The REBOL Documentation Project

-- FR - Documentation REBOL - Articles Techniques --

Articles Techniques

Les mots de REBOL et leur contexte, ou la Bindologie

REBOLtof

Première publication : 1er juillet 2005, et mis en ligne le vendredi 1er juillet 2005

Résumé :

Cette article constitue une traduction de l'article DE référence en matière de mots et de contexte en REBOL.

Cette article constitue une traduction de l'article DE référence en matière de mots et de contexte en REBOL.

Auteur : Ladislav Mecir
Traducteur : Christophe Coussement
Article original : sur le site de l'auteur

1 Historique

Version	Date	Commentaire	Auteur

1.0	01 Jul 05	Initial	REBOLtof-at-yahoo-dot-com
1.1	07 Jul 05	Correction de typos et ajout d'améliorations suggérées par Ladislav Mecir	REBOLtof-at-yahoo-dot-com

2 Remerciements

Je souhaite remercier tous ceux qui ont contribués à cet article. En particulier Galt Barber, Mark Dickson, Elan Goldman, Brian Hawley, Gregg Irwin, Thomas Jensen, Holger Kruse, Larry Palmiter, Patrick Philipot, Gabriele Santilli, Frank Sievertsen et Romano Paolo Tenca. Les erreurs éventuelles sont de mon fait..

3 références

Les articles de Ladislav (REBOL Articles) où vous trouverez d'autres références utiles.

Le code de cet article a été testé sous REBOL/View 1.2.8. Il est possible que des versions plus anciennes engendrent des résultats diffèrents.

Un lecteur intéressé par le test du code proposé en exemple, peut procéder de la sorte :

```
do http://www.fm.vslib.cz/~ladislav/rebol/contexts.r
```

ce qui définira automatiquement toutes les fonctions servant à illustrer cet article.

Les dernières versions des fonctions NM-USE et de CFUNC peuvent être retrouvées dans les fichiers nm-use.r and in the cfunc.r.

4 Mots

Cet article traite des mots REBOL. Les mots REBOL, ainsi que les autres valeurs de REBOL, possédent un type. Voyons à présent les différentes valeurs possibles de mots :

```
type? first [rebol] ; == word!
type? first [rebol:] ; == set-word!
type? first [:rebol] ; == get-word!
type? first ['rebol] ; == lit-word!
type? first [/rebol] ; == refinement!
```

Tous ces mots REBOL ont en commun un pseudo-type appelé **ANY-WORD!** :

```
any-word? first [rebol] ; == true
any-word? first [rebol:] ; == true
any-word? first [:rebol] ; == true
any-word? first ['rebol] ; == true
any-word? first [/rebol] ; == true
```

5 Orthographe

La propriété intéressante suivante de REBOL est l'orthographe (NdT : *spelling*). Nous la définissons de la manière suivante :

```
spelling: func [
    {retourne l'orthographe d'un mot}
    word [any-word!]
] [
    if word? :word [return mold word]
    if set-word? :word [return head remove back tail mold :word]
    next mold :word
]
```

L'orthographe d'un mot est une chaîne et est une des propriétés que les exemples ci-dessus ont en commun :

```
spelling first [rebol] ; == "rebol"
spelling first [rebol:] ; == "rebol"
spelling first [:rebol] ; == "rebol"
spelling first ['rebol] ; == "rebol"
spelling first [/rebol] ; == "rebol"
```

Observation (Orthographe et Egalité) :

Les mots ayant une orthographe égale sont égaux, même s'ils ont des datatypes différents

Illustration :

```
equal? first [rebol] first [rebol:] ; == true
```

Observation (Orthographe irrégulière) :

Normalement, les mots n'ont pas une orthographe contenant des espaces, commençant par un point, etc... D'autre part, il est possible de créer un mot possédant n'importe quelle orthographe, de la manière suivante :

```
unusual: make word! ":unusual word:"
type? unusual ; == word!
spelling unusual ; == ":unusual word:"
```

6 Alias

Qu'est-ce qui est nécessaire pour que deux mots soient égaux ?

Observation (Egalité et Alias) :

Deux mots sont égaux s'ils disposent de la même orthographe ou s'ils sont des alias.

Illustration :

```
; créons un alias "revolutionary" au mot 'rebol
alias 'rebol "revolutionary"
;'rebol et 'revolutionary seront des mots égaux avec une orthographe différente:
equal? 'rebol 'revolutionary ; == true
equal? spelling 'rebol spelling 'revolutionary ; == false
```

Cette connaissance peut être utilisée afin de définir une fonction REBOL qui va nous dire si deux mots sont des alias :

```
aliases?: func [
    {trouve si word1 et word2 sont des alias}
    word1 [any-word!]
    word2 [any-word!]
] [
    found? all [
        equal? :word1 :word2
        not equal? spelling :word1 spelling :word2
    ]
]

aliases? 'rebol 'mean ; == false
aliases? 'rebol 'rebol ; == false
aliases? 'rebol 'revolutionary ; == true
```

7 Les mots peuvent référer à des valeurs REBOL

Une propriété importante des mots est leur capacité de référer des valeurs REBOL. Ainsi, afin d'obliger un mot 'rebolution à référer à une chaîne "uprising", nous pouvons faire :

```
rebolution: "uprising"
set 'rebolution "uprising"
set/any 'rebolution "uprising"
set first [rebolution:] "uprising"
set first [:rebolution] "uprising"
```

etc...

