##  language 1:10
```

Lorsque `substitute` est utilisée dans le corps d'une fonction et qu'elle reçoit en entrée le nom d'un argument, elle retourne l'expression non évaluée fournie en entrée à cet argument dans l'appel de la fonction. La fonction `fct2` retourne donc une expression non évaluée (comme `quote` ou `call`).

```r
fct3 <- function(x){
  return(deparse(substitute(x)))
}
test <- fct3(1:10)
str(test)
```

```
##  chr "1:10"
```

Dans le corps de la fonction `fct3`, l'expression non évaluée retournée par `substitute` est transformée en chaîne de caractères par la fonction `deparse`. La fonction `deparse` permet de faire l'inverse de la fonction `parse` :

- `deparse` : expression → chaîne de caractères,
- `parse` : chaîne de caractères → expression.

6.2 Fonction `match.call`

Avec `substitute`, nous pouvons accéder à une expression assignée à un argument dans l'appel d'une fonction. La fonction `match.call`, que nous avons déjà mentionnée dans les notes sur les fonctions, permet quant à elle d'accéder à l'appel complet de la fonction.

```r
fct4 <- function(x){
  return(match.call())
}
test <- fct4(1:10)
str(test)
```

```
##  language fct4(x = 1:10)
```

La fonction `match.call` retourne une expression non évaluée, comme `substitute`. Dans cette expression, il est possible d'accéder à des arguments en particulier grâce à l'opérateur `$`.

```r
fct5 <- function(x, y){
  appel <- match.call()
  appel_details <- list(
    appel_complet = appel,
    arg_x_exp = appel$x,
    arg_x_car = deparse(appel$x),
    arg_y_exp = appel$y
  )
  return(appel_details)
}
test <- fct5(x = 1:10, y = "a")
str(test)
```

```
## List of 4
##  $ appel_complet: language fct5(x = 1:10, y = "a")
##  $ arg_x_exp    : language 1:10
##  $ arg_x_car    : chr "1:10"
##  $ arg_y_exp    : chr "a"
```

Dans cet exemple, la valeur fournie à l'argument `y` ne contient pas un appel à une fonction ou un opérateur. Il s'agit d'un seul élément, de type caractère. Son expression non évaluée est donc la valeur en caractères elle-même.

---

7 Résumé

Manipuler des éléments de langage sans les évaluer

- assignation d'une valeur à un nom : `assign` ;
- retour de la valeur assignée à un nom : `get` ;
- appel à une fonction en fournissant les arguments dans une liste : `do.call` ;
- manipulation de formules :
  - créer une formule à partir d'une chaîne de caractères : `as.formula`,
  - mettre à jour une formule : `update.formula` ;
- manipulation d'instructions :
  - capturer une instruction sans l'évaluer : `quote` ou `expression`,
  - évaluer une expression : `eval`,
  - écrire un appel de fonction sous forme d'expression (non évaluée) : `call`,
  - créer une expression (non évaluée) à partir d'une chaîne de caractères : `parse` (argument `text`) ;
- manipulation de l'appel d'une fonction :
  - capturer sans l'évaluer l'expression assignée à un argument dans un appel de fonction : `substitute`,
  - transformer une expression en chaîne de caractères : `deparse`,
  - capturer sous forme d'expression un appel de fonction : `match.call`.

Références

- R Core Team (2021). *R Language Definition*. R Foundation for Statistical Computing. Chapitre 6. URL http://cran.r-project.org/doc/manuals/r-patched/R-lang.html#Computing-on-the-language
- Wickham, H. (2014). *Advanced R*. CRC Press.
  - Chapitre 13 *Non-standard evaluation*, URL http://adv-r.had.co.nz/Computing-on-the-language.html
  - Chapitre 14. *Expressions*, URL http://adv-r.had.co.nz/Expressions.html
- Wickham, H. (2019). *Advanced R*. 2e édition. Chapman and Hall/CRC. Section 4. *Metaprogramming* URL https://adv-r.hadley.nz/metaprogramming.html