The REBOL Documentation Project

-- FR - Documentation REBOL - Manuels --

Manuels

Manuel de l'utilisateur - Chapitre 9 - Les Fonctions

Philippe Le Goff

Première publication : 4 mai 2005, et mis en ligne le mercredi 4 mai 2005

Résumé :

Ce document est la traduction française du Chapitre 9 du User Guide de REBOL/Core, qui concerne les Fonctions

Ce document est la traduction française du Chapitre 9 du User Guide de REBOL/Core, qui concerne les Fonctions.

Traducteur : Philippe Le Goff

Historique de la traduction

Date	Version	Commentaires	Auteur	Email
14 avril 2005 21:02	1.0.0	Traduction initiale	Philippe Le Goff	lp—legoff—free—fr

Vue d'ensemble

Plusieurs sortes de fonctions existent dans le langage REBOL :

- "Native" : une fonction qui est évaluée directement par le processeur. C'est le plus bas niveau pour les fonctions du langage.
- "Function" : une fonction de plus haut niveau définie par un bloc et évaluée en évaluant les fonctions au sein du bloc. Encore appelée "fonction utilisateur" (*user-defined function*).
- "Mezzanine" : un nom pour des fonctions de haut niveau qui font partie intégrante du langage. Ce ne sont cependant pas des fonctions natives .
- "Operator" : utilisé comme un opérateur. Quelques exemples : +, -, * et /.
- "Routine" : utilisée pour appeler des fonctions d'une librairie externe (pour REBOL/Command).

Evaluation des fonctions

Le chapitre consacré aux Expressions présente le détail de ce qu'est l'évaluation. La façon dont les arguments de fonctions sont évalués dicte l'ordre général des mots et des valeurs dans le langage.

La section suivante présente plus en détail comment est réalisée cette évaluation des fonctions.

Arguments

Les fonctions recoivent des arguments et renvoient des résultats.

Quoique certaines fonctions, comme **now** (date et heure courante), n'en nécessite pas, la plupart des fonctions nécessite un ou plusieurs arguments.

Les arguments fournis à une fonction sont traités par l'interpréteur et ensuite passés à la fonction.

Les arguments sont traités de la même manière, quelque soit le type de la fonction appelée : native, opérateur, utilisateur, ou autre.

Par exemple, la fonction **send** attend deux arguments :

```
friend: luke@rebol.com
message: "message in a bottle"
send friend message
```

Le mot "friend" est d'abord évalué et sa valeur (luke@rebol.com) est fournie à la fonction **send** comme premier argument. Ensuite le mot "message" est évalué, et sa valeur devient le second argument.

Pensez à ces valeurs "friend" et "message" comme étant substituées dans la ligne avant **send**, pour donner :

```
send luke@rebol.com "message in a bottle"
```

Si vous passez trop peu d'arguments à une fonction, un message d'erreur est renvoyé.

Par exemple, la fonction **send** attend deux arguments et si vous n'en fournissez qu'un, une erreur est générée :

```
send friend
** Script Error: send is missing its message argument.
** Where: send friend
```

A contrario, si *trop* d'arguments sont fournis, les valeurs en surplus sont ignorées :

```
send friend message "urgent"
```

Dans l'exemple précédent, **send** a deux arguments, de telle sorte que la chaîne "urgent" qui vient comme troisième argument, est ignorée.

Notez qu'aucune erreur ne se produit. Dans ce cas, il n'y a aucune fonction attendant le troisième argument. D'autre part, dans certains cas, le troisième argument pourrait provenir d'une autre fonction qui a été évaluée avant **send**.

Les arguments d'une fonction sont évalués de la gauche vers la droite. Cet ordre est respecté, sauf lorsque les arguments eux-mêmes sont des fonctions.

Par exemple, si vous écrivez :

```
send friend detab copy message
```

le second argument doit être calculé par l'évaluation successive des fonctions **detab** et **copy**.

Le résultat de **copy** doit être passé à **detab**, et le résultat de **detab** devra être passé à **send**.

Dans l'exemple précédent, la fonction **copy** prend un seul argument, le message, et retourne la copie de message. Le message copié est passé à la fonction **detab**, qui enléve les tabulations et renvoie le message sans elles.

Notez comment les résultats des fonctions passent de la droite vers la gauche lorsque l'expression est évaluée.