Pour obtenir la valeur référée par un mot, nous pouvons faire :

```
:rebolution
get 'rebolution
get/any 'rebolution
```

etc...

8 Les mots peuvent avoir un contexte

La capacité de référer des valeurs REBOL est très proche d'une propriété des mots REBOL nommées "liaison" (NdT : *binding*). Seuls les mots liés à un contexte (mots possédant un contexte) peuvent référer des valeurs REBOL.

```
; a refinement
get/any /rebol
** Script Error: rebol word has no context
** Near: get/any /rebol
```

Il peut être utile de découvrir si un mot dispose d'un contexte. Voici une manière de procéder :

```
has-context?: function [
     {le word dispose-t-il d'un contexte?}
     word [any-word!]
 ] [err] [
     not all [
         error? err: try [any-type? get/any :word]
         err: disarm err
         equal? err/id 'not-defined
     ]
 ]
```

Tests :

```
has-context? 'rebol ; == true
has-context? /rebol ; == false
```

Définition (Mots sans liens) :

Les mots ne disposant pas de contexte sont appelés mots sans liens Words having no context we call unbound words.

9 Liaison des mots

Il est facile de trouver un mot ayant une orthographe et un type spécifiques.

Parfois, nous souhaiterons disposer d'un mot de ce genre, dans un contexte particulier.

Ce travail consiste en général à trouver un mot ayant l'orthographe et le type d'un mot WORDS donné et le contexte d'un KNOWN-WORD donné.

Afin de trouver un tel mot, nous pouvons utiliser la fonction BIND.

10 Généralisation du BIND

La fonction BIND accepte uniquement une valeur du type WORD! au lieu d'un ANY-WORD! comme argument WORDS. Cela est pareil pour l'argument KNOWN-WORD. Si nous souhaitons obtenir un résultat pour une argument du type ANY-WORD!, nous pouvons utiliser la généralisation suivante de la fonction BIND :

```
bindany: function [
    {
        trouve un mot à l'orthographe égale
        et au type égal à WORD
        et d'une liaison égale à KNOWN-WORD
    }
    [catch]
    word [any-word!]
    known-word [any-word!]
] [wordt knownt words knowns spec result] [
    spec: third :bind
    words: next find spec 'words
    wordt: first words
    change/only words reduce [any-word!]
    knowns: next find spec 'known-word
    knownt: first knowns
    change/only knowns reduce [any-word!]
    error? result: try [bind :word :known-word]
    change/only words wordt
    change/only knowns knownt
    if error? result [throw result]
    :result
]
```

11 Propriétés de BIND

Il est important de connaître les propriétés de la fonction BIND car elles sont intimement liées aux propriétés des mots.

11.1 Liaison avec aucun contexte

Observation (Liaison avec aucun contexte) :

Si le KNOWN-WORD n'a pas de contexte, BIND déclenche une erreur.

Illustration :

```
a-word: second first context [rebol: 1] ; == rebol
has-context? a-word ; == false
```

```
bind 'a a-word
** Script Error: rebol word has no context
** Near: bind 'a a-word
```

11.2 Liaison quand l'argument WORDS est un WORD !

Observation (Cas trivial) :

Si l'argument WORDS et l'argument KNOWN-WORD ont une liaison (NdT : *binding*) égale, le BIND retourne l'argument WORDS.

Illustration :

```
known-word: words: 'a ; == a
result: bind words known-word
same? words result ; == true
```

Observation (Liaison effective) :

S'il existe un mot ayant une orthographe égale et un type identique à celui de l'argument WORDS et le même contexte que l'argument KNOWN-WORD, alors BIND retourne ce mot.

Illustration :

```
words: 'a ; == a
known-word: use [a b] ['b] ; == b
result: bind words known-word ; == a
same? words result ; == false
```

Nous pouvons constater que le résultat dispose de la même orthographe et du même type que le mot WORDS, mais qu'il n'est pas le mot WORDS.

Observation (Liaison inefficace) :

S'il n'existe pas de mot ayant les mêmes orthographe et type que l'argument WORDS et une liaison égale à celle de l'argument KNOWN-WORD, BIND retourne l'argument WORDS.

Illustration :

```
words: 'c ; == c
known-word: use [a b] ['b] ; == b
result: bind words known-word ; == c
same? words result ; == true
```

Dans ce cas, BIND retourne simplement le mot WORDS.

