Guide de style de codage

Table of Contents

1. Introduction . . . 1
2. Longueur d'une ligne de code . . . 1
3. Indentation . . . 2
4. Mise en forme des blocs . . . 2
5. Conventions de nommage . . . 4
6. Majuscules . . . 6
7. Macros (Red/System) . . . 6
8. Définitions de fonctions . . . 7
9. Appels de fonctions . . . 9
10. Commentaires . . . 10
11. Syntaxe de chaîne de caractères . . . 11
12. Usage du retour à la ligne . . . 11

1. Introduction

Red est un langage homoiconique où le code est représenté comme les données. Une conséquence de cette propriété est que le langage est de forme presque totalement libre, afin de permettre toutes les manières possibles de formater les données, et aussi d'être assez flexible pour les besoins de formatage spécifiques aux DSL. Le contenu qui suit est *juste un des très nombreuses manières* de formatter du code Red.

Ce document décrit le style de codage *officiel utilisé dans le code source Red*, par conséquent le respect de ce style de codage est un prérequis de toute requête de mise à jour soumise au dépôt Github `red/red`.

Red/System étant un dialecte de Red, il partage les mêmes règles de syntaxe et de style de codage. Les règles spécifiques à Red/System sont indiquées comme telles.

Les objectifs des règles qui suivent sont de maximiser la lisibilité, y compris en gardant un nombre optimal de lignes de code visible à l'écran tout en minimisant le besoin de commentaires.

2. Longueur d'une ligne de code

Il n'y a pas de nombre de colonnes maximum strictement défini pour une ligne de code particulière, cela pouvant varier avec le type de police utilisé (taille, police proportionnelle ou de largeur fixe) ou les effets de mise en valeur. Il devrait être possible de lire une ligne de code complète (à l'exception des commentaires) dans un éditeur occupant au plus la moitié d'un moniteur de largeur 1080p. Sur les écrans que nous utilisons pour coder la base de code Red, cela fait environ 100 colonnes. Dans la description ci-dessous, *une taille excessive* ou *des expressions trop*

longues feront référence à des tailles de lignes de code qui ne satisfont pas les critères ci-dessus mentionnés.

3. Indentation

La base de code Red utilise des tabulations d'une taille de **4** colonnes pour indenter le code source. Cela donne un bon compromis entre des valeurs trop petites (comme deux colonnes) et trop grandes (comme 8 colonnes). L'utilisation de tabulations signifie également que vous pouvez l'ajuster à vos préférences personnelles dans votre éditeur, tout en respectant la règle (faites alors attention seulement à l'alignement à droite en utilisant les tabulations).

Tous les fichiers Red apportés en contribution au dépôt `red/red` devraient contenir le champ suivant dans leur en-tête:

```
Tabs: 4
```

Chaque fois que vous allez à la ligne après avoir ouvert un bloc ou une parenthèse, vous devriez indenter d'une tabulation.

Correct

```
func [
    arg1
    arg2
    ...
][
    print arg1
    ...
]
```

Incorrect

```
func [
arg1              ;-- indentation manquante!
arg2
...
][
       print arg1      ;-- indentation excessive!
       ...
]
```

4. Mise en forme des blocs

Toutes les règles suivantes s'appliquent aux blocs `[]` aussi bien qu'aux parenthèses `()`.

Les blocs vides ne contiennent aucun espace:

```
a: []
```

Les blocs contigus n'ont pas besoin d'un espace entre la fin de l'un et le début d'un autre:

```
[][]
[]()
```

```
[
   hello
][                      ;-- pas d'espace nécessaire
   world
]
```

Cependant, il est acceptable d'utiliser des espaces entre les blocs imbriqués:

```
 array: [[] [] [] []]
 list:  [ [] [] [] [] ]

 either a = 1 [["hello"]][["world"]]
 either a = 1 [ ["hello"] ][ ["world"] ]
```

Pour les expressions contenant de petits blocs, ils sont habituellement ouverts et fermés sur la même ligne:

```
b: either a = 1 [a + 1][3]
```

Si la ligne est trop longue, le bloc devrait couvrir plusieurs lignes avec un niveau d'indentation:

Correct

```
 b: either a = 1 [
     a + 1
 ][
     123 + length? mold a
 ]
```

Incorrect

```
 b: either a = 1
     [a + 1][123 + length? mold a]
```

Ce style est mauvais parce qu'il rompt la possibilité de copier/coller du code dans la console Red (`either` sera évalué avant que les arguments du bloc ne soient détectés).