L'évaluation effectuée ici peut être clarifiée en utilisant des parenthèses. Les items entre parenthèses sont évalués d'abord. (Cependant, les parenthèses ne sont pas nécessaires et elles ralentiraient légérement l'évaluation).

```
send friend (detab (copy message))
```

L'effet en cascade des résultats passés aux fonctions est assez pratique. Voici un exemple qui utilise deux fois **insert** à l'intérieur de la même expression :

```
file: %image
insert tail insert file %graphics/ %.jpg
print file
graphics/image.jpg
```

Ici, un nom de répertoire et un suffixe ont été ajoutés à un nom de fichier. Des parenthèses peuvent être utilisées pour clarifier l'ordre de l'évaluation.

```
insert (tail (insert file %graphics/)) %.jpg
```

Une remarque concernant les parenthèses : Les parenthèses sont une bonne pratique lorsqu'on débute en REBOL. Cependant, vous devriez rapidement être capable de vous passer de cette aide, et d'écrire directement les expressions sans y mettre de parenthèses. Ne pas utiliser de parenthèses permet à l'interpréteur d'aller plus vite.

Types de données d'un argument

Habituellement, les fonctions utilisent des arguments ayant un type de données spécifiques. Par exemple, le premier argument de la fonction **send** peut uniquement être une adresse email ou un bloc d'adresses email. N'importe quel autre autre type de données produira une erreur.

```
send 1234 "numbers"
** Script Error: send expected address argument of type: email block.
** Where: send 1234 "numbers"
```

Dans l'exemple précédent, le message d'erreur explique que l'argument *address* de la fonction **send** doit être soit une adresse email, soit un bloc. Un moyen rapide de savoir quels types d'arguments sont acceptés par une fonction est de taper "help " suivi du nom de la fonction, à l'invite de commande, dans la console :

```
help send
USAGE:
    SEND address message /only /header header-obj
DESCRIPTION:
    Send a message to an address (or block of addresses)
```

```
    SEND is a function value.
ARGUMENTS:
    address -- An address or block of addresses (Type: email block)
    message -- Text of message. First line is subject. (Type: any)
REFINEMENTS:
    /only -- Send only one message to multiple addresses
    /header -- Supply your own custom header
        header-obj -- The header to use (Type: object)
```

La section ARGUMENTS indique le type de données attendu pour chaque argument. Notez que le second argument peut être de n'importe quel type (*any*). Ainsi il est possible d'écrire :

```
send luke@rebol.com $1000.00
```

Raffinements

Un raffinement autorise une variante dans l'usage normal de la fonction.

Les raffinements permettent aussi de fournir des arguments optionnels. Les raffinements peuvent exister aussi bien pour des fonctions natives que des fonctions utilisateurs.

Les raffinements sont spécifiés en faisant suivre le nom de la fonction par un slash, puis par le nom du raffinement.

Par exemple :

```
copy/part  (copy just part of a string)
find/tail  (return the tail of the match)
load/markup  (return XML/HTML tags and strings)
```

Les fonctions peuvent aussi inclure plusieurs raffinements :

```
find/case/tail (match case and return tail)
insert/only/dup (insert entire block multiple times)
```

Vous avez déjà vu comment la fonction **copy** est utilisée pour dupliquer une chaîne. Par défaut, **copy** renvoie une copie de son argument :

```
string: "no time like the present"
print copy string
no time like the present
```

En utilisant le raffinement **/part**, **copy** renvoie seulement une partie de la chaîne :

```
print copy/part string 7
no time
```

Dans l'exemple précédent, le raffinement **/part** spécifie que seulement sept premiers caractères de la chaîne doivent être copiés.

Pour savoir quels raffinements existent pour une fonction comme **copy**, utilisez là encore l'aide en ligne :

```
help copy
USAGE:
    COPY value /part range /deep
DESCRIPTION:
     Returns a copy of a value.
     COPY is an action value.
ARGUMENTS:
     value -- Usually a series (Type: series port bitset)
REFINEMENTS:
     /part -- Limits to a given length or position.
         range -- (Type: number series port)
     /deep -- Also copies series values within the block.
```

Notez que le raffinement **/part** nécessite un argument supplémentaire. Les raffinements n'imposent pas tous des arguments supplémentaires.

Par exemple, le raffinement **/deep** spécifie que **copy** effectuera des copies de tous les sous-blocs rencontrés. Aucun nouvel argument n'est requis.

Lorsque plusieurs arguments sont utilisés avec une fonction, l'ordre des arguments est déterminé par l'ordre dans lequel les raffinements sont indiqués.

Par exemple :

```
str: "test"
insert/dup/part str "this one" 4 5
print str
this this this this test
```

Inverser l'ordre des raffinements **/dup** et **/part** modifie l'ordre attendu des arguments. Vous pouvez voir la différence :

```
str: "test"
insert/part/dup str "this one" 4 5
print str
thisthisthisthisthistest
```

L'ordre des raffinements indique l'ordre des arguments.

Valeurs de fonction

L'exemple précédent décrivait comment les fonctions renvoient des valeurs quant elles sont évaluées.

Pourtant, parfois, vous voudrez obtenir la fonction elle-même en tant que valeur, et pas la valeur qu'elle retourne. Ceci peut être fait en faisant précéder le nom de la fonction du caractère " **:**" ou en utilisant **get function**.

