The REBOL Documentation Project

-- FR - Documentation REBOL - Manuels --

Manuels

Manuel de l'utilisateur - Annexe 2 - Les Erreurs

Philippe Le Goff

Première publication : 8 novembre 2005, et
mis en ligne le mardi 8 novembre 2005

Résumé :

Ce document est la traduction française de l'Annexe 2 du User Guide de REBOL/Core, qui concerne les erreurs.

Ce document est la traduction française de l'Annexe 2 du User Guide de REBOL/Core, qui concerne les erreurs.

1 Historique de la traduction

Date	Version	Commentaires	Auteur	Email
19 Septembre 2005 19:56	1.0.0	Traduction initiale	Philippe Le Goff	lp—legoff—free—fr

2 Présentation

Les erreurs sont des exceptions qui se produisent lors de certaines situations anormales. Ces situations peuvent se rencontrer pour des erreurs de syntaxe jusqu'à des erreurs liées aux accès réseau ou fichiers.

Voici quelques illustrations :

```
12-30
** Syntax Error: Invalid date -- 12-30.
** Where: (line 1) 12-30
1 / 0
** Math Error: Attempt to divide by zero.
** Where: 1 / 0
read %nofile.r
** Access Error: Cannot open /example/nofile.r.
** Where: read %nofile.r
```

Les erreurs sont traitées au sein de langage comme des valeurs de datatype **error !**. Une erreur est un objet qui, s'il est évalué, affichera un message d'erreur et stoppera le programme. Vous pouvez aussi capturer et manipuler des erreurs dans vos scripts. Les erreurs peuvent être passées à des fonctions, ou récupérées depuis des fonctions, et affectées à des variables.

3 Catégories d'erreurs

Il y a plusieurs catégories d'erreurs.

3.1 Erreurs de syntaxe

Les erreurs de syntaxe surviennent quand un script REBOL utilise une syntaxe incorrecte. Par exemple, si un crochet fermant est manquant ou qu'une apostrophe ne ferme pas une chaîne de caractère, une erreur de syntaxe va être générée. Ces erreurs se produisent uniquement durant le chargement ou l'évaluation d'un fichier ou d'une chaîne.

3.2 Erreurs de script

Les erreurs de script sont en général des erreurs d'exécution. Par exemple, un argument invalide fourni à une fonction causera une erreur de script.

3.3 Erreurs mathématiques

Les erreurs mathématiques se produisent quand une opération mathématique ne peut être effectuée. Par exemple, si une division par zéro est tentée, une erreur se produit.

3.4 Erreurs d'accès

Les erreurs d'accès apparaissent quand un problème intervient lors de l'accès à un fichier, à un port ou au réseau. Par exemple, une erreur d'accès se produira si vous essayez de lire un fichier qui n'existe pas.

3.5 Erreurs utilisateur

Les erreurs utilisateurs sont générées explicitement par un script en créant une valeur d'erreur et en la retournant.

3.6 Erreurs internes

Les erreurs internes sont produites par l'interpréteur REBOL.

4 Capture des erreurs

Vous pouvez récupérer les erreurs avec la fonction **try**. La fonction **try** est similaire à la fonction **do**. Elle évalue un bloc, mais renvoie toujours une valeur, même quand une erreur se produit. Si aucune erreur n'est apparue, la fonction **try** retourne la valeur du bloc.

Par exemple :

```
print try [100 / 10]
10
```

Quand une erreur apparaît, **try** renvoie cette erreur. Si vous écrivez :

```
print try [100 / 0]
** Math Error: Attempt to divide by zero.
** Where: 100 / 0
```

l'erreur est renvoyée par la fonction **try** et la fonction **print** n'est pas effectuée.