11.3 Liaison quand l'argument WORDS est un bloc

```
Observation (Liaison avec un bloc, sans copie):
```

Si le raffinement /COPY n'est pas utilisé, BIND remplace les éléments du block par le résultat de leur liaison. Il existe cependant une exception à cette règle : BIND ne lie pas les raffinements contenus dans le bloc.

12 Mots égaux avec une liaison différente

Des mots égaux ne doivent pas nécessairement avoir une même liaison. En fait, l'opposé est vrai : pour tous les mots, nous pouvons créer un mot avec une orthographe égale et une liaison différente.

```
nonsame: function [
    {
        pour un mot donné, crée un mot
        d'une orthographe égale
       et une liaison différente
    }
    word [any-word!] {the given word}
] [w] [
    w: any [
        all [lit-word? :word do reduce [:word]]
        all [get-word? :word to word! :word]
        all [refinement? :word to word! :word]
        :word
    ]
    first use reduce [:w] reduce [reduce [:w]]
]
```

Voyons si cette fonction fait ce pour quoi elle est créée :

```
word1: 'a ; == a
word2: nonsame word1 ; == a
equal? spelling word1 spelling word2 ; == true
set word1 1
set word2 2
get word1 ; == 1
get word2 ; == 2
same? word1 word2 ; == false
```

Le test démontre que WORD1 et WORD2 ont la même orthographe. Leur liaisons sont différentes, elles peuvent référer à des valeurs différentes en même temps. Un autre test :

```
word1: /a ; == /a
word2: nonsame word1 ; == a
same? word1 word2 ; == false
```

Observation (Orthographe et liaison) :

Deux mots avec la même orthographe ont la même liaison si et seulement s'ils sont les mêmes.

13 Comparaison de liaison

Nous en connaissons maintenant suffisamment afin de comparer les liaisons des mots :

```
equal-binding?: func [
    {word1 et word2 ont-ils la même liaison ?}
    word1 [any-word!]
    word2 [any-word!]
] [
    if not has-context? :word2 [return not has-context? :word1]
    same? :word1 bindany nonsame :word1 :word2
]
```

14 Comparaison de mots

Observation (Ressemblance des mots) :

Deux mots sont les mêmes, si leur orthographe et leur liaison sont égales.

Observation (Ressemblance du référentiel) :

Deux mots utilisent une même référence s'ils sont égaux et si leurs liaisons sont égales. Ce cas traite de la situation de deux alias qui sont différents mais qui utilisent une référence.

```
same-references?: func [
    word1 [any-word!]
    word2 [any-word!]
] [
    found? all [
        equal? :word1 :word2
        equal-binding? :word1 :word2
    ]
]
```

15 Visualisations et contexte

Définition :

Une collection non-vide de mots est une visualisation (ou un contexte), si :

- tous les mots de la collection ont une liaison égale
- tout mot n'appartenant pas à la collection a une liaison différente de celle des mots dans la collection

Observation (Hiérarchie de contexte) :

De cette définition, nous pouvons déduire qu'aucune visualisation ne peut être une partie d'une autre visualisation.

Observation (Visualisation du contexte d'un mot) :

Pour chaque mot, il existe une visualisation qui contient ce mot. J'appellerai celle-ci la visualisation du contexte du mot.

Preuve :

Prenons un KNOWN-WORD. La visualisation de ce contexte sera une collection de tous les mots qui ont une liaison égale à celle que le KNOWN-WORD a.

La fonction suivante, basée sur le code de Thomas Jensen, illustre la preuve, bien qu'elle n'est pas "complète" :

```
visualize-context: function [
    known-word [any-word!]
] [result] [
    if not has-context? :known-word [return first system/words]
    result: make block! 0
    foreach word first system/words [
        if not same? word bindany word :known-word [
            append result word
        ]
    ]
    result
]
```

16 Visualisation d'un objet

Définition :

La visualisation d'un objet O est la visualisation du (in o 'self) contexte du mot.

Comme la fonction BIND, la fonction IN n'est pas aussi générale que nous souhaiterions l'avoir. Mais il est possible de modifier les spécifications de la fonction IN afin de poivoir utiliser un ANY-WORD comme argument :

```
inany: function [
    {
        returns a word in OBJECT
        having equal spelling and equal type
        as WORD
    }
    object [object!]
    word [any-word!]
] [wordt words spec result] [
    spec: third :in
    words: next find spec 'word
    wordt: first words
    change/only words reduce [any-word!]
    result: in object :word
    change/only words wordt
    :result
]
```

Observation (Visualisation d'un objet) :

La visualisation d'un objet O est la collection de tous les mots qui peuvent être obtenus comme résultat de l'expression (inany o :word).

Définition :

A la place de dire "un mot est dans la visualisation d'un objet", je dirai "un mot est dans un objet".