Par exemple, pour définir un mot, **pr**, comme étant la fonction **print**, vous devriez écrire :

```
pr: :print
```

Ou encore :

```
pr: get `print
```

A présent, **pr** est équivalent à la fonction **print** :

```
pr "this is a test"
this is a test
```

Définir des fonctions

Vous pouvez définir vos propres fonctions utilisables comme des fonctions natives.

Ces fonctions sont appelées "fonctions utilisateur" (user-defined functions). Les fonctions définies par l'utilisateur ont pour type de données (datatype) : **function !**

does

Vous pouvez écrire des fonctions simples qui ne nécessitent pas d'arguments, avec la fonction **does**.

Cet exemple définit une nouvelle fonction qui affiche l'heure courante :

```
print-time: does [print now/time]
print-time
10:30
```

la fonction **does** renvoie une valeur, qui est la nouvelle fonction. Dans l'exemple, le mot "print-time" est associé à cette fonction.

Cette valeur de fonction peut être associée à un mot, être passée à une autre fonction, retournée comme résultat d'une fonction, sauvée dans un bloc, ou immédiatement évaluée.

Le paragraphe ci-dessous concernant la fonction has ne fait pas partie de la traduction originale. Il a été rajouté par souci de cohérence.

has

Une autre façon d'écire des fonctions simples sans arguments est d'utiliser **has**. br /> Cette fonction prend par contre des variables locales et est définie ainsi :

```
has var-locales body
```

Le premier argument (var-locales) est un bloc de variable(s) locale(s), le second, le corps de la fonction.

Par exemple :

```
jolie-ville: has [ville] [
    ville: ask "Où habitez-vous ? "
    print [ ville  " est une jolie ville "]
]

jolie-ville             Où habitez-vous ? Bruxelles
Bruxelles  est une jolie ville
```

func

Les fonctions qui nécessitent des arguments sont définies avec la fonction **func** qui accepte deux arguments :

```
func spec body
```

Le premier argument est un bloc fournissant les spécifications d'interface de la fonction.

C'est à dire : une description de la fonction, ses arguments, les types de données permis pour les arguments, les descriptions des arguments, et d'autres éléments. Le second argument est un bloc de code qui est évalué à chaque appel et évaluation de la fonction.

Voici un exemple d'une nouvelle fonction appelée **sum** :

```
sum: func [arg1 arg2] [arg1 + arg2]
```

La fonction **sum** accepte deux arguments, comme spécifié dans le premier bloc.

Le second bloc est le corps de la fonction, lequel, lorsqu'il est évalué, conduit à l'addition des deux arguments *arg1* et *arg2*. La nouvelle fonction est retournée en tant que valeur à la fonction **func** puis le mot **sum** lui est affecté.

En pratique :

```
print sum 123 321
444
```

le résultat de l'addition de arg1 et arg2 est renvoyé puis affiché.

func est définie dans REBOL. **func** est une fonction qui fabrique d'autres fonctions. Elle réalise un **make** sur le type de données **function !** (datatype). **func** est définie ainsi :

```
func: make function! [args body] [
    make function! args body
```

```
]
```

Spécifications d'interface

Le premier bloc d'une définition de fonction est appelé sa "*spécification d'interface*".

Ce bloc inclut une description de la fonction, ses arguments, les types de données permis pour les arguments, les descriptions des arguments, et d'autres éléments. La spécification d'interface est un dialecte de REBOL (en ceci qu'il y a des régles d'évaluation différentes de celles pour le code normal).

Le bloc de spécification posséde le format :

```
[
    "function description"
    [optional-attributes]
argument-1 [optional-type]
"argument description"
argument-2 [optional-type]
"argument description"
...
/refinement
"refinement description"
refinement-argument-1 [optional-type]
"refinement argument description"
...
]
```

Les champs du bloc de spécification sont :

NDT : ici, certains items n'ont pas été traduits, pour rester en correspondance avec le code ci-dessus.

- Description : un courte description de la fonction. C'est une chaîne de caractères qui peut être consultable par d'autres fonctions telle que l'aide en ligne "help", pour expliciter l'usage et l'objet de la fonction.
- Attributs : un bloc qui décrit des propriétés spéciales de la fonction, comme son comportement dans les cas d'erreur. Il devrait être étendu dans le futur pour inclure des flags pour les optimisations.
- Argument : une variable qui est utilisée pour accéder à un argument dans le corps de la fonction.
- Arg Type : un bloc identifiant les types de données qui seront acceptés par la fonction. Si un type de données non identifié dans ce bloc est passée à la fonction, une erreur se produira .
- Arg Description : une courte description de l'argument. Comme pour la description de la fonction, elle peut être consultée par d'autres fonctions comme **help**.
- Refinement : un mot décrivant le raffinement, qui précise qu'un comportement spécial sera effectué par la fonction.
- Refinement Description : une courte description du raffinement.
- Refinement Argument : une variable utilisée pour le raffinement.
- Refinement Argument Type : un bloc identifiant le type de données attendus pour le raffinement.