Pour manipuler les erreurs dans un script, vous devez empêcher REBOL d'évaluer l'erreur. Vous pouvez éviter l'évaluation d'une erreur en la passant à une fonction. Par exemple, la fonction **error ?** renverra la valeur **true** si son argument est bien une erreur :

```
print error? try [100 / 0]
true
```

Vous pouvez aussi afficher le type de données de la valeur renvoyée par **try** :

```
print type? try [100 / 0]
error!
```

La fonction **disarm** convertit une erreur en un objet-erreur qui peut être examiné. Dans l'exemple ci-dessous, la variable *error* fait référence à un objet renvoyé par **disarm** :

```
error: disarm try [100 / 0]
```

Quand une erreur est passée à la fonction **disarm**, elle est transformée en objet (type de données **object !**) et n'est plus du type de données **error !**. L'évaluation de cet objet-erreur devient possible :

```
probe disarm try [100 / 0]
make object! [
    code: 400
    type: 'math
    id: 'zero-divide
```

```
    arg1: none
    arg2: none
    arg3: none
    near: [100 / 0]
    where: none
]
```

Les valeurs d'erreur peuvent être affectées à un mot avant d'être passées à la fonction **disarm**. Pour attribuer un mot à une erreur, celui-ci doit être précédé par une fonction afin d'éviter que l'erreur soit réemployée ailleurs. Par exemple :

```
disarm err: try [100 / 0]
```

Associer l'erreur à un mot vous permet de récupérer sa valeur plus tard. L'exemple ci-dessous illustre le cas où il peut y avoir ou non une erreur :

```
either error? result: try [100 / 0] [
    probe disarm result
][
    print result
]
```

5 L'objet erreur

L'objet erreur montré précédemment a la structure :

```
make object! [
    code: 400
    type: 'math
    id: 'zero-divide
    arg1: none
    arg2: none
    arg3: none
    near: [100 / 0]
    where: none
]
```

avec les champs suivants :

code : *Le numéro du code d'erreur. Obsolète, ne devrait plus être utilisé.*

type : *Le champ **type** identifie la catégorie d'erreur. Il s'agit toujours d'un mot comme syntax, script, math, access, user, ou internal.*

id : *la champ id est le nom de l'erreur, sous forme de mot REBOL. Il identifie spécifiquement l'erreur au sein de sa catégorie.*

arg1 : *Ce champ contient le premier argument du message d'erreur. Par exemple, il peut inclure le*

type de donnée de la valeur ayant causé l'erreur.

arg2 : *Ce champ contient le second argument du message d'erreur.*

arg3 : *Ce champ contient le troisième argument du message d'erreur.*

near : *Le champ near est un morceau de code censé indiquer l'endroit où l'erreur s'est produite.*

where : *Ce champ est un champ réservé.*

Vous pouvez écrire du code qui va contrôler chacun des champs de l'objet erreur. Dans cet exemple, l'erreur est affichée seulement si le champ ***id*** indique une division par zéro :

```
error: disarm try [1 / 0]
if error/id = 'zero-divide [
    print {It is a Divide by Zero error}
]
It is a Divide by Zero error
```

Le champ ***id*** de l'erreur fournit aussi le bloc qui sera affiché par l'interpréteur. Par exemple :

```
error: disarm try [print zap]
probe get error/id
[:arg1 "has no value"]
```

Ce bloc est défini par l'objet **system/error**.

6 Générer des erreurs

Il est possible de fabriquer des erreurs pour votre propre usage. Le moyen le plus simple est de les générer en faisant appel à la fonction **make**. Voici un exemple :

```
make error! "this is an error"
** User Error: this is an error.
** Where: make error! "this is an error"
```

N'importe quelle erreur existante peut être générée en la fabriquant avec un argument de type **block !** . Ce bloc doit contenir le nom de la catégorie d'erreur, et la désignation spécifique (id) de celle-ci. Les arguments arg1, arg2, et arg3 définissent l'erreur dans l'objet erreur créé.

Voici un exemple :

```
make error! [script expect-set series! number!]
** Script Error: Expected one of: series! - not: number!.
** Where: make error! [script expect-set series! number!]
```

NdT : ici *script* fait référence à la catégorie, *expect-set* à l'id, et *series !*, *number !* sont les arguments.

Les erreurs personnalisées peuvent être incluses dans l'objet **system/error**, et la catégorie "user". Ceci est réalisé en fabriquant une nouvelle catégorie "user" avec de nouvelles entrées. Ces entrées sont utilisées lorsque des erreurs se produisent. Pour illustrer ceci, l'exemple suivant ajoute une erreur dans la catégorie "user".

```
system/error/user: make system/error/user [
    my-error: "a simple error"
]
```

A présent, une erreur peut être produite, et utiliser le message de l'id *my-error* :

```
if error? err: try [
    make error! [user my-error]
] [probe disarm err]
make object! [
    code: 803
    type: 'user
    id: 'my-error
    arg1: none
    arg2: none
    arg3: none
    near: [make error! [user my-error]]
    where: none
]
```

