Guide Utilisateur de REBOL/Core

pour REBOL version 2.3

Envoyez vos commentaires/corrections

Chapitre 1 - Introduction

Ce document est la traduction française du Chapitre 1 du User Guide de REBOL/Core, c'est à dire l'Introduction au langage.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
27 avril 2005 21:05	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. A propos de REBOL

Voici rapidement quelques remarques concernant REBOL :

- REBOL est l'acronyme pour **Relative Expression-Based Object Language**.
- REBOL se prononce "reb-ol" comme dans "rebel with a cause".(**NDT**: intraduisible !)
- REBOL est un langage "messager". Son principal propos est de fournir une meilleure approche pour les communications et l'informatique distribuée.

- REBOL a été conçu par Carl Sassenrath, l'architecte responsable du système d'exploitation de l'Amiga OS, le premier système d'exploitation multi-tâches pour ordinateurs personnels.
- REBOL est plus qu'un langage de programmation. C'est aussi un langage pour représenter des données et des métadonnées. REBOL fournit une méthode unique pour le traitement, le stockage, et l'échange d'information.
- REBOL est porté sur plus de 40 systèmes d'exploitation. Un script écrit en sur Windows fonctionnera aussi bien sur Linux, UNIX, et d'autres plate-formes, ... sans nécessairement de changements.
- REBOL introduit le concept de dialectes - des sous-langages petits, efficaces, spécifiques à un domaine, pour le code, les données et les méta-données.
- Les tailles - très petites - des distributions de REBOL sont *intentionnellement* conservées ainsi, bien qu'elles incluent des centaines de fonctions, des douzaines de types de données, une aide en ligne, plusieurs protocoles Internet, une gestion d'erreurs, la compression, une console pour le débuggage, et plus encore.
- Les programmes REBOL sont faciles à écrire. Vous avez seulement besoin d'un éditeur de texte. Un programme peut être une ligne unique ou une application complète.
- REBOL/Core sert de fondation pour toute la technologie REBOL. Quoique conçus pour être simple et productif pour des débutants, le langage offre de nouvelles possibilités pour les professionnels.

La **version graphique** de REBOL, appelée **REBOL/View**, se trouve construite sur la base de REBOL/Core. Elle peut être trouvée sur le site Web de REBOL.

[Retour au sommaire]

3. Au sujet de ce manuel

Ce Manuel fournit les informations de base nécessaires à l'utilisation de REBOL/Core.

Il suppose que le lecteur soit déjà familier avec les concepts généraux de programmation et avec la terminologie des systèmes d'exploitation.

3.1 Suggestion pour les nouveaux programmeurs

Si vous êtes nouveau dans la programmation, REBOL est un excellent moyen pour démarrer.

Il existe quelques concepts généraux que REBOL utilise partout. Par exemple, le concept REBOL de série est utilisé partout depuis les structures des données jusqu'aux blocs de code.

Une fois que vous aurez appris les concepts et les méthodes propres aux séries, ceux-ci pourront être appliqués partout dans vos programmes. Vous devez *bien* apprendre ces concepts.
Vous en serez récompensé plus tard. Les chapitres du Manuel Utilisateur sont ordonnés pour faciliter votre apprentissage.

Si vous rencontrez des difficultés dans l'usage de REBOL, n'en soyez pas irrité. Beaucoup de personnes peuvent vous aider. La *Mailing List de REBOL* (voir plus loin la section concernant le Support) est composée de centaines de personnes qui se font un plaisir d'aider les débutants à démarrer.
N'hésitez pas à aller sur ce forum pour quelque raison que ce soit.

3.2 Suggestion pour les programmeurs expérimentés

Si vous êtes déjà familier avec d'autres langages de programmation tels que C, C++, Java, Pascal, Python, PERL, Basic, etc., soyez avertis : REBOL est tout à fait différent.

Vous devez savoir que REBOL n'est pas juste conçu pour être différent, mais plutôt pour donner aux programmeurs une plus grande force d'expression.
Les programmeurs qui ont maîtrisé REBOL suggèrent que la meilleure approche est d'oublier ce que vous connaissez déjà pour d'autres langages.
Pourquoi ? parce que vous ne pouvez pas concevoir des programmes REBOL de la même façon.
Bien sûr, vous pourriez créer des programmes REBOL avec un air de C, mais, si vous faisiez cela, vous perdriez énormément d'avantages offerts par REBOL.

En termes techniques, REBOL est un langage hautement réflectif, fonctionnel, symbolique, avec des régles à portée définitionnelle.
Si vous ne savez pas ce que cela signifie, ce n'est pas grave. (**NDT**: ouf !)
REBOL exploite des avancées dans la science informatique, mais vous n'avez pas besoin d'être un savant informaticien pour l'utiliser. En tant que programmeur chevronné, vous serez tenté de passer outre la plupart des chapitres de ce Manuel. Pour la plupart, c'est très bien.
Cependant, des concepts comme les séries sont critiques pour comprendre REBOL.
Si vous ne prenez pas le temps de maîtriser de tels concepts, vous trouverez qu'il est difficile d'être complétement à l'aise avec le langage REBOL.

[Retour au sommaire]

4. Conventions propres à ce manuel

Le tableau suivant décrit les conventions typographiques utilisées dans le Manuel.

Item	Convention	Exemple
Les Mots pré-définis dans le langage (comme les noms de fonctions, des variables spéciales, des objets système).	Bold, green, monospace	**`Append at change`**
Mots ne faisant pas partie du langage, tels que des noms de fichiers ou de répertoires, des noms de programmes ou de variables.	Green, monospace	`myfile window-color`
Exemples de code	Boxed bold monospace	**`do %feedback.r`**
Résultats affichés à la console REBOL	Boxed blue monospace	`true`

NDT : il s'agit des conventions propres au User Guide original.
Dans cette traduction française, ces conventions sont légérement différentes :

- le code est mis en évidence (retrait, paragraphe avec trame grisée, bordure noire, police de type "Courrier"),
- les mots clés du langage sont en gras,
- et les mots propres aux variables dans les exemples ou certains mots non traduits sont en italique.

[Retour au sommaire]

5. Support Technique

Pour des questions générales ou un *feedback* concernant les produits REBOL ou notre site Web, merci d'utiliser la page de feedback . Habituellement, nous répondons aux messages sous 24 ou 48 heures. N'oubliez pas d'inclure une adresse email valide si vous voulez une réponse.

5.1 Informations et nouvelles pour les développeurs

Le site Web pour les développeurs REBOL (www.rebol.net/) fournit les dernières informations ou nouveautés techniques, la documentation, des discussions, des bétas et plus encore. C'est également sur ce site que vous trouverez le Blog de Carl, un espace d'idées, de réflexions, et de suggestions par l'inventeur et le constructeur de REBOL , Carl Sassenrath.

5.2 Listes de Discussions et Forums

- Mailing List REBOL
  La liste de discussion de REBOL est un forum pour des questions-réponses autour de tous les thèmes liés à REBOL. Vous pouvez aussi consulter les anciens messages sur l'archive de la mailing liste sur rebol.org.

- REBOL Talk Forum
  Un forum Web indépendant consacré à des échanges à propos de REBOL.

- Groupe Google REBOL
  C'est un nouveau groupe de discussion (web/email) qui a récemment démarré sur les Google Groups. Il est encore en phase expérimentale.

- Autres lieux d'échanges
  REBOL Technologies abrite aussi plusieurs groupes de discussions privés utilisant notre technologie IOS ou le système ALTME de Safeworlds Inc.
  Voir les annonces et les informations pour les membres sur www.rebol.net.

5.3 Corrections de Bug et propositions d'amélioration

Les clients, les développeurs, et les utilisateurs de REBOL peuvent maintenant directement chercher des informations liées à des problèmes connus, ou remonter de nouveaux bugs, ou effectuer des demandes d'amélioration en utilisant notre base de données RAMBO.

5.4 Bibliothèque de scripts REBOL.org

Le site www.rebol.org est un site web de partage de ressources, avec une bibliothèque de scripts et d'exemples.
Ce site comprend aussi de nombreux tutoriels, comme les archives des messages de la Mailing List REBOL.

5.5 Nouvelles versions : alpha et beta

Nous publions des versions non finalisées de nos produits. Ce service est à destination des clients et des développeurs expérimentés seulement.
Ces pages permettent d'accéder à des versions alphas ou bétas, pas à des versions finales.

[Retour au sommaire]

6. Vos remarques sont bienvenues

Pour nous aider à améliorer les prochaines évolutions de cette documentation, nous aimerions savoir quelles corrections ou clarifications vous semblent importantes.
Envoyez-les sur la page de Feeback de notre site Web. Merci d'inclure le titre, la version, et le chapitre concernés de ce Manuel.

Chapitre 2 - Les Opérations

Ce document est la traduction française du Chapitre 2 du User Guide de REBOL/Core, qui concerne les Opérations.

Contenu

8. Mise à jour

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
3 mai 2005 17:46	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Installation de REBOL

L'installation de REBOL prend seulement quelques secondes et est très facile, non intrusive, et non perturbante !!

Pour REBOL/Core, la seule procédure d'installation consiste à décompresser les fichiers de la distribution, et à les placer dans n'importe quel répertoire sur votre ordinateur.

Pour d'autres produits REBOL, l'installation peut nécessiter de fournir des informations complémentaires, comme l'endroit où mettre les fichiers.

Voir les notes spécifiques à la version choisie.

2.1 Les fichiers de la distribution

REBOL/Core comprend dans sa distribution de base les fichiers :

rebol (.exe)	un programme exécutable qui démarre la console REBOL.
rebol.r	un fichier exécuté au lancement de REBOL (mais non nécessaire à son fonctionnement)
setup.html	des informations concernant l'installation et la mise en place.
changes.html	les modifications relatives aux versions récentes.
license.txt	la license REBOL.

D'autres fichiers peuvent être fournis, selon le type de produit et la version de REBOL.

2.2 Variable d'environnement : HOME

Bien que cela ne soit pas nécessaire, si votre système d'exploitation utilise la variable d'environnement HOME, REBOL l'utilisera pour localiser ses fichiers de démarrage.

Pour beaucoup de systèmes d'exploitation comme UNIX, ou Linux, la variable HOME est positionnée par défaut (de sorte que vous n'avez pas à vous en préoccupez).

2.3 Paramétrage du Réseau

La première fois que vous utilisez REBOL, des informations vous sont demandées concernant le réseau. Ces informations sont optionnelles. Certains protocoles, comme le SMTP (email) ou le FTP, nécessitent une adresse email ou un nom de serveur SMTP. De plus, si vous êtes derrière un pare-feu (firewall) ou si vous utilisez un serveur de proxy, vous aurez à donner des informations spécifiques pour accèder à Internet.

Pour paramètrer le réseau :

- Saisir votre adresse email. Par exemple, nom@exemple.com.
- Saisir le nom de votre serveur SMTP. Par exemple : mail.exemple.com. Utilisez le nom usuel de votre serveur SMTP. Si vous n'êtes pas sûr du nom du serveur, contactez votre administrateur réseau ou votre fournisseur d'accès à Internet pour le nom de votre serveur de mail.
- Indiquez si vous utilisez un serveur de proxy. Si vous êtes directement connecté à Internet avec un modem ou Ethernet, saisissez N (non). Si vous passez par un proxy ou un pare-feu (firewall), fournissez les informations demandées dans les paramétrages de Proxy et du Pare-feu, comme ci-dessous.

Une fois que vous aurez répondu aux questions posées, REBOL créera un fichier **user.r** et écrira dedans les paramétres Réseau. Vous pouvez changer ces paramétrages à n'importe quel moment en éditant le fichier **user.r**.

2.4 Paramétrages du Proxy et d'un Pare-feu

Des serveurs de proxy ou des firewalls sont couramment utilisés par les entreprises ou les organisations pour sécuriser les accès depuis et vers Internet.
Afin que REBOL puisse accèder à Internet via ces systèmes, vous devrez fournir quelques informations :

Quand REBOL demande si vous utilisez un serveur de proxy, répondez Y (yes).

Saisissez le nom de votre serveur de proxy (host). Il s'agit du serveur ou du pare-feu sur votre réseau, qui est utilisé pour cela.
Saisissez le numéro du port utilisé par le serveur de proxy pour ses requêtes. Typiquement, c'est le port 1080, mais cela peut varier.

Si vous ne connaissez pas le numéro du port, regardez les paramètres de votre navigateur Web, ou demandez-le à votre administrateur réseau.

REBOL est configuré par défaut pour utiliser le protocole de proxy SOCKS.
Vous pouvez indiquer un autre type en éditant le fichier **user.r** ou en fournissant à la fonction **set-net** les informations exactes sur le type de proxy utilisé. Les types suivants sont supportés :

```
socks   - utilise la dernière version de SOCKS (5)
socks5  - utilise un proxy socks5
socks4  - utilise un proxy socks4
generic - le proxy générique CERN
none    - pas de proxy
```

Ces paramétres sont fournies en tant que sixième argument de la fonction **set-net**, qui est appelée pour le fichier **user.r**.

Pour plus d'information sur la modification des caractéristiques du proxy dans le fichier **user.r**, voir

le chapitre sur les protocoles Réseau.

2.5 Contrat de license

Le contrat de license REBOL pour un utilisateur, que vous acceptez en téléchargeant ou en installant REBOL, peut être consulté à n'importe quel moment à partir de la console REBOL, en y saisissant le mot **license**.

[Retour au sommaire]

3. Le démarrage de REBOL

REBOL fonctionne sur une large variété de plate-formes. REBOL se démarre comme d'autres applications de votre système. Selon les spécificités de votre système d'exploitation, REBOL peut être lancé selon l'une ou n'importe laquelle des possibilités suivantes : une icône, une ligne de commande , une autre application.

3.1 A partir d'une icône

REBOL peut être lancé en double-cliquant sur l'icône du programme REBOL, ou un fichier ayant l'extension "**.r**" ou une icône de raccourci REBOL.

Si vous double-cliquez sur l'icône du programme, REBOL s'initialise, affiche la console et attend une saisie en ligne de commande.

Si vous voulez exécuter REBOL via un script, vous pouvez également procéder ainsi :

- glisser le script sur l'icône du programme
- ou associer le fichier avec l'exécutable REBOL
- ou créer un raccourci ou un alias avec les informations concernant l'exécutable le script.

Voir le manuel de votre système d'exploitation pour plus d'information.

3.2 A partir de la ligne de commande

Depuis la ligne de commande (le shell), allez dans le répertoire qui contient le fichier **rebol.exe** (ou rebol pour les systèmes autres que Windows), et saisissez **rebol** ou **./rebol**.

Sur certains systèmes d'exploitation, tels qu'UNIX, vous pouvez créer des alias de commandes qui sont susceptibles d'exécuter REBOL avec un jeu d'arguments et de fichiers. De plus, UNIX vous permet de créer des scripts shell qui incluent un chemin comme :

```
!#/path/to/rebol
```

au tout début du fichier. Quand vous tapez le nom d'un fichier à la ligne de commande, UNIX lancera REBOL pour exécuter le script.

3.3 Depuis une autre application

Pour écrire et débugger des scripts REBOL, il est possible de configurer votre éditeur de texte préféré pour qu'il exécute REBOL avec votre script courant passé en argument. Chaque éditeur de

texte procéde différemment.

Par exemple, dans l'éditeur CodeWright de Premia, vous pouvez utiliser les options de l'interpréteur du langage pour configurer l'usage de REBOL. De sorte que vous n'aurez plus qu'à presser une simple combinaison de touches pour que votre script en cours soit sauvegardé, puis évalué par l'interpréteur REBOL.

3.4 Problèmes de sécurité

Par défaut, le niveau de sécurité est configuré pour éviter que des scripts modifient des répertoires ou des fichiers.

3.4.1 Sécurité des ports

La fonction **secure** autorise une grande flexibilité dans la gestion et le contrôle des aspects de REBOL relatifs à la sécurité. La configuration courante de la sécurité est retournée comme résultat de la fonction **secure**.

Les paramètres de sécurité utilisent un dialecte REBOL, qui est, un langage à l'intérieur du langage.

Le dialecte normal consiste en un bloc de paires de valeurs. La première valeur de la paire spécifie ce qui doit être sécurisé :

file	concerne la sécurité des fichiers
net	concerne la sécurité du réseau

NdT :

A noter que suivant les versions de REBOL, sont aussi possibles les mots : **library** (pour l'accès aux librairies dynamiques), ou **shell** (pour l'accès au shell du système d'exploitation).

Fournir un nom de fichier ou de répertoire vous permet de déterminer des niveaux de sécurité plus fins, pour ce fichier spécifique ou ce répertoire.

La seconde valeur de la paire caractérise le niveau de sécurité.
Ce peut être soit un mot ou un bloc de mots pour un niveau de sécurité.

Les mots possibles sont :

allow	un accès sans aucune restriction
ask	demander la permission en cas de tentative d'accès.
throw	renvoie une erreur en cas de tentative d'accès.
quit	fermeture de la session REBOL en cours, en cas de tentative d'accès.

Par exemple, pour autoriser tous les accès réseau, mais fermer la session REBOL en cas de tentative d'accès aux fichiers :

```
secure [
    net allow ; les accès réseaux sont autorisés
    file quit ; cloture de la session REBOL si accès sur des fichiers
]
```

Si un bloc est utilisé plutôt qu'un mot pour la sécurité, il doit contenir des paires type de niveau de sécurité - type d'accès.

Ceci vous permet de détailler finement la sécurité que vous désirez. Les types possibles d'accès sont :

read	contrôle l'accès en lecture
write	contrôle de l'accès en écriture, et les modifications (effacement, changement de nom)
all	contrôle complet

Les paires sont analysées dans l'ordre où elles apparaissent, de sorte que les paires définies à la fin du bloc peuvent avoir pour effet de modifier la sécurité de celles définies au début. Il est possible ainsi de définir un type d'accès sans explicitement définir tous les autres.

Par exemple :

```
secure [
    net allow
    file [
    ask all
    allow read
    ]
]
```

Les définitions précédentes du niveau de sécurité invite à demander l'autorisation de l'utilisateur pour toutes les opérations sur les fichiers, sauf pour la lecture, qui est autorisée. Cette façon de faire peut aussi être utilisée pour des fichiers ou des répertoires particuliers. Par exemple :

```
secure [
    net allow
    file quit
    %source/ [ask read]
]
```

Ici, l'utilisateur est alerté si une tentative d'accès est faite sur le répertoire *%source/*. Sinon, et par défaut, c'est la fermeture de la session REBOL (**quit**).

Il existe un cas particulier pour lequel la fonction **secure** prend un seul mot en argument, ce mot étant l'une des actions possibles de sécurité. Dans ce cas, le niveau de sécurité pour *la partie réseau **et** l'accès aux fichiers* est globalement affecté par cette action :

```
secure quit
```

La fonction **secure** accepte aussi l'argument **none**, ce qui définit un accès sans aucune restriction (identique aux possibilités offertes par **allow**).

```
secure none
```

Le niveau de sécurité est maintenant :

```
secure [
    net allow
    file [
    ask all
    allow read
    ]
]
```

Si aucune restriction de sécurité n'est spécifiée, que ce soit pour le réseau ou pour l'accès aux fichiers, le fonctionnement par défaut est : **ask**, c'est à dire interroger l'utilisateur pour savoir que faire.

Le paramétrage courant ne pourra pas être modifié si une erreur se produit durant l'analyse de l'argument, le bloc définissant la sécurité.

3.4.2 Précédents paramétres de sécurité

La fonction **secure** renvoie les paramétrages précédents de sécurité, avant qu'un nouveau paramétrage ne soit défini. Il s'agit d'un bloc de paramétres pour le réseau et le système de fichiers, suivi éventuellement de ceux propres à un fichier ou un répertoire. Le mot **query** peut être utilisé pour obtenir les paramètres de sécurité sans les modifer

```
current-security: secure query
```

Vous pouvez modifier le niveau courant de sécurité en récupérant les paramètres courants, puis en les modifiant, et enfin en fournissant les nouvelles valeurs à la fonction **secure**.

Abaisser le niveau de sécurité déclenche une demande envers l'utilisateur, pour validation. Une exception cependant existe quand la session REBOL est en mode *quiet*, avec dans ce cas (baisse de la sécurité), la fermeture de la session.

Aucune demande de confirmation n'est faite à l'utilisateur dans le cas où le niveau de sécurité s'accroît. Notez que les requêtes concernant la sécurité incluent maintenant une option pour permettre un accès total pour le reste du traitement des scripts.

Quand REBOL est exécuté à partir de la ligne de commande (shell), l'argument **-s** est équivalent à :

```
secure allow
```

et l'argument **+s** est équivalent à :

```
secure quit
```

Vous pouvez utilisez aussi l'argument **--secure** avec l'un des types de niveaux de sécurité pour les accès réseau et fichiers :

```
rebol --secure throw
```

3.5 Arguments du programme

Il existe plusieurs arguments pouvant être spécifiés en ligne de commande, ou dans un script batch, ou dans les propriété d'un raccourci.

Pour voir les arguments et les options possibles pour n'importe quelle version du langage REBOL, saisissez la commande **usage** à l'invite de la console, pour obtenir :

```
The command line usage is:

    REBOL <options> <script> <arguments>

All fields are optional. Supported options are:

    --cgi (-c)       Check for CGI input
    --do expr        Evaluate expression
    --help (-?)      Display this usage information
    --nowindow (-w)  Do not open a window
    --noinstall (-i) Do not install (View)
    --quiet (-q)     Don't print banners
    --reinstall (+i) Force an install (View)
    --script file    Explicitly specify script
    --secure level   Set security level:
                     (allow ask throw quit)
    --trace (-t)     Enable trace mode
    --uninstall (-u) Uninstall REBOL (View)

Other command line options:

    +q               Force not quiet (View)
    -s               No security
    +s               Full security
    -- args          Provide args without script

Examples:

    REBOL script.r
    REBOL script.r 10:30 test@domain.dom
    REBOL script.r --do "verbose: true"
    REBOL --cgi -s
    REBOL --cgi --secure throw --script cgi.r "debug: true"
    REBOL --secure none
```

Le format de la ligne de commande est :

```
REBOL options script arguments
```

avec :

options	une ou plusieurs options du programme. Voir "Spécifier des options" ci-dessous pour plus de détails.
script	le nom du fichier du script que vous voulez exécutez. Si le nom contient des espaces, il doit être entouré de guillemets.
arguments	les arguments passés au script sous forme de chaîne de caractère. Ces arguments peuvent être récupérés depuis le script.

Tous les arguments ci-dessus sont optionnels, et n'importe quelle combinaison est autorisée.

Icônes / raccourcis :

Pour certains sytèmes d'exploitation, comme Windows, ou AmigaOS, vous pouvez utiliser des raccourcis avec des icônes, et définir des propriétés aux raccourcis qui incluent les options précédentes. Vous pouvez ainsi définir des raccourcis qui exécuteront directement vos scripts REBOL avec les bonnes options.

3.6 Fichier script

Typiquement, vous exécutez REBOL en fournissant le nom du fichier correspondant au script que vous voulez évaluez. Un seul nom de fichier est autorisé. Par exemple :

```
REBOL script.r
```

Si le nom du fichier contient des espaces, ce nom doit être entouré de guillemets :

```
REBOL "mon chemin avec des espaces.r"
```

3.7 Spécifier des options

Les options possibles sont identifiées avec un signe plus (**+**) ou moins (**-**), avant un caractère ou par un double tiret (**--**) avant un mot.
C'est l'usage standard pour définir des options à un programme pour la plupart des systèmes d'exploitation.

Voici plusieurs exemples d'utilisation.

Pour évaluer un script avec une option, comme l'option -s, qui permet d'exécuter le script sans

aucune sécurité, saisissez :

```
REBOL -s script.r
```

Pour obtenir des informations sur l'usage de REBOL, tapez :

```
REBOL -?
REBOL --help
```

Pour exécuter REBOL sans ouvrir une nouvelle fenêtre (ce qui se produit si vous avez besoin de rediriger la sortie vers un fichier ou un serveur), saisissez :

```
REBOL -w
REBOL --nowindow
```

Pour prévenir l'affichage des informations de démarrage, ce qui peut être nécessaire en cas de redirection vers un fichier ou un server, tapez :

```
REBOL -q
REBOL --quiet
```

Pour évaluer une expression REBOL depuis la ligne de commande :

```
REBOL --do "print 1 + 2"
REBOL --do "verbose: true" script.r
```

Ceci vous permet d'évaluer un script distant :

```
REBOL --do "do http://www.rebol.com/speed.r"
```

Pour changer le niveau de sécurité de REBOL, tapez :

```
REBOL -s script.r
REBOL --secure none script.r
```

Pour utilisez des scripts REBOL en mode CGI (voir la partie sur CGI - Common Gateway Interface, dans le chapitre consacré aux protocoles réseaux, pour plus d'infomation) :

```
REBOL -c cgi-script.r
REBOL --cgi
```

Des options multiples sont aussi possibles. Les options mono-caractère doivent être spécifiées ensemble. Les options utilisant des mots doivent être séparées par des espaces :

```
REBOL -cs cgi-script.r
REBOL --cgi --secure none cgi-script.r
```

L'exemple précédent évalue le script cgi-script.r en mode CGI, sans sécurité. La méthode compacte est préférable pour divers serveurs Web qui limitent le nombre d'arguments permis sur la ligne de commande. (comme le serveur Web Apache sur Linux).

3.8 Redirection vers un fichier

Sur la plupart des systèmes d'exploitation, il est possible de rediriger l'entrée et la sortie standard depuis et vers un fichier. L'exemple :

```
rebol -w script.r > output-file
```

redirige la sortie vers un fichier . Pareillement,

```
rebol -w script.r < input-file
```

redirige l'entrée depuis un fichier.

Lors de la redirection des entrées/sorties vers un fichier :

Utilisez l'option -w pour éviter d'ouvrir une console REBOL, qui interférerait avec la redirection standard des entrées-sorties .

3.9 Arguments des scripts

Tout ce qui est fourni sur la ligne de commande, après le nom d'un fichier script, est passé à ce script en tant qu'argument. Ceci permet d'écrire des scripts qui acceptent directement des arguments depuis la ligne de commande.

Par exemple, si vous démarrez REBOL avec la ligne :

```
REBOL script.r 10:30 test@domain.dom
```

10:30 et test@domain.com seront pris comme des arguments.

Il y a deux façons d'obtenir les arguments de la ligne de commande. La première méthode renvoie les arguments comme un **bloc** de valeurs REBOL :

```
probe system/options/args
["10:30" "test@domain.dom"]
```

La seconde méthode renvoie les arguments sous la forme d'une **chaîne** :

```
probe system/script/args
"10:30 test@domain.dom"
```

Note relative aux versions :

Les versions précédentes retournaient un bloc de valeurs à partir de **script/args** (tout comme **options/args**). Il est conseillé de vérifier que votre script manipule le bon type de données **args** selon ce qui est indiqué ci-dessus.

3.10 Fichiers de démarrage

Lorsque REBOL démarre, il tente de charger les fichiers **rebol.r** et **user.r** .

Ces fichiers sont optionnels, mais ils peuvent être utilisés pour définir le paramétrage réseau, des fonctions usuelles, et pour initialiser des données utilisées par vos scripts.

Le fichier **rebol.r** manipule des fonctions spéciales ou des extensions pour REBOL, fournies à part de la distribution standard. Il est suggéré de ne pas éditer ce fichier; en effet, il est écrasé à chaque nouvelle version de REBOL.

Le fichier **user.r** permet de gérer des préférences utilisateur. Vous pouvez éditer ce fichier et y ajouter n'importe quelle définition ou donnée nécessaire à vos besoins.

Sur des systèmes multi-utilisateur, il peut y avoir un fichier **user.r** par utilisateur. Bien que ce fichier **user.r** ne fasse pas partie de la distibution standard, il est automatiquement généré s'il n'existe pas.

Lorsque REBOL démarre, l'interpréteur vérifie d'abord l'existence des fichiers **rebol.r** et **user.r** dans le répertoire "home" et, s'ils ne sont pas trouvés là, dans le répertoire courant.

Pour définir un répertoire HOME, vous devez définir une variable d'environnement au niveau du système, pour le contexte utilisateur ou script approprié.
Notez que certains systèmes, comme UNIX ou Linux, font parfois déjà cela, de sorte que cette définition n'est pas nécessaire.

Sur Windows XP, pour définir la variable HOME, vous ajouterez, par exemple :

```
set HOME=C:\REBOL
```

à votre environnement, via les étapes suivantes :

- Choisir Panneau de configuration, dans le menu Démarrer, puis Paramètres,

- Double-cliquez sur l'icône Système puis dans l'onglet Avancé, cliquez sur le bouton "variables d'environnement"
- Cliquez sur le bouton Nouveau
- Saisissez HOME dans le champ "nom de la variable" et "C:\REBOL" (ou le chemin où vous avez placé l'exécutable REBOL) dans le champ "valeur de la variable".

Sur les systèmes UNIX, vous pouvez définir le chemin vers REBOL en ajoutant une ligne, comme celle qui suit à votre fichier .profile ou .bashrc par exemple :

```
set HOME=/usr/bin/rebol
```

Pour certaines versions de REBOL, le chemin est stocké dans un fichier .rebol situé dans votre répertoire personnel (home directory).

[Retour au sommaire]

4. Quitter REBOL

A n'importe quel moment, pour quitter REBOL, vous pouvez sélectionner "**Quit**" dans le menu "**File**" de la console, ou bien en tapant **quit** ou **q** en ligne de commande.

Vous ouvez également quitter le programme depuis un script :

```
if now/time > 12:00 [quit]
```

La console REBOL peut aussi provoquer la fermeture de la session si une erreur se produit durant le démarrage.

> **Exit** n'est pas **Quit** :
>
> N'utilisez pas le mot **exit** pour sortir de REBOL. Ce mot est utilisé pour forcer la sortie de fonctions, et renverra une erreur s'il est utilisé en ligne de commande depuis la console.

[Retour au sommaire]

5. Usage de la console

Lorsque vous démarrez REBOL/Core, la console s'ouvre pour la saisie et l'affichage. Si vous fournissez en argument au programme un script, celui-ci est évalué et vous voyez l'affichage de la sortie, pour ce script. Le prompt en input ressemble à :

```
>>
```

Si vous tapez une expression sur la ligne de commande, en validant par la touche "Entrée", elle est évaluée et le résultat est retourné avec l'indicateur suivant :

```
==
```

Par exemple :

```
>> 100 + 20
== 120
>> now - 7-Dec-1944
== 20341
```

> **Modification du prompt** :
>
> Les indicateurs du prompt et de réponse peuvent être modifiés. Voir l'Annexe sur Console pour plus d'information.

La console s'active également lorsqu'une erreur est rencontrée dans un script ou que la fonction **halt** est rencontrée dans le script.

5.1 Saisie sur plusieurs lignes

Si vous commencez un bloc sur la ligne de commande et que vous ne le terminez pas, le bloc est "étendu" sur la ligne suivante. Ceci est indiqué par une ligne qui commence avec un crochet, suivie d'une indentation. La ligne sera indentée avec quatre espaces pour chaque bloc ouvert. Par exemple :

```
loop 10 [
[    print "example"
[    if odd? random 10 [
[        print "here"
[        ]
[    ]
```

Ceci est aussi valable pour les chaînes de caractères s'étalant sur plusieurs lignes et comprises entre accolades.

```
Print {This is a long
{    string that has more
{    than one line.}
```

Les crochets et les accolades qui apparaissent au sein de chaînes entre quotes sont ignorés. Vous pouvez quitter une saisie en cours à n'importe quel moment en pressant la touche Echap (ESC).

5.2 Interruption d'un script

Un script peut être interrompu en pressant la touche Echap (ou ESC), qui retourne immédiatement à la ligne de commande. Pour certains types d'opérations système ou d'activité réseau, il peut y avoir un délai entre l'appui sur Echap et le retour du prompt.

5.3 Rappel d'historique

Chaque ligne saisie dans REBOL est mise en mémoire pour être rappelée ultérieurement si nécessaire. Les touches "Fléche haut" et "Fléche bas" sont utilisées pour faire défiler la liste des lignes précédemment tapées. Par exemple, presser la touche "Fléche haut" une fois rappelle la précédente ligne tapée à la console.

Les lignes de commande de l'historique peuvent être écrites dans un fichier, en sauvant le bloc d'historique. Voir l'Annexe sur Console pour plus d'information.

5.4 Compléter automatiquement un mot

Pour améliorer la saisie des mots et noms de fichiers longs, il est possible avec la console REBOL que ces mots soient complétés automatiquement. Après la saisie de quelques lettres du mot, tapez sur la touche "Tabulation". Si les lettres saisies permettent d'identifier de manière unique le mot, le restant de ce mot est affiché.

Par exemple, si vous saisissez :

```
>> sq
```

puis que vous appuyez sur la touche "Tabulation",

```
>> square-root
```

Si les lettres saisies ne permettent pas d'identifier sans ambiguïté le mot, vous pouvez appuyer encore sur la touche "Tabulation" pour obtenir la liste des choix possibles. Par exemple, saisissez :

```
>> so
```

Puis pressez deux fois de suite sur la touche "Tabulation" pour avoir :

```
>> sort source
so
```

et vous pourrez saisir le reste du mot, ou suffisamment de lettres pour le caractériser de manière unique. L'aide à la saisie fonctionne pour tous les mots, notamment les mots définis par l'utilisateur.

Elle fonctionne également avec les noms de fichiers, quand ceux-ci sont précédés du signe "%" (pourcentage).

```
>> print read %r
```

L'appui sur la touche "Tabulation" produira l'effet suivant :

```
>> print read %rebol.r
```

selon le contenu de votre répertoire courant.

5.5 Indicateur d'activité

Lorsque REBOL attend la fin d'une opération réseau, une indicateur d'activité apparaît pour indiquer que quelque chose est en cours. Vous pouvez changer cet indicateur avec votre propre motif (mode caractère). Voir l'Annexe sur Console pour plus d'information.

5.6 Connections Réseau

Lorsqu'une connection réseau s'effectue, un message apparaît dans la console. Par exemple, si vous saisissez :

```
>> read http://www.rebol.com
connecting to: www.rebol.com
```

Si besoin, vous pouvez désactiver cet affichage en utilisant l'option **quiet**. Voir l'Annexe sur Console pour plus d'information.

5.7 Terminal virtuel

La console fournit des possibilités propres à un terminal virtuel, comme le mouvement du curseur, son adressage, l'édition de ligne, l'effacement de l'écran, des raccourcis clavier, et la gestion de la position du curseur. Le terminal virtuel utilise le code de caractères ANSI. Ceci vous permet d'écrire des programmes pour terminaux indépendants des systèmes d'exploitation, comme des éditeurs de texte, des clients email, ou des émulateurs telnet. Plus d'information dans l'Annexe de la Console.

[Retour au sommaire]

6. Obtenir de l'aide

Il existe plusieurs sources d'information : l'aide en ligne de REBOL, la fonction **source**, la documentation sur le site Web de REBOL, la bibliothèque de scripts REBOL (library), la mailing liste, et les envois de feedback ou bugs (Rambo, par exemple).

6.1 L'aide en ligne

La fonction **help** d'aide en ligne constitue un moyen rapide d'obtenir une information synthétique sur les mots REBOL. Il y a plusieurs manières d'utiliser **help**.

Saisissez **help** ou juste **?** en ligne de commande pour voir un résumé (en anglais) de cette fonction :

```
The help function provides a simple way to get
```

```
information about words and values.  To use it
supply a word or value as its argument:

    help insert
    help find

To view all words that match a pattern:

    help "path"
    help to-

To view all words of a specified datatype:

    help native!
    help datatype!

There is also word completion from the command
line.  Type a few chars and press TAB to complete
the word.  If nothing happens, there is more than
one word that matches.  Enough chars are needed
to uniquely identify the word.

Other useful functions:

    about - for general info
    usage - for the command line arguments
    license - for the terms of user license
    source func - print source for given function
    upgrade - updates your copy of REBOL

For more information, see the user guides.
```

Si vous donnez à **help** un mot comme argument, **help** affichera toute l'information relative à ce mot. Par exemple, si vous tapez :

```
help insert
```

vous verrez :

```
USAGE:
    INSERT series value /part range /only /dup count
```

== == DESCRIPTION: == Inserts a value into a series and returns the series after the insert. == INSERT is an action value. == == ARGUMENTS: == series -- Series at point to insert (Type: series port bitset) == value -- The value to insert (Type: any-type) == == REFINEMENTS: == /part -- Limits to a given length or position. == range -- (Type: number series port) == /only -- Inserts a series as a series. == /dup -- Duplicates the insert a specified number of times. == count -- (Type: number)

La fonction **help** permet aussi de trouver des mots qui contiennent une chaîne de caractères spécifiques. Par exemple, pour trouvez tous les mots qui incluent la chaîne *path*, saisissez dans la console :

```
? "path"
```

et le résultat devrait être :

```
Found these words:
     clean-path     (function)
     lit-path!      (datatype)
     lit-path?      (action)
     path           (file)
     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)
```

Vous pouvez également également effectuer des recherches par datatypes, par types de données. Par exemple, pour voir la liste de tous les mots qui sont du type **function!**, vous pouvez tapez la commande :

```
? function!
```

et le résultat s'affichera :

```
Found these words:
     ?              (function)
     ??             (function)
     about          (function)
     alert          (function)
     alter          (function)
     append         (function)
     array          (function)
     ask            (function)
...
```

Pour obtenir une liste de tous les types de données REBOL :

```
? datatype!
Found these words:
     action!        (datatype)
     any-block!     (datatype)
     any-function!  (datatype)
     any-string!    (datatype)
     any-type!      (datatype)
     any-word!      (datatype)
     binary!        (datatype)
     bitset!        (datatype)
     block!         (datatype)
```

```
     char!          (datatype)
     datatype!      (datatype)
     date!          (datatype)
     decimal!       (datatype)
     email!         (datatype)
...
```

En ce qui concerne les Objets :

La fonction help ne fournit pas d'aide utile concernant les objets REBOL, par exemple :

```
help system/options/home
system/options/home is a path.
```

N'essayez pas de faire :

```
help system
```

car cette opération prend plusieurs minutes, et produit plus d'un mégaoctet de sortie texte à l'écran.

6.2 Consultation du code source

Les utilisateurs expérimentés peuvent apprendre beaucoup sur certaines fonctions REBOL spécifiques, en consultant leur code source. La fonction **source** affiche le code pour n'importe quelle fonction REBOL définie. Si vous tapez ceci :

```
source join
```

alors le code **source** de la fonction **join** sera affiché :

```
join: func [
    "Concatenates values."
    value "Base value"
    rest "Value or block of values"
][
    value: either series? value [copy value] [form value]
    repend value rest
]
```

Les fonctions pré-définies REBOL comprennent les fonctions mezzanine (fonctions internes implémentées en REBOL), et des fonctions définies par l'utilisateur. Les fonctions natives sont des fonctions internes implémentées en code machine, et leurs codes source ne peuvent être affichées.

6.3 Téléchargement de documentation

Vous disposez sur le site Web de REBOL (http://www.rebol.com/) d'un ensemble de documentations actualisées.

En supplément de ce manuel, il existe un Dictionnaire REBOL qui couvre tous les mots pré-définis existants en REBOL. Si l'aide en ligne ou ce guide ne donnent pas assez d'information sur un mot REBOL, regardez dans le dictionnaire pour une description détaillée.
Le dictionnaire est mis à jour à chaque version de REBOL et se trouve disponible sur le site Web : http://www.REBOL.com/docs/dictionary.html

6.4 Bibliothèque de scripts

Le site Web REBOL contient une bibliothèque de scripts avec de nombreux exemples couvrant une large variété de sujets. La bibliothèque est divisée en catégories pour rendre plus facile la recherche d'un script spécifique à une fonction donnée. Vous pouvez également chercher dans la bibliothèque des scripts qui contiennent un mot particulier.
La bibliothèque de scripts peut être trouvée sur le site : http://www.REBOL.com/library/library.html

6.5 La Mailing liste

Il est possible d'obtenir aussi de l'aide de la communauté des utilisateurs REBOL en rejoignant la liste de discussion. Pour vous inscrire, envoyer un email à rebol-request@rebol.com avec comme sujet juste le mot "**subscribe**". Par exemple :

```
send rebol-request@rebol.com "subscribe"
```

Vérifiez au préalable que votre adresse email est bien correctement paramétrée avec la fonction **set-net**.

6.6 Nous contacter

Nous souhaitons savoir ce que vous pensez; merci de nous contacter pour :

- Les rapports concernant les problèmes, bugs, crashs.
- Nous dire comment vous utilisez REBOL
- Faire des suggestions
- Obtenir plus d'informations concernant nos produits

Vous pouvez nous contacter via la page de Feedback sur notre site Web.

[Retour au sommaire]

7. Erreurs

7.1 Messages d'erreur

Il existe plusieurs types de messages d'erreur au sein de REBOL. Lorsqu'une erreur se produit, un message est affiché qui explique de quelle erreur il s'agit, et approximativement, où elle s'est produite. Par exemple, si vous saisissez :

```
abc
** Script Error: abc has no value.
** Where: abc
```

Le type d'erreur est indiqué par les premiers mots du message. Dans l'exemple ci-dessus, l'erreur est du type "Script Error". Les erreurs de script sont les plus communes et se produisent quand vous utilisez improprement une fonction du langage ou avec des arguments incorrects. D'autres types d'erreurs sont décrits dans le tableau ci-dessous sur **Types d'Erreur**.

Type d'erreur	Description
Syntax errors (Erreur de syntaxe)	Se produit lorsqu'un script contient une valeur invalide, ou un en-tête manquant, ou des crochets, accolades, parenthèses, ou guillemets non appariés.
Math errors (Erreur Mathématique)	Survient quand un nombre est divisé par zéro ou qu'un dépassement de capacité se produit.
Access errors (Erreur d'accés)	Se produit lorsqu'un accés à un fichier, un répertoire ou une opération ne peuvent être réalisés, ou que des restrictions d'accés existent.
Throw errors	Survient quand un break, exit, ou un throw est utilisé de façon incorrecte.
User errors (Erreurs Utilisateur)	Sont définies par l'utilisateur dans un script.
Internal errors (Erreurs internes)	Elles surviennent lorsqu'un problème se produit dans l'interpréteur REBOL. Si vous rencontrez l'une de ces erreurs, merci de nous en informer via la page de Feedback.

La plupart des types d'erreur peuvent être capturées et traitées dans votre script.
Voir le chapitre sur les Expressions, Section "Test de Bloc" pour une description de la fonction **try**.

L'Annexe consacrée aux Erreurs inclut aussi des informations utiles concernant les erreurs.

7.2 Redirection des erreurs

Quand des erreurs sont rencontrées dans des sessions non interactives, comme celles en mode CGI (-c ou --cgi), ou en mode sans fenêtre (-w ou --nowindow), la session se termine automatiquement.

Si un script se termine durant son exécution en mode non interactif, il vous est possible de rediriger les erreurs rencontrées vers un fichier.

Quand des erreurs sont rencontrées en mode non interactif, comme avec :

```
REBOL -cs my_script.r >> my_script.log
```

Comme pour la plupart des systèmes d'exploitation, ici, la sortie standard est redirigée vers le fichier *my_script.log*.

[Retour au sommaire]

8. Mise à jour

Au démarrage, une bannière est affichée qui identifie la version du programme.
Les numéros de version ont le format :

```
version.revision.update.platform.variation
```

Par exemple, le numéro de version :

```
2.3.0.3.1
```

indique que vous avez une version 2, revision 3, update 0, pour Windows 95/98/NT (plate-forme REBOL numéro 3.1). Une liste de tous les identifiants des plate-formes est disponible sur http://www.rebol.com.

Vous pouvez également obtenir le numéro de votre version en tapant en ligne de commande :

```
print system/version
```

Seule la dernière version est supportée par REBOL Technologies. Vous pouvez vérifier que vous avez la dernière version et automatiquement effectuer une mise à jour si nécessaire.
Pour cela, vérifiez que vous pouvez vous connecter à Internet, et depuis la console, saisissez :

```
upgrade
```

REBOL vous retournera l'un ou l'autre des messages suivants (par exemple) concernant votre version :

```
This copy of Windows 95/98/NT iX86 REBOL/core 2.3.0.3.1
is currently up to date.
```

ou:

```
This copy of Windows 95/98/NT iX86 REBOL/core 2.1.2.3.1
is not up to date. Current version is: 2.3.0.3.1.
Download current version?
To upgrade to the latest version, type Y (yes). Otherwise, type
N (no).
```

Chapitre 3 - Présentation rapide

Ce document est la traduction française du Chapitre 3 du User Guide de REBOL/Core, la présentation rapide du langage.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
18 mai 2005 - 19:31	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Présentation générale

Ce chapitre a pour objectif de vous familiariser rapidement avec le langage REBOL. A l'aide d'exemples, ce chapitre présente les concepts de base et la structure du langage, et ceci, depuis le concept concernant les valeurs des données jusqu'à la maîtrise des opérations liées au réseau.

[Retour au sommaire]

3. Valeurs

Un script est écrit avec un ensemble de valeurs. De nombreux types de valeurs existent et vous êtes confrontés à beaucoup d'entre eux dans le quotidien de vos journées.
Lorsque cela est possible, REBOL permet aussi d'utiliser des formats internationaux pour des valeurs telles que les nombres décimaux, les données monétaires, les dates et les heures.

3.1 Nombres

Les nombres sont écrits sous forme d'entiers, de décimaux ou en notation scientifique. Par exemple :

```
1234 -432 3.1415 1.23E12
```

Et vous pouvez aussi écrire au format Européen :

```
123,4 0,01 1,2E12
```

3.2 Heures

Les données horaires sont écrites en heures et en minutes, avec les secondes en option, chaque partie étant séparée par le symbole "**:**".

Par exemple :

```
12:34 20:05:32 0:25.345 0:25,345
```

Les secondes peuvent inclure les fractions de secondes (milliseconde). Les heures peuvent aussi être indiquées au format anglo-saxon AM et PM (sans espace) :

```
12:35PM 9:15AM
```

3.3 Dates

Les dates sont écrites dans l'un ou l'autre des formats internationaux : *jour-mois-année* ou *année-mois-jour*. Une date peut aussi inclure une heure et une indication de fuseau horaire. Le nom ou l'abréviation du mois peuvent être utilisés de façon à rendre le format plus semblable à celui en usage aux États-unis. Par exemple :

```
20-Apr-1998 20/Apr/1998 (USA)

20-4-1998 1998-4-20     (international)

1980-4-20/12:32         (date et heure)

1998-3-20/8:32-8:00     (avec fuseau horaire)
```

3.4 Données monétaires

Les données monétaires sont écrites avec un symbole optionnel de trois lettres représentant la devise, suivies d'une grandeur numérique. Par exemple :

```
$12.34  USD$12.34  CAD$123.45  DEM$1234,56
```

3.5 Tuples

Les "tuples" sont utilisés pour des numéros de version, des valeurs RGB (Rouge-Vert-Bleu) de couleur, et des adresses réseau. Ils sont écrits sous la forme de nombres entiers compris entre 0 et 255 et séparés par des points.

Par exemple :

```
2.3.0.3.1  255.255.0  199.4.80.7
```

Au moins deux points sont requis (sinon, le nombre sera interprété comme une valeur décimale, pas un tuple). Exemple :

```
2.3.0  ; tuple
2.3.   ; tuple
2.3    ; decimal
```

3.6 Chaînes de caractères

Les chaînes de caractères sont écrites en une seule ou plusieurs lignes. Les chaînes monolignes sont incluses entre des guillemets ("). Les chaînes s'étendant sur plusieurs lignes sont incluses entre des accolades **{ }** .

Les chaînes qui comprennent des apostrophes, des marques de tabulations ou des sauts de lignes doivent être incluses entre des accolades comme pour le format multi-lignes.
Par exemple :

```
"Here is a single-line string"

{Here is a multiline string that
contains a "quoted" string.}
```

Les caractères spéciaux (échappements) à l'intérieur de chaînes sont spécifiés avec le caractère (^). Voir le paragraphe relatif aux chaînes de caractères dans le chapitre sur les Valeurs, pour la table des caractères spéciaux.

3.7 Balises

Les balises sont utilisées dans les langages comme XML et HTML. Les balises sont incluses entre les caractères "<" et ">>". Par exemple :

```
<title> </body>

<font size="2" color="blue">
```

3.8 Adresses email

Les adresses email sont écrites directement en REBOL. Elles doivent inclure le symbole "@".

Par exemple :

```
info@rebol.com

pres-bill@oval.whitehouse.gov
```

3.9 URLs

La plupart des types d'URLs Internet sont acceptés directement sous REBOL. Les URLs doivent commencer par un nom de protocole (HTTP par exemple) suivi d'un chemin (path).

Par exemple :

```
http://www.rebol.com

ftp://ftp.rebol.com/sendmail.r

ftp://freda:grid@da.site.dom/dir/files/

mailto:info@rebol.com
```

3.10 Noms de fichiers

Les noms de fichiers sont précédés par le symbole "%" (pourcentage), qui permet de faire la distinction

avec les autres mots. Par exemple :

```
%data.txt

%images/photo.jpg

%../scripts/*.r
```

3.11 Paires

Les paires sont utilisées pour indiquer des coordonnées spatiales, comme des positions sur un écran. Elles sont utilisées pour indiquer aussi bien des positions ou des tailles. Les coordonnées sont séparées par un "**x**".

Par exemple :

```
100x50

1024x800

-50x200
```

3.12 Issues

Les "issues" sont des numéros d'identification, des séquences particulières de caractères, comme des numéros de téléphone, des numéros de séries, ou de carte de crédit.

Par exemple :

```
#707-467-8000

#0000-1234-5678-9999

#MFG-932-741-A
```

3.13 Binaires

Les binaires sont des chaînes d'octets de n'importe quelle longueur. Ils peuvent être encodés directement en hexadécimal ou en base-64.

Par exemple :

```
#{42652061205245424F4C}

64#{UkVCT0wgUm9ja3Mh}
```

[Retour au sommaire]

4. Mots

Les mots sont les symboles qu'utilise REBOL. Un mot peut être ou non une variable, selon l'usage qui en est fait. Les mots peuvent aussi être directement utilisés comme symboles.

```
show next image

Install all files here

Country State City Street Zipcode

on off true false one none
```

REBOL n'a pas de mots-clés; il n'existe pas de restriction sur les mots utilisés, et sur la façon dont ils sont utilisés. Par exemple, vous pouvez définir votre propre fonction, appelée **print**, et l'utiliser au lieu de celle prédéfinie dans le langage pour l'affichage de valeurs.

Les mots sont insensibles à la casse, et peuvent inclure des traits d'union et d'autres caractères spéciaux, comme :

```
+ - `  * ! ~ & ? |
```

Les exemples suivants présentent quelques mots valides :

```
number?  time?  date!
image-files  l'image
++ -- == +-
***** *new-line*
left&right left|right
```

La fin d'un mot est indiqué par un espace, un saut de ligne, ou l'un des caractères suivants :

```
[ ] ( ) { } " : ; /
```

Les caractères suivants, par contre, ne sont pas autorisés dans les mots :

```
@ # $ % ^ ,
```

[Retour au sommaire]

5. Blocs

REBOL se compose de groupes de valeurs et de mots placés dans des blocs. Les blocs sont utilisés pour le code, les listes, les matrices, les tableaux, les répertoires, les associations et autres formes ordonnées. Un bloc est une sorte de série, dans laquelle les valeurs sont organisées dans un ordre spécifique.

Un bloc est inclus entre deux crochets [ ]. A l'intérieur du bloc, les valeurs et les mots peuvent être organisés selon n'importe quel ordre. Les exemples suivants illustrent différentes formes de blocs valides :

```
[white red green blue yellow orange black]

["Spielberg" "Back to the Future" 1:56:20 MCA]

[
    Ted    ted@gw2.dom   #213-555-1010
    Bill   billg@ms.dom  #315-555-1234
    Steve  jobs@apl.dom  #408-555-4321
]

[
    "Elton John"  6894  0:55:68
    "Celine Dion" 68861 0:61:35
    "Pink Floyd"  46001 0:50:12
]
```

Les blocs sont utilisés pour du code aussi bien que pour des données, comme le montrent les exemples suivants :

```
loop 10 [print "hello"]

if time > 10:30 [send jim news]

sites: [
    http://www.rebol.com [save %reb.html data]
    http://www.cnn.com   [print data]
    ftp://www.amiga.com  [send cs@org.foo data]
]

foreach [site action] sites [
    data: read site
    do action
]
```

Un fichier script est aussi un bloc.

Bien qu'il n'inclut pas de crochets, le bloc est implicite. Par exemple, si les lignes ci-dessous sont mises dans un fichier script :

```
red
green
blue
yellow
```

Lorsque le fichier sera évalué, ce sera sous la forme d'un bloc contenant les mots *red*, *green*, *blue*, et *yellow*. Cela revient à écrire :

```
[red green blue yellow]
```

[Retour au sommaire]

6. Variables

Les mots peuvent être utilisés comme variables faisant référence à des valeurs. Pour définir un mot en tant que variable, faites-le suivre de deux points (:), puis de la valeur attribuée à la variable, comme indiqué dans les exemples suivants :

```
age: 22
snack-time: 12:32
birthday: 20-Mar-1997
friends: ["John" "Paula" "Georgia"]
```

Une variable peut faire référence à n'importe quel type de valeur, notamment des fonctions (voir chapitre sur les Fonctions) ou des objets (voir celui sur les Objets).

Une variable fait référence à une valeur, spécifique à un contexte donné qui peut être un bloc, une fonction, ou un programme complet.

En dehors de ce contexte, la variable peut faire référence à d'autres valeurs, ou à aucune. Le contexte d'une variable peut s'étendre sur le programme entier (portée globale) ou être restreint à un bloc particulier, une fonction, ou un objet. Dans d'autres langages, le contexte d'une variable est souvent identifié comme "*la portée d'une variable*".

[Retour au sommaire]

7. Évaluation

Les blocs sont évalués pour renvoyer leurs résultats. Quand un bloc est évalué, les valeurs de ses variables sont obtenues. Les exemples suivants évaluent les variables *age*, *snack-time*, *birthday*, et *friends* qui ont été définies dans le paragraphe précédent :

```
print age
22
if current-time > snack-time [print snack-time]
12:32
print third friends
Georgia
```

Un bloc peut être évalué plusieurs fois en utilisant une boucle, comme le montrent les exemples suivants :

```
loop 10 [prin "*"]  ;("prin" n'est pas une erreur de frappe, voir le manuel)
**********
loop 20 [
    wait 8:00
    send friend@rebol.com read http://www.cnn.com
]

repeat count 3 [print ["count:" count]]
count: 1
count: 2
count: 3
```

L'évaluation d'un bloc renvoie un résultat. Dans les exemples suivants, 5 et PM sont les résultats de l'évaluation, respectivement, de chaque bloc :

```
print do [2 + 3]
5
print either now/time < 12:00 ["AM"]["PM"]
PM
```

En REBOL, il n'y a pas d'opérateur particulier pour les règles de priorité, dans l'évaluation d'un bloc. Les valeurs et les mots d'un bloc sont toujours évalués du premier au dernier, comme l'illustre l'exemple :

```
print 2 + 3 * 10
50
```

L'usage de parenthèses peut permettre de contrôler l'ordre de l'évaluation, comme dans les exemples ci-dessous :

```
2 + (3 * 10)
32
(length? "boat") + 2
6
```

Vous pouvez aussi évaluer un bloc et retourner chacun des résultats calculés à l'intérieur du bloc. C'est l'objet de la fonction **reduce** :

```
reduce [1 + 2  3 + 4  5 + 6]
3 7 11
```

[Retour au sommaire]

8. Fonctions

Une fonction est un bloc avec des variables qui donnent de nouvelles valeurs à chaque fois que le bloc est évalué.

Ces variables sont appelées les *arguments* de la fonction. Dans l'exemple suivant, le mot *sum* fait référence à une fonction qui va accepter deux arguments, *a* et *b* :

```
sum: func [a b] [a + b]
```

Dans l'exemple précédent, **func** est utilisée pour définir une nouvelle fonction.

Le premier bloc dans la fonction définit les arguments de cette fonction. Le second bloc est le bloc de code qui sera évalué quand la fonction sera utilisée. Dans cet exemple, le second bloc additionne les deux valeurs en arguments et renvoie le résultat.

Le prochain exemple illustre le fonctionnement de la fonction *sum* qui a été définie auparavant :

```
print sum 2 3
5
```

Certaines fonctions nécessitent des variables locales tout comme des arguments. Pour définir ce genre de fonction, utilisez **function** au lieu de **func**, comme le montre l'exemple suivant :

```
average: function [series] [total] [
    total: 0
    foreach value series [total: total + value]
    total / (length? series)
]

print average [37 1 42 108]
47
```

Dans l'exemple ci-dessus, le mot *series* est un argument et le mot *total* est une variable locale utilisée par la fonction, à des fins de calcul.

Le bloc d'argument de la fonction peut contenir des libellés décrivant l'objet et l'usage de la fonction, et de ses arguments, comme illustré ci-dessous :

```
average: function [
    "Return the numerical average of numbers"
    series "Numbers to average"
] [total] [
    total: 0
    foreach value series [total: total + value]
    total / (length? series)
]
```

Les chaînes de caractères sont stockées avec la fonction et peuvent être consultées lors d'une demande d'aide, au sujet de la fonction, avec l'aide en ligne :

```
help average
USAGE:
    AVERAGE series
DESCRIPTION:
    Return the numerical average of numbers
    AVERAGE is a function value.
ARGUMENTS:
    series -- Numbers to average (Type: any)
```

[Retour au sommaire]

9. Paths

NdT : le terme anglais "path" a été conservé.

Si vous utilisez des fichiers et des URLs, alors vous êtes déjà familier avec le concept de "paths" (chemins). Un path fournit un ensemble de valeurs qui sont utilisées pour "naviguer" d'un point à un autre. Dans le cas d'un fichier, un "path" spécifie le parcours au travers les différents répertoires jusqu'à l'endroit où se trouve le fichier.

En REBOL, les valeurs dans un path sont appelées des raffinements.

Le symbole slash (**/**) est utilisé pour séparer les mots et les valeurs dans un path, comme le montrent les exemples suivants pour un fichier ou une URL.

```
%source/images/globe.jpg

http://www.rebol.com/examples/simple.r
```

Les paths sont aussi utilisés pour sélectionner des valeurs de blocs, ou des caractères dans des chaînes , accéder à des variables au sein d'objets, et encore pour les raffinements d'une fonction, comme le montrent les exemples :

```
USA/CA/Ukiah/size (block selection)
names/12          (string position)
account/balance   (object function)
match/any         (function option)
```

La fonction **print** dans l'exemple ci-dessous montre la simplicité d'utilisation des paths pour accéder à une mini base de données à partir de bloc :

```
towns: [
    Hopland [
    phone #555-1234
    web   http://www.hopland.ca.gov
    ]

    Ukiah [
    phone #555-4321
    web   http://www.ukiah.com
    email info@ukiah.com
    ]
]

print towns/ukiah/web
http://www.ukiah.com
```

[Retour au sommaire]

10. Objets

Un objet est un ensemble de variables dont les valeurs sont spécifiques à un contexte. Les objets sont utilisés pour manipuler des structures de données et des comportements complexes.

L'exemple suivant montre comment un compte bancaire est décrit par un objet dont les attributs et les fonctions sont spécifiés :

```
account: make object! [
  name: "James"
  balance: $100
  ss-number: #1234-XX-4321
  deposit:  func [amount] [balance: balance + amount]
  withdraw: func [amount] [balance: balance - amount]
]
```

Dans l'exemple ci-dessus, les mots *name*, *balance*, *ss-numer*, *deposit*, et *withdraw* sont des variables locales de l'objet *account*. Les variables *deposit* et *withdraw* sont des fonctions qui sont définies au sein de l'objet. Les variables de l'objet *account* sont accessibles avec un path :

```
print account/balance
$100.00
account/deposit $300

print ["Balance for" account/name "is" account/balance]
Balance for James is $400.00
```

L'exemple suivant montre comment créer un autre compte avec un nouveau solde, mais possédant toutes les autres valeurs provenant du premier compte.

```
checking-account: make account [
    balance: $2000
]
```

Vous pouvez aussi créer un compte bancaire qui étend les caractéristiques de l'objet *account* en ajoutant le nom de la banque et la date de la dernière opération, comme illustré ci-dessous :

```
checking-account: make account [
    bank: "Savings Bank"
    last-active: 20-Jun-2000
]

print checking-account/balance
$2000.00
print checking-account/bank
Savings Bank
print checking-account/last-active
20-Jun-2000
```

[Retour au sommaire]

11. Scripts

Un script est un fichier qui contient un bloc pouvant être chargé et évalué. Le bloc peut contenir du code ou des données, et aussi typiquement un certain nombre de sous-blocs.

Les scripts nécessitent un en-tête (**header**) pour identifier la présence de code. L'en-tête peut inclure le titre du script, la date, et d'autres informations. Dans l'exemple suivant, le premier bloc contient l'information propre à l'en-tête.

```
REBOL [
    Title: "Web Page Change Detector"
    File:  %webcheck.r
    Author: "Reburu"
    Date:  20-May-1999
    Purpose: {
    Determine if a web page has changed since it was
    last checked, and if it has, send the new page
    via email.
    }
    Category: [web email file net 2]
]

page: read http://www.rebol.com

page-sum: checksum page

if any [
    not exists? %page-sum.r
    page-sum <> (load %page-sum.r)
][
    print ["Page Changed" now]
    save %page-sum.r page-sum
    send luke@rebol.com page
]
```

[Retour au sommaire]

12. Fichiers

En REBOL, les fichiers peuvent facilement être manipulés. La liste suivante décrit quelques opérations relatives aux fichiers.

Récupérer le contenu d'un fichier texte :

```
data: read %plan.txt
```

Vous pouvez afficher un fichier texte avec :

```
print read %plan.txt
```

Pour écrire dans un fichier texte :

```
write %plan.txt data
```

Par exemple, vous pouvez écrire l'heure courante avec :

```
write %plan.txt now
```

Vous pouvez aussi facilement ajouter des données en fin de fichier :

```
write/append %plan data
```

Les fichiers binaires peuvent être lus ou écrits avec :

```
data: read/binary %image.jpg
write/binary %new.jpg data
```

Pour charger un fichier comme un bloc ou une valeur REBOL :

```
data: load %data.r
```

Sauvegarder un bloc ou une valeur dans un fichier est tout aussi facile :

```
save %data.r data
```

Pour évaluer un fichier en tant que script (ce qui nécessite un en-tête valide pour cela) :

```
do %script.r
```

Vous pouvez lister le contenu d'un répertoire avec :

```
dir: read %images/
```

et, vous pouvez alors afficher les noms de fichiers avec :

```
foreach file dir [print file]
```

Pour créer un répertoire :

```
make-dir %newdir/
```

Pour savoir quel est le répertoire courant :

```
print what-dir
```

Si vous avez besoin d'effacer un fichier :

```
delete %oldfile.txt
```

Vous pouvez aussi renommer un fichier avec :

```
rename %old.txt %new.txt
```

Pour avoir les propriétés d'un fichier :

```
print size? %file.txt

print modified? %file.txt

print dir? %image
```

[Retour au sommaire]

13. Réseau

Il existe plusieurs protocoles pré-inclus dans REBOL. Ces protocoles sont faciles à utiliser et nécessitent juste un minimum de connaissances relatives au Réseau.