- Refinement Argument Description : une courte description de l'argument du raffinement

Tous ces champs sont optionnels.

A titre d'exemple, le bloc d'argument de la fonction **sum** (vue précédemment) est redéfini pour restreindre le type de données des arguments. Les descriptions de la fonction et des arguments sont également rajoutées :

```
sum: func [
    "Return the sum of two numbers."
    arg1 [number!] "first number"
    arg2 [number!] "second number"
][
    arg1 + arg2
]
```

A présent, le type de données des arguments est automatiquement contrôlé, ce qui conduit à générer des erreurs comme :

```
print sum 1 "test"
** Script Error: sum expected arg2 argument of type: number.
** Where: print sum 1 "test"
```

Pour autoriser d'autres types de données, il est possible d'en indiquer plusieurs :

```
sum: func [
    "Return the sum of two numbers."
    arg1 [number! tuple! money!] "first number"
    arg2 [number! tuple! money!] "second number"
][
    arg1 + arg2
]
print sum 1.2.3 3.2.1
4.4.4
print sum $1234 100
$1334.00
```

A présent, la fonction **sum** acceptera un nombre, un "tuple", une valeur monétaire comme arguments. Si au sein de la fonction, vous devez distinguer quel est le type de données passées, vous pouvez utiliser les fonctions habituelles pour les tests de type de données (datatypes).

```
if tuple? arg1 [print arg1]
if money? arg2 [print arg2]
```

Comme la fonction **sum** est maintenant décrite, la fonction **help** peut à présent fournir les informations pratiques sur elle :

```
help sum
USAGE:
    SUM arg1 arg2
```

```
DESCRIPTION:
     Return the sum of two numbers.
     SUM is a function value.
ARGUMENTS:
     arg1 -- first number (Type: number tuple money)
     arg2 -- second number (Type: number tuple money)
```

Arguments littéraux

Comme décrit auparavant, l'interpréteur évalue les arguments des fonctions et les passe au corps de la fonction. Cependant, il y a des fois où vous ne voudrez pas que les arguments des fonctions soient évalués.

Par exemple, si vous avez besoin de passer un mot et d'y accéder dans le corps de la fonction, vous ne voulez pas qu'il soit évaluez comme un argument.

La fonction d'aide en ligne, **help**, qui attend en argument un mot, est un bon exemple pour cela :

```
help print
```

Pour éviter que **print** soit évalué, la fonction **help** doit spécifier que son argument ne doit pas être évalué.

Pour indiquer que l'argument ne doit pas être évalué, faites précéder le nom de l'argument avec une apostrophe (signifiant un mot littéral).

Par exemple :

```
zap: func [`var] [set var 0]
test: 10
zap test
print test
10
```

L'argument *var* est précédé d'une apostrophe, ce qui informe l'interpréteur qu'il doit être pris tel quel *sans* être d'abord évalué. L'argument est passé comme un *mot*.

Par exemple :

```
say: func [`var] [probe var]
say test
test
```

L'exemple affiche le mot qui est passé en tant qu'argument.

Un autre exemple est une fonction qui incrémente de 1 une variable et renvoie le résultat (analogue à la fonction *incrément* ++ en C) :

```
++: func ['word] [set word 1 + get word]
count: 0
```

```
++ count
print count
1
print ++ count
2
```

Récupérer les arguments

Les arguments de fonction peuvent aussi spécifier que la valeur d'un mot doit être récupérée mais pas évaluée. C'est assez similaire à la situation des arguments littéraux décrite ci-dessus, mais plutôt que de passer le mot, c'est la valeur du mot qui est passée sans être évaluée.

Pour spécifier qu'un argument doit être récupéré sans évaluation, faites précéder le nom de l'argument du caractère " **:**". Par exemple, la fonction suivante accepte des fonctions en arguments :

```
print-body: func [:fun] [probe second :fun]
```

print-body affiche le corps de la fonction qui lui est fournie en argument. L'argument est précédé de deux points, qui indique que la valeur du mot devrait être obtenue, mais ne pas être évaluée.

```
print-body reform
[form reduce value]
print-body rejoin
[
    if empty? block: reduce block [return block]
    append either series? first block [copy first block] [
        form first block] next block
]
```

Définir des raffinements

Les raffinements peuvent être utilisés pour spécifier des variantes à l'utilisation normale de la fonction, ou encore pour fournir des arguments optionnels. Les raffinements sont ajoutés au bloc de spécification d'interface de la fonction, sous la forme de mots précédés du symbole "slash" (***/***). A l'intérieur du corps de la fonction, le mot lié au raffinement doit être testé comme une valeur logique afin de déterminer si le raffinement a été fourni, lorsque la fonction a été appelée.