Pour créer des erreurs avec plus d'informations, définissez une erreur qui utilise les données courantes lorsqu'elle est générée. Ces données seront incluses dans l'objet erreur, et affichées en tant qu'éléments de cet objet. Par exemple, pour utiliser les trois champs d'arguments dans l'objet erreur :

```
system/error/user: make system/error/user [
    my-error: [:arg1 "doesn't go into" :arg2 "using" :arg3]
]

if error? err: try [
    make error! [user my-error [this] "that" my-function]
] [probe disarm err]
make object! [
    code: 803
    type: 'user
    id: 'my-error
    arg1: [this]
    arg2: "that"
    arg3: 'my-function
    near: [make error! [user my-error [this] "that" my-function]]
    where: none
]
```

Le message d'erreur produit pour my-error peut être affiché, sans bloquer l'exécution du script :

```
disarmed: disarm err
```

```
print bind (get disarmed/id) (in disarmed 'id)
this doesn't go into that using my-function
```

Une nouvelle catégorie d'erreur peut aussi être créée si vous avez besoin de grouper un ensemble spécifique d'erreurs :

```
system/error: make system/error [
    my-errors: make object! [
    code: 1000
    type: "My Error Category"
    error1: "a simple error"
    error2: [:arg1 "doesn't go into" :arg2 "using" :arg3]
    ]
]
```

Le type défini l'objet erreur correspond au type d'erreur affiché quand l'erreur se produit. L'exemple suivant illustre la génération d'erreurs pour deux sortes d'erreurs (error1 et error2) dans la catégorie my-error .

Création d'une erreur à partir d'error1. Cette erreur ne requiert aucun argument.

```
disarmed: disarm try [make error! [my-errors error1]]
print get disarmed/id
a simple error
```

La création d'une erreur à partir d'error2 nécessite, elle, trois arguments :

```
disarmed: disarm try [
make error! [my-errors error2 [this] "that" my-function]]
print bind (get disarmed/id) (in disarmed 'id)
this doesn't go into that using my-function
```

Et pour finir, la description de la catégorie définie dans my-errors peut être obtenue ainsi :

```
probe get in get disarmed/type 'type
My Error Category
```

7 Messages d'erreurs

La liste ci-dessous présente toutes les erreurs définies dans l'objet **system/error** :

7.1 Erreur de syntaxe

7.1.1 invalid

Les données ne peuvent pas être transposées en un type valide de données REBOL. En d'autres termes, une valeur mal formée a été évaluée.

Message :

```
["Invalid" :arg1 "--" :arg2]
```

Exemple :

```
filter-error try [load "1024AD"]
** Syntax Error: Invalid integer -- 1024AD
** Where: (line 1) 1024AD
```

7.1.2 missing

Un bloc, une chaîne de caractère ou une expression à parenthèses (paren !) souffre d'un défaut : Une parenthèse, un crochet, un guillemet, ou une accolade manque.

Message :

["Missing" :arg2 "at" :arg1]

Exemple :

```
filter-error try [load "("]
** Syntax Error: Missing ) at end-of-script
** Where: (line 1) (
```

7.1.3 header

Une évaluation d'un fichier en tant que script REBOL a été essayée, mais le fichier ne présente pas d'en-tête REBOL (header).

Message :

```
Script is missing a REBOL header
```

Exemple :

```
write %no-header.r {print "data"}
filter-error try [do %no-header.r]
** Syntax Error: Script is missing a REBOL header
** Where: do %no-header.r
```

3.2 Erreurs de script

7.2.4 no-value

Une évaluation a été essayée sur un mot qui n'est pas défini.

Message :

```
[:arg1 "has no value"]
```

Example :

```
filter-error try [undefined-word]
** Script Error: undefined-word has no value
** Where: undefined-word
```

7.2.5 need-value

Un essai de définir un mot sans rien a été fait. Un set-word (mot à définir) a été utilisé sans argument.

Message :

```
[:arg1 "needs a value"]
```

Exemple :

```
filter-error try [set-to-nothing:]
** Script Error: set-to-nothing needs a value
** Where: set-to-nothing:
```

7.2.6 no-arg

Une fonction a été évaluée sans lui fournir tous les arguments attendus.