13.1 HTTP

L'exemple suivant montre comment utiliser le protocole HTTP pour lire une page web :

```
page: read http://www.rebol.com
```

L'exemple ci-dessous récupère une image dans une page web, et la sauvegarde en local dans un fichier :

```
image: read/binary http://www.page.dom/image.jpg

write/binary %image.jpg image
```

13.2 FTP

Ici, le code suivant lit et écrit des fichiers sur un serveur en utilisant le protocole FTP :

```
file: read ftp://ftp.rebol.com/test.txt

write ftp://user:pass@site.dom/test.txt file
```

L'exemple suivant retourne le contenu d'un répertoire en FTP :

```
print read ftp://ftp.rebol.com/pub
```

13.3 SMTP

Les exemples suivants envoient un courrier électronique avec le protocole SMTP :

```
send luke@rebol.com "Use the force."
```

Ici le contenu d'un fichier texte est envoyé par email

```
send luke@rebol.com read %plan.txt
```

13.4 POP

L'exemple suivant récupère des emails avec le protocole POP et affiche tous les messages courants de la boîte de réception, en les laissant sur le serveur :

```
foreach message read pop://user:pass@mail.dom [
    print message
]
```

13.5 NNTP

L'exemple qui suit récupère les news en utilisant le protocole NNTP (network news transfer protocol), en lisant toutes les nouvelles relatives à un groupe en particulier :

```
messages: read nntp://news.server.dom/comp.lang.rebol
```

Ici, l'exemple ci-dessous permet la lecture et l'impression de tous les groupes de discussions :

```
news-groups: read nntp://news.some-isp.net

foreach group news-groups [print group]
```

13.6 DAYTIME

L'exemple suivant retourne l'heure courante d'un serveur :

```
print read daytime://everest.cclabs.missouri.edu
```

13.7 WHOIS

Il est possible avec l'exemple ci-dessous de savoir qui est responsable d'un domaine, avec le protocole Whois :

```
print read whois://rebol@rs.internic.net
```

13.8 FINGER

L'exemple suivant ramène des informations concernant un utilisateur avec le protocole Finger :

```
print read finger://username@host.dom
```

13.9 DNS

Ici, une adresse Internet est déterminée à partir d'un nom de domaine, et inversement un nom de domaine à partir d'une adresse IP :

```
print read dns://www.rebol.com

print read dns://207.69.132.8
```

13.10 TCP

Les connections directes avec TCP/IP sont également possibles en REBOL.

L'exemple qui illustre ceci est un serveur simple, mais pratique, qui attend des connections sur un port, et exécute ce qui a été reçu :

```
server-port: open/lines tcp://:9999

forever [
    connection-port: first server-port
    until [
    wait connection-port
    error? try [do first connection-port]
    ]
    close connection-port
]
```

Chapitre 4 - Les Expressions

Ce document est la traduction française du Chapitre 4 du User Guide de REBOL/Core, qui concerne les Expressions.

Contenu

[Retour au sommaire]

1. Historique de la Traduction

Date	Version	Commentaires	Auteur	Email
19 avril 2005 17:55	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr
19 mai 2005 08:38	1.0.1	Corrections mineures	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Présentation

L'objectif premier de REBOL est d'établir une méthode standard de communication qui soit commune à tous les systèmes informatiques. REBOL fournit un moyen simple, direct pour énoncer toutes sortes d'informations avec une syntaxe minimale, et une grande flexibilité.

Par exemple, examinez la ligne suivante :

```
Sell 100 shares of "Acme" at $47.97 per share
```

Le code ressemble beaucoup à de l'anglais, ce qui le rend facile à rédiger si vous l'écrivez et facile à comprendre si vous le recevez. Pourtant, cette ligne est actuellement une expression valide en REBOL, de sorte que votre ordinateur pourra aussi la comprendre et réagir en fonction.

Notez que cette ligne de code est un "dialecte" de REBOL. Elle ne peut être directement évaluée. (Voir plus loin concernant ce concept de dialecte.)

REBOL fournit donc un langage commun entre vous et votre ordinateur. De plus, si votre ordinateur envoie cette expression à celui de votre courtier, qui utilise aussi REBOL, l'ordinateur de votre courtier peut comprendre l'expression et la traiter en conséquence.

REBOL fournit donc un langage commun aux ordinateurs. La ligne précédente peut ainsi être envoyée à des millions d'autres ordinateurs qui eux aussi peuvent la traiter.

Le code suivant est un autre exemple d'expression REBOL :

```
Reschedule exam for 2-January-1999 at 10:30
```

L'expression ci-dessus (écrite dans un autre "dialecte") peut provenir de votre médecin qui l'a saisie, ou peut être originaire d'une application qu'il utilise. Il n'y a pas de probléme. Ce qui est important, c'est que l'expression peut être utilisable sans considérer le type d'ordinateur, de console, de télévision, d'instruments informatiques que vous utilisez.

Les données (nombres, chaînes de caractères, prix, dates, heures) dans toutes les expressions indiquées sont des formats standard REBOL.

Par ailleurs, les mots dépendent du contexte dans lequel ils sont interprétés pour convenir de leur sens. Les mots comme "sell", "at", et "read" ont différentes significations selon les contextes.

Les mots sont des expressions relatives - leur sens est dépendant du contexte.

Les expressions peuvent être traitées de deux manières : soit directement par l'interpréteur REBOL, soit indirectement par un script REBOL.

Un script traité indirectement est appelé un dialecte. Les exemples précédents sont des dialectes et, par ailleurs, sont traités par script.

L'exemple suivant n'est pas un dialecte car il est exécuté directement par l'interpréteur REBOL :

```
send master@rebol.com read http://www.rebol.com
```

Dans cet exemple, les mots "send" et "read" sont des fonctions traitées par l'interpréteur REBOL.

Au niveau de REBOL, la différence est que l'information est soit traitée directement, soit indirectement. La différence ne porte pas sur le fait que l'information soit du code ou des données mais sur la façon dont elle va être interprétée (directement ou non).

Le code REBOL est souvent manipulé comme une donnée et une donnée est fréquemment traitée comme du code, de sorte que la division classique entre données et code s'estompe.

La façon dont l'information est traitée détermine s'il s'agit de code ou de données.

[Retour au sommaire]

3. Blocs

Les expressions REBOL sont basées sur ce concept : vous pouvez combiner des valeurs et des mots dans des blocs.

Dans les scripts, un bloc est normalement compris entre deux crochets [ ].

Tout ce qui est à l'intérieur des crochets est une partie du bloc. Le bloc peut s'étendre sur plusieurs lignes, et son format est complétement libre.

Les exemples suivants montrent différentes manières de présenter le contenu de blocs :

```
[white red green blue yellow orange black]
["Spielberg" "Back to the Future" 1:56:20 MCA]
[
```

```
    "Bill"  billg@ms.dom  #315-555-1234
    "Steve" jobs@apl.dom  #408-555-4321
    "Ted"   ted@gw2.dom   #213-555-1010
]
sites: [
    http://www.rebol.com [save %reb.html data]
    http://www.cnn.com   [print data]
    ftp://www.amiga.com  [send cs@org.foo data]
]
```

Certains blocs ne nécessitent pas de crochets car ils sont implicites.

Par exemple, dans un script REBOL, il n'y a pas de crochets entourant de part et d'autre le contenu du script, et cependant, le contenu du script est un bloc.

Les crochets d'un "script-bloc" sont implicites.

La même chose est vraie pour les expressions tapées à l'invite de la console (prompt), ou pour des messages REBOL envoyés entre ordinateurs -- chacun est un bloc implicite.

Un autre aspect important des blocs est qu'ils impliquent des informations supplémentaires.
Les blocs rassemblent en effet des jeux de valeurs dans un ordre particulier.

Ceci étant, un bloc peut être utilisé comme une donnée, tout comme un ensemble.

Ceci sera décrit dans le chapitre sur les Séries.

[Retour au sommaire]

4. Valeurs

REBOL fournit en interne du langage un ensemble de valeurs qui peuvent être exprimées et échangées entre tous les systèmes. Ces valeurs sont les élèments primaires pour composer toutes les expressions REBOL.

4.1 Valeurs directes et indirectes

Les valeurs peuvent être directement ou indirectement exprimées.

Une valeur directement exprimée est connue comme elle est écrite que ce soit au niveau lexical ou littéral.
Par exemple, le nombre 10 ou l'heure 10:30 sont des valeurs directement exprimées.

Une valeur exprimée indirectement demeure inconnue tant qu'elle n'est pas évaluée.

Les valeurs **none**, **true**, et **false** nécessitent toutes des mots pour les représenter. Ces valeurs sont indirectement exprimées, car elles doivent être évaluées pour être connues. Ceci est aussi vrai d'autres valeurs, comme les listes, les tables de hachage (**hash!**), les fonctions, les objets.

4.2 Types de données des valeurs

NDT : le mot anglais "datatype", utilisé pour "type de données" sera conservé,

afin d'alléger la lecture.

Chaque valeur REBOL posséde un type de données particulier. Le type de données (*datatype*) d'une valeur définit :

- L'ensemble des valeurs possibles pour un type de données. Par exemple, le datatype logique (*logic!*) peut seulement être : **true** ou **false**.
- Les opérations qui peuvent être réalisées. Par exemple, vous pouvez additionner deux entiers, mais vous ne pouvez pas additionner deux valeurs logiques.
- La façon dont les valeurs sont stockés en mémoire. Certains datatypes peuvent être stockés directement (comme les nombres), tandis que d'autres sont stockés indirectement (comme les chaînes de caractères).

Par convention, les noms des types de données REBOL sont suivis d'un point d'exclamation (**!**) pour aider à les distinguer.

Par exemple :

```
integer!
char!
word!
string!
```

Les mots utilisés pour les types de données sont simplement des *mots*. Ils sont juste comme beaucoup d'autres mots en REBOL. Il n'y a rien de "magique" concernant le "**!**" utilisé pour les représenter.

Voir l'annexe concernant les Valeurs, pour une description de tous les datatypes REBOL.

[Retour au sommaire]

5. Evaluation des Expressions

Evaluer une expression revient à calculer sa valeur.

REBOL opére en évaluant les séries d'expressions constituant un script et ensuite retourne le résultat.
Evaluer, c'est aussi faire un traitement, exécuter un script.

L'évaluation est réalisée sur des blocs. Les blocs peuvent être saisis à la console, ou chargés à partir d'un fichier, un script. Dans les deux cas, le processus d'évaluation est le même.

5.1 Evaluation depuis la console

Toute expression pouvant être évaluée dans un script peut aussi l'être depuis l'invite de console, fournissant un moyen simple de tester individuellement les expressions d'un script.

Par exemple, si vous saisissez l'expression suivante à la console :

```
>> 1 + 2
```

L'expression est évaluée et le résultat suivant est retourné :

```
== 3
```

> **Concernant les exemples de code ...**
>
> Dans l'exemple ci-dessus, l'invite de la console (>>) et l'indicateur de réponse (==) sont indiqués pour vous donner une idée de la façon dont les résultats se présentent dans la console.
>
> Pour les exemples suivants, ils ne seront plus affichés. Vous pouvez cependant taper vous-même ces exemples pour vérifier les résultats.

5.2 Evaluation de valeurs simples

Une valeur connue directement est simplement retournée.

Par exemple, si vous saisissez la ligne suivante :

```
10:30
```

la valeur 10:30 est retournée. C'est le comportement de toutes les valeurs directement fournies. Ceci inclut :

```
integer    1234
decimal    12.34
string     "REBOL world!"
time       13:47:02
date       30-June-1957
tuple      199.4.80.1
money      $12.49
pair       100x200
char       #"A"
binary     #{ab82408b}
email      info@rebol.com
issue      #707-467-8000
tag        <IMG SRC="xray.jpg">
file       %xray.jpg
url        http://www.rebol.com/
block      [milk bread butter]
```

5.3 Evaluation de blocs

Normalement, les blocs ne sont pas évalués. Par exemple, la saisie du bloc suivant :

```
```

```
[1 + 2]
```

renverra le même bloc :

```
[1 + 2]
```

5.3.1 do

Le bloc n'est pas évalué; il est simplement traité comme une donnée. Pour évaluer un bloc, utiliser la fonction **do**, comme le montre l'exemple suivant :

```
do [1 + 2]
3
```

La fonction **do** renvoie le résultat de l'évaluation.
Ci-dessus, le nombre 3 est retourné.

Dans un bloc contenant plusieurs expressions, seul le résultat de la dernière expression est renvoyé :

```
do [
    1 + 2
    3 + 4
]
7
```

Dans cet exemple, les deux expressions sont évaluées, mais seul le résultat de 3 + 4 est renvoyé.

Un certain nombre de fonctions telles que **if**, **loop**, **while**, et **foreach** évaluent un bloc faisant partie d'elles-mêmes.

Ces fonctions sont discutées en détail plus loin dans ce chapitre, mais voici quelques illustrations :

```
if time > 12:30 [print "past noon"]
past noon
loop 4 [print "looping"]
looping
looping
looping
looping
```

C'est un point important à se rappeler : les blocs sont traités comme des données, sauf s'ils sont explicitement évalués par une fonction.
Seule une fonction peut provoquer leur évaluation.

5.4 Réduire des blocs

Quand vous évaluez un bloc avec la fonction **do**, seule la valeur de la dernière expression est retournée comme résultat. Cependant, vous voudrez parfois que les valeurs de toutes les expressions soient retournées.

5.4.1 reduce

Pour renvoyer le résultat de l'évaluation de chaque expression du bloc, utilisez la fonction **reduce**.

Dans l'exemple suivant, **reduce** est utilisé pour retourner le résultat particulier de chacune des expressions du bloc :

```
reduce [
    1 + 2
    3 + 4
]
[3 7]
```

Ci-dessus, le bloc a été ramené ("réduit") aux résultats de son évaluation. La fonction **reduce** renvoie les résultats dans un bloc.

La fonction **reduce** est importante car elle vous permet de créer des blocs d'expressions, qui sont évalués et passés à d'autres fonctions.

Reduce évalue chaque expression dans un bloc et place le résultat de l'expression dans un nouveau bloc.
Ce nouveau bloc est renvoyé comme résultat de **reduce**.

Certaines fonctions, comme **print**, utilise **reduce** comme partie intégrante dans leur fonctionnement, comme le montre l'exemple suivant :

```
print [1 + 2  3 + 4]
3 7
```

Les fonctions **rejoin**, **reform**, et **remold** l'utilisent également en interne :

```
print rejoin [1 + 2  3 + 4]
37
print reform [1 + 2  3 + 4]
3 7
print remold [1 + 2  3 + 4]
[3 7]
```

Les fonctions **rejoin**, **reform**, et **remold** sont basées sur les fonctions **join**, **form**, et **mold**, mais réduisent les blocs en premier lieu.

5.5 Evaluation des scripts

La fonction **do** peut être utilisée pour évaluer des scripts entiers.

Normalement, **do** évalue un bloc, comme indiqué dans l'exemple suivant :

```
do [print "Hello!"]
Hello!
```

Mais, lorsque **do** évalue un nom de fichier au lieu d'un bloc, le fichier est chargé dans l'interpréteur sous la forme d'un bloc, puis évalué :

```
do %script.r
```

Un en-tête REBOL valide est requis, comme décrit dans le chapitre sur les Scripts. L'en-tête identifie que le fichier contient du code et non un texte aléatoire.

5.6 Evaluation des chaînes

La fonction **do** peut être utilisée pour évaluer des expressions qui peuvent être trouvées à l'intérieur de chaînes de caractères. Par exemple, l'expression suivante :

```
do "1 + 2"
3
```

retourne le résultat 3.

D'abord, la chaîne est transformée en bloc, puis le bloc est évalué.

L'évaluation d'une chaîne peut parfois s'avérer pratique mais ne devra être utilisée que si nécessaire.

Par exemple, pour créer un processeur de ligne de commande REBOL, saisissez l'expression suivante:

```
forever [probe do ask "=> "]
```

L'expression ci-dessus attend avec le symbole "=>" que vous tapiez une chaîne de caractères. Le texte fourni doit être évalué et le résultat doit ensuite être affiché. (Bien sûr, ce n'est pas vraiment aussi simple, car le script pourrait produire une erreur).

Sauf nécessité, l'évaluation de chaînes de caractère n'est pas généralement une bonne pratique. L'évaluation de chaînes est moins efficace que l'évaluation de blocs, et, de plus, le contexte des mots dans une chaîne n'est pas connu.

Par exemple, l'expression suivante :

```
do form ["1" "+" "2"]
```

est beaucoup moins efficace que de taper :

```
do [1 + 2]
```

Des blocs REBOL peuvent être construits tout aussi facilement que des chaînes, et les blocs sont plus efficaces pour évaluer les expressions.

5.7 Evaluation d'erreurs

Des erreurs peuvent se produire pour plusieurs raisons durant l'évaluation. Par exemple, si vous divisez un nombre par zéro, l'évaluation est arrêtée et une erreur est affichée :

```
100 / 0
** Math Error: Attempt to divide by zero.
** Where: 100 / 0
```

Une erreur courante est d'utiliser un mot avant qu'il soit définit :

```
size + 10
** Script Error: size has no value.
** Where: size + 10
```

Une autre erreur courante est de ne pas fournir les valeurs correctes pour une fonction :

```
10 + [size]
** Script Error: Cannot use add on block! value.
** Where: 10 + [size]
```

Parfois les erreurs ne sont pas évidentes, et vous aurez à faire des tests pour déterminer la cause de l'erreur.

[Retour au sommaire]

6. Mots

Les expressions sont construites à partir de valeurs et de mots. Les mots sont utilisés pour représenter une signification. Un mot représente une idée ou une valeur particulière.

Dans les exemples précédents de ce chapitre, un certain nombre de mots ont été employés sans explication, dans des expressions.

Par exemple, les mots **do**, **reduce**, **try** sont utilisés, mais pas expliqués.

Les mots sont évalués légérement différemment des valeurs directement exprimées. Quand un mot est évalué, sa valeur est recherchée, évaluée, puis retournée comme résultat.

Par exemple, si vous saississez le mot suivant :

```
```

```
zero
0
```

La valeur *zero* est retournée. Le mot *zero* est prédéfini comme étant le nombre 0.
Quand le mot est recherché, un *zero* est trouvé et fournit le résultat.

Quand des mots comme **do** et **print** sont analysés, leur valeurs se trouvent être des fonctions, plutôt que de simples valeurs. Dans de tels cas, la fonction est évaluée et le résultat de la fonction est renvoyé.

6.1 Validité des noms pour les mots

Les mots sont composés de caractères alphabétiques, de nombres, et de n'importe lesquels des caractères suivants :

```
? ! . ' + - * & | = _ ~
```

Un mot ne doit pas commencer par un nombre, et il y a aussi quelques restrictions sur les mots pouvant être interprétés comme des nombres.

Par exemple, -1 et +1 sont des nombres, pas des mots.

La fin d'un mot est marquée par un espace, une nouvelle ligne, ou l'un des caractères suivants :

```
[ ] ( ) { } " : ; /
```

Par ailleurs, les crochets d'un bloc ne font pas partie d'un mot.

Le bloc qui suit contient le mot **test** :

```
[test]
```

Les caractères suivants ne sont pas autorisés dans les mots, car ils induisent une mauvaise interprétation des mots ou génèrent une erreur :

```
@ # $ % ^ ,
```

Les mots peuvent être de n'importe quelle longueur, mais ne peuvent pas se prolonger après l'extrémité d'une ligne.

```
this-is-a-very-long-word-used-as-an-example
```

Les lignes suivantes présentent des exemples de mots valides :

```
Copy print test
number?  time?  date!
image-files  l'image
++ -- == +-
***** *new-line*
left&right left|right
```

REBOL n'est pas sensible à la casse des caractères. Les mots suivants font tous référence au même mot :

```
blue
Blue
BLUE
```

La casse d'un mot est conservée lorsqu'il est affiché.

Les mots peuvent être réutilisés. La signification d'un mot est dépendante de son contexte, de sorte que des mots identiques peuvent être réutilisés dans des contextes différents.

Il n'y a pas de mot-clé en REBOL, vous pouvez réutiliser n'importe quel mot, même ceux qui sont pré-définis.

Par exemple, vous pouvez utilisez un mot dans votre code différemment de la façon dont l'interpréteur utilise ce mot.

Du bon choix des mots :

Choisissez soigneusement les mots que vous employez . Les mots sont employés pour mettre en valeur une signification. Si vous choisissez soigneusement vos mots, il sera plus facile pour vous et pour les autres de comprendre vos scripts.

6.2 Usage des mots

Les mots sont utilisés de deux manières : comme symboles ou comme variables. Dans le bloc suivant, les mots sont utilisés en tant que symboles pour des couleurs :

```
[red green blue]
```

Dans la ligne suivante :

```
print second [red green blue]
green
```

Les mots n'ont pas d'autre signification que celle utilisée en tant que nom pour les couleurs. Tous

les mots utilisés au sein des blocs servent de symboles jusqu'à ce qu'ils soient évalués.

Quand un mot est évalué, il est utilisé comme une variable. Dans l'exemple précédent, les mots **print** et **second** sont des variables reliées aux fonctions natives utilisées.

Un mot peut être écrit de quatre manières différentes pour indiquer comment il doit être traité, comme ceci est expliqué dans le chapitre consacré aux valeurs et plus spécifiquement aux mots.

Format	Ce qu'il fait
word	Evalue un mot. C'est la façon la plus simple et la plus naturelle d'écrire un mot. Si un mot référence une fonction, il sera évalué. Sinon, la valeur du mot sera retournée.
word:	Définit ou attribue la valeur d'un mot. Une nouvelle valeur est donnée. La valeur peut être n'importe quoi, et même une fonction. Voir la section sur définition des Mots, ci-dessous.
:word	Récupère la valeur d'un mot mais sans l'évaluer. Ceci est utile pour faire référence à des fonctions, et d'autres données, sans les évaluer. Voir la section sur définition des Mots, ci-dessous.
'word	Traite le mot comme un symbole, mais ne l'évalue pas. Le mot est lui-même la valeur.

6.3 Définition des mots

Un mot suivi du caractère (**:**) est utilisé pour définir ou attribuer sa valeur :

```
age: 42
lunch-time: 12:32
birthday: 20-March-1990
town: "Dodge City"
test: %stuff.r
```

Vous pouvez définir un mot pour afin qu'il représente n'importe quel type de valeur. Dans les exemples précédents, les mots ont été définis en tant qu'entier, heure, date, chaîne de caractère, et fichier.

Vous pouvez aussi définir des mots correspondant à des types plus complexes de valeurs.

Par exemple, les mots suivants sont attribués à des valeurs de blocs et de fonctions :

```
towns: ["Ukiah" "Willits" "Mendocino"]
code: [if age > 32 [print town]]
say: func [item] [print item]
```

Pourquoi les mots sont-ils définis ainsi ?

Dans beaucoup de langage, les mots sont définis avec un signe "égal", comme dans :

```
age = 42
```

En REBOL, les mots sont spécifiés avec le caractère "**:**". Il y a une raison importante à cela. Ceci réduit l'opération de définition des mots à une seule valeur lexicale. La représentation d'une opération de définition est **atomique**.

La différence entre les deux approches peut être vue sur ce exemple :

```
print length? [age: 42]
2
print length? [age = 42]
3
```

REBOL est un langage **réflectif**, il est capable de manipuler son propre code. Cette méthode permet d'écrire du code qui va manipuler facilement, et en une simple opération, la définition des valeurs.

Bien sûr, une autre raison est que le signe "=" est utilisé dans les opérations de comparaison.

Plusieurs mots peuvent être définis en une fois par un effet de "cascade" de définitions. Par exemple, chacun des mots suivants est défini à la valeur 42 :

```
age: number: size: 42
```

Les mots peuvent aussi être définis avec la fonction **set** :

```
set 'time 10:30
```

Dans cet exemple, **set** attribue au mot **time** la valeur : 10:30 . le mot **time** est écrit comme un mot litéral (usage de l'apostrophe) de sorte qu'il n'est pas évalué.

La fonction **set** peut aussi définir plusieurs mots :

```
set [number num ten] 10

print [number num ten]
10 10 10
```

Dans l'exemple précédent, remarquez que les mots n'ont pas besoin d'apostrophes car ils sont à

l'intérieur d'un bloc, qui n'est pas évalué.

La fonction **print** montre l'attribution à chaque mot de la valeur 10.

Si un bloc de valeurs est fourni à **set**, chacune des valeurs sera attribuée à un mot. Dans l'exemple suivant, les mots **one**, **two**, et **three** prendront respectivement la valeur **1**, **2** et **3** :

```
set [one two three] [1 2 3]

print three
3
print [one two three]
1 2 3
```

Voir l'annexe concernant les valeurs, pour une description de tous les datatypes REBOL.

6.4 Récupérer la valeur des mots

Pour récupérer la valeur d'un mot qui a été précédemment défini, placez le symbole **:** devant le mot. Un mot préfixé par deux points (**:**) récupére la valeur du mot, mais sans l'évaluer sauf si c'est une fonction.

Par exemple, la ligne suivante :

```
drucken: :print
```

définit un nouveau mot **drucken** (l'équivalent allemand pour **print**) pour faire référence à la même fonction que **print**. En effet, **:print** renvoie la fonction pour **print** mais sans l'évaluer.

A présent, **drucken** se comporte comme la fonction **print** :

```
drucken "test"
test
```

print et **drucken** référencent la même valeur, la fonction effectuant l'affichage.

Ceci peut aussi être réalisé avec la fonction **get**. Quand un mot litéral est donné, **get** renvoie sa valeur, mais sans l'évaluer :

```
stampa: get 'print
stampa "test"
test
```

La possibilité de récupérer la valeur d'un mot est aussi importante, par exemple si vous voulez connaître cette valeur mais sans l'évaluer.

Vous pouvez déterminer si un mot est une fonction native en utilisant la ligne suivante :

```
print native? :if
true
```

La fonction **if** n'est pas évaluée, mais elle est passée à la fonction **native?** qui vérifie si elle est bien du type de données **native!**.

Sans le caractère **:** placé devant, la fonction **if** aurait été évaluée et parce qu'il n'y a pas d'arguments, une erreur se produirait.

6.5 Mots Littéraux (Literal Words)

La possibilité de traiter un mot comme un "littéral" est commode.

Aussi bien **set** que **get**, mais également d'autres fonctions comme **value?**, **unset**, **protect** et **unprotect**, attendent un mot littéral.

Les mots littéraux peuvent être écrits de l'une ou l'autre manière : en préfixant le mot avec une apostrophe, (`), ou en placant le mot dans un bloc.

Vous pouvez utiliser l'apostrophe devant le mot à évaluer :

```
word: 'this
```

Dans l'exemple ci-dessus, le mot littéral *this*, et non la valeur de celui-ci, est affecté à la variable *word*. La variable *word* utilise juste symboliquement le nom.

L'exemple ci-dessous montre que si vous affichez la valeur du mot, vous aurez ceci :

```
print word
this
```

Vous pouvez aussi obtenir des mots littéraux à partir de blocs non évalués. Dans l'exemple suivant, la fonction **first** récupére le premier mot du bloc. Ce mot est alors passé à la variable *word*.

```
word: first [this and that]
```

N'importe qeul mot peut être utilisé comme littéral. Il peut ou non faire référence à une valeur. Le mot **print** a une valeur mais il peut cependant être utilisé comme un littéral car les mots littéraux ne sont pas évalués.

```
word: 'here
print word
here
word: 'print
print word
print
```

L'exemple suivant illustre l'importance des valeurs littérales :

```
video: [
    title "Independence Day"
    length 2:25:24
    date   4/july/1996
]
print select video 'title
Independence Day
```

Dans cet exemple, le mot *title* est recherché dans le bloc. Si l'apostrophe était manquante à ce mot *title*, alors sa valeur devrait être utilisée. Si *title* n'avait pas de valeur attribuée, une erreur serait affichée.

Voir l'annexe concernant les valeurs, pour plus de détail concernant les mots littéraux.

6.6 Mots non définis

Un mot qui n'a pas de valeur est **unset**, non attribué.

Si un mot non défini est évalué, une erreur se produit :

```
>> outlook
** Script Error: outlook has no value.
** Where: outlook
```

Le message d'erreur dans l'exemple précédent indique qu'aucune valeur n'a été attribuée à ce mot. Le mot est non défini. Ne confondez pas ceci avec un mot qui aurait été défini à **none**, qui est une valeur valide.

Un mot préalablement défini peut être déclaré "unset" à n'importe quel moment en utilisant la fonction :

```
unset 'word
```

Lorsqu'un mot devient "usnet", sa valeur est perdue. Pour déterminer si un mot est "unset", utilisez la fonction **value?** qui prend en argument un mot littéral :

```
if not value? 'word [print "word is not set"]
word is not set
```

Ceci peut être commode pour des scripts qui appellent d'autres scipts. Ci-dessous, un script initialise un paramétre par défaut (*test-mode*) qui n'a pas été défini auparavant

```
if not value? 'test-mode [test-mode: on]
```

6.7 Protection des mots

Vous pouvez éviter qu'un mot soit modifié, avec la fonction **protect**.

```
protect 'word
```

Tenter de redéfinir un mot protégé génére une erreur :

```
word: "here"
** Script Error: Word word is protected, cannot modify.
** Where: word: "here"
```

Un mot peut être "déprotégé" en utilsant la fonction **unprotect** :

```
unprotect 'word
word: "here"
```

Les fonctions **protect** et **unprotect** peuvent aussi accepter un bloc de mots :

```
protect [this that other]
```

Les mots et les fonctions importantes du système peuvent être protégés en utilisant la fonction **protect-system**.

La protection des fonctions et des mots du système est particulièrement utile pour les débutants qui pourraient accidentellement modifier des mots importants.

Si **protect-system** est placé dans votre fichier **user.r**, alors tous les mots prédéfinis seront protégés.

[Retour au sommaire]

7. Evaluation conditionnelle

Comme mentionné précédemment, les blocs ne sont pas évalués, normalement. La fonction **do** est requise pour forcer l'évaluation d'un bloc. Parfois, vous aurez besoin qu'un bloc soit évalué sous condition. Le paragraphe suivant décrit plusieurs façons de faire cela :

7.1 Bloc Conditionnel

La fonction **if** prend deux arguments. Le premier argument est une condition et le second argument est un bloc. Si la condition est : **true**, le bloc est évalué, sinon il n'est pas évalué.

```
if now/time > 12:00 [print "past noon"]
past noon
```

La condition est normalement une expression qui est évaluée à *true* ou à *false*; cependant, d'autres valeurs peuvent aussi être fournies.

Seules les valeurs *false* ou *none* évite à un bloc d'être évalué.

Ceci peut être pratique pour vérifier les résultats des fonctions **find**, **select**, **next**, et d'autres fonctions, qui retournent **none** :

```
string: "let's talk about REBOL"
if find string "talk" [print "found"]
found
```

La fonction **either** améliore la fonction **if** en incluant un troisième argument, qui est le bloc à évaluer si la condition est fausse (*false*) :

```
either now/time > 12:00 [
    print "after lunch"
][
    print "before lunch"
]
after lunch
```

La fonction **either** interpréte également une valeur **none** comme étant **false**.

Les fonctions **if** et **either** renvoient le résultat de l'évaluation de leurs blocs. Dans le cas de la fonction **if**, la valeur du bloc est retournée seulement si le bloc est évalué.
La fonction **if** est commode pour initialiser de façon conditionnelle des variables :

```
flag: if time > 13:00 ["lunch eaten"]
print flag
lunch eaten
```

Avec la fonction **either**, l'exemple précédent peut être réécrit ainsi :

```
print either now/time > 12:00 [
    "after lunch"
][
    "before lunch"
]
after lunch
```

Bien que **if** et **either** soient toutes deux des fonctions, leurs arguments de type *block* peuvent être une expression quelconque dont le résultat donnerait un bloc lors de son évaluation.

Dans l'exemple suivant, des mots (*notice*, *sleep*) sont utilisés pour représenter l'argument de type *block!* pour **if** et **either**.

```
notice: [print "Wake up!"]
if now/time > 7:00 notice           ;--  mot notice un bloc simple
Wake up!
notices: [                          ;-- notices un bloc de bloc
    [print "It's past sunrise!"]
    [print "It's past noon!"]
    [print "It's past sunset!"]
]
if now/time > 12:00 second notices
It's past noon!
sleep: [print "Keep sleeping"]     ; - le mot sleep
either now/time > 7:00 notice sleep
Wake up!
```

Les expressions conditionnelles utilisées pour le premier argument de **if** ou **either** peuvent être composées de fonctions très diverses, fonctions de logique ou de comparaison.

Voir le chapitre sur les Maths pour plus d'information.

Evitez une erreur courante :

Une erreur communément faite en REBOL est d'oublier le second bloc pour **either** ou, a contrario, d'ajouter un second bloc pour **if**.

Les deux exemples suivants sont tous deux générateurs d'erreurs difficiles à détecter :

```
either age > 10 [print "Older"]
if age > 10 [print "Older"] [print "Younger"]
```

Ces types d'erreurs peuvent être difficiles à détecter, aussi, ayez cela à l'esprit si ces fonctions ne vous paraissent pas ressembler à ce qu'elles devraient.

7.2 Any et All

Les fonctions **any** et **all** fournissent un raccourci pour évaluer certains types d'expressions conditionnelles. Ces fonctions peuvent être utilisées de nombreuses manières : avec **either** en conjuguaison avec **if**, avec **either** seulement, avec d'autres fonctions conditionnelles, ou séparement.

Any et **all** acceptent toutes deux un bloc d'expressions, ce bloc étant évalué ainsi : la première expression est évaluée, puis la suivante, etc...

La fonction **any** retourne la première expression *true* rencontrée, et la fonction **all** retourne sa première expression *false*.

N'oubliez pas qu'une expression *false* peut être aussi *none*, et qu'une expression *true* peut être n'importe quelle valeur SAUF *false* et *none*.

La fonction **any** renvoie donc la première valeur rencontrée qui ne soit pas fausse (*false*), sinon elle renvoie **none**.

Les deux fonctions **any** et **all** évaluent juste ce qui leur est nécessaire.

Par exemple, dés que **any** a trouvé une expression "vraie" (*true*), les expressions restantes ne sont pas évaluées.

Voici un exemple d'usage de la fonction **any** :

```
size: 50
if any [size < 10 size > 90] [
    print "Size is out of range."
]
```

Le comportement de **any** est aussi pratique pour définir des valeurs par défaut. Par exemple, les lignes de code suivantes redéfinissent la valeur de *number* à 100, mais seulement si sa valeur initiale est **none** :

```
number: none
print number: any [number 100]
100
```

Pareillement, si vous avez potentiellement plusieurs valeurs, vous pouvez utiliser la première à avoir une valeur autre que *non* :

```
num1: num2: none
num3: 80
print number: any [num1 num2 num3]
80
```

Vous pouvez utiliser **any** avec des fonctions comme **find** pour retourner systématiquement un résultat valide :

```
data: [123 456 789]
print any [find data 432 999]
999
```

De la même manière, la fonction **all** peut être utilisée pour des conditions qui nécessitent d'être toutes "vraies" (*true*) :

```
if all [size > 10 size < 90] [print "Size is in range"]
Size is in range
```

Vous pouvez vérifier que des valeurs ont bien été définies avant d'évaluer une fonction :

```
a: "REBOL/"
b: none
probe all [string? a string? b append a b]
none
b: "Core"
probe all [string? a string? b append a b]
REBOL/Core
```

7.3 Boucles conditionnelles

Les fonctions **until** et **while** répétent l'évaluation d'un bloc jusqu'à ce qu'une condition soit remplie.

La fonction **until** répéte l'évaluation d'un bloc jusqu'à ce que cette évaluation renvoie **true** (c'est à dire : ni **false**, ni **none**).

L'évaluation du bloc est toujours réalisée au moins une fois. La fonction **until** renvoie la valeur de ce bloc.

L'exemple ci-dessous affichera chaque mot du bloc *color*. Le bloc commence par afficher le premier mot du bloc. Puis on se déplace à la couleur suivante, et ainsi pour chaque couleur dans le bloc. Lorsque la fin du bloc est atteinte et que la fonction **tail?** renvoie **true**, alors la boucle **until** se termine.

```
color: [red green blue]
until [
    print first color
    tail? color: next color
]
red
green
blue
```

N.B. : la fonction **break** peut être utilisée pour sortir de la boucle **until** à n'importe quel moment.

La fonction **while** répéte l'évaluation de ses deux arguments (des blocs), jusqu'au moment où le premier bloc renvoie **true**.

Le premier bloc est le bloc conditionnel, le second bloc est le bloc d'évaluation.

Quand le bloc conditionnel est évalué et renvoie **true**, le second bloc n'est pas évalué, et la boucle se termine.
Voici un exemple identique au précédent. La boucle **while** continuera d'afficher une couleur du bloc *color* tant qu'il y aura des couleurs à afficher.

```
color: [red green blue]
while [not tail? color] [
    print first color
    color: next color
]
red
green
blue
```

Le bloc conditionnel peut contenir plusieurs expressions, à condition que la dernière expression renvoie la condition. Pour illustrer cela, l'exemple suivant ajoute une ligne "*print index? color*" au bloc conditionnel. Cette ligne affichera l'index de la valeur courante de la couleur. Le test sur la fin du bloc de couleur s'effectue ensuite, fournissant la condition utilisée pour la boucle :

```
color: [red green blue]
while [
    print index? color
    not tail? color
][
    print first color
    color: next color
]
1
red
2
green
3
blue
4
```

La dernière valeur du bloc est retournée par la fonction **while**.

N.B. : la fonction **break** peut être utilisée, là encore, pour forcer la sortie de la boucle, à n'importe quel moment.

7.4 Erreurs classiques

> Les expressions conditionnelles sont fausses seulement lorsque leur évaluation retourne **false** ou **none**, et elles sont **true** (vraies) pour **toute autre valeur**.

Toutes les expressions conditionnelles dans les exemples suivants renvoient **true**, même pour des valeurs comme zéro ou un bloc vide :

```
if true [print "yep"]
yep
if 1 [print "yep"]
yep
if 0 [print "yep"]
yep
if [] [print "yep"]
yep
```

L'expression conditionnelle suivante renvoie **false** :

```
if false [print "yep"]
```

```
if none [print "yep"]
```

N'incluez PAS d'expressions conditionnelles dans un bloc. Les expressions conditionnelles comprises dans un bloc retournent toujours un résultat **true** :

```
if [false] [print "yep"]
yep
```

Ne confondez pas **either** et **if**. Par exemple, si vous souhaitez écrire :

```
either some-condition [a: 1] [b: 2]
```

mais écrivez ceci à la place :

```
if some-condition [a: 1] [b: 2]
```

la fonction **if** va ignorer le second bloc. Il n'y aura pas d'erreur, mais le second bloc ne sera jamais évalué.

Le contraire est aussi vrai. Si vous écrivez la ligne suivante, en oubliant le second bloc :

```
either some-condition [a: 1]
```

la fonction **either** n'évaluera pas correctement le code et pourra produire un résultat erroné.

[Retour au sommaire]

8. Evaluations en boucle

Les fonctions **while** et **until** ci-dessus ont été employées pour une boucle jusqu'à ce qu'une condition soit remplie.

Il y a d'autres fonctions permettant de réaliser une boucle, un certain nombre de fois.

8.1 Loop

La fonction **loop** évalue un bloc autant de fois qu'indiqué. L'exemple suivant affiche une ligne de quarante tirets :

```
loop 40 [prin "-"]
----------------------------------------
```

Notez que la fonction **prin** est identique à la fonction **print**, mais affiche son argument sans ajouter de retour à la ligne.

La fonction **loop** renvoie la valeur de l'évaluation finale du bloc :

```
i: 0
print loop 40 [i: i + 10]
400
```

8.2 Repeat

La fonction **repeat** rajoute à la fonction **loop** la possibilité de contrôler votre compteur de boucle. Le premier argument de la fonction **repeat** est un mot qui sera utilisé pour manipuler la valeur du compteur :

```
repeat count 3 [print ["count:" count]]    ; count est le mot "compteur"
count: 1
count: 2
count: 3
```

Le résultat du dernier bloc est aussi retourné par la fonction :

```
i: 0
print repeat count 10 [i: i + count]
55
```

Dans cet exemple, le mot *count* a seulement une valeur à l'intérieur du bloc à répéter. En d'autres termes, la valeur de *count* est locale au bloc. A l'issue de la boucle, *count* référence la valeur définie qu'il pouvait avoir avant.

8.3 For

La fonction **for** rajoute à **repeat** une valeur de départ, une autre de fin, et un incrément à préciser. Chacune de ces valeurs peut être positive ou négative.

L'exemple ci-dessous démarre son compteur à zéro et va jusqu'à 50, avec un incrément de 10, à chaque itération.

```
for count 0 50 10 [print count]
0
10
20
30
40
50
```

La fonction **for** fait une itération jusqu'à la valeur de fin *incluse*.

L'exemple ci-dessous indique une valeur de fin égale à 55. Cette valeur ne sera jamais atteinte parce que le compteur est incrémenté de 10 à chaque itération. La boucle se terminera donc à 50.

```
for count 0 55 10 [prin [count " "]]
0 10 20 30 40 50
```

L'exemple suivant montre la valeur du compteur est décrémentée. Il commence à quatre et diminue jusqu'à zéro d'une unité à la fois.

```
for count 4 0 -1 [print count]
4
3
2
1
0
```

La fonction **for** travaille aussi avec des nombres décimaux, des valeurs monétaires, des dates/heures, des séries, et des caractères.

Soyez sûrs d'utiliser pour les valeurs de début et de fin le même type de données.

Voici plusieurs exemples d'utilisation de boucle avec d'autres types de données :

```
for count 10.5 0.0 -1 [prin [count " "]]
10.5 9.5 8.5 7.5 6.5 5.5 4.5 3.5 2.5 1.5 0.5
for money $0.00 $1.00 $0.25 [prin [money " "]]
$0.00 $0.25 $0.50 $0.75 $1.00
for time 10:00 12:00 0:20 [prin [time " "]]
10:00 10:20 10:40 11:00 11:20 11:40 12:00
for date 1-jan-2000 4-jan-2000 1 [prin [date " "]]
1-Jan-2000 2-Jan-2000 3-Jan-2000 4-Jan-2000
for char #"a" #"z" 1 [prin char]
abcdefghijklmnopqrstuvwxyz
```

La fonction **for** peut aussi être utilisée sur des séries.

Voici un exemple sur une valeur de type chaîne. Le mot *end* définit la chaîne de caractères *str*, avec son index courant sur le caractère "d".

La fonction **for** parcourt la série de caractères, un par un, et s'arrête lorsque la position définie par *end* est atteinte :

```
str: "abcdef"
end: find str "d"
for s str end 1 [print s]
abcdef
bcdef
cdef
def
```

8.4 Foreach

La fonction **foreach** fournit une façon commode de répéter l'évaluation d'un bloc pour chaque élément d'une série. Elle fonctionne avec tous types de blocs, et de séries de type caractère.

Dans l'exemple ci-dessous, chaque mot dans le bloc sera affiché

```
colors: [red green blue]
foreach color colors [print color]
red
green
blue
```

Dans l'exemple suivant, chaque caractère dans la chaîne sera affiché :

```
string: "REBOL"
foreach char string [print char]
REBOL
```

Ici, chaque nom de fichier dans un répertoire est retourné :

```
files: read %.
foreach file files [
    if find file ".t" [print file]
]
file.txt
file2.txt
newfile.txt
output.txt
```

Quand un bloc contient des groupes de valeurs qui sont en lien les unes avec les autres, la fonction **foreach** peut récupérer toutes les valeurs du groupe en même temps.

Par exemple, voici un bloc qui contient une heure, une chaîne de caractères (un nom), et un prix.

En fournissant comme argument à la fonction **foreach** un bloc de mots pour le groupe, chacune de ces valeurs peut être cherchée et affichée :

```
movies: [
     8:30 "Contact"      $4.95
    10:15 "Ghostbusters" $3.25
    12:45 "Matrix"       $4.25
]
foreach [time title price] movies [
    print ["watch" title "at" time "for" price]
]
watch Contact at 8:30 for $4.95
watch Ghostbusters at 10:15 for $3.25
watch Matrix at 12:45 for $4.25
```

Dans l'exemple ci-dessus, le bloc de valeurs *[time title price]* mentionne que trois valeurs seront cherchées dans la série *movies* pour chaque évaluation du bloc.

Dans la boucle **foreach**, les variables utilisées (ici *time, title, price*) sont locales.

Leur valeur est uniquement définie au sein du bloc en cours d'évaluation.

Une fois la boucle finie, les variables retournent aux valeurs qu'elles pouvaient avoir auparavant.

8.5 Forall and Forskip

Tout comme **foreach**, la fonction **forall** évalue un bloc pour chaque valeur dans la série. Cependant, il y a quelques différences importantes.

La fonction **forall** manipule la série, en partant du début de la série. Au cours des itérations, **forall** modifie la position à l'intérieur de la série.

```
colors: [red green blue]
forall colors [print first colors]
red
green
blue
```

Dans l'exemple ci-dessus, après chaque évaluation du bloc, la série est avancée à la position suivante. Quand **forall** rend la main, l'index de la série *color* est sur la fin (*tail*) de la série.

Pour continuer à utiliser la série, vous aurez besoin de la repositionner sur la position du début (*head*), avec la ligne suivante :

```
colors: head colors
```

La fonction **forskip** évalue un bloc par groupe de valeurs dans une série. Le second argument de la fonction **forskip** est le nombre d'élements à "sauter" entre deux itérations.

Comme **forall**, **forskip** manipule la série, en commencant avec l'index au début de la série. Puis **forskip** parcourt la série en modifiant l'index de position.

Après chaque évaluation du bloc, l'index de série est avancé du nombre de positions sautées jusqu'à la nouvelle position.

L'exemple suivant illustre le fonctionnnement de **forskip** :

```
movies: [
     8:30 "Contact"      $4.95
    10:15 "Ghostbusters" $3.25
    12:45 "Matrix"       $4.25
]
forskip movies 3 [print second movies]
Contact
Ghostbusters
Matrix
```

Dans l'exemple ci-dessus, **foskip** retourne la série *movies*.

Vous devrez utiliser la fonction **head** pour repositionner la série sur sa position de départ (*head*).

8.6 Forever

La fonction **forever** évalue un bloc continuement, sans sortir, ou juste si la fonction **break** est rencontrée.

L'exemple suivant utilise **forever** pour vérifier l'existence d'un fichier, toutes les dix minutes :

```
forever [
    if exists? %datafile [break]
    wait 0:10
]
```

8.7 Break

Vous pouvez arrêter la répétition de l'évaluation d'un bloc avec la fonction **break**.

La fonction **break** est pratique quand une condition particulière est rencontrée et que la boucle doit être arrêtée.

La fonction **break** est utilisable avec tous les types de boucles.

Dans l'exemple suivant, la boucle se termine si un nombre est supérieur à 5 :

```
repeat count 10 [
    if (random count) > 5 [break]
    print "testing"
]
testing
testing
testing
```

La fonction **break** ne renvoie pas de valeur, sauf si le raffinement **return** est utilisé :

```
print repeat count 10 [
    if (random count) > 5 [break/return "stop here"]
    print "testing"
    "normal exit"
]
testing
testing
testing
stop here
```

Dans cet exemple, si la boucle **repeat** se termine sans que la condition particulière *(random count)*

> *5* se réalise, le bloc renvoie la chaîne "*normal exit*".

Sinon, **break/return** retournera la chaîne "*stop here*".

[Retour au sommaire]

9. Evaluation sélective

Il y a plusieurs méthodes pour évaluer sélectivement des expressions en REBOL. Ces méthodes fournissent pour l'évaluation une manière de trier plusieurs choix, sur la base d'une valeur de clé.

9.1 Select

La fonction **select** est souvent utilisée pour obtenir une valeur spécifique ou un bloc, à partir d'une valeur cible.

Si vous définissez un bloc de valeurs et d'actions à faire, vous pouvez utiliser **select** pour rechercher l'action correspondante à une valeur.

```
cases: [
    center [print "center"]
    right  [print "right"]
    left   [print "left"]
]
action: select cases 'right
if action [do action]
right
```

Dans cet exemple, la fonction **select** trouve le mot *rigth* et renvoie le bloc qui suit ce mot. (Si pour une raison ou une autre, la recherche était infructueuse, la valeur **none** serait retournée.)

Le bloc est alors évalué. Les valeurs utilisées dans cet exemple sont des mots, mais il peut y avoir n'importe quelle sorte de valeur :

```
cases: [
     5:00 [print "everywhere"]
    10:30 [print "here"]
    18:45 [print "there"]
]
action: select cases 10:30
if action [do action]
here
```

9.2 Switch

La fonction **select** est utilisée tellement souvent qu'il existe de cette fonction une version particulière, appelée **switch** : cette version inclut l'évaluation du bloc résultant.

La fonction **switch** rend plus simple et facile la réalisation directe d'évaluation sélective.

Par exemple, pour effectuer un choix sur une simple valeur numérique :

```
switch 22 [
    11 [print "here"]
    22 [print "there"]
]
there
```

La fonction **switch** renvoie également la valeur du bloc évalué, de sorte que l'exemple ci-dessus pourrait être ré-écrit sous la forme :

```
str: copy "right "
print switch 22 [
    11 [join str "here"]
    22 [join str "there"]
]
right there
```

et :

```
car: pick [Ford Chevy Dodge] random 3
print switch car [
    Ford  [351 * 1.4]
    Chevy [454 * 5.3]
    Dodge [154 * 3.5]
]
2406.2
```

Les différentes sélections peuvent être de n'importe quel type de données valides, ce qui inclut les nombres, les chaînes de caractères, les mots, les dates/heures, les urls, et les fichiers.

En voici d'ailleurs quelques illustrations :

- Chaînes de caractères :

```
person: "kid"
switch person [
    "dad" [print "here"]
    "mom" [print "there"]
    "kid" [print "everywhere"]
]
everywhere
```

- Mots :

```
person: 'kid
switch person [
    dad [print "here"]
    mom [print "there"]
    kid [print "everywhere"]
```

```
]
everywhere
```

- Type de données :

```
person: 123
switch type?/word [
    string! [print "a string"]
    binary! [print "a binary"]
    integer! [print "an integer number"]
    decimal! [print "a decimal number"]
]
an integer number
```

- Fichiers :

```
file: %rebol.r
switch file [
    %user.r [print "here"]
    %rebol.r [print "everywhere"]
    %file.r [print "there"]
]
everywhere
```

- URLs :

```
url: ftp://ftp.rebol.org
switch url [
    http://www.rebol.com [print "here"]
    http://www.cnet.com [print "there"]
    ftp://ftp.rebol.org [print "everywhere"]
]
everywhere
```

- Balises :

```
tag: <LI>
print switch tag [
    <PRE>   ["Preformatted text"]
    <TITLE> ["Page title"]
    <LI>    ["Bulleted list item"]
]
Bulleted list item
```

- Heures :

```
time: 12:30
```

```
switch time [
     8:00 [send wendy@domain.dom "Hey, get up"]
    12:30 [send cindy@rebol.dom "Join me for lunch."]
    16:00 [send group@every.dom "Dinner anyone?"]
]
```

9.2.1 Sélection par défaut

Une sélection par défaut peut être indiquée quand aucun des autres cas ne correspond.

Utilisez le raffinement **/default** pour définir le choix par défaut :

```
time: 7:00
switch/default time [
     5:00 [print "everywhere"]
    10:30 [print "here"]
    18:45 [print "there"]
] [print "nowhere"]
nowhere
```

9.2.2 Cas usuels

Si vous avez couramment des cas, où le résultat devrait être le même pour plusieurs valeurs, vous pouvez définir un mot pour manipuler un bloc de code commun :

```
case1: [print length? url]   ; le bloc commun
url: http://www.rebol.com
switch url [
    http://www.rebol.com case1
    http://www.cnet.com [print "there"]
    ftp://ftp.rebol.org case1
]
20
```

9.2.3 Autres cas

D'autres valeurs que des blocs peuvent être évaluées pour les sélections.

Cet exemple illustre l'évaluation d'un fichier en correspondance avec le jour de la semaine :

```
switch now/weekday [
    1 %monday.r
    5 %friday.r
    6 %saturday.r
]
```

Ainsi, si c'est vendredi (*friday*), c'est le fichier friday.r qui est évalué et le résultat est retourné.

Ce type d'évaluation marche aussi pour des URLs :

```
switch time [
     8:30 ftp://ftp.rebol.org/wakeup.r
    10:30 http://www.rebol.com/break.r
    18:45 ftp://ftp.rebol.org/sleep.r
]
```

Les choix pour **switch** sont inclus dans un bloc, et par conséquent, peuvent être définis à part de bloc **switch [...]** :

```
schedule: [
     8:00 [send wendy@domain.dom "Hey, get up"]
    12:30 [send cindy@dom.dom "Join me for lunch."]
    16:00 [send group@every.dom "Dinner anyone?"]
]
switch 8:00 schedule
```

[Retour au sommaire]

10. Stopper une évaluation

L'évaluation d'un script peut être stoppée à n'importe quel moment en pressant la touche (ESC) sur le clavier ou en utilisant les fonctions **halt** et **quit**.

La fonction **halt** arrête l'évaluation et retourne à l'invite de commande dans la console REBOL :

```
if time > 12:00 [halt]
```

La fonction **quit** arrête l'évaluation et **quitte** l'interpréteur REBOL :

```
if error? try [print test] [quit]
```

[Retour au sommaire]

11. Test de blocs

Il y a des fois où vous voudrez évaluer du code, mais ceci sans que l'évaluation du reste de votre script s'arrête, si une erreur se produit.

Par exemple, vous réalisez une division, mais vous ne voulez pas que votre script s'arrête si une division par zéro se produit.

La fonction **try** vous permet de récupérer les erreurs durant l'évaluation d'un bloc. Elle est presque identique à **do**.

La fonction **try** renverra normalement le résultat du bloc; cependant, si une erreur se produit, elle renverra la valeur de l'erreur à la place.

Dans l'exemple suivant, quand la division par zéro se produit, le script transmet une erreur à la fonction **try**, et l'évaluation continue à partir de ce point.

```
for num 5 0 -1 [
    if error? try [print 10 / num] [print "error"]
]
2
2.5
3.33333333333333
5
10
error
```

D'autres informations concernant la gestion des erreurs se trouvent dans l'Annexe consacrée aux Erreurs.

Chapitre 5 - Les scripts

Ce document est la traduction française du Chapitre 5 du User Guide de REBOL/Core, qui concerne les scripts.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
20 mai 2005 - 21:31	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Présentation générale

Le terme "script" fait référence non seulement à de simples fichiers à évaluer mais aussi à du code incorporé dans d'autres types de fichiers (comme des pages Web), ou à des morceaux de code source sauvegardés en tant que fichiers de données ou transmis comme messages.

2.1 Suffixe des scripts

Typiquement, les scripts REBOL prennent un suffixe **.r** à leur noms de fichiers. Cependant, cette convention n'est pas obligatoire. L'interpréteur lit les fichiers quelque soit leur suffixe et recherche dans leur contenu un en-tête (**header**) valide de script REBOL.

2.2 Structure

La structure d'un script est libre. L'indentation et l'usage d'espaces peuvent être utilisés pour clarifier la structure et le contenu d'un script. De plus, vous êtes encouragés à utiliser les conventions standard de style REBOL pour rendre vos scripts universellement lisibles. Voir le Guide de Style pour plus d'informations, à la fin de ce chapitre.

[Retour au sommaire]

3. En-têtes

3.1 En-tête

Précédant directement le corps du script, chaque script doit posséder un en-tête qui identifie son propos, ainsi que d'autres attributs du script.

Un en-tête de script (le "header") peut contenir le nom du script, de l'auteur, la date, le numéro de version, le nom du fichier, et des informations supplémentaires.

Les fichiers de données REBOL qui ne sont pas destinés à une évaluation directe ne nécessitent pas un en-tête.

Les en-têtes sont pratiques pour plusieurs raisons.

Ils identifient un script comme étant du texte source valide pour l'interpréteur REBOL. L'interpréteur utilise l'en-tête pour afficher le titre du script, et déterminer quelles ressources et quelles versions sont nécessaires avant d'évaluer le script. Les en-têtes fournissent un moyen normalisé de communiquer le titre, le sujet, l'auteur et d'autres détails du script.

Vous pouvez souvent déterminer à partir d'un en-tête de script si celui-ci vous intéressera.

Les bibliothèques de scripts et les sites web utilisent les en-têtes pour générer automatiquement les

répertoires, les catégories de scripts, et les références croisées.

Certains éditeurs de texte permettent de consulter et de mettre à jour les en-têtes de scripts, afin de créer un historique des informations telles l'auteur, la date, la version.

La forme générale d'un en-tête de script est :

```
REBOL [block]
```

Pour que l'interpréteur reconnaisse l'en-tête, le bloc doit immédiatement suivre le mot REBOL. Seuls des espaces blancs (espaces, tabulations, et lignes) sont autorisés entre le mot REBOL et le bloc.

Le bloc qui suit le mot REBOL va présenter le script. Il est souhaitable d'avoir au minimum un en-tête du type :

```
REBOL [
    Title:  "Scan Web Sites"
    Date:   2-Feb-2000
    File:   %webscan.r
    Author: "Jane Doer"
    Version: 1.2.3
]
```

Lorsqu'un script est chargé, le bloc d'en-tête est évalué et les valeurs sont attribuées aux mots leur faisant référence. Ces valeurs sont utilisées par l'interpréteur (pour construire un objet REBOL **system/script/header**) et peuvent aussi être utilisées par le script lui-même.

Notez que les mots définis avec des valeurs uniques peuvent aussi être définis avec des valeurs multiples, il suffit de fournir ces valeurs dans un bloc :

```
REBOL [
    Title: "Scan Web Sites"
    Date:   12-Nov-1997
    Author: ["Ema User" "Wasa Writer"]
]
```

Les en-têtes peuvent être plus complexes, avec des informations sur l'auteur, la licence, le formatage, les versions requises, l'historique des révisions, et plus encore.

Puisque le bloc est utilisé pour construire l'objet "**header**" (**system/script/header**), il peut aussi être étendu avec de nouvelles informations.

Cela signifie qu'un en-tête de script peut être adapté selon le besoin, mais ceci devrait être fait avec précaution pour éviter les ambiguïtés ou la redondance d'informations.

Un en-tête complet pourrait ressembler à quelque chose comme cela :

```
REBOL [
```

```
    Title:   "Full REBOL Header Example"
    Date:    8-Sep-1999
    Name:    'Full-Header  ; Pour le titre de la fenêtre

    Version: 1.1.1
    File:    %headfull.r
    Home:    http://www.rebol.com/rebex/

    Author:  "Carl Sassenrath"
    Owner:   "REBOL Headquarters"
    Rights:  "Copyright (C) Carl Sassenrath 1999"

    Needs:   [2.0 ODBC]
    Tabs:    4

    Purpose: {
        The purpose or general reason for the program
        should go here.
    }

    Note: {
        An important comment or notes about the program
        can go here.
    }

    History: [
        0.1.0 [5-Sep-1999 "Created this example" "Carl"]
        0.1.1 [8-Sep-1999 {Moved the header up, changed
            comment on extending the header, added
            advanced user comment.} "Carl"]
    ]

    Language: 'English
]
```

3.2 Scripts avec préambules

L'écriture des scripts n'impose pas de commencer avec un en-tête. Les scripts peuvent commencer avec n'importe quel texte, ce qui leur permettrait d'être inclus dans des messages emails, des pages web, et d'autres fichiers.

Cependant, l'en-tête marque pour REBOL le début du script, et le texte qui le suit est le corps du script. Le texte qui apparaît avant un en-tête est appelé le préambule ("*preface*") et ce texte est ignoré pendant l'évaluation.

Le texte qui apparaît avant les en-têtes est ignoré par REBOL et peut donc être utilisé pour des commentaires, des en-têtes d'email, des balises HTML, etc.

```
un essai de script
REBOL [
  Title:   "Exemple de préambule"
  Date:    8-Jul-1999
]

print "Ce fichier a un préambule avant son en-tête"
```

3.3 Inclusion de scripts

Si un script doit être suivi par du texte sans lien avec le script lui-même, le script doit être placé entre deux crochets [ ] :

```
Un peu de texte avant le script.

[
    REBOL [
        Title:   "Embedded Example"
        Date:    8-Nov-1997
    ]
    print "done"
]

Ici du texte après le script.
```

Seul un espace blanc est compris entre le premier crochet et le mot REBOL.

[Retour au sommaire]

4. Arguments des scripts

Quand un script est évalué, il peut accéder aux informations le concernant.

Celles-ci se trouvent dans l'objet REBOL **system/script**. Cet objet contient les champs pour le titre du script (**system/script/title**), les informations mises dans l'en-tête (**system/script/header**), et plus encore.

Item	Description
Title	Le libellé décrivant le script
Header	L'en-tête du script, sous la forme d'un objet. Peut être utilisé pour accéder au titre du script, au nom de l'auteur, sa version, la date, et d'autres champs.
Parent	Si le script a été évalué à partir d'un autre script, cet objet donne des informations pour ce script parent.
Path	La localisation (path) du fichier dans l'arborescence des fichiers, ou l'URL à partir de laquelle le script a été évalué.
Args	Les arguments du script. Ils sont passés depuis la ligne de commande, ou à partir de la fonction **do** qui a été utilisée pour évaluer le script.

Quelques exemples illustrant l'usage de l'objet **system/script** :

```
print system/script/title
```

```
print system/script/header/date

do system/script/args

do system/script/path/script.r
```

Le dernier exemple évalue un script appelé script.r, se trouvant dans le même répertoire (system/script/path) que le script en cours d'évaluation.

4.1 Options de programme

Les scripts peuvent aussi accéder aux options fournies à l'interpréteur REBOL quand il a été démarré. Elles peuvent être trouvées dans l'objet **system/option**.

Cet objet contient les champs suivants :

Item	Description
Home	Le chemin vers l'interpréteur, dans l'environnement de votre système d'exploitation. Il s'agit du chemin indiqué dans la variable HOME de votre environnement, ou de la base de registre si votre système l'exploite. C'est le chemin utilisé aussi pour trouver les fichiers **rebol.r** et **user.r**.
Script	Le nom du fichier de script initialement fourni quand l'interpréteur a été lancé.
Path	Le chemin vers le répertoire courant.
Args	Les arguments initiaux fournis à l'interpréteur en ligne de commande.
Do-arg	La chaîne fournie en argument à l'option --do en ligne de commande.

L'objet **system/options** peut aussi contenir des options supplémentaires qui ont été fournies en ligne de commande.
Saisissez :

```
probe system/options
```

pour étudier le contenu de l'objet **system/options**.

Exemples:

```
print system/options/script

probe system/options/args

print read system/options/home/user.r
```

[Retour au sommaire]

5. Exécution des scripts

Il y a deux manières d'exécuter un script : en tant que script initial, quand REBOL est démarré, ou à partir de la fonction **do**.

Pour exécuter un script au lancement de l'interpréteur, donnez le nom du script en ligne de commande juste après le nom de l'interpréteur (rebol ou rebol.exe) :

```
rebol script.r
```

Lors de l'initialisation de l'interpréteur, le script sera évalué.

A la fonction **do**, fournissez le nom de fichier du script en argument. Le fichier sera chargé dans l'interpréteur et évalué :

```
do %script.r

do http://www.rebol.com/script.r
```

La fonction **do** renvoie le résultat du script quand l'évaluation est terminée. Notez que le script doit inclure en en-tête (header) REBOL valide.

5.1 Chargement des scripts

Les scripts peuvent être chargés comme des données, avec la fonction **load**.

Cette fonction lit le script, et traduit le script en valeurs, mots et blocs, mais sans évaluer le script. Le résultat de la fonction **load** est un bloc, sauf si une seule valeur a été chargée, auquel cas cette valeur est retournée.

L'argument de la fonction **load** est un nom de fichier, une URL, ou une chaîne :

```
load %script.r
load %datafile.txt
load http://www.rebol.org/script.r
load "print now"
```

La fonction **load** réalise les étapes suivantes.

Elle

- lit le texte du fichier, de l'URL, ou de la chaîne de caractères.
- recherche un en-tête de script, s'il est présent.
- traduit les données trouvées après l'en-tête, s'il y en a.
- renvoie un bloc contenant les valeurs traduites.

Par exemple, si un script appelé *buy.r* contient le texte :

```
Buy 100 shares at $20.00 per share
```

il pourra être chargé avec la ligne :

```
data: load %buy.r
```

et il renverra dans un bloc :

```
probe data
[Buy 100 shares at $20.00 per share]
```

Remarquez que l'exemple "Buy" précédent est un dialecte de REBOL et non du code directement exécutable. Voir le chapitre 4 sur les Expressions pour plus d'information.

A noter aussi qu'un fichier ne nécessite pas un en-tête pour être chargé. L'en-tête est nécessaire seulement si le fichier doit être exécuté en tant que script.

La fonction **load** possède quelques raffinements dont voici la description :

Item	Description
/header	Inclue l'en-tête (header) s'il est présent
/next	Charge seulement la valeur suivante, une valeur à la fois. Ceci est pratique pour parser, analyser des scripts REBOL.
/markup	Traite le fichier comme un fichier HTML ou XHTML et renvoie un bloc permettant de manipuler les balises et le texte.

En principe, **load** ne renvoie pas l'en-tête du script. Mais, si le raffinement **/header** est utilisé, l'en-tête est le premier item du bloc renvoyé par **load**.