Par exemple, le code suivant ajoute un raffinement à la fonction **sum**, qui a déjà été décrite dans les exemples précédents :

```
sum: func [
    "Return the sum of two numbers."
    arg1 [number!] "first number"
    arg2 [number!] "second number"
    /average "return the average of the numbers"
][
    either average [arg1 + arg2 / 2][arg1 + arg2]
]
```

La fonction **sum** posséde le raffinement ***/average***. Dans le corps de la fonction, le mot *average* est

testé par la fonction **either**, qui retourne *true*, si ce raffinement a été fourni :

```
print sum/average 123 321
222
```

Pour définir un raffinement qui accepte des arguments supplémentaires, faites suivre le raffinement des définitions des arguments :

```
sum: func [
    "Return the sum of two numbers."
    arg1 [number!] "first number"
    arg2 [number!] "second number"
    /times "multiply the result"
    amount [number!] "how many times"
][
    either times [arg1 + arg2 * amount][arg1 + arg2]
]
```

La variable *amount* est seulement valide lorsque le raffinement **times** est présent. Voici un exemple :

```
print sum/times 123 321 10
4440
```

N'oubliez pas de contrôler l'existence du raffinement, avant d'utiliser les arguments supplémentaires. Si l'argument d'un raffinement est utilisé sans que le raffinement soit spécifié, il prendra une valeur : **none**.

Variables locales

Une variable locale est un mot dont la valeur est définie *à l'intérieur du contexte* de la fonction. Les modifications d'une variable locale affecteront seulement la fonction dans laquelle la variable est définie. Si le même mot est utilisé en dehors de la fonction, il ne sera pas affecté par les changements de la variable locale du même nom, dont les valeurs sont définies dans le contexte de la fonction.

Par convention, les variables locales sont caractérisées par le raffinement **/local**. Le raffinement **/local** est suivi par la liste de mots qui sont utilisés comme variables locales au sein de la fonction.

```
average: func [
    block "Block of numbers"
    /local total length
][
    total: 0
    length: length? block
    foreach num block [total: total + num]
    either length > 0 [total / length][0]
]
```

Ici, les mots **total** et **length** sont les variables locales à la fonction.

Une autre méthode pour créer ces variables locales est d'utiliser la fonction **function**, qui est identique

à **func**, mais accepte un bloc séparé définissant les mots locaux :

```
average: function [
    block "Block of numbers"
][
    total length        ][
    total: 0
    length: length? block
    foreach num block [total: total + num]
    either length > 0 [total / length][0]
]
```

Dans cet exemple, notez que le raffinement **/local** n'est pas utilisé avec la fonction **function**.

La fonction **function** crée ce raffinement à votre place. Si une variable locale est utilisée avant que sa valeur ne soit définie dans le corps de sa fonction, elle prendra une valeur **none**.

Variables locales contenant des séries

Les variables locales qui manipulent des séries doivent être copiées (**copy**) si ces séries sont utilisées plusieurs fois. Par exemple, si vous désirez que la chaîne "*****" soit identique à chaque fois que vous appelez la fonction *start-name*, vous devrez écrire :

```
star-name: func [name] [
    stars: copy "**"
    insert next stars name
    stars
]
```

Sinon, si vous écrivez simplement :

```
star-name: func [name] [
    stars: "**"
    insert next stars name
    stars
]
```

vous utiliserez la même chaîne chaque fois. Et chaque fois que la fonction est employée, la valeur précédente apparaîtra dans le résultat.

```
print star-name "test"
*test*
print star-name "this"
*thistest*
```

C'est un principe IMPORTANT à se rappeler, car si vous l'oubliez, vous risquez d'observer des résultats aberrants dans vos programmes.

Renvoyer une valeur

Comme vous le savez depuis le chapitre sur les Expressions, l'évaluation d'un bloc renvoie la *dernière* valeur évaluée qu'il contient :

```
do [1 + 3  5 + 7]
12
```

C'est aussi vrai pour les fonctions. La dernière valeur est retournée comme le résultat de la fonction :

```
sum: func [a b] [
    print a
    print b
    a + b
]
print sum 123 321
123
321
444
```

De plus, il est possible d'utiliser la fonction **return** pour arrêter l'évaluation de la fonction à n'importe quel point et retourner la valeur :

```
find-value: func [series value] [
    forall series [
        if (first series) = value [
            return series
        ]
    ]
    none
]
probe find-value [1 2 3 4] 3
[3 4]
```

Dans l'exemple, si la valeur "3" est trouvée, la fonction retourne la série à la position de la correspondance.

Sinon, la fonction renvoie **none**.

Pour arrêter l'évaluation d'une fonction sans retourner de valeur, utilisez la fonction **exit** :

```
source: func [
    "Print the source code for a word"
    'word [word!]
][
    prin join word ": "
    if not value? word [print "undefined" exit]
    either any [
        native? get word op? get word action? get word
    ][
        print ["native" mold third get word]
    ][print mold get word]
```

```
]
```

Retourner plusieurs valeurs

Pour qu'une fonction renvoie plusieurs valeurs, utilisez un bloc. Vous pouvez faire cela facilement en retournant un bloc qui a été "réduit" (avec **reduce**).