Message :

```
[:arg1 "is missing its" :arg2 "argument"]
```

Exemple :

```
f: func [b][probe b]
filter-error try [f]
** Script Error: f is missing its b argument
** Where: f
```

7.2.7 expect-arg

Un argument a été fourni à une fonction mais n'était pas du type de données attendu.

Message :

```
[:arg1 "expected" :arg2 "argument of type:" :arg3]
```

Exemple :

```
f: func [b [block!]][probe b]
filter-error try [f "string"]
** Script Error: f expected b argument of type: block
** Where: f "string"
```

7.2.8 expect-set

Deux valeurs de type serie ! sont utilisées l'une avec l'autre d'une façon non compatible. Par exemple, en tentant la réunion entre une chaîne de caractères et un bloc.

Message :

```
["Expected one of:" :arg1 "- not:" :arg2]
```

Exemple :

```
filter-error try [union [a b c] "a b c"]
** Script Error: Expected one of: block! - not: string!
** Where: union [a b c] "a b c"
```

7.2.9 invalid-arg

Voici une erreur générique lorsqu'on manipule incorrectement des valeurs. Par exemple, lorsque qu'un set-word est utilisé à l'intérieur du bloc de spécification d'une fonction :

Message :

```
["Invalid argument:" :arg1]
```

Exemple :

```
filter-error try [f: func [word:][probe word]]
** Script Error: Invalid argument: word
```

```
** Where: func [word:] [probe word]
```

7.2.10 invalid-op

Un essai a été fait d'utiliser un opérateur qui a déjà été redéfini. L'opérateur utilisé n'est plus un opérateur valide.

Message :

```
["Invalid operator:" :arg1]
```

Exemple :

```
*: "operator redefined to a string"
filter-error try [5 * 10]
** Script Error: Invalid operator: *
** Where: 5 * 10
```

7.2.11 no-op-arg

Un opérateur mathématique ou de comparaison a été utilisé sans que ne soit fourni un deuxième argument.

Message :

```
Operator is missing an argument
```

Exemple :

```
filter-error try [1 +]
** Script Error: Operator is missing an argument
** Where: 1 +
```

7.2.12 no-return

Une fonction attendant d'un bloc une valeur de retour ne peut rien retourner. Par exemple, lors de l'usage des fonctions **while** et **until**.

Message :

```
Block did not return a value
```

Examples :

```
filter-error try [ ; first block returns nothing
    while [print 10][probe "ten"]
]
10
** Script Error: Block did not return a value
** Where: while [print 10] [probe "ten"]
filter-error try [
    until [print 10] ; block returns nothing
]
10
** Script Error: Block did not return a value
** Where: until [print 10]
```

7.2.13 not-defined

Un mot utilisé n'était pas défini dans quel que contexte que ce soit.

Message :

```
[:arg1 "is not defined in this context"]
```

7.2.14 no-refine

Une tentative a été faite d'utiliser pour une fonction un raffinement qui n'existe pas.

Message :

```
[:arg1 "has no refinement called" :arg2]
```

Exemple :

```
f: func [/a] [if a [print "a"]]
filter-error try [f/b]
** Script Error: f has no refinement called b
** Where: f/b
```

7.2.15 invalid-path

Un essai a été fait d'accéder à une valeur dans un bloc ou un objet en utilisant un path qui n'existe pas au sein du bloc ou de l'objet.

Message :

```
["Invalid path value:" :arg1]
```

Exemple :

```
blk: [a "a" b "b"]
filter-error try [print blk/c]
** Script Error: Invalid path value: c
** Where: print blk/c
obj: make object! [a: "a" b: "b"]
filter-error try [print obj/d]
** Script Error: Invalid path value: d
** Where: print obj/d
```

7.2.16 cannot-use

Une opération a été exécutée sur une valeur d'un type de données incompatible avec l'opération. Par exemple, en essayant d'ajouter une chaîne de caractères à un nombre.

Message :

```
["Cannot use" :arg1 "on" :arg2 "value"]
```

Exemple :

```
filter-error try [1 + "1"]
** Script Error: Cannot use add on string! value
** Where: 1 + "1"
```

7.2.17 already-used

Un essai a été fait pour faire un alias avec un mot qui a déjà été utilisé en alias.