Le raffinement **/next** charge la valeur suivante et renvoie un bloc contenant deux valeurs. La première est la valeur suivante dans la série. La seconde valeur retournée est la position dans la chaîne suivant immédiatement le dernier item chargé.

Le raffinement **/markup** charge des données XML et HTML sous la forme d'un bloc de balises et de chaînes de caractères. Toutes les balises sont du type **tag!**. Les autres données sont traitées comme des chaînes de caractères.

Si le contenu du fichier suivant est chargé avec **load/markup** :

```
<title>This is an example</title>
```

```
```

un bloc sera créé :

```
probe data
[<title> "This is an example" </title>]
```

5.2 Sauvegarde des scripts

Des données peuvent être sauvegardées dans un fichier de script selon un format facilement récupérable sous REBOL avec la fonction **load**.

C'est une manière pratique de sauver des données et des blocs de données. Par ce moyen, il est possible de créer une mini base de données.

La fonction **save** attend deux arguments : un nom de fichier et, soit le bloc, soit la valeur à sauvegarder.

```
data: [Buy 100 shares at $20.00 per share]

save %data.r data
```

Les données *data* sont écrites au format texte pour REBOL, ce qui permet de les charger ultérieurement avec :

```
data: load %data.r
```

De simples valeurs peuvent aussi être sauvées et chargées. Par exemple, un horodatage peut être sauvé avec :

```
save %date.r now
```

et plus tard rechargé avec :

```
stamp: load %date.r
```

Dans l'exemple précédent, comme *stamp* est une valeur unique, elle n'est pas mise dans un bloc lorsqu'elle est chargée.

Pour sauvegarder un fichier script avec un en-tête, celui-ci doit être fourni en argument, à load avec le raffinement **/header**, soit sous la forme d'un objet ou sinon d'un bloc :

```
header: [Title: "This is an example"]

save/header %data.r data header
```

5.3 Commentaires dans les scripts

L'usage de commentaires est pratique pour clarifier l'objet de certains paragraphes de scripts. L'en-tête d'un script fournit la description générale du script et les commentaires une description plus courte des fonctions. C'est également une bonne idée de mettre des commentaires pour d'autres parties de votre code.

Un commentaire mono-ligne est précédé d'un point-virgule (**;**). Tout ce qui suit le point-virgule, et jusqu'à la fin de la ligne, est pris comme un commentaire :

```
zertplex: 10   ; set to the highest quality
```

Vous pouvez aussi utiliser des chaînes de caractères pour les commentaires. Par exemple, vous pouvez créer des commentaires sur plusieurs lignes avec une chaîne de caractères placée entre accolades.

```
{
    This is a long multilined comment.
}
```

Cette façon de commenter fonctionne uniquement si la chaîne n'est pas interprétée comme un argument d'une fonction. Si vous voulez être sûr qu'un commentaire multi-lignes soit reconnu comme tel et non interprété comme du code, faites-le précéder du mot **comment** :

```
comment {
    This is a long multilined comment.
}
```

La fonction **comment** indique à REBOL qu'il faut ignorer le bloc ou la chaîne qui le suit. Notez que les commentaires sous forme de blocs ou de chaînes font actuellement partie du bloc global du script. Faites attention de ne pas mettre ces commentaires dans des blocs de données, car ils pourraient être évalués comme des parties de données.

[Retour au sommaire]

6. Guide de Style

Les scripts REBOL n'ont pas de formalisme particulier. Vous pouvez écrire un script en utilisant l'indentation, la longueur de ligne ou les marqueurs de fin de ligne que vous voulez. Vous pouvez mettre chaque mot sur une ligne séparée ou les joindre ensemble sur une seule grande ligne.

Bien que le formatage de votre script n'affecte pas l'interpréteur, la présentation affecte, elle, la lisibilité. A cause de cela, REBOL Technologies suggère de suivre un certain style pour écrire vos scripts, qui va être décrit dans cette section.

Bien sûr, vous n'êtes pas obligés de suivre l'une ou l'autre de ses suggestions. Pourtant, le style du code est plus important qu'il ne paraît à première vue. La lisibilité et la réutilisation des scripts peuvent en être facilitées. Les utilisateurs peuvent juger la qualité de vos scripts par la clarté de

votre style. Un script alambiqué va souvent de pair avec un code alambiqué. Les codeurs expérimentés trouvent d'habitude qu'un style clair et cohérent rend leur code plus facile à produire, à maintenir et à contrôler.

6.1 Formatage

Utilisez les indications de style suivantes pour clarifier la présentation de vos scripts :

6.1.1 Indentez le contenu pour le clarifier

Le contenu d'un bloc est indenté, mais pas les crochets **[ ]** encadrant le bloc. C'est parce que les crochets ont un niveau de priorité plus important pour la syntaxe, et parce qu'ils définissent le bloc mais ne sont pas le contenu du bloc. De plus, il est plus facile de se focaliser sur les ruptures entre des blocs adjacents quand les crochets sont repérables.

Lorsque cela est possible, un crochet ouvrant **[** demeure sur la ligne avec l'expression qui lui est associée. Le crochet fermant **]** peut être suivi de plusieurs expressions au même niveau. Des règles identiques s'appliquent pour les parenthèses **( )** et les accolades **{ }**.

```
if check [do this and that]

if check [
    do this and do that
    do another thing
    do a few more things
]

either check [do something short][
    do something else]

either check [
    when an expression extends
    past the end of a block...
][
    this helps keep things
    straight
]

while [
    do a longer expression
    to see if it's true
][
    the end of the last block
    and start of the new one
    are at the WHILE level
]

adder: func [
    "This is an example function"
    arg1 "this is the first arg"
    arg2 "this is the second arg"
][
    arg1 + arg2
]
```

Une exception est faite pour les expressions qui appartiennent normalement à une ligne simple,

mais se prolongent sur plusieurs lignes :

```
if (this is a long conditional expression that
    breaks over a line and is indented
)[
    so this looks a bit odd
]
```

Ceci s'applique aussi à des valeurs qui sont normalement groupées ensemble, mais qui doivent être adaptées à la ligne.

```
[
    "Hitachi Precision Focus" $1000 10-Jul-1999
        "Computers Are Us"

    "Nuform Natural Keyboard" $70 20-Jul-1999
        "The Keyboard Store"
]
```

6.1.2 Taille standard pour les tabulations

La taille standard de tabulation pour REBOL est de quatre espaces. Comme les programmeurs utilisent différents éditeurs de texte pour les scripts, il est suggérer d'employer les espaces plutôt que les tabulations.

6.1.3 Détabuler avant de publier

Dans beaucoup de navigateurs, ou shells, ou lecteurs, le caractère de tabulation (ASCII 9) ne correspond pas à quatre espaces, donc utilisez votre éditeur ou REBOL pour ôter les tabulations d'un script avant de le publier sur le Net. La fonction suivante transforme chaque tabulation d'un fichier en un bloc de quatre espaces.

```
detab-file: func [file-name [file!]] [
    write file-name detab read file-name
]
detab-file %script.r
```

La fonction suivante convertit des tabulations à huit espaces en tabulations à quatre espaces :

```
detab-file: func [file-name [file!]] [
    write file-name detab entab/size read file-name 8
]
```

6.1.4 Limitez les longueurs de ligne à 80 caractères

Pour faciliter la lecture et la portabilité entre éditeurs de texte et clients email, limitez les lignes à 80 caractères. Les lignes trop longues sont mal formatées dans les clients email, sont difficiles à lire et posent des problèmes de chargement.

6.2 Libellés des mots

Les mots de votre code sont en premier lieu ce qui est exposé à un utilisateur, de sorte qu'il vous faut les choisir soigneusement. Un script doit être clair et concis. Lorsque c'est possible, les mots doivent être parlants. Voici les convention de nommage pour REBOL.

6.2.1 Utilisez les mots les plus courts ayant le sens le plus fort

Quand c'est possible, les mots directs donnent plus de sens :

```
size  time  send  wait  make  quit
```

Un mot à portée locale peut souvent être réduit à un mot simple. D'un autre côté, des mots plus descriptifs sont préférables pour les mots à portée globale.

6.2.2 Employez des mots entiers si possible

Ce que vous conservez en abrégeant un mot est rarement le meilleur.

Saisissez *date* et non pas *dt*, ou *image-file* et non pas *imgfl*.

6.2.3 Mettez un trait d'union aux mots ayant des libellés complexes

Le style standard est d'utiliser un trait d'union, et non la casse des caractères, pour distinguer des mots.

```
group-name image-file  clear-screen  bake-cake
```

6.2.4 Débutez les noms de fonction par un verbe

Les noms de fonction commencent avec un verbe et sont suivis par un nom, un adverbe, et un adjectif. Certains noms peuvent aussi être utilisés comme verbes.

```
make  print  scan  find  show  hide  take
rake-coals  find-age  clear-screen
```

Eviter autant que possible les mots inutiles. Par exemple, **quit** est aussi clair que **quit-system**.

Quand un nom est utilisé comme verbe, utilisez des caractères spéciaux comme le caractère (**?**) là où cela est possible. Par exemple, la fonction donnant la longueur d'une série est **length?**.

Voici d'autres fonctions REBOL utilisant cette convention de nommage :

```
size?  dir?  time?  modified?
```

6.2.5 Commencez les variables avec des noms

Les mots représentant des objets ou des variables qui manipulent des données devraient commencer par un nom. Ils peuvent aussi comprendre un adjectif si nécessaire :

```
image  sound  big-file  image-files  start-time
```

6.2.6 Utilisez des mots standards

Il y a des noms standards en REBOL qui devraient être utilisés pour des types d'opérations similaires. Par exemple :

```
make-blub           ; créer quelque chose de nouveau (make)
free-blub           ; libérer les ressources (free)
copy-blub           ; copier un contenu (copy)
to-blub             ; transformer en quelque chose (to-...)
insert-blub         ; insérer quelque chose (insert)
remove-blub         ; enlever (remove)
clear-blub          ; remettre à zéro
```

6.3 En-têtes de script

L'intérêt des en-têtes est clair. Les en-têtes donnent aux utilisateurs un résumé du script et permettent à d'autres scripts de traiter cette information (comme pour cataloguer de script). Un en-tête minimum de script fournit le titre, la date, le nom du fichier, et le sujet du script. D'autres champs peuvent aussi être fournis comme les auteurs, des notes, l'usage et les pré-requis.

```
REBOL [
    Title: "Local Area Defringer"
    Date:  1-Jun-1957
    File:  %defringe.r
    Purpose: {
        Stabilize the wide area ignition transcriber
        using a double ganged defringing algorithm.
    }
]
```

6.4 En-tête de fonctions

Il est commode d'avoir une description dans le bloc de spécification d'une fonction. Limitez ce texte à une ligne de 70 caractères ou moins.

Au sein de cette description, mentionnez quel type de valeur est attendu normalement par la fonction.

```
defringe: func [
    "Return the defringed localization radius."
    area "Topo area to defringe"
    time "Time allotted for operation"
    /cost num "Maximum cost permitted"
    /compound "Compound the calculation"
][
```

```
    ...code...
]
```

6.5 Nom des fichiers de script

La meilleure manière de nommer un fichier est de penser à la façon dont vous pourriez le retrouver dans quelques mois. Les noms courts et clairs sont assez souvent les meilleurs. Les noms complexes devraient être évités, à moins d'être significatifs.

De plus, en nommant le script, pensez à la manière dont le nom ressortira dans le listing d' un répertoire. Par exemple, conservez les fichiers ayant un lien entre eux en faisant débuter leur nom par un mot commun.

```
%net-start.r
%net-stop.r
%net-run.r
```

6.6 Insertion d'exemples

Le cas échéant, fournissez des exemples dans votre script pour présenter son fonctionnement et pour permettre aux utilisateurs de vérifier rapidement s'il fonctionne correctement sur leur système.

6.7 Déboguage incorporé

Il est souvent utile de construire des fonctions de débogage intégrées au script. C'est particulièrement vrai pour les scripts utilisant le réseau et manipulant des fichiers, et pour lesquels où il n'est pas souhaitable d'émettre et d'écrire des fichiers tant qu'on est en mode de test. De tels tests peuvent être gérés avec une variable de contrôle au début du script.

```
verbose: on
check-data: off
```

6.8 Minimiser le nombre de variables globales

Dans les longs scripts, et autant que possible, évitez d'utiliser des variables globales qui portent leur état interne d'une partie ou d'une fonction à l'autre.

Pour les petits scripts, ce n'est pas toujours pratique. Mais reconnaissez que ces scripts courts peuvent devenir au fil du temps de plus en plus grands.

Si vous avez un ensemble de variables globales fortement reliées entre elles, pensez à utiliser un objet pour les regrouper :

```
user: make object! [
    name:  "Fred Dref"
    age:   94
    phone: 707-555-1234
    email: dref@fred.dom
]
```

[Retour au sommaire]

7. Clarification du format d'un script

Voici un court script qui peut être utilisé pour nettoyer l'indentation d'un script. Il fonctionne en analysant la syntaxe REBOL et en reconstruisant chaque ligne du script. Cet exemple peut être trouvé dans la bibliothèque de scripts sur www.REBOL.com.

```
out: none ; output text
spaced: off ; add extra bracket spacing
indent: "" ; holds indentation tabs

emit-line: func [] [append out newline]

emit-space: func [pos] [
    append out either newline = last out [indent] [
        pick [#" " ""] found? any [
            spaced
            not any [find "[(" last out
                     find ")]" first pos]
        ]
    ]
]

emit: func [from to] [
    emit-space from append out copy/part from to
]

clean-script: func [
    "Returns new script text with standard spacing."
    script "Original Script text"
    /spacey "Optional spaces near brackets/parens"
    /local str new
] [
    spaced: found? spacey
    out: append clear copy script newline
    parse script blk-rule: [
        some [
            str:
            newline (emit-line) |
            #";" [thru newline | to end] new:
                (emit str new) |
            [#"[" | #"("]
                (emit str 1 append indent tab)
                blk-rule |
            [#"]" | #")"]
                (remove indent emit str 1) |
            skip (set [value new]
                load/next str emit str new) :new
        ]
    ]
    remove out ; remove first char
]

script: clean-script read %script.r

write %new-script.r script
```

Chapitre 6 - Les Séries

Ce document est la traduction française du Chapitre 6 du User Guide de REBOL/Core, qui concerne les Séries.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
11 avril 2005 7:15	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Concepts de base

Le concept de série est simple, et c'est un concept fondamental employé partout dans REBOL. Afin de comprendre REBOL, vous devez comprendre comment créer et manipuler une série.

Une série est un ensemble de valeurs arrangées selon un ordre spécifique. C'est aussi simple que cela.

```
1 2 3 4
```

```
A B C D
"ABCD"
10:30 4:20 7:11
```

Il existe différents types de séries en REBOL.
Un bloc, une chaîne de caractères, une liste, une URL, un chemin (path), un email, un fichier, une balise, un binaire, un jeu de caractères, un port, une table de hachage, une structure codée, et une image : voilà des séries qui peuvent être manipulées avec le même ensemble commun de fonctions.

2.1 Parcourir une série

Une série est un ensemble ordonné de valeurs, vous pouvez la parcourir d'une position à une autre.
En guise d'exemple, prenons une série de trois couleurs définie par le bloc suivant :

```
colors: [red green blue]
```

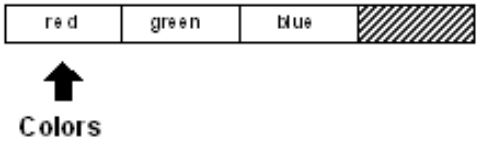
Il n'y a rien de particulier concernant ce bloc.
C'est une série contenant trois mots. Elle constitue un ensemble de valeurs : red, green, et blue.
Les valeurs sont organisées selon cet ordre : red est en premier, green en second, et blue en troisième.

La première position dans le bloc est appelée son début : **head**.
C'est la position qu'occupe le mot : red. La dernière position dans le bloc est appelée sa fin : **tail**.
C'est une position située immédiatement après le dernier mot dans le bloc.

Si vous dessiniez un diagramme du bloc, il ressemblerait à ceci :

Remarquez que la fin (**tail**) est juste **après** la fin du bloc. Ce point est important, il sera clarifié un peu plus loin.

La variable "colors" est utilisée pour faire référence au bloc. Elle est actuellement positionnée sur le début **head** du bloc :




```
print head? colors
true
```

La variable "colors" est analogue à un curseur de position pour le bloc.

```
print index? colors
```

```
1
```

Le bloc a une longueur de : trois .

```
print length? colors
3
```

Le premier élément dans le bloc est :

```
print first colors
red
```

Le second élément dans le bloc est :

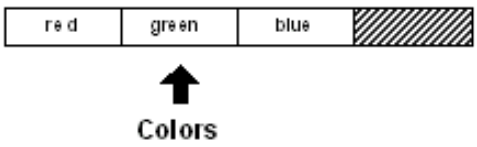
```
print second colors
green
```

Vous pouvez déplacer la variable "colors", notre curseur de position, le long du bloc en utilisant différentes fonctions.

Pour déplacer la variable "colors" à la position suivante dans le bloc de couleurs, utilisez la fonction : **next**

```
colors: next colors
```

La fonction **next** permet de se déplacer d'une valeur plus loin dans le bloc et retourne la nouvelle position comme résultat. La variable "colors" est donc maintenant mise sur cette nouvelle position :



La position de la variable "colors" a donc été changée. Maintenant cette variable ne pointe plus sur le début (**head**) du bloc :

```
print head? colors
false
```

Elle pointe sur la deuxième position dans le bloc :

```
print index? colors
2
```

D'autre part, si vous cherchez à obtenir le premier item de "colors", vous aurez :

```
print first colors
green
```

La position de la valeur retournée par la fonction first est relative à la position courante de la variable "colors", le curseur, dans le bloc.

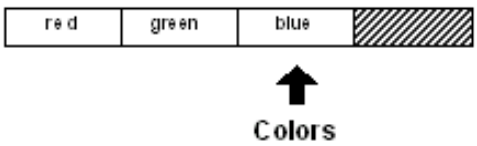
La valeur retournée n'est pas la première couleur dans le bloc, mais la première couleur qui suit **immédiatement** la position courante du curseur "colors" dans le bloc.
De la même manière, si vous recherchez la longueur du bloc ou la seconde couleur, celles-ci sont aussi relatives à la position courante :

```
print length? Colors
2
print second colors
blue
```

Vous pouvez encore vous déplacer à la position suivante, et observer un fonctionnement similaire :

```
colors: next colors
print index? colors
3
print first colors
blue
print length? colors
1
```

Le schéma du bloc ressemble à présent à ceci :



La variable "colors" est maintenant sur la dernière couleur dans le bloc, mais elle n'est pas encore à la position finale (**tail**).

```
print tail? colors
false
```

Pour atteindre la fin, il faut encore se déplacer, avec **next**, à la position suivante :

```
colors: next colors
```

Maintenant, la variable "colors" pointe sur la fin du bloc (**tail**). Elle n'est plus positionnée sur une couleur valide. La dernière couleur du bloc a été dépassée.

Si vous essayez le code suivant, vous obtiendrez :

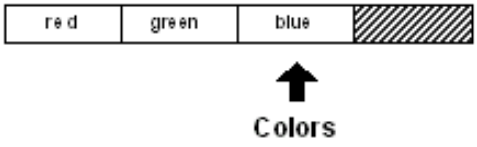
```
print tail? colors
true
print index? colors
4
print length? Colors
0
print first colors
** Script Error: Out of range or past end.
** Where: print first colors
```

Vous recevez une erreur dans le dernier cas parce qu'il n'y a plus d'item valide lorsque la fin du bloc a été dépassée.

Il est aussi possible de se déplacer en arrière dans le bloc. Si vous écrivez :

```
colors: back colors
```

vous vous déplacerez d'une position en arrière dans la série.



Tout le code suivant fonctionnera comme auparavant :

```
print index? colors
3
print first colors
blue
```

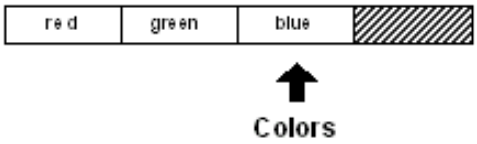
2.2 Sauts de valeurs

Les exemples précédents montraient comment se déplacer d'un élèment à la fois dans la série. Cependant, vous voudrez parfois parcourir la série en sautant plusieurs items à la fois, avec la fonction **skip**.
Supposons que la variable "colors", notre curseur, soit placée au début de la série :

Vous pouvez aller en avant par saut de deux éléments avec le code suivant :

```
colors: skip colors 2
```

La fonction **skip** est similaire à la fonction **next** en ceci qu'elle retourne également la série à la nouvelle position.



Pour contrôler l'index de la nouvelle position :

```
print index? colors
3
print first colors
blue
```

Pour un déplacement en arrière, utilisez **skip** avec une valeur négative :

```
colors: skip colors -1
```

Ce code ci-dessus a le même effet que la fonction **back**. En effet, un saut de -1 fait se déplacer d'un élément en arrière.


```
print first colors
green
```

Notez que vous ne pouvez pas dépasser la fin ou le début d'une série.
Si vous essayez de faire cela, **skip** ira seulement aussi loin que possible. La fonction ne génerera pas d'erreur. Si vous vous déplacez trop loin en avant, **skip** ira à la fin de la série :

```
colors: skip colors 20
print tail? colors
true
```

Si vous allez trop loin en arrière, **skip** renverra la série à son début :

```
colors: skip colors -100
print head? colors
true
```

Pour sauter directement au début de la série, utilisez plutôt la fonction **head** :

```
colors: head colors
print head? colors
true
print first colors
red
```

Vous pouvez retourner à la fin avec la fonction **tail**:

```
colors: tail colors
print tail? colors
true
```

2.3 Extraire des valeurs

Certains des exemples précédents faisaient usage des fonctions **first** et **second** pour extraire d'une série des valeurs spécifiques. L'ensemble des fonctions ordinales est :

```
first
second
third
fourth
fifth
sixth
seventh
eighth
ninth
tenth
last
```

> N.d.T.: les fonctions ordinales supérieures à fifth ont été rajoutées dans les dernières mises à jour de Core.

Ces fonctions ordinales sont fournies pour rendre plus pratique la récupération de valeurs à partir de positions simples dans une série. Voici quelques exemples :

```
colors: [red green blue gold indigo teal]
print first colors
red
print third colors
blue
print fifth colors
indigo
print last colors
teal
```

Pour une extraction à partir d'une position numérique, utilisez la fonction **pick** :

```
print pick colors 3
blue
print pick colors 5
indigo
```

Une écriture simplifiée consiste à utiliser un "path" :

```
print colors/3
blue
print colors/5
indigo
```

Rappelez-vous, comme vu plus tôt, que la récupération d'une valeur est effectuée **relativement** à la variable de série (le curseur) que vous fournissez.
Si la variable "colors" pointait une autre position dans la série, votre résultat aurait été différent.

L'extraction d'une valeur après la fin de la série retournerait une erreur avec les fonctions ordinales, et **none** avec la fonction **pick** ou un "path" utilisant **pick**.

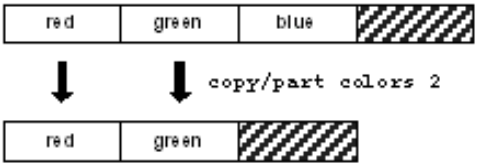
```
print pick colors 10
none
print colors/10
none
```

2.4 Extraire une sous-série

Vous pouvez extraire plusieurs valeurs d'une série avec la fonction **copy**.
Pour cela, utilisez **copy** avec le raffinement **/part**, en spécifiant le nombre de valeurs à extraire :

```
colors: [red green blue]
sub-colors: copy/part colors 2
probe sub-colors
[red green]
```

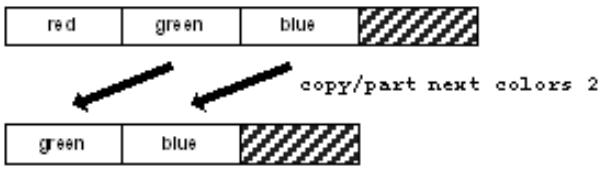
Schématiquement, ceci devrait donner quelque chose comme :



Pour copier une sous-série depuis n'importe quelle position à l'intérieur de la série, il faut d'abord se positionner sur une position de départ.
L'exemple suivant montre comment se déplacer à la seconde position dans la série, avec la fonction **next**, avant d'effectuer la copie :

```
sub-colors: copy/part next colors 2
probe sub-colors
[green blue]
```

Ce qui devrait ressembler à :



la longueur de la liste à copier peut être indiquée comme position de fin, tout comme un nombre d'exemplaires.
Noter que la position indique où la copie devrait s'arrêter, pas la position de fin.

```
probe copy/part colors next colors
[red]
probe copy/part colors back tail colors
[red green]
probe copy/part next colors back tail colors
[green]
```

Ceci peut être utile avec la fonction **find** qui retourne comme résultat la position de la série :

```
file: %image.jpg
print copy/part file find file "."
image
```

2.5 Insertion et Ajout

Vous pouvez insérer une ou plusieurs nouvelles valeurs à n'importe quel endroit dans la série en utilisant la fonction **insert**.

Quand on insére une valeur à une position dans la série, un espace est créé en décalant les valeurs qui suivent vers la fin de la série.

Par exemple, le bloc :

```
colors: [red green]
```

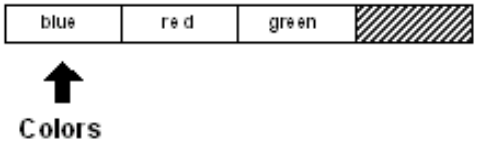
devrait ressembler à :

Pour insérer une nouvelle valeur au début du bloc, là où le curseur (la variable "colors") est à présent positionné :

```
insert colors 'blue
```

Les mots 'red et 'green sont décalés et le mot 'blue est inséré au début de la liste (il a été préfixé par une apostrophe car il s'agit d'un mot, à ne pas évaluer).

Notez que la variable "colors" demeure positionnée sur le début de la liste.




```
probe colors
[blue red green]
```

Notez encore que la valeur de retour de la fonction **insert** n'a pas été utilisée :
elle n'est pas passée à une variable ou à une fonction.

La ligne suivante permettrait d'affecter cette valeur retournée par **insert** à la variable "colors" :

```
colors: insert colors 'blue
```

En ce cas, l'action sur le bloc serait identique à l'insertion de l'exemple précédent, mais la position du curseur "colors" changerait. En effet, sa position devient ici la valeur retournée par **insert**.

La position renvoyée à partir d'**insert** est celle suivant immédiatement le point d'insertion.

Une insertion peut être faite n'importe où dans la série. La position de l'insertion peut être spécifiée, et elle peut inclure la fin (**tail**).

Insérer une valeur en fin de série revient à faire un ajout à celle-ci.

```
colors: tail colors
insert colors 'gold
probe colors
[blue red green gold]
```

Avant l'insertion :

Après l'insertion :

Le mot 'gold a été inséré en fin de série.

Un autre moyen d'insérer des valeurs à la fin d'une série est d'utiliser la fonction **append**.

Append fonctionne comme **insert** mais insére toujours les valeurs en fin de liste. L'exemple précédent deviendrait :

```
append colors 'gold
```

Le résultat est identique.

Les fonctions **insert** et **append** accepte aussi un bloc d'argument à insérer. Par exemple,

```
colors: [red green]
insert colors [blue yellow orange]
probe colors
[blue yellow orange red green]
```

Si vous voulez insérer de nouvelles valeurs entre les mots red et green :

```
colors: [red green]
insert next colors [blue yellow orange]
probe colors
[red blue yellow orange green]
```

Les fonctions **insert** et **append** possédent d'autres possibilités, avec leurs raffinements, qui seront détaillées ultérieurement dans un autre section.

2.6 Enlever des valeurs

Vous pouvez ôter une ou plusieurs valeurs depuis n'importe quel endroit de la série en utilisant la fonction **remove**.

Par exemple, avec le bloc :

```
colors: [red green blue gold]
```

comme schématisé ici :

vous pouvez enlever le premier élément du bloc avec la ligne :

```
remove colors
```

Le bloc devient :

Son contenu peut être édité avec :

```
probe colors
[green blue gold]
```

La fonction **remove** ôte des valeurs relativement à la position courante de notre curseur, la variable "colors".
Vous pouvez enlever des valeurs depuis n'importe quel endroit dans la série en indiquant la position.

```
remove next colors
```

Le bloc ressemble à présent à :

Plusieurs éléments peuvent être ôtés en utilisant le raffinement **/part**.

```
remove/part colors 2
```

Ceci enlève les valeurs restantes, laissant un bloc vide :

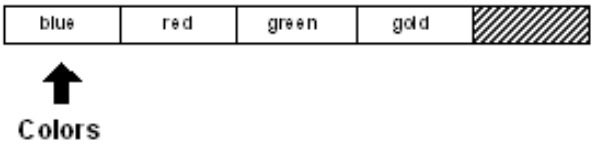
Comme pour **insert/part**, l'argument fourni à **remove/part** peut aussi être une position à l'intérieur du bloc.

Effacer toutes les valeurs restantes est une opération courante.
La fonction **clear** permet de faire cela directement.
Clear enlève toutes les valeurs depuis la position courante jusqu'à la fin.

Par exemple :

```
Colors: [blue red green gold]
```

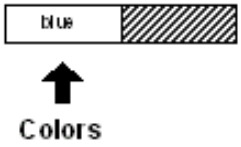
comme montré ici :



Tout ce qui se trouve après le mot blue peut être ôté avec :

```
clear next colors
```

Le bloc devient :



Vous pouvez entièrement vider le bloc avec :

```
clear colors
```

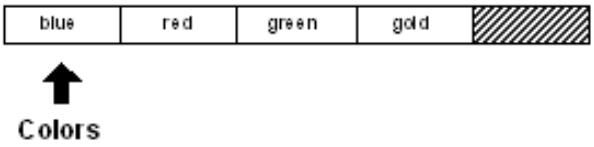
2.7 Modifier des valeurs

Un jeu suplémentaire de fonctions est fourni pour permettre la modification de valeurs dans une série.

La fonction **change** remplace une ou plusieurs valeurs par des nouvelles. Quoique ceci puisse être fait avec les fonctions **insert** et **append**, il est plus efficace d'utiliser **change**.

Le bloc suivant est défini :

```
colors: [blue red green gold]
```



Sa deuxième valeur peut être changée avec la ligne :

```
change next colors 'yellow
```

et le bloc devient :

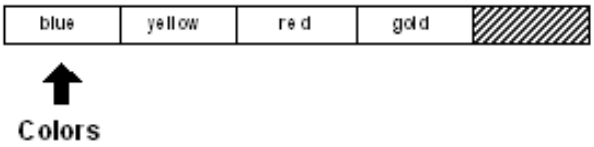
Le bloc est devenu à présent :

```
probe colors
[blue yellow green gold]
```

La fonction **poke** vous permet d'effectuer un changement à une position particulière, relativement au curseur, la variable "colors".
La fonction **poke** est similaire à la fonction **pick** décrite précédemment.

```
poke colors 3 'red
```

Le bloc est à présent :



avec :

```
probe colors
[blue yellow red gold]
```

La fonction **change** possède des raffinements qui seront décrits plus loin.

[Retour au sommaire]

3. Fonctions relatives aux séries

Voici un résumé des fonctions relatives aux séries.

La plupart d'entre elles ont déjà été décrites en détail dans les sections précédentes. D'autres seront explicitées plus loin.

3.1 Fonctions de création

Fonction	Description
make	Crée une nouvelle série d'un certain type.
copy	Copie une série

3.2 Fonction de navigation

Fonction	Description
next	Retourne la position suivante dans la série.
back	Retourne la position précédente dans la série.
head	Retourne la position du début dans la série.
tail	Retourne la position de fin dans la série.
skip	Retourne la position plus ou moins un entier.
at	Retourne la position plus ou moins un entier, mais en utilisant la même indexation que la fonction **pick**.

3.3 Fonctions d'Information

Fonction	Description

head?	Retourne true si le curseur est sur le début de la série.
tail?	Retourne true si le curseur est sur la fin de la série.
index?	retourne l'index par-rapport au début de la série.
length?	retourne la longueur d'une série à partir de la position courante.
offset?	retourne la distance entre deux positions dans la série.
empty?	retourne true si la série est vide à partir de cette position.

3.4 Fonctions d'extraction

Fonction	Description
pick	extrait une valeur unique à partir d'une position dans une série.
copy/part	extrait une sous-série à partir d'une série.
first	extrait la première valeur d'une série.
second	extrait la seconde valeur d'une série.
third	extrait la troisième valeur d'une série.
fourth	extrait la quatrième valeur d'une série.
fifth	extrait la cinquième valeur d'une série.
sixth	extrait la sixième valeur d'une série.
seventh	extrait la septième valeur d'une série.
eigth	extrait la huitième valeur d'une série.
ninth	extrait la neuvième valeur d'une série.
tenth	extrait la dixième valeur d'une série.

last	extrait la dernière valeur d'une série.

3.5 Fonction de modification

Fonction	Description
insert	insére des valeurs dans une série.
append	ajoute des valeurs à la fin d'une série.
remove	ôte des valeurs d'une série.
clear	efface les valeurs de la position courante jusqu'à la fin de la série.
change	modifie les valeurs dans une série.
poke	modifie les valeurs à une position donnée, dans une série.

3.6 Fonctions de recherche

Fonction	Description
find	recherche une valeur dans une série.
select	recherche une valeur dans une série et en cas de succés renvoie les valeurs qui suivent
replace	cherche et remplace des valeurs dans une série.
parse	parse les valeurs dans une série.

3.7 Fonctions de tri

Fonction	Description
sort	trie les valeurs d'une série dans un ordre.
reverse	inverse l'ordre des valeurs dans une série

3.8 Fonctions de groupes de données

Fonction	Description
unique	retourne un ensemble de données uniques, en ayant supprimé les doublons.
intersect	renvoie seulement les valeurs trouvées communes aux deux séries.
union	retourne l'union de deux séries.
difference	renvoie les valeurs non communes à chaque série.
exclude	retourne toutes les valeurs de la *première* série en argument, moins celles, communes aux deux, de la deuxième série

[Retour au sommaire]

4. Types de données des séries

Tous les types de données des séries peuvent être regroupés en deux grandes catégories. Chaque catégorie comprend la valeur du type de données (*datatype*) et la fonction testant ce type.

4.1 Types bloc

Type du bloc	Description
Block!	blocs de valeurs
Paren!	blocs de valeurs entre parenthèses
Path!	paths de valeurs
List!	listes
Hash!	tableaux associatifs

4.2 Types Chaîne de caractères

Type de chaîne	Description
String!	chaînes de caractères

Binary!	suite d'octets
Tag!	balises HTML et XML
File!	noms de fichiers
URL!	urls
Email!	emails
Image!	données d'Image
Issue!	codes particuliers

4.3 Pseudo-types

Les types de données de séries sont regroupés aussi en quelques pseudo-types qui rendent plus facile le test d'un type et la gestion des arguments de fonctions :

Pseudo-type	Description
series!	un type de donnée "series"
any-block!	n'importe que type de donnés "bloc"
any-string!	n'importe quel type de données "chaîne"

4.4 Fonction pour tester le type

Tests du type Bloc :

```
block? paren? path? list? hash?
```

Tests du type chaîne :

```
string? binary? tag? file? url? email? image? issue?
```

Autres fonctions de tests de pseudo-type :

```
series? any-block? any-string?
```

[Retour au sommaire]

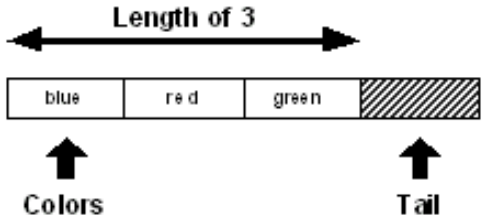
5. Information sur les séries

5.1 Length?

La longueur d'une série est le nombre d'items (éléments pour un bloc, ou caractères pour une chaîne) depuis la position courante jusqu'à la fin de la série.
La fonction **length?** renvoie le nombre d'items jusqu'à la fin.

```
colors: [blue red green]
print length? colors
3
```

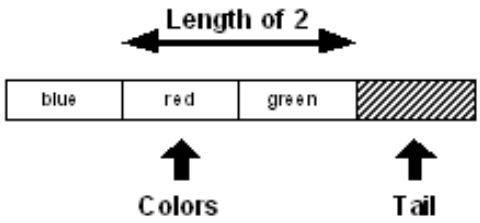
Les trois éléments sont comptabilisés pour le calcul de la longueur :



Si la variable "colors" est avancée à la position suivante,

```
colors: next colors
print length? colors
2
```

la longueur devient : deux .



Autres exemples d'usage de **length?** :

```
print length? "Ukiah"
5
print length? []
0
print length? ""
0
data: [1 2 3 4 5 6 7 8]
print length? data
8
```

```
data: next data
print length? data
7
data: skip data 5
print length? data
2
```

5.2 Head?

Le début de la série est la position de sa première valeur.
Si une série est à son début, la fonction **head?** renvoie *true* :

```
data: [1 2 3 4 5]
print head? data
true
data: next data
print head? data
false
```

5.3 Tail?

La fin de la série est la position qui suit immédiatement la dernière valeur valide.
Si la variable de série pointe sur la fin, la fonction **tail?** renverra *true* :

```
data: [1 2 3 4 5]
print tail? data
false
data: tail data
print tail? data
true
```

5.4 Empty?

La fonction **empty?** est équivalente à la fonction **tail?**.

```
print empty? data
true
```

Si la fonction **empty?** renvoie *true*, cela signifie qu'il n'y a plus de valeurs entre la position courante et la fin de la série.
Cependant, *il peut rester des valeurs dans la série*.
Ces valeurs peuvent être présentes avant la position courante.
Si vous voulez déterminer si la série est vide *du début à la fin*, utilisez :

```
print empty? head data
false
```

5.5 Index?

L'index est la position dans la série relativement à son début. Afin de connaître cette information pour une variable de série, il faut utiliser la fonction **index?** :

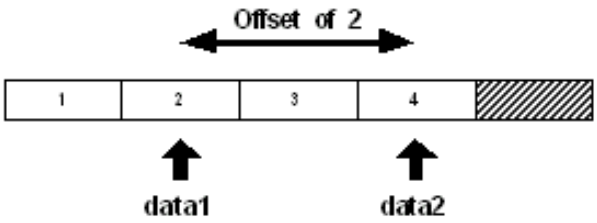
```
data: [1 2 3 4 5]
print index? data
1
data: next data
print index? data
2
data: tail data
print index? data
6
```

5.6 Offset?

L'écart entre deux positions au sein d'une série peut être déterminé avec la fonction **offset?**.

```
data: [1 2 3 4]
data1: next data
data2: back tail data
print offset? data1 data2
4
```

Dans cet exemple, l'écart est la différence entre la position 2 et la position 4 :



[Retour au sommaire]

6. Créer et copier une série

Une nouvelle série peut être créée avec les fonctions **make** et **copy**.

La fonction **make** peut servir à créer une nouvelle série à partir d'un type de données et d'une taille initiale. La taille est une estimation de ce qui serait nécessaire pour la série.
Si la taille initiale est trop petite, celle-ci sera automatiquement augmentée, mais avec une petite perte de performances.

```
block: make block! 50

string: make string! 10000
list: make list! 128
file: make file! 64
```

La fonction **copy** crée une nouvelle série en dupliquant une série existante :

```
string: copy "Message in a bottle"
new-string: copy string
block: copy [1 2 3 4 5]
new-block: copy block
```

La fonction **copy** est également importante, en pratique, avec les fonctions qui modifient le contenu d'une série.

Par exemple, si vous voulez changer la casse d'une chaîne de caractères sans modifier l'original, employez **copy** :

```
string: uppercase copy "Message in a bottle"
```

6.1 Copie partielle

La fonction **copy** utilise le raffinement **/part** qui prend un seul et unique argument, pouvant être soit un nombre entier (le nombre à copier d'éléments de la série), soit une position dans la série indiquant la dernière position pour la copie.

```
str: "Message in a bottle"
print str
Message in a bottle
print copy/part str find str " "
Message
new-str: copy/part (find str "in") (find str "bottle")
print new-str
in a
blk: [ages [10 12 32] sizes [100 20 30]]
new-blk: copy/part blk 2
probe new-blk
[ages [10 12 32]]
```

6.2 Copies de sous-séries

Des blocs peuvent contenir d'autres blocs, et/ou des chaînes. Quand un tel bloc est copié, ses sous-series ne le sont pas.
Les sous-séries sont référencées directement et ont les mêmes données de série que le bloc original. Si vous modifiez l'une ou l'autre de ces sous-séries, vous les modifiez également dans le bloc d'origine.

Le raffinement **copy/deep** forcera la copie de toutes les sous-séries à l'intérieur d'un bloc :

```
blk-one: ["abc" [1 2 3]]
probe blk-one
["abc" [1 2 3]]
```

L'exemple suivant effectue une copie normale de "blk-one" vers "blk-two" :

```
blk-two: copy blk-one
probe blk-one
["abc" [1 2 3]]
probe blk-two
["abc" [1 2 3]]
```

Si la chaîne (ou le bloc) contenue dans "blk-two" est modifiée, les valeurs contenues dans la série blk-one sont elles aussi modifiées.

```
append blk-two/1 "DEF"
append blk-two/2 [4 5 6]
probe blk-one
["abcDEF" [1 2 3 4 5 6]]
probe blk-two
["abcDEF" [1 2 3 4 5 6]]
```

Utiliser **copy/deep** permet la copie de toutes les valeurs de type "série" trouvées dans le bloc :

```
blk-two: copy/deep blk-one
append blk-two/1 "ghi"
append blk-two/2 [7 8 9]
probe blk-one
["abcDEF" [1 2 3 4 5 6]]
probe blk-two
["abcDEFghi" [1 2 3 4 5 6 7 8 9]]
```

6.3 Copie afin d'initialiser

L'utilisation de **copy** sur une série de type chaîne ou bloc permet de créer une série unique :

```
str: copy ""
blk: copy []
```

En utilisant **copy**, on s'assure qu'une nouvelle série sera initialisée pour un mot chaque fois que ce mot sera initialisé.

Voici un exemple qui illustre l'importance de cela :

```
print-it: func [/local str] [
    str: ""
    insert str "ha"
    print str
]
print-it
ha
```

```
print-it
haha
print-it
hahaha
```

Dans cet exemple, parce que **copy** n'a pas été utilisé, la série vide "str", de type chaîne, est modifiée à chaque appel de la fonction "print-it".
La chaîne "ha" est insérée dans la série "str" chaque fois que "print-it" est appelée.

L'examen du code source de print-it montre le noeud du problème :

```
source print-it
print-it: func [/local str] [
    str: "hahaha"         <-- ici
    insert str "ha"
    print str
]
```

Bien que "str" soit une variable locale, sa valeur est globale. Pour éviter ce problème, la fonction devrait copier une chaîne vide ou utiliser **make string!**.

```
print-it: func [/local str] [
    str: copy ""
    insert str "ha"
    print str
]
print-it
ha
print-it
ha
print-it
ha
```

[Retour au sommaire]

7. Iteration sur une série

Une boucle est nécessaire pour parcourir la série. Il existe quelques fonctions pour cela qui peuvent aider à automatiser ce processus d'itération.

7.1 Boucle Foreach

La fonction **foreach** permet de parcourir une série en ayant définit un mot ou plusieurs mots décrivant les valeurs dans la série.
La fonction **foreach** prend trois arguments : un mot ou un bloc de mots qui manipule les valeurs pour chaque itération, la série, et finalement un bloc à évaluer lors de chaque itération.

```
colors: [red green blue yellow orange gold]
foreach color colors [print color]
red
green
```

```
blue
yellow
orange
gold
foreach [c1 c2] colors [print [c1 c2]]
red green
blue yellow
orange gold
foreach [c1 c2 c3] colors [print [c1 c2 c3]]
red green blue
yellow orange gold
```

Ceci est très pratique avec des blocs contenant des données reliées entre elles :

```
people: [
    "Bob" bob@example.com 12
    "Tom" tom@example.net 40
    "Sam" sam@example.org 22
]
foreach [name email age] people [
    print [name email age]
]
Bob bob@example.com 12
Tom tom@example.net 40
Sam sam@example.org 22
```

Remarquez que la fonction **foreach** n'avance pas l'index courant dans la série, de sorte qu'il n'est pas nécessaire de remettre à zéro la variable représentant la série.

7.2 Boucle While

L'approche la plus flexible est d'utiliser une boucle **while**, qui vous permet de faire sans problèmes ce que vous voulez avec la série :

```
colors: [red green blue yellow orange]
while [not tail? colors] [
    print first colors
    colors: next colors
]
red
green
blue
yellow
orange
```

La méthode ci-dessous montre comment insérer des valeurs sans saisir de doublons :

```
colors: head colors
while [not tail? colors] [
    if colors/1 = 'yellow [
        colors: insert colors 'blue
```

```
    ]
    colors: next colors
]
```

L'exemple illustre aussi comment **insert** renvoie la position immédiatement suivant l'insertion.

Pour effacer une valeur sans en sauter accidentellement une , utilisez le code suivant :

```
colors: head colors
while [not tail? colors] [
    either colors/1 = 'blue [
        remove colors
    ][
        colors: next colors
    ]
]
```

Remarquez que si **remove** a été utilisée, la fonction **next** n'est pas appelée.

7.3 Boucle Forall

La fonction **forall** est identique à **while**, mais simplifie encore l'approche.
La boucle avec **forall** démarre à partir de l'index courant et parcourt la série jusqu'à la fin en évaluant un bloc à chaque itération. La fonction **forall** prend deux arguments : une variable de série et le bloc à évaluer à chaque itération.

```
colors: [red green blue yellow orange]
forall colors [print first colors]
red
green
blue
yellow
orange
```

Forall parcourt la série en avancant le curseur à chaque itération, de sorte que la variable de série "colors" sera positionnée sur la fin de la série, lorsque la boucle sera terminée :

```
print tail? colors
true
```

Par conséquent, cette variable de série "colors" doit être remise à zéro avant d'être à nouveau employée :

```
colors: head colors
```

Egalement, si un bloc évalué modifie la série, attention à éviter les trous ou les répétitions de valeurs.

La fonction **forall** fonctionne bien dans la plupart des cas; mais si vous avez un doute, utilisez **while** à la place.

```
forall colors [
    if colors/1 = 'blue [remove colors]
    print first colors
]
red
green
yellow
orange
```

7.4 Boucle Forskip

Tout comme **forall**, la fonction **forskip** parcourt la série en partant de la position courante, mais saute un nombre spécifié de valeurs à chaque fois.

La fonction **forskip** prend trois arguments : une variable de série, le nombre de valeurs à sauter entre deux itérations, et le bloc à évaluer à chaque itération.

```
colors: [red green blue yellow orange]
forskip colors 2 [print first colors]
red
blue
orange
```

La fonction **forskip** laisse la série à sa fin, nécessitant de la remettre au début si besoin.

```
print tail? colors
true
colors: head colors
```

7.5 La fonction Break

N'importe laquelle de ces boucles peut être arrêtée à tout moment avec la fonction **break** placée dans le bloc d'évaluation.

Voyez le chapitre du Manuel Utilisateur concernant les Expressions pour plus d'informations sur la fonction **break**.

[Retour au sommaire]

8. Recherche dans une série

La fonction **find** recherche au travers d'une série de type bloc ou chaîne une valeur ou un modèle. Cette fonction possède de nombreux raffinements qui permettent beaucoup de variations dans les paramètres de recherche.

8.1 Recherche simple

L'usage le pus courant et le plus simple pour la fonction **find** est de rechercher une valeur dans un bloc ou une chaîne. Dans ce cas, **find** nécessite seulement deux arguments :
la série où chercher et la valeur à trouver.

Un exemple d'utilisation de **find** sur un bloc :

```
colors: [red green blue yellow orange]
where: find colors 'blue
probe where
[blue yellow orange]
print first where
blue
```

La fonction **find** peut aussi rechercher des valeurs selon le type de données.

Ceci peut être très utile :

```
items: [10:30 20-Feb-2000 Cindy "United"]
where: find items date!
print first where
20-Feb-2000
where: find items string!
print first where
United
```

Un exemple d'utilisation de **find** sur une chaîne est :

```
colors: "red green blue yellow orange"
where: find colors "blue"
print where
blue yellow orange
```

Lorsqu'une recherche échoue, la valeur **none** est renvoyée.

```
colors: [red green blue yellow orange]
probe find colors 'indigo
none
```

8.2 Résumé des raffinements

Find possède plusieurs raffinements qui autorisent une grande variété de paramétres de recherche :

Raffinement	Description

/part	limite la recherche dans une série à une longueur donnée ou une position de fin.
/only	Manipule une valeur de série comme une valeur unique
/case	Utilise une comparaison de chaînes sensible à la casse (maj/min).
/any	Permet d'utiliser des caractères jokers ("wildcards") autorisant des correspondances avec n'importe quel(s) caractère(s) : un astérisque (*) dans le modéle signifie : n'importe quelle chaîne, et un point d'interrogation (?) correspond à : n'importe quel caractère.
/with	Permet l'usage de caractères jokers ("wildcards") avec des caractères différents de l'astérisque (*) et du point d'interrogation (?). Ceci permet d'avoir un modéle contenant des astérisques et des points d'interrogation.
/match	Recherche un modéle commencant à la position courante de la série, plutôt que de chercher la première occurence d'une valeur ou d'une chaîne. Renvoie la position de fin de la correspondance si celle-ci est trouvée.
/tail	renvoie la position qui suit la correspondance sur une recherche fructueuse, plutôt que de renvoyer la position de la correspondance.
/last	Recherche en arrière d'une correspondance, en commencant à la fin de la série.
/reverse	Recherche en arrière d'une correspondance, en commencant à la position courante.

8.3 Recherche partielle

Le raffinement **/part** permet que la recherche soit limitée à une portion spécifique de la série.

par exemple, vous pouvez vouloir restreindre une recherche à une ligne donnée ou à une portion de texte. Tout comme **insert/part** et **remove/part**, **find/part** prend en argument soit un nombre, soit une position de fin . L'exemple suivant restreint la recherche aux trois premiers items :

```
colors: [red green blue yellow blue orange gold]
probe find/part colors 'blue
[blue yellow blue orange gold]
```

La recherche suivante sur une chaîne est restreinte aux 15 premiers caractères:

```
text: "Keep things as simple as you can."
print find/part text "as" 15
as simple as you can.
```

L'exemple ci-dessous utilise un marquage de positions.
La recherche est réduite à une seule ligne de texte :

```
text: {
    This is line one.
    This is line two.
}
start: find text "this"
end: find start newline
item: find/part start "line" end
print item
line one.
```

8.4 Position finale

La fonction **find** retourne la position dans la série où un item a été trouvé.

Le raffinement **/tail** renverra la position qui suit immédiatement l'item trouvé.

Voici un exemple :

```
filename: %script.txt
print find filename "."
.txt
print find/tail filename "."
txt
clear change find/tail filename "." "r"
print filename
script.r
```

Dans cet exemple, **clear** est nécessaire pour enlever "xt" qui suit "t".

8.5 Recherche en arrière

Le dernier exemple de la section précédente ne marcherait pas si le nom du fichier possédait plus d'un point "." Par exemple :

```
filename: %new.script.txt
print find filename "."
.script.txt
```

Dans cet exemple, nous voulons la dernière occurence (la dernière correspondance trouvée) du point "." dans la chaîne.
Celle-ci peut être trouvée en utilisant le raffinement **/last**.

Le raffinement **/last** permet une rechercher arrière au travers d'une série.

```
print find/last filename "."
.txt
```

Le raffinement **/last** peut être combiné avec **/tail** pour produire :

```
print find/last/tail filename "."
txt
```

Si vous voulez continuer à rechercher en arrière dans la chaîne, vous aurez besoin du raffinement **/reverse**. Ce raffinement permet une recherche arrière à partir de la position courante jusqu'au début de la série, plutôt qu'une recherche depuis le début jusqu'à la fin.

```
where: find/last filename "."
print where
.txt
print find/reverse where "."
.script.txt
```

Notez que **/reverse** continue la recherche juste après la position de la dernière correspondance. Ceci évite de retrouver deux fois le même point "." encore .

8.6 Recherches multiples

Vous pouvez facilement réutiliser la fonction **find** pour chercher des occurrences multiples d'une valeur ou d'une chaîne.
Voici un exemple qui devrait afficher toutes les chaînes rencontrées dans un bloc :

```
blk: load %script.r
while [blk: find blk string!] [
    print first blk
    blk: next blk
]
```

L'exemple suivant compte le nombre de nouvelles lignes dans un script.

Il utilise juste le raffinement **/tail** pour éviter une boucle infinie et renvoie donc la position qui est immédiatement après la correspondance.

```
text: read %script.r
count: 0
while [text: find/tail text newline] [count: count + 1]
```

Pour effectuer une recherche répétée en arrière, utiliser le raffinement **/reverse**.

L'exemple suivant affiche toutes les index de positions dans l'ordre inverse pour le texte d'un script :

```
while [text: find/reverse tail text newline] [
    print index? text
]
```

8.7 Correspondances

Le raffinement **/match** modifie le comportement de **find** pour effectuer une rechercher de modéle à la position courante de la série. Ce raffinement permet aux opérations de parsing d'être effectuées en recherchant dans la suite de la série des correspondances avec le modéle fourni.
Voir le chapitre concernant le Parsing pour la recherche d'items.

Un simple exemple de recherche de correspondance est le suivant :

```
blk: [1432 "Franklin Pike Circle"]
probe find/match blk integer!
["Franklin Pike Circle"]
probe find/match blk 1432
["Franklin Pike Circle"]
probe find/match blk "test"
none
str: "Keep things simple."
probe find/match str "keep"
" things simple."
print find/match str "things"
none
```

Remarquez dans cet exemple qu'aucune recherche n'est réalisée. Soit le commencement de la série correspond, soit il ne correspond pas. Si il y a une correspondance, la variable de série est alors avancée à la position qui suit immédiatement l'item trouvé, permettant l'analyse de la séquence suivante.

Voici un exemple d'analyseur écrit avec **find/match** :

```
grammar: [
    ["keep" "make" "trust"]
    ["things" "life" "ideas"]
    ["simple" "smart" "happy"]
]
parse-it: func [str /local new] [
    foreach words grammar [
        foreach word words [
            if new: find/match str word [break]
        ]
       if none? new [return false]
       str: next new  ;skip space
   ]
   true
]
print parse-it "Keep things simple"
true
print parse-it "Make things smart"
true
print parse-it "Trust life well"
false
```

La recherche de modéle peut être rendue sensible à la casse avec le raffinement **/case** (distinction majuscules/minuscules). Les possibilités de **/match** peuvent être grandement étendues avec l'usage du raffinement **/any**.

8.8 Recherche avec des caractères jokers (wildcards)

Le raffinement **/any** permet d'utiliser des caractères jokers (wildcards) pour une recherche.

Le point d'interrogation (?) et l'astérisque (*) agissent comme des caractères de substitution pour remplacer respectivement "un caractère quelconque" et "un ensemble quelconque de plusieurs caractères".

Le raffinement **/any** peut être utilisé en conjonction avec **find** (avec ou sans le raffinement **/match**)

Exemples:

```
str: "abcdefg"
print find/any str "c*f"
cdefg
print find/any str "??d"
bcdefg
email-list: [
    mack@REBOL.dom
    judy@somesite.dom
    jack@REBOL.dom
    biff@REBOL.dom
    jenn@somesite.dom
]
foreach email email-list [
    if find/any email *@REBOL.dom [print email]
]
mack@REBOL.dom
jack@REBOL.dom
biff@REBOL.dom
```

L'exemple suivant utilise le raffinement **/match** pour tenter de trouver une correspondance à un modéle sur l'ensemble de la série:

```
file-list: [
    %REBOL.exe
    %notes.html
    %setup.html
    %feedback.r
    %nntp.r
    %rebdoc.r
    %REBOL.r
    %user.r
]
foreach file file-list [
    if find/match/any file %reb*.r [print file]
]
rebdoc.r
REBOL.r
none
```

Si l'un ou l'autre des caractères jokers standards (*) et (?) font partie de ce qui devrait être à trouver, des caractères de substitution différents peuvent être spécifiés avec le raffinement **/with**.

8.9 Fonction select

Une variante commode de la fonction **find** est la fonction **select**, qui retourne la valeur qui suit celle trouvée.

La fonction **select** est souvent utilisée pour consulter une valeur dans des blocs de données. La fonction **select** prend les mêmes types d'arguments que la fonction **find** : la série où chercher et la valeur à trouver.

Cependant, contrairement à **find**, qui renvoie une position dans la série, la fonction **select** retourne la *valeur* qui suit la correspondance.

```
colors: [red green blue yellow orange]
print select colors 'green
blue
```

La fonction **select** peut être utilisée pour accéder au contenu d'une petite base de données :

```
email-book: [
    "George" harrison@guru.org
    "Paul" lefty@bass.edu
    "Ringo" richard@starkey.dom
    "Robert" service@yukon.dom
]
```

Le code suivant détermine une adresse email spécifique :

```
print select email-book "Paul"
lefty@bass.edu
```

Il est possible d'employer la fonction **select** pour extraire un bloc d'expressions, qui peut être évalué ensuite. Par exemple, avec les données suivantes :

```
cases: [
    10 [print "ten"]
    20 [print "twenty"]
    30 [print "thirty"]
]
```

un bloc peut ainsi être évalué grâce à **select** :

```
do select cases 10
ten
```

```
do select cases 30
thirty
```

8.10 Recherche et remplacement

Pour remplacer des valeurs au sein d'une série, vous pouvez utiliser la fonction **replace**.
Cette fonction recherche une valeur particulière dans une série, puis la remplace par une autre.

La fonction **replace** prend trois arguments : la série où chercher, la valeur à remplacer, et la nouvelle valeur.

```
str: "hello world hello"
probe replace str "hello" "aloha"
"aloha world hello"
data: [1 2 8 4 5]
probe replace data 8 3
[1 2 3 4 5]
probe replace data 4 `four
[1 2 3 four 5]
probe replace data integer! 0
[0 2 3 four 5]
```

Utiliser le raffinement **/all** pour remplacer toutes les occurences trouvées depuis la position courante jusqu'à la fin de la série.

```
probe replace/all data integer! 0
[0 0 0 four 0]
code: [print "hello" print "world"]
replace/all code 'print 'probe
probe code
[probe "hello" probe "world"]
do code
helloworld
str: "hello world hello"
probe replace/all str "hello" "aloha"
"aloha world aloha"
```

[Retour au sommaire]

9. Trier une série

La fonction **sort** offre une méthode simple et rapide de trier des séries.
Elle est plus pratique pour des blocs de données, mais peut aussi être utilisée sur des chaînes de caractères.

9.1 Tri simple

Les exemples de tris les plus simples sont :

```
names: [Eve Luke Zaphod Adam Matt Betty]
```

```
probe sort names
[Adam Betty Eve Luke Matt Zaphod]
print sort [321.3 78 321 42 321.8 12 98]
12 42 78 98 321 321.3 321.8
print sort "plosabelm"
abellmops
```

Remarquez que **sort** a un effet **destructeur** sur la série fournie en argument. Elle modifie l'ordre des données d'origine. Pour éviter cela, utilisez **copy**, comme dans l'exemple suivant :

```
probe sort copy names
```

Par défaut, le tri est insensible à la casse :

```
print sort ["Fred" "fred" "FRED"]
Fred fred FRED
print sort "G4C28f9I15Ed3bA076h"
0123456789AbCdEfGhI
```

Mais avec le raffinement **/case**, le tri devient sensible aux majuscules/minuscules :

```
print sort/case "gCcAHfiEGeBIdbFaDh"
ABCDEFGHIabcdefghi
print sort/case ["Fred" "fred" "FRED"]
FRED Fred fred
print sort/case "g4Dc2BI8fCF9i15eAd3bGaE07H6h"
0123456789ABCDEFGHIabcdefghi
```

Beaucoup d'autres types de données peuvent être triées :

```
print sort [1.3.3.4 1.2.3.5 2.2.3.4 1.2.3.4]
1.2.3.4 1.2.3.5 1.3.3.4 2.2.3.4
print sort [$4.23 $23.45 $62.03 $23.23 $4.22]
$4.22 $4.23 $23.23 $23.45 $62.03
print sort [11:11:43 4:12:53 4:14:53 11:11:42]
4:12:53 4:14:53 11:11:42 11:11:43
print sort [11-11-1999 10-11-9999 11-4-1999 11-11-1998]
11-Nov-1998 11-Apr-1999 11-Nov-1999 10-Nov-9999
print sort [john@doe.dom jane@doe.dom jack@jill.dom]
jack@jill.dom jane@doe.dom john@doe.dom
print sort [%user.r %REBOL.r %history.r %notes.html]
history.r notes.html REBOL.r user.r
```

9.2 Tri par groupe

Souvent, il est nécessaire de trier un ensemble de données comme des enregistrements comprenant plus d'une valeur.

Le raffinement **/skip** permet de trier des enregistrements ayant une longueur fixe. Ce raffinement prend un argument supplémentaire : un nombre entier indiquant la longueur de chaque enregistrement.

Voici un exemple qui trie un bloc contenant des prénoms, noms, âges, et emails. Le bloc est tri selon la première colonne, le prénom.

```
names: [
    "Evie" "Jordan" 43 eve@jordan.dom
    "Matt" "Harrison" 87 matt@harrison.dom
    "Luke" "Skywader" 32 luke@skywader.dom
    "Beth" "Landwalker" 104 beth@landwalker.dom
    "Adam" "Beachcomber" 29 adam@bc.dom
]
sort/skip names 4
foreach [first-name last-name age email] names [
    print [first-name last-name age email]
]
Adam Beachcomber 29 adam@bc.dom
Beth Landwalker 104 beth@landwalker.dom
Evie Jordan 43 eve@jordan.dom
Luke Skywader 32 luke@skywader.dom
Matt Harrison 87 matt@harrison.dom
```

9.3 Fonctions de comparaison

Le raffinement **/compare** permet de réaliser des comparaisons spécifiques sur les données au cours du tri.

Ce raffinement nécessite un argument supplémentaire, qui est la fonction de comparaison à utiliser pour trier les données.

Une fonction de comparaison est écrite comme une fonction normale, mais prend deux arguments.

Ces arguments sont les valeurs à comparer. Une fonction de comparaison renvoie *true* si la première valeur doit être placée *avant* la seconde et *false* si elle doit être mise *après* .

Une comparaison classique place des données dans l'ordre croissant :

```
ascend: func [a b] [a < b]
```

Si la première valeur est inférieure à la deuxième, alors *true* est retourné par la fonction, et la première valeur est placée avant la deuxième valeur.

```
data: [100 101 -20 37 42 -4]
probe sort/compare data :ascend
[-20 -4 37 42 100 101]
```

Pareillement :

```
descend: func [a b] [a > b]
```

Si la première valeur est supérieure à la seconde , alors la valeur *true* est renvoyée et les données sont triées avec les plus grandes valeurs d'abord. Le tri s'effectue dans l'ordre décroissant.

```
probe sort/compare data :descend
[101 100 42 37 -4 -20]
```

Noter que dans chacun des cas la fonction de comparaison est passée avec son nom précédé de deux points. Le nom précédé de deux points force la fonction à être passée à **sort** sans être d'abord évaluée. La fonction de comparaison peut aussi être fournie directement :

```
probe sort/compare data func [a b] [a &gt; b]
[101 100 42 37 -4 -20]
```

[Retour au sommaire]

10. Série en tant qu'ensemble de données

Quelques fonctions travaillent sur les séries en tant qu'ensemble de données.

Ces fonctions permettent de réaliser des opérations comme trouver l'intersection ou l'union de deux séries.

10.1 Unique

La fonction **unique** renvoie un ensemble de valeurs sans doublons.

Exemples:

```
data: [Bill Betty Bob Benny Bart Bob Bill Bob]
probe unique data
[Bill Betty Bob Benny Bart]
print unique "abracadabra"
abrcd
```

10.2 Intersect

La fonction **intersect** prend deux séries en arguments et retourne une série contenant leurs valeurs communes .