Par exemple :

```
find-value: func [series value /local count] [
    forall series [
        if (first series) = value [
            reduce [series  index? series]
        ]
    ]
    none
]
```

La fonction renvoie un bloc constitué de : la série et de l'index où la valeur a été trouvée.

```
probe find-value [1 2 3 4] 3
[[3 4] 3]
```

La fonction **reduce** est nécessaire pour créer un bloc de valeurs à partir du bloc de mots fourni.

Ne retournez pas les variables locales elles-mêmes. C'est un mode de fonctionnement non supporté (actuellement).

Pour associer facilement des variables à la valeur de retour de la fonction, employez la fonction **set** :

```
set [block index] find-value [1 2 3 4] 3
print block
3 4
print index
3
```

Fonctions imbriquées

Des fonctions peuvent en définir d'autres.

Les sous-fonctions peuvent être globales, locales, ou retournées en tant que résultat, selon le but choisi.

Par exemple, pour créer une fonction globale à l'intérieur d'une fonction, rattachez-la à une variable globale :

```
make-timer: func [code] [
    timer: func [time] code
```

```
]
make-timer [wait time]
timer 5
```

Pour rendre locale une fonction, définissez-la comme une variable locale :

```
do-timer: func [code delay /local timer] [
    timer: func [time] code
    timer delay
    timer delay
]
do-timer [wait time] 5
```

La fonction *timer* existe seulement durant l'instant où la fonction **do-timer** est évaluée.

Pour retourner la fonction en tant que résultat :

```
make-timer: func [code] [
    func [time] code
]
timer: make-timer [wait time]
timer 5
```

Utilisez des variables locales correctes :

Vous devriez éviter d'employer des variables locales à la fonction de niveau supérieur, dans une sous-fonction imbriquée.

Par exemple :

```
make-timer: func [code delay] [
    timer: func [time] [wait time + delay]
]
```

Ici, le mot *delay* appartient dynamiquement à la fonction de **make-timer**. Ceci devrait être évité du fait que la valeur de *delay* changera dans des appels suivants à **make-timer**.

Fonctions anonymes

Les noms de fonctions sont des variables. En REBOL, une variable est une variable, indépendamment de ce qu'elle manipule. Il n'y a rien de spécial concernant les variables de fonctions. En outre, les fonctions ne nécessitent pas de noms.

Vous pouvez créer une fonction et l'évaluer immédiatement, la stocker dans un bloc, la passer comme argument à une autre fonction, ou retourner son résultat. Une telle fonction serait dite "anonyme".

Voici un exemple qui crée un bloc de fonctions anonymes :

```
funcs: []
repeat n 10 [
    append funcs func [t] compose [t + (n * 100)]
]
print funcs/1 10
110
print funcs/5 10
510
```

Les fonctions peuvent aussi être créées, puis passées à d'autres fonctions . Par exemple, quand vous utilisez la fonction **sort** avec votre propre fonction de comparaison, vous fournissez votre fonction comme argument à **sort/compare** :

```
sort/compare data func [a b] [a > b]
```

Fonctions conditionnelles

Puisque des fonctions sont créées dynamiquement par évaluation, vous pouvez déterminer comment vous souhaitez créer une fonction, sur la base d'une autre information. C'est une manière de fournir du code conditionnel comme cela existe dans des langages de macro ou d'autres langages de programmation.

Au sein du langage REBOL, ce type de code conditionnel est construit avec du code REBOL classique.

En particulier, vous pouvez créer une version d'une fonction pour le débuggage, qui affichera des informations supplémentaires :

```
test-mode: on
timer: either test-mode [
    func [delay] [
        print "delaying..."
        wait delay
        print "resuming"
    ]
][
    func [delay] [wait delay]
]
```

Ici, l'une des deux fonctions (*func [delay]*)est définie en se basant sur le test (*either test-mode*) de la valeur du "test-mode" courant. Vous pouvez encore écrire de façon plus concise :

```
timer: func [delay] either test-mode [[
    print "delaying..."
    wait delay
```

```
    print "resuming"
]][[wait delay]]
```

Attributs de fonctions

Les attributs de fonctions permettent de contrôler spécifiquement certains comportements comme la méthode qu'emploie une fonction pour manipuler les erreurs ou pour rendre la main (*exit*).

Les attributs sont des mots spécifiés dans un bloc optionnel, à l'intérieur des spécifications d'interface. Il y a actuellement deux attributs (ce sont des fonctions) : **catch** et **throw**.