Si le premier bloc est assez petit et peut loger sur la même ligne, alors seuls les blocs suivants couvrent plusieurs lignes:

```
print either a = 1 ["hello"][
    append mold a "ceci est une expression très longue"
]

while [not tail? series][
    print series/1
    series: next series
]
```

5. Conventions de nommage

Les **noms de variables** devraient être des **noms** d'un seul mot. Choisissez des mots qui sont courts et le plus significatifs possible. Les noms communs devraient être utilisés d'abord (*particulièrement s'ils sont déjà utilisés dans du code source Red dans le même contexte*). Au besoin, utilisez un dictionnaire de synonymes pour trouver le meilleur mot pour votre usage. Les mots d'une seule lettre ou abrégés (excepté lorsque le mot abrégé est d'usage courant) devraient être évités autant que possible.

Dans les noms faits de plusieurs mots ceux-ci sont séparés par un tiret `-`. Utilisez un nom de deux mots uniquement lorsqu'un mot unique adéquat ne peut être trouvé ou prêterait trop à confusion avec d'autres déjà utilisés. Les noms de variables faits de plus de deux mots devraient n'être utilisés que dans de rares cas. L'utilisation dans la mesure du possible de noms d'un seul mot rend le code horizontalement bien plus compact, ce qui augmente beaucoup la lisibilité. Evitez une verbosité inutile.

Correct

```
code: 123456
name: "John"
table: [2 6 8 4 3]
lost-items: []

unless tail? list [author: select list index]
```

Incorrect

```
code_for_article: 123456
Mytable: [2 6 8 4 3]
lostItems: []

unless tail? list-of-books [author-property: select list-of-books selected-index]
```

Les **noms de fonctions** devraient s'efforcer d'être des *verbes* d'un seul mot, afin d'exprimer une action, bien que des noms de deux ou trois mots soient souvent nécessaires. Dépasser trois mots devrait être évité autant que possible. Les conventions de nommage des variables s'appliquent également aux noms de fonctions. Un nom ou un adjectif suivi par un point d'interrogation est aussi accepté. Souvent, cela indique que la valeur renvoyée est du type `logic!`, mais cela n'est pas une règle stricte, étant donné qu'il est pratique de former des noms d'actions d'un seul mot pour récupérer une propriété (e.g. `length?`, `index?`). Lorsque vous formez des noms de fonctions avec deux mots ou plus, mettez toujours le verbe en première position. Si les noms de variables et de fonctions sont choisis avec soin, le code devient presque auto-documenté, ce qui réduit souvent le besoin de commentaires.

Correct

```
make:   func [...
reduce: func [...
allow:  func [...
crunch: func [...
```

Incorrect

```
length:    func [...
future:    func [...
position:  func [...
blue-fill: func [...       ;-- devrait être fill-blue
```

Il y a une exception à ces règles de nommage qui s'applique aux noms d'API du système d'exploitation ou de tiers non-Red. Afin de rendre les noms de fonctions et de champs de structures spécifiques aux API facilement reconnaissables, leur nom original devrait être utilisé. Cela aide visuellement à distinguer de tels noms importés des noms courants du code Red ou Red/System. Par exemple:

```
tagMSG: alias struct! [
    hWnd   [handle!]
    msg    [integer!]
    wParam [integer!]
    lParam [integer!]
    time   [integer!]
    x      [integer!]
    y      [integer!]
 ]

#import [
   "User32.dll" stdcall [
       CreateWindowEx: "CreateWindowExW" [
           dwExStyle    [integer!]
           lpClassName  [c-string!]
           lpWindowName [c-string!]
           dwStyle      [integer!]
           x            [integer!]
           y            [integer!]
           nWidth       [integer!]
           nHeight      [integer!]
           hWndParent   [handle!]
           hMenu        [handle!]
           hInstance    [handle!]
           lpParam      [int-ptr!]
           return:      [handle!]
       ]
   ]
]
```