Exemples:

```
probe intersect [Bill Bob Bart] [Bob Ted Fred]
[Bob]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
```

```
probe intersect lunch dinner
[ham carrot]
print intersect [1 3 2 4] [3 5 4 6]
3 4
string1: "CBAD"    ; A B C D scrambled
string2: "EDCF"    ; C D E F scrambled
print sort intersect string1 string2
CD
```

Intersect peut être utilisée entre "bitsets" :

```
all-chars: "ABCDEFGHI"
charset1: charset "ABCDEF"
charset2: charset "DEFGHI"
charset3: intersect charset1 charset2
print find charset3 "E"
true
print find charset3 "B"
false
```

Le raffinement **/case** permet d'extraire les valeurs communes aux deux séries, en tenant compte de la casse :

```
probe intersect/case [Bill bill Bob bob] [Bart bill Bob]
[bill Bob]
```

10.3 Union

La fonction **union** prend deux séries en arguments et renvoie une série réunissant les valeurs de chacune , mais sans doublons.

Exemples :

```
probe union [Bill Bob Bart] [Bob Ted Fred]
[Bill Bob Bart Ted Fred]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe union lunch dinner
[ham cheese bread carrot salad rice]
print union [1 3 2 4] [3 5 4 6]
1 3 2 4 5 6
string1: "CBDA"    ; A B C D scrambled
string2: "EDCF"    ; C D E F scrambled
print sort union string1 string2
ABCDEF
```

La fonction **union** peut aussi travailler avec des "bitsets" :

```
charset1: charset "ABCDEF"
```

```
charset2: charset "DEFGHI"
charset3: union charset1 charset2
print find charset3 "C"
true
print find charset3 "G"
true
```

Le raffinement ***/case*** donne à la fonction **union** une sensibilité à la casse, les majuscules et minuscules seront distinguées :

```
probe union/case [Bill bill Bob bob] [bill Bob]
[Bill bill Bob bob]
```

10.4 Difference

La fonction **difference** prend deux séries en arguments et renvoie une série qui contient toutes les valeurs qui ne sont pas communes aux deux.

Exemples:

```
probe difference [1 2 3 4] [1 2 3 5]
[4 5]
probe difference [Bill Bob Bart] [Bob Ted Fred]
[Bill Bart Ted Fred]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe difference lunch dinner
[cheese bread salad rice]
string1: "CBAD"    ; A B C D scrambled
string2: "EDCF"    ; C D E F scrambled
print sort difference string1 string2
ABEF
```

Là encore, le raffinement ***/case*** permet d'utiliser la fonction **difference** avec une sensibilité aux majuscules/minuscules.

```
probe difference/case [Bill bill Bob bob] [Bart bart bill Bob]
[Bill bob Bart bart]
```

10.5 Exclude

> NDT : ce paragraphe est le regroupement de deux parties du document original sur la fonction exclude.

Une variante de la fonction **difference** est la fonction **exclude**. La fonction **exclude** prend deux séries en arguments et renvoie une série qui va contenir toutes les valeurs de la *première* série,

moins celles, communes aux deux, de la deuxième série.

Exemples :

```
probe exclude [1 2 3 4] [1 2 3 5]
[4]
```

(Notez que le résultat ci-dessus ne contient pas 5 comme c'était le cas avec la fonction **difference** vue précedemment.)

```
probe exclude [Bill Bob Bart] [Bob Ted Fred]
[Bill Bart]
probe exclude "abcde" "ace"
"bd"
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe exclude lunch dinner
[cheese bread]
string1: "CBAD"    ; A B C D scrambled
string2: "EDCF"    ; C D E F scrambled
print sort difference string1 string2
AB
```

Le raffinement **/case** permet une exclusion sensible à la casse :

```
probe exclude/case [Bill bill Bob bob] [Bart bart bill Bob]
[Bill bob]
```

[Retour au sommaire]

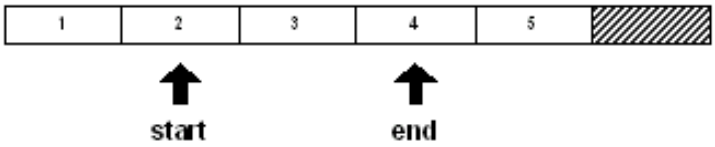
11. Multiples variables de série

Plusieurs variables de série peuvent référencer la même série.

Par exemple :

```
data: [1 2 3 4 5]
start: find data 3
end: find start 4
print first start
2
print first end
4
```

Les variables "start" et "end" font réference à la même série. Elles pointent différentes positions, mais la série qu'elles référencent est la même.

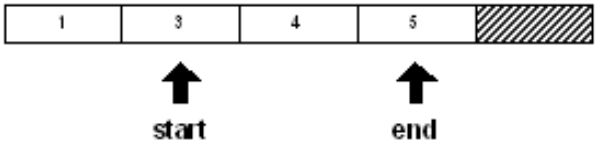


Si on utilise les fonctions **insert** ou **remove** sur une série, les valeurs dans la série sont décalées, et les variables "start" et "end" peuvent ne plus se rapporter aux mêmes valeurs.

Par exemple, si une valeur est enlevée de la série à la position de "start" :

```
remove start
print first start
3
print first end
5
```

La série a été décalée vers la gauche et les variables se rapportent à présent à différentes valeurs



Notez que les positions d'index des variables n'ont pas changées, mais ce sont les valeurs de la série qui ont changées.
La même situation peut se produire en utilisant la fonction **insert**.

Parfois cet effet secondaire fonctionnera à votre avantage. Parfois non, et cela vous obligera à modifier votre code.

[Retour au sommaire]

12. Raffinements de modification

Les fonctions **change**, **insert**, et **remove** peuvent prendre des raffinements supplémentaires pour modifier leur comportement.

12.1 Part

Le raffinement **/part** accepte un nombre ou une position de la série et l'utilise pour limiter l'effet de la fonction.

Par exemple, avec la série suivante :

```
str: "abcdef"
blk: [1 2 3 4 5 6]
```

vous pouvez changer une partie de "str" et "blk" en utilisant **change/part** :

```
change/part str [1 2 3 4] 3
```

```
probe str
1234def
change/part blk "abcd" 3
probe blk
["abcd" 4 5 6]
```

Vous pouvez insérer une partie d'une série à la fin de "str" et de "blk" en utilisant **insert/part**.

```
insert/part tail str "-ghijkl" 4
probe str
1234def-ghi
insert/part tail blk ["--" 7 8 9 10 11 12] 4
probe blk
["abcd" 4 5 6 "--" 7 8 9]
```

Pour ôter un morceau des séries "str" et "blk", utiliser **remove/part**. Noter comment **find** est utilisé pour obtenir la position de la série :

```
remove/part (find str "d") (find str "-")
probe str
1234-ghi
remove/part (find blk 4) (find blk "--")
probe blk
["abcd" "--" 7 8 9]
```

12.2 Only

Le raffinement **/only** modifie ou insére un bloc tel quel plutôt que ses valeurs propres.

Exemples:

```
blk: [1 2 3 4 5 6]
```

Vous pouvez remplacer la valeur 2 dans le bloc "blk" avec le bloc [a b c] et insérer le bloc [$1 $2 $3] à la position du 5.

```
change/only (find blk 2) [a b c]
probe blk
[1 [a b c] 3 4 5 6]
insert/only (find blk 5) [$1 $2 $3]
probe blk
[1 [a b c] 3 4 [$1.00 $2.00 $3.00] 5 6]
```

12.3 Dup

Le raffinement **/dup** modifie ou insére une valeur un certain nombre de fois

Exemples:

```
str: "abcdefghi"
blk: [1 2 3 4 5 6]
```

Vous pouvez changer les quatre premières valeurs dans la série "str" ou "blk" pour une astérisque (*) avec :

```
change/dup str "*" 4
probe str
****efghi
change/dup blk "*" 4
probe blk
["*" "*" "*" "*" 5 6]
```

Pour insérer un tiret (-) quatre fois avant la dernière valeur dans la chaîne ou le bloc :

```
insert/dup (back tail str) #"-" 4
probe str
****efgh----i
insert/dup (back tail blk) #"-" 4
probe blk
["*" "*" "*" "*" 5 #"-" #"-" #"-" #"-" 6]
```

Chapitre 7 - Les Séries de blocs

Ce document est la traduction française du Chapitre 7 du User Guide de REBOL/Core, qui concerne
les séries sous forme de blocs.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
22 mai 2005 7:42	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Blocs de Blocs

Quand un bloc apparaît en tant que valeur au sein d'un autre bloc, il est compté comme un **seul** élément et cela, quelque soit le nombre de valeurs qu'il contient.

Par exemple :

```
values: [
    "new" [1 2]
    %file1.txt ["one" ["two" %file2.txt]]
]
probe values
["new" [1 2] %file1.txt ["one" ["two" %file2.txt]]]
```

La longueur de *values* est quatre éléments. La seconde et la quatrième valeur sont comptés comme des éléments unitaires :

```
print length? values
4
```

Les blocs à l'intérieur d'autres blocs ne perdent pas leur caractéristique de bloc. Dans l'exemple ci-dessous, la fonction **second** est utilisée pour extraire la deuxième valeur du bloc *values*.
Pour afficher le bloc, saisissez :

```
probe second values
[1 2]
```

Pour connaître la longueur de ce bloc, tapez :

```
print length? second values
2
```

Pour afficher le type de données (datatype) :

```
print type? second values
block
```

De la même manière, les fonctions sur les séries peuvent être exploitées sur d'autres types de valeurs dans les blocs. Dans l'exemple suivant, **pick** est utilisé pour extraire *%file1.txt* du bloc *values*.

Pour récupérer la valeur du troisième élément, saisissez :

```
probe pick values 3
%file1.txt
```

Pour récupérer la longueur de la valeur :

```
print length? pick values 3
9
```

Pour voir le type de données associé à la valeur extraite :

```
print type? pick values 3
file
```

[Retour au sommaire]

3. Paths, chemins pour les blocs imbriqués

La notation avec les paths est très pratique pour les blocs imbriqués.

La quatrième valeur de la série *values* est un bloc contenant d'autres blocs. L'exemple suivant utilise un path pour récupérer des informations dans ce bloc.

Pour voir les valeurs de ce bloc, tapez :

```
probe values/4
["one" ["two" %file2.txt]]
probe values/4/2
["two" %file2.txt]
```

Pour obtenir les longueurs :

```
print length? values/4
2
print length? values/4/2
2
```

Et pour voir le type de données, saisissez :

```
print type? values/4
block
print type? values/4/2
block
```

Les deux séries contenues dans cette quatrième valeur sont aisément accessibles. Pour voir ces valeurs, tapez :

```
probe values/4/2/1
two
probe values/4/2/2
%file2.txt
```

Pour obtenir les longueurs de ces valeurs :

```
print length? values/4/2/1
3
print length? values/4/2/2
9
```

et pour leurs datatypes (types de données) :

```
print type? values/4/2/1
string
print type? values/4/2/2
file
```

Pour modifier ces valeurs :

```
change (next values/4/2/1) "o"
probe values/4/2/1
too
change/part (next find values/4/2/2 ".") "r" 3
probe values/4/2/2
%file2.r
```

Les exemples précédents illustrent la capacité que possède REBOL à manipuler des valeurs imbriquées dans des blocs. Notez que, dans les derniers exemples, la fonction **change** est utilisée pour modifier une chaîne et un nom de fichier avec trois niveaux d'imbrication.

L'affichage du bloc *values* produit le résultat suivant :

```
probe values
["new" [1 2] %file1.txt ["one" ["too" %file2.r]]]
```

[Retour au sommaire]

4. Tableaux

Les blocs sont utilisés pour créer des tableaux. Un exemple de tableau bi-dimensionnel statique est :

```
arr: [
    [1   2   3  ]
    [a   b   c  ]
    [$10 $20 $30]
]
```

Vous pouvez obtenir les valeurs d'un tableau avec les fonctions d'extraction relatives aux séries :

```
probe first arr
[1 2 3]
probe pick arr 3
[$10.00 $20.00 $30.00]
probe first first arr
1
```

Vous pouvez aussi utiliser des paths pour obtenir les valeurs d'un tableau :

```
probe arr/1
[1 2 3]
probe arr/3
[$10.00 $20.00 $30.00]
probe arr/3/2
$20.00
```

Les paths peuvent encore être utilisés pour changer les valeurs dans un tableau :

```
arr/1/2: 20

probe arr/1
[1 20 3]
arr/3/2: arr/3/1 + arr/3/3

probe arr/3/2
$40.00
```

4.1 Création de tableaux

La fonction **array** crée dynamiquement un tableau.

Cette fonction prend en argument soit un nombre entier, soit un bloc de nombres entiers, et elle retourne en résultat un bloc : le tableau. Par défaut, les cellules d'un tableau sont initialisées à **none**.

Pour initialiser les cellules d'un tableau avec d'autres valeurs, utilisez le raffinement **/initial**, qui est expliqué dans la section suivante.

Lorsqu'un tableau est fourni avec **un seul nombre entier**, c'est un tableau à **une dimension**, de la taille du nombre, qui est retourné.

```
arr: array 5
probe arr
[none none none none none]
```

Quand un bloc de plusieurs nombres entiers est passé en argument, le tableau est à plusieurs dimensions.
Chaque nombre entier donne respectivement la taille de la dimension correspondante.

Voici un exemple d'un tableau possédant six cellules, sur deux lignes et trois colonnes :

```
arr: array [2 3]
probe arr
[[none none none] [none none none]]
```

Il est possible de faire un tableau à trois dimensions en rajoutant un autre nombre entier au bloc en argument :

```
arr: array [2 3 2]
foreach lst arr [probe lst]
[[none none] [none none] [none none]]
[[none none] [none none] [none none]]
```

Le bloc d'entiers qui est passé à la fonction **array** peut être très grand selon ce que la mémoire de votre système supporte.

4.2 Valeurs initiales

Pour initialiser les cellules d'un tableau à une valeur autre que **none**, utilisez le raffinement **/initial**.

Voici quelques exemples :

```
arr: array/initial 5 0
probe arr
[0 0 0 0 0]
arr: array/initial [2 3] 0
probe arr
[[0 0 0] [0 0 0]]
arr: array/initial 3 "a"
probe arr
["a" "a" "a"]
arr: array/initial [3 2] 'word
probe arr
[[word word] [word word] [word word]]
arr: array/initial [3 2 1] 11:11
probe arr
[[[11:11] [11:11]] [[11:11] [11:11]] [[11:11] [11:11]]]
```

[Retour au sommaire]

5. Composition de blocs

La fonction **compose** est pratique pour créer des blocs avec des valeurs dynamiques. Elle peut être utilisée pour créer aussi bien des données et du code.

La fonction **compose** attend un bloc en argument et renvoie en résultat un bloc composé de chacune des valeurs du bloc en argument.

Les valeurs entre parenthèses sont évaluées en priorité, avant que le bloc ne soit retourné.
Par exemple :

```
probe compose [1 2 (3 + 4)]
[1 2 7]
probe compose ["The time is" (now/time)]
["The time is" 10:32:45]
```

Si des parenthèses encadrent un bloc, alors chaque valeur de ce bloc sera utilisée :

```
probe compose [a b ([c d])]
[a b c d]
```

Pour éviter cela dans le résultat, vous devez inclure ce bloc dans un autre bloc :

(**NdT** : c'est-à-dire protéger le bloc par un autre bloc).

```
probe compose [a b ([[c d]])]
[a b [c d]]
```

Un bloc sans éléments est sans effet :

```
probe compose [a b ([]) c d]
[a b c d]
```

Lorsque **compose** s'applique sur un bloc comprenant des sous-blocs, les sous-blocs ne sont pas évalués, même s'ils contiennent des parenthèses :

```
probe compose [a b [c (d e)]]
[a b [c (d e)]]
```

Si vous souhaitez que les sous-blocs soient évalués, utilisez le raffinement **/deep**. Le raffinement **/deep** entraîne l'évaluation de toutes les valeurs entre parenthèses, indépendamment de la position où elles se trouvent :

```
probe compose/deep [a b [c (d e)]]
[a b [c d e]]
```

Chapitre 8 - Les Séries : chaînes

Ce document est la traduction française du Chapitre 8 du User Guide de REBOL/Core, qui concerne
les séries sous forme de chaînes de caractères.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
26 mai 2005 22:17	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Fonctions relatives aux chaînes de caractères

De nombreuses fonctions permettent de manipuler ou de créer des chaînes de caractères. Ces fonctions peuvent transformer des chaînes, y effectuer des recherches, les compresser ou les décompresser, modifier leur espacement, les analyser, et les convertir. Ces fonctions agissent sur tous les types de données liés aux chaînes de caractères, comme **string!**, **binary!**, **tag!**, **file!**,

URL!, **email!**, et **issue!**.

Les fonctions de création, de modification et de recherche ont été présentées dans le chapitre sur les Séries. Cette présentation incluait les fonctions de chaînes :

copy	permet la copie d'une partie ou de toute une chaîne
make	alloue un espace mémoire pour la chaîne, crée un type
insert	insère un caractère ou une sous-chaîne dans une autre
remove	ôte un ou plusieurs caractères d'une chaîne
change	change un ou plusieurs caractères dans une chaîne
append	insère un caractère ou une sous-chaîne à la fin d'une chaîne
find	trouve ou effectue une correspondance entre chaînes
replace	trouve une chaîne et la remplace par une autre

De surcroît, les fonctions permettant de parcourir les séries, comme **next**, **back**, **head**, et **tail** ont déjà été présentées. Elles sont utilisées pour se déplacer dans les chaînes. Également, les fonctions de test des séries vous permettent de déterminer votre position dans une chaîne (NdT : comme **index?**, **tail?**, **head?**).

Ce chapitre va présenter d'autres fonctions qui permettent la conversion de valeurs REBOL en chaînes de caractères. Ces fonctions sont fréquemment utilisées, comme avec les fonctions **print** et **probe**.

Elles comprennent :

form	convertit des valeurs avec des espaces en un format humainement lisible
mold	convertit des valeurs en un format REBOL
join	concatène des valeurs
reform	réduit (évalue) des valeurs avant de les traduire avec **form**
remold	réduit (évalue) des valeurs avant de les traduire avec **mold**
rejoin	réduit (évalue) des valeurs avant de les concaténer

Ce chapitre décrira aussi les fonctions de chaîne de caractères :

detab	remplace les tabulations par des espaces
entab	remplace les espaces par des tabulations
trim	enlève les espaces blancs ou les lignes vierges autour de chaînes de caractère
uppercase	change en majuscules
lowercase	change en minuscules
checksum	calcule la valeur de checksum pour une chaîne
compress	compresse la chaîne
decompress	décompresse la chaîne
enbase	encode une chaîne sur une autre base numérique
debase	convertit une chaîne encodée

dehex convertit des valeurs hexadécimales ASCII en caractères (ex. dans les URLs).

[Retour au sommaire]

3. Conversion de valeurs en chaînes de caractères

3.1 Join

La fonction **join** prend deux arguments et les concatènent en une seule chaîne.

Le type de données de la série retournée est basé sur celui du premier argument.
Quand le premier argument est une valeur de type "série", le même type est retourné :

```
str: "abc"
file: %file
url: http://www.rebol.com/

probe join str [1 2 3]
abc123
probe join file ".txt"
%file.txt
probe join url %index.html
http://www.rebol.com/index.html
```

Quand le premier argument n'est pas une série, la fonction **join** le convertit en chaîne de caractère, puis effectue la concaténation :

```
print join $11 " dollars"
$11.00 dollars
print join 9:11:01 " elapsed"
9:11:01 elapsed
print join now/date " -- today"
30-Jun-2000 -- today
print join 255.255.255.0 " netmask"
255.255.255.0 netmask
print join 412.452 " light-years away"
412.452 light-years away
```

Quand le deuxième argument ajouté est un bloc, les valeurs de ce bloc sont évaluées et ajoutées au résultat :

```
print join "a" ["b" "c" 1 2]
abc12
print join %/ [%dir1/ %sub-dir/ %filename ".txt"]
%/dir1/sub-dir/filename.txt
print join 11:09:11 ["AM" " on " now/date]
11:09:11AM on 30-Jun-2000
print join 312.423 [123 987 234]
312.423123987234
```

3.2 Rejoin

La fonction **rejoin** est identique à **join**, mis à part le fait qu'elle prend un argument de type bloc, qui est évalué :

```
print rejoin ["try" 1 2 3]
try123
print rejoin ["h" 'e #"l" (to-char 108) "o"]
hello
```

3.3 Form

La fonction **form** transforme une valeur en chaîne de caractères :

```
print form $1.50
$1.50
print type? $1.50
money
print type? form $1.50
string
```

L'exemple suivant utilise **form** pour trouver un nombre par sa valeur décimale :

```
blk: [11.22 44.11 11.33 11.11]
foreach num blk [if find form num ".11" [print num]]
44.11
11.11
```

Lorsque **form** est utilisée sur un bloc, toutes les valeurs du bloc sont converties en chaînes de caractères, avec des espaces entre chacune d'elles :

```
print form [11.22 44.11 11.33]
11.22 44.11 11.33
```

La fonction **form** n'évalue pas les valeurs d'un bloc. Elle transforme des mots en chaînes de caractères :

```
print form [a block of undefined words]
a block of undefined words
print form [33.44 num "-- unevaluated string:" str]
33.44 num -- unevaluated string: str
```

3.4 Reform

La fonction **reform** est identique à **form**, excepté que les blocs sont réduits (évalués) avant d'être

convertis en chaînes.

```
str1: "Today's date is:"
str2: "The time is now:"
print reform [str1 now/date newline str2 now/time]
Today's date is: 30-Jun-2000 The time is now: 14:41:44
```

La fonction d'affichage **print** est basée sur la fonction **reform**.

3.5 Mold

La fonction **mold** convertit une valeur en chaîne de caractère utilisable par REBOL. Les chaînes créées avec **mold** peuvent être retransformées en valeurs REBOL avec la fonction **load**.

```
blk: [[11 * 4] ($15 - $3.89) "eleven dollars"]
probe blk
[[11 * 4] ($15.00 - $3.89) "eleven dollars"]
molded-blk: mold blk
probe molded-blk
{[[11 * 4] ($15.00 - $3.89) "eleven dollars"]}
print type? blk
block
print type? molded-blk
string
probe first blk
[11 * 4]
probe first molded-blk
#"["
```

Les chaînes renvoyées par **mold** peuvent être récupérées par REBOL :

```
new-blk: load molded-blk
probe new-blk
[[11 * 4] ($15.00 - $3.89) "eleven dollars"]
print type? new-blk
block
probe first new-blk
[11 * 4]
```

La fonction **mold** n'évalue pas les valeurs d'un bloc :

```
money: $11.11
sub-blk: [inside another block mold this is unevaluated]
probe mold [$22.22 money "-- unevaluated block:" sub-blk]
{[$22.22 money "-- unevaluated block:" sub-blk]}
probe mold [a block of undefined words]
[a block of undefined words]
```

3.6 Remold

La fonction **remold** s'utilise comme **mold**, à l'exception du fait que les blocs sont réduits (évalués) avant d'être convertis.

```
str1: "Today's date is:"
probe remold [str1 now/date]
{["Today's date is:" 30-Jun-2000]}
```

3.7 Fonctions pour gérer les espaces

3.7.1 Trim

La fonction **trim** enlève tous les espaces multiples d'une chaîne. Par défaut, **trim** enlève les espaces en excès au début et à la fin d'une chaîne :

```
str: "  line of text with spaces around it "
print trim str
line of text with spaces around it
```

Remarquez que la chaîne est modifiée dans le processus :

```
print str
line of text with spaces around it
```

Pour exécuter **trim** sur une copie d'une chaîne, écrivez :

```
print trim copy str
line of text with spaces around it
```

La fonction **trim** inclut quelques raffinement afin d'indiquer où les espaces doivent être supprimés dans la chaîne :

/head	enlève les espaces en début de chaîne
/tail	enlève les espaces en fin de chaîne
/auto	enlève les espaces à chaque ligne, relativement à la première ligne
/lines	enlève les sauts de lignes, et les remplace par des espaces
/all	enlève tous les espaces, les tabulations,et les sauts de lignes
/with	enlève tous les caractères spécifiés

Utilisez les raffinements **/head** et **/tail** pour ôter les espaces en début et en fin de chaîne :

```
probe trim/head copy str
line of text with spaces around it
probe trim/tail copy str
```

```
line of text with spaces around it
```

Utilisez le raffinement **/auto** pour effacer les espaces résiduels sur des lignes multiples, tout en conservant intacte l'indentation :

```
str: {
    indent text
        indent text
            indent text
        indent text
    indent text
}
print str
indent text
    indent text
        indent text
    indent text
indent text
probe trim/auto copy str
{indent text
    indent text
        indent text
    indent text
indent text
}
```

Le raffinement **/lines** permet d'enlever les espaces en début et en fin de chaîne, mais également de convertir les sauts de lignes en espaces :

```
probe trim/lines copy str
{indent text indent text indent text indent text indent text}
```

L'usage du raffinement **/all** enlève tous les espaces, les tabulations, les sauts de lignes :

```
probe trim/all copy str
indenttextindenttextindenttextindenttextindenttext
```

La raffinement **/with** permet d'éliminer de la chaîne tous les caractères qui ont été spécifiés avec **/with**. Dans l'exemple suivant, les espaces, les sauts de lignes, et les caractères "e" et "t" sont supprimés :

```
probe trim/with copy str " ^/et"
indnxindnxindnxindnxindnx
```

3.7.2 Detab et Entab

Les fonctions **detab** et **entab** transforment les tabulations en espaces et inversement, les espaces en tabulations.

```
str:
{^(tab)line one
^(tab)^(tab)line two
^(tab)^(tab)^(tab)line three
^(tab)line^(tab)full^(tab)of^(tab)tabs}
print str
line one
        line two
            line three
    line    full    of  tabs
```

Par défaut, la fonction **detab** convertit chaque tabulation en une série de **quatre** espaces (le standard pour le style REBOL). Toutes les tabulations dans la chaîne seront transformées en espaces, quelque soient leurs positions.

```
probe detab str
{    line one
        line two
            line three
    line    full    of  tabs}
```

Remarquez que les fonctions **detab** et **entab** modifient la chaîne qu'elles prennent en argument. Pour travailler sur une copie de la chaîne source, utilisez la fonction **copy**.

La fonction **entab** convertit les espaces en tabulations. Chaque série de quatre espaces sera transformée en une tabulation. Seuls les espaces en début de ligne seront transformés en tabulations.

```
probe entab str
{^-line one
^-^-line two
^-^-^-line three
^-line^-full^-of^-tabs}
```

Vous pouvez utiliser le raffinement **/size** pour spécifier la taille des tabulations. Par exemple, si vous voulez convertir chaque tabulation en série de huit espaces, ou transformer ces huit espaces en une tabulation, vous pouvez utiliser cet exemple :

```
probe detab/size str 8
{        line one
                line two
                        line three
        line    full    of      tabs}
probe entab/size str 8
{^-line one
^-^-line two
^-^-^-line three
^-line^-full^-of^-tabs}
```

3.8 Uppercase et Lowercase

Deux fonctions permettent le changement de la casse des caractères : **uppercase** et **lowercase**.

La fonction **uppercase** prend une chaîne en argument et la met en majuscule.

```
print uppercase "SamPle TExT, tO test CASES"
SAMPLE TEXT, TO TEST CASES
```

La fonction **lowercase** effectue le travail inverse, elle met les caractères en minuscule :

```
print lowercase "Sample TEXT, tO teST Cases"
sample text, to test cases
```

Pour transformer juste une partie de la chaîne, utilisez le raffinement **/part** :

```
print upppercase/part "ukiah" 1
Ukiah
```

3.9 Checksum

La fonction **checksum** renvoie la valeur de checksum (somme de contrôle) d'une chaîne. Plusieurs types de checksum peuvent être calculés

CRC24	contrôle de redondance cyclique (défaut)
TCP	checksum Internet TCP 16-bit.
Secure	Retourne un checksum sécurisé par cryptage

NdT :

Les éléments qui suivent correspondent à l'aide en ligne sur la fonction **checksum**, et non à la documentation officielle du User Guide, qui semble caduque sur ce point.

Checksum permet l'usage des raffinements :

/tcp	renvoie la valeur de checksum Internet 16 bits
/secure	renvoie la valeur de checksum sécurisé par cryptage
/hash	retourne une valeur d'index pour une table de hachage.
/method	utilise une méthode de cryptage, qui peut être SHA1 ou MD5

/key	utilise une clé pour une authentification HMAC (Keyed-Hashing for Message Authentication Code).

> **NdT** :
>
> Voir l'aide en ligne pour plus de détails.

Par défaut, c'est la valeur de checksum CRC qui est calculée :

```
print checksum "hello"
52719
print checksum (read http://www.rebol.com/)
356358
```

Pour calculer la checksum TCP, utilisez le raffinement **/tcp** :

```
print checksum/tcp "hello"
10943
```

Le raffinement **/secure** retournera une valeur binaire, pas un entier. Utilisez le raffinement **/secure**, pour calculer ce type de checksum :

```
print checksum/secure "hello"
#{AAF4C61DDCC5E8A2DABEDE0F3B482CD9AEA9434D}
```

Le raffinement **/method** permet d'utiliser (par exemple pour contrôler des mots de passe) une méthode de cryptage MD5 ou SHA1 :

```
print checksum/method "password2005" 'md5
#{9FDC0F6F1A5A0443F1C6E2393BE936DE}
print checksum/method "password2005" 'sha1
#{3E352A5705741521337C01537AEA0A54DD13B993}
```

3.10 Compression et décompression

La fonction **compress** va compresser une chaîne et retourner un type de donnée binaire. Dans l'exemple suivant, un petit fichier est compressé :

```
Str:
{I wanted the gold, and I sought it,
  I scrabbled and mucked like a slave.
Was it famine or scurvy -- I fought it;
```

```
  I hurled my youth into a grave.
I wanted the gold, and I got it --
  Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
  And somehow the gold isn't all.}

print [size? str "bytes"]
306 bytes
bin: compress str

print [size? bin "bytes"]
156 bytes
```

Remarquez que le résultat de la compression est du type de données binaire (**binary!**).

La fonction **decompress** décompresse une chaîne qui a été préalablement compressée.

```
print decompress bin
I wanted the gold, and I sought it,
  I scrabbled and mucked like a slave.
Was it famine or scurvy -- I fought it;
  I hurled my youth into a grave.
I wanted the gold, and I got it --
  Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
  And somehow the gold isn't all.
```

Sauvegarde de vos données :

Conservez toujours une copie non compressée de vos données compressées. Si vous perdez un seul octet d'une chaîne binaire compressée, il sera difficile de récupérer les données. Ne sauvegardez pas des fichiers d'archives en format compressé à moins que vous n'ayiez des copies d'origine non compressées.

3.11 Modification de la base numérique

Pour être envoyé sous forme de texte, les chaînes binaires doivent être encodées en hexadécimal ou en base 64. Ceci est fréquemment réalisé pour le courrier électronique, ou le contenu de groupes de nouvelles (newsgroups).

La fonction **enbase** encodera une chaîne binaire :

```
line: "No! There's a land!"
print enbase line
Tm8hIFRoZXJlJ3MgYSBsYW5kIQ==
```

Les chaînes encodées peuvent être décodées avec la fonction **debase**. Notez que le résultat est

une valeur de type binaire. Pour convertir cette valeur en une chaîne de caractères (**string!**), utilisez la fonction **to-string**.

```
b-line: debase e-line
print type? b-line
binary
probe b-line
#{4E6F21205468657265277320612061206C616E6421}
print to-string b-line
No! There's a land!
```

Le raffinement **/base** peut être utilisé avec les fonctions **enbase** et **debase**, pour spécifier un encode en base-2 (binaire), base-16 (hexadécimal) ou base-64.

Voici quelques exemples utilisant l'encodage en base 2 :

```
e2-str: enbase/base str 2
print e2-str
01100001
b2-str: debase/base e2-str 2
print type? b2-str
binary
probe b2-str
#{61}
print to-string b2-str
a
```

Quelques exemples avec un encodage en hexadécimal (base-16) :

```
e16-line: enbase/base line 16
print e16-line
4E6F21205468657265277320612061206C616E6421
b16-line: debase/base e16-line 16
print type? b16-line
binary
probe b16-line
#{4E6F21205468657265277320612061206C616E6421}
print to-string b16-line
No! There's a land!
```

3.12 Décodage hexadécimal pour Internet

La fonction **dehex** convertit les caractères encodés en hexadécimal des URLs Internet ou CGI en chaînes de caractères. La représentation hexadécimal ASCII se présente dans un URL ou une chaîne CGI comme %xx, où xx est une valeur hexadécimale.

```
str: "there%20seem%20to%20be%20no%20spaces"
print dehex str
there seem to be no spaces
print dehex "%68%65%6C%6C%6F"
```

```
hello
```

Chapitre 9 - Les Fonctions

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

Contenu

[Retour au sommaire]

1. 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

[Retour au sommaire]

2. 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).

[Retour au sommaire]

3. 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.

3.1 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.

3.2 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 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
```

3.3 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	copie juste une partie d'une chaîne
find/tail	renvoie la fin de la correspondance
load/markup	retourne des balises XML/HTML et des chaînes

Les fonctions peuvent aussi inclure plusieurs raffinements :

find/case/tail	cherche une correspondance en respectant la casse des caractères et renvoie la dernière
insert/only/dup	duplique plusieurs fois un bloc entier

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.

3.4 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
```

[Retour au sommaire]

4. 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!**

4.1 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.

4.2 has

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

Une autre façon d'écrire des fonctions simples sans arguments est d'utiliser **has**. br /> Cette fonction prend par contre uniquement 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              <-- demande d'un nom de ville
Où habitez-vous ? Bruxelles
Bruxelles  est une jolie ville
```

4.3 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
]
```

4.4 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)
```

4.5 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 *increment* ++ en C) :

```
++: func ['word] [set word 1 + get word]
count: 0
++ count
print count
1
print ++ count
2
```

4.6 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
]
```

4.7 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**.

4.8 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       <---- ici
][
    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**.

4.9 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.

4.10 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]
]
```

4.11 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
```

[Retour au sommaire]

5. 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**.

[Retour au sommaire]

6. 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]
```

[Retour au sommaire]

7. 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]]
```

[Retour au sommaire]

8. 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.

8.1 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 < 0 [
        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.

8.2 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 < time] block
]
```

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 < time] block
]
```

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.

[Retour au sommaire]

9. 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
]
```

[Retour au sommaire]

10. 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       <-- ici on est dans b-func
a: 11       <-- ici aussi
21          <-- ici encore
```

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
```

[Retour au sommaire]

11. 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"
]
```

[Retour au sommaire]

12. 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]
< [value1 value2]
<= [value1 value2]
<> [value1 value2]
= [value1 value2]
== [value1 value2]
=? [value1 value2]
> [value1 value2]
>= [value1 value2]
? ['word]
?? ['name]
about []
abs [value]
absolute [value]
...
```

[Retour au sommaire]

13. 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!]
]
```

Chapitre 10 - Les Objets

Ce document est la traduction française du Chapitre 10 du User Guide de REBOL/Core, qui concerne les Objets.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
4 avril 2005 17:52	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Présentation

Les objets permettent de regrouper un ensemble de variables et leurs valeurs dans un contexte commun.

Un objet peut inclure des valeurs scalaires, des séries, des fonctions ou d'autres objets.

Les objets sont pratiques pour travailler avec des structures complexes car ils permettent d'encapsuler des variables et le code qui leur est associé, et de les passer simplement à des fonctions.

[Retour au sommaire]

3. Création d'objets

De nouveaux objets sont construits avec la fonction **make**.

La fonction **make** requiert deux arguments et retourne un nouvel objet. Le format de la fonction **make** est le suivant :

```
new-object: make parent-object new-values
```

Le premier argument, *parent-object*, est l'objet "parent" à partir duquel le nouvel objet sera construit. Si aucun objet parent n'est valable, par exemple losqu'on définit pour la première fois un objet , on utilise par défaut le type **object!** :

```
new-object: make object! new-values
```

Le second argument, *new-values*, est un bloc contenant les définitions des variables et leurs valeurs initiales pour ce nouvel objet.

Chaque variable définie au sein de ce bloc est une variable de l'instance de l'objet.

Par exemple, si le bloc contenait deux définitions de variables :

```
mon-exemple: make object! [
    var1: 10
    var2: 20
]
```

L'objet *mon-exemple* posséde ici deux variables de type **integer!** (valeurs entières). Le bloc définissant les variables est évalué, il peut inclure n'importe quelle expression pour calculer les valeurs des variables :

```
mon-exemple: make object! [
    var1: 10
    var2: var1 + 10
    var3: now/time
]
```

Une fois que l'objet a été créé, il peut servir de prototype pour créer d'autre objets:

```
mon-exemple2: make mon-exemple []
```

L'exemple ci-dessus crée une seconde instance de l'objet *mon-exemple*. De nouvelles valeurs peuvent être mises dans le bloc :

```
mon-exemple2: make mon-exemple [
    var1: 30
    var2: var1 + 10
]
```

Ci-dessus, l'objet *mon-exemple2* posséde des valeurs différentes de l'objet *mon-exemple* original.

L'objet *mon-exemple2* peut aussi étendre la définition de l'objet prototype *mon-exemple* en lui ajoutant de nouvelles variables :

```
mon-exemple2: make mon-exemple [
    var4: now/date
    var5: "example"
]
```

Le résultat est un objet qui a cinq variables : trois proviennent de l'objet original, deux sont nouvelles.

Le processus d'extension de la définition d'un objet peut être ainsi répété de nombreuses fois.

Il est aussi possible de créer un objet qui contient des variables initialisées à une valeur quelconque. Par exemple, en utilisant une initialisation en cascade :

```
mon-example3: make object! [
    var1: var2: var3: var4: none
]
```

Dans le code précédent, les quatres variables *var1* à *var4* sont mises à "**none**" au sein de l'objet.

Pour résumer, le processus de création d'un objet passe par les étapes suivantes :

1. Usage de la fonction **make** pour créer un nouvel objet basé sur un prototype (un objet parent) ou sur le type **object!**.
2. Ajout au nouvel objet des variables définies dans le bloc .
3. Evaluation du bloc, ce qui entraîne l'affectation des variables définies dans le bloc, avec leurs valeurs pour le nouvel objet.
4. Le nouvel objet est retourné comme résultat.

[Retour au sommaire]

4. Clonage des objets

Quand un objet parent est utilisé pour créer un nouvel objet, l'objet parent est cloné plutôt que "hérité".

Ceci signifie que si l'objet parent est modifié, il n'y a pas d'effet sur l'objet fils.

Pour illustrer ceci, le code suivant montre la création d'un compte bancaire avec l'objet *bank-account*, pour lequel les variables sont mises à "**none**" :

```
bank-account: make object! [
    first-name:
    last-name:
    account:
```

```
    solde: none
]
```

Pour utiliser le nouvel objet, des valeurs sont fournies lors de la création d'un compte client *luke* :

```
luke: make bank-account [
    first-name: "Luke"
    last-name: "Lakeswimmer"
    account: 89431
    solde: $1204.52
]
```

Puisque les nouveaux comptes sont initiés sur l'objet *bank-account* , il est pratique d'employer une fonction et quelques variables globales pour les créer.

```
last-account: 89431
bank-bonus: $10.00
make-account: func [
    "Returns a new account object"
    f-name [string!] "First name"
    l-name [string!] "Last name"
    start-solde [money!] "Starting solde"
][
    last-account: last-account + 1
    make bank-account [
        first-name: f-name
        last-name: l-name
        account: last-account
        solde: start-solde + bank-bonus
    ]
]
```

A présent, la création d'un nouveau compte pour le client *Fred* se réduira seulement à la ligne suivante :

```
fred: make-account "Fred" "Smith" $500.00
```

[Retour au sommaire]

5. L'accés aux objets

Les variables à l'intérieur des objets peuvent être atteintes avec les paths.

Un **path** est composé du nom de l'objet suivi par le nom de la variable à atteindre.

Ainsi, le code suivant permet d'atteindre les variables dans l'objet mon-exemple :

```
example/var1
```

```
example/var2
```

Quelques illustrations avec l'objet "compte bancaire" :

```
print luke/last-name
Lakeswimmer
print fred/solde
$510.00
```

Avec un **path**, les variables d'un objet peuvent aussi être modifiées :

```
fred/solde: $1000.00
print fred/solde
$1000.00
```

Vous pouvez utiliser aussi la fonction **in** pour accéder à des variables d'objet en récupérant leurs mots (words) depuis le contexte de l'objet :

```
print in fred 'solde
solde
```

Le mot (**word**) *'solde* fait partie du contexte de l'objet *Fred*.

Il est possible de connaître la valeur de *solde* dans le contexte de l'objet *Fred*, en utilisant la fonction **get** :

```
print get in fred 'solde
$1000.00
```

Le deuxième argument de la fonction **in** est un mot littéral (literal word).

Ceci vous permet de changer dynamiquement les mots selon vos besoins :

```
words: [first-name last-name solde]
foreach word words [print get in fred word]
FredSmith
$1000.00
```

Chaque mot dans le bloc est utilisé dans la boucle foreach pour obtenir sa valeur dans l'objet.

La fonction **in** peut aussi servir pour attribuer des valeurs aux variables d'un objet, en conjuguaison avec la fonction **set** .

```
set in fred 'solde $20.00
```

```
print fred/solde
$20.00
```

Si un mot n'est pas défini au sein d'un objet, la fonction **in** renvoie la valeur **none**.

Ceci peut être mis à profit pour déterminer si une variable existe ou non dans un objet.

```
if get in fred 'bank [print fred/bank]
```

[Retour au sommaire]

6. Fonctions et object (méthodes)

Un objet peut contenir des variables faisant référence à des fonctions dans l'objet.

Ce peut être utile, car les fonctions sont encapsulées dans le contexte de l'objet, et peuvent accéder à d'autres variables dans l'objet directement, sans passer par l'usage d'un **path**.

En guise d'exemple, l'objet *mon-autre-exemple* va inclure des fonctions qui vont calculer de nouvelles valeurs au sein de l'objet :

```
mon-autre-exemple: make object! [
    var1: 10
    var2: var1 + 10
    var3: now/time
set-time: does [var3: now/time]
calculate: func [value] [
    var1: value
    var2: value + 10
]
]
```

Remarquez que les fonctions peuvent se référer aux variables de l'objet directement, sans utiliser de **paths**.

Ceci est possible car les fonctions sont définies dans le même contexte que les variables auquelles elles accédent.

Pour définir un nouvel horaire pour la variable *var3* :

```
mon-autre-exemple/set-time
```

Cet exemple évalue la fonction qui va attribue à la variable *var3* l'heure courante.

Pour calculer de nouvelles valeurs pour *var1* et *var2* :

```
mon-autre-exemple/calculate 100
print example/var2
```

```
110
```

Dans le cas de l'objet *compte bancaire*, les fonctions pour un dépôt et un retrait peut être ajoutées à la définition courante :

```
bank-account: make bank-account [
    depot: func [amount [money!]] [
        solde: solde + amount
    ]
    retrait: func [amount [money!]] [
        either negative? solde [
            print ["Denied. Account overdrawn by"
                absolute solde]
        ][solde: solde - amount]
    ]
]
```

Ici, les fonctions se réfèrent à la variable *solde* directement au sein du contexte de l'objet.

Ceci parce qu'elles font elles-mêmes partie de ce contexte.

A présent, si un nouveau compte est créé, il contiendra les fonctions pour le dépôt et le retrait d'argent.

Par exemple :

```
lily: make-account "Lily" "Lakeswimmer" $1000
print lily/solde
$1010.00
lily/depot $100
print lily/solde
$1110.00
lily/retrait $2000
print lily/solde
-$890.00
lily/retrait $2.10
Denied. Account overdrawn by $890.00
```

[Retour au sommaire]

7. Prototype d'objets

N'importe quel objet peut servir de prototype pour créer de nouveaux objets.

Le compte bancaire *lily* créé précédemment peut être utilisé pour construire de nouveaux objets :

```
maya: make lily []
```

Ceci définit une instance de l'objet.

L'objet est une copie de l'objet *lily* et possède des valeurs identiques :

```
print lily/solde
-$890.00
print maya/solde
-$890.00
```

Vous pouvez modifier les nouveaux objets en fournissant de nouvelles valeurs à l'intérieur du bloc qui les définit :

```
maya: make lily [
    first-name: "Maya"
    solde: $10000
]
print maya/solde
$10000.00
maya/depot $500
print maya/solde
$10500.00
print maya/first-name
Maya
```

L'objet *lily* sert de prototype pour créer le nouvel objet *maya*.

Remarque:
un mot qui n'a pas été redéfini pour le nouvel objet continue d'avoir les valeurs de l'ancien objet :

```
print maya/last-name
Lakeswimmer
```

De nouveaux mots peuvent être ajoutés à l'objet :

```
maya: make lily [
    email: maya@example.com
    birthdate: 4-July-1977
]
```

[Retour au sommaire]

8. Référence à self

Chaque objet inclut une variable pré-définie appelée **self**.

A l'intérieur du contexte de l'objet, la variable **self** fait référence à l'objet lui-même.

Cette variable peut être utilisée pour passer à l'objet d'autres fonctions ou pour le retourner en tant que résultat d'une fonction.

Dans l'exemple suivant, la fonction *show-date* nécessite un objet en argument et c'est **self** qui lui est passé pour cela :

```
show-date: func [obj] [print obj/date]
example: make object! [
    date: now
    show: does [show-date self]
]
example/show
16-Jul-2000/11:08:37-7:00
```

Un autre exemple d'utilisation de la variable **self** est ici la fonction *new* pour le clonage de l'objet :

```
person: make object! [
    name: days-old: none
    new: func [name' birthday] [
        make self [
            name: name'
            days-old: now/date - birthday
        ]
    ]
]
lulu: person/new "Lulu Ulu" 17-May-1980
print lulu/days-old
7366
```

[Retour au sommaire]

9. Encapsulation

L'usage des objets est un bon moyen d'encapsuler un ensemble de variables qui ne devrait pas apparaître dans le contexte global.

Quand des variables d'une fonction sont définies comme globales, elles peuvent involontairement être modifiées par d'autres fonctions.

La solution à ce problème de variables globales est de les encapsuler dans un objet.

Ainsi, une fonction peut encore accéder à ses variables, mais celles-ci ne peuvent plus être accédées depuis le contexte global.

Par exemple :

```
bank: make object! [
last-account: 89431
bank-bonus: $10.00
set <i>make-account</i> func [
    "Returns a new account object"
    f-name [string!] "First name"
    l-name [string!] "Last name"
    start-solde [money!] "Solde Initial"
```

```
][
    last-account: last-account + 1
    make bank-account [
        first-name: f-name
        last-name: l-name
        account: last-account
        solde: start-solde + bank-bonus
    ]
]
]
```

Ici, les variables sont protégées de modifications accidentielles, car encapsulées dans le contexte de l'objet *bank*.

Remarque :

La fonction *make-account* a été définie en utilisant la fonction **set**, plutôt que par une affectation normale.

Avec l'usage de **set**, la fonction *make-account* devient une fonction du contexte global.

Cependant, si elle peut être utilisée de la même manière que d'autres fonctions, elle ne nécessite pas l'usage de **paths**.

```
bob: make-account "Bob" "Baker" $4000
```

[Retour au sommaire]

10. Réflectivité

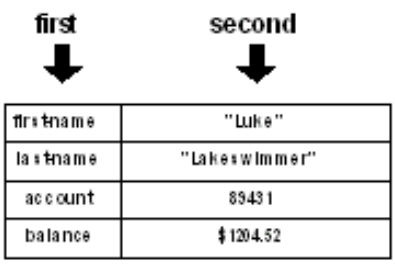
Comme beaucoup d'autres types de données REBOL, vous pouvez accéder aux composants des objets de tel sorte qu'il devient possible d'écrire des outils utiles pour les créer, les monitorer, ou les débugger.

Les fonctions **first** et **second** vous permettent d'accéder aux composants d'un objet.

La fonction **first** renvoie les mots définis pour un objet.

La fonction **second** renvoie les valeurs de ces mots.

Le diagramme suivant montre les liens entre les valeurs retournées par les fonctions **first** et **second**.



L'intéret de **first** est qu'elle permet d'obtenir la liste des mots de l'objet sans connaitre quoi que ce

soit sur lui :

```
probe first luke
[self first-name last-name account solde]
```

Dans l'exemple ci-dessus, la liste renvoyée contient le mot **self** référence à l'objet lui-même.

Vous pouvez exclure **self** de la liste en utilisant **next** :

```
probe next first luke
[first-name last-name account solde]
```

A présent, vous pouvez écrire une fonction qui va sonder le contenu d'un objet :

```
probe-object: func [object][
    foreach word next first object [
        print rejoin [word ":" tab get in object word]
    ]
]
probe-object fred
first-name: Luke
last-name: Lakeswimmer
account: 89431
solde: $1204.52
```

En sondant les objets de cette façon, attention à éviter les boucles infinies !

Par exemple, si vous essayer de connaître certains objets qui contiennent des références à eux-mêmes, votre code peut conduire à une boucle infinie.

C'est d'ailleurs la raison pour laquelle vous ne pouvez sonder l'objet **system** directement.
L'objet **system** contient en effet beaucoup de références à lui-même.

Chapitre 11 - Maths

Ce document est la traduction française du Chapitre 11 du User Guide de REBOL/Core, qui concerne les expressions mathématiques.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
1er juin 2005 19:33	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Présentation

REBOL permet d'assurer un ensemble complet d'opérations mathématiques et trigonométriques. La plupart de ces opérateurs peuvent manipuler plusieurs types de données, comme les nombres entiers et décimaux, les tuples, des valeurs de type temps et date. Certains de ces types de données peuvent même être mélangés ou forcés.

[Retour au sommaire]

3. Types de données scalaires

Les fonctions mathématiques de REBOL agissent de façon régulière sur une grande variété de types de données scalaires (numériques). Ces types de données incluent :

Type de données	Description
Integer!	nombres sur 32 bits sans décimales
Decimal!	nombres sur 64 bits avec décimales
Money!	valeurs monétaires sur 64 bits avec décimales
Time!	heures, minutes, secondes, dixièmes jusqu'à millième de secondes

Date!	jour, mois, année, heure, fuseau horaire
Pair!	coordonnées graphiques ou taille
Tuple!	versions, couleurs, adresses réseau

Voici ci-dessous quelques exemples pour illustrer un certain nombre d'opération sur les types de donnée scalaires. Notez que les opérateurs renvoient les résultats adéquats pour chaque type de données.

Les types de données **integer!** et **decimal!** :

```
print 2 + 1
3
print 2 - 1
1
print 2 * 10
20
print 20 / 10
2
print 21 // 10
1
print 2.2 + 1
3.2
print 2.2 - 1
1.2
print 2.2 * 10
22
print 2.2 / 10
0.22
print random 10
5
```

Le type de données **time!** :

```
print 2:20 + 1:40
4:00
print 2:20 + 5
2:20:05
print 2:20 + 60
2:21
print 2:20 + 2.2
2:20:02.2
print 2:20 - 1:20
1:00
print 2:20 - 5
2:19:55
print 2:20 - 120
2:18
print 2:20 * 2
4:40
```

```
print 2:20 / 2
1:10
print 2:20:01 / 2
1:10:00.5
print 2:21 // 2
0:00
print - 2:20
-2:20
print random 10:00
5:30:52
```

Le type de données **date!** :

```
print 1-Jan-2000 + 1
2-Jan-2000
print 1-Jan-2000 - 1
31-Dec-1999
print 1-Jan-2000 + 31
1-Feb-2000
print 1-Jan-2000 + 366
1-Jan-2001
birthday: 7-Dec-1944
print ["I've lived" (now/date - birthday) "days."]
I've lived 20305 days.
print random 1-1-2000
29-Apr-1695
```

Le type de données **money!** :

```
print $2.20 + $1
$3.20
print $2.20 + 1
$3.20
print $2.20 + 1.1
$3.30
print $2.20 - $1
$1.20
print $2.20 * 3
$6.60
print $2.20 / 2
$1.10
print $2.20 / $1.10
2
print $2.21 // 2
$0.21
print random $10.00
$6.00
```

Le type de données **pair!** :

```
print 100x200 + 10x20
110x220
```

```
print 10x10 + 3
13x13
print 10x20 * 2x4
20x80
print 100x100 * 3
300x300
print 100x30 / 10x3
10x10
print 100x30 / 10
10x3
print 101x32 // 10x3
1x2
print 101x32 // 10
1x2
print random 100x20
67x12
```

Le type de données **tuple!** :

```
print 1.2.3 + 3.2.1
4.4.4
print 1.2.3 - 1.0.1
0.2.2
print 1.2.3 * 3
3.6.9
print 10.20.30 / 10
1.2.3
print 11.22.33 // 10
1.2.3
print 1.2.3 * 1.2.3
1.4.9
print 10.20.30 / 10.20.30
1.1.1
print 1.2.3 + 7
8.9.10
print 1.2.3 - 1
0.1.2
print random 10.20.30
8.18.12
```

[Retour au sommaire]

4. Ordre de l'évaluation

Il y a deux règles à se rappeler pour l'évaluation d'expressions mathématiques :

- Les expressions sont évaluées de gauche à droite.
- Les opérateurs ont priorité sur les fonctions

L'évaluation des expressions de gauche à droite est indépendante du type d'opérateur utilisé. Par exemple :

```
print 1 + 2 * 3
```

```
9
```

Dans l'exemple ci-dessus, notez que le résultat n'est pas sept, comme cela le serait si la multiplication avait priorité sur l'addition.

Remarque importante

Le fait que les expressions mathématiques soient évaluées de la gauche vers la droite sans se soucier des opérateurs donne un comportement différent de la plupart des autres langages de programmation. Beaucoup de langages possèdent des règles de priorité où vous devez vous rappeler ce qui détermine l'ordre dans lequel les opérateurs seront évalués. Par exemple, une multiplication est faite avant une addition. Certains langages possèdent plus d'une dizaine de règles de ce genre.

En REBOL, plutôt que d'imposer à l'utilisateur de se remémorer les priorités des opérateurs, vous avez seulement à vous rappeler la règle "de-gauche-à-droite".

Encore plus important, pour un code poussé, perfectionné, comme des expressions qui manipulent d'autres expressions (de la réflectivité, par exemple), vous n'avez pas besoin de réordonner les termes sur la base d'une priorité. L'ordre d'évaluation demeure simple.

Pour la plupart des expressions mathématiques, l'évaluation de gauche à droite fonctionne assez bien et est simple à se rappeler. D'un autre côté, comme cette règle est différente d'autres langages de programmation, elle peut être la cause d'erreurs de programmation, donc soyez vigilants.

La meilleure solution est de vérifier votre travail. Vous pouvez aussi utiliser des parenthèses si nécessaire, afin de clarifier vos expressions (voir ci-dessous), et vous pouvez toujours saisir votre expression dans la console, pour vérifier le résultat.

S'il est nécessaire d'évaluer une expression dans un autre ordre, réordonnez-la ou utilisez des parenthèses :

```
print 2 * 3 + 1
7
print 1 + (2 * 3)
7
```

Quand des fonctions sont mélangées avec des opérateurs, les opérateurs sont évalués en premier, puis les fonctions :

```
print absolute -10 + 5
5
```

Dans l'exemple ci-dessus, l'addition est d'abord réalisée, et son résultat passé à la fonction "valeur absolue".

Dans l'exemple suivant :

```
print 10 + sine 30 + 60
11
```

l'expression est évaluée dans cet ordre :

```
30 + 60 => 90
sine 90 => 1
10 + 1 => 11
print
```

Pour changer l'ordre afin que le sinus de 30 soit calculé en premier, utilisez des parenthèses :

```
print 10 + (sine 30) + 60
70.5
```

ou réorganisez l'expression :

```
print 10 + 60 + sine 30
70.5
```

[Retour au sommaire]

5. Fonctions et Opérateur standards

Cette section décrit les fonctions et opérateurs mathématiques standards utilisés en REBOL.

5.1 absolute

Les expressions :

```
absolute value

abs value
```

renvoient la valeur absolue de l'argument *value*.
Les types de données possibles sont : integer, decimal, money, time, pair.

```
print absolute -10
10
print absolute -1.2
```

```
1.2
print absolute -$1.2
$1.20
print absolute -10:20
10:20
print absolute -10x-20
10x20
```

5.2 add

Les expressions :

```
value1 + value2

add value1 value2
```

renvoient la somme des valeurs *value1* et *value2*.
S'utilise avec les types de données integer, decimal, money, time, tuple, pair, date, char.

```
print 1 + 2
3
print 1.2 + 3.4
4.6
print 1.2.3 + 3.4.5
4.6.8
print $1 + $2
$3.00
print 1:20 + 3:40
5:00
print 10x20 + 30x40
40x60
print #"A" + 10
K
print add 1 2
3
```

NdT : remarquez que l'expression

```
+ 1 2
3
```

est aussi valable, quoique moins "naturelle".
Il en est ainsi avec plusieurs opérateurs.

5.3 complement

L'expression :

```
complement value
```

renvoie le complément numérique (bitwise complement) d'une valeur.
Pour types de données integer, decimal, tuple, logic, char, binary, string, bitset, image.

```
print complement 10
-11
print complement 10.5
-11
print complement 100.100.100
155.155.155
```

NdT: il est possible d'utiliser **complement** sur des caractères :

```
complement "I"
"¶"       (le symbole de saut de ligne de Ms-Word)
complement "V"
"©"       (le symbole Copyright)
```

5.4 divide

Les expressions :

```
value1 / value2

divide value1 value2
```

retournent le résultat de la division de *value1* par *value2*.
S'utilise sur les types de données integer, decimal, money, time, tuple, pair, char .

```
print 10 / 2
5
print 1.2 / 3
0.4
print 11.22.33 / 10
1.2.3
print $12.34 / 2
$6.17
print 1:20 / 2
0:40
print 10x20 / 2
5x10
```

```
print divide 10 2
5
```

NdT : l'expression

```
/ 1 2
0.5
```

marche également.

5.5 multiply

Les expressions :

```
value1 * value2

multiply value1 value2
```

renvoient le résultat de la multiplication de *value1* par *value2*.
Fonctionne avec les datatypes integer, decimal, money, time, tuple, pair, char.

```
print 10 * 2
20
print 1.2 * 3.4
4.08
print 1.2.3 * 3.4.5
3.8.15
print $10 * 2
$20.00
print 1:20 * 3
4:00
print 10x20 * 3
30x60
print multiply 10 2
20
```

NdT : l'expression

```
* $10 2
$20.00
```

est également valable.

5.6 negate

Les expressions :

```
- value

negate value
```

changent le signe de la valeur. Les types de données possibles sont : integer, decimal, money, time, pair, char.

```
print - 10
-10
print - 1.2
-1.2
print - $10
-$10.00
print - 1:20
-1:20
print - 10x20
-10x-20
print negate 10
-10
```

5.7 random

L'expression :

```
random value
```

renvoie une valeur aléatoire qui est inférieure ou égale à la valeur de l'argument.

Notez que pour les nombres entiers, **random** commence à 1, pas à 0, et va inclure la valeur de l'argument fourni. Ceci permet à **random** d'être utilisé directement dans des fonctions comme **pick**.

Quand un nombre décimal est utilisé, le résultat est du type décimal, arrondi à un entier.

Le raffinement **/seed** réinitialise le générateur de nombres aléatoires. Utilisez d'abord le raffinement **/seed** avec **random** si vous voulez générer un nombre unique aléatoire. Vous pouvez utiliser la date et l'heure courante pour fabriquer une base unique :

```
random/seed now
```

S'utilise avec les types de données integer, decimal, money, time, tuple, pair, date, char, string, et block.

```
print random 10
5
print random 10.5
2
print random 100.100.100
79.95.66
print random $100
$32.00
print random 10:30
6:37:33
print random 10x20
2x4
print random 30-Jun-2000
27-Dec-1171
```

5.8 remainder

Les expressions :

```
value1 // value2

remainder value1 value2
```

renvoient le reste de la division de *value1* par *value2*.
Fonctionne avec les datatypes integer, decimal, money, time, tuple, pair .

```
print 11 // 2
1
print 11.22.33 // 10
1.2.3
print 11x22 // 2
1x0
print remainder 11 2
1
```

5.9 subtract

Les expressions :

```
value1 - value2

subtract value1 value2
```

renvoient le résultat de la soustraction entre *value2* et *value1*.
S'utilise avec les types de données integer, decimal, money, time, tuple, pair, date, char.

```
print 2 - 1
```

```
1
print 3.4 - 1.2
2.2
print 3.4.5 - 1.2.3
2.2.2
print $2 - $1
$1.00
print 3:40 - 1:20
2:20
print 30x40 - 10x20
20x20
print #"Z" - 1
Y
print subtract 2 1
1
```

[Retour au sommaire]

6. Conversion de type

Lorsque des opérations mathématiques sont réalisées entre des types de données différents, normalement, le type de données non entier ou non décimal est retourné. Quand des entiers sont combinés avec des nombres décimaux, c'est le datatype décimal qui est retourné.

[Retour au sommaire]

7. Fonctions de comparaison

Toutes les fonctions de comparaison renvoient une valeur logique : **true** ou **false**.

7.1 equal

Les expressions :

```
value1 = value2

equal? value1 value2
```

renvoient **true** si la première et la seconde valeur sont égales.
Fonctionne avec les types de données : integer, decimal, money, time, date, tuple, char et series.

```
print 11-11-99 = 11-11-99
true
print equal? 111.112.111.111 111.112.111.111
true
print #"B" = #"B"
true
print equal? "a b c d" "A B C D"
true
```

7.2 greater

Les expressions :

```
value1 > value2

greater? value1 value2
```

retournent **true** si la première valeur est supérieure à la seconde valeur.
S'utilise avec les types de données integer, decimal, money, time, date, tuple, char et series.

```
print 13-11-99 > 12-11-99
true
print greater? 113.111.111.111 111.112.111.111
true
print #"C" > #"B"
true
print greater? [12 23 34] [12 23 33]
true
```

7.3 greater-or-equal

Les expressions :

```
value1 >= value2

greater-or-equal? value1 value2
```

retournent **true** si la première valeur est supérieure ou égale à la seconde valeur.
S'utilise avec les types de données integer, decimal, money, time, date, tuple, char et series.

```
print 11-12-99 >= 11-11-99
true
print greater-or-equal? 111.112.111.111 111.111.111.111
true
print #"B" >= #"A"
true
print greater-or-equal? [b c d e] [a b c d]
true
```

7.4 lesser

Les expressions :

```
value1 < value2

lesser? value1 value2
```

retournent **true** si la première valeur est inférieure à la seconde valeur.
S'utilise avec les types de données integer, decimal, money, time, date, tuple, char et series.

```
print 25 <; 50
true
print lesser? 25.3 25.5
true
print $2.00 < $2.30
true
print lesser? 00:10:11 00:11:11
true
```

7.5 lesser-or-equal

Les expressions :

```
value1 <= value2

lesser-or-equal? value1 value2
```

retournent **true** si la première valeur est inférieure ou égale à la seconde valeur.
S'utilise avec les types de données integer, decimal, money, time, date, tuple, char et series.

```
print 25 <= 25
true
print lesser-or-equal? 25.3 25.5
true
print $2.29 <= $2.30
true
print lesser-or-equal? 11:11:10 11:11:11
true
```

7.6 not equal to

Les expressions :

```
value1 <> value2

not-equal? value1 value2
```

retournent **true** si la première valeur n'est pas égale à la seconde valeur.
S'utilise avec les types de données integer, decimal, money, time, date, tuple, char et series.

```
print 26 v 25
true
print not-equal? 25.3 25.5
true
print $2.29 <> $2.30
```

```
true
print not-equal? 11:11:10 11:11:11
true
```

7.7 same

Les expressions :

```
value1 =? value2

same? value1 value2
```

renvoient **true** si les deux mots font référence à la même valeur.
Par exemple, lorsque vous voulez savoir si deux mots font référence à la même valeur d'index dans une série.
Fonctionne avec tous les types de données.

```
reference-one: "abcdef"
reference-two: reference-one
print same? reference-one reference-two
true
reference-one: next reference-one
print same? reference-one reference-two
false
reference-two: next reference-two
print same? reference-one reference-two
true
reference-two: copy reference-one
print same? reference-one reference-two
false
```

7.8 strict-equal

Les expressions :

```
value1 == value2

strict-equal? value1 value2
```

retournent **true** si la première et la seconde valeurs sont strictement identiques. **Strict-equal** peut être utilisée comme la version sensible à la casse de **equal?** (=) pour les chaînes de caractères et pour différencier les entiers des décimaux lorsque les valeurs sont identiques.
Fonctionne avec tous les types de données.

```
print strict-equal? "abc" "ABC"
false
print equal? "abc" "ABC"
true
print strict-equal? "abc" "abc"
```

```
true
print strict-equal? 1 1.0
false
print equal? 1 1.0
true
print strict-equal? 1.0 1.0
true
```

7.9 strict-not-equal

L'expression :

```
strict-not-equal? value1 value2
```

renvoie **true** si la première et la seconde valeurs ne sont pas strictement égales. **strict-not-equal** peut être utilisée comme la version sensible à la casse de **not-equal?** (<>) pour les chaînes de caractères et pour différencier les entiers des décimaux lorsque leurs valeurs sont identiques. Fonctionne avec tous les types de données.

```
print strict-not-equal? "abc" "ABC"
true
print not-equal? "abc" "ABC"
false
print strict-not-equal? "abc" "abc"
false
print strict-not-equal? 1 1.0
true
print not-equal? 1 1.0
false
print strict-not-equal? 1.0 1.0
false
```

[Retour au sommaire]

8. Fonctions Logarithmiques

8.1 exp

L'expression :

```
exp value
```

calcule E (nombre naturel) à la puissance de l'argument.