Les messages d'erreur sont affichés, typiquement, lorsque l'une d'elles se produit à l'intérieur du corps de la fonction.

catch

Si l'attribut **catch** est spécifié, les erreurs émises à l'intérieur de la fonction seront capturées automatiquement par celle-ci.

Les erreurs ne sont pas affichées au sein de la fonction, mais au point où la fonction se trouvait.

Ceci peut être utile si vous avez une fonction de type mezzanine et que vous voulez

voir apparaître l'erreur précisément là où elle s'est produite :

```
   root: func [[catch] num [number!]] [
       if num             throw make error! "only positive numbers"
       ]
       square-root num
   ]
root 4
2
root -4
**User Error: only positive numbers
**Where: root -4
```

Remarquez que dans cet exemple, l'erreur se produit où **root** a été appelée quoique l'erreur réelle se soit produite dans le corps de la fonction. Ceci est dû à l'usage de l'attribut **catch**.

Sans l'attribut **catch**, l'erreur se produirait dans la fonction **root** :

```
root: func [num [number!]] [
    square-root num
]
root -4
** Math Error: Positive number required.
** Where: square-root num
```

L'utilisateur peut ne pas rien savoir du contenu de la fonction **root**. Et le message d'erreur porterait à confusion. En effet, l'utilisateur connait seulement **root**, mais ici l'erreur se trouve dans l'usage de la racine carrée.

Ne confondez pas l'attribut **catch** avec la fonction **catch**. Bien qu'ils se ressemblent, la fonction **catch** s'applique à n'importe quel bloc à évaluer.

throw

L'attribut **throw** vous permet d'écrire vos propres fonctions de contrôle, comme dans **for**, **foreach**, **if**, **loop** et **forever**, en forçant vos fonctions à rendre la main (**return** ou **exit**).

Par exemple, la fonction **loop-time** :

```
loop-time: func [time block] [
    while [now/time  ]
```

évalue un bloc, jusqu'à ce qu'un temps déterminé soit atteint ou dépassé.

Cette boucle peut être utilisée à l'intérieur d'une fonction :

```
do-job: func [job][
    loop-time 10:30 [
        if error? try [page: read http://www.rebol.com]
            [return none]
    ]
    page
]
```

Maintenant, que se produit-il quand le bloc *[return none]* est évalué ?

Puisque ce bloc est évalué par la fonction **loop-time**, le retour se fait dans cette fonction, et pas pour la fonction **do-job**. Ceci peut être évité avec l'attribut **throw** :

```
loop-time: func [[throw] time block] [
    while [now/time  ]
```

L'attribut **thrown** force le retour ou la sortie qui se produit, à être répercuté au niveau supérieur, ce qui conduirait la fonction précédente **do-job** à rendre la main.

Références avant définition

Parfois, un script a besoin de faire référence à une fonction avant que celle-ci ne soit définie.

Ceci peut se faire de la manière suivante : il est possible de faire référence à une fonction, sans que celle-ci soit définie, pour peu que cette fonction ne soit pas (encore) évaluée.

```
buy: func [item] [
```

```
    append own item
    sell head item   ; la fonction "sell" apparait avant d'être définie
                     ; mais elle n'est évaluée car dans le corps d'une fonction
]
sell: func [item] [
    remove find own item
]
```

Portée des variables

Le contexte des variables est appelé leur portée. La portée d'une variable peut être globale ou locale. REBOL utilise une forme de portée statique, qui est appelée : portée de definition (definitional scoping). La portée d'une variable est déterminée lorsque son contexte est défini. Dans le cas d'une fonction, elle est déterminée par la façon dont la fonction est définie.

Toutes les variables locales définies dans une fonction ont une portée relative à cette fonction. Les fonctions et les objets imbriqués sont susceptibles d'accéder aux mots de leurs "parents".

```
a-func: func [a] [
    print ["a:" a]
    b-func: func [b] [
        print ["b:" b]
        print ["a:" a]
        print a + b
    ]
    b-func 10
]
a-func 11
a: 11
b: 10        a: 11        21
```

Notez ici que la fonction **b-func** a accès à la variable *a* de *a-func*.

Les mots liés en dehors d'une fonction maintiennent leurs liens, même évalués dans la fonction.

C'est une conséquence de cette portée statique, et cela vous permettrait d'écrire vos propres fonctions d'évaluations de bloc. (comme pour **if**, **while**, **loop**).

Par exemple, voici une fonction **ifs** qui évalue un bloc sur trois, elle se base sur le test conditionnel d'un signe :

```
ifs: func [
    "If positive do block 1, zero do block 2, minus do 3"
    condition block1 block2 block3
][
    if positive? condition [return do block1]
    if negative? condition [return do block3]
    return do block2
]
```

```
print ifs 12:00 - now/time ["morning"]["noon"]["night"]
night
```

Les blocs passés à la fonction peuvent contenir les mêmes mots que ceux utilisés dans la fonction, sans interférer avec ces mots définis localement à la fonction. Ceci est dù au fait que les mots passés à la fonction ne lui sont pas liés.