Observation (Mots d'objet) :

Pour chaque objet O et pour chaque mot que WORD contient, ce WORD est dans O si et seulement si l'expression (same ? :word inany o :word) est vraie.

Définition :

J'appellerai "contexte global" l'objet REBOL/WORDS. Les mots contenus dans ce contexte global seront des mots globaux.

La fonction suivante nous aide à trouver si un mot se trouve dans le contexte global :

```
global?: func [
    {trouve si un mot est global}
    word [any-word!]
] [
    same? :word inany rebol/words :word
]
```

Observation (Visualisation simplifiée) :

Le bloc obtenu comme le résultat de l'expression (first object) est une visualisation simplifiée de l'OBJECT. Au contraire de la visualisation complète, il contient des mots non-liés. De plus, il ne contient pas tous les alias de ses mots et ne contient que des mots du type WORD!

Illustration :

```
alias 'rebol "rebellious"
o: make object! [rebellious: 1]
first o ; == [self rebellious]
not has-context? first first o ; == true
in o 'rebol ; == rebol
```

La tâche inverse à la recherche de la visualisation d'un objet est de trouver l'objet dans lequel se trouve un mot. La fonction suivante, basée sur l'idée de Romano Paolo Tenca retourne l'objet régulier (objet pour lequel l'attribut SELF est l'objet lui-même) dans lequel le mot se trouve. Dans le cas d'un objet irrégulier, la fonction retourne NONE.

```
in-object?: function [
    {trouve l'objet régulier dans lequel le mot se trouve}
    word [any-word!]
] [self] [
    all [
        has-context? :word
        not same? 'self self: bindany 'self :word
        object? set/any 'self get/any self
        same? :word inany self :word
        self
    ]
]
```

17 Les fonctions MAKE, TO et les mots de Rebol

Observation (MAKE, TO et mots globaux) :

Les mots créés par MAKE WORD !, MAKE SET-WORD !, MAKE GET-WORD !, MAKE LIT-WORD !, MAKE REFINEMENT !, TO WORD !, TO SET-WORD !, TO GET-WORD !, TO LIT-WORD !, TO

REFINEMENT ! sont globaux.

Illustration :

```
global? make word! first first rebol/words ; == true
global? to word! first first rebol/words ; == true
```

Observation (Elargissement automatique) :

Le contexte global est élargi quand les fonctions MAKE, TO et LOAD créent un nouveau mot. Par contre, la fonction BIND n'élargi pas le contexte global, comme nous l'avons décrit dans l'Observation (Liaison inefficace). Identiquement, la fonction IN n'élargi pas le contexte global.

Observation (MAKE, TO and Unbound Words) : All words contained in the result of MAKE BLOCK ! and TO BLOCK ! and in its subblocks are unbound, if the SPEC argument is a string.

Illustration :

```
has-context? first make block! "unbound" ; == false
has-context? first first first make block! "[[unbound-too]]" ; == false
```

18 Mots locaux, visualisations locales

Définition :

J'appelle mots locaux les mots qui sont liés et qui ne sont pas globaux.

```
local?: func [
    {find out, if a given word is local}
    word [any-word!]
] [
    not any [
        not has-context? :word
        global? :word
    ]
]
```

Définition :

Une visualisation est appelée "visualisaion d'un contexte" local si elle ne contient que des mots locaux.

Observation (Visualisdation de contexte locaux) :

Exception faite des visulisations de contextes globaux et de la collection de tous les mots non liés, chaque visualisation est une visualisation de de contexte local.

Observation (Elargissement des contexte locaux) :

Les contextes locaux ne peuvent pas être élargis.

19 Visualisation du contexte des fonctions et de USE

En plus des visualisations d'objets définis par les utilisateurs, il existe des visualisation de contexte de fonction et des visualisation de contexte de USE (NdT : use est mis en majuscule pour la clarté du propos). Une différence entre la visualisation d'un objet et celle des fonctionset de USE, est que ces dernières ne contiennent pas de mot égal au mot 'self.

20 Liaison calculées

Explorons le comportement du code REBOL.

```
code-string: {
    checkpoint 1
    'f 'g 'h
    use [g h] [
        checkpoint 2
        'f 'g 'h
        use [h] [
            checkpoint 3
            'f 'g 'h
        ]
    ]
}
```

La fonction CHECKPOINT va colorier le code de la manière suivante : les mots non liés seront en jaune, les mots globaux en bleu, les mots liés par le premier USE seront en rouge et ceux liés au contexte créé par le second USE seront en magenta :

```
checkpoint: func [i] [
    append result rejoin ["^/^/    Checkpoint " i "^/    "]
    parse code rule: [
        (append result #"[")
        any [
            (append result #" ")
            set word any-word! (
                any [
                    if not has-context? :word [
                        append result
```

```
                        append result mold :word
                        append result
                    ]
                    if global? :word [
                        append result
                        append result mold :word
                        append result
                    ]
                    if equal-binding? :word code/8/4 [
                        append result
                        append result mold :word
                        append result
                    ]
                    if equal-binding? :word code/8/8/5 [
                        append result
                        append result mold :word
                        append result
                    ]
                ]
            ) | into rule | set word skip (
                append result mold :word
            )
        ]
        (append result #"]")
    ]
]
```