6. Majuscules

Tous les noms de variables et de fonctions devraient être en minuscules par défaut, sauf bonne raison d'utiliser des majuscules telle que:

- le nom est un acronyme e.g. GMT (Greenwich Mean Time)
- le nom est associé à une API du système d'exploitation ou de tiers (non-Red)

7. Macros (Red/System)

Appliquez les mêmes conventions de nommage pour choisir les noms de macros Red/System. Les macros utilisent généralement des majuscules pour leurs noms, c'est une manière de les distinguer facilement du reste du code (sauf si l'on a l'intention explicite de les faire ressembler à du code normal, comme les définitions de types de données pseudo-personnalisés). Lorsque des mots multiples sont utilisés, ils sont séparés par un tiret de soulignement `_` pour accroître encore la différence avec le code habituel.

(TBD: extraire tous les noms d'un seul mot utilisés dans la base de code Red à titre d'exemples)

8. Définitions de fonctions

La règle générale est de garder le bloc de spécifications sur une seule ligne. Le bloc corps peut être sur la même ligne ou sur plusieurs lignes. Dans le cas de Red/System, comme les blocs de spécifications tendent à être plus longs, la plupart des blocs de spécifications de fonctions couvrent plusieurs lignes, aussi à des fins de cohérence visuelle même les petits blocs sont souvent décomposés.

Correct

```
do-nothing: func [][]
increment: func [n [integer!]][n + 1]

increment: func [n [integer!]][
   n + 1
]

increment: func [
    n [integer!]
][
    n + 1
]
```

Incorrect

```
do-nothing: func [
][
]

do-nothing: func [

][

]

increment: func [
    n [integer!]
][n + 1]
```

Quand le bloc de spécifications est trop long il devrait être décomposé sur plusieurs lignes. Si ce bloc est décomposé, chaque qualification de type doit être sur la même ligne que son argument. Le bloc d'attributs optionnels devrait être sur sa propre ligne. Chaque raffinement commence sur une nouvelle ligne. S'il est suivi d'un seul argument, cet argument peut être sur la même ligne ou sur une nouvelle ligne avec une indentation (soyez juste cohérent avec les autres raffinements dans le même bloc de spécifications). Pour le raffinement `/local`, si les mots locaux ne sont pas suivis par

une annotation de type, ils peuvent être mis sur la même ligne.

Quand on déplie le bloc de spécifications sur plusieurs lignes, il est recommandé d'aligner les qualifications de types de données des arguments consécutifs sur la même colonne pour une lecture facilitée. Un tel alignement se fait de préférence en utilisant des tabulations (si vous suivez strictement ces règles de codage), ou sinon, par des espaces.

Correct

```
make-world: func [
    earth   [word!]
    wind    [bitset!]
    fire    [binary!]
    water   [string!]
    /with
        thunder [url!]
    /only
    /into
        space   [block! none!]
    /local
    plants animals men women computers robots
][
    ...
]
```

Incorrect

```
make-world: func [
   [throw] earth [word!]       ;-- le bloc d'attributs n'est pas sur sa propre ligne
       wind    [bitset!]
       fire [binary!]          ;-- qualification de type non alignée
       water   [string!]
       /with
           thunder [url!]
       /only
       /into space [block! none!]  ;-- non cohérent avec la mise en forme de /with
       /local
           plants animals      ;-- ligne coupée trop tôt
           men women computers robots
][
    ...
]
```

Pour les chaînes de documentation, la principale (décrivant la fonction) devrait être sur sa propre ligne si le bloc de spécifications est décomposé. Les chaînes de documentation d'arguments et de raffinements devraient être sur la même ligne que l'objet qu'elles décrivent. Les chaînes de documentation commencent par une lettre majuscule et n'ont pas besoin de point final (il est ajouté automatiquement lors de l'affichage à l'écran par la fonction `help`).