8.2 log-10

L'expression :

```
log-10 value
```

renvoie le logarithme de base 10 de l'argument.

8.3 log-2

L'expression :

```
log-2 value
```

renvoie le logarithme de base 2 de l'argument.

8.4 log-e

L'expression :

```
log-e value
```

renvoie le logarithme naturel (base E) de l'argument.

8.5 power

Les expressions :

```
value1 ** value2

power value1 value2
```

renvoient le résultat du calcul de *value1* à la puissance *value2*.

8.6 square-root

L'expression :

```
square-root value
```

renvoie la racine carrée de l'argument *value*.

[Retour au sommaire]

9. Fonctions Trigonométriques

Les fonctions trigonométriques travaillent normalement en degrés. Il faut utiliser le raffinement **/radians** avec l'une ou l'autre des fonctions trigonométriques pour travailler en radians.

9.1 arccosine

L'expression :

```
arccosine value
```

renvoie la fonction cosinus inverse pour l'argument.

9.2 arcsine

L'expression :

```
arcsine value
```

renvoie la fonction sinus inverse pour l'argument.

9.3 arctangent

L'expression :

```
arctangent value
```

renvoie la fonction tangente inverse pour l'argument.

9.4 cosine

L'expression :

```
cosine value
```

renvoie le cosinus de l'argument.

9.5 sine

L'expression :

```
sine value
```

renvoie le sinus de l'argument.

9.6 tangent

L'expression :

```
```

```
tangent value
```

renvoie la tangente de l'argument.

[Retour au sommaire]

10. Fonctions logiques

Les fonctions logiques peuvent être effectuées sur des valeurs logiques et sur des valeurs scalaires incluant les types de données integer, char, tuple, et bitset.

Avec des valeurs logiques, les fonctions logiques renvoient des valeurs booléennes. Quand elles utilisent d'autres types de valeurs, les fonctions logiques travaillent au niveau des bits.

10.1 and

La fonction **and** compare deux valeurs logiques et renvoie **true** seulement si les deux valeurs sont elles aussi **true** :

```
print (1 < 2) and (2 < 3)
true
print (1 < 2) and (4 < 3)
false
```

Quand elle est utilisée avec des nombres entiers, la fonction **and** effectue une comparaison bit à bit, et renvoie 1 si chacun des bits est à 1, ou 0 si ni l'un ni l'autre n'est 1 :

```
print 3 and 5
1
```

10.2 or

La fonction **or** compare deux valeurs logiques et renvoie **true** si l'une ou l'autre des deux est **true**, ou renvoie **false** si les deux sont **false**.

```
print (1 < 2) or (2 < 3)
true
print (1 < 2) or (4 < 3)
true
print (3 < 2) or (4 < 3)
false
```

Quand elle est utilisée avec des nombres entiers, la fonction **or** effectue une comparaison bit à bit, et renvoie 1 si l'un des bits est 1, ou 0 si l'un et l'autre bit sont à 0 :

```
print 3 or 5
7
```

10.3 xor

La fonction **xor** compare deux valeurs logiques et retourne **true** si et seulement si l'une des valeurs est **true** et l'autre **false**.

```
print (1 < 2) xor (2 < 3)
false
print (1 < 2) xor (4 < 3)
true
print (3 < 2) xor (4 < 3)
false
```

Utilisée avec des nombres entiers, **xor** compare bit à bit ces nombres et renvoie 1 si et seulement si un bit est à 1 tandis que l'autre est à 0. Sinon, elle renvoie 0 :

```
print 3 xor 5
6
```

10.4 complement

La fonction **complement** renvoie le complément logique ou binaire d'une valeur. Elle est utilisée pour avoir l'inverse binaire de nombres entiers ou inverser un ensemble de bits (bitsets).

```
print complement true
false
print complement 3
-4
```

10.5 not

Pour une valeur logique, la fonction **not** renverra **true** si l'argument est **false** et **false** si, au contraire, l'argument est **true**. Elle n'effectue pas d'opérations numériques binaires.

```
print not true
false
print not false
true
```

[Retour au sommaire]

11. Erreurs

Les erreurs mathématiques sont émises quand des opérations illégales sont réalisées, ou quand un débordement de calcul (overflow) se produit.

Les erreurs suivantes peuvent être rencontrées dans les opérations mathématiques.

11.1 Tentative de division par zéro

Une tentative a été faite de diviser un nombre par 0.

```
1 / 0
** Math Error: Attempt to divide by zero
** Where: connect-to-link
** Near: 1 / 0
```

11.2 Débordement de calcul

Une tentative de calcul d'un nombre trop grand pour REBOL a été faite.

```
1E+300 + 1E+400
** Math Error: Math or number overflow
** Where: connect-to-link
** Near: 1 / 0
```

11.3 Nombre positif requis

Une tentative de calcul a été faite avec un nombre négatif sur un opérateur mathématique qui accepte juste des nombres positifs.

```
log-10 -1
** Math Error: Positive number required
** Where: connect-to-link
** Near: log-10 -1
```

11.4 Impossible d'utiliser l'opérateur avec un type de données

Une tentative de calcul entre des types de données incompatibles. Le type de données du second argument dans l'opération est retourné tel quel.

```
10:30 + 1.2.3
** Script Error: Cannot use add on time! value
** Where: connect-to-link
** Near: 10:30 + 1.2.3
```

Chapitre 12 - Fichiers

Ce document est la traduction française du Chapitre 12 du User Guide de REBOL/Core, qui concerne les fichiers.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email

27 juin 2005 19:33	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Présentation

Une caractéristique importante de la puissance de REBOL est sa capacité à manipuler les fichiers et les répertoires. REBOL fournit plusieurs fonctions sur mesure permettant des opérations allant de la lecture d'un simple fichier jusqu'à l'accès direct aux fichiers et répertoires. Pour plus d'informations sur l'accès direct aux fichiers et répertoires, voir le chapitre sur les Ports.

[Retour au sommaire]

3. Noms et Paths

REBOL fournit une convention de nommage des chemins et des fichiers qui se veut indépendante du type de machines.

3.1 Noms des fichiers

Dans les scripts, les noms de fichiers et les chemins sont écrits avec un signe pourcentage **(%)** suivi par une suite de caractères :

```
%examples.r
%big-image.jpg
%graphics/amiga.jpg
%/c/plug-in/video.r
%//sound/goldfinger.mp3
```

Le signe pourcentage **(%)** est nécessaire pour éviter que les noms de fichiers soient interprétés comme des mots au sein du langage.

Bien que cela ne soit pas une bonne habitude, des espaces peuvent être inclus dans les noms de fichiers en incluant le nom du fichier entre des guillemets **(" ")**.

Les guillemets évitent que le nom du fichiers soit interprété comme une série de plusieurs mots :

```
%"this file.txt"
%"cool movie clip.mpg"
```

La convention standard Internet d'utiliser le signe pourcentage **(%)** et un code hexadécimal est aussi autorisée pour les noms de fichiers. Quand ceci est fait, les guillemets ne sont pas requis.

Les noms de fichiers précédents peuvent encore être écrits sous la forme :

```
%this%20file.txt
%cool%20movie%20clip.mpg
```

Notez que le suffixe standard pour les scripts REBOL est "**.r**". Sur les systèmes où cette convention produit un conflit avec un autre type de fichier, un suffixe "**.reb**" peut être utilisé à la place.

3.2 Chemins

Les chemins vers les fichiers sont décrits par un signe pourcentage **(%)** suivi par une séquence de noms de répertoires qui sont chacun séparés par le signe **(/)**.

```
%dir/file.txt
%/file.txt
%dir/
%/dir/
%/dir/subdir/
%../dir/file.txt
```

Le caractère standard pour séparer les répertoires est le caractère slash **(/)**, pas le backslash **(\)**. Si des caractères "backslash" sont trouvés, ils sont convertis en caractères "slash" :

```
probe %\some\cool\movie.mpg
%/some/cool/movie.mpg
```

REBOL fournit une méthode standard, indépendante du système d'exploitation pour spécifier les chemins vers les répertoires. Les chemins peuvent être relatifs au répertoire courant ou absolus à partir du plus haut niveau de l'arborescence du système d'exploitation.

Les chemins vers les fichiers qui ne commencent pas par un symbole slash **(/)** sont des chemins relatifs.

```
%docs/intro.txt
%docs/new/notes.txt
%"new mail/inbox.mbx"
```

La convention standard qui utilise deux points successifs **(..)** pour indiquer un répertoire parent ou un point unique **(.)** pour faire référence au répertoire courant est aussi supportée. Par exemple :

```
%.
%./
%./file.txt
%..
%../
%../script.r
%../../plans/schedule.r
```

Les chemins vers les fichiers utilisent la convention Internet qui fait commencer un chemin absolu avec un slash **(/)**. Le slash indique le point de départ depuis le plus haut niveau de l'arborescence du système. (Généralement, les chemins absolus devraient être évités pour rendre les scripts indépendants du contexte de la machine.

L'exemple :

```
%/home/file.txt
```

devrait référer à un volume disque ou une partition nommée "*home*".

Voici d'autres exemples :

```
%/ram/temp/test.r
%/cd0/scripts/test/files.r
```

Pour faire référence à un disque C comme cela est souvent le cas pour Windows, la notation est :

```
%/C/docs/file.txt
%"/c/program files/qualcomm/eudora mail/out.mbx"
```

Notez que dans les lignes précédentes, le disque C n'est pas écrit avec :

```
%c:/docs/file.txt
```

L'exemple précédent n'est pas indépendant du format utilisé pour la machine et provoque une erreur.

Si le premier nom de répertoire est absent, et que le chemin commence avec deux slashs consécutifs, **(*//*)**, alors le chemin du fichier est relatif au volume courant :

```
%//docs/notes
```

3.3 Sensibilité à la casse

Par défaut, en REBOL, les noms de fichiers ne sont pas sensibles à la casse des caractères. Malgré tout, lorsque de nouveaux fichiers sont créés via le langage, leurs noms conservent la casse avec lesquels ils sont créés :

```
write %Script-File.r file-data
```

L'exemple précédent crée un nom de fichier avec un S et un F majuscules.

De plus, lorsque des noms de fichiers sont issus de répertoires, leur casse est conservée :

```
print read %/home
```

Pour les systèmes sensibles à la casse, comme UNIX, REBOL trouve la correspondance la plus proche pour le nom de fichier.

Par exemple, si un script demande à lire %test.r, mais trouve seulement %TEST.r, ce fichier %TEST.r sera lu. Ce comportement est nécessaire pour permettre de conserver des scripts indépendants du contexte machine.

3.4 Fonctions de noms de fichiers

Diverses fonctions sont fournies pour vous aider à créer des noms de fichiers et des chemins (paths). Elles sont listées ci-dessous :

to-file	Convertit des chaînes de caractères et des blocs en noms de fichiers ou en chemin vers un fichier.
split-path	Scinde un chemin en deux parties : celle relative au répertoire et celle relative au nom de fichier .
clean-path	Renvoie un chemin absolu qui est équivalent à n'importe quel chemin fourni contenant (..) ou (.).
what-dir	Renvoie le chemin absolu du répertoire courant.

[Retour au sommaire]

4. Lecture des fichiers

Les fichiers sont lus comme des séries de caractères (mode texte) ou d'octets (mode binaire). La source pour le fichier est soit un fichier local sur votre système, ou un fichier à partir du réseau.

4.1 Lecture de fichiers texte

Pour lire un fichier texte local, utilisez la fonction **read** :

```
text: read %file.txt
```

La fonction **read** renvoie une chaîne qui contient le texte entier du fichier. Dans l'exemple ci-dessus, la variable *text* fait référence à cette chaîne.

Au sein de cette chaîne renvoyée par **read**, les fins de lignes sont convertis en caractères newline, indifféremment du genre de fin de ligne qu'utilise votre système d'exploitation. Ceci permet d'écrire des scripts qui recherche des sauts de lignes, sans se soucier des caractères particuliers qui constitue une fin de ligne.

```
next-line: next find text newline
```

Un fichier peut aussi être lu en séparant les lignes, qui sont stockées dans un bloc :

```
lines: read/lines %file.txt
```

Voir le paragraphe "Conversion de Lignes" pour plus d'informations concernant newline et la lecture ligne par ligne.

Pour lire un fichier par morceaux, utilisez la fonction **open** qui est décrite dans le chapitre sur les Ports.

Pour voir le contenu d'un fichier texte, vous pouvez le lire en utilisant **read** et l'afficher en utilisant **print** :

```
print read %service.txt
I wanted the gold, and I sought it,I scrabbled and mucked like
a slave.
```

4.2 Lecture de fichiers binaire

Pour lire un fichier binaire comme une image, un programme ou un son, utilisez **read/binary** :

```
data: read/binary %file.bin
```

La fonction **read/binary** renvoie une série binaire qui comprend le contenu entier du fichier. Dans l'exemple ci-dessus, la variable data fait référence à la série binaire. Aucune conversion d'aucun type n'est réalisée pour ce fichier.

Pour lire un fichier binaire par morceaux, utilisez la fonction **open** comme décrite dans le chapitre sur les Ports.

4.3 Lecture au travers du réseau

Les fichiers peuvent être lus à partir du réseau. Par exemple, pour voir un fichier texte à partir du réseau en utilisant le protocole HTTP :

```
print read http://www.rebol.com/test.txt
Hellotherenewuser!
```

Le fichier peut être écrit localement en une ligne de code :

```
write %test.txt read http:/www.rebol.com/test.txt
```

Dans le processus d'écriture, le fichier aura ses terminaisons de ligne converties en ce qui est utilisé pour cela sur votre système d'exploitation.

Pour lire et sauver un fichier binaire, comme une image, utilisez la ligne suivante :

```
write %image.jpg read/binary http:/www.rebol.com/image.jpg
```

Voyez le chapitre sur les protocoles Réseau pour plus d'information et d'exemples sur les moyens d'accéder à des fichiers au travers le réseau.

[Retour au sommaire]

5. Ecriture de fichiers

Vous pouvez écrire un fichier de caractères (texte) ou d'octets (binaire). L'emplacement du fichier peut soit être local sur votre système, soit sur le réseau.

5.1 Ecriture de fichiers texte

Pour écrire un fichier texte localement, utilisez la ligne de code suivante :

```
write %file.txt "sample text here"
```

Celle-ci écrit le texte entier dans le fichier. Si un fichier contient des caractères newline, ils seront convertis en ceux que votre système d'exploitation utilise. Ceci permet de manipuler les fichiers d'une manière homogène, mais de les écrire en utilisant la convention en vigueur sur votre système.

Par exemple, la ligne de code suivante transforme n'importe quel texte ayant un style de terminaison de ligne (UNIX, Macintosh, PC, Amiga) en celui utilisé localement par votre système :

```
write %newfile.txt read %file.txt
```

La ligne précédente lit le fichier en convertissant les fins de ligne vers le standard REBOL, puis à l'écriture du fichier, en convertissant celui-ci au format propre au système d'exploitation local.

Pour ajouter quelque chose à la fin d'un fichier, utilisez le raffinement **/append** :

```
write/append %file.txt "encore du texte"
```

Un fichier peut aussi être écrit à partir de lignes distinctes stockées dans un bloc.

```
write/lines %file.txt lines
```

Pour écrire un fichier texte morceaux par morceaux, utilisez la fonction **open** décrite dans le chapitre sur les Ports.

5.2 Ecriture de fichiers binaires

Pour écrire des fichiers binaires comme une image, un programme, un son, utilisez **write/binary** :

```
write/binary %file.bin data
```

La fonction **write/binary** crée le fichier si celui-ci n'existe pas ou l'écrase s'il existe. Aucune conversion d'aucune sorte n'est réalisée pour le fichier.

Pour écrire un fichier binaire morceaux par morceaux, utilisez la fonction **open** décrite dans le chapitre sur les Ports.

5.3 Ecriture de fichiers sur le réseau

Les fichiers peuvent aussi être écrits sur le réseau. Par exemple, pour écrire un fichier texte en utilisant le protocole FTP, utilisez :

```
write ftp://ftp.domain.com/file.txt "save this text"
```

Le fichier peut être lu localement et écrit sur un emplacement en réseau via une ligne comme celle-ci

```
write ftp://ftp.domain.com/file.txt read %file.txt
```

Dans le processus, le fichier a ses fins de lignes convertis au format standard CRLF. Pour écrire un fichier binaire comme une image, via le réseau, utilisez le code suivant :

```
write/binary ftp://ftp.domain.com/file.txt/image.jpg read/binary %image.jpg
```

Voyez le chapitre sur les protocoles Réseau pour plus d'information et d'exemples sur les moyens d'accéder à des fichiers via le réseau.

[Retour au sommaire]

6. Conversion de Ligne

Quand un fichier est lu en tant que texte, toutes les fins de lignes sont converties en caractères **newline** (line feed). Les caarctères LF (utilisés comme caractères de fin de lignes pour Amiga, Linux, et les systèmes UNIX), les retours chariots CR (utilisés sur Macintosh) ou la combinaison CR/LF (PC et Internet) sont tous transformés en leurs équivalents **newline**.

L'usage d'un caractère standard au sein d'un script permet de le faire marcher indépendamment du contexte machine. Par exemple, pour chercher et compter tous les caractères **newligne** à l'intérieur d'un fichier texte :

```
text: read %file.txt
count: 0
while [spot: find text newline][
    count: count + 1
    text: next spot
]
```

La conversion de ligne est aussi pratique pour la lecture de fichiers distants :

```
text: read ftp://ftp.rebol.com/test.txt
```

Quand un fichier est écrit, le caractère **newline** est converti dans le type de fin de ligne pour le système d'exploitation cible. Par exemple, le caractère newline est transformé en CRLF pour les PCs, LF sur UNIX ou AMIGA, ou CR pour un Macintosh. Les fichiers sur le réseau sont écrits avec CRLF.

La fonction suivante transforme n'importe quel texte, quel que soit le style de fin de ligne en celui qu'utilise le système d'exploitation local :

```
convert-lines: func [file] [write file read file]
```

Le fichier est lu et tous les caractères de fin de ligne sont transformés en caractère newline, puis le fichier est écrit et les caractères newline sont convertis dans le type nécessaire pour le système d'exploitation local.

La conversion de ligne peut être désactivée en lisant le fichier texte en mode binaire.

Par exemple, la ligne suivante :

```
write/binary %newfile.txt read/binary %oldfile.txt
```

préserve les fins de ligne du fichier texte original (*%oldfile.txt*).

[Retour au sommaire]

7. Blocs de Lignes

Les fichiers textes peuvent facilement être atteints et gérés sous forme de lignes individuelles, plutôt que comme une seule série de caractères. Par exemple, pour lire un fichier sous la forme d'un bloc de lignes :

```
lines: read/lines %service.txt
```

L'exemple précédent renvoie un bloc contenant une série (au sens REBOL) de chaînes de caractères (une pour chaque ligne), sans caractères de fin de ligne. Les lignes vierges sont représentées par des chaînes vides.

Pour afficher une ligne spécifique, vous pouvez utiliser le code suivant :

```
print first lines
print last lines
print pick lines 100
print lines/500
```

Pour afficher toutes les lignes d'un fichier, utilisez l'exemple de code suivant.

```
foreach line lines [print line]
I wanted the gold, and I sought it,
I scrabbled and mucked like a slave.
```

```
Was it famine or scurvy -- I fought it;
I hurled my youth into a grave.
I wanted the gold, and I got it --
Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
And somehow the gold isn't all.
```

Pour afficher toutes les lignes qui contiennent la chaîne "*gold*", utilisez la ligne de code suivante :

```
foreach line lines [
   if find line "gold" [print line]
]
I wanted the gold, and I sought it,
I wanted the gold, and I got it --
And somehow the gold isn't all.
```

Vous pouvez écrire un fichier texte ligne par ligne en utilisant la fonction **write** avec le raffinement **/lines** :

```
write/lines %output.txt lines
```

Pour écrire un fichier à partir de lignes spécifiques d'un bloc, utilisez :

```
write/lines %output.txt [
    "line one"
    "line two"
    "line three"
]
```

En fait, les fonctions **read/lines** et **write/lines** peuvent être combinées pour traiter des fichiers ligne par ligne. Par exemple, le code suivant efface tous les commentaires d'un script REBOL :

```
script: read/lines %script.r
foreach line script [
    where: find line ";"
    if where [clear where]
]
write/lines %script1.r script
```

Le bout de script précédent est indiqué à des fins de démonstration. En effet, en plus d'effacer les commentaires, le code effacerait aussi les points virgules valides présents dans les chaînes de caractères entre apostrophes.

Les fichiers peuvent également être lus ligne par ligne depuis le réseau :

```
data: read/lines http://www.rebol.com

print pick (read/lines ftp://ftp.rebol.com/test.txt) 3
```

```
new
```

Le rafinement **/lines** peut aussi être utilisé avec la fonction **open** pour lire une ligne à la fois à partir d'une saisie à la console. Voir le chapitre sur les Ports pour plus d'information.

De surcroît, **/lines** peut servir, avec le raffinement **/append**, à ajouter des lignes à un fichier, depuis un bloc.

[Retour au sommaire]

8. Information sur les fichiers et les répertoires

Il existe de nombreuses fonctions permettant d'avoir des informations utiles sur un fichier comme : s'il existe, sa taille en octets, lorsqu'il a été modifié, ou s'il s'agit d'un répertoire.

8.1 Contrôle de Répertoire

Pour déterminer si un nom de fichier est celui d'un répertoire, utilisez la fonction **dir?**.

```
print dir? %file.txt
false
print dir? %.
true
```

La fonction **dir?** fonctionne également avec la plupart des protocoles réseau :

```
print dir? ftp://www.rebol.com/pub/
true
```

8.2 Existence de fichier

Pour déterminer si un fichier existe, utilisez la fonction **exists?** :

```
print exists? %file.txt
```

Pour savoir si un fichier existe avant d'essayer de le lire :

```
if exists? file [text: read file]
```

Pour éviter d'écraser un fichier, vous pouvez contrôler sa présence :

```
if not exists? file [write file data]
```

La fonction **exists?** fonctionne aussi avec la plupart des protocoles réseau :

```
print exists? ftp://www.rebol.com/file.txt
```

8.3 Taille de fichier

Pour obtenir la taille en octets d'un fichier, utilisez la fonction **size?** :

```
print size? %file.txt
```

La fonction **size?** est utilisable avce la plupart des protocoles réseau :

```
print size? ftp://www.rebol.com/file.txt
```

8.4 Date de modification d'un fichier

Pour obtenir la date à laquelle un fichier a été modifié, utilisez la fonction **modified?** :

```
print modified? %file.txt
30-Jun-2000/14:41:55-7:00
```

Tous les systèmes d'exploitation ne conservent pas la date de création d'un fichier, donc pour gardez les scripts REBOL indépendants du système d'exploitation, utilisez **modified?** juste lorsque la date de dernière modification est accessible.

La fonction **modified?** fonctionne aussi avec la plupart des protocoles réseaux :

```
print modified? ftp://www.rebol.com/file.txt
```

8.5 Information relative à un fichier

La fonction **info?** récupére toutes les informations sur les fichiers et les répertoires en même temps. Ces informations sont retournées sous la forme d'un objet :

```
probe info? %file.txt
make object! [
    size: 306
    date: 30-Jun-2000/14:41:55-7:00
    type: 'file
]
```

Pour afficher des informations concernant tous les fichiers du répertoire courant, utilisez :

```
foreach file read %. [
```

```
    info: info? file
    print [file info/size info/date info/type]
]
build-guide.r 22334 30-Jun-2000/14:24:43-7:00 file
code/ 11 11-Oct-1999/18:37:04-7:00 directory
data.r 41 30-Jun-2000/14:41:36-7:00 file
file.txt 306 30-Jun-2000/14:41:55-7:00 file
```

La fonction **info?** est utilisable avec beaucoup de protocoles réseau :

```
probe info? ftp://www.rebol.com/file.txt
```

[Retour au sommaire]

9. Répertoires

Il y a plusieurs fonctions prêtes à l'emploi pour lire des répertoires, gérer des sous-répertoires, créer de nouveaux répertoires, renommer et effacer des fichiers.

De plus, il existe les fonctions standards, pour connaître, modifier et lister le répertoire courant.

Pour plus d'informations pour l'accés aux répertoires, voir le chapitre sur les Ports.

9.1 Lire un répertoire

Les répertoires sont lus de la même manière que les fichiers. La fonction **read** renvoie un bloc de noms de fichiers au lieu de données texte ou binaires.

Pour connaître tous les noms de fichiers du répertoire courant, utilisez la ligne suivante de code :

```
read %.
```

L'exemple précédent lit le répertoire entier et renvoie un bloc composé des noms des fichiers.

Pour afficher les noms de tous les fichiers dans un répertoire, utilisez la ligne de code suivante :

```
print read %intro/
CVS/ history.t intro.t overview.t quick.t
```

A l'intérieur du bloc renvoyé, les noms des répertoires sont spécifiés avec un slash final. Pour afficher chaque nom de fichier sur un ligne différente, saisissez :

```
foreach file read %intro/ [print file]
CVS/
history.t
intro.t
overview.t
quick.t
```

Voici une manière facile d'afficher seulement les répertoires qui ont été trouvés :

```
foreach file read %intro/ [
    if #"/" = last file [print file]
]
CVS/
```

Si vous voulez lire un répertoire présent sur le réseau, n'oubliez pas d'inclure le symbole "slash" à la fin de l'URL pour indiquer au protocole que vous faites référence à un répertoire :

```
print read ftp://ftp.rebol.com/
```

9.2 Créer un répertoire

La fonction **make-dir** permet de créer un nouveau répertoire.

Le nom du nouveau répertoire doit être relatif au répertoire courant ou à un chemin absolu.

```
make-dir %new-dir
make-dir %local-dir/
make-dir %/work/docs/old-docs/
```

Le slash final est optionnel pour cette fonction. En interne, la fonction **make-dir** appelle la fonction **open** avec le raffinement **/new**.

La ligne :

```
close open/new %local-dir/
```

crée également un nouveau répertoire. Le slash final est par contre important dans cet exemple, car il indique qu'un répertoire doit être créé plutôt qu'un fichier.

Si vous utilisez la fonction **make-dir** pour créer un répertoire déjà existant, une erreur sera générée. L'erreur peut être capturée avec la fonction **try**. L'existence du répertoire doit être contrôlée auparavant avec la fonction **exists?**.

9.3 Renommage des répertoires et des fichiers

Pour renommer un fichier, utilisez la fonction **rename** :

```
rename %old-file %new-file
```

L'ancien nom de fichier doit inclure le chemin d'accés complet au fichier, mais ceci n'est pas nécessaire pour le nouveau nom de fichier. En effet, la fonction **rename** n'est pas destinée à

déplacer des fichiers entre différents répertoires.(Beaucoup de systèmes d'exploitation ne permettent pas cette fonctionnalité.)

```
rename %../docs/intro.txt %conclusion.txt
```

Si l'ancien nom de fichier est un répertoire (indiqué par un slash final), la fonction **rename** renommera le répertoire :

```
rename %../docs/ %manual/
```

Si le fichier ne peut être renommé, une erreur se produira. L'erreur peut être capturée avec la fonction **try**.

9.4 Effacer des répertoires et des fichiers

Les fichiers peuvent être effacés avec la fonction **delete** :

```
delete %file
```

Le fichier à supprimer doit avoir un chemin d'accès complet :

```
delete %source/docs/file.txt
```

Un bloc de plusieurs fichiers au sein d'un même répertoire peut aussi être supprimé en une fois :

```
delete [%file1 %file2 %file3]
```

Un ensemble de fichiers peut être supprimé en utilisant un caractère joker et le raffinement **/any** :

```
delete/any %file*
delete/any %secret.?
```

Le caractère joker "astérisque" **(*)** est équivalent à "tous les caractères", et le caractère joker "point d'interrogation" **(?)** équivaut à remplacer un unique caractère.

Pour supprimer un répertoire, mettez à son nom le slash final :

```
delete %dir/
delete %../docs/old/
```

Si le fichier ne peut être supprimé, une erreur sera générée. Il est possible de capturer cette erreur avec la fonction **try**.

9.5 Répertoire courant

Utilisez la fonction **what-dir** pour déterminer le répertoire courant :

```
print what-dir
/work/REBOL/
```

La fonction **what-dir** fait référence au répertoire courant relatif au script en oeuvre comme indiqué dans la variable **system/script/path**.

9.6 Modifier le répertoire courant

Pour modifier le répertoire courant, utilisez la fonction **change-dir** :

```
change-dir %new-path/to-dir/
```

Si le slash final n'est pas inclus, la fonction l'ajoute.

9.7 Listing du Répertoire courant

Pour lister le contenu du répertoire courant, utilisez :

```
list-dir
```

Le nombre de colonnes utilisées pour afficher le contenu du répertoire dépend de la taille de fenêtre de la console, et de la longueur maximale des noms de fichiers.

Chapitre 13 - Protocoles Réseau

Ce document est la traduction française du Chapitre 13 du User Guide de REBOL/Core, qui concerne les protocoles Réseau.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
5 juin 2005 21:02	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Présentation

REBOL inclut en standard plusieurs des plus importants protocoles Internet.

Ces protocoles sont faciles à utiliser au sein de vos scripts; ils ne requièrent aucune librairie ou fichier à inclure, et la plupart des opérations peuvent être réalisées en une seule ligne de code.

La liste ci-dessous indique les protocoles réseau supportés :

DNS Domain Name Service : traduit les noms d'ordinateur en adresses IP et vice-versa.

Finger Permet d'obtenir des informations sur des utilisateurs, par leurs profils.

Whois Récupére des informations sur l'enregistrement d'un domaine (domain registration).

Daytime Network Time Protocol : permet d'obtenir l'heure depuis un serveur.

HTTP	Hypertext Transfer Protocol. Utilisé pour le Web.
SMTP	Simple Mail Transfer Protocol. Utilisé pour l'envoi d'e-mails.
POP	Post Office Protocol. Utilisé pour récupérer des mails.
FTP	File Transfer Protocol. Transfert de fichiers avec un serveur.
NNTP	Network News Transfer Protocol. Pour émettre ou lire des news Usenet.
TCP	Transmission Control Protocol. Protocole de base d'Internet.
UDP	User Datagram Protocol. Protocole non orienté connexion basé sur l'envoi de datagrammes.

De plus, vous pouvez créer des agents pour d'autres protocoles Internet ou créer votre propre protocole.

[Retour au sommaire]

3. Bases Réseau pour REBOL

3.1 Modes de fonctionnement

Il existe de modes de base pour les opérations réseau : atomique ou basé sur un port.

Les opérations réseau en mode "atomique" sont celles qui sont accomplies avec une unique fonction. Par exemple, vous pouvez lire une page Web entière avec un seul appel à la fonction **read**. Il n'est pas nécessaire de séparer l'ouverture de la connexion et la lecture. Tout cela est fait automatiquement avec la fonction **read**.

Par exemple, vous pouvez saisir :

```
print read http://www.rebol.com
```

Le serveur cible est trouvé et ouvert, a page Web est transférée, et la connexion est fermée.

Les opérations réseau basées sur les ports sont celles qui s'appuient sur une approche traditionnelle en programmation. Elles supposent l'ouverture d'un port, et effectuent diverses opérations sur le port. Par exemple, si vous voulez lire votre courrier électronique depuis un serveur POP, message par message, vous devrez utiliser cette méthode. Voici un exemple qui lit et affiche tous vos emails.

```
pop: open pop://user:pass@mail.example.com
forall pop [print first pop]
close pop
```

L'approche atomique est plus facile, mais aussi plus limitée. L'approche basée sur la gestion des ports autorise plus d'opérations, mais suppose aussi une plus grande compréhension des aspects réseau.

3.2 Spécification de ressources réseaux

REBOL fournit deux approche pour spécifier des ressources réseau : les spécifications d'URLs et de ports.

Les URLs (Uniform Resource Locators) sont utilisées pour identifier une ressource réseau, comme une page Web, un site FTP, une adresse email, un fichier, une autre ressource ou un service. Les URLs

sont un type de données intrinsèque à REBOL, et elles peuvent être exprimées directement dans le langage.

La notation standard pour les URLs consiste à écrire le "scheme" (NdT : souvent le protocole), suivi de sa spécification :

```
scheme:specification
```

Le scheme est souvent le nom du protocole, tel que HTTP, FTP, SMTP, et POP; d'autre part, ce n'est pas une nécessité. Un "scheme" peut être n'importe quel nom qui identifie la méthode utilisée pour accéder à une ressource.

Le format de la spécification relative à un "scheme" dépend de celui-ci; cependant, la plupart des schemes partagent un format commun pour identifier les serveurs réseau, les noms d'utilisateur, les mots de passe, les numéros de ports, et les chemins vers les fichiers. Voici quelques formats couramment utilisés :

```
scheme://host

scheme://host:port

scheme://user@host

scheme://user:pass@host

scheme://user:pass@host:port

scheme://host/path

scheme://host:port/path

scheme://user@host/path

scheme://user:pass@host/path

scheme://user:pass@host:port/path
```

Voici la liste des champs utilisés dans les formats précédents (Network Resource Specification).

scheme	Le nom utilisé pour identifier le type de ressource, souvent le même que le protocole. Par exemple, HTTP, FTP, et POP.
host	Le nom réseau ou l'adresse pour une machine. Par exemple, www.rebol.com, cnn.com, accounting.
port	Le numéro de port de la machine cible, pour le scheme en cours. Normalement, les ports sont standardisés, donc cette information n'est pas requise la plupart du temps. Exemples : 21, 23, 80, 8000.
user	un nom d'utilisateur pour accéder à la ressource.
pass	Un mot de passe pour authentifier le nom d'utilisateur.
path	Un chemin de fichier ou une autre méthode pour référencer la ressource. Cette valeur dépend du scheme utilisé. Certains schemes incluent des modèles et des arguments de scripts (comme avec CGI).

Une autre manière d'identifier une ressource est une spécification de port en REBOL. En fait, lorsqu'une URL est utilisée, elle est automatiquement convertie en spécification de port. Une spécification de port

peut accepter beaucoup plus d'arguments qu'une URL, mais nécessite plusieurs lignes pour les définir.

Une spécification de port est une définition d'objet sous forme de bloc, qui fournit chacun des paramètres nécessaires pour accéder à la ressource réseau. Par exemple, l'URL pour accéder à un site Web est :

```
read http://www.rebol.com/developer.html
```

mais elle peut aussi être écrite sous la forme :

```
read [
    scheme: 'HTTP
    host: "www.rebol.com"
    target: %/developer.html
]
```

L'URL pour une ressource FTP à lire peut être :

```
read ftp://bill:vbs@ftp.example.com:8000/file.txt
```

mais elle peut aussi être écrite sous la forme :

```
read [
    scheme: 'FTP
    host: "ftp.example.com"
    port-id: 8000
    target: %/file.txt
    user: "bill"
    pass: "vbs"
]
```

De plus, il y a beaucoup d'autres champs pour le port qui peuvent être mentionnés, comme la durée pour un time-out, le type d'accès, et la sécurité.

3.3 Schemes, Agents (Handlers) et Protocoles

Le fonctionnement de REBOL pour le réseau exploite les schemes pour identifier les agents (handlers) qui communique avec les protocoles.

En REBOL, le scheme est utilisé pour identifier la méthode d'accès à une ressource. Cette méthode utilise un objet codé qui est appelé un agent. Chacun des schemes, supportés par REBOL, pour une URL (comme HTTP, FTP) a un agent. La liste des schemes peut être obtenue avec :

```
probe next first system/schemes
[default Finger Whois Daytime SMTP POP HTTP FTP NNTP]
```

De surcroît, il existe de schemes de bas niveau qui ne sont pas indiqué ici. Par exemple, les schemes TCP et UDP sont utilisés pour les communications directes et de bas niveau.

De nouveaux schemes peuvent être ajoutés à cette liste. Par exemple, vous pouvez définir votre propre scheme, appelé FTP2, qui utilisera des caractéristiques spéciales pour l'accès FTP, comme fournir automatiquement votre nom d'utilisateur et votre mot de passe, de sorte que vous n'ayez pas à les inclure dans les URLs FTP.

La plupart des agents (handlers) sont utilisés pour fournir une interface à un protocole réseau. Un protocole est utilisé pour communiquer entre des périphériques divers, comme des clients et des serveurs.

Bien que chaque protocole soit légèrement différent dans sa façon de communiquer, il peut y avoir des choses communes avec d'autres protocoles.

Par exemple, la plupart des protocoles requièrent une connexion réseau à ouvrir, à lire, à écrire et à fermer. Ces opérations communes sont accomplies en REBOL par un agent par défaut.

Cet agent rend des protocoles comme finger, whois, et daytime presque triviaux à implémenter.

Les agents pour les schemes sont écrits sous forme d'objets. L'agent par défaut sert d'objet racine (root object) pour tous les autres agents. Quand un agent nécessite un champ particulier, comme une valeur de time-out à utiliser pour lire les données, si la valeur n'est pas définie spécifiquement dans l'agent, elle sera fournie par l'agent par défaut.

Donc, les agents surchargent l'agent par défaut, avec leurs champs et leurs valeurs. Vous pouvez aussi créer des agents qui utilisent les valeurs par défaut d'autres agents. Par exemple, vous pouvez utiliser un agent FTP2 qui prend ses champs manquants d'abord dans l'agent FTP, puis ensuite dans l'agent par défaut.

Lorsqu'un port est utilisé pour accéder à des ressources réseau, il est lié à un agent spécifique. Ensemble, l'agent et le port forme une unité qui est utilisée pour fournir l'information sur les données, le code, et le statut permettant de traiter tous les protocoles.

Le code source pour les agents peut être obtenu à partir de l'objet **system/scheme**. Ceci peut être utile si vous voulez modifiez le comportement d'un agent ou construire le votre. Par exemple, pour visualiser le code de l'agent Whois, saisissez :

```
probe get in system/schemes 'whois
```

Notez que ce que verrez est un mélange de l'agent par défaut avec l'agent whois. Le code source actuel qui est utilisé pour créer l'agent Whois fait seulement quelques lignes :

```
make Root-Protocol [
    open-check:  [[any [port/user ""]] none]
    net-utils/net-install Whois make self [] 43
]
```

3.4 Surveillance d'agents

A des fins de déboguage, vous pouvez surveiller les actions de chaque agent. Chaque agent produit sa propre trace pour le déboguage, qui indique quelles opérations ont été réalisées. Pour mettre en route le déboguage réseau, activez-le avec la ligne suivante :

```
trace/net on
```

Pour désactiver le déboguage réseau, utilisez :

```
trace/net off
```

Voici un exemple :

```
read pop://carl:poof@zen.example.com
URL Parse: carl poof zen.example.com none none none
Net-log: ["Opening tcp for" POP]
connecting to: zen.example.com
Net-log: [none "+OK"]
Net-log: {+OK QPOP (version 2.53) at zen.example.com starting.}
Net-log: [["USER" port/user] "+OK"]
Net-log: "+OK Password required for carl."
Net-log: [["PASS" port/pass] "+OK"]
** User Error: Server error: tcp -ERR Password supplied for "carl"
is incorrect.
** Where: read pop://carl:poof@zen.example.com
```

[Retour au sommaire]

4. Démarrage initial

Les fonctionnalités réseau REBOL sont intégrées. Pour créer des scripts qui utilisent des protocoles réseau, vous n'avez pas besoin d'inclure des fichiers spéciaux ou des librairies. Le seul pré-requis est de fournir l'information minimale nécessaire pour activer les protocoles pour atteindre les serveurs ou passer les pare-feux ou les proxys. Par exemple, pour envoyer un e-mail, le protocole SMTP nécessite un nom de serveur SMTP et une adresse e-mail de réponse.

4.1 Paramétrages de base pour le réseau

Quand vous utilisez REBOL pour la première fois, il vous sera demandé d'indiquer les paramétrages nécessaires au réseau, lesquels seront stockés dans le fichier user.r.

REBOL utilise ce fichier pour charger les paramétrages réseau nécessaires à chacun de ses démarrages. Si un fichier user.r n'est pas créé, et que REBOL ne peut trouver un fichier user.r existant dans son environnement, aucun paramétrage ne sera chargé. Voir le chapitre sur les Opérations pour plus d'informations.

Pour modifier les paramétrages réseau, saisissez **set-user** à l'invite de commande. Ceci relance le même script de configuration pour le réseau exécuté lorsque REBOL démarre pour la première fois. Ce script est chargé à partir du fichier rebol.r. Si ce fichier ne peut être trouvé, ou si vous voulez éditer le paramétrage directement, vous pouvez utiliser un éditeur de texte pour modifier le fichier user.r.

Au sein du fichier user.r, les paramétrages réseau se trouvent dans un bloc qui suit la fonction set-net. Au minimum, le bloc devrait contenir deux items :

- Votre adresse email à utiliser dans les champs "From" et "Reply" d'un email et pour un login anonyme.
- Votre serveur par défaut; ce peut être également votre serveur de mail primaire.

De plus, vous pouvez indiquez un certain nombre d'autres items :

- Un serveur différent pour le mail entrant (pour POP).
- Un serveur proxy (pour se connecter au réseau).
- Un numéro de port pour le proxy.
- Le type de proxy (voir les paramétrages du Proxy ci-dessous)

Vous pouvez aussi ajouter des lignes après la fonction set-net pour configurer d'autres genres de protocoles. Par exemple, vous pouvez définir les valeurs de time-out pour les protocoles, définir un mode FTP passif, un identifier "user-agent" pour le HTTP, des proxys séparés pour des protocoles différents, et plus.

Un exemple de bloc **set-net** est :

```
set-net [user@domain.dom mail.server.dom]
```

Le premier champ spécifie votre adresse email, et le second champ indique votre serveur de mail par défaut (remarquez qu'il n'est pas nécessaire de mettre des guillemets ici). Pour la plupart des réseaux, c'est suffisant et aucun autre paramétrage n'est nécessaire (à moins que vous n'utilisiez un serveur proxy). Votre serveur par défaut sera également utilisé si aucun autre serveur spécifique n'est mentionné.

De plus, si vous utilisez un serveur POP (pour les courriers entrants) différent de votre serveur SMTP (courrier sortant), vous pouvez le spécifier aussi

```
set-net [
    user@domain.dom
    mail.server.dom
    pop.server.dom
]
```

Toutefois, si les serveurs POP et SMTP sont les mêmes, ceci n'est pas nécessaire.

4.2 Paramétrages du Proxy

Si vous utilisez un proxy ou un pare-feu (firewall), vous pouvez fournir à la fonction **set-net** les paramétrages du proxy. Ceci comprend le nom ou l'adresse du serveur proxy, un numéro de port pour accéder au serveur, et en option, le type de proxy.

Par exemple :

```
set-net [
    email@addr
    mail.example.com
    pop.example.com
    proxy.example.com
    1080
    socks
]
```

Cet exemple utilisera un serveur de proxy appelé proxy.example.com sur son port TCP 1080 avec la méthode "socks" pour le proxy. Pour utiliser un serveur socks4, utilisez le mot "socks4" au lieu de socks. Pour utilisez le serveur générique CERN, utilisez le mot "generic".

Vous pouvez aussi définir un serveur de proxy spécifique pour un scheme (protocole).

Chaque protocole possède son propre objet proxy que vous pouvez adapter spécifiquement selon le scheme. Voici un exemple de paramétrages de proxy pour FTP :

```
system/schemes/ftp/proxy/host: "proxy2.example.com"

system/schemes/ftp/proxy/port-id: 1080

system/schemes/ftp/proxy/type: 'socks
```

Dans ce cas, seul le protocole FTP utilise un serveur de proxy spécial. Notez que chaque nom de machine doit être une chaîne et que le type de proxy doit être un mot littéral.

Voici deux exemples supplémentaires. Le premier exemple définit un proxy de type générique (CERN) pour le HTTP :

```
system/schemes/http/proxy/host: "wp.example.com"

system/schemes/http/proxy/port-id: 8080

system/schemes/http/proxy/type: 'generic
```

Dans l'exemple ci-dessus, toutes les requêtes HTTP passent à travers un proxy de type générique sur l'adresse wp.example.com en utilisant le port 8080.

```
system/schemes/smtp/proxy/host: false

system/schemes/smtp/proxy/port-id: false

system/schemes/smtp/proxy/type: false
```

Dans l'exemple ci-dessus, l'intégralité du courrier sortant ne passe pas par un serveur de proxy. La valeur **false** empêche que le serveur de proxy par défaut soit utilisé. Si vous mettiez ces champs à none, alors le serveur de proxy par défaut sera utilisé, s'il a été configuré.

Si vous voulez contourner (bypasser) les paramétrages de proxy pour des machines particulières, comme celles situées sur votre réseau local, vous pouvez fournir une liste de machines autorisées (bypass list).

Voici une liste pour le serveur de proxy par défaut :

```
system/schemes/default/proxy/bypass:
    ["host.example.net" "*.example.com"]
```

Notez que l'astérisque **(*)** et le point d'interrogation **(?)** peuvent être utilisés pour des correspondances de machines. L'astérisque **(*)** est utilisé dans l'exemple précédent pour autoriser toutes les machines dont le nom se termine par *example.com*.

Pour définir une liste de machines autorisées seulement pour le scheme HTTP, utilisez :

```

```

```
system/schemes/http/proxy/bypass:
    ["host.example.net" "*.example.com"]
```

4.3 Autres paramétrages

En supplément des paramètres du proxy, vous pouvez définir des valeurs de time-out pour tous les schemes (par défaut) ou pour des schemes spécifiques. Par exemple, pour augmenter la valeur du time-out pour tous les schemes, vous pouvez écrire :

```
system/schemes/default/timeout: 0:05
```

Ceci définit un time-out réseau de 5 minutes. Si vous voulez juste augmenter le time-out pour le scheme SMTP, vous pouvez écrire :

```
system/schemes/smtp/timeout: 0:10
```

Certains schemes possèdent des champs personnalisés. Par exemple, le scheme FTP vous permet de définir un mode passif pour tous les transferts :

```
system/schemes/ftp/passive: on
```

Le mode FTP passif est pratique car les serveurs FTP configurés ainsi n'essayent pas de se connecter en retour au travers de votre pare-feu.

Lorsque vous essayer d'accéder à des sites Web, vous pouvez vouloir utiliser un champ "user-agent" différent dans la requête HTTP, afin d'obtenir de meilleurs résultats sur les quelques sites qui détectent le type de navigateur :

system/schemes/http/user-agent: "Mozilla/4.0"

4.4 Accéder aux paramétrages

Chaque fois que REBOL démarre, il lit le fichier user.r pour trouver ses paramétrages réseau. Ces paramétrages sont réalisés avec la fonction **set-net**.

Les scripts peuvent accéder à ces paramètres au travers de l'objet system/schemes.

```
system/user/email ; used for email from and reply
system/schemes/default/host - your primary server
system/schemes/pop/host - your POP server
system/schemes/default/proxy/host - proxy server
system/schemes/default/proxy/port-id - proxy port
system/schemes/default/proxy/type - proxy type
```

Ci-dessous se trouve une fonction qui renvoie un bloc contenant le paramétrage réseau dans le même ordre que la fonction **set-net** les acceptent :

```
get-net: func [][
```

```
    reduce [
    system/user/email
    system/schemes/default/host
    system/schemes/pop/host
    system/schemes/default/proxy/host
    system/schemes/default/proxy/port-id
    system/schemes/default/proxy/type
    ]
]

probe get-net
```

[Retour au sommaire]

5. DNS - Domain Name Service

DNS est le service réseau qui traduit les noms de domaine en leur adresse IP. De surcroît, vous pouvez utiliser DNS pour trouver une machine et son nom de domaine à partir d'une adresse IP.

Le protocole DNS peut être utilisé de trois manières : vous pouvez rechercher l'adresse IP primitive d'un nom de machine, ou un nom de domaine pour une adresse IP, et vous pouvez trouver le nom et l'adresse IP de votre machine locale.

Pour retrouver l'adresse IP d'une machine spécifique dans un domaine spécifique, saisissez :

```
print read dns://www.rebol.com
207.69.132.8
```

Vous pouvez aussi obtenir le nom de domaine qui est associé avec une adresse IP particulière :

```
print read dns://207.69.132.8
rebol.com
```

Notez qu'il n'est pas incongru pour cette recherche en reverse DNS de retourner le résultat **none**. Il existe des machines qui n'ont pas de noms.

```
print read dns://11.22.33.44
none
```

Pour déterminer le nom de votre système, essayez la lecture DNS d'une URL vide de la forme :

```
print read dns://
crackerjack
```

Les données renvoyées ici dépendent du type de la machine. Cela peut être un nom de machine sans domaine, comme indiqué précédemment, mais être aussi un nom de machine complet, comme crackerjack.example.com. Ceci dépend du système d'exploitation et de la configuration réseau du système.

Voici un exemple qui recherche et affiche les adresses IP pour un certain nombre de sites Web :

```
domains: [
    www.rebol.com
    www.rebol.org
    www.mochinet.com
    www.sirius.com
]

foreach domain domains [
    print ["address for" domain "is:"
    read join dns:// domain]
]
address for www.rebol.com is: 207.69.132.8
address for www.rebol.org is: 207.66.107.61
address for www.mochinet.com is: 216.127.92.70
address for www.sirius.com is: 205.134.224.1
```

[Retour au sommaire]

6. Whois

le protocole whois renvoie des informations concernant des noms de domaines depuis un référentiel central. Le service Whois est fourni par les organisations qui font fonctionner Internet. Whois est souvent utilisé pour retrouver les informations d'enregistrement d'un domaine Internet ou d'un serveur.

Il peut vous dire qui est propriétaire du domaine, comment leur contact technique peut être joint, et d'autres informations.

Pour obtenir ces informations, utilisez la fonction **read** avec une URL Whois.

Cette URL doit contenir le nom de domaine et le nom du serveur Whois séparés par un signe **(@)**.

Par exemple, pour obtenir des informations concernant *example.com* depuis le référentiel Internet :

```
print read whois://example.com@rs.internic.net
connecting to: rs.internic.net
Whois Server Version 1.1
Domain names in the .com, .net, and .org domains can now be
registered with many different competing registrars. Go to
http://www.internic.net for detailed information.
Domain Name: EXAMPLE.COM
Registrar: NETWORK SOLUTIONS, INC.
Whois Server: whois.networksolutions.com
Referral URL: www.networksolutions.com
Name Server: NS.ISI.EDU
Name Server: VENERA.ISI.EDU
Updated Date: 17-aug-1999
<<< Last update of whois database: Sun, 16 Jul 00 03:16:34 EDT >>>
The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains
and Registrars.
```

Le code précédent est seulement un exemple. Le détail de l'information renvoyée, et les serveurs qui supportent Whois changent de temps en temps.

Si au lieu d'un nom de domaine, vous fournissez un mot, toutes les entrées qui correspondent à ce mot seront renvoyées :

```
print read whois://example@rs.internic.net
connecting to: rs.internic.net
Whois Server Version 1.1
Domain names in the .com, .net, and .org domains can now be
registered with many different competing registrars. Go to
http://www.internic.net for detailed information.
EXAMPLE.512BIT.ORG
EXAMPLE.ORG
EXAMPLE.NET
EXAMPLE.EDU
EXAMPLE.COM
To single out one record, look it up with "xxx", where xxx is one
of the of the records displayed above. If the records are the same, look them
up with "=xxx" to receive a full display for each record.
<<< Last update of whois database: Sun, 16 Jul 00 03:16:34 EDT >>>
The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains
and Registrars.
```

Le protocole Whois n'accepte pas les URLs, comme www.example.com, à moins que l'URL fasse partie du nom de la société enregistrée.

[Retour au sommaire]

7. Finger

Le protocole finger retrouve des informations spécifiques à un utilisateur stockées dans le fichier log de l'utilisateur.

Pour pouvoir demander des informations sur un utilisateur à un serveur, celui-ci doit exécuter ce protocole finger. L'information est demandée en appelant avec **read** une URL finger qui comprend le nom de l'utilisateur et un nom de domaine, et se présente au format email :

```
print read finger://username@example.com
```

L'exemple précédent renvoie les informations concernant l'utilisateur référencé par username@example.com. L'information retournée dépend de celle que l'utilisateur a fourni et des paramétrages du serveur finger. Egalement, les détails de l'information retournée sont propres à chaque serveur; les exemples ci-dessous décrivent seulement des serveurs génériques. La plupart des serveurs peuvent avoir des comportements non standards sur les requêtes finger.

Par exemple, l'information suivante pourrait être retournée :

```
Login: username
Name: Firstname Lastname
Directory: /home/user
Shell: /usr/local/bin/tcsh
Office: City, State +1 555 555 5555
Last login Wed Jul 28 01:10 (PDT) on ttyp0 from some.example.com
No Mail.
No Plan.
```

Remarquez que finger informe de la dernière connexion de l'utilisateur sur la machine, et aussi s'il y a des courriers électroniques en attente pour lui. Si l'utilisateur lit un email à partir de son compte, parfois finger fournit cette information : lorsque l'email a été reçu et la dernière fois que l'utilisateur a récupéré son email :

```
New mail received Sun Sep 26 11:39 1999 (PDT)
Unread since Tue Sep 21 04:45 1999 (PDT)
```

Le serveur finger peut aussi renvoyer le contenu d'un fichier plan ou un fichier projet s'ils existent. Les utilisateurs peuvent inclure n'importe quelle information qu'ils souhaitent dans un fichier plan ou projet.

Il est aussi possible de retrouver des informations sur les utilisateurs en utilisant leur nom ou leur prénom. Les serveurs finger ont besoin que vous mettiez en majuscules

Certains serveurs finger demandent que soient écrits en majuscule les noms tels qu'ils apparaissent dans le fichier de login ou dans le fichier en ligne utilisé par le serveur finger, pour retrouver les informations sur l'utilisateur. D'autres serveurs finger sont plus tolérants vis-à-vis de la mise en majuscules.

Un serveur finger répondra aux requêtes sur le vrai nom en renvoyant toutes les listes qui correspondent aux critères de recherches. Par exemple, si il y a plusieurs utilisateurs qui possèdent le même prénom zaphod, si vous saisissez la requête :

```
print read finger://Zaphod@main.example.com
```

celle-ci renverra tous les utilisateurs ayant pour prénom ou nom zaphod. Certains serveurs finger renvoient un listing d'utilisateurs quand le nom de l'utilisateur est omis. Par exemple, le code :

```
print read finger://main.example.com
```

retournera une liste de tous les utilisateurs connectés sur la machine, si le service finger installé sur celle-ci l'autorise.

Certaines machines limitent le service finger pour des raisons de sécurité. Elles peuvent demander un nom d'utilisateur valide, et renvoyer seulement les informations liées à cet utilisateur. Si vous interrogez un serveur finger de ce genre, sans fournir des informations sur l'utilisateur, le serveur vous répondra qu'il attend des informations sur un utilisateur spécifique.

Si un serveur ne supporte pas le protocole finger, REBOL retourne une erreur d'accès :

```
print read finger://host.dom
connecting to: host.dom
Access Error: Cannot connect to host.dom.
Where: print read finger://host.dom
```

[Retour au sommaire]

8. Daytime - Network Time Protocol

Le protocole daytime retourne le jour et l'heure courante. Pour se connecter à une serveur daytime, utiliser **read** avec une URL daytime. Cette URL comprend le nom du serveur devant renvoyer la date :

```
print read daytime://everest.cclabs.missouri.edu
Fri Jun 30 16:40:46 2000
```

Le format de l'information renvoyée par les serveurs peut varier, selon le serveur. Notez que l'indication du fuseau horaire peut ne pas être présente.

Si le serveur que vous interrogez ne supporte pas le protocole daytime, REBOL renvoie une erreur :

```
print read daytime://www.example.com
connecting to: www.example.com
** Access Error: Cannot connect to www.example.com.
** Where: print read daytime://www.example.com
```

[Retour au sommaire]

9. HTTP - Hyper Text Transfer Protocol

Le Web (World Wide Web - WWW) est caractérisé par deux technologies fondamentales : HTTP, et HTML. HTTP est l'acronyme de "Hyper Text Transfer Protocol", le protocole qui contrôle comment des serveurs Web et des navigateurs Web communiquent les uns avec les autres. HTML signifie "Hyper Text Markup Language" qui définit la structure et le contenu d'une page Web.

Pour récupérer une page Web, le navigateur envoie sa requête à un serveur Web utilisant HTTP. A la réception de la requête, le serveur l'interprète, parfois en utilisant des scripts CGI (voir CGI - Common Gateway Interface), et renvoie des données. Ces données peuvent être n'importe quoi, dont du HTML, du texte, des images, des programmes ou du son.

9.1 Lecture d'une page Web

Pour lire une page Web, utilisez la fonction **read** avec une URL HTTP.

Par exemple :

```
page: read http://www.rebol.com
```

Ceci retourne une page Web pour www.rebol.com. Notez qu'une chaîne de caractères qui contient le code HTML (NdT : c'est-à-dire le code source de la page HTML), est renvoyée par la commande **read**. Aucune image ou graphique, ni aucune information n'est ramenée. Pour cela, il est nécessaire d'effectuer des opérations de lectures en complément. La page Web peut être affichée sous la forme de code HTML en utilisant **print**, elle peut être écrite dans un fichier avec la commande **write**, ou être envoyée dans un email avec la commande **send**.

```
print page

write %index.html page

send zaphod@example.com page
```

La page peut être manipulée de bien des façons, en utilisant diverses fonctions REBOL, comme **parse**, **find**, et **load**.

Par exemple, pour chercher au sein d'une page Web toutes les occurrences du mot REBOL, vous pouvez écrire :

```
parse read http://www.rebol.com [
    any [to "REBOL" copy line to newline (print line)]
]
```

9.2 Scripts sur des sites Web

Un serveur Web peut manipuler plus que des scripts HTML. Les serveurs Web sont tout à fait pratiques pour fournir également des scripts REBOL.

Vous pouvez charger des scripts REBOL directement depuis un serveur Web avec la commande **load** :

```
data: load http://www.rebol.com/data.r
```

Vous pouvez aussi évaluer des scripts directement depuis un serveur Web avec la commande **do** :

```
data: do http://www.rebol.com/code.r
```

> Avertissement :
> Soyez prudent avec cet usage de **do**. Evaluer sans précaution des scripts sur des serveurs Internet ouverts peut provoquer des dégâts. Evaluez un script uniquement si vous êtes certain de la fiabilité de sa source, si vous avez contrôlé sa source ou que vous avez conservé vos paramétrages de sécurité REBOL à leur plus haut niveau.

De plus, les pages Web qui contiennent du HTML peuvent aussi contenir des scripts REBOL insérés dedans, et peuvent être exécuter avec :

```
data: do http://www.rebol.com/example.html
```

Pour savoir si un script existe dans une page avant de l'évaluer, utilisez la fonction **script?** .

```
if page: script? http://www.rebol.com [do page]
```

La fonction **script?** lit la page depuis le site Web et renvoie celle-ci à partir de la position de l'en-tête REBOL.

9.3 Chargement de pages avec balises

Les pages HTML et XML peuvent être rapidement converties en bloc REBOL avec la fonction **load/markup**. Cette fonction renvoie un bloc constitué de toutes les balises et les chaînes de caractères trouvées dans la page. Tous les espaces et les sauts de ligne sont conservés.

Pour filtrer une page en ôtant toutes les balises Web et juste imprimer le texte, saisissez :

```
tag-text: load/markup http://www.rebol.com
text: make string! 2000

foreach item tag-text [
    if string? item [append text item]
]

print text
```

Vous pouvez alors effectuer une recherche dans ce texte avec des modèles. Il contient tous les espaces et les sauts de ligne du fichier HTML original.

Voici un autre exemple qui contrôle tous les liens trouvés dans une page Web pour s'assurer de l'existence des pages à laquelles ces liens font référence :

```
REBOL []

page: http://www.rebol.com/developer.html
set [path target] split-path page
system/options/quiet: true      ; turn off connexion msgs
tag-text: load/markup page
links: make block! 100

foreach tag tag-text [  ; find all anchor href tags
    if tag? tag [
    if parse tag [
        "A" thru "HREF="
        [{"} copy link to {"} | copy link to ">"]
        to end
    ][
        append links link
    ]
    ]
]

print links

foreach link unique links [    ; try each link
    if all [
    link/1 <> #"#"
    any [flag: not find link ":"
         find/match link "http:"]
    ][
    link: either flag [path/:link][to-url link]
    prin [link "... "]
    print either error? try [read link]
        ["failed"]["OK"]
    ]
]
```

9.4 Autres Fonctions

Pour vérifier si une page Web existe, utilisez la fonction **exists?**, laquelle renvoie **true** si la page existe.

```
if exists? http://www.rebol.com [
    print "page still there"
]
```

> Note :
>
> Habituellement, il est plus rapide dans la plupart des cas de juste lire la page, plutôt que de vérifier d'abord si elle existe. Par ailleurs, le script appelle deux fois le serveur et ceci peut être assez consommateur de temps.

Pour connaître la date de dernière modification d'une page Web, utilisez la fonction **modified?** :

```
print modified? http://www.rebol.com/developer.html
```

Cependant, tous les serveurs Web ne fournissent pas cette information de date de modification. Typiquement, les pages générées dynamiquement ne renvoient pas de date de modification.

Une autre manière de déterminer si une page Web a changé est de l'interroger régulièrement et de la contrôler. Une façon pratique de vérifier si la page Web a changé est d'utiliser la fonction **checksum**.

Si la précédente valeur de checksum calculée diffère de la valeur courante, cela signifie que la page Web a été changée depuis le dernier contrôle. Voici un exemple qui utilise cette technique. Il vérifie une page toutes les huit heures.

```
forever [
    page: read http://www.rebol.com
    page-sum: checksum page
    if any [
    not exists? %page-sum
    page-sum &lt;&gt; (load %page-sum)
    ][
    print ["Page changed" now]
    save %page-sum page-sum
    send luke@rebol.com page
    ]
    wait 8:00
]
```

Lorsque la page est modifiée, elle est envoyée par email à Luke.

9.5 Agir comme un Navigateur

Normalement, REBOL s'identifie lui-même vis-à-vis d'un serveur Web quand il lit une page. Cependant, certains serveurs sont programmés pour répondre uniquement à certains navigateurs. Si une requête à

un serveur ne retourne pas la bonne page Web, vous pouvez modifier la requête pour la rendre identique à une venant d'un autre type de navigateur Web.

S'identifier comme étant un navigateur Web particulier est fait par de nombreux programmes, afin de permettre à des sites Web de répondre correctement. Cependant, cette pratique peut conduire à faire échouer l'utilisation normale après l'identification du navigateur.

Pour changer les requêtes HTTP et leur donner une ressemblance avec celles envoyées par Netscape 4.0, vous pouvez modifiez la valeur du user-agent au sein de l'agent (handler) HTTP :

```
system/options/http/user-agent: "Mozilla/4.0"
```

Modifier cette variable affecte toutes les requêtes HTTP qui suivent.

9.6 Envoi de requêtes CGI

Les requêtes HTTP CGI peuvent être émises de deux manières. Vous pouvez inclure les données de la requête dans l'URL ou bien, vous pouvez fournir les données de la requête au travers d'une opération d'envoi HTTP (POST).