Ceci est la description de comment ce code REBOLest interprété :

```
;L'interpréteur REBOL crée d'abord un block de code
code: make block! code-string
result: ""
; marquons le code
checkpoint 0
; ensuite, le code est lié au contexte global
code: bind code 'rebol
;et enfin, le code est interprété
do code
append result
print result
```

Le résultat sera :

Checkpoint 0 [checkpoint 1 'f 'g 'h use [g h] [checkpoint 2 'f 'g 'h use [h] [checkpoint 3 'f 'g 'h]]]
Checkpoint 1 [checkpoint 1 'f 'g 'h use [g h] [checkpoint 2 'f 'g 'h use [h] [checkpoint 3 'f 'g 'h]]]
Checkpoint 2 [checkpoint 1 'f 'g 'h use [g h] [checkpoint 2 'f 'g 'h use [h] [checkpoint 3 'f 'g 'h]]]
Checkpoint 3 [checkpoint 1 'f 'g 'h use [g h] [checkpoint 2 'f 'g 'h use [h] [checkpoint 3 'f 'g 'h]]]

Ceci prouve que :

- au point de contrôle initial, tous les mots dans CODE ne sont pas liés
- au premier point de contrôle, tous les mots sont globaux
-

au second point de contrôle, le premier appel à USE a remplacé tous les mots 'g et 'h de son bloc de corps et de ses sous-blocs par des mots locaux de USE1

- au troisième point de contrôle, le second USE a remplacé le 'h dans le bloc le plus intérieur par le mot local de USE2

Observation (Liaison calculée) :

Durant l'interprétation, la liaison des mots REBOL contenus dans le code est changée (les mots sont remplacés par des mots avec une liaison différente) jusqu'à ce qu'il soient correctement liés et évalués. C'est pourquoi le créateur de REBOL nomme ce comportement la liaison calculée.

21 Portée Définitionnelle

Nous pouvons observer quelque chose comme une "hiérarchie de portée" durant l'exécution du code (voir la section précédente). Comme nous l'vaons vu, ceci n'est qu'un effet collatéral de la liaison calculée.

Avec l'aide de la liaison calculée, nous pouvons simplement créer un extrait de code qui ne présente pas cette "hiérarchie de portée".

```
; crée un bloc CODE-BLK contenant un mot 'a
code-blk: copy [a]
a: 12
;ajoute maintenant un autre mot 'a dans CODE-BLK
make object! [append code-blk 'a a: 13]
code-blk ; == [a a]
; teste si CODE-BLK contient des mots égaux
equal? first code-blk second code-blk ; == true
;teste  si CODE-BLK est un bloc avec une portée mixte
same? first code-blk second code-blk ; == false
```

Le code ci-dessus peut servir à prouver qu'il n'existe pas de "contexte courant" en REBOL.

22 USE

Afin d'être aussi précis que possible, nous allons écrire la définition de la fonction USE en Rebol.

La fonction suivante crée un nouveau contexte inaccessible et retourne un mot le représentant.

```
sim-make-context: func [
    words [block!]
] [
    first use words reduce [reduce [first words]]
]
```

La description de USE :

```
sim-use: function [
    "simulation de USE"
    [throw]
    words [block!] "Local word(s) to the block"
    body [block!] "Block to evaluate"
] [new-context] [
    if word? words [words: reduce [words]]
    if not empty? words [
        ; crée un nouveau contexte
        new-context: sim-make-context words
        ; lie le corps au nouveau contexte
        bind body new-context
    ]
    do body
]
```

Observation (SIM-USE et BODY) :

SIM-USE modifie son argument BODY quand il se lie au nouveau contexte. Si nous souhaitons laisser l'argument BODY inchangé, nous devons utiliser BIND/COPY au lieu de BIND.

Comparons le comportement de SIM-USE et celui de USE :

```
body: ['a]
body-copy: copy body
same? first body first body-copy ; == true
use [a] body
same? first body first body-copy ; == false
```

Comme nous venons de la montrer, la même chose est vraie pour SIM-USE et pour le USE original. Nous pouvons constater que la simulation est si précise qu'elle nous a aidé à révéler un bogue dans le code suivant :

```
f: func [x] [
    use [a] [
        either x = 1 [
            a: "OK"
            f 2
            a
        ] [
            a: "BUG!"
            "OK"
        ]
    ]
]
f 1 ; == "BUG!"
```

Explication/correction :

La propriété USE observée change le corps de la fonction F durant la seconde exécution du USE. Après cette modification, il ne contient plus le mot 'a qui était déjà mis sur "OK" pendant le premier appel de F. Au lieu de cela, il contient seulement le mot 'a qui était mis sur "BUG !" durant le second appel de F.