Message :

```
["Alias word is already in use:" :arg1]
```

Exemple :

```
alias 'print "prink"
filter-error try [alias 'probe "prink"]
** Script Error: Alias word is already in use: prink
** Where: alias 'probe "prink"
```

7.2.18 out-of-range

Un essai est fait de modifier un index invalide d'une série.

Message :

```
["Value out of range:" :arg1]
```

Exemple :

```
blk: [1 2 3]
filter-error try [poke blk 5 "five"]
** Script Error: Value out of range: 5
** Where: poke blk 5 "five"
```

7.2.19 past-end

Une tentative d'accès à une série au delà de la longueur de la série.

Message :

```
Out of range or past end
```

Exemple :

```
blk: [1 2 3]
filter-error try [print fourth blk]
** Script Error: Out of range or past end
** Where: print fourth blk
```

7.2.20 no-memory

Le système ne dispose plus d'assez de mémoire pour terminer l'opération.

Message :

```
Not enough memory
```

7.2.21 wrong-denom

Une opération mathématique a été effectuée sur des valeurs monétaires de dénominations différentes. Par exemple, en tentant d'ajouter USD$1.00 à DEN$1.50.

Message :

```
[:arg1 "not same denomination as" :arg2]
```

Exemple :

```
filter-error try [US$1.50 + DM$1.50]
** Script Error: US$1.50 not same denomination as DM$1.50
** Where: US$1.50 + DM$1.50
```

7.2.22 bad-press

Une tentative pour décompresser une valeur binaire corrompue ou en format non compressé.

Message :

```
["Invalid compressed data - problem:" :arg1]
```

Exemple :

```
compressed: compress {some data}
change compressed "1"
filter-error try [decompress compressed]
** Script Error: Invalid compressed data - problem: -3
** Where: decompress compressed
```

7.2.23 bad-port-action

Une tentative pour effectuer une action non supportée sur un port. Par exemple, en tentant d'utiliser **find** sur un port TCP.

Message :

```
["Cannot use" :arg1 "on this type port"]
```

7.2.24 needs

Se produit lors de l'exécution d'un script qui nécessite une nouvelle version de REBOL ou quand un fichier ne peut être trouvé. Cette information devra être trouvée dans l'en-tête du script REBOL.

Message :

```
["Script needs:" :arg1]
```

7.2.25 locked-word

Apparaît lorsqu'une tentative est faite visant à modifier un mot protégé. Le mot devra avoir été protégé avec la fonction **protect**.

Message :

```
["Word" :arg1 "is protected, cannot modify"]
```

Exemple :

```
my-word: "data"
protect 'my-word
filter-error try [my-word: "new data"]
** Script Error: Word my-word is protected, cannot modify
** Where: my-word: "new data"
```

7.2.26 dup-vars

Une fonction a été évaluée et possède plusieurs occurrences du même mot défini dans son bloc de spécification. Par exemple, si le mot arg a été défini à la fois comme premier et second argument.

Message :

```
["Duplicate function value:" :arg1]
```

Exemple :

```
filter-error try [f: func [a /local a][print a]]
** Script Error: Duplicate function value: a
** Where: func [a /local a] [print a]
```

3.4 Erreurs d'accès

7.3.27 cannot-open

Un fichier ne peut être ouvert. Il peut s'agir d'un fichier en local ou en réseau. L'une des raisons la plus courante est la non existence du répertoire.

Message :

```
["Cannot open" :arg1]
```

Exemple :

```
filter-error try [read %/c/path-not-here]
** Access Error: Cannot open /c/path-not-here
** Where: read %/c/path-not-here
```

7.3.28 not-open

Un essai a été fait d'utiliser un port qui était fermé.

Message :

```
["Port" :arg1 "not open"]
```

Exemple :

```
p: open %file.txt
close p
filter-error try [copy p]
** Access Error: Port file.txt not open
** Where: copy p
```

7.3.29 already-open

Se produit lorsqu'on essaye d'ouvrir un port déjà ouvert.

Message :

```
["Port" :arg1 "already open"]
```

Exemple :

```
p: open %file.txt
filter-error try [open p]
** Access Error: Port file.txt already open
** Where: open p
```

7.3.30 already-closed

Se produit lorsqu'on essaye de fermer un port qui a déjà été fermé.