Correct

```
increment: func ["Add 1 to the argument value" n][n + 1]

make-world: func [
    "Build a new World"
    earth    [word!]      "1st element"
    wind     [bitset!]    "2nd element"
    fire     [binary!]    "3rd element"
    water    [string!]
    /with         "Additional element"
        thunder [url!]
    /only         "Not implemented yet"
    /into         "Provides a container"
        space [unset!]    "The container"
    /local
        plants animals men women computers robots
][
   ...
]
```

Incorrect

```
make-world: func ["Build a new World"  ;-- devrait être sur une nouvelle ligne
    earth  [word!]     "1st element"
    wind   [bitset!]     "2nd element" ;-- indentation excessive
    fire   [binary!]
    "3rd element"          ;-- devrait être sur la même ligne que `fire`
    water  [string!]
    /with          "Additional element"
           thunder [url!]
    /only "Not implemented yet"    ;-- devrait être aligné avec les autres chaînes de
documentation
    /into
          "Provides a container"      ;-- devrait suivre le raffinement
           space [unset!]  "The container"
    /local
        plants animals men women computers robots
][
       ...
]
```

9. Appels de fonctions

Les arguments suivent l'appel de fonction sur la même ligne. Si la ligne devient trop longue, les arguments peuvent être décomposés sur plusieurs lignes (un argument par ligne) avec une indentation.

Correct

```
foo arg1 arg2 arg3 arg4 arg5

process-many
    argument1
    argument2
    argument3
    argument4
    argument5
```

Incorrect

```
foo arg1 arg2 arg3
    arg4 arg5

foo
    arg1 arg2 arg3
    arg4 arg5

process-many
    argument1
        argument2
            argument3
                argument4
                    argument5
```

Pour les expressions longues avec de nombreuses parties imbriquées, il peut être difficile de repérer les limites de chaque expression. Il est permis (mais pas obligatoire) d'utiliser des parenthèses pour grouper un appel imbriqué avec ses arguments.

```
head insert (copy/part [1 2 3 4] 2) (length? mold (2 + index? find "Hello" #"o"))

head insert
    copy/part [1 2 3 4] 2
    length? mold (2 + index? find "Hello" #"o")
```

10. Commentaires

Dans la base de code Red:

- les commentaires sont écrits en utilisant le préfixe `;--` (indication visuelle plus forte)
- les commentaires d'une seule ligne commencent colonne 57 (meilleure position en moyenne, sinon colonne 53).
- les commentaires sur plusieurs lignes se font en utilisant plusieurs préfixes d'une ligne plutôt

que des constructions `comment {⋯}`.

La règle général est de mettre les commentaires sur la même ligne que le début du code correspondant plutôt que sur une nouvelle ligne, afin d'économiser notablement l'espace vertical. Cependant, si le commentaire est utilisé pour séparer plusieurs lignes de code, alors on peut le mettre sur une nouvelle ligne.

11. Syntaxe de chaîne de caractères

Utilisez `""` pour les chaînes d'une seule ligne. La forme `{}` est réservée pour les chaînes multilignes. Le respect de cette règle garantit:

- une représentation plus cohérente du code source avant et après le chargement (LOAD) du code
- une meilleure transmission du sens

Le cas où une chaîne d'une seule ligne inclut le caractère " lui-même fait exception à la règle. Dans ce cas, on préfère utiliser la forme `{}` plutôt que le code d'échappement du guillemet `^"` car cela est plus lisible.

12. Usage du retour à la ligne

TBD