Si nous pouvons conserver le corps de F, alors nous obtenons le comportement correct :

```
f: func [x] [
    use [a] copy/deep [
        either x = 1 [
            a: "OK"
            f 2
            a
        ] [
            a: "BUG!"
            "OK"
        ]
    ]
]
f 1 ; == "OK"
```

Une autre manière de corriger ce comportement est d'utiliser notre propre version de USE, qui ne modifiera pas son propre argument BODY :

```
nm-use: func [
    {
        Défini les mots locaux au bloc.
        Ne modifie pas l'argument BODY.
    }
    [throw]
    words [block!] {Local words to the block}
    body [block!] {Block to evaluate}
] [
    use words copy/deep body
]
```

23 MAKE OBJECT !

Nous utiliserons la même méthode que pour la description du USE. La première partie de la simulation sera une fonction créant un objet "blanc", c'est à dire un objet contenant les mots locaux désirés, mais pas mis à un contenu. Parce qu'un telle fonction n'existe pas en REBOL, nous allons créer la nôtre pour le besoin de la simulation :

```
blank-object: function [
    {crée un objet blanc}
    set-words [block!]
] [object] [
   unset in object: context compose [
       return self (set-words)
   ] 'self
   object
```

```
]
```

En plus, nous avons besoin d'une fonction qui évalue l'argument SPEC de la même manière que MAKE OBJECT ! le fait, c'est à dire en interceptant RETURN, THROW ou BREAK :

```
spec-eval: func [
    {évalue SPEC comme MAKE OBJECT! le fait}
    spec [block!]
] [
    any-type? catch [loop 1 spec]
]
```

La simulation MAKE OBJECT ! :

```
sim-make-object!: function [
    {MAKE OBJECT! simulation}
    spec [block!]
] [set-words object sw] [
    ; trouve tous les set-words dans SPEC
    set-words: copy []
    parse spec [
        any [
            copy sw set-word! (append set-words sw) |
            skip
        ]
    ]
    ; crée un objet blanc avec les mots locaux désirés
    object: blank-object set-words
    ; fait référer le 'self de l'objet vers l'objet
    object/self: object
    ; lie la SPEC à l'objet blanc
    bind spec in object 'self
    ; evaluation
    spec-eval spec
    ; retourne la valeur de 'self comme résultat
    return get/any in object 'self
]
```

Observation (SIM-MAKE-OBJECT ! et SPEC) :

SIM-MAKE-OBJECT ! modifie son argument SPEC quand il est lié au nouveau contexte. Si nous désirons laissé l'argiment SPEC inchangé, nous devons utiliser le BIND/CPOPY à la place du BIND.

Ce comportement nous méne à découvrir un bug similaire à celui décrit dans la section traitant du USE :

```
f: func [x] [
    get in sim-make-object! [
        a: "OK"
        if x = 1 [
            a: "BUG!"
```

```
            f 2
            a: "OK"
        ]
    ] 'a
]
f 1 ; == "BUG!"
```

L'explication et la correction son similiare à la fonction USE. Le mot a : dans a : "OK" créé après l'appel récursif à F et lié à l'objet F créé le premier, a été remplacé par le mot a : lié à l'objet F créé pendant l'appel récursif.

Cela signifie que l'expression a : "OK" n'a pas d'effet sur l'objet F créé le premier, et donc la dernière valeur de 'a est retenue, c'est à dire "BUG". Si nous conservons le coprs de F, nous pouvons obtenir le comportement correct :

```
f: func [x] [
    get in make object! copy/deep [
        a: "OK"
        if x = 1 [
            a: "BUG!"
            f 2
            a: "OK"
        ]
    ] 'a
]
f 1 ; == "OK"
```

Comme vous pouvez le constater, le code ci-dessus copie le bloc BODY avant de le lier au contexte. Des bugs semblables furent découverts quand le "deep copy" n'était pas utilisé lorsque la fonction FUNC créait des fonctions REBOL.

24 MAKE PROTO

Ceci est la simulation de la situation où la fonction MAKE obtient n prototype de l'objet à créer.

tout d'abord, nous avons besoin d'un fonction BIND-LINE particulière :

```
specbind: function [
    {lie seulement les known-words}
    block [block!]
    known-words [block!]
] [p w bind-one kw] [
    bind-one: [
        p:
        [
            copy w any-word! (
                if kw: find known-words first w [
                    change p bind w first kw
                ]
            ) | copy w [path! | set-path! | lit-path!] (
```

```
                if kw: find known-words first first w [
                    change p bind w first kw
                ]
            ) | into [any bind-one] | skip
        ]
    ]
    parse block [any bind-one]
    block
]
```