Message :

```
["Port" :arg1 "already closed"]
```

Exemple :

```
p: open %file.txt
close p
filter-error try [close p]
** Access Error: Port file.txt not open
** Where: close p
```

7.3.31 invalid-spec

Apparaît lorsqu'on essaye de créer un port avec la fonction **make**, en utilisant des spécifications incorrectes ou inadaptées ne permettant pas de le créer.

Message :

```
["Invalid port spec:" :arg1]
```

Exemple :

```
filter-error try [p: make port! [scheme: 'naughta]]
** Access Error: Invalid port spec: scheme naughta
** Where: p: make port! [scheme: 'naughta]
```

7.3.32 socket-open

Le système d'exploitation ne dispose plus de sockets à allouer.

Message :

```
["Error opening socket" :arg1]
```

7.3.33 no-connect

Une connexion défectueuse avec une autre machine. C'est une erreur générique qui couvre plusieurs situations possibles d'erreur lors de la connexion. Si une raison plus précise est connue, une erreur plus spécifique est générée.

Message :

```
["Cannot connect to" :arg1]
```

Exemple :

```
filter-error try [read http://www.host.dom/]
** Access Error: Cannot connect to www.host.dom
```

```
** Where: read http://www.host.dom/
```

7.3.34 no-delete

Se produit lorsqu'on essaye de supprimer un fichier qui est utilisé par un autre processus, ou protégé.

Message :

```
["Cannot delete" :arg1]
```

Exemple :

```
p: open %file.txt
filter-error try [delete %file.txt]
** Access Error: Cannot delete file.txt
** Where: delete %file.txt
```

7.3.35 no-rename

Une tentative a été faite de renommer un fichier utilisé par un autre processus ou protégé.

Message :

```
["Cannot rename" :arg1]
```

Exemple :

```
p: open %file.txt
filter-error try [rename %file.txt %new-name.txt]
** Access Error: Cannot rename file.txt
** Where: rename %file.txt %new-name.txt
```

7.3.36 no-make-dir

Une tentative de création d'un répertoire avec un chemin (path) qui n'existe pas ou qui a été protégé en écriture.

Message :

```
["Cannot make directory" :arg1]
```

Exemple :

```
filter-error try [make-dir %/c/no-path/dir]
```

```
** Access Error: Cannot make directory /c/no-path/dir/
** Where: m-dir path return path
```

7.3.37 timeout

Le délai de time-out s'est écoulé sans qu'une réponse ait été reçue d'une autre machine. Le time-out est défini dans l'attribut **timeout** du port.

Message :

```
Network timeout
```

7.3.38 new-level

Une tentative a été faite dans un script d'abaisser la sécurité, vers un niveau qui a été interdit. Lorsqu'un script demande à abaisser le niveau de sécurité et que l'utilisateur refuse, cette erreur est générée.

Message :

```
["Attempt to change security level to" :arg1]
```

Exemple :

```
secure quit
filter-error try [secure none] ; denied request
secure none
```

7.3.39 security

Une violation de sécurité s'est produite. Ceci arrive lorsqu'une tentative d'accès à un fichier ou au réseau est réalisée, avec un niveau de sécurité positionné sur "throw".

Message :

```
REBOL - Security Violation
```

Exemple :

```
secure throw
filter-error try [open %file.txt]
** Access Error: REBOL - Security Violation
** Where: open %file.txt
secure none
```

7.2.15 invalid-path

Un path mal formé a été employé.

Message :

```
["Bad file path:" :arg1]
```

Exemple :

```
filter-error try [read %/]
```

3.6 Erreurs internes

7.4.41 bad-path

Un chemin commençant par un mot invalide a été évalué.

Message :

```
["Bad path:" arg1]
```

Exemple :

```
path: make path! [1 2 3]
filter-error try [path]
** Internal Error: Bad path: 1
** Where: path
```

7.4.42 not-here

Se produit lorsqu'on essaye d'utiliser des caractéristiques de REBOL/Command ou de REBOL/View à partir de REBOL/Core.

Message :

```
[arg1 "not supported on your system"]
```

7.4.43 stack-overflow

Un débordement de la mémoire du système en exécutant une opération.

Message :

```
["Stack overflow"]
```

Exemple :

```
call-self: func [][call-self]
filter-error try [call-self]
** Internal Error: Stack overflow
** Where: call-self
```

7.4.44 globals-full

Le nombre maximum autorisé de définitions de mots globaux a été dépassé.

Message :

```
["No more global variable space"]
```