Une requête avec une URL CGI utilise une URL normale. L'exemple ci-dessous envoie au script CGI test.r la valeur 10 pour sa variable *data*.

```
read http://www.example.com/cgi-bin/test.r?data=10
```

L'émission d'une requête CGI avec post nécessite que vous fournissiez les données CGI en tant que partie du raffinement **custom** de la fonction **read**. L'exemple ci-dessous montre comment est émise la requête CGI :

```
read/custom http://www.example.com/cgi-bin/test.r [
    post "data: 10"
]
```

Dans cet exemple, le raffinement ***/custom*** est utilisé pour fournir des informations supplémentaires pour la lecture avec **read**. Le second argument est un bloc qui débute avec le mot *post* et continue avec la chaîne à envoyer.

La méthode "post" est pratique pour envoyer facilement du code REBOL et des données à un serveur Web en mode CGI. L'exemple suivant illustre ceci :

```
data: [sell 10 shares of "ACME" at $123.45]

read/custom http://www.example.com/cgi-bin/test.r reduce [
    `post mold data
]
```

La fonction **mold** produira une chaîne formatée pour REBOL prête à être émise vers le serveur Web.

[Retour au sommaire]

10. SMTP - Simple Mail Transport Protocol

Le protocole SMTP (Simple Mail Transport Protocol) détermine les transferts de messages électroniques via Internet. Le SMTP définit les interactions entre les serveurs Internet qui contribuent à relayer les courriers depuis leur expéditeur jusqu'à leur destinataire.

10.1 Envoi d'Email

Un courrier électronique est envoyé avec le protocole SMTP en utilisant la fonction **send**. Cette fonction peut expédier un courrier électronique vers une ou plusieurs adresses emails.

Pour que la fonction **send** opère correctement, vos paramètres réseau doivent être définis. La fonction **send** nécessite que vous spécifiez une adresse email (champ From d'un email), et votre serveur d'email par défaut. Voir le début de ce chapitre.

La fonction **send** prend deux arguments : une adresse email et un message.

Par exemple :

```
send user@example.com "Hi from REBOL"
```

Le premier argument doit être un email ou un bloc d'adresses emails (block). Le second argument peut être de n'importe quel type de données (datatype).

```
send luke@rebol.com $1000.00

send luke@rebol.com 10:30:40

send luke@rebol.com bill@ms.dom

send luke@rebol.com [Today 9-Apr-99 10:30]
```

Chacun de ces simples messages emails peut être interprété côté receveur (avec REBOL) ou visualisé avec un client normal de messagerie électronique. Vous pouvez envoyer un fichier complet d'abord en le lisant, puis en le passant comme second argument à la fonction **send** :

```
send luke@rebol.com read %task.txt
```

Des données binaires, comme des images ou des programmes exécutables, peuvent aussi être envoyées :

```
send luke@rebol.com read/binary %rebol
```

Les données binaires sont encodées de façon à permettre leur transfert sous forme de texte. Pour expédier un message binaire auto-extractible, vous pouvez écrire :

```
send luke@rebol.com join "REBOL for the job" [
    newline "REBOL []" newline
    "write/binary %rebol decompress "
```

```
    compress read/binary %rebol
]
```

Lorsque le message est réceptionné, le fichier peut être extrait en utilisant la fonction **do**.

10.2 Destinataires multiples

Pour envoyer un message à de multiples destinataires, vous pouvez utiliser un bloc d'adresses emails :

```
send [luke@rebol.com ben@example.com] message
```

Dans ce cas, chaque message est individuellement adressé avec seulement un nom de destinataire apparaissant dans le champ To (identique à l'adressage en copie cachée BCC).

Le bloc d'adresses email peut être de n'importe quelle longueur ou même être un fichier que vous chargez. Il vous faut juste être attentif à avoir des adresses emails valides, et non des chaînes de caractères qui, elles, sont ignorées.

```
friends: [
    bob@cnn.dom
    betty@cnet.dom
    kirby@hooya.dom
    belle@apple.dom
    ...
]
send friends read %newsletter.txt
```

10.3 Courrier en masse

Si vous expédiez du courrier électronique à un groupe important, vous pouvez réduire la charge sur votre serveur en distribuant à chacun dans le groupe un simple message. C'est l'objet du raffinement **/only**. Il utilise une propriété du protocole SMTP pour envoyer seulement un message à des adresses emails multiples. En utilisant la liste "friends" de l'exemple précédent :

```
send/only friends message
```

Les messages ne sont pas adressés individuellement. Vous pouvez avoir vu ce mode dans certains des emails que vous pouvez recevoir. Lorsque vous recevez un courrier en masse, votre adresse n'apparaît pas dans le champ To. Le mode d'envoi en masse du SMTP devrait être utilisé pour les listes de diffusion, et pas pour de l'envoi de Spam. Le Spam est contraire à la Net-étiquette, il est illégal dans de nombreux pays et états, et peut conduire à votre exclusion de votre Fournisseur d'Accès Internet, et d'autres sites.

10.4 Ligne de sujet et en-têtes

Par défaut, la fonction **send** utilise la première ligne de l'argument *message* comme ligne de sujet pour le courrier électronique. Pour fournir une ligne de sujet personnalisée, vous devrez donner un en-tête d'email à la fonction **send**.

En complément du sujet, vous pouvez indiquer une organisation, une date, un champ CC, et même vos propres champs personnalisés.

Pour indiquer un en-tête, utiliser le raffinement **/header** de la fonction **send**, et incluez l'en-tête sous forme d'un objet. L'objet servant d'en-tête doit être composé à partir de l'objet **system/standard/email**. Par exemple :

header: make system/standard/email [

```
Subject: "Seen REBOL yet?"
Organization: "Freedom Fighters"
```

]

Notez que les champs standards comme l'adresse From, ne sont pas requis et sont automatiquement complétés par la fonction send.

L'en-tête est ensuite fourni en tant qu'argument à **send/header** :

```
send/header friends message header
```

Le courrier électronique ci-dessus est émis en utilisant l'en-tête personnalisé pour chacun des messages.

10.5 Déboguer vos scripts

Lors des tests de vos scripts utilisant **send**, il est judicieux de vous expédier à vous même le courrier électronique d'abord, avant de l'expédier à d'autres. Vérifiez et testez scrupuleusement vos scripts pour être sûrs de ce que vous voulez réaliser. Une erreur commune est d'envoyer un nom de fichier plutôt que son contenu. Par exemple, si vous écrivez :

```
send person %the-data-file.txt
```

ceci envoie le nom du fichier, et non son contenu.

[Retour au sommaire]

11. POP - Post Office Protocol

Le protocole POP (Post Office Protocol) vous permet de récupérer le courrier électronique qui attend dans votre boîte aux lettres, sur un serveur de mails. POP définit un certain nombre d'opérations sur la façon d'accéder à votre boîte aux lettres (BAL) et de stocker des emails sur votre serveur.

11.1 Lecture d'Email

Vous pouvez lire tout votre courrier électronique en une seule ligne sans effacer quoique ce soit de votre serveur de courrier. Ceci est réalisé en lisant avec POP une URL comprenant votre nom d'utilisateur (compte de courrier), votre mot de passe, et le serveur de mails.

```
mail: read pop://user:pass@mail.example.com
```

Les courriers sont renvoyés sous la forme d'un bloc de plusieurs chaînes de caractères, que vous pouvez afficher une par une avec un code comme celui-ci :

```
foreach message mail [print message]
```

Pour lire individuellement des emails depuis le serveur, vous aurez besoin d'ouvrir un port de connexion avec le serveur puis de gérer chaque message un par un. Pour ouvrir un port POP :

```
mailbox: open pop://user:pass@mail.example.com
```

Dans cet exemple, "mailbox" est traitée comme une série, et la plupart des fonctions standards propres aux séries sont utilisables comme length?, first, second, third, pick, next, back, head, tail, head?, tail?, remove, et clear.

Pour déterminer le nombre de messages électroniques sur le serveur, utilisez la fonction length?.

```
print length? mailbox
37
```

De plus, vous pouvez extraire la taille totale de tous les messages et leur taille individuellement avec :

```
print mailbox/locals/total-size

print mailbox/locals/sizes
```

NdT : on utilise ici une méthode de l'objet *mailbox* renvoyé par la fonction **open**.

Pour afficher le premier, le second, et le dernier message électronique, vous pouvez écrire :

```
print first mailbox

print second mailbox

print last mailbox
```

Vous pouvez aussi utiliser le fonction **pick** pour rapatrier un message spécifique :

```
print pick mailbox 27
```

Vous pouvez récuperer et afficher chaque message du plus ancien au plus récnent en utilisant une boucle **loop** qui est identique à celle utilisée pour d'autres types de série :

```
```

```
while [not tail? mailbox] [
    print first mailbox
    mailbox: next mailbox
]
```

Vous aussi lire vos courriers électroniques du plus récent au plus ancien avec la boucle suivante :

```
mailbox: tail mailbox

while [not head? mailbox] [
    mailbox: back mailbox
    print first mailbox
]
```

Une fois terminées les opérations sur le port, fermez-le. Ceci est fait avec la ligne suivante :

```
close mailbox
```

11.2 Suppression d'emails

Comme avec les séries, la fonction **remove** peut être appelée pour effacer un seul message, et la fonction **clear** peut être utilisée pour effacer tous les messages depuis la position courante dans la liste, la série, jusqu'à la fin de *mailbox*.

Par exemple, pour lire un message, sauvez-le dans un fichier et effacez-le du serveur.

```
mailbox: open pop://user:pass@mail.example.com
write %mail.txt first mailbox
remove mailbox
close mailbox
```

Le message électronique est effacé du serveur lorsque la fonction **close** est exécutée. Pour effacer le 22ème message du serveur, vous pouvez écrire :

```
user:pass@mail.example.com
remove at mailbox 22
close mailbox
```

Vous pouvez effacer un nombre donné de messages en utilisant le raffinement **/part** avec la fonction **remove** :

```
remove/part mailbox 5
```

Pour effacer tous les messages de votre boîte aux lettres, utilisez la fonction **clear** :

```
mailbox: open pop://user:pass@example.com
```

```
clear mailbox
close mailbox
```

La fonction **clear** peut aussi est utilisée à différentes positions dans la série **mailbox**, de façon à n'effacer que les messages entre ces positions et jusqu'à la fin de la série.

11.3 Manipulation d'en-tête de courrier électronique

Les courriers électroniques peuvent inclure un en-tête. L'en-tête contient des informations sur l'expéditeur, le sujet, la date et d'autres champs.

En REBOL, les en-têtes d'email sont manipulés en tant qu'objets qui contiennent tous les champs nécessaires. Pour transformer un message email en objet, vous pouvez utiliser la fonction **import-email**. Par exemple :

```
msg: import-email first mailbox

print first msg/from  ; the email address
print msg/date
print msg/subject
print msg/content
```

Vous pouvez alors facilement écrire un filtre qui scanne votre courrier électronique pour les messages qui débutent par un sujet particulier :

```
mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    msg: import-email first mailbox
    if find/match msg/subject "[REBOL]" [
    print msg/subject
    ]
    mailbox: next mailbox
]

close mailbox
```

Voici un autre exemple qui vous alerte lorsque un email provenant d'un groupe d'amis est reçu :

```
friends: [orson@rebol.com hans@rebol.com]

messages: read pop://user:pass@example.com

foreach message messages [
    msg: import-email message
    if find friends first msg/from [
    print [msg/from newline msg/content]
    send first msg/from "Got your email!"
    ]
]
```

Ce filtre de spam efface du serveur tous les messages qui ne contiennent pas votre adresse email quelque part dans le message :

```
mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    mailbox: either find first mailbox user@example.com
    [next mailbox][remove mailbox]
]

close mailbox
```

Voici une simple liste email qui reçoit des messages et les envoie à un groupe. Le serveur accepte juste les courriers des personnes du groupe.

```
group: [orson@rebol.com hans@rebol.com]

mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    message: import-email first mailbox
    mailbox: either find group first message/from [
    send/only group first mailbox
    remove mailbox
    ][next mailbox]
]

close mailbox
```

[Retour au sommaire]

12. FTP - File Transfer Protocol

Le protocole FTP (File Transfer Protocol) est extrêmement utilisé sur Internet pour transférer des fichiers depuis et vers une machine distante. Le FTP est couramment utilisé pour télécharger et mettre à jour des pages d'un site Web, et pour avoir en ligne des archives de fichier (sites de téléchargement).

12.1 Utilisation de FTP

Avec REBOL, les opérations relatives au protocole FTP sont effectuées de la même manière que si on avait des fichiers locaux.

Les fonctions telles que **read**, **write**, **load**, **save**, **do**, **open**, **close**, **exists?**, **size?**, **modified?**, et d'autres encore sont utilisables avec FTP.

REBOL fait la distinction entre les fichiers locaux et les fichiers accessibles par FTP, au moyen de l'utilisation d'une URL FTP.

L'accès à des serveurs FTP peut être libre ou contrôlé. Des accès libres permettent à n'importe qui de se connecter au site FTP et de télécharger des archives, des fichiers. Ceci s'appelle un accès anonyme et est fréquemment utilisé pour des sites de téléchargement publics.

Les accès contrôlés nécessitent que vous fournissiez un nom d'utilisateur et un mot de passe pour accéder au site. C'est le principe pour la mise à jour de pages Web sur un site Web.

Bien que le protocole FTP ne requiert pas que votre configuration réseau REBOL soit OK, si vous utilisez un accès anonyme, une adresse email est souvent demandée. Cette adresse est trouvée dans

l'objet **system/user/email**.

Normalement, lorsque vous démarrez REBOL, cette information est définie à partir de votre fichier **user.r**. Voir la section sur le démarrage initial pour plus de détails. Si vous utilisez le protocole FTP au travers d'un pare-feu ou d'un serveur proxy, FTP doit être configuré pour opérer en mode passif. Le mode passif ne nécessite pas des connexions en retour depuis le serveur FTP vers le client, pour des transferts de données. Ce mode crée seulement des connexions sortantes depuis votre machine et permet d'avoir un haut niveau de sécurité. Pour engager le mode passif, vous devez positionner une variable dans l'agent (handler) du protocole FTP.

```
system/schemes/ftp/passive: true
```

Si vous ignorez si ce mode est nécessaire, essayez d'abord sans. Si cela ne fonctionne pas, paramétrez la variable comme ci-dessus.

12.2 URLs FTP

A la base, une URL FTP possède la forme suivante :

```
ftp://user:pass@host/directory/file
```

Pour des accès anonymes, le nom d'utilisateur (user) et le mot de passe (password) peuvent être omis :

```
ftp://host/directory/file
```

La plupart des exemples dans cette section utilise cette forme simple ; cependant, ils marchent aussi avec un nom d'utilisateur et un mot de passe.

Pour atteindre un répertoire distant, terminez l'URL par le symbole "slash" (/), comme avec :

```
ftp://user:pass@host/directory/

ftp://host/directory/

ftp://host/
```

Vous trouverez plus loin plus d'informations sur l'accès à des répertoires distants.

Il est commode de placer l'URL dans une variable et d'utiliser les paths pour fournir des noms de fichiers. Ceci permet de faire référence à l'URL avec juste un mot.

Par exemple :

```
site: ftp://ftp.rebol.com/pub/
read site/readme.txt
```

Cette technique est mise en oeuvre dans les sections qui suivent.

12.3 Transfert de fichiers Texte

Le protocole FTP établit une distinction entre les fichiers texte et les fichiers binaires. Lors du transfert de fichiers texte, FTP convertit les caractères de fin de ligne. Cela n'est pas souhaitable pour les fichiers binaires.

Pour lire un fichier texte, passez à la fonction **read** une URL FTP :

```
file: read ftp://ftp.site.com/file.r
```

Ceci met le contenu du fichier dans une chaîne (ici, *file*). Pour écrire ce fichier localement, utilisez cette ligne :

```
write %file.r read ftp://ftp.site.com/file.r
```

La plupart des raffinements de la fonction **read** sont également utilisables. Par exemple, vous pouvez utilisez **read/lines** avec :

```
data: read/lines ftp://ftp.site.com/file.r
```

Cet exemple renvoie le fichier sous la forme d'un bloc de lignes. Voir le chapitre sur les Fichiers pour plus d'informations sur les raffinements de la fonction **read**.

Pour écrire un fichier texte sur le serveur FTP, utilisez la fonction **write** :

```
write ftp://ftp.site.com/file.r read %file.r
```

La fonction **write** peut prendre aussi des raffinements. Voir le chapitre sur les Fichiers.

Comme normalement avec les transferts de fichiers texte, toutes les fins de lignes seront correctement converties durant le transfert FTP.

Voici un simple script qui met à jour les fichiers de votre site Web :

```
site: ftp://wwwuser:secret@www.site.dom/pages

files: [%index.html %home.html %info.html]

foreach file files [write site/:file read file]
```

Ceci ne devrait pas être utilisé pour transférer des images ou des fichiers de son, qui sont binaires. Utilisez la technique montrée dans la section suivante sur le Transfert de fichiers binaires.

En complément des fonctions **read** et **write**, vous pouvez aussi utilisez **load**, **save**, et **do** avec FTP.

```
data: load ftp://ftp.site.com/database.r
```

```
save ftp://ftp.site.com/data.r data-block

do ftp://ftp.site.com/scripts/test.r
```

12.4 Transfert de fichiers binaires

Pour évitez la conversion des caractères de fin de ligne, lors du transfert de fichiers binaires (images, archives zippés, fichiers exécutables), utilisez le raffinement **/binary**.

Par exemple, pour lire un fichier binaire depuis un serveur FTP :

```
data: read/binary ftp://ftp.site.com/file
```

Pour faire en local une copie du fichier :

```
write/binary %file read/binary ftp://ftp.site.com/file
```

Pour écrire un fichier binaire sur un serveur :

```
write/binary ftp://ftp.site.com/file read/binary %file
```

Aucune conversion de fin de ligne n'est réalisée.

Pour transférer un ensemble de fichiers graphiques sur un site Web, utilisez le script :

```
site: ftp://user:pass@ftp.site.com/www/graphics

files: [%icon.gif %logo.gif %photo.jpg]

foreach file files [
    write/binary site/:file read/binary file
]
```

12.5 Ajout à des fichiers

Le protocole FTP vous permet aussi d'ajouter du texte ou des données à un fichier existant. Pour faire cela, utilisez le raffinement **write/append** comme cela est décrit dans le chapitre sur les Fichiers.

```
write/append ftp://ftp.site.com/pub/log.txt reform
    ["Log entry date:" now newline]
```

Ceci peut aussi être fait avec des fichiers binaires.

```
write/binary/append ftp://ftp.site.com/pub/log.txt
    read/binary %datafile
```

12.6 Consultation de répertoires

Pour lire le contenu d'un répertoire FTP distant, faites suivre le nom du répertoire d'un symbole "/" (slash).

```
print read ftp://ftp.site.com/
pub-files: read ftp://ftp.site.com/pub/
```

Le slash terminal (/) indique qu'il s'agit d'un accès à un répertoire et non à un fichier. Le slash n'est pas toujours nécessaire mais il est recommandé dés lors que vous savez que vous accédez à un répertoire.

Le bloc de noms de fichiers qui est renvoyé comprend tous les éléments du répertoire. Au sein du bloc, les noms de répertoires sont signalés avec un slash à la fin de leur nom.

Par exemple :

```
foreach file read ftp://ftp.site.com/pub/ [
    print file
]
readme.txt
rebol.r
rebol.exe
library/docs/
```

Vous pouvez aussi utilisez la fonction **dir?** sur un élément pour déterminer s'il s'agit d'un fichier ou d'un répertoire.

12.7 Information concernant les fichiers

Les mêmes fonctions qui fournissent de l'information concernant les fichiers locaux peuvent aussi fournir des informations sur les fichiers distants FTP. Ceci inclut les fonctions **modified?**, **size?**, **exists?**, **dir?**, et **info?**.

Vous pouvez utilisez la fonction exists? pour savoir si un fichier existe :

```
if exists? ftp://ftp.site.com/pub/log.txt [
    print "Log file is there"
]
```

Ceci marche également avec les répertoires, mais pensez à inclure le slash final après le nom du répertoire :

```
if exists? ftp://ftp.site.com/pub/rebol/ [
    print read ftp://ftp.site.com/pub/rebol/
]
```

Pour connaître la taille ou la date de modification d'un fichier :

```
```

```
print size? ftp://ftp.site.com/pub/log.txt

print modified? ftp://ftp.site.com/pub/log.txt
```

Pour déterminer si un nom d'élément est celui d'un répertoire :

```
if dir? ftp://ftp.site.com/pub/text [
    print "It's a directory"
]
```

Vous pouvez obtenir toutes ces informations en une seule requête avec la fonction **info?** :

```
file-info: info? ftp://ftp.site.com/pub/log.txt

probe file-info

print file-info/size
```

Pour effectuer la même action sur un répertoire :

```
probe info? ftp://ftp.site.com/pub/
```

Pour afficher le contenu d'un répertoire :

```
files: open ftp://ftp.site.com/pub/

forall files [
    file: first files
    info: info? file
    print [file info/date info/size info/type]
]
```

12.8 Créer un répertoire

De nouveaux répertoires FTP peuvent être créés avec la fonction **make-dir** :

```
make-dir ftp://user:pass@ftp.site.com/newdir/
```

12.9 Suppression de fichiers

En supposant que vous ayez les permissions appropriées pour cela, des fichiers peuvent être supprimés du serveur FTP en utilisant la fonction **delete** :

```
delete ftp://user:pass@ftp.site.com/upload.txt
```

Vous pouvez aussi effacer des répertoires :

```
delete ftp://user:pass@ftp.site.com/newdir/
```

Notez que le répertoire doit être vide pour que sa suppression puisse se faire.

12.10 Renommage de fichiers

Vous pouvez renommer un fichier avec la ligne :

```
rename ftp://user:pass@ftp.site.com/foo.r %bar.r
```

Le nouveau nom du fichier sera bar.r.

Le protocole FTP permet aussi de déplacer un fichier vers un autre répertoire avec :

```
rename ftp://user:pass@ftp.site.com/foo.r %pub/bar.r
```

Pour renommer un répertoire sur un site FTP, là encore n'oubliez pas le slash à la fin du nom du répertoire :

```
rename ftp://user:pass@ftp.site.com/rebol/ rebol-old/
```

12.11 Au sujet des mots de passe

Les exemples ci-dessus inclut le mot de passe au sein des URLs, mais si vous prévoyez de partager votre script, vous ne voulez probablement pas que cette information soit connue. Voici une manière simple de demander le mot de passe via un interrogation en ligne de commande (prompt) et de construire l'URL adéquate :

```
pass: ask "Password? "

data: read join ftp://user: [pass "@ftp.site.com/file"]
```

Ou, vous pouvez demander à la fois le nom de l'utilisateur et le mot de passe :

```
user: ask "Username? "
pass: ask "Password? "
data: read join ftp:// [
    user ":" pass "@ftp.site.com/file"
]
```

Vous pouvez aussi ouvrir une connexion FTP en spécifiant un port plutôt qu'une URL. Ceci vous permet d'utiliser n'importe quel mot de passe, même ceux pouvant contenir des caractères spéciaux qui ne sont pas facile à écrire dans une URL.

Un exemple de spécification pour un port ouvrant une connexion FTP serait :

```
ftp-port: open [
    scheme: `ftp
    host: "ftp.site.com"
    user: ask "Username? "
    pass: ask "Password? "
]
```

Voir la partie sur la spécification de ressources réseau ci-dessus pour plus de détail.

12.12 Transfert de fichiers volumineux

Le transfert de fichiers volumineux nécessite quelques considérations particulières. Vous souhaitez sans doute transférer un fichier par morceaux pour réduire la quantité de mémoire requise par votre ordinateur, et pour fournir à l'utilisateur un retour sur l'évolution du transfert.

Voici un exemple qui télécharge un très gros fichier par morceaux.

```
inp: open/binary/direct ftp://ftp.site.com/big-file.bmp
out: open/binary/new/direct %big-file.bmp
buf-size: 200000
buffer: make binary! buf-size + 2

while [not zero? size: read-io inp buffer buf-size][
    write-io out buffer size
    total: total + size
    print ["transferred:" total]
]
```

Utilisez absolument le raffinement **/direct**, faute de quoi le fichier entier sera mis en buffer en interne de REBOL. Les fonctions **read-io** et **write-io** permettent de réutiliser la mémoire du buffer qui a déjà été allouée. D'autres fonctions comme **copy** alloue de la mémoire supplémentaire.

Si le transfert s'interrompt, vous pouvez redémarrer le transfert FTP à partir de l'endroit où il s'est arrêté. Pour cela, examinez le fichier résultant (out) ou la taille pour déterminer d'où recommencer le transfert.

Ouvrez à nouveau le fichier avec le raffinement **/custom** en spécifiant le mot restart et l'endroit d'où redémarrer la lecture.

Voici un exemple avec la fonction **open** où la variable "total" indique la longueur déjà lue du fichier :

```
inp: open/binary/direct/custom
ftp://ftp.site.com/big-file.bmp
reduce ['restart total]
```

Notez que le redémarrage d'un transfert FTP fonctionne seulement avec des transferts binaires. Il ne peut être effectué avec des transferts de fichiers texte parce que les conversions de caractères de fin de ligne induisent des modifications de taille.

[Retour au sommaire]

13. NNTP - Network News Transfer Protocol

Le protocole NNTP (Network News Transfer Protocol) est la base pour des dizaines de milliers de newsgroups qui assurent un forum public pour des millions d'utilisateurs d'Internet. REBOL comprend deux niveaux de support pour le protocole NNTP.

- Le support interne qui autorise des fonctionnalités et des accès très limités. C'est le *scheme NNTP*.
- Un niveau supérieur de fonctionnalité qui est assuré par le scheme **news**, implémenté dans le fichier appelé **nntp.r**.

13.1 Lecture d'une liste de newsgroup

NNTP comprend deux composants : une liste de newsgroups supportés par un serveur de newsgroup dédié (typiquement, les newsgroups sont sélectionnés par les fournisseurs d'accès à Internet); et une base de données de messages en cours qui se rapportent à des newsgroups particuliers.

Pour retrouver la liste des messages de tous les newsgroups pour un serveur de news spécifique, utilisez la fonction **read** avec une URL NNTP telle que :

```
groups: read nntp://news.example.com
```

Cette opération peut durer un certain temps, selon votre connexion; il y a des milliers de newsgroups.

13.2 Lire tous les messages

Si vous utilisez une connexion rapide, vous pouvez lire tous les messages relatif à un newsgroup avec :

```
messages: read nntp://news.example.com/alt.test
```

Cependant, soyez prudents. Certains newsgroups peuvent avoir des milliers de messages. Cela peut prendre un long moment pour télécharger tous les messages, et être très consommateur en mémoire, pour les manipuler.

13.3 Lecture de messages particuliers

Pour lire des messages spécifiques, ouvrez NNTP avec un port, et utilisez les fonctions relatives aux séries pour accéder aux messages. C'est assez similaire au fonctionnement vu précédemment pour la lecture de vos emails avec un port POP.

Par exemple :

```
group: open nntp://news.example.com/alt.test
```

Vous pouvez utilisez la fonction **length?** pour déterminer le nombre de messages valables pour le newsgroup :

```
print length? group
```

Pour lire le premier message pour le newsgroup, utilisez la fonction **first** :

```
message: first group
```

Pour sélectionner un message spécifique dans le groupe, via son index, utilisez **pick** :

```
message: pick group 37
```

Pour créer un simple boucle permettant de scanner tous les messages contenant un mot-clé :

```
forall group [
    if find msg: first first group "REBOL" [
    print msg
    ]
]
```

Rappelez-vous qu'à la fin de la boucle, la série est positionnée sur sa fin (tail). Si vous avez besoin de revenir au début de la série des messages :

```
group: head group
```

N'oubliez pas non plus de fermer le port une fois que vous avez terminé de l'utiliser :

```
close group
```

13.4 Manipulation des en-têtes de News

Les messages des news incluent systématiquement un en-tête. L'en-tête stocke des informations sur l'expéditeur, le résumé, des mot-clés, le sujet, la date, et d'autres champs également.

Les en-têtes sont manipulés sous la forme d'objet REBOL. Pour convertir un message de news, en objet d'en-tête, vous pouvez utilisez la fonction **import-email**.

Par exemple,

```
message: first first group
header: import-email message
```

Vous pouvez à présent accéder aux différents champs du message de news :

```
print [header/from header/subject header/date]
```

Les différents newsgroups et les différents clients utilisent différents champs pour leurs en-têtes. Pour

voir les champs valables pour un message particulier, affichez le premier item de l'objet d'en-tête, ici appelé header :

```
print first header
```

13.5 Expédier un message

Avant d'envoyer un message, vous devez créer un en-tête pour lui. Voici un en-tête générique qui peut être utilisé pour les newsgroups :

```
news-header: make object! [
    Path: "not-for-mail"
    Sender: Reply-to: From: system/user/email
    Subject: "Test message"
    Newsgroups: "alt.test"
    Message-ID: none
    Organization: "Docs For All"
    Keywords: "Test"
    Summary: "A test message"
]
```

Avant de l'envoyer, vous devez créer un numéro d'identification global pour lui. Voici une fonction qui réalise cela :

```
make-id: does [
    rejoin [
    "<"
    system/user/email/user
    "."
    checksum form now
    "."
    random 999999
    "@"
    read dns://
    ">"
    ]
]
print news-header/message-id: make-id
<carl.4959961.534798@fred.example.com>
```

A présent, vous pouvez combiner l'en-tête avec le message. Ils doivent être séparés par au moins une ligne blanche. Le contenu du message est lu à partir d'un fichier.

```
write nntp://news.example.net/alt.test rejoin [
    net-utils/export news-header
    newline newline
    read %message.txt
    newline
]
```

[Retour au sommaire]

14. CGI - Common Gateway Interface

Le mode CGI (Common Gateway Interface) est utilisé avec de nombreux serveurs Web pour effectuer des traitements en plus et au delà de l'interface Web normale.

Les requêtes CGI sont soumises par des navigateurs Web à des serveurs Web. Typiquement, lorsque qu'un serveur reçoit une requête CGI, il exécute un script qui traite la requête et renvoie un résultat au navigateur. Ces scripts CGI sont écrits dans de très nombreux langages, et l'un des manières les faciles de manipuler du CGI est d'utiliser REBOL.

14.1 Paramétrage du serveur CGI

Le paramètrage d'un accés CGI est différent pour chaque serveur Web. Voir les instructions fournies avec votre serveur.

Typiquement, un serveur possède une option permettant d'activer le mode CGI. Vous devez activer cette option et fournir un chemin vers le répertoire où se trouvent vos scripts CGI. Un répertoire courant pour les scripts CGI s'appelle *cgi-bin*.

Sur les serveurs Web Apache, l'option ExecCGI active le mode CGI, et vous devez indiquer un répertoire (cgi-bin) pour vos scripts. C'est le mode de fonctionnement par défaut d'Apache.

Pour configurer CGI pour Microsoft IIS, allez dans les propriétés pour cgi-bin, et cliquez sur le bouton pour la configuration. Sur le panneau de configuration, cliquez sur "add" et entrez le chemin vers l'exécutable rebol.exe. La syntaxe pour cela est :

```
C:\rebol\rebol.exe -cs %s %s
```

Les deux symboles %s sont nécessaires pour passer correctement le script et les arguments en ligne de commande à REBOL. Ajoutez l'extension propre aux fichiers REBOL (.r), et mettez le dernier champ sur PUT, DELETE. L'item "script engine" n'a pas besoin d'être sélectionné.

L'option -cs qui est passée à REBOL permet le mode CGI et autorise le script à accéder à tous les fichiers (!! Voir notes ci-dessous sur comment les scripts peuvent limiter les accès à des fichiers pour des répertoires spécifiques.)

D'autres serveurs Web que ceux décrits au-dessus nécessitent d'être configuré pour pouvoir exécuter l'exécutable REBOL avec des fichiers portant l'extension .r et avec l'option demandée -cs.

14.2 Scripts CGI

Avant de pouvoir exécuter un script sur la plupart des serveurs CGI, celui-ci devra disposer des permissions de fichiers adéquates. Sur les systèmes de type Unix ou ceux utilisant un serveur Apache, il sera nécessaire de modifier les permissions pour autoriser le script en lecture et en exécution pour tous les utilisateurs. Cela peut être fait avec la fonction Unix chmod.

Si vous être débutant dans ces concepts, vous devriez lire le manuel de votre système d'exploitation, ou demander à votre administrateur système avant de modifier les permissions du fichier.

Pour Apache, et divers autres serveurs Web qui exécutent des scripts REBOL, vous devez placer un en-tête dédié au début de chaque script. L'en-tête indique le chemin vers l'exécutable REBOL et l'option -cs. Voici un simple script CGI qui affiche la chaîne de caractères "hello!".

```
#!/path/to/rebol -cs
```

```
REBOL [Title: "CGI Test Script"]

print "Content-Type: text/plain"

print ""  ; required

print "Hello!"
```

De nombreuses choses peuvent empêcher un script CGI de fonctionner correctement. Testez d'abord ce simple script avant d'en essayez de plus complexe. Si votre script ne marche pas, voici quelques points à vérifiez :

- Vous avez activé l'option CGI sur votre serveur Web.
- La première ligne du script commence par #! et le chemin exact vers REBOL.
- L'option -cs est fournie à REBOL.
- Le script commence avec l'affichage de "Content-Type:" (!!voir ci-dessous)
- Le script est dans le bon répertoire. (normalement, le répertoire cgi-bin).
- Le script a les permissions adéquates (lecture et exécution pour tous).
- Le script contient le saut de ligne nécessaire. Certains serveurs n'exécuteront pas le script s'il celui-ci ne contient pas le caractère CR (carriage return) pour les sauts de lignes. Vous devrez convertir le fichier. (Utilisez REBOL pour faire cela en une ligne : write file read file).
- Le script ne contient pas d'erreurs. Testez-le hors du mode CGI pour être sûr que le script se charge (n'a pas d'erreurs de syntaxe) et fonctionne proprement. Fournissez lui quelques données en exemple, et testez-le avec.
- Tous les fichiers auquel le script doit accéder ont les permissions adéquates.

Souvent l'un ou plusieurs de ces points n'est pas correct et empêche votre script de marcher. Vous aurez une erreur au lieu de voir une page Web. Si cette erreur est du type "Server Error" ou "CGI error", alors typiquement, il y a quelque chose à faire avec les droits ou le paramétrage du script. S'il s'agit d'un message d'erreur REBOL, alors le script est exécuté, mais vous avez une erreur à l'intérieur du script.

Dans le script présenté ci-dessus en exemple, la ligne "Content-Type" est critique. C'est la partie de l'en-tête HTTP qui est renvoyée vers le navigateur et qui l'informe du type de contenu qu'il va recevoir. Cette ligne est suivie d'une ligne vierge, qui la sépare du contenu.

Différents types de contenu peuvent être retourné. L'exemple précédent était en texte, mais vous pouvez aussi retourner du HTML comme montré dans l'exemple suivant. (Voir le manuel de votre serveur Web pour plus d'informations sur les types de contenu).

Le type de contenu et la ligne vierge peuvent être combinées en une seule ligne. L'ajout du symbole (^/) (accent circonflexe suivi d'un slash) est assimilable à une ligne vierge, qui effectue la séparation d'avec le contenu.

```
print "Content-Type: text/plain^/"
```

C'est une bonne habitude de toujours afficher cette ligne immédiatement au début de votre script. Cela permet le renvoi des messages d'erreur vers le browser si votre script rencontre une erreur.

Voici un simple script CGI qui affiche l'heure :

```
#!/path/to/rebol -cs

REBOL [Title: "Time Script"]

print "Content-Type: text/plain^/"
```

```
print ["The time is now" now/time]
```

14.3 Générer du contenu HTML

Il y a autant de manières de créer du contenu HTML qu'il y a de façons de créer des chaînes de caractères. Ce code génère une page qui affiche un compteur du nombre de visiteurs :

```
#!/path/to/rebol -cs

REBOL [Title: "HTML Example"]

print "Content-Type: text/html^/"

count: either exists? %counter [load %counter][0]
save %counter count: count + 1

print [
    {<HTML><BODY><H2>Web Counter Page</H2>
    You are visitor} count {to this page!<P>
    </BODY></HTML>}
]
```

Le script en exemple ci-dessus charge et sauvegarde le compteur via un fichier texte. Pour rendre accessible ce fichier, il est nécessaire de lui donner les droits appropriés afin de le rendre accessible par tous les utilisateurs.

14.4 Environnement et variables CGI

Lorsqu'un script CGI s'exécute, le serveur fournit des informations à REBOL concernant la requête CGI et ses arguments. Toutes ces informations sont placées sous la forme d'un objet dans l'objet **system/options**. Pour voir les attributs de cet objet, saisissez :

```
probe system/options/cgi
make object! [
    server-software: none
    server-name: none
    gateway-interface: none
    server-protocol: none
    server-port: none
    request-method: none
    path-info: none
    path-translated: none
    script-name: none
    query-string: none
    remote-host: none
    remote-addr: none
    auth-type: none
    remote-user: none
    remote-ident: none
    Content-Type: none
    content-length: none
    other-headers: []
]
```

Bien sûr, votre script ignorera la plupart de ces informations, mais certaines d'entre elles peuvent être utiles. Par exemple, vous voudrez créer un fichier log qui enregistre les adresses réseau des machines effectuant les requêtes, ou vérifiant le type de navigateur utilisé.

Pour générer une page CGI qui affiche ces informations dans votre navigateur :

```
#!/path/to/rebol -cs

REBOL [Title: "Dump CGI Server Variables"]

print "Content-Type: text/plain^/"

print "Server Variables:"

probe system/options/cgi
```

Si vous voulez utilisez ces informations dans un fichier log, vous devrez les écrire dans un fichier. Par exemple, pour enregistrer les adresses des visiteurs de votre page CGI, vous devrez écrire :

```
write/append/lines %cgi.log
    system/options/cgi/remote-addr
```

Les raffinements **/append** et **/lines** forcent l'écriture à s'effectuer à la suite des enregistrements précédents et ligne par ligne. Voici une autre approche qui inscrit plusieurs items sur la même ligne :

```
write/append %cgi.log reform [
    system/options/cgi/remote-addr
    system/options/cgi/remote-ident
    system/options/cgi/content-type
    newline
]
```

14.5 Requêtes CGI

Il existe deux méthodes pour fournir des données à votre script CGI : GET et POST. La méthode GET encode les données CGI dans l'URL. Elle est utilisée pour fournir des informations au serveur. Vous aurez remarqué sans doute que certaines URLs ressemblent à celle-ci :

```
http://www.example.com/cgi-bin/test.r?&data=test
```

La chaîne de caractères qui suit le point d'interrogation (?) fournit les arguments au script CGI. Parfois, ceux-ci peuvent être assez longs. La chaîne est passée à votre script lorsque celui s'exécute. Elle peut être obtenue à partir de l'attribut **cgi/query-string**.

Par exemple, pour afficher cette chaîne depuis un script :

```
print system/options/cgi/query-string
```

Les données contenues dans la chaîne peuvent inclure n'importe quelle data dont vous avez besoin.

Cependant, parce que cette chaîne fait partie de l'URL, les données doivent y être encodées. Il y a des restrictions sur les caractères qui y sont autorisés.

De plus, lorsque ces données sont produites par un formulaire HTML, elles sont encodées de façon standard. Ces données peuvent être décodées et mises dans un objet avec le code :

```
cgi: make object! decode-cgi-query
    system/options/cgi/query-string
```

La fonction **decode-cgi-query** renvoie un bloc qui contient les noms des variables et leurs valeurs. Voir l'exemple avec le formulaire HTML dans la section suivante.

La méthode POST transmet les données CGI sous forme d'une chaîne. Les données n'ont pas besoin d'être encodées. Elles peuvent être dans n'importe quel format que vous le souhaitez et peuvent même être binaires. Les données sont lues à partir de l'entrée standard. Vous devrez les lire depuis l'entrée standard avec un code comme celui-ci :

```
data: make string! 2002
read-io system/ports/input data 2000
```

Ceci devrait sélectionner les 2000 premiers octets de données POST et les mettre dans une chaîne de caractères.

Une bonne pratique pour les données POST est d'utiliser un dialecte REBOL et de créer un petit analyseur (parseur). Les données POST peuvent être chargées et analysées sous forme de bloc. Voir le chapitre sur le Parsing.

> Avertissement au sujet des blocs :
>
> Ce n'est pas une bonne idée de passer à REBOL des blocs pour qu'ils soient directement évalués, car cela induit un risque sur la sécurité. Par exemple, quelqu'un pourrait envoyer via POST un bloc qui permettrait de lire ou d'écrire des fichiers sur le serveur. Par ailleurs, passer des blocs qui sont interprétés par votre script (via un dialecte) est sans danger.

Voici un exemple de script qui affiche les datas POST dans votre navigateur :

```
#!/path/to/rebol -cs

REBOL [Title: "Show POST data"]

print "Content-Type: text/html^/"
data: make string! 10000
foreach line copy system/ports/input [
    repend data [line newline]
]

print [
    <HTML><BODY>
    {Here is the posted data.}
```

```
    <HR><PRE>data</PRE>
    </BODY></HTML>
]
```

14.6 Traitement des formulaires HTML

Le protocole CGI est utilisé souvent afin de traiter des formulaires HTML. Les formulaires acceptent des champs variés et les soumettent au serveur Web avec la méthode GET ou la méthode POST.

Voici un exemple qui utilise la méthode GET du CGI pour traiter un formulaire et envoyer un email en guise de résultat. Il y a deux parties : la page HTML et le script CGI.

Voici une page HTML qui comprend un formulaire :

```
<HTML><BODY>

<FORM ACTION="http://example.com/cgi-bin/send.r" METHOD="GET">

<H1>CGI Emailer</H1><HR>

Enter your email address:<P>

<INPUT TYPE="TEXT" NAME="email" SIZE="30"><P>

<TEXTAREA NAME="message" ROWS="7" COLS="35">
Enter message here.
</TEXTAREA><P>

<INPUT TYPE="SUBMIT" VALUE="Submit">

</FORM>
</BODY></HTML>
```

Lorsque le formulaire HTML est validé, il est traité par la méthode GET et les datas passées au script **send.r**. Voici un exemple de script. Ce script décode les données du formulaire et envoie le message électronique. Il retourne une page de confirmation.

```
#!/path/to/rebol -cs

REBOL [Title: "Send CGI Email"]

print "Content-Type: text/html^/"

cgi: make object! decode-cgi-query
    system/options/cgi/query-string

print {<HTML><BODY><H1>Email Status</H1><HR><P>}

failed: error? try [send to-email cgi/email cgi/message]

print either failed [
    {The email could not be sent.}
][
    [{The email to} cgi/email {was sent.}]
]

print {</BODY><HTML>}
```

Ce script doit être nommé send.r et être placé dans le répertoire cgi-bin. Ses droits doivent permettrent qu'il soit lu et exécuté par tous.

Lorsque le formulaire HTML aura été soumis par un navigateur, le script s'exécutera. Il décode la chaîne de caractères de la requête CGI dans l'objet **cgi**. L'objet a comme variables l'email et le message qui sont utilisés par la fonction **send**. Avant l'envoi du courrier électronique, le champ "email", de type *string*, est converti en type de données *email*.
La fonction **send** est placée à l'intérieur d'un bloc **try** pour capturer les erreurs pouvant se produire, et le message qui est généré dans ce cas.

D'autres exemples de scripts CGI peuvent être trouvés dans la bibliothèque de scripts REBOL : http://www.rebol.com/library/library.html.

[Retour au sommaire]

15. TCP - Transmission Control Protocol

En supplément de tous les protocoles décrits précédemment, vous pouvez créer vos propres serveurs et clients avec le protocole TCP (Transmission Control Protocol).

15.1 Créer des clients

Des ports TCP peuvent être ouverts avec REBOL de la même façon qu'avec les autres protocoles, en utilisant une URL TCP. Pour ouvrir une connexion TCP vers un serveur Web (HTTP), sur le port 80 :

```
http-port: open tcp://www.example.com:80
```

Une autre manière d'ouvrir une connexion TCP est de fournir les spécifications de port directement. Cela remplace l'ouverture d'une URL et est souvent plus commode :

```
http-port: open [
    scheme: 'tcp
    host: "www.example.com"
    port-id: 80
]
```

Puisque les ports sont des séries, vous pouvez utilisez les fonctions relatives aux séries pour émettre et recevoir des données. L'exemple ci-dessous effectue une requête sur le serveur HTTP ouvert dans l'exemple précédent. Il utilise la fonction **insert** pour placer des datas dans la série, le port http-port qui les envoie au serveur :

```
insert http-port join "GET / HTTP/1.0^/^/"
```

Les deux caractères de nouvelle ligne (^/) sont utilisés pour signifier au serveur que l'en-tête HTTP a été émis.

Les caractères (^/ ou newline) sont automatiquement convertis en séquences CR LF car le port a été ouvert en mode texte. Le serveur traite la requête HTTP et retourne un résultat au port. Pour lire le résultat, utilisez la fonction **copy** :

```
```

```
while [data: copy http-port] [prin data]
```

Cette boucle continuera de récupérer les datas jusqu'à ce que **none** soit retourné par la fonction **copy**. Ce comportement diffère selon les protocoles. Un "**none**" est retourné car le serveur ferme la connexion. D'autres protocoles peuvent utiliser un caractère spécial pour marquer la fin du transfert.

A présent que toutes les datas ont été reçues, le port HTTP doit être fermé :

```
close http-port
```

Voici un autre exemple qui ouvre un port POP en TCP sur un serveur :

```
pop: open/lines tcp://fred.example.com:110
```

Cet exemple utilise le raffinement **/lines**. La connexion sera à présent orientée lignes. Les données seront lues et écrites sous forme de lignes.

Pour lire la première ligne depuis le serveur :

```
print first pop
+OK QPOP (version 2.53) at fred.example.com starting.
```

Pour émettre vers le serveur un nom d'utilisateur pour le login POP :

```
insert pop "user carl"
```

Puisque le port est ouvert en mode ligne, un caractère de fin de ligne est émis aprés la chaîne "user carl" insérée.

La réponse du serveur POP peut être lue avec :

```
print first pop
+OK Password required for carl.
```

Et le reste de la communication devrait s'effectuer :

```
insert pop "pass secret"

print first pop
+OK carl has 0 messages (0 octets).
insert pop "quit"

first pop
+OK Pop server at fred.example.com signing off.
```

La connexion doit enfin être fermée :

```
close pop
```

15.2 Création de serveurs

Pour créer un serveur, vous devrez attendre des demandes de connexions et y répondre lorsqu'elles se produisent. Pour définir un port sur votre machine qui peut utilisé pour attendre les connexions entrantes :

```
listen: open tcp://:8001
```

Remarquez que vous n'indiquez pas de nom de machine, seulement un numéro de port. Ce type de port est appelé un port d'écoute (listen port). Votre système accepte maintenant les connexions sur le port numéro 8001.

Pour attendre une connexion d'une autre machine, vous attendez sur le port d'écoute.

```
wait listen
```

Cette fonction ne se terminera pas tant qu'une connexion n'aura pas été réalisée.

> NOTE : Il existe diverses options possibles pour wait. Par exemple, vous pouvez attendre sur plusieurs ports ou aussi avec un time-out.

Vous pouvez maintenant ouvrir le port pour la connexion depuis la machine qui a contacté votre système.

```
connexion: first listen
```

Ceci renvoie la connexion qui a été faite sur le port d'écoute. C'est un port comme tous les autres, et il peut être utilisé pour émettre et recevoir des données, au moyen des fonctions **insert**, **copy**, **first**, et des autres fonctions relatives aux séries.

```
insert connexion "you are connected^/"

while [newline <> char: first connexion] [
    print char
]
```

Lorsque la communication est complète, la connexion doit être fermée.

```
close connexion
```

Vous êtes à présent prêt pour la connexion suivante sur le port d'écoute. Vous pouvez attendre encore et utiliser **first** encore pour la nouvelle connexion.

Lorsque votre serveur en a fini, vous devez fermer le port d'écoute avec :

```
close listen
```

15.3 Un tout petit serveur

Voici un serveur REBOL assez commode qui nécessite seulement que quelques lignes de code. Ce serveur évalue le code REBOL qui lui est envoyé. Les lignes de REBOL en provenance d'un client sont lues jusqu'à ce qu'une erreur se produise. Chaque ligne doit être une expression REBOL compléte. Elle peut être de n'importe quelle longueur mais doit faire une seule ligne.

```
server-port: open/lines tcp://:4321

forever [
    connexion-port: first server-port
    until [
    wait connexion-port
    error? try [do first connexion-port]
    ]
    close connexion-port
]
close server-port
```

Si une erreur se produit, la connexion est fermée et le serveur se met en attente de la connexion suivante.

Voici un exemple de script pour un client qui vous permet de rentrer à distance des commandes REBOL :

```
server: open/lines tcp://localhost:4321
until [error? try [insert server ask "R> "]]
close server
```

Ici la requête est faite pour déterminer si la connexion a été interrompue du fait d'une erreur.

15.4 Test du code TCP

Pour tester le code de votre serveur, connectez-vous depuis votre propre machine, plutôt que d'avoir un serveur et un client. Ceci peut être fait avec deux processus REBOL distincts ou même un seul processus.

Pour vous connecter en local sur votre machine, vous pouvez utilisez une ligne comme celle-ci :

```
port: open tcp://localhost:8001
```

Voici un exemple qui crée deux ports connectés entre eux en mode ligne. Il s'agit d'une sorte de port

"echo" puisque vous émettez des datas vers vous-mêmes. C'est un bon test pour votre code et l'usage du réseau :

```
listen: open/lines tcp://:8001
remote: open/lines tcp://localhost:8001
local: first listen
insert local "How are you?"
print first remote  ; response
close local
close remote
close listen
```

[Retour au sommaire]

16. UDP - User Datagram Protocol

Le protocole UDP (User Datagram Protocol) est un autre protocole de transport qui fournit une méthode pour communiquer entre machines, mais qui n'est pas orienté connexion, à la différence de TCP.

Il permet d'envoyer des datagrammes, des paquets de données entre des machines. L'utilisation d'UDP est assez différente de celle de TCP. Le protocole UDP est plus simple, mais il est aussi moins sûr. Il n'y a pas de garantie qu'un paquet atteigne toujours sa destination. De plus, UDP ne dispose de mécanisme de contrôle de flux. Si vous envoyez des messages trop rapidement, les paquets peuvent être perdus.

Comme pour TCP, la fonction **wait** peut être utilisée pour attendre qu'un nouveau paquet arrive et la fonction **copy** peut être utilisée pour renvoyer les données. S'il n'y a pas de données, **copy** attend jusqu'à ce qu'il y en ait. Notez cependant que la fonction **insert** n'attend jamais.

Voici un exemple de script pour un petit serveur UDP :

```
udp: open udp://:9999
wait udp
print copy udp
insert udp "response"
close udp
```

Les messages insérés dans le port *udp* ici par le serveur sont expédiés vers le client de qui a été reçu le dernier message. Ceci permet aux réponses d'être émises pour les messages entrants. Cependant, contrairement à TCP, vous n'avez pas de connexion continue entre les machines. Chaque transfert de paquet donne lieu à un échange spécifique.

Le script client pour communiquer avec le serveur ci-dessus pourrait être :

```
udp: open udp://localhost:9999
insert udp "Test"
wait udp
print copy udp
close udp
```

Vous devez aussi savoir que la taille maximale d'un paquet UDP dépend du système d'exploitation. 32 Ko et 64 Ko représentent des valeurs courantes. Afin d'envoyer de grandes quantités de données, vous devrez mettre en buffer les datas, et les diviser en paquets plus petits. Par ailleurs, votre programmation

doit être particulièrement soignée pour être certain que chaque partie des données est bien reçue. Rappelez-vous qu'avec UDP, il n'y a pas de garantie.

Chapitre 14 - Les Ports

Ce document est la traduction française du Chapitre 14 du User Guide de REBOL/Core, qui concerne les Ports.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
29 avril 2005 17:55	1.0.1	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr
20 mai 18:55	1.0.1	Corrections mineures	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Généralités

Les ports permettent d'accéder à des séries externes résultant de fichiers, du réseau, de consoles,

de périphériques externes, d'évenements, de codecs, ou de bases de données.

Les données d'un port sont traitées en utilisant les fonctions standards de REBOL propres aux séries, comme décrit dans le chapitre sur les Séries.

Les ports sont utilisés aussi bien pour des entrées que pour des sorties de données.

Les types de données manipulés par un port dépendent de la façon dont celui-ci a été ouvert.

Trois types de données sont possibles :

Type	Description
String	une série d'octets, avec conversion des sauts de lignes (par défaut)
Binary	une série d'octets, sans modification de la donnée
Block	une série de valeurs REBOL

Un port peut être ouvert dans l'un des deux modes suivants (*buffering modes*) :

Mode	Description
Mise en buffer (*buffered*)	toutes les données sont conservées en mémoire (par défaut)
Direct	les données ne sont pas conservées en mémoire

De plus, un port peut être ouvert avec :

Type	Description
Wait	le port attend, écoute, pour savoir si des données arrivent (par défaut)
No-wait	aucune attente de données

[Retour au sommaire]

3. Ouverture d'un port

3.1 La fonction open

La fonction **open** initialise l'accès à un port selon des paramètres, qui lui ont été spécifiés.
La fonction peut être invoquée avec un nom de fichier, une URL, ou un objet.
De plus, il existe plusieurs raffinements qui affecterons l'opération d'ouverture ou l'accès au contenu du port.

La méthode la plus simple pour utiliser **open** est de lui fournir un nom de fichier, ou une URL en argument.

Dans l'exemple ci-dessous, un port de type fichier est ouvert :

```
fp: open %file.txt
```

La variable *fp* référence le port.
Si le port n'est pas ouvert, une erreur se produira.
Si besoin, l'erreur peut être capturée avec la fonction **try**.

Par défaut, le port est ouvert en mode "*buffered*".

Cela signifie qu'un fichier est accèdé et modifié en mémoire, et que les changements ne sont pas écrits dans le fichier tant que le port n'est pas fermé ou mis à jour.

Pour les fichiers, la fonction **open** créera automatiquement le fichier si celui-ci n'existe pas auparavant.

```
close open %somefile.txt
if exists? %somefile.txt [print "somefile exists"]
somefile exists
```

Le raffinement **/new** peut être utilisé pour remplacer un fichier existant.

```
write %somefile.txt "text in some file"
print read %somefile.txt
text in some file
close insert open/new %somefile.txt "new data"
print read %somefile.txt
new data
```

Une fois le port ouvert, les opérations relatives aux séries comme **copy**, **insert**, **remove**, **clear**, **first**, **next**, et **length?** peuvent être utilisées pour accèder au contenu du port et le modifier.

3.2 Raffinements de la fonction open

La fonction **open** accepte un certain nombre de raffinements qui peuvent être utilisés pour modifier son comportement :

Raffinement	Description

/binary	les données du port sont binaires
/string	les données sont de type texte, les fins de ligne sont automatiquement converties
/with	précise une fin de ligne spéciale
/lines	manipule les données une ligne à la fois ou comme un bloc de lignes
/direct	ne met pas en mémoire
/new	crée ou recrée la cible d'un port
/read	ouverture en mode lecture seule
/write	open for write only operation
/no-wait	pas d'attente pour les données
/skip	saute une partie des données
/allow	définit les attributs des fichiers
/custom	permet des raffinements spéciaux

[Retour au sommaire]

4. Fermeture d'un port

L'accès à un port se termine quand la fonction **close** est invoquée.

Toutes les données en mémoire qui n'ont pas été sauvées seront écrites dans le fichier cible.

L'exemple ci-dessous ferme le port *fp* utilisé précedemment.

```
close fp
```

Si vous tentez de fermer un port qui n'est pas ouvert, une erreur se produira.

Un port qui a été fermé peut être réouvert, avec la fonction **open** :

```
```

```
open fp
```

[Retour au sommaire]

5. Lecture du contenu d'un port

La fonction **copy** (utilisée pour les séries) va servir à lire les données d'un port, une fois celui-ci ouvert :

```
print copy fp
I wanted the gold, and I sought it,I scrabbled and mucked like
a slave....
```

Cette fonction attend normalement les données du port.

Si vous ne voulez pas attendre la fin des données, ouvrez le port avec le raffinement **/no-wait**.

Et pour lire une partie seulement des données, utilisez **copy/part** :

```
print copy/part fp 35
I wanted the gold, and I sought it,
```

Notez que le deuxième argument de **copy** peut être soit une longueur, soit une position au sein de la série, le contenu du port.

Pour lire juste une partie des données du port, vous pouvez utiliser les fonctions **find** et **copy**.

```
a: find fp "famine"
print copy/part a find a newline
famine or scurvy -- I fought it;
```

Les fonctions ordinales comme **first**, ou de position comme **next**, peuvent aussi être utilisées sur le port :

```
print first fp
I
print first next next fp
w
```

La fonction **copy** retournera **none** lorsque toutes les données du port auront été lues.

En mode **/no-wait**, la fonction **copy** renverra une *chaîne vide* si aucune donnée n'est disponible avec le port.

```
tp: open/direct/binary/no-wait tcp://system:8000
content: make binary! 1000
```

```
while [wait tp  data: copy tp] [append content data]
close tp
```

[Retour au sommaire]

6. Ecriture dans un port

La fonction **insert** sert pour l'écriture de données dans un port.

```
insert fp "I was a fool to seek it."
```

Si le port est en mode "*buffered*", la modification sera répercutée seulement lorsque le port sera fermé ou mis à jour (avec la fonction **update**).
Si le port est ouvert avec **/direct**, alors tout changement est immédiatement pris en compte.

Tous les raffinements de la fonction **insert** peuvent être utilisés sur le port.

Par exemple, l'écriture de 20 caractères "espace" dans le port (avec **/dup**):

```
insert/dup fp " " 20
```

Vous pouvez aussi utiliser sur le port les fonctions habituelles de modifications des séries, comme **remove**, **clear**, **change**, **append**, **replace**, etc.

Pour effacer un seul caractère, ou bien plusieurs :

```
remove fp

remove/part fp 20
```

Et pour effacer tous les caractères restants, écrivez :

```
clear fp
```

[Retour au sommaire]

7. Mise à jour d'un port

La fonction **update** force un port à rafraîchir son état selon le fonctionnement du périphérique externe. (ici fichier).

Par exemple, lors de l'écriture d'un fichier présent en cache, la fonction **update** permet de forcer le vidage du buffer.

En lecture, la fonction **update** peut être utilisée pour être sûr que toutes les données en attente ont été mises en buffer.

```
update fp
```

[Retour au sommaire]

8. Attente sur un port

La fonction **wait** est essentielle aux programmes gérant de manière asynchrone des échanges de données.

Avec **wait**, vous pouvez attendre des données d'un ou plusieurs ports, ou bien qu'un time-out se produise.

La fonction peut accepter en argument un port unique :

```
wait port
```

mais un **bloc de plusieurs ports** peut aussi être fourni :

```
wait [port1 port2 port3]
```

De plus, une valeur de time-out peut également être indiquée, sous la forme d'un nombre de secondes ou comme une valeur de type **time!** :

```
wait [port1 port2 10]

wait [port1 port2 0:00:05]
```

Le premier exemple indique un time-out de 10 secondes.
Le second exemple sera en time-out dans 5 secondes.

La fonction **wait** renvoie le premier port qui est prêt ou **none** si un time-out s'est produit.

```
ready: wait [port1 port2 10]
if ready [data: copy ready]
```

L'exemple précédent lira des données du premier port prêt, si un time-out ne se produit pas avant.

Pour obtenir un bloc comprenant tous les ports prêts, utilisez le raffinement **/all** :

```
ready: wait/all [port1 port2 10]
if ready [
    foreach port ready [
    append data copy port
    ]
```

```
]
```

Cet exemple ajoutera les données de tous les ports disponibles à une seule série, appelée ici *data*.

Vous pouvez aussi utiliser la fonction **dispatch** pour évaluer un bloc, ou une fonction basée sur les résultats de **wait** vis-à-vis de plusieurs ports.

```
dispatch [
    port1 [print "port1 awake"]
    port2 [print "port2 awake"]
    10 [print "time-out!"]
]
```

Usage de /no-wait et /direct

Pour utilisez **wait** avec la plupart des ports, vous aurez besoin de spécifier les raffinements **/no-wait** et **/direct** avec **open**. Ceci permet d'indiquer que les fonctions habituelles pour l'accès aux données ne doivent pas être bloquantes et que les données ne sont pas mises en cache (*buffered mode*).

```
port1: open/no-wait/direct tcp://system:8000
```

[Retour au sommaire]

9. Autres modes pour un port

9.1 Mode ligne

La fonction **open** permet un accès en mode ligne.

Dans ce mode, la fonction **first** retournera une ligne de texte, plutôt qu'un caractère. (NdT: par exemple, avec un fichier texte séquentiel).

L'exemple ci-dessous lit un fichier une ligne à la fois :

```
fp: open/lines %file.txt
print first fp
I wanted the gold, and I got it --
print third fp
Yet somehow life's not what I thought it,
```

Le raffinement **/lines** est aussi utile pour les protocoles Internet qui sont orientés "lignes".

```
tp: open/lines tcp://server:8000
```

```
print first tp
```

9.2 Ecriture et lecture seules

Vous pouvez utiliser le raffinement **/read** pour ouvrir un port en lecture seule :

```
fp: open/read %file.txt
```

Les modifications faites en cache ne sont pas répercutées au fichier.

Pour ouvrir un port en écriture seulement, utilisez le raffinement **/write** :

```
fp: open/write %file.txt
```

Les ports de type fichiers ouverts avec le raffinement **/write** ne liront pas les données courantes à l'ouverture du port.

La fermeture, ou la mise à jour d'un port en écriture seule force les données existantes dans le fichier à être remplacées (écrasées).

```
insert fp "This is the law of the Yukon..."
close fp
print read %file.txt
This is the law of the Yukon...
```

9.3 Accés en mode direct

Le raffinement **/direct** ouvre un port sans mise en buffer (mémoire) des données.

C'est assez pratique pour accèder à des fichiers par morceaux, notamment lorsque le fichier est trop grand pour être stocké en mémoire.

```
fp: open/direct %file.txt
```

La lecture des données avec **copy** entraîne la modification de la position de la tête de la série, pour le port :

```
print copy/part fp 40
I wanted the gold, and I sought it,^/ I
print copy/part fp 40
scrabbled and mucked like a slave.^/Was i
```

En mode direct, le port sera toujours sur la position de tête :

```
print head? fp
true
```

La fonction **copy** renverra **none** lorsque la fin des données du port est atteinte.

Voici un exemple qui utilise des ports en mode direct pour copier un fichier quelle que soit sa taille.

```
from-port: open/direct %a-file.jpg
to-port: open/direct %a-file.jpg
while [data: copy/part from-port 100000 ][
    append to-port data
]
close from-port
close to-port
```

9.4 Sauter des données

Il existe deux façons de sauter les données existantes dans le port.

Premièrement, vous pouvez ouvrir le port avec le raffinement **/skip**. La fonction **open** passera automatiquement dans le port au point spécifié.

Par exemple :

```
fp: open/direct/skip %file.big 1000000

fp: open/skip http://www.example.com/bigfile.dat 100000
```

Deuxièmement, vous pouvez utiliser la fonction **skip** sur le port.
Pour les fichiers qui sont ouverts avec les raffinements **/direct** et **/binary**, l'opération de saut est identique à l'opération seek (recherche/déplacement dans un fichier)

Les données ne sont pas lues dans le cache. Ce n'est pas possible dans le mode **/string** car les sauts de lignes interférent avec la valeur de saut.

```
fp: open/direct/binary %file.dat
fp: skip fp 100000
```

[Retour au sommaire]

10. Permissions d'accès sur les fichiers

Les fichiers créés par REBOL prennent des permissions d'accès par défaut.
Sur les systèmes d'exploitation Windows et Macintosh, les fichiers sont créés avec des privilèges permettant leur contrôle total. Sur les systèmes Unix, les fichiers sont créés avec des permissions selon l'*umask* courant.

Lorsqu'on utilise **open** ou **write** pour accèder à un fichier, le raffinement **/allow** peut être utilisé pour définir les permissions d'accès.

Le raffinement **/allow** prend un bloc en argument. Ce bloc peut être constitué de n'importe lequel ou de tous les mots : **read**, **write**, **execute**.