Et voici la simulation :

```
make-proto: function [
    {simulation MAKE PROTO}
    proto [object!]
    spec [block!]
] [set-words object sw word value spc body pwords] [
    set-words: copy []
    ; prend les mots locaux de proto
    foreach word next first proto [
        append set-words to set-word! word
    ]
    ; prend tous les set-words de SPEC
    parse spec [
        any [
            copy sw set-word! (append set-words sw) |
            skip
        ]
    ]
    ; crée un objet blanc avec les mots locaux désirés
    object: blank-object set-words
    ; fait pointer le 'self de l'objet vers l'objet
    object/self: object
    ; cpie le contenu de proto
    pwoords: bind first proto in object 'self
    repeat i (length? first proto) - 1 [
        word: pick next first proto i
        any-type? set/any 'value pick next second proto i
        any [
            all [string? get/any 'value set in object word copy value]
            all [
                block? get/any 'value
                value: specbind copy/deep value pwords
                set in object word value
            ]
            all [
                function? get/any 'value
                spc: load mold third :value
                body: specbind copy/deep second :value pwords
                set in object word func spc body
            ]
```

```
            any-type? set/any in object word get/any 'value
        ]
    ]
    ; lie PEC
    bind spec in object 'self
    ; evalue
    spec-eval spec
    ; retourne la valeur de 'self comme résultat
    return get/any in object 'self
]
```

25 Fonctions avec une manière simili-MAKE-OBJECT de traiter les mots locaux

Avant de nous lancer dans l'évaluation de fonctions, nous pouvons nous demander si nous pouvons utiliser la même méthode de traitement des mots locaux que celle utilisée par la fonction CONTEXT.

La réponse est positive et la fonction le démontrant est définie ci-dessous.

D'abord, créons une fonction pouvant extraire tous les mots locaux des SPEC d'une fonction :

```
locals?: function [
    {extrait le mots locaux des specs}
    spec [block!]
] [locals] [
    locals: make block! length? spec
    parse spec [
        any [
            set item any-word! (
                append locals to word! :item
            ) | skip
        ]
    ]
    locals
]

lfunc: function [
    {défini une fonction avec des set-word! implicitement locaux}
    [catch]
    spec [block!]
    body [block!] "La bloc de corps de la fonction"
] [vars sw] [
    vars: copy []
    parse body [
        any [
            set sw set-word! (append vars to word! :sw) |
            skip
        ]
    ]
```

```
    vars: exclude vars locals? spec
    spec: head insert insert tail copy spec /local vars
    throw-on-error [make function! spec body]
]
```

26 Modèle de fonction REBOL

Notre modèle de fonction Rebol sera un objet Rebol SIM-FUNCTION! disposant des attributs nécessaires. Les attributs absolument nécessaires sont SPEC et BODY.

Afin de modéliser précisément le comportement des fonctions Rebol, notre SIM_FUNCTION ! nécessite un autre attribut nommé CONTEXT et deux autres attributs : STACK et RECURSION-LEVEL pour rendre le comportement des fonctions Rebol durant les appels récursifs :

```
sim-function!: make object! [
    spec: none
    body: none
    context: none
    stack: none
    recursion-level: none
]
```

27 Modèle de fonction Rebol FUNC

Cette fonction reçoit les attributs SPEC et BODY et crée une nouvelle SIM-FUNCTION! et l'initialise :

```
sim-func: function [
    {crée une Sim-function!}
    spec [block!]
    body [block!]
] [result aw] [
    ; un nouveau SIM-FUNCTION! est créé
    result: make sim-function! []
    ; SPEC et BODY sont copiés en profondeur
    result/spec: copy/deep spec
    result/body: copy/deep body
    ; les mots de contexte sont récupérés de SPEC
    result/context: locals? spec
    if not empty? result/context [
        ; le contexte est créé
        result/context: sim-make-context result/context
        ; le corps de fonction est lié au contexte
        bind result/body first result/context
    ]
    ; une pile vide est créée
    result/stack: make block! 0
    ; RECURSION-LEVEL est mis à zéro
    result/recursion-level: 0
```

```
    result
]
```

28 Modèle d'évaluation de fonction

La première action de l'évaluation d'une fonction est de collecter les arguments (évalués et non évalués) de la fonction et de contrôler leur type etc... Nous sauterons cette partie car cela compliquerait inutilement les choses (beaucoup de méthodes différentes pour évaluer, contrôler les arguments, faire intervenir des arguments optionnels etc...). Cet effort supplémentaire ne semble ni nécessaire ni indiqué dans notre cas.

Notre fonction d'évaluation obtient une SIM-FUNCTION ! ensemble avec un bloc de valeurs qu'elle stockera dans son contexte de mots locaux (entre autres toutes les valeurs de ses arguments, arguments optionnels, raffinements et autres).

Nous allons modéliser le cas le plus fréquent d'une fonction, sans l'attribut THROW.