L'exemple suivant passe à la fonction **ifs** les mots *block1, block2 et block3* comme des mots pré-définis. La fonction **ifs** ne fait pas de confusion entre les mots passés en arguments et les mots de mêmes noms définis localement :

```
block1: "morning right now"
block2: "just turned noon"
block3: "evening time"
print ifs (12:00 - now/time) [block1][block2][block3]
evening time
```

Réflectivité des Propriétés

La spécification de toutes les fonctions peut être obtenue et manipulée pendant l'exécution.

Par exemple, vous pouvez afficher le bloc de spécification d'une fonction avec :

```
probe third :if
[
    "If condition is TRUE, evaluates the block."
    condition
    then-block [block!]
    /else "If not true, evaluate this block"
    else-block [block!]
]
```

Le code du corps des fonctions peut être obtenu avec :

```
probe second :append
[
    head either only [
        insert/only tail series :value
    ][
        insert tail series :value
    ]
]
```

Les fonctions peuvent être dynamiquement interrogées pendant l'évaluation.

C'est ainsi que les fonctions **source** et **help** fonctionnent et que les messages d'erreur sont formatés.

En plus, ce système est utile pour créer vos propres versions des fonctions existantes. Par exemple,

une fonction utilisateur **print** peut être créée avec exactement les mêmes spécifications que l'originale **print**, mais envoie sa sortie vers une chaîne plutôt que vers l'écran.

```
output: make string! 1000
print-str: func third :print [
    repend output [reform :value newline]
]
```

Le nom de l'argument utilisé pour **print-str** est obtenu à partir de la spécification d'interface de **print**.

Vous pouvez examiner cette spécification avec :

```
probe third :print
[
    "Outputs a value followed by a line break."
    value "The value to print"
]
```

Fonction d'aide en ligne : Help

Une information utile sur toutes les fonctions du système peut être récupérée avec la fonction **help** :

```
help send
USAGE:
    SEND address message /only /header header-obj
DESCRIPTION:
     Send a message to an address (or block of addresses)
     SEND is a function value.
ARGUMENTS:
     address -- An address or block of addresses (Type: email block)
     message -- Text of message. First line is subject. (Type: any)
REFINEMENTS:
     /only -- Send only one message to multiple addresses
     /header -- Supply your own custom header
         header-obj -- The header to use (Type: object)
```

Toutes ces informations proviennent de la définition de la fonction.

De l'aide peut être obtenue pour tous les types de fonctions, pas uniquement pour les fonctions internes ou natives.

La fonction **help** peut aussi être utilisée pour les fonctions utilisateur.

La documentation qui serait affichée concernant une fonction provient de la définition de celle-ci.

Vous pouvez aussi rechercher de l'aide sur des fonctions en indiquant une partie de leurs noms.

Par exemple, dans la console, vous pouvez taper :

```
Help "path"
Found these words:
     clean-path     (function)
     lit-path!      (datatype)
     lit-path?      (action)
     path!          (datatype)
     path-thru      (function)
     path?          (action)
     set-path!      (datatype)
     set-path?      (action)
     split-path     (function)
     to-lit-path    (function)
     to-path        (function)
     to-set-path    (function)
```

pour afficher tous les mots qui contiennent la chaîne "path".

Pour voir une liste de toutes les fonctions valables dans REBOL, tapez **what** dans la console :

```
what
* [value1 value2]
** [number exponent]
+ [value1 value2]
- [value1 value2]
/ [value1 value2]
// [value1 value2]
&lt; [value1 value2]
&lt;= [value1 value2]
&lt;&gt; [value1 value2]
= [value1 value2]
== [value1 value2]
=? [value1 value2]
&gt; [value1 value2]
&gt;= [value1 value2]
? ['word]
?? ['name]
about []
abs [value]
absolute [value]
...
```

Afficher le code source

Une autre technique pour apprendre REBOL et pour gagner du temps dans l'écriture de vos propres fonctions, est de regarder comment sont définies les fonctions mezzanine de REBOL.

Vous pouvez utiliser la fonction **source** pour cela :

```
source source
source: func [
    "Prints the source code for a word."
    'word [word!]
][
    prin join word ": "
    if not value? word [print "undefined" exit]
    either any [native? get word op? get word action? get word] [
        print ["native" mold third get word]
    ] [print mold get word]
]
```

Ci-dessus le propre code de la fonction **source**.

Notez que vous ne pourrez voir le code source des fonctions natives car elles existent seulement dans le code du binaire. Cependant, la fonction **source** affichera la spécification d'interface de la fonction native.

Par exemple :

```
source add
add: native [
    "Returns the result of adding two values."
    value1 [number! pair! char! money! date! time! tuple!]
    value2 [number! pair! char! money! date! time! tuple!]
]
```