> **Restrictions liées au système d'exploitation**
>
> Le raffinement **/allow** permettra de définir des permissions d'accès uniquement sur les systèmes d'exploitation qui le permettent. Si le système ne supporte pas l'attribution de certaines permissions, elles seront ignorées.
> Par exemple, les fichiers sur les systèmes Unix peuvent être défini comme exécutables (execute), mais les systèmes Windows et Macintosh ne supportent pas cette option.
>
> Pour un système Unix, les possibilités de permissions de fichiers sont restreintes à celles de l'utilisateur.
>
> Suivant le système, l'usage d'**/allow** conduit à effacer les permissions d'accès pour les utilisateurs.

Pour mettre un fichier en lecture seule, utilisez **open/allow**, ou **write/allow** avec un bloc read :

```
write/allow %file.txt [read]
```

Pour donner les droits en lecture et en exécution à un fichier :

```
open/allow %file.txt [read execute]
```

Vous pouvez définir des droits similaires, en écriture :

```
write/allow %file.txt [read write]
```

Pour supprimer tous les accès à un fichier (pour les systèmes d'exploitation qui font la différence), mettez un bloc vide pour les permissions :

```
write/allow %file.txt []
```

Pour un accès complet :

```
write/allow %file [read write execute]
```

[Retour au sommaire]

11. Ports relatifs aux répertoires

Les ports relatifs aux répertoires vous permettent d'accèder directement à ceux-ci.

Quand vous ouvrez un répertoire, vous obtenez l'accés au répertoire présenté comme un bloc de noms de fichiers.

```
mydir: open %intro/
forall mydir [print first mydir]
CVS/
history.t
intro.t
overview.t
quick.t
close mydir
```

Vous pouvez avancer à une position spécifique à l'intérieur de la série du répertoire et effacer un fichier, avec par exemple le code suivant :

```
dir: open %.
remove next dir
close dir
```

Ce code efface le second fichier dans le répertoire courant.
De la même manière,

```
remove at dir 5
```

devrait effacer le cinquiéme fichier dans le répertoire, et :

```
clear dir
```

devrait effacer tous les fichiers du répertoire.

Pour effacer tous les fichiers dont le nom contient "junk", vous pouvez écrire :

```
dir: open %intro/
while [not tail? dir] [
    either find first dir "junk" [remove dir][
        dir: next dir
    ]
]
close dir
```

Les modifications apportées au répertoire sont répercutées quand le port relatif au répertoire est fermé ou quand il est mis à jour.

Pour forcer l'application d'un changement, utilisez le code suivant :

```
update dir
```

NdT : il semblerait que **update** ne fonctionne pas, avec le scheme 'directory.

La méthode d'accès à un répertoire peut aussi être utilisée pour changer les noms des fichiers. Après l'ouverture du port, la ligne :

```
change at dir 3 %newname.txt
```

permet de renommer le troisième fichier dans le répertoire. Pareillement, le nom de n'importe lequel des fichiers dans le répertoire peut être modifié.

Voici un exemple pour renommer tous les fichiers dans un répertoire en ajoutant le mot REBOL au nom initial :

```
dir: open %intro/
forall dir [insert first dir "REBOL"]
close dir
```

NdT : l'exemple ne fonctionne pas. Bug signalé à RT.

Chapitre 15 - Le Parsing

Ce document est la traduction française du chapitre 15 du User Guide de REBOL/Core, qui concerne le Parsing.

Contenu

[Retour au sommaire]

1. Historique de la Traduction

Date	Version	Commentaires	Auteur	Email
5 avril 2005 10:37	1.0.0	Traduction initiale et relecture	Philippe Le Goff	lp--legoff--free--fr

---

[Retour au sommaire]

2. Introduction

Le parsing segmente une suite de caractères ou de valeurs en plus petits élements. Il peut être utilisé pour reconnaître des caractères ou des valeurs qui apparaissent dans un ordre spécifique. En plus de fournir une approche aisée et puissante des expressions régulières, et de la recherche de motifs, le parsing vous permet de créer votre propre dialecte pour un usage spécifique.

La fonction **parse** se présente sous la forme générale :

```
parse series rules
```

Le premier argument fourni, **series**, est ce qui va être parsé, et peut être une chaîne de caractères ou un bloc.

Si l'argument est un chaîne, celle-ci est parsée caractére par caractére.

Si l'argument est un bloc, il est parsé par valeur.

Le second argument, **rules**, indique comment l'argument "series" est parsée.

L'argument **rules** peut être une chaîne de types simples à parser, ou un bloc pour un parsing plus sophistiqué.

La fonction **parse** accepte deux raffinements : **/all** et **/case** .

Le rafinement **/all** permet de parser tous les caractères à l'intérieur d'une chaîne, incluant tous les délimiteurs , comme l'espace, le caractère de tabulation, celui de nouvelle ligne, la virgule, et le point-virgule.

Le raffinement **/case** autorise le parsing d'une chaîne avec prise en compte de la casse de la chaîne (majuscule/minuscule).

Quand **/case** n'est pas spécifié, les majuscules et les minuscules sont traitées sans distinction.

[Retour au sommaire]

3. Parsing simple

3.1 Cas usuel

Une forme simple du parsing est le splitting (segmentation) de chaînes :

```
parse string none
```

La fonction **parse** segmente l'argument (la chaîne de caractères) en input en un bloc composé de plusieurs chaînes, coupant chaque chaîne partout où est rencontré un délimiteur comme l'espace, la tabulation, le caractère de nouvelle ligne (newline), la virgule, et le point-virgule.

Un argument "none" fourni à la fonction **parse** lui indique qu'**aucun** autre de ces délimiteurs n'est accepté.

Par exemple :

```
probe parse "The trip will take 21 days" none
["The" "trip" "will" "take" "21" "days"]
```

De la même manière,

```
probe parse "here there,everywhere; ok" none
["here" "there" "everywhere" "ok"]
```

Dans l'exemple ci-dessus, les virgules et points-virgules ont été otés des chaînes résultantes.

3.2 D'autres délimiteurs

Vous pouvez spécifier d'autres délimiteurs dans l'argument "rules", qui peuvent être combinés avec les délimiteurs standards (**espace, tabulation, virgule, newline, point virgule**).

Par exemple, le code suivant effectue un parsing d'un numéro de téléphone, en ayant ajouté un tiret "-" aux délimiteurs :

```
probe parse "707-467-8000" "-"
["707" "467" "8000"]
```

Le code ci-dessous ajoute le symbole **égal (=)** et les guillemets **(")** aux délimiteurs :

```
probe parse <IMG SRC="test.gif" WIDTH="123">  {="}
["IMG" "SRC" "test.gif" "WIDTH" "123"]
```

3.3 Pour n'avoir aucun des délimiteurs standards

Pour ne pas utiliser les délimiteurs standards du parsing, il faut utiliser le raffinement **/all**. Avec le raffinement **/all**, seuls les délimiteurs passés dans l'argument **rules** sont utilisés.

Ci-dessous, l'exemple montre le parsing d'une chaîne basé sur les vigules uniquement; les autres délimiteurs sont ignorés.
Ainsi les espaces à l'intérieur des chaînes ne sont pas enlevés :

```
probe parse/all "Harry, 1011 Main St., Ukiah" ","
["Harry" " 1011 Main St." " Ukiah"]
```

Vous pouvez parser des chaînes de caractères qui contiennent des caractères "null" comme séparateurs (comme pour certains types de fichiers):

```
parse/all nulled-string "^(null)"
```

[Retour au sommaire]

4. Régles de Grammaire

La fonction **parse** peut accepter des régles de grammaire écrites dans un dialecte de REBOL.

Les dialectes sont des sous-langages de REBOL qui utilisent les mêmes formes lexicales pour tous les types de données, mais permettent un ordonnancement différent des valeurs au sein d'un bloc.

Au sein du dialecte, la grammaire et le vocabulaire de REBOL sont modifiés pour être rendre similaires en structure à la forme BNF (Backus-Naur- Form) bien connue, qui est souvent utilisée pour spécifier des régles de grammaires, des protocoles réseau, des formats d'en-tête, etc.

Pour définir une régle, on utilise un bloc indiquant l'ordre des entrées.

Par exemple, si vous souhaitez parser une chaîne et renvoyer les caractères "the phone", vous pouvez utiliser la régle :

```
parse string ["the phone"]
```

4.1 Alternatives

Pour permettre un nombre indéfini d'espaces ou aucun espace entre les mots (soit 0 ou plus), écrivez la régle ainsi :

```
parse string ["the" "phone"]
```

Il est possible d'alterner des régles avec une bar vertical (|).

Par exemple :

```
["the" "phone" | "a" "radio"]
```

accepte des chaînes de caractéres qui contiennent l'une ou l'autre des chaînes suivantes :

```
the phone
a radio
```

Une régle peut contenir des blocs qui sont traités comme des sous-régles.

Le code suivant :

```
[ ["a" | "the"] ["phone" | "radio"] ]
```

acceptent des chaînes de caractères qui contiennent l'une ou l'autre des chaînes suivantes :

```
a phone
a radio
the phone
the radio
```

Afin d'améliorer la lisibilité, il est judicieux d'écrire des sous-régles comme des blocs à part et de leur donner un nom caractéristique :

```
article: ["a" | "the"]
moyen-de-com: ["phone" | "radio"]
parse string [article moyen-de-com]
```

4.2 Nombre variables d'occurences

En plus de rechercher une instance unique d'une chaîne, vous pouvez fournir un nombre ou une plage de nombres correspondant au nombre d'occurrences possibles d'un motif.

L'exemple suivant illustre cela :

```
[3 "a" 2 "b"]
```

qui acceptera n'importe quelle chaîne de caractères du genre :

```
aaabb
```

L'exemple suivant montre l'usage d'une plage de nombres :

```
[1 3 "a" "b"]
```

acceptera des chaînes de caractères correspondant à l'une ou l'autre des possibilités suivantes :

```
ab aab aaab
```

Le point de départ d'une plage de nombres peut être 0 (zero), indiquant que celui-ci est optionnel.

```
[0 3 "a" "b"]
```

acceptera des chaînes de caractères du genre :

```
b ab aab aaab
```

4.3 some et any

Il est possible d'utiliser le mot **some** pour spécifier que un ou plusieurs caractères doivent être recherchés. Sur le même principe, il est possible d'utiliser le mot **any** pour spécifier que ou plusieurs caractères doivent être cherchés.

Par exemple, **some** est utilisé dans le code suivant :

```
[some "a" "b"]
```

qui accepte des chaînes contenant une ou plusieurs occurrences
des caractères a ou b :

```
ab aab aaab aaaab
```

L'exemple suivant montre l'usage de **any** :

```
[any "a" "b"]
```

Ici seront acceptées les chaînes de caractères contenant zéro ou plusieurs caractères a ou b :

```
b ab aab aaab aaaab
```

Les mots **some** et **any** peuvent aussi être utilisés sur des blocs.

Par exemple :

```
[some ["a" | "b"]]
```

accepte des chaînes de caractères contenant **n'importe quelle** combinaison des caractères a et b.

Une autre manière d'exprimer qu'un caractère est optionnel est de fournir dans les alternatives de choix le mot **none** :

```
["a" | "b" | none]
```

Cet exemple indique que les chaînes acceptées peuvent contenir a ou b ou **none** . Le mot **none** est un moyen pratique pour spécifier des modéles ("patterns") optionnels ou pour gérer les cas d'erreur lorsqu'aucun modéle ou motif ne concorde.

[Retour au sommaire]

5. Saut de caractères (Skipping Input)

Les mots **skip**, **to**, et **thru** permettent de se déplacer dans les entrées.

Skip peut être utilisé pour sauter un caractère, ou en conjugaison avec la fonction **repeat** pour sauter plusieurs caractères :

```
["a" skip "b"]
["a" 10 skip "b"]
["a" 1 10 skip "b"]
```

Pour aller à un caractère spécifique, utilisez **to** :

```
["a" to "b"]
```

L'exemple précédent fait commencer le parsing au caractère "a" et le termine à "b" mais **sans inclure** "b".

Pour inclure le caractère "b" dans le parsing, utiliser **thru** :

```
["a" thru "b"]
```

Cet exemple fait commencer le parsing au caractère "a", le termine à "b", mais celui-ci est à présent **inclus** dans le parsing .

La régle suivante permet par exemple de sélectionner le titre d'une page html et l'affiche :

```
page: read http://www.REBOL.com/
parse page [thru <title>  copy text to </title> ]
print text
REBOL Technologies
```

Le premier **thru** trouve la balise "<title> " et va immédiatement après. Ensuite, le chaîne en input est copiée dans une variable appelée "**text**", jusqu'à ce que la balise (tag) "</title> " soit atteinte (mais sans la dépasser sinon, la balise serait inclus dans la variable "text").

[Retour au sommaire]

6. Recherches par types de données

Durant le parsing sur des chaînes de caractères, les types de données et les mots peuvent être utilisés pour être croisés avec les caractères de la chaîne en input :

Type de correspondance	Description
"abc"	correspondance avec la chaîne compléte
#"c"	correspondance avec un caractère unique
tag (<u>, ,...)	correspondance avec une balise
end	correspondance avec la fin de l'entrée
(bitset) ensemble de caractères	correspondance avec n'importe quel caractére dans ceux proposés

Pour utiliser ces mots (à l'exception de **bitset**, qui est expliqué plus loin) dans une seule régle, il suffit d'écrire :

```
[<B>  ["excellent" | "incredible"] #"!" </B>  end]
```

Cet exemple va parser les chaînes de caractères :

```
<B> excellent! </B>
<B> incredible! </B>
```

Le mot **end** spécifie que plus rien ne suit dans le flux d'entrée. L'entrée a été complétement parsée. Ceci est optionnel et selon que la valeur de retour de la fonction d'analyse nécessite d'être vérifiée. (Voir la section sur l'évaluation ci-dessous, pour plus d'information.)

Le type de données "**bitset**" nécessite plus d'explications.
Les "**bitset**" sont utilisés pour définir de façon efficace des ensembles, des jeux de caractères. La fonction **charset** vous permet de définir des caractères uniques ou encore un ensemble de caractères.

Par exemple, la ligne :

```
digit: charset "0123456789"
```

définira un jeu de caractères contenant des chiffres.

Il est alors possible de définir des régles comme :

```
[3 digit "-" 3 digit "-" 4 digit]
```

qui vont parser des numéros de téléphones du genre :

```
707-467-8000
```

Pour accepter un nombre quelconque de chiffres, il est possible d'écrire une régle :

```
digits: [some digit]
```

Un jeu de caractères peut aussi spécifier une plage de caractères.

Par exemple, le bitset "digit" précédent peut être ré-écrit ainsi :

```
digit: charset [#"0" - #"9"]
```

Bien sûr, il est possible de combiner des caractères spécifiques et des plages de caractères :

```
the-set: charset ["+-." #"0" - #"9"]
```

Pour compléter ces notions, voici la description des caractères alphanumériques :

```
alphanum: charset [#"0" - #"9" #"A" - #"Z" #"a" - #"z"]
```

Ces jeux de caractères peuvent être modifiés avec les fonctions **insert** et **remove**, ou encore être créés avec des combinaisons de caractères via les fonctions **union** et **intersect**.

La ligne suivante recopie l'ensemble de caractères "digit" et lui ajoute le caractère "." :

```
digit-dot: insert copy digit "."
```

Ci-dessous, la définition de plusieurs jeux de caractères utiles pour le parsing :

```
digit: charset [#"0" - #"9"]
alpha: charset [#"A" - #"Z" #"a" - #"z"]
alphanum: union alpha digit
```

[Retour au sommaire]

7. Récursivité des Régles

Voici un exemple de régles qui vont analyser des expressions mathématiques et donner une priorité aux opérateurs arithmétiques utilisés :

```
expr:    [term ["+" | "-"] expr | term]
term:    [factor ["*" | "/"] term | factor]
factor:  [primary "**" factor | primary]
primary: [some digit | "(" expr ")"]
digit:   charset "0123456789"
```

A présent, nous pouvons analyser n'importe quel type d'expressions mathématiques.
Les exemples suivants renvoient "**true**", indiquant que les expressions fournies sont valides :

```
probe parse "1 + 2 * ( 3 - 2 ) / 4" expr
true
probe parse "4/5+3**2-(5*6+1)" expr
true
```

Remarquez que certaines de ces régles font référence à elles-mêmes.

Par exemple, la régle **expr** inclut **expr** .
C'est une technique utile pour définir des répétitions et des combinaisons de régles. La régle est récursive, elle fait référence à elle-même.

Quand des régles récursives sont utilisées, il faut faire attention afin d'éviter une récursivité sans fin.

Par exemple :

```
expr: [expr ["+" | "-"] term]
```

va créer une boucle infinie parce que la première chose que la régle "**expr**" fait est d'utiliser encore "**expr**" .

[Retour au sommaire]

8. Evaluation

Normalement, vous analysez une chaîne en vue de produire un résultat.

Vous pouvez faire plus que vérifier la conformité de la chaîne, vous voulez faire quelque chose **pendant** qu'elle est analysée.

Par exemple, vous pouvez vouloir sélectionner des morceaux de diverses parties de la chaîne en input, créer des blocs de valeurs reliées entre elles, ou calculer une valeur.

8.1 Valeur retournée par la fonction parse

Les exemples dans les précedentes sections ont montré comment parser des chaînes, mais aucun résultat n'était produit.

La valeur retournée par l'analyse permet de vérifier qu'une chaîne a la grammaire, et la structure indiquée; elle indique le succès ou non de l'analyse.

Les exemples suivants illustrent cela :

```
probe parse "a b c" ["a" "b" "c"]
true
probe parse "a b" ["a" "c"]
false
```

La fonction **parse** renvoie "true" seulement si elle atteint la fin de la chaîne de caractères fournie en argument (input).

Une correspondance infructeuse entre la régle et la chaîne stoppe l'analyse de celle-ci.

Si l'analyse dépasse les valeurs à rechercher avant atteindre la fin de la série, elle ne traverse pas la série et retourne "false".

```
probe parse "a b c d" ["a" "b" "c"]
false
probe parse "a b c d" [to "b" thru "d"]
true
probe parse "a b c d" [to "b" to end]
true
```

8.2 Expressions dans les règles

Au sein d'une régle, vous pouvez inclure une expression REBOL qui sera à évaluer, lorsque l'analyse atteindra ce point dans la régle.

Des parenthéses sont utilisées pour indiquer de telles expressions :

```
string: "there is a phone in this sentence"
probe parse string [
    to "a"
    to "phone" (print "found phone")
    to end
]
found phone
true
```

L'exemple ci-dessus effectue un parsing de la chaîne "string" et affiche le message "found phone" à

la fin de l'analyse. Si la chaîne ou le mot "phone" n'existe pas, ou que l'analyse ne peut être faite, l'expression n'est pas évaluée.

Les expressions à évaluer peuvent apparaître n'importe où à l'intérieur d'une régle, et plusieurs expressions peut exister dans différentes parties d'une régle.

Par exemple, le code suivant affichera différentes réponses selon ce qui aura été trouvé dans la chaîne en input (en entrée) :

```
string: "there is a phone in this sentence"
parse string [
    "a" | "the"
    to "phone" (print "answer") |
    to "radio" (print "listen") |
    to "tv"    (print "watch")
]
answer
string: "there is the radio on the shelf"
parse string [
    "a" | "the"
    to "phone" (print "answer") |
    to "radio" (print "listen") |
    to "tv"    (print "watch")
]
listen
```

Voici un exemple qui comptabilise le nombre de fois où la balise <pre> apparait dans une page HTML :

```
count: 0
page: read http://www.REBOL.com/docs/dictionary.html
parse page [any [thru <pre>  (count: count + 1)]]
print count
777
```

8.3 Copie de l'entrée

L'action la plus commune réalisée avec parse est de récupérer des parties de la chaîne en cours d'analyse.

Ceci peut être effectué avec le mot **copy** suivi du nom de la variable vers laquelle vous voulez copier le morceau de chaîne.

L'exemple suivant montre le parsing d'un titre de page Web :

```
parse page [thru <title>  copy text to </title> ]
print text
REBOL/Core Dictionary
```

La régle utilisée dans cet exemple permet de se déplacer dans la chaîne en input jusqu'à trouver la balise <title> .

A partir de ce point, une copie du flux de caractères en entrée est faite dans la variable "text". La copie continue jusqu'à ce que la balise <title> soit trouvée.

La copie peut aussi être réalisée avec des blocs entiers de régles.

Par exemple :

```
[copy heading ["H" ["1" | "2" | "3"]]
```

Ici, la variable "heading" contient les chaînes H1, H2, ou H3.

Ceci fonctionne aussi pour de grandes régles multi-blocs.

8.4 Indexation de l'entrée

L'action **copy** effectue une copie du morceau de chaîne trouvé, mais ce n'est pas toujours souhaitable.
Dans certains cas, il est plus judicieux de sauver dans une variable la position courante dans le flux en input .

Note :
Le mot **copy** utilisé dans le parsing est différent de la fonction **copy** rencontrée dans les expressions REBOL.
Le parsing utilise un dialecte de REBOL, et le mot **copy** y a un sens différent au sein de celui-ci.

Dans l'exemple suivant, la variable "begin" pointe vers une position de référence dans la page en input, juste après <title> .

La variable "ending", elle, fait référence dans la page à une position juste avant </title>.
Ces variables peuvent être manipulées comme elles le seraient avec n'importe quelle autre série.

```
page: read http://www.REBOL.com/docs/dictionary.html
parse page [
    thru <title>  begin: to </title>  ending:
    (change/part begin "Word Reference Guide" ending)
]
```

Vous pouvez constater que le parsing ci-dessus a réellement changé le contenu du titre, *in situ* du contenu de la variable "page".

```
parse page [thru <title>  copy text to </title> ]
print text
Word Reference Guide
```

Voici un autre exemple permettant d'indexer la position de chaque balise <table>dans un fichier HTML :

```
page: read http://www.REBOL.com/index.html
tables: make block! 20
```

```
parse page [
    any [to "<table" mark: thru ">"
        (append tables index? mark)
    ]
]
```

Le bloc "tables" contient à présent la position de chaque balise :

```
foreach table tables [
    print ["table found at index:" table]
]
table found at index: 836
table found at index: 2076
table found at index: 3747
table found at index: 3815
table found at index: 4027
table found at index: 4415
table found at index: 6050
table found at index: 6556
table found at index: 7229
table found at index: 8268
```

Note :
La position courante dans le flux d'entrée peut aussi être modifiée. Le paragraphe suivant explique comment réaliser cela.

8.5 Modification de la chaîne :

A présent qu'il est possible d'obtenir une position dans la chaîne en input, vous pouvez aussi utiliser d'autres fonctions relatives aux séries, comme **insert**, **remove**, et **change**.

Pour écrire un script qui remplace tous les points d'interrogation (?) par des points d'exclamation, vous pouvez écrire :

```
str: "Où est la dinde?  Avez-vous vu la dinde?"
parse str [some [to "?" mark: (change mark "!") skip]]
print form str
Où est la dinde !  Avez-vous vu la dinde !
```

Le mot **skip** en fin de régle avance d'un caractère le flux en entrée, jusqu'au caractère suivant (ce n'est pas nécessaire dans ce cas, mais c'est une bonne pratique).

Un autre exemple, pour insérer l'heure courante chaque fois que le mot "time" apparaît dans un texte :

```
str: "at this time, I'd like to see the time change"
parse str [
    some [to "time"
        mark:
        (remove/part mark 4  mark: insert mark now/time)
        :mark
```

```
    ]
]
print str
at this 14:42:12, I'd like to see the 14:42:12 change
```

Attention au mot **:mark** utilisé ci-dessus.
Il décale l'index de l'input à une nouvelle position.

La fonction **insert** retourne la nouvelle position juste après le point d'insertion de l'heure courante.
Le mot **:mark** est utilisé pour positionner l'index de l'input à cet endroit.

8.6 Utilisation d'objets

Lors d'une analyse grammaticale importante, à partir d'un ensemble de règles, des variables sont définies pour rendre cette grammaire plus lisible. Cependant, les variables utilisées appartiennent au contexte global et peuvent être source de conflit ou de confusion avec d'autres variables ayant le même nom quelque part ailleurs dans le programme.

La solution pour régler ce problème est **d'utiiser un objet** pour rendre toutes les régles de grammaire **locale au contexte de cet objet**.

Par exemple :

```
tag-parser: make object! [
    tags: make block! 100
    text: make string! 8000
    html-code: [
        copy tag ["<" thru "> "] (append tags tag) |
        copy txt to "<" (append text txt)
    ]
    parse-tags: func [site [url!]] [
        clear tags clear text
        parse read site [to "<;" some html-code]
        foreach tag tags [print tag]
        print text
    ]
]
tag-parser/parse-tags http://www.REBOL.com
```

8.7 Debuggage

Une fois les régles écrites, plusieurs tests de débuggage sont souvent nécessaires.

En particulier, vous voudrez sans doute savoir jusqu'où vous avez été dans le parsing d'une règle.
La fonction **trace** peut être employée pour observer l'évolution d'une opération d'analyse, mais elle peut produire des milliers de lignes, qu'il sera difficile de passer en revue.

Une meilleure façon de débugger est d'introduire des expressions de débuggage dans les régles de parsing.

Par exemple, pour débugger la régle :

```
```

```
[to "<IMG" "SRC" "=" filename ">"]
```

Ici, l'insertion de la fonction **print** au plus proche des parties à surveiller vous permettra de suivre l'évolution du parsing au travers de la régle :

```
[to "<IMG" (print 1) "SRC" "=" (print 2)
    filename (print 3) ">"]
```

Cet exemple affichera : 1, 2, puis 3 lorsque la régle sera exécutée.

Une autre approche est d'afficher une partie de la chaîne en input durant le parsing :

```
[
   to "<IMG" here: (print here)
   "SRC" "=" here: (print here)
    filename here: (print here) ">"
]
```

Si cette méthode est réguliérement utilisée, vous pouvez créer ... une régle pour cela :

here: [where: (print where)]

```
[
   to "<IMG" here
   "SRC" "=" here
    filename here ">"
]
```

La fonction **copy** peut aussi être utiisée pour indiquer quelles sous-chaînes ont été analysées pendant que la règle était active.

[Retour au sommaire]

9. Jongler avec les espaces

La fonction **parse** ignore en principe tous les espaces blancs existants entre les motifs à chercher.

Par exemple, la régle :

```
["a" "b" "c"]
```

acceptent les chaînes qui correspondent à:

```
abc
a bc
ab c
```

```
a b c
a  b  c
```

ou autres combinaisons avec des espaces .

Pour imposer une convention spécifique pour les espaces, utilisez **parse** avec la raffinement **/all**. Dans l'exemple précédent, l'usage de ce raffinement conduirait le parsing à avoir la seule correspondance : **(abc)**.

```
parse/all "abc" ["a" "b" "c"]
```

La spécifiaction du raffinement **/all** force chaque caractère (en incluant les délimiteurs standards comme l'espace, la tabulation, la virgule, le point virgule, le retour à la ligne) a être manipulés par la régle fournie.

Pour utiliser des caractères d'espacements dans vos régles, créez un jeu de caractères qui définit les caractères valides comme :

```
spacer: charset reduce [tab newline #" "]
```

Si vous voulez un seul caractère d'espacement entre chaque lettre, écrivez :

```
["a" spacer "b" spacer "c"]
```

Pour permettre de multiples espacements, il est possible d'écrire :

```
spaces: [some spacer]
["a" spaces "b" spaces "c"]
```

Pour des régles plus sophistiquées, créez un ensemble de caractéres qui vous laisse scanner la chaîne en entrée jusqu'à trouver un caractère "espace" :

```
non-space: complement spacer
to-space: [some non-space | end]
words: make block! 20
parse/all text [
    some [copy word to-space (append words word) spacer]
]
```

L'exemple ci-dessus construit un bloc de mots (**words**) à partir de la chaîne "text" fournie en input.

La fonction **complement** inverse le jeu de caractères.
A présent, la variable "non-space" contient tout, excepté les caractères d'espacement définis précédemment.

La régle "to-space" accepte un ou plusieurs caractères différents du caractère "espace" ou la fin du flux d'entrée.
La régle principale attend un mot, copie le mot jusqu'à trouver un espace, ajoute le mot au bloc "words", puis saute le caractère "espace", et recommence avec le mot suivant.

[Retour au sommaire]

10. Parsing de blocs et Dialectes

Il est possible de parser des blocs de la même manière que des chaînes. Un ensemble de régles spécifie l'ordre dans lequel les valeurs sont attendues.

Cependant, à la différence du parsing des chaînes de caractères, le parsing de blocs ne s'effectue pas via les caractères ou les délimiteurs standard.
L'analyse des blocs est faite au niveau de la valeur, ce qui facilite la spécification des règles de grammaire et rend les traitements beaucoup plus rapides.

Le parsing de blocs est la manière la plus simple pour créer des **dialectes REBOL**.

Les **dialectes** sont des sous-langages de REBOL qui utilisent les mêmes formes lexicales pour tous les types de données, mais permettent un ordonnancement différent des valeurs au sein d'un bloc.

L'ordre des valeurs n'a pas besoin d'être conforme à l'ordre normalement requis pour les arguments de fonctions REBOL.

Les dialectes fournissent un moyen très puissant pour parser des expressions et cela dans des domaines spécifiques d'utilisation.

Par exemple, les propres régles du parseur sont définies comme un dialecte.

10.1 Correspondance de mots

Lors de l'analyse d'un bloc, pour avoir une correspondance avec un mot REBOL, il faut le spécifier comme un mot littéral :

```
'name
'when
'empty
```

10.2 Correspondance avec des types de données

Vous pouvez avoir une correspondance avec la valeur de n'importe quel type de données en indiquant le mot REBOL du datatype.

Voir ci-dessous.

Type de données du mot	Description
string!	correspondance avec n'importe chaîne entre guillemets

time!	correspondance avec valeurs de type time
date!	correspondance avec valeurs de type date
tuple!	correspondance avec valeurs de type tuple

NOTE :
Attention à ne pas oublier le "!" qui fait partie du nom, sinon une erreur sera générée.

10.3 Caractéres non autorisés

Les opérations d'analyse permises pour les blocs sont celles qui traitent des caractères spécifiques. Par exemple, une correspondance ne peut pas être établie avec la première lettre d'un mot ou d'une chaîne, ni avec les caractères d'espacement ou newline.

10.4 Exemples de Dialectes

Quelques courts exemples peuvent aider à illustrer le parsing de blocs :

```
block: [when 10:30]
print parse block ['when 10:30]
print parse block ['when time!]
parse block ['when set time time! (print time)]
```

A noter qu'un mot spécifique peut être testé en utilisant son équivalent littéral (literal word) dans la régle (comme pour 'when dans l'exemple).
Un type de données peut être testé dans la régle plutôt qu'une valeur, par exemple dans les lignes ci-dessus contenant "time!".

De plus, une variable peut être définie à une valeur avec le mot **set**.

Comme avec des chaînes de caractères, des alternatives dans les régles peuvent être spécifiées lorsqu'on effectue le parsing de bloc :

```
rule: [some [
    'when set time time! |
    'where set place string! |
    'who set persons [word! | block!]
]]
```

Ces régles permettent de rentrer des informations dans n'importe quel ordre :

```
parse [
    who Fred
    where "Downtown Center"
    when 9:30
] rule
print [time place persons]
```

Cet exemple pourrait employer une affectation classique de variable, mais il illustre comment rendre possible un ordonnancement variable dans le bloc en input.

Ici, un autre exemple qui évalue le résultat du parsing :

```
rule: [
    set count integer!
    set str string!
    (loop count [print str])
]
parse [3 "great job"] rule
parse [3 "hut" 1 "hike"] [some rule]
```

Et finalement, un exemple un peu plus sophistiqué :

```
rule: [
    set action ['buy | 'sell]
    set number integer!
    'shares 'at
    set price money!
    (either action = 'sell [
            print ["income" price * number]
            total: total + (price * number)
        ][
            print ["cost" price * number]
            total: total - (price * number)
        ]
    )
]
total: 0
parse [sell 100 shares at $123.45] rule
print ["total:" total]
total: 0
parse [
    sell 300 shares at $89.08
    buy  100 shares at $120.45
    sell 400 shares at $270.89
] [some rule]
print ["total:" total]
```

10.5 Parsing de sous-blocs

Lors du parsing d'un bloc, si un sous-bloc est trouvé, il est habituellement traité comme une unique valeur de type **block!**.
Pour parser un sous-bloc, vous devez invoquer le parseur de façon récursive sur celui-ci.

Le mot **into** autorise cette capacité de récursivité.
Cela suppose au sein du bloc en input que la valeur suivante à parser soit un sous-bloc . C'est comme si le type de données **block!** était fourni. Si cette valeur suivante à parser n'est pas un bloc, le parsing échoue et **into** cherche des alternatives ou quitte l'analyse par la régle.

Si la valeur à manipuler est bien un bloc, la régle fournie au parseur qui suit immédiatement le mot **into** est utilisée pour commencer le parsing du sous-bloc.

Cette régle est traitée comme une régle secondaire.

```
rule: [date! into [string! time!]]
data: [10-Jan-2000 ["Ukiah" 10:30]]
print parse data rule
```

Toutes les opérations usuelles du parseur peuvent être appliquées à into.

```
rule: [
    set date date!
    set info into [string! time!]]
]
data: [10-Jan-2000 ["Ukiah" 10:30]]
print parse data rule
print info
rule: [date! copy items 2 into [string! time!]]
data: [10-Jan-2000 ["Ukiah" 10:30] ["Rome" 2:45]]
print parse data rule
probe items
```

[Retour au sommaire]

11. Résumé des possibilités de Parsing

11.1 Formes Générales

Opérateur	Description
|	alternative dans une régle
[block]	sous-régle
(paren)	évalue une expression REBOL

11.2 Quantificateurs

Opérateur	Description
none	aucune correspondance
opt	zero ou une fois

some	une ou plusieurs fois
any	zero ou plusieurs fois
12	répéte le modéle 12 fois
1 12	répéte le modéle 1 à 12 fois
0 12	répéte le modéle 0 à 12 fois

11.3 Saut de valeurs

Opérateur	Description
skip	saute un caractère (ou plusieurs si **repeat** est utilisée avec **skip**)
to	avance dans la chaîne en input (le flux) à une valeur ou un type de donnée (datatype)
thru	avance dans la chaîne en input jusqu'à une valeur ou un type de donnée (datatype)

11.4 Récupérer des valeurs

Opérateur	Description
set	attribue la valeur suivante à une variable (set time time!)
copy	copie la correspondance suivante dans une variable

11.5 Utiliser des Mots

Opérateur	Description
word	valeur d'un mot
word:	marque la position actuelle dans la série en input
¦word	Définit la position courante de la série en input

'word	Correspondance littérale avec le mot (parsing d'un bloc)

11.6 Correspondance de valeurs (Parsing de bloc uniquement)

Opérateur	Description
"fred"	Correspondance avec la chaîne "fred"
%data	Correspondance avce le nom du fichier %data
10:30	Correspondance avec l'heure 10:30
1.2.3	Correspondance avec le tuple 1.2.3

11.7 Mots et types de Données (Datatypes)

Mot	Description
type!	correspondance avec n'importe quel datatype comme celui fourni (type)

Annexe 1 - Les valeurs

Ce document est la traduction française de l'Annexe 1 du User Guide de REBOL/Core, qui concerne les Valeurs.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
18 Août 2005 21:02	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Valeurs relatives aux nombres

2.1 Decimal

2.1.1 Concept

Le type de données **decimal!** est basé sur le standard IEEE, c'est à dire des nombres sur 64 bits en virgule flottante. Ils se distinguent des nombres entiers via le séparateur décimal (un point, ou une virgule, peuvent être utilisés pour un usage international, voir les notes ci-dessous).

2.1.2 Format

Les valeurs décimales sont une séquence de chiffres numériques suivis par un point ou une virgule, puis encore d'autres chiffres.

Un signe (+) ou (-) présent immédiatement avant le premier chiffre indique le signe (positif ou négatif). Les zéros présents avant le séparateur des décimales sont ignorés. Les espaces, les points virgules et les points en trop ne sont pas autorisés.

```
1.23
123.
123.0
0.321
0.123
1234.5678
```

Une virgule (elle est la norme dans de nombreux pays) peut être utilisée au lieu d'un point pour représenter le séparateur décimal

```
1,23
0,321
1234,5678
```

Utilisez une apostrophe simple (`) pour séparer les chiffres dans les grands nombres décimaux. Les apostrophes simples peuvent apparaître après le premier chiffre dans le nombre, mais pas avant ce premier chiffre.

```
100'234'562.3782
100'234'562,3782
```

La notation scientifique peut être utilisée pour spécifier l'exposant d'un nombre en ajoutant au nombre la lettre E ou e suivie d'une série de chiffres. L'exposant peut être un nombre positif ou négatif.

```
1.23E10
1.2e007
123.45e-42
56,72E300
-0,34e-12
0.0001e-001
```

Les nombres décimaux s'étendent de 2.2250738585072e-308 jusqu'à 1.7976931348623e+308 et peuvent avoir une précision sur 15 chiffres.

2.1.3 Création

Utilisez la fonction **to-decimal** pour convertir une chaîne de caractère (**string!**), un nombre entier (**integer!**), un bloc (**block!**), ou type de données **decimal!** en nombre décimal :

```
probe to-decimal "123.45"
123.45
probe to-decimal 123
123
probe to-decimal [-123 45]
-1.23E+47
probe to-decimal [123 -45]
1.23E-43
probe to-decimal -123.8
-123.8
probe to-decimal 12.3
12.3
```

Si un nombre décimal et un nombre entier sont combinés dans une expression, le nombre entier est converti en nombre décimal :

```
probe 1.2 + 2
3.2
probe 2 + 1.2
3.2
probe 1.01 > 1
true
probe 1 > 1.01
false
```

2.1.4 Infos connexes

Utilisez **decimal?** pour déterminer si une valeur est bien du type de données **decimal!** :

```
print decimal? 0.123
true
```

Utilisez les fonctions **form**, **print** et **mold** avec un nombre en argument pour afficher la valeur décimale dans sa forme la plus réduite :

- entière s'il est possible de l'afficher ainsi.
- décimal sans exposant s'il n'est pas trop grand ou trop petit.
- en notation scientifique s'il est trop grand ou trop petit.

Par exemple ,

```
probe mold 123.4
123.4
probe form 2222222222222222
2.22222222222222E+15
print 1.00001E+5
100001
```

Les apostrophes simples (quotes) et un signe "plus" (+) précédant le nombre n'apparaissent pas dans l'affichage d'un décimal :

```
print +1'100'200.222'112
1100200.222112
```

2.2 Integer

2.2.1 Concept

Le type de données **integer!** caractérise les nombres positifs et négatifs (et zéro) sur 32 bits. Contrairement aux nombres décimaux, les nombres entiers ne contiennent pas de décimales.

2.2.2 Format

Les valeurs entières consistent en une séquence de chiffres. Un signe plus (+) ou un signe moins (-) placé immédiatement avant le premier chiffre indique le signe. (Il ne peut y avoir d'espace entre le signe et le premier chiffre). Les zéros qui précédent le premier chiffre sont ignorés.

```
0
1234
+1234
-1234
00012
-0123
```

N'utilisez pas de points, ni de virgules dans les entiers. Si une virgule ou un point est trouvé dans un entier, celui-ci sera interprété comme une valeur décimale. D'autre part, vous pouvez utilisez une apostrophe (`) pour séparer les chiffres dans les grands nombres entiers. Les apostrophes peuvent apparaître n'importe où après le premier chiffre dans le nombre, mais pas avant ce premier chiffre.

```
2'147'483'647
```

Les nombres entiers varient entre -2147483648 et 2147483647.

2.2.3 Création

Utilisez la fonction **to-integer** pour convertir un type de données **string!**, **logic!**, **decimal!**, ou **integer!** en un entier :

```
probe to-integer "123"
123
probe to-integer false
0
probe to-integer true
1
probe to-integer 123.4
123
probe to-integer 123.8
```

```
123
probe to-integer -123.8
-123
```

Si un décimal et un entier sont combinés dans une expression, le nombre entier sera converti en nombre décimal :

```
probe 1.2 + 2
3.2
probe 2 + 1.2
3.2
probe 1.01 > 1
true
probe 0 < .001
true
```

2.2.4 Infos connexes

Utilisez **integer?** pour déterminer si une valeur a pour type de données **integer!**.

```
probe integer? -1234
true
```

Utilisez les fonctions **form**, **print** et **mold** avec un argument de type **integer!** pour afficher la valeur entière sous forme d'une chaîne de caractères :

```
probe mold 123
123
probe form 123
123
print 123
123
```

Les entiers qui sont en dehors de l'intervalle décrit plus haut ou qui ne peuvent être représentés sur 32 bits génèrent une erreur.

[Retour au sommaire]

3. Valeurs relatives aux séries

3.1 Binary

3.1.1 Concept

Les valeurs binaires comprennent arbitrairement des données binaires de n'importe quel type. N'importe quelle séquence d'octets peut être stockée, comme une image, un son, un fichier exécutable, des données compressées, et des données cryptées.

3.1.2 Format

Les chaînes binaires sont écrites sous la forme d'un signe (#) dièse, suivi d'une chaîne de caractères incluse entre deux accolades. Les caractères au sein de la chaîne sont encodés dans l'un des divers formats comme l'indique le nombre (optionnel) qui précède le signe (#).

Le codage en base 16 (hexadécimal) est le format par défaut.

```
#{3A18427F 899AEFD8} ; default base-16
2#{10010110110010101001011011001011} ; base-2
64#{LmNvbSA8yw9CB0aGvXmgUkVCu2Uz934b} ; base-64
```

Les espaces, les tabulations et les sauts de lignes sont autorisés dans la chaîne. Une donnée binaire peut s'étendre sur plusieurs lignes.

```
probe #{
    3A
    18
    92
    56
}
#{3A189256}
```

Les chaînes de caractères, pour lesquelles manquent des caractères permettant de retrouver un résultat binaire correct, sont décalées sur la droite (au niveau des bits).

3.1.3 Création

La fonction **to-binary** convertit des données dans le type **binary!**, avec l'encodage existant par défaut dans **system/options/binary-base** :

```
probe to-binary "123"
#{313233}
probe to-binary "today is the day..."
#{746F646179206973207468652064617920E2E2E}
```

Pour convertir un nombre entier en son équivalent en valeur binaire, mettez-le dans un bloc :

```
probe to-binary [1]
#{01}
probe to-binary [11]
#{0B}
```

La conversion d'une série de nombres entiers en binaire se fait par la conversion de chacun des entiers, et en concaténant l'ensemble en une seule valeur binaire :

```
probe to-binary [1 1 1 1]
#{01010101}
```

3.1.4 Infos supplémentaires

L'usage de la fonction **binary?** permet de déterminer si une valeur est de type de données **binary!**.

```
probe binary? #{616263}
true
```

Les valeurs binaires font partie du type **series!** :

```
probe series? #{616263}
true
probe length? #{616263} ; three hex values in this binary
3
```

Les fonctions **enbase** et **debase** sont très importantes dans la manipulation des valeurs de type **binary!**.
La fonction **enbase** convertit des chaînes en leur représentation en base-2, en base 16 ou en base 64. La fonction **debase** transforme des chaînes codées en une valeur binaire, en utilisant la base numérique spécifiée dans **system/options/binary-base**.

3.2 Block

3.2.1 Concept

Les blocs sont des groupes de valeurs et de mots. Les blocs sont utilisés partout, comme depuis un script lui-même jusqu'aux blocs de données et de code fournis dans le script. Les valeurs de type **block** sont indiquées avec des crochets ouvrants et fermants ([ ]) et une quantité variable de datas contenues entre eux.

```
[123 data "hi"]  ; bloc avec des datas
[]               ; bloc vide
```

Les blocs peuvent permettre de manipuler des enregistrements :

```
woodsmen: [
    "Paul" "Bunyan" paul@bunyan.dom
    "Grizzly" "Adams" grizzly@adams.dom
    "Davy" "Crocket" davy@crocket.dom
]
```

Les blocs peuvent contenir du code :

```
[print "this is a segment of code"]
```

Les blocs sont également rattachés au type **series!**, et tout ce qui peut être réalisé avec une série

peut l'être aussi avec un bloc.

Il est possible d'effectuer des recherches dans des blocs :

```
probe copy/part (find woodsmen "Grizzly") 3
[
    "Grizzly" "Adams" grizzly@adams.dom]
```

Les blocs peuvent être modifiés :

```
append woodsmen [
    "John" "Muir" john@muir.dom
]
probe woodsmen
[
    "Paul" "Bunyan" paul@bunyan.dom
    "Grizzly" "Adams" grizzly@adams.dom
    "Davy" "Crocket" davy@crocket.dom
    "John" "Muir" john@muir.dom
]
```

Les blocs peuvent être évalués :

```
blk: [print "data in a block"]
do blk
data in a block
```

Les blocs peuvent contenir d'autres blocs :

```
blks: [
    [print "block one"]
    [print "block two"]
    [print "block three"]
]
foreach blk blks [do blk]
block one
block two
block three
```

3.2.2 Format

Les blocs peuvent contenir un nombre variable de valeurs ou aucune valeur. Ils peuvent s'étendre sur plusieurs lignes et peuvent inclure n'importe quel type de valeurs, y compris d'autres blocs.

Un bloc vide :

```
[ ]
```

Un bloc de nombres entiers :

```
[24 37 108]
```

Un en-tête (header) REBOL :

```
REBOL [
    Title: "Test Script"
    Date: 31-Dec-1998
    Author: "Ima User"
]
```

Un bloc conditionnel et un bloc évalué d'une fonction :

```
while [time < 10:00] [
    print time
    time: time + 0:10
]
```

Les mots dans un bloc ont besoin d'être définis :

```
blk: [undefined words in a block]
probe value? pick blk 1
false
```

Les blocs autorisent n'importe quel nombre de lignes, d'espaces, ou de tabulations. Les lignes et les espaces peuvent être placés n'importe où au sein du bloc, tant qu'ils ne segmentent pas des valeurs.

3.2.3 Création

La fonction **to-block** convertit une data en une valeur de type de données **block!** :

```
probe to-block luke@rebol.com
[luke@rebol.com]
probe to-block {123 10:30 "string" luke@rebol.com}
[123 10:30 "string" luke@rebol.com]
```

3.2.4 Autres informations

Utilisez **block?** pour déterminer si une valeur est ou non du datatype **block!** .

```
probe block? [123 10:30]
true
```

Les blocs étant un sous-ensemble du pseudo type **series!**, vous pouvez utilisez **series?** pour vérifier qu'un bloc est bien de ce type :

```
probe series? [123 10:30]
true
```

L'usage de la fonction **form** sur une valeur de type **block !** produit une chaîne de caractères à partir du contenu du bloc :

```
probe form [123 10:30]
123 10:30
```

L'usage de la fonction **mold** sur une valeur de type block a pour effet, elle, de créer une chaîne de caractères avec le contenu du bloc, comme **form**, mais en permettant que le résultat soit récupérable sous forme d'un bloc REBOL :

```
probe mold [123 10:30]
[123 10:30]
```

Les types de données **hash!** et **list!** sont très similaires au type **block!**. Ils sont utilisables de la même façon que les valeurs de type **block!** mais possèdent des caractéristiques particulières. Les valeurs de type **list!** sont conçues pour permettre des modifications de listes plus rapidement que les valeurs **block!**, et les valeurs de type **hash!** permettent de gérer des tris et l'indexation de données. Ils sont assez commodes lorsqu'on se sert de jeux de données importants.

3.3 Email

3.3.1 Concept

Une adresse email est un type de données. En REBOL, le type de données **email!** exprime facilement ce que représente des adresses email.

```
send luke@rebol.com {some message}

emails: [
    john@keats.dom
    lord@byron.dom
    edger@guest.dom
    alfred@tennyson.dom
]
mesg: {poetry reading at 8:00pm!}
foreach email emails [send email mesg]
```

Le type de données **email!** fait partie aussi du type **series!**, de sorte que les règles s'appliquant aux séries s'appliquent aussi aux valeurs de type **email!** :

```
probe head change/part jane@doe.dom "john" 4
john@doe.dom
```

3.3.2 Format

Le format standard d'une adresse email est le suivant : un nom, suivi du symbole arobase (@), suivi d'un nom de domaine. Une adresse email peut être de n'importe quelle longueur, mais ne doit pas comprendre un seul caractère non autorisé comme les crochets, les apostrophes (quotes), les accolades, les espaces, les caractères de fin de ligne, etc.

Les valeurs suivantes ont un format valide pour le type de données **email!** :

```
info@rebol.com
123@number-mail.org
my-name.here@an.example-domain.com
```

Les caractères minuscules et majuscules sont conservés dans les adresses email.

3.3.3 Raffinements

Deux raffinements peuvent être utilisés avec une valeur **email!** pour récupérer le nom d'utilisateur ou le domaine. Ces raffinements sont :

/user	Récupère le nom d'utilisateur.
/host	Récupère le domaine.

Voici quelques illustrations du fonctionnement de ces raffinements :

```
email: luke@rebol.com
probe email/user
luke
probe email/host
rebol.com
```

3.3.4 Création

La fonction **to-email** transforme une donnée en une valeur de type **email!** :

```
probe to-email "info@rebol.com"
info@rebol.com
probe to-email [info rebol.com]
info@rebol.com
probe to-email [info rebol com]
info@rebol.com
probe to-email [user some long domain name out there dom]
user@some.long.domain.name.out.there.dom
```

3.3.5 Autres Informations

Utilisez la fonction **email?** pour déterminer si une valeur est de type **email!**.

```
probe email? luke@rebol.com
true
```

Le type de donnée **email!** étant un sous-ensemble du type de donnée **series!**, vous pouvez aussi utilisez **series?** pour savoir si la valeur testée est une série :

```
probe series? luke@rebol.com
true
probe pick luke@rebol.com 5
#"@"
```

3.4 File

3.4.1 Concept

Le type de données **file!** peut concerner un nom de fichier, un nom de répertoire ou un chemin complet dans une arborescence (path).

```
%file.txt
%directory/
%directory/path/to/some/file.txt
```

Les valeurs de type **file!** sont aussi des séries et peuvent être manipulées comme telles :

```
probe find %dir/path1/path2/file.txt "path2"
%path2/file.txt
f: %dir/path/file.txt
probe head remove/part (find f "path/") (length? "path/")
%dir/file.txt
```

3.4.2 Format

Les fichiers se caractérisent par un signe "pourcentage" (%) suivi par une suite de caractères :

```
load %image.jpg
prog: load %examples.r
save %this-file.txt "This file has few words."
files: load %../programs/
```

Les caractères "inhabituels" dans les noms de fichiers doivent être encodés avec leurs équivalents hexadécimaux selon la convention en vigueur sur Internet. Un nom de fichier avec un espace (en hexadécimal : %20) devrait ressembler à ceci :

```
probe %cool%20movie%20clip.mpg
%cool%20movie%20clip.mpg
print %cool%20movie%20clip.mpg
cool movie clip.mpg
```

Une autre possibilité est d'englober le nom de fichier entre deux apostrophes :

```
probe %"cool movie clip.mpg"
%cool%20movie%20clip.mpg
print %"cool movie clip.mpg"
cool movie clip.mpg
```

Le caractère standard pour séparer des répertoires dans un chemin est le caractère slash (/), et NON le backslash (\). Cependant, le langage REBOL transforme automatiquement les symboles backslash trouvés dans les noms de fichiers en caractères slash :

```
probe %\some\path\to\some\where\movieclip.mpg
%/some/path/to/some/where/movieclip.mpg
```

3.4.3 Création

La fonction **to-file** transforme une donnée en une valeur de type de données **file!** :

```
probe to-file "testfile"
%testfile
```

Lorsqu'un bloc est passé en argument à **to-file**, les éléments du bloc sont concaténés pour former un chemin de fichier, le dernier élément du bloc servant de nom de fichier :

```
probe to-file [some path to a file the-file.txt]
%some/path/to/a/file/the-file.txt
```

3.4.4 Autres informations

Utilisez la fonction **file?** pour savoir si une valeur est de type **file!** :

```
probe file? %rebol.r
true
```

Comme le type **file!** est un sous-ensemble de **series!**, la fonction **series?** est aussi utilisable :

```
probe series? %rebol.r
true
```

3.5 Hash

3.5.1 Concept

Un hash est un bloc présentant une organisation particulière afin d'y trouver très rapidement des données. Lorsqu'une recherche est réalisée sur une valeur de type **hash!**, celle-ci est réalisée en utilisant une indexation spécifique à la table, qui permet pour de grands blocs, d'accélérer la recherche d'un facteur 100 et plus.

3.5.2 Format

Les blocs de type **hash!** doivent être construits en utilisant la fonction **make** ou la fonction **to-hash**. Il n'y a pas de format spécifique.

3.5.3 Création

Utilisez **make** pour initialiser un hash :

```
hsh: make hash! 10 ; alloue un espace pour 10 élements
```

La fonction **to-hash** transforme une donnée en une valeur de type **hash!**.

Transformer un bloc :

```
blk: [1 "one" 2 "two" 3 "three" 4 "four"]
probe hash: to-hash blk
make hash! [1 "one" 2 "two" 3 "three" 4 "four"]
print select hash 2
two
```

Transformer diverses valeurs :

```
probe to-hash luke@rebol.com

probe to-hash 123.5

probe to-hash {123 10:30 "string" luke@rebol.com}
```

3.5.4 Autres infos

Utilisez **hash?** pour tester le type de données.

```
hsh: to-hash [1 "one" 2 "two" 3 "three" 4 "four"]
probe hash? Hsh
true
```

Le type **hash!** étant un sous-ensemble du type **series!**, la fonction **series?** peut là encore être

utilisée :

```
probe series? hsh
true
```

L'usage de la fonction **form** sur une valeur de type hash renvoie une chaîne de caractères composée avec le contenu du hash :

```
probe form hsh
"1 one 2 two 3 three 4 four"
```

Avec la fonction **mold**, on crée une chaîne de caractère avec la valeur hash et son contenu, mais celle-ci peut être récupérable sous forme d'une valeur de type hash pour REBOL :

```
probe mold hsh
make hash! [1 "one" 2 "two" 3 "three" 4 "four"]
```

3.6 Image

3.6.1 Concept

Le type de données **image!** permet la manipulation des images RGB. Ce datatype est utilisé avec REBOL/View. Les formats d'image supportés sont le GIF, le JPEG, et le BMP. Les images chargées en mémoire peuvent être manipulées comme des séries.

3.6.2 Format

Les images sont en principe chargées à partir d'un fichier. Cependant, elles peuvent être exprimées sous forme de code source, ou aussi fabriquées. L'exemple suivant illustre l'exemple d'une image exprimée sous forme d'un bloc incluant la taille de l'image et les données RGB.

```
image: make image! [192x144 #{
    B34533B44634B44634B54735B7473
    84836B84836B84836BA4837BA4837
    BC4837BC4837BC4837BC4837BC483 ...
}
```

3.6.3 Création

Des images "vides" peuvent être créées avec les fonctions **make** ou **to-image** :

```
empty-img: make image! 300x300
empty-img: to-image 150x300
```

La taille de l'image est fournie.

Les images peuvent aussi être créées avec des copies d'écran ou des objets de type **'face**. Par exemple, en utilisant **make** ou **to-image** :

```
face-shot: make image! face
face-shot: to-image face
```

Utilisez la fonction **load** pour charger en mémoire une image à partir d'un fichier. Si le format de l'image n'est pas supporté, le chargement n'aura pas lieu.

Pour charger en mémoire une image :

```
img: load %bay.jpg
```

3.6.4 Autres informations

Utilisez la fonction **image?** pour savoir si une valeur est ou non du type **image!** :

```
probe image? img
```

Les valeurs de type **image!** sont des séries :

```
probe <b>series?</b> img
```

Le raffinement **/size** renvoie la taille de l'image sous la forme d'une valeur de type **pair!** :

```
probe img/size
```

Les valeurs des pixels d'une image sont obtenus en utilisant la fonction **pick** et modifiées avec **poke**. La valeur retournée par la fonction **pick** est un tuple RGB. (voir plus loin les tuples). La valeur remplacée avec la fonction **poke** doit elle aussi être un tuple RGB.

Récupérer des pixels spécifiques :

```
probe pick img 1

probe pick img 1500
```

Modifier des pixels particuliers :

```
poke img 1 255.255.255
probe pick img 1

poke img 1500 0.0.0
```

```
probe pick img 1500
```

3.7 Issue

3.7.1 Concept

Une valeur de type **issue!** consiste en une série de caractères utilisés pour mettre en forme des choses comme des numéros de téléphones, des numéros de séries ou de modèles, des numéros de cartes de crédits.

Les valeurs de type **issue!** sont un sous-ensemble de celles de type **series!**, et peuvent être manipulées comme telles :

```
probe copy/part find #888-555-1212 "555" 3
#555
```

3.7.2 Format

Les valeurs de type **issue!** commencent avec le signe dièse (#) et continuent avec une série de caractères jusqu'au premier caractère délimiteur (comme l'espace) trouvé.

```
#707-467-8000
#A-0987654321-CD-09876
#1234-5678-4321-8765
#MG82/32-7
```

Les valeurs qui contiennent des caractères délimiteurs devraient être écrites sous forme de chaînes de caractères plutôt que sous forme d'"issues".

3.7.3 Création

La fonction **to-issue** transforme une donnée en une valeur de datatype **issue!**.

```
probe to-issue "1234-56-7890"
#1234-56-7890
```

3.7.4 Autres informations

Utilisez **issue?** pour savoir si une valeur est ou non du type de donnée **issue!**.

```
probe issue? #1234-56-7890
true
```

Comme les valeurs "issues" sont un sous-ensemble de valeurs series, la fonction **series?** s'applique aussi :

```
```

```
probe series? #1234-56-7890
true
```

La fonction **form** renvoie une chaîne de caractères constituée par la séquence de caractères de la valeur **issue!** mais sans le signe (#):

```
probe form #1234-56-7890
1234-56-7890
```

La fonction **mold** s'utilise comme la fonction **form**, elle renvoie une chaîne pouvant être interprétée par REBOL comme une valeur **issue!** :

```
probe mold #1234-56-7890
#1234-56-7890
```

La fonction **print** permet d'afficher une valeur de type **issue!** après un "**reform**" sur celle-ci :

```
print #1234-56-7890
1234-56-7890
```

3.8 List

3.8.1 Concept

Les listes sont des énumérations sous formes de blocs qui permettent des ajouts et des suppressions très efficaces en terme de performances. Elles peuvent être utilisables dans les cas où de nombreuses modifications sont à effectuer sur de grands volumes de données (grands blocs).

3.8.2 Format

Les listes doivent être construites en utilisant la fonction **make** ou la fonction **to-list**. Il n'y a pas de format lexical spécifique.

Les valeurs de type **list!** ne sont pas des subsituts aux valeurs **block!**. Il existe quelques différences entre les blocs et les listes :

L'insertion dans une liste modifie son index, que se positionne juste après le point d'insertion. La suppression d'un élément référencé dans une liste conduit à remettre à jour la position finale de la liste (tail).

Les exemples suivants montre la différence de comportement entre l'insertion dans un bloc et dans une liste.

Initialisation d'un bloc et d'une liste :

```
blk: [1 2 3]

lst: to-list [1 2 3]
```

Insertion dans un bloc et dans une liste :

```
insert blk 0

insert lst 0
```

Regardez le mot dans le bloc et dans la liste juste après l'insertion. Remarquez que le bloc blk pointe sur la tête du bloc (**head**), comme avant l'insertion de la valeur 0, mais la liste lst pointe sur la position juste *après* le point d'insertion :

```
print blk
0 1 2 3
print lst
1 2 3
print head lst
0 1 2 3
```

Les exemples suivants montrent la différence entre un bloc et une liste, lors de la suppression d'un élément.

Initialisation du bloc et de la liste :

```
blk: [1 2 3]

lst: to-list [1 2 3]
```

Suppression au sein du bloc et de la liste :

```
remove blk

remove lst
```

Si on regarde l'état du bloc et de la liste, on constate qu'à présent la liste lst pointe sur la fin de la série (**tail**) :

```
print blk
2 3
print tail? lst
true
print head lst
2 3
```

Si vous ne voulez pas pointer sur la fin de la série, après avoir supprimé une valeur, il convient de vous déplacer dans la série et d'enlever la valeur après l'index courant. L'exemple suivant illustre cela :

Initialisation de la liste :

```
lst: to-list [1 2 3]
```

Déplacement dans la série et suppression de la valeur après l'index courant :

```
remove back (lst: next lst)
```

Si on regarde à l'endroit où la valeur a été supprimée :

```
probe lst
make list! [2 3]
print tail? lst
false
```

3.8.3 Création

Il faut utiliser la fonction **make** pour initialiser une valeur de type **list!** :

```
lst: make list! 10 ; alloue de l'espace pour 10 éléments
```

La fonction **to-list** convertit son argument en une valeur de type de donnée **list!** :

Pour transformer un bloc :

```
blk: [1 "one" 2 "two" 3 "three" 4 "four"]
probe to-list blk
make list! [1 "one" 2 "two" 3 "three" 4 "four"]
```

3.8.4 Informations complémentaires

Utilisez **list?** pour déterminer si une valeur est ou non du type de données **list!**.

```
lst: to-list [1 "one" 2 "two" 3 "three" 4 "four"]
probe list? Lst
true
```

Puisque les listes sont aussi des séries, la fonction **series?** est aussi utile :

```
probe series? lst
true
```

L'usage de la fonction **form** sur une liste génère une chaîne de caractères composée à partir du contenu de la liste :

```
probe form lst
"1 one 2 two 3 three 4 four"
```

Avec la fonction **mold**, c'est presque comme pour **form**, si ce n'est que le résultat peut être immédiatement récupérable par REBOL sous forme d'une valeur **list!** :

```
probe mold lst
make list! [1 "one" 2 "two" 3 "three" 4 "four"]
```

3.9 Paren

3.9.1 Concept

Une valeur de type **paren!** est un bloc (Ndt : avec usage de **paren**thèses) qui est immédiatement évalué. Cette valeur est en tous points identique à un bloc, à l'exception du fait qu'elle est évaluée lorsqu'elle est rencontrée, et que le résultat de l'évaluation est retourné. Lorsqu'on l'utilise à l'intérieur d'une expression à évaluer, une valeur de type **paren!** permet de contrôler l'ordre de l'évaluation.

```
print 1 + (2 * 3)
7
print 1 + 2 * 3
9
```

Comme pour un bloc, une valeur de type **paren!** peut être modifiée. Cependant, si on souhaite faire référence à cette valeur **paren!**, des précautions doivent être prises pour éviter qu'elle soit évaluée.

Si vous stockez une valeur **paren!** dans une variable, vous devrez utilisez la forme **get-word** (c'est-à-dire **:word**) pour prévenir une éventuelle évaluation. Les "parens" étant un sous-ensemble des séries, ce qui peut être fait avec une série est aussi possible avec une valeur **paren!**.

```
paren: first [(1 + 2 * 3 / 4)]
(1 + 2 * 3 / 4)
print type? :paren
paren
print length :paren
7
print first :paren
1
print last :paren
4
insert :paren [10 + 5 *]
probe :paren
(10 + 5 * 1 + 2 * 3 / 4)
print paren
12.75
```

3.9.2 Format

Les valeurs de type **paren!** sont identifiées par des parenthèses ouvrantes et fermantes. Elles peuvent s'étendre sur plusieurs lignes et contenir n'importe quelle donnée, dont d'autres valeurs **paren!**.

3.9.3 Création

Le fonction **make** peut être utilisée pour définir une valeur **paren!** :

```
paren: make paren! 10
insert :paren 10
insert :paren `+
insert :paren 20

print :paren
20 + 10
print paren
30
```

La fonction **to-paren** transforme une donnée en une valeur de datatype **paren!** :

```
probe to-paren "123 456"
(123 456)
probe to-paren [123 456]
(123 456)
```

3.9.4 En plus

Utilisez **paren?** pour tester le type de la donnée.

```
blk: [(3 + 3)]
probe pick blk 1
(3 + 3)
probe paren? pick blk 1
true
```

Comme les *parens* sont un sous-ensemble de celui des séries, il est possible d'utiliser **series?** :

```
probe series? pick blk 1
true
```

L'utilisation de **form** sur une valeur de type **paren!** crée une chaîne de caractères à partir du contenu de cette valeur :

```
```

```
probe form pick blk 1
3 + 3
```

3.10 Path

3.10.1 Concept

Les paths (**NdT** : on pourrait traduire par "chemins", l'idée générale étant d'avoir une valeur en REBOL qui traduit l'idée de se déplacer dans une arborescence, ou de préciser une particularité d'une fonction, par exemple) sont un ensemble de mots et de valeurs délimités par des slashs (/).