La première partie de notre modèle exécute le BODY :

```
exec: func [body] [do body]
```

La simulation :

```
sim-evaluate: function [
    {evalue une sim-function!}
    sim-f {the evaluated sim-function!}
    values [block!] {the supplied values}
] [old-values result] [
    ; repère l'appel récurcif
    if (sim-f/recursion-level: sim-f/recursion-level + 1) > 1 [
        ; récupére les anciennes valeurs des mots du contexte
        old-values: make block! length? sim-f/context
        foreach word sim-f/context [
            insert/only tail old-values get/any word
        ]
        ; met ces anciennes valeurs sur la pile
        sim-f/stack: head insert/only sim-f/stack old-values
    ]
    ; stocke les valeurs données dans des mots de contextes locaux
    set/any sim-f/context values
    ; execute le corps de la fonction
    error? set/any 'result exec sim-f/body
    ; rétablit les valeurs précédentes à partir de la pile, si nécessaire
    if (sim-f/recursion-level: sim-f/recursion-level - 1) > 0 [
        set/any sim-f/context first sim-f/stack
        ; termine la pile
        sim-f/stack: remove sim-f/stack
    ]
    return get/any 'result
```

```
]
```

Notre modèle utilise juste un contexte pour toute la durée de vie de la SIM-FUNCTION !, sans devoir changer les liaisons de son BODY. J'appelle ce comportement la Rustine de la Récursion Dynamique (NdT *Dynamic Recursion Patch*, je n'ai pas trouvé de meilleure traduction).

Quelques tests :

```
probeblk: func [] [
    prin mold blk
    prin ": "
    print mold reduce blk
]

recfun: sim-func [x] [
    append blk 'x
    either x = 2 [
        probeblk
    ] [
        sim-evaluate recfun [2]
    ]
]
blk: copy []
sim-evaluate recfun [1] ; [x x]: [2 2]
probeblk ; [x x]: [1 1]
```

Si nous comparons le comportement simulé à celui de fonctions Rebol réelles, nous obtenons :

```
recfun: func [x] [
    append blk 'x
    either x = 2 [
        probeblk
    ] [
        recfun 2
    ]
]
blk: copy []
recfun 1 ; [x x]: [2 2]
probeblk ; [x x]: [1 1]
```

Ceci démontre que la simulation est vraiment précise et que les fonctions Rebol utilisent également la Rustine de la Récursion Dynamique (NdT *Dynamic Recursion Patch*).

Bien que la Rustine de la Récursion Dynamique (NdT *Dynamic Recursion Patch*) puisse accélérer l'évaluation dans certains cas, elle présente des désavantages :

```
f-returning-x: func [x] [
    func [] [x]
]
f-returning-ok: f-returning-x "OK"
f-returning-ok ; == "OK"
f-returning-bug: f-returning-x "BUG!"
; so far so good, but now:
```

```
f-returning-ok ; == "BUG!"
```

29 Fonctions avec liaisons calculées (Fin)

Comme nous l'avons vu ci-dessus, les liaisons calculées ont leurs mérites, alors que la Rustine de la Recursion Dynamique (NdT *Dynamic Recursion Patch*) n'est pas idéale. Les résultats m'ont inspiré l'implémentation des Fonctions avec liaisons calculées et leur comparaison avec le comportement des Rustines de Récursion Dynamique (NdT *Dynamic Recursion Patch*)

Les fonctions avec liaisons calculées créent un nouveau contexte à chaque fois qu'elles sont appelées, et lient leur corps en conséquence. Nous pouvons utiliser une partie de la dernière simulation afin de les implémenter :

```
cfunc: function [
    {crée une conclusion}
    [catch]
    spec [block!]
    body [block!]
] [locals in-new-context body2 i item] [
    locals: copy []
    body2: reduce [
        'set [new-locals new-body] 'use 'locals 'copy/deep reduce [
            reduce [locals copy/deep body]
        ]
    ]
    i: 1
    parse spec [
        any [
            set-word! | set item any-word! (
                append locals to word! :item
                append body2 compose [
                    error? set/any pick new-locals (i) get/any pick locals (i)
                ]
                i: i + 1
            ) | skip
        ]
    ]
    in-new-context: function [
        {do body with locals in new context}
        [throw]
        locals
    ] [new-body new-locals] append body2 [do new-body]
    throw-on-error [
        func spec reduce [:in-new-context locals]
    ]
]
```

Le premier test :

```
recfun: cfunc [x] [
```

```
    append blk 'x
    either x = 2 [
        probeblk
    ] [
        recfun 2
    ]

]
blk: copy []
recfun 1 ; [x x]: [1 2]
probeblk ; [x x]: [1 2]
```

Ce qui donne quand même un meilleur résultat. Le second test :

```
f-returning-x: cfunc [x] [
    func [] [x]
]
f-returning-ok: f-returning-x "OK"
f-returning-ok ; == "OK"
f-returning-bug: f-returning-x "BUG!"
; so far so good, but now:
f-returning-ok ; == "OK"
```

La fin.