Les paths sont utilisés pour naviguer vers quelque chose, préciser ou trouver quelque chose.

Les mots et les valeurs d'un path sont appelés des raffinements, et ils sont assemblés pour donner une direction, un sens de "navigation" au travers d'une valeur ou d'une fonction. Les paths sont utilisés avec les blocs, les fichiers, les chaînes de caractères, les listes, les hashs, les fonctions, et les objets.

La façon dont les paths agissent dépend du type de données avec lesquels ils sont utilisés.

Les paths peuvent être utilisés pour sélectionner des valeurs dans des blocs, ou des caractères dans une chaîne, accéder à des variables dans des objets, donner un comportement particulier à une fonction :

Type de path	Action
USA/CA/Ukiah/size	sélection dans un bloc
names/12	position dans une chaîne
account/balance	fonction dans un objet
match/any	raffinement d'une fonction

L'exemple ci-dessous montre la simplicité d'usage qu'offre un path pour accéder aux informations d'une mini base de données créées à partir de quelques blocs :

```
towns: [
    Hopland [
    phone #555-1234
    web   http://www.hopland.ca.gov
    ]

    Ukiah [
    phone #555-4321
    web   http://www.ukiah.com
    email info@ukiah.com
    ]
]
```

```
print towns/ukiah/web
http://www.ukiah.com
```

Résumé des concepts de paths :

Type de path	Type de mot	Type de Test	Fonction de conversion
path/word:	set-path!	set-path?	to-set-path
path/word	path!	path?	to-path
'path/word	lit-path!	lit-path?	to-lit-path

Exemples de paths :

Pour évaluer une fonction (une méthode) liée à un objet :

```
obj: make object! [
    hello: func [] [print "hello! hello!"]
]
obj/hello
hello! hello!
```

Pour évaluer un attribut d'un objet :

```
obj: make object! [
    text: "do you believe in magic?"
]
probe obj/text
do you believe in magic?
```

Des raffinements de fonction :

```
hello: func [/again] [
    print either again ["hello again!"]["hello"]
]
hello/again
hello again!
```

Pour effectuer une sélection à partir de blocs :

```
USA: [
    CA [
    Ukiah [
```

```
        population 15050
        elevation [610 feet]
    ]
    Willits [
        population 5073
        elevation [1350 feet]
    ]
    ]
]
print USA/CA/Ukiah/population
15050
print form USA/CA/Willits/elevation
1350 feet
```

Pour récupérer des élements à partir d'une série, via leur position numérique dans celle-ci :

```
string-series: "abcdefg"
block-series: ["John" 21 "Jake" 32 "Jackson" 43 "Joe" 52]
block-with-sub-series: [ "abc" [4 5 6 [7 8 9]]]
probe string-series/4
#"d"
probe block-series/3
Jake
probe block-series/6
43
probe block-with-sub-series/1/2
#"b"
probe block-with-sub-series/2/2
5
probe block-with-sub-series/2/4/2
8
```

Les mots fournis en tant que paths sont symboliques, et par conséquent ne sont pas évalués. Ceci permet de construire la forme la plus intuitive pour référencer un objet.

Pour utiliser la référence au mot, une référence explicite à la valeur du mot est requise :

```
city: 'Ukiah
probe USA/CA/:city     ; << :city fait ici référence à 'Ukiah
[
    population 15050
    elevation "610 feet"
]
```

Les chemins dans les blocs, les hashs, ou les objets sont évalués en utilisant la correspondance avec le mot au plus haut niveau dans le path, et en vérifiant si le mot est de type **block!**, **hash!** ou **object!**. Ensuite le mot suivant est recherché en tant que mot présent dans le bloc, le hash ou l'objet, et une sélection se fait. La valeur qui suit le mot recherché est renvoyée. Quand la valeur retournée est un bloc, un hash, ou un objet, le path peut être exprimé ainsi :

Récupération de la valeur associée au mot CA dans USA :

```
```

```
probe USA/CA
[
    Ukiah [
    population 15050
    elevation "610 feet"
    ]
    Willits [
    population 9935
    elevation "1350 feet"
    ]
]
```

Récupération de la valeur associée avec Willits dans USA/CA :

```
probe USA/CA/Willits
[
    population 9935
    elevation "1350 feet"
]
```

Récupération de la valeur associée au mot "population" dans USA/CA/Willits :

```
probe USA/CA/Willits/population
9935
```

Quand un mot utilisé dans un path n'existe pas à un endroit donné de la stucture, une erreur se produit :

```
probe USA/CA/Mendocino
** Script Error: Invalid path value: Mendocino.
** Where: probe USA/CA/Mendocino
```

Les paths peuvent être utilisés pour modifier des valeurs dans des blocs et des objets :

```
USA/CA/Willits/elevation: "1 foot, after the earthquake"
probe USA/CA/Willits
[
    population 9935
    elevation "1 foot, after the earthquake"
]
obj/text: "yes, I do believe in magic."
probe obj
make object! [
    text: "yes, I do believe in magic."
]
```

Les blocs, les hashs, les fonctions et les objets peuvent être mélangés dans les paths.

Pour choisir des éléments dans un bloc, lui-même dans un objet :

```
obj: make object! [
    USA: [
    CA [
        population "too many"
    ]
    ]
]
probe obj/USA/CA/population
too many
```

Utilisation de raffinements de fonctions au sein d'un objet :

```
obj: make object! [
    hello: func [/again] [
    print either again [
        "hello again"
    ] [
        "oh, hello"
    ]
    ]
]
obj/hello/again
hello again
```

Les paths sont aussi des séries, de sorte que ce qui peut être réalisé avec une série peut aussi l'être avec une valeur de type **path!** :

```
root: [sub1 [sub2 [
    word "a word at the end of the path"
    num 55
]   ]   ]
path: 'root/sub1/sub2/word
probe :path
root/sub1/sub2/word
```

Dans l'exemple précédent, la notation :path a été utilisée pour récupérer le path lui-même, et non sa valeur :

```
probe path
a word at the end of the path
```

Pour connaître la longueur d'un path :

```
probe length? :path
4
```

Trouver un mot à l'intérieur d'un path :

```
probe find :path 'sub2
sub2/word
```

Modifier un mot dans un path :

```
change find :path 'word 'num
probe :path
root/sub1/sub2/num
probe path
55
```

3.10.2 Format

Les paths sont exprimés relativement à un mot-racine, en fournissant un certain nombre de raffinements, chacun étant séparé des autres par le symbole slash (/).

Ces raffinements peuvent être des mots ou des valeurs. Leur interprétation particulière dépend du type de données du mot situé à la racine.

Les mots fournis comme raffinement dans les paths sont symboliques et ne sont pas évalués. Ceci est nécessaire pour garder une forme de référencement assez intuitive. Pour utilisez la référence d'un mot, une référence explicite est nécessaire comme vu précédemment :

```
root/:word
```

Cet exemple utilise la valeur de la variable, plutôt que son nom.

3.10.3 Création

Vous pouvez créer un path vierge d'une taille donnée comme dans l'exemple suivant :

```
path: make path! 10
insert :path `test
insert tail :path `this
print :path
test/this
```

La fonction **to-path** transforme son argument en une valeur de type **path!**.

```
probe to-path [root sub]
root/sub
probe to-path "root sub"
root/sub
```

La fonction **to-set-path** convertit des valeurs en type **set-word** (mot défini) :

```
probe to-set-path "root sub"
root/sub:
```

La fonction **to-lit-path** convertit des valeurs en type **lit-word** (mot litéral):

```
probe to-lit-path "root sub"
'root/sub
```

3.10.4 Informations complémentaires

Utilisez **path?**, **set-path?**, et **lit-path?** pour déterminer le type de données d'une valeur.

```
probe path? second [1 two "3"]
false
blk: [sub1 [sub2 [word 1]]]
blk2: [blk/sub1/sub2/word: 2]
if set-path? (pick blk2 1) [print "it is set"]
it is set
probe lit-path? first ['root/sub]
true
```

Comme les paths sont un sous-ensemble du pseudo type **series!**, la fonction **series?** peut également être utilisée :

```
probe series? pick [root/sub] 1
true
```

L'application de la fonction **form** sur un path génére une chaîne de caractères à partir du path :

```
probe form pick [root/sub] 1
root/sub
```

Avec la fonction **mold**, on crée aussi une chaîne de caractère comme pour **form**, mais celle-ci peut être rechargée sous forme d'un path REBOL :

```
probe mold pick [root/sub] 1
root/sub
```

3.11 String

3.11.1 Concept

Les chaînes (strings) sont des séries de caractères. Toutes les opérations possibles sur les valeurs de type série peuvent également être faites avec des chaînes de caractère.

3.11.2 Format

Les valeurs de type **string!** se présentent sous la forme d'une séquence de caractères entourés par des apostrophes " " ou des accolades {}. Les chaînes incluses entre des apostrophes sont limitées à une unique ligne, et ne peuvent contenir certains caractères non imprimables.

```
"This is a short string of characters."
```

Les chaînes comprises entre accolades sont utilisées pour de grands morceaux de texte, pouvant s'étaler sur plusieurs lignes. Tous les caractères de la chaîne, comme les espaces, les tabulations, les apostrophes, et les sauts de ligne font partie de la chaîne.

```
{This is a long string of text that will
not easily fit on a single line of source.
These are often used for documentation
purposes.}
```

Les accolades sont comptées dans la chaîne, de sorte qu'une chaîne de caractères peut inclure d'autres accolades à condition que le nombre d'accolades ouvrantes soit identique au nombre d'accolades fermantes.

```
{
This is another long string of text that would
never fit on a single line. This string also
includes braces { a few layers deep { and is
valid because there are as many closing braces }
as there are open braces } in the string.
}
```

Vous pouvez inclure des caractères spéciaux et effectuer certaines opérations dans les chaînes de caractères en préfixant ces caractères spéciaux par le symbole (^) :

Inclusion de caractères spéciaux :

Caractère	Définition
^"	Insère une double quote (").
^}	Insère une accolade fermante (}).
^^	Insère le symbole *caret* (^).

^/	Insère le symbole slash
^(line)	Débute une nouvelle ligne.
^-	Insère une tabulation.
^(tab)	Insère aussi une tabulation.
^(page)	Démarre une nouvelle page
^(back)	Efface un caractère à gauche du point d'insertion.
^(null)	Insère le caractère "null".
^(escape)	Insère le caractère "escape".
^(letter)	Insère un caractère de contrôle (A-Z).
^(xx)	Insère un caractère ASCII via son équivalent hexadécimal (xx). Ce format permettra une extension pour les caractères Unicode dans le futur.

3.11.3 Création

Utilisez **make** pour allouer une quantité d'espace mémoire pour une chaîne de caractères vide :

```
make string! 40'000 ; espace pour 40k (caractères)
```

La fonction **to-string** transforme le type de la donnée fournie en argument, en type de donnée **string!**.

```
probe to-string 29-2-2000
"29-Feb-2000"
probe to-string 123456.789
"123456.789"
probe to-string #888-555-2341
"888-555-2341"
```

Transformer un bloc de données en une chaîne, avec la fonction **to-string**, a pour conséquence de concaténer les élements du bloc, mais sans les évaluer :

```
probe to-string [123 456]
"123456"
probe to-string [225.225.225.0 none true 'word]
```

```
"225.225.225.0nonetrueword"
```

3.11.4 Informations connexes

Utilisez **string?** ou **series?** pour déterminer si une valeur est de type **string!** :

```
print string? "123"
true
print series? "123"
true
```

Les fonctions **form** et **mold** sont fortement corrélées aux chaînes, dans le sens où elles génèrent des chaînes de caractères à partir de données d'autres types. La fonction **form** permet l'obtention d'une forme humainement lisible d'un type de donnée spécifique, tandis que la fonction **mold** fabrique une version utilisable par le langage.

```
probe form "111 222 333"
"111 222 333"
probe mold "111 222 333"
{"111 222 333"}
```

3.12 Tag

3.12.1 Concept

Les tags (balises) sont utilisés dans les langages à balises pour indiquer comment des zones de texte doivent être traitées. Par exemple, le tag <HTML> au début d'un fichier indique qu'il devrait être analysé au moyen des régles du langage HTML.

Une balise avec un slash (/), tel que </HTML> indique la fermeture du tag.

Les tags, en tant que sous-ensemble des séries peuvent être manipulés comme tels :

```
a-tag: <img src="mypic.jpg">
probe a-tag
<img src="mypic.jpg">
append a-tag { alt="My Picture!"}
probe a-tag
<img src="mypic.jpg" alt="My Picture!">
```

3.12.2 Format

Les tags valides commencent par le symbole (<) et se termine avec (>).

Par exemple :

```
<a href="index.html">
<img src="mypic.jpg" width="150" height="200">
```

3.12.3 Création

La fonction **to-tag** convertit une donnée en une autre de type **tag!** :

```
probe to-tag "title"
<title>
```

Vous pouvez utiliser la fonction **build-tag** pour construire des tags, et y inclure des attributs. La fonction **build-tag** prend un seul argument, un bloc. Dans ce bloc, le premier mot est utilisé pour nommer le tag, et les mots suivants servent à définir des paires d'attributs :

```
probe build-tag [a href http://www.rebol.com/]
<a href="http://www.rebol.com/">
probe build-tag [
    img src %mypic.jpg width 150 alt "My Picture!"
]
<img src="mypic.jpg" width="150" alt="My Picture!">
```

3.12.4 Autres infos

L'usage de la fonction **tag?** permet de déterminer si une valeur est bien de type de données **tag!** .

```
probe tag? <a href="http://www.rebol.com/">
true
```

Comme les tags sont aussi des séries, la fonction **series?** peut être appelée :

```
probe series? <a href="http://www.rebol.com/">
true
```

La fonction **form** renvoie le tag qui lui est fourni, sous la forme d'une chaîne :

```
probe form <a href="http://www.rebol.com/">
{<a href="http://www.rebol.com/">}
```

La fonction **mold** renvoie elle aussi une chaîne :

```
probe mold <a href="http://www.rebol.com/">
{<a href="http://www.rebol.com/">}
```

La fonction **print** affiche un tag sur la sortie standard :

```
print <a href="http://www.rebol.com/">
<a href="http://www.rebol.com/">
```

3.13 URL

3.13.1 Concept

Le terme "URL" est l'acronyme de Uniform Ressource Locator, un standard Internet utilisé pour accéder à des ressources comme des pages Web, des images, des fichiers, et du courrier électronique au travers du réseau.

Le type d'URL la plus connue est celle utilisée pour le Web comme http://www.rebol.com. Les valeurs URLs sont des séries, et peuvent être manipulées comme telles :

```
url: http://www.rebol.com/reboldoc.html
probe to-file find/reverse (tail url) "rebol"
%reboldoc.html
```

3.13.2 Format

La première partie d'un URL indique son protocole de communication, appelé le "scheme". Le langage REBOL supporte plusieurs types de protocoles, pour les pages web (HTTP:), le transfert de fichier (FTP:), les newsgroups (NNTP:), le courrier électronique (MAILTO:), les fichiers (FILE:), finger (FINGER:), whois (WHOIS:), daytime (DAYTIME:), post office (POP:), le protocole de transmission (TCP:) et celui de résolution des noms de domaines (DNS:). Ces noms de protocoles, ces "schemes" sont suivis dans l'URL par une série de caractères dépend ant du protocole utilisé :

```
http://host.dom/path/file
ftp://host.dom/path/file
nntp://news.some-isp.net/some.news.group
mailto:name@domain
file://host/path/file
finger://user@host.dom
whois://rebol@rs.internic.net
daytime://everest.cclabs.missouri.edu
pop://user:passwd@host.dom/
tcp://host.dom:21
dns://host.dom
```

Certains champs sont optionnels. Par exemple, le nom d'hôte peut être suivi par un numéro de port, si ce dernier est différent de celui par défaut. Une URL FTP peut aussi comprendre un mot de passe :

```
ftp://user:password@host.dom/path/file
```

Les caractères dans l'URL doivent respectés les conventions Internet. Certains caractères doivent être encodés en hexadécimal, en les faisant précéder du caractère d'échappement "%" :

```
probe http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt
http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt
print http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt
http://www.somesite.dom/odd(dir)/odd{file}.txt
```

3.13.3 Création

La fonction **to-url** transforme un bloc en une valeur de type **url!**, le premier élément dans le bloc est le "scheme", le second élément est le domaine (avec ou sans user:pass et le port), les éléments suivants sont le path et le nom du fichier.

```
probe to-url [http www.rebol.com reboldoc.html]
http://www.rebol.com/reboldoc.html
probe to-url [http www.rebol.com %examples "websend.r"]
http://www.rebol.com/examples/websend.r
probe to-url [http usr:pass@host.com:80 "(path)" %index.html]
http://usr:pass@host.com:80/%28path%29/index.html
```

3.13.4 Autres infos

La fonction **url?** permet de tester si le type de données est bien **url!**.

```
probe url? ftp://ftp.rebol.com/
true
```

La fonction **series?** est également utilisable pour vérifier le type de données.

```
probe series? http://www.rebol.com/
true
```

[Retour au sommaire]

4. Autres valeurs

4.1 Character

4.1.1 Concept

Les caractères ne sont pas des chaînes (strings); ce sont les valeurs unitaires à partir desquelles les chaînes sont construites. Un caractère peut être imprimable, non imprimable, ou encore être un caractère de contrôle.

4.1.2 Format

Une valeur **char!** est écrite avec le signe dièse (#) suivi par une chaîne comprise entre deux apostrophes.

Le signe (#) est nécessaire pour distinguer un caractère d'une chaîne :

- #"R" ; le caractère : R
- "R" ; une chaîne de caractère a un seul caractère : R

Les caractères peuvent inclure des séquences d'échappement, qui commence avec le symbole (^) et sont suivis par un ou plusieurs caractères. Cette codification va inclure les caractères #"^A" to #"^Z", c'est-à-dire du "contrôle A" au "contrôle Z" (majuscule et minuscule sont identiques).

```
#"^A" #"^Z"
```

De plus, si des parenthèses sont utilisées au sein du caractère, elles signifient qu'il s'agit d'une valeur spéciale. Par exemple, le caractère "nul" peut être écrit ainsi :

```
"^@"
"^(null)"
"^(00)"
```

La dernière ligne est écrite en format hexadécimale (base 16). Les parenthèses autour de la valeur permettent de prévoir l'extension en Unicode (16 bits) du type char! dans le futur.

La table ci-dessous présente les caractères de contrôle pouvant être utilisés avec REBOL :

Character	Definition
#"(null)" or #"@"	nul (zero)
#"(line)", #"/" or, #"."	fin de ligne
#"(tab)" or #"-"	tabulation horizontale
#"(page)"	nouvelle page (et page eject)
#"(esc)"	escape
#"(back)"	retour arrière (backspace)
#"(del)"	delete
#"^"	caractére "caret"
#"^""	apostrophe
#"(00)" to #"(FF)"	formes hexa des caractères

4.1.3 Création

Les caractères peuvent être convertis depuis et vers d'autres types de données avec la fonction **to-char** :

```
probe to-char "a"
#"a"
probe to-char "z"
#"z"
```

Les caractères suivent le standard ASCII, et peuvent être construits en spécifiant leurs équivalents ASCII :

```
probe to-char 65
#"A"
probe to-char 52
#"4"
probe to-char 52.3
#"4"
```

Une autre méthode pour obtenir un caractère est de récupérer le premier caractère d'une chaîne :

```
probe first "ABC"
#"A"
```

Alors que les caractères dans les chaînes ne sont sensibles à la casse, ils le deviennent lorsqu'ils sont sous forme de caractères individuels :

```
probe "a" = "A"    ; insensibles à la casse : MAJ = min
true
probe #"a" = #"A"    ; sensibles à la casse :  MAJ <> min
false
```

Cependant, dans la plupart des fonctions, lorsqu'ils sont utilisés, la comparaison n'est pas sensible à la casse à moins de spécifier cette option. Par exemple :

```
select [#"A" 1] #"a"
1
select/case [#"A" 1] #"a"
none
find "abcde" #"B"
"bcde"
find/case "abcde" #"B"
none
switch #"A" [#"a" [print true]]
true
```

4.1.4 Autres informations

Utilisez **char?** pour déterminer si une valeur est de type **char!**.

```
probe char? "a"
false
probe char? #"a"
true
```

La fonction **form** retourne le caractère sans le symbole (#).

```
probe form #"A"
"A"
```

La fonction **mold** renvoie le caractère avec le signe (#), les apostrophes (double quotes), et aussi les séquences d'échappement pour les caractères qui le nécessitent :

```
probe mold #"A"
{#"A"}
```

4.2 Date

4.2.1 Concept

Tout autour de la planète, les dates sont écrites dans divers formats. Malgré tout, la plupart des pays utilisent le format jour-mois-année (JJ-MM-AA). L'une des exceptions à cela sont les Etats-Unis, qui utilisent couramment le format mois-jour-année (MM-JJ-AA).
Par exemple, une date écrite sous la forme 2/1/1999 est ambiguë. Le mois pourrait être interprété soit comme février soit comme janvier. Certains pays utilisent le trait d'union (-) comme séparateur, d'autres le symbole slash (/), et encore d'autres utilisent le point (.).

Et pour finir, les ordinateurs personnels utilisent souvent des dates dans le format ISO année-mois-jour (AA-MM-JJ).

4.2.2 Format

Le langage REBOL est flexible, et il permet au type de données date! d'être exprimé dans des formats divers. Par exemple, le premier jour de mars peut s'écrire sous l'un ou l'autre des formats suivants :

```
probe 1/3/1999
1-Mar-1999
probe 1-3-1999
1-Mar-1999
probe 1999-3-1  ;ISO format
1-Mar-1999
```

L'année peut aller de 1 jusqu'à 9999. (**NdT** : si quelqu'un veut bien patienter jusque là pour voir s'il

n'y a pas un bug ;-)).
Les jours relatifs aux années bissextiles (le 29 février) n'existent que pour ces années-là.

```
probe 29-2-2000
29-Feb-2000
probe 29-2-2004
29-Feb-2004
probe 29-2-2002
** Syntax Error: Invalid date -- 29-2-2002
** Near: (line 1) probe 29-2-2002
```

Les champs des dates peuvent être séparés avec le symbole "slash" (/) ou "tiret" (-). Les dates peuvent être écrites soit au format "année-mois-jour", soit au format "jour-mois-année" :

```
probe 1999-10-5
5-Oct-1999
probe 1999/10/5
5-Oct-1999
probe 5-10-1999
5-Oct-1999
probe 5/10/1999
5-Oct-1999
```

Parce que les formats de date internationaux ne sont pas beaucoup utilisés aux Etats-Unis, un nom de mois ou son abréviation peuvent aussi être utilisés :

```
probe 5/Oct/1999
5-Oct-1999
probe 5-October-1999
5-Oct-1999
probe 1999/oct/5
5-Oct-1999
```

Quand l'année correspond au dernier champ, elle peut écrite avec 2 ou 4 chiffres :

```
probe 5/oct/99
5-Oct-1999
probe 5/oct/1999
5-Oct-1999
```

Cependant, il est préférable d'écrire l'année sur 4 chiffres. En effet, des problèmes peuvent survenir sinon, lors de comparaison de dates ou en effectuant des opérations de tris.

Quand deux chiffres sont utilisés pour exprimer l'année, l'interprétation de celle-ci est relative à l'année en cours et est uniquement valide pour une période de 50 ans dans le passé ou dans le futur.

```
probe 28-2-66   ; fait référence à 1966
```

```
28-Feb-1966
probe 12-Mar-20 ; fait référence à 2020
12-Mar-2020
probe 11-3-45   ; fait référence à 2045, pas à 1945
11-Mar-2045
```

Il est recommandé d'utiliser une année sur 4 chiffres afin d'éviter les problèmes potentiels. Pour représenter les dates dans le premier siècle (rarement fait puisque le calendrier Grégorien n'existait pas), utilisez des zéros supplémentaires pour représenter le siècle (comme dans 9-4-0029).

L'heure est séparée de la date par le symbole slash (/). Le fuseau horaire est ajouté ensuite en utilisant les signes (+) ou (-) sans aucun espace. Les fuseaux horaires sont écrits sous forme de décalage horaire (plus ou moins) relativement à l'heure GMT. La résolution pour le fuseau horaire est d'une demi-heure. Si le décalage horaire est un nombre entier, on suppose qu'il s'agit d'heures :

```
probe 4/Apr/2000/6:00+8:00
4-Apr-2000/6:00+8:00
probe 1999-10-2/2:00-4:00
2-Oct-1999/2:00-4:00
probe 1/1/1990/12:20:25-6
1-Jan-1990/12:20:25
```

Il ne peut y avoir d'espaces au sein d'une date. Par exemple, l'expression

```
10 - 5 - 99
```

sera être interprétée comme une soustraction, pas une date.

4.2.3 Accéder aux données d'une variable date!

Quelques raffinements peuvent être utilisés avec une valeur de type **date!** pour récupérer l'un ou l'autre des champs prédéfinis :

Raffinement	Description
/day	retourne le jour.
/month	retourne le mois.
/year	retourne l'année.
/julian	retourne le jour de l'année.
/weekday	renvoie le jour dans la semaine (1-7/Mon-Sun).

/time	retourne l'heure (si elle existe).
/hour	retourne l'heure (si elle existe).
/minute	retourne les minutes (si cette information existe).
/second	renvoie les secondes (si cette information existe).
/zone	renvoie le fuseau horaire (s'il existe).

Voici comment ces raffinements fonctionnent :

```
some-date: 29-Feb-2000
probe some-date/day
29
probe some-date/month
2
probe some-date/year
2000
days: ["Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"]
probe pick days some-date/weekday
Tue
```

Quand l'heure est fournie, les raffinements relatifs à l'heure peuvent être utilisés. Les raffinements **/hour**, **/minute** et **/second** peuvent être utilisés avec le raffinement **/time** qui isole les données horaires, afin de les manipuler :

```
lost-time: 29-Feb-2000/11:33:22.14-8:00
probe lost-time/time
11:33:22.14
probe lost-time/time/hour
11
probe lost-time/time/minute
33
probe lost-time/time/second
22.14
probe lost-time/zone
-8:00
```

4.2.4 Création

Utilisez la fonction **to-date** pour convertir des valeurs en dates :

```
probe to-date "5-10-1999"
5-Oct-1999
probe to-date "5 10 1999 10:30"
5-Oct-1999/10:30
```

```
probe to-date [1999 10 5]
5-Oct-1999
probe to-date [5 10 1999 10:30 -8:00]
5-Oct-1999/10:30-8:00
```

> Pour la conversion vers une date, l'année DOIT être spécifiée avec 4 chiffres.

Diverses opérations mathématiques peuvent être réalisées sur les dates :

```
probe 5-Oct-1999 + 1
6-Oct-1999
probe 5-10-1999 - 10
25-Sep-1999
probe 5-Oct-1999/23:00 + 5:00
6-Oct-1999/4:00
```

4.2.5 Autres informations

Utilisez la fonction **date?** pour déterminer si une valeur appartient à ce type de données.

```
probe date? 5/1/1999
true
```

La fonction **to-idate** renvoie la date sous forme d'une chaîne au format standard en vigueur sur Internet.

Le format de date pour Internet est : jour de la semaine, jour dans le mois, mois, année, l'heure (sur 24 heures), et le décalage horaire vis-à-vis de l'heure GMT, soit :

```
probe to-idate now
Fri, 30 Jun 2000 14:42:26 -0700
```

La fonction **now** renvoie la date et l'heure courante (au format long incluant le décalage horaire) :

```
probe now
30-Jun-2000/14:42:26-7:00
```

4.3 Logic

4.3.1 Concept

Le type de données **logic!** permet de représenter deux états : **true** ou **false**, c'est-à-dire : vrai ou faux.

Ce type de donnée est souvent utilisé pour des comparaison comme avec :

```
age: 100
probe age = 100
true
time: 10:31:00
probe time < 10:30
false
str: "this is a string"
probe (length? str) > 10
true
```

Le type de données **logic!** est couramment utilisé avec des fonctions conditionnelles telles que **if**, **while**, et **until** :

```
if age = 100 [print "Centennial human"]
Centennial human
while [time > 6:30] [
    send person "Wake up!"
    wait [0:10]
]
```

Le complément d'une valeur logique est obtenue avec la fonction **not** :

```
there: place = "Ukiah"
if not there [...]
```

4.3.2 Format

Normalement, les valeurs logiques sont récupérées à partir de la comparaison d'expression. Cependant, des mots peuvent être définis comme valeurs logiques et utilisés comme expressions logiques : "**on**" ou "**off**".

```
print-me: false
print either print-me ["turned on"]["turned off"]
turned off
print-me: true
print either print-me ["turned on"]["turned off"]
turned on
```

La valeur **false** n'est pas équivalente au nombre entier zéro, ni non plus à **none**. Cependant, dans les expressions conditionnelles, **false** et **none** ont le même effet :

```
print-me: none
print either print-me ["turned on"]["turned off"]
turned off
```

N'importe quelle valeur assignée à un mot conduit au même effet que si vous aviez **true** :

```
print-me: "just a string"
print either print-me ["turned on"]["turned off"]
turned on
print-me: 11-11-1999
print either print-me ["turned on"]["turned off"]
turned on
```

Les mots suivants sont prédéfinis pour manipuler des valeurs logiques :

```
true
on     ;identique à true
yes    ;identique à true
false
off    ;identique à false
no     ;identique à false
```

Donc, au lieu de **true** et **false**, lorsque cela a du sens, les mots **on** et **off**, ou **yes** et **no** peuvent aussi être utilisés :

```
print-me: yes
print either print-me ["turned on"]["turned off"]
turned on
print-me: no
print either print-me ["turned on"]["turned off"]
turned off
print-me: on
print either print-me ["turned on"]["turned off"]
turned on
print-me: off
print either print-me ["turned on"]["turned off"]
turned off
```

4.3.3 Création

La fonction **to-logic** transforme des valeurs de type **integer!** ou **none!** en type de données **logic!** :

```
probe to-logic 0
false
probe to-logic 200
true
probe to-logic none
false
probe to-logic []
true
probe to-logic "a"
true
probe to-logic none
false
```

4.3.4 Informations complémentaires

La fonction **logic?** permet de savoir si la valeur testée est bien de ce type :

```
probe logic? 1
false
probe logic? on
true
probe logic? false
true
```

Utilisez les fonctions **form**, **print**, et **mold** pour afficher une valeur **logic!** :

```
probe form true
true
probe mold false
false
print true
true
```

4.4 Money

4.4.1 Concept

Il existe divers symboles internationaux pour les valeurs monétaires. Certains de ses symboles apparaissent avant le montant, et d'autres après. Comme standard pour représenter des valeurs monétaires internationales, le langage REBOL utilise le format monétaire des États-Unis, mais permet aussi d'inclure des caractéristiques spécifiques.

4.4.2 Format

Le type de données **money!** utilise la notation en virgule flottante standardisée par l'IEEE, qui fournit une précision sur 15 chiffres, centimes inclus.

Le langage limite la longueur de la valeur à 64 caractères. Les valeurs qui sont en dehors (trop grandes ou trop petites) ou qui ne peuvent être représentées sur 64 caractères génèrent une erreur.

Les valeurs monétaires sont préfixées avec un indicateur (optionnel) pour la devise, suivies d'un signe dollar ($). Un signe plus (+) ou moins (-) peut apparaître, immédiatement suivi par le premier caractère (indication de la devise, ou signe dollar), afin d'indiquer le signe.

```
$123
-$123
$123.45
US$12
US$12.34
-US$12.34
$12,34
-$12,34
```

```
DEM$12,34
```

Pour simplifier la lisibilité des grands nombres, et les mettre en morceaux lisibles, une simple quote (`) peut être placée n'importe où entre deux chiffres à l'intérieur du nombre, mais pas avant le nombre.

```
probe $1'234.56
$1234.56
probe $1'234'567,89
$1234567.89
```

N'utilisez pas de virgules ou de points pour couper des valeurs monétaires, car ces deux caractères représentent le séparateur décimal. Le datatype **money!** est un type de données hybride. Conceptuellement, le montant est un scalaire. Cependant, parce que la désignation de la devise est stockée sous forme de chaîne, le type de données **money!** possède deux éléments :

string!	l'élément désignant le type de devise (USD, EUR, YEN, etc.), qui peut avoir 3 caractères maximum.
decimal!	le montant

Pour illustrer ceci, la valeur monétaire suivante est préfixée avec la chaîne USD pour "dollar US" :

```
my-money: USD$12345.67
```

Voici ces deux composantes :

```
probe first my-money
USD
probe second my-money
12345.67
probe pick my-money 3       ; seulement deux éléments
none
```

Si aucun indicateur de devise n'est employé, l'élément qui y fait référence est vide.

```
my-money: $12345.67

probe first my-money
""
probe second my-money
12345.67
```

Les devises internationales peuvent être spécifiées avec l'indicateur de devises, comme ici :

```
my-money: DKM$12'345,67
```

```
probe first my-money
DKM
probe second my-money
12345.67
```

4.4.3 Création

Utilisez la fonction **to-money** pour transformer des valeurs de type **string!**, **integer!**, **decimal!**, ou **block!** en valeurs de type **money!**.

```
probe to-money 123
$123.00
probe to-money "123"
$123.00
probe to-money 12.34
$12.34
probe to-money [DEM 12.34]
DEM$12.34
probe to-money [USA 12 34]
USA$12.34
```

Les valeurs de type **money!** peuvent être ajoutées, soustraites, et comparées avec d'autres monnaies de la même devise. Une erreur se produit si des devises différentes sont mélangées pour ce genre d'opérations (les conversions automatiques ne sont pas prévues actuellement).

```
probe $100 + $10
$110.00
probe $100 - $50
$50.00
probe equal? DEM$100.11 DEM$100.11
true
```

Les valeurs monétaires peuvent être multipliées et divisées avec des nombres entiers ou décimaux.

Les valeurs monétaires peuvent aussi être divisées par d'autres valeurs monétaires, le résultat étant un nombre entier ou décimal.

```
probe $100 + 11
$111.00
probe $100 / 4
$25.00
probe $100 * 5
$500.00
probe $100 - 20.50
$79.50
probe 10 + $1.20
$11.20
probe 10 - $0.25
$9.75
probe $10 / .50
$20.00
```

```
probe 10 * $0.75
$7.50
```

4.4.4 Autres informations

Vous pouvez utilisez **money?** pour savoir si une valeur est de type de données **money!**.

```
probe money? USD$12.34
true
```

Utilisez les fonctions **form**, **print**, et **mold** avec un argument de type **money!** pour afficher une valeur monétaire, avec l'indicateur de devise et le signe dollar ($), sous forme d'un nombre décimal avec une précision à deux chiffres.

```
probe form USD$12.34
USD$12.34
probe mold USD$12.34
USD$12.34
print USD$12.34
USD$12.34
```

4.5 None

4.5.1 Concept

Le datatype **none!** contient une unique valeur qui représente "rien" ou "aucune valeur". Le concept de **none** est différent de celui d'un bloc vide, d'une chaîne vide, ou d'un caractère nul. C'est une valeur qui représente une *non-existence*.

Une valeur **none!** peut être renvoyée par des fonctions très différentes, en particulier celles qui sont utilisées avec des séries (par exemple, **pick** et **find**).

Le mot REBOL **none** est défini comme faisant partie du type de donnée **none!** et il fait référence à une valeur **none!**. Le mot **none** n'est pas équivalent à **zero** ou à **false**.

Cependant, **none** va être interprété comme **false** par beaucoup de fonctions. Une valeur **none!** autorise beaucoup d'usages comme par exemple servir de valeur de retours à des fonctions relatives aux séries comme **pick**, **find**, et **select** :

```
if (pick series 30) = none [...]
```

Dans des bases de données, une valeur **none** peut être un moyen de remplacer des valeurs manquantes :

```
email-database: [
    "Bobby" bob@rebol.com 40
    "Linda" none 23
    "Sara"  sara@rebol.net 33
```

```
]
```

None peut aussi être utilisée comme valeur logique :

```
secure none
```

4.5.2 Format

Le mot **none** est prédéfini pour manipuler une valeur de type **none!**. Bien que **none** ne soit pas équivalent à **zero** ou **false**, elle est valide dans des expressions conditionnelles et a le même effet que **false** :

```
probe find "abcd" "e"
none
if find "abcd" "e" [print "found"]
```

4.5.3 Création

4.5.4 Autres informations

Il est possible d'utiliser **none?** pour déterminer si une valeur est ou non du type **none!**.

```
print none? 1
false
print none? find [1 2 3] 4
true
```

Les fonctions **form**, **print** et **mold** retournent la valeur **none** lorsqu'un argument **none** leur est fourni :

```
probe form none
none
probe mold none
none
print none
none
```

4.6 Pair

4.6.1 Concept

Le type de données **pair!** est utilisé pour indiquer des coordonnées dans l'espace, comme par exemple des positions à l'écran. Les "pairs" peuvent être utilisées aussi pour définir des tailles en plus des positions.

Les valeurs de type **pair!** sont en particulier utilisées dans REBOL/View.

4.6.2 Format

Une valeur de type **pair!** est définie par deux nombres entiers séparés par un caractère "x".

```
100x50

1024x800

-50x200
```

4.6.3 Création

Utilisez la fonction **to-pair** pour transformer des blocs ou des chaînes de caractères :

```
p: to-pair "640x480"
probe p
640x480
p: to-pair [800 600]
probe p
800x600
```

4.6.4 Autres

La fonction **pair?** permet de savoir si la valeur qui lui est fournie en argument est bien de ce type de données :

```
probe pair? 400x200
true
probe pair? pair
true
```

Les pairs peuvent être utilisées avec la plupart des opérateurs mathématiques associés aux nombres entiers :

```
100x200 + 10x20

10x20 * 2x4

100x30 / 10x3

100x100 * 3

10x10 + 3
```

Il est possible d'extraire des valeurs de type **pair!** leurs composantes individuelles :

```
pair: 640x480
probe first pair
640
```

```
probe second pair
480
```

Toutes les valeurs de type **pair!** supportent les raffinements **/x** et **/y**. Ces raffinements permettent la consultation et la manipulation, de façon spécifique, des coordonnées x et y.

Pour récupérer individuellement chaque coordonnée :

```
probe pair/x
640
probe pair/y
480
```

Pour modifier l'une ou l'autre des coordonnées :

```
pair/x: 800
pair/y: 600
probe pair
800x600
```

4.7 Raffinement

4.7.1 Concept

Les raffinements sont des "modificateurs", comme peuvent l'être les adjectifs utilisés dans les langages humains. Un raffinement indique une variation dans l'usage, ou une extension dans le sens, d'une fonction, d'un objet, d'un nom de fichier, d'une URL, ou d'un chemin (path). Les raffinements sont toujours symboliques dans leurs valeurs.

Ils sont utilisés dans les fonctions :

```
block: [1 2]
append/only block [3 4]
```

mais aussi avec les objets :

```
print system/version
```

ou les fichiers :

```
dir: %docs/core
print read dir/file.txt
```

ou encore les urls :

```
site: http://www.rebol.com
print read site/index.html
```

4.7.2 Format

Les raffinements sont formés par l'association d'un slash {/} et d'un mot REBOL valide (voir la section sur les mots ci-dessous). Par exemple :

```
/only
/test1
/save-it
```

Les raffinements sont habituellement concaténés à d'autres mots, comme dans le cas de :

```
port: open/binary file
```

Mais les raffinements peuvent aussi être écrits seuls, comme lorsqu'ils apparaissent dans la spécification d'une fonction :

```
save-data: function [file data /limit /reload] ...
```

4.7.3 Création

Les raffinements peuvent aussi être créés directement dans le code source :

```
/test
```

ou être composés avec la fonction **to-refinement** :

```
probe to-refinement "test"
/test
```

4.7.4 Et aussi...

Pour tester une valeur, la fonction **refinement?** est utilisable :

```
probe refinement? /test
true
probe refinement? 'word
false
```

4.8 Time

4.8.1 Concept

Le langage REBOL supporte une expression standard du temps en heures, minutes, secondes, jusqu'à la milliseconde et plus. Des valeurs de temps négatives ou positives sont permises.

Le type de donnée **time!** utilise un affichage relatif, plutôt que l'heure absolue.

Par exemple, 10:30 représente 10 heures et 30 minutes, plutôt que 10:30 A.M. ou P.M.

4.8.2 Format

Les valeurs de type **time!** se présentent sous la forme d'un ensemble de nombres entiers séparés par le symbole (:). Les heures et les minutes sont nécessaires, mais les secondes sont optionnelles. : Pour chaque champ (heure:minutes:secondes), les zéros additionnels sont ignorés

```
10:30
0:00
18:59
23:59:50
8:6:20
8:6:2
```

Les valeurs de minutes et de secondes peuvent être supérieures à 60.

Ces valeurs plus grandes que 60 sont automatiquement converties. Ainsi, 0:120:00 est la même chose que 2:00.

```
probe 00:120:00
2:00
```

Les diziémes ou centiémes de seconde sont définis en utilisant le séparateur décimal dans le champ des secondes. (Utilisez un point ou une virgule comme séparateur décimal).

Les champs heures et minutes deviennent optionnels quand un nombre décimal est fourni. Les éléments inférieurs à la seconde sont encodés en nanosecondes, soit au milliardième de seconde :

```
probe 32:59:29.5
32:59:29.5
probe 1:10,25
0:01:10.25
probe 0:0.000000001
0:00:00.000000001
probe 0:325.2
0:05:25.2
```

Les valeurs de type **time!** peuvent être suivies par les chaînes AM ou PM, mais AUCUN n'espace n'est permis dans ce cas.

L'ajout de la chaîne PM revient à ajouter 12 heures à l'heure indiquée :

```
probe 10:20PM
22:20
probe 3:32:20AM
3:32:20
```

Les valeurs **time!** retournées se présentent avec un format standard en heures, minutes, secondes, et fractions de secondes, indépendamment de la façon dont elles ont été saisies :

```
probe 0:87363.21
24:16:03.21
```

4.8.3 Accès aux champs

Les valeurs de type **time!** possèdent trois raffinements qui servent à retourner des informations spécifiques :

Raffinement	Description
/hour	Renvoie la valeur de l'heure
/minute	Renvoie la valeur pour les minutes
/second	Renvoie la valeur des secondes

Voici comment utiliser ces raffinements :

```
lapsed-time: 91:32:12.14
probe lapsed-time/hour
91
probe lapsed-time/minute
32
probe lapsed-time/second
12.14
```

Les données horaires avec des fuseaux horaires peuvent uniquement être utilisées dans des valeurs de type **date!**.

4.8.4 Création

Les valeurs **time!** peuvent être créées avec la fonction **to-time** :

```
probe to-time "10:30"
10:30
probe to-time [10 30]
```

```
10:30
probe to-time [0 10 30]
0:10:30
probe to-time [10 30 20.5]
10:30:20.5
```

Dans les exemples précédents, les valeurs ne sont pas évaluées. Pour évaluer des valeurs comme des expressions mathématiques, utilisez la fonction **reduce**.

```
probe to-time reduce [10 30 + 5]
10:35
```

Dans les différentes opérations mathématiques mettant en oeuvre des valeurs de type **time!**, celles-ci ou les nombres entiers et décimaux impliqués sont manipulés ainsi :

```
probe 10:30 + 1
10:30:01
probe 10:00 - 10
9:59:50
probe 0:00 - 10
-0:00:10
probe 5:10 * 3
15:30
probe 0:0:0.000000001 * 1'500'600
0:00:00.0015006
probe 8:40:20 / 4
2:10:05
probe 8:40:20 / 2:20:05
3
probe 8:40:20 // 4:20
0:00:20
```

4.8.5 Autres informations

La fonction **time?** permet de savoir si son argument a pour datatype **time?** :

```
probe time? 10:30
true
probe time? 10.30
false
```

Utilisez la fonction **now** avec le raffinement **/time** pour retourner la date et l'heure courante :

```
print now/time
14:42:15
```

La fonction **wait** est utilisée pour attendre quelque chose, une certain durée, ou un port, ou les deux. Si la valeur est de type **time!**, **wait** permet une temporisation égale au temps indiqué. Si la

valeur est de type **date!/time!**, la fonction **wait** permet une temporisation jusqu'à la date et l'heure indiquée.
Si la valeur est de type **integer!** ou **decimal!**, **wait** permet d'attendre le nombre de secondes indiqué.

Si la valeur est un port, la fonction **wait** attendra un événement sur ce port.

Si un bloc est fourni en argument à **wait**, l'attente se fera pour chacun des arguments (ports ou valeurs de temps) indiqués. Si un événement s'est produit sur un port, **wait** retourne ce port, ou retourne none si un time-out s'est produit. Par exemple :

```
probe now/time
14:42:16
wait 0:00:10
probe now/time
14:42:26
```

4.9 Tuple

4.9.1 Concept

Il est courant de représenter des numéros de versions, des adresses Internet, ou encore des valeurs de couleurs (RGB) sous la forme d'une séquence de trois ou quatre nombres entiers. Ces types de valeurs sont appelées des tuples (**tuple!**) (comme dans quin**tuple**) et sont représentées sous forme d'entiers séparés par des points.

```
1.3.0 2.1.120 1.0.2.32     ; version
199.4.80.250 255.255.255.0 ; adresse ou masque réseau
0.80.255 200.200.60        ; couleurs en RGB (rouge/vert/bleu)
```

4.9.2 Format

Chaque entier dans une valeur de type **tuple!** peut être comprise entre 0 et 255. Les nombres entiers négatifs génèrent une erreur. De trois à dix entiers peuvent être spécifiés dans un tuple.

Dans le cas où seulement deux entiers sont donnés, il doit y avoir au moins *deux* points, sinon la valeur sera assimilée à un nombre décimal.

```
probe 1.2     ; décimal
1.2
probe type? 1.2
decimal!
probe 1.2.3   ; un tuple
1.2.3
probe 1.2.    ; un autre tuple
1.2.0
probe type? 1.2.
tuple!
```

4.9.3 Création

Utilisez la fonction **to-tuple** pour convertir des données en tuple :

```
probe to-tuple "12.34.56"
12.34.56
probe to-tuple [12 34 56]
12.34.56
```

4.9.4 Et aussi

La fonction **tuple?** permet de déterminer si une valeur est ou non du type **tuple!**.

```
probe tuple? 1.2.3.4
true
```

Utilisez la fonction **form** pour afficher un tuple sous la forme d'une chaîne de caractères :

```
probe form 1.2.3.4
1.2.3.4
```

La fonction **mold** permet de convertir un tuple en chaîne de caractère, mais de façon à ce que celle-ci soit récupérable en tant que tuple :

```
probe mold 1.2.3.4
1.2.3.4
```

La fonction **print** affiche le tuple sur la sortie standard :

```
print 1.2.3.4
1.2.3.4
```

4.10 Words

4.10.1 Concept

Les mots (words) sont les symboles utilisés par REBOL. Un mot peut être ou non une variable, selon la façon dont il est utilisé. Les mots sont souvent utilisés comme symboles.

REBOL n'a pas de mots clés, il n'y a pas de restrictions sur les mots utilisés et la façon dont ils sont utilisés.

Par exemple, vous pouvez définir votre propre fonction appelée **print** et vous en servir à la place de la fonction prédéfinie dans le langage.

Il y a quatre formats différents pour l'utilisation des mots, selon l'opération qui est requise.

Action	Type de Mot	Type de Test	Conversion
word:	set-word!	set-word?	to-set-word
:word	get-word!	get-word?	to-get-word
word	word!	word?	to-word
'word	lit-word!	lit-word?	to-lit-word

4.10.2 Format

Les mots sont composés de caractères alphabétiques, de nombres, et de n'importe lequel des caractères suivants :

```
? ! . ' + - * & | = _ ~
```

Un mot ne peut pas commencer par un nombre, et il existe aussi quelques restrictions sur les mots qui pourraient être interprétés comme des nombres. Par exemple, -1 et +1 sont des nombres, pas des mots. La fin d'un mot est marquée par un espace, un caractère de fin de ligne, ou l'un des caractères suivants :

```
[ ] ( ) { } " : ; /
```

Par ailleurs, les crochets d'un bloc ne sont pas compris dans un mot :

```
[test]
```

Les caractères suivants ne sont pas autorisés dans des mots :

```
@ # $ % ^ ,
```

Les mots peuvent être de n'importe quelle longueur, mais ne peuvent pas dépasser une ligne.

```
this-is-a-very-long-word-used-as-an-example
```

Voici quelques exemples de mots :

```
Copy print test
```

```
number?  time?  date!

image-files  l'image

++ -- == +-

***** *new-line*

left&right left|right
```

Le langage REBOL n'est pas sensible à la casse des caractères. Les mots suivants :

```
blue

Blue

BLUE
```

font tous référence au même mot.

La casse des caractères est préservée lorsque le mot est affiché. Les mots peuvent être réutilisés. La signification d'un mot dépend de son contexte, de sorte qu'un mot peut être réutilisé dans différents contextes.

Vous pouvez réutiliser n'importe quel mot, même les mots REBOL prédéfinis. Par exemple, le mot REBOL **if** peut être réutilisé dans votre code différemment de la façon dont l'interpréteur REBOL l'utilise.

4.10.3 Création

La fonction **to-word** convertit son argument en une valeur de type **word!**.

```
probe to-word "test"
test
```

La fonction **to-set-word** convertit des valeurs en valeurs de type **set-word!** (mot défini).

```
probe make set-word! "test"
test:
```

La fonction **to-get-word** renvoie des valeurs de type **to-get-word!**.

```
probe to-get-word "test"
:test
```

La fonction **to-lit-word** renvoie des valeurs de type **lit-word!** (mot littéral).

```

```

```
probe to-lit-word "test"
'test
```

4.10.4 Autres informations

Utilisez les fonctions **word?**, **set-word?**, **get-word?**, et **lit-word?** pour tester le datatype :

```
probe word? second [1 two "3"]
true
if set-word? first [word: 10] [print "it is set"]
it is set
probe get-word? second [pr: :print]
true
probe lit-word? first ['foo bar]
true
```

Annexe 2 - Les Erreurs

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

Contenu

[Retour au sommaire]

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

[Retour au sommaire]

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.

[Retour au sommaire]

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.

[Retour au sommaire]

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
]
```

[Retour au sommaire]

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 reservé.

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**.

[Retour au sommaire]

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
```

[Retour au sommaire]

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
```

7.2 Erreurs de script

7.2.1 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
```

7.2.2 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:
```

7.2.3 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
```

7.2.4 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"
```

7.2.5 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"
```

7.2.6 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]
```

7.2.7 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
```

7.2.8 no-op-arg

Un opérateur mathématique ou de comparaison a été utilisé sans que 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 +
```

7.2.9 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]
```

7.2.10 not-defined

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

Message :

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

7.2.11 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
```

7.2.12 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
```

7.2.13 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"
```

7.2.14 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"
```

7.2.15 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"
```

7.2.16 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
```

7.2.17 no-memory

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

Message :

```
Not enough memory
```

7.2.18 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
```

7.2.19 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
```

7.2.20 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"]
```

7.2.21 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]
```

7.2.22 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"
```

7.2.23 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]
```

7.3 Erreurs d'accès

7.3.1 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
```

7.3.2 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
```

7.3.3 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
```

7.3.4 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
```

7.3.5 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]
```

7.3.6 socket-open

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

Message :

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

7.3.7 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/
```

7.3.8 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
```

7.3.9 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
```

7.3.10 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
```

7.3.11 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
```

7.3.12 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
```

7.3.13 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
```

7.3.14 invalid-path

Un path mal formé a été employé.

Message :

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

Exemple :

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

7.4 Erreurs internes

7.4.1 bad-path

Un chemin commencant 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
```

7.4.2 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"]
```

7.4.3 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
```

7.4.4 globals-full

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

Message :

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

Annexe 3 - La console

Ce document est la traduction française de l'Annexe 3 du User Guide de REBOL/Core, qui concerne la console.

Contenu

[Retour au sommaire]

1. Historique de la traduction

Date	Version	Commentaires	Auteur	Email
16 Septembre 2005 07:03	1.0.0	Traduction initiale	Philippe Le Goff	lp--legoff--free--fr

[Retour au sommaire]

2. Le prompt de la ligne de commande

NdT :
On appelle "prompt" l'indicateur alphanumérique qui indique que la console attend une saisie. En console Dos, ou Unix, cette notion est bien connue. REBOL possède aussi en mode console un prompt.

Le prompt par défaut de la ligne de commande est ">>". Il est possible de modifier cela avec par exemple le code suivant :

```
system/console/prompt: "Input: "
```

Le prompt devient alors :

```
Input:
```

Le prompt peut être un bloc qui sera évalué à chaque fois. La ligne suivante affiche l'heure courante :

```
system/console/prompt: [reform [now/time " >> "]]
```

Le résultat devrait ressembler à un prompt comme ceci :

```
10:30 >>
```

[Retour au sommaire]

3. Indicateur de résultat

Par défaut, l'indicateur de résultat est "==", il peut être aussi modifié avec une ligne telle que :

```
system/console/result: "Result: "
```

Ces exemples modifiant les paramètres par défaut peuvent être placés dans votre fichier **user.r** pour les rendre permanent, à chaque démarrage de REBOL.

[Retour au sommaire]

4. Rappel de l'historique

Chaque ligne saisie dans la console REBOL est conservée dans un bloc, en historique, et peut être rappelée plus tard en utilisant les touches "flèche haut" et "flèche bas" (up et down).

Par exemple, si vous appuyez une seule fois sur la touche "flèche haut", vous rappellerez la dernière ligne de commande saisie.

Le bloc d'historique contenant toutes les lignes entrées à la console peut être consulté en appelant l'attribut **history** de l'objet **system/console**.

```
probe system/console/history
```

Vous pouvez sauvegarder ce bloc dans un fichier :

```
save %history.r system/console/history
```

et il peut être rechargé plus tard :

```
system/console/history: load %history.r
```

Ces lignes peuvent être placées dans le fichier user.r pour sauvegarder et recharger votre historique entre deux sessions.

[Retour au sommaire]

5. Indicateur d'activité

Lorsque REBOL attend qu'une opération (par exemple, une requête réseau) se termine, un indicateur d'activité apparaît à l'écran pour indiquer que quelque chose est en cours.

Vous pouvez changer l'indicateur d'activité avec un ligne comme celle-ci par exemple :

```
system/console/busy: "123456789-"
```

Quand REBOL travaille en mode "silencieux" (quiet mode), l'indicateur d'activité n'est pas affiché.

[Retour au sommaire]

6. Opérations spécifiques avec la console

La console se comporte comme un "terminal virtuel" qui vous permet des opérations comme le mouvement et l'adressage du curseur, l'édition de la ligne, l'effacement de l'écran, l'usage de touches de contrôle, et de requêtes sur la position du curseur. Les séquences de contrôle suivent le standard ANSI. Cela vous permet de définir votre propre terminal (indépendamment de la plate-forme sur laquelle vous êtes) pour des programmes comme des éditeurs de textes, des client emails, ou des émulateurs telnet.

Les fonctionnalités de la console s'appliquent aux entrées et aux sorties. Pour les entrées, les touches de fonctions seront converties en séquences d'échappement de plusieurs caractères. Pour les sorties, les séquences d'échappement peuvent être utilisées pour contrôler l'affichage du texte dans la fenêtre de la console. Aussi bien les entrées que les sorties commencent avec le caractère d'échappement ANSI : 27 en décimal (1B en hexa). Le caractère suivant dans la séquence indique les contrôles en entrée ou l'opération en sortie pour le contrôle du terminal. Les caractères ANSI sont sensibles à la casse et nécessitent en principe d'être en majuscules.

6.1 Séquences entrées au clavier

Les séquences saisies avec les touches de fonctions sont listées dans la table ci-dessous (pour les systèmes et les shells qui les supportent, comme Linux, BSD, etc.). Pour recevoir ces séquences en un flux de caractères non évalués, vous devez bloquer le mode "lignes" pour le port "input" de REBOL :

```
set-modes system/ports/input [lines: false]
```

A présent, vous pouvez récupérer l'entrée à partir du port (avec les fonctions **copy** ou **read-io**) ou utiliser une fonction comme input pour récupérer chaque caractère :

```
while [
    code: input
    code <> 13  ; ENTER
][
    probe code
]
```

Voici quelques unes des fonctions ANSI les plus courantes :

Touches de fonction	Comme séquences d'échappement	Comme bloc REBOL
F1	ESC O P	[27 79 80]
F2	ESC O Q	[27 79 81]
F3	ESC O R	[27 79 82]
F4	ESC O S	[27 79 83]
F5	ESC [1 5 ~	[27 91 49 53 126]
F6	ESC [1 7 ~	[27 91 49 55 126]
F7	ESC [1 8 ~	[27 91 49 56 126]
F8	ESC [1 9 ~	[27 91 49 57 126]
F9	ESC [2 0 ~	[27 91 50 48 126]
F10	ESC [2 1 ~	[27 91 50 49 126]
F11	ESC [2 2 ~	[27 91 50 50 126]
F12	ESC [2 3 ~	[27 91 50 51 126]
Home	ESC [1 ~	[27 91 49 126]
End ou Fin	ESC [4 ~	[27 91 52 126]
Page-up	ESC [5 ~	[27 91 53 126]

Page-down	ESC [6 ~	[27 91 54 126]
Insert	ESC [2 ~	[27 91 50 126]
Up	ESC [A	[27 91 65]
Down	ESC [B	[27 91 66]
Left	ESC [D	[27 91 68]
Right	ESC [C	[27 91 67]

6.2 Séquences en sortie pour le terminal

Il y a plusieurs variantes dans les séquences de caractère de contrôle. Certaines commandes sont précédées par un chiffre (en ASCII), indiquant que l'opération doit être réalisée le nombre de fois indiquées par le chiffre.

Par exemple aussi, la commande permettant le déplacement du curseur peut être précédée par deux nombres séparés d'un point virgule, pour indiquer que la position du curseur doit être modifiée selon la ligne et la colonne courante. Les commandes de contrôle du curseur (les majuscules sont requises) sont indiquées dans la table suivante :

Séquence en sortie	**Description**
(1B)	Séquence d'échappement à utiliser avant les codes ci-dessous
D	Déplace le curseur d'un espace vers la gauche
C	Déplace le curseur d'un espace vers la droite
A	Déplace le curseur d'un espace vers le haut
B	Déplace le curseur d'un espace vers le bas
n D	Déplace le curseur de n espaces vers la gauche
n C	Déplace le curseur de n espaces vers la droite
n A	Déplace le curseur de n espaces vers le haut
n B	Déplace le curseur de n espaces vers le bas

r ; c H	Déplace le curseur vers la ligne r, colonne c*
H	Déplace le curseur vers le coin gauche supérieur (home)*
P	Efface un caractère à la droite de la position courante
n P	Efface deux caractères à la droite de la position courante
@	Insère un espace blanc à la position courante
n @	Insère n espaces blancs à la position courante
J	Efface l'écran et déplace le curseur dans le coin supérieur gauche (home)*
K	Efface tous les caractères entre la position courante et la fin de la ligne
6n	Place la position courante du curseur dans un buffer (input buffer)
7n	Place les dimensions de l'écran dans un buffer (input buffer)

Le coin supérieur gauche est défini à la colonne 1, ligne 1.

L'exemple suivant déplace le curseur de huit espaces vers la droite :

```
print "^(1B)[10CHi!"
Hi
```

Cet exemple déplace le curseur de sept espaces vers la gauche et efface le restant de la ligne :

```
cursor: func [parm [string!]][join "^(1B)[" parm]
print ["How are you" cursor "7D" cursor "K"]
How a
```

Pour trouver la taille courante de la fenêtre de la console, vous pouvez utiliser cet exemple :

```
cons: open/binary [scheme: 'console]

print cursor "7n"
screen-dimensions: next next to-string copy cons
33;105R
close cons
```

L'exemple précédent ouvre la console, envoie un caractère de contrôle vers le buffer de l'input, et copie la valeur retournée. Elle lit la valeur (dimension de l'écran) qui est renvoyée après le caractère de contrôle et ferme la console. La valeur renvoyée est la hauteur et la largeur, séparées par un point-virgule (;), et suivies par un "R". Dans cet exemple, l'écran fait 33 lignes et 105 colonnes.

*** : Autoscrolling**

L'affichage d'un caractère dans le coin inférieur droit de certains terminaux force la création d'une nouvelle ligne, et un réajustement de l'écran. D'autres terminaux n'ont pas le même comportement. Ces différences de fonctionnement doivent être prises en compte pour écrire des scripts REBOL multi-platformes.

REBOL/Core Changes

Core Versions 2.3.0 - 2.5.6
Including /View, /SDK, and /Command

Contents:

1. Core 2.5.6

Miscellaneous fixes made during the version 2.5.6 releases of the /SDK, /Command, and /SDK/Command products.

1.1 Command Line Startup (Fixes CGI/shell scripts)

REBOL ignores CR characters (13) in script files, but a problem occurred if a CR appeared on the command line that started REBOL. REBOL would display HELP information. This problem has been fixed and solves the problem that may occur when you used an FTP binary transfer of a CGI script to your Linux/Unix web server.

1.2 Implied Quit

All REBOL products are now consistent in what they do when a script has finished running, but did not execute a QUIT. They now now force a QUIT at the end of the script. If you want to go to the REBOL console, put a HALT in your script.

1.3 Corrupt Datatype Failure

In some situations REBOL would crash at the end of a script with a "corrupt datatype" error. This was caused by optimizations made in 2.5.3 that freed unused internal structures. The problem would only occur when a script caused memory recycling and did not include a QUIT or HALT at its end. This has been fixed.

1.4 Help Native! Fixed

Requesting HELP on NATIVE! has been fixed (the native! word has been restored).

1.5 Get on NONE

A GET of NONE is now allowed and will return NONE.

```
print get none
```

This makes it easier to write code that fetches values from an object when the word within the object may be missing. For example, the code below will no longer cause an error if the word is missing from the object:

```
value: get in object 'word
```

This makes code simpler for cases like CGI object form values:

```
cgi: construct decode-cgi read-cgi
name: any [get in cgi 'name "default"]
```

In the example above, if the cgi/name does not exist, the default string is used. No error will occur if the name is missing from the CGI object.

1.6 Set-Browser-Path

The SET-BROWSER-PATH function now accepts REBOL filename format. It will also accept a string that is in local OS file format (when provided as a string! type). You can also set it to NONE.

The SET-BROWSER-PATH function will also return it's prior setting.

If you wish to disable this function (for security reasons), you can set it to NONE or unset the function.

```
set-browser-path: none

unset 'set-browser-path
```

1.7 DO On Command Line

The --do option is processed again on the command line. This allows you to pass expressions to be executed from the shell command line. For example:

```
rebol --do "probe system/options/args"
```

Note that --do now happens after the rebol.r and user.r files are evaluated (letting you initialize your own custom functions and values) but before the script on the command line is evaluated.

The --do option can be used to define variables prior to the execution of your script. For example:

```
rebol make-website.r --do "verbose: true"
```

would set the VERBOSE variable to TRUE at the start of execution. To set the default value in the script, you might use code such as:

```
if not value? 'verbose [verbose: false]
```

1.8 FIND/Any Bug Fixed.

A serious error was reported related to using FIND with the /any refinement (wildcards). For some cases FIND/any would return a valid string rather than NONE. For example:

```
find/any next "abc" "d"
```

returned "c". This problem has been fixed.

1.9 Molding Empty File Names

MOLDing an empty filename resulted in source code that could not be loaded. In rare cases, this might case a SAVE to create a file that LOAD could not handle. Molding an empty file name now outputs the characters %"" to avoid this problem. This fixes the problem in MOLD, SAVE, and PROBE.

2. Core 2.5.5

2.1 Main Purpose

The primary purpose of the Core 2.5.5 release is to bring REBOL/Core into closer compatibility with the new Mac OS X and REBOL/SDK kernels.

2.2 New Core License

REBOL/Core licensed has changed to allow personal, educational, or commercial use of the CORE software free of charge.

Note that this license change only applies to REBOL/Core (and will be added to the next release of REBOL/View), but it does not apply to REBOL/SDK, /Command, /IOS or other REBOL commercial products.

2.3 Startup Files (user.r and rebol.r)

Beta V2.5.4 did not evaluate the rebol.r and user.r files. V2.5.5 evaluates them again, although using a slightly different method. This should not affect your program. The order is still:

1. Check the current directory first.
2. Check the system/options/home directory second.

This allows you to have a single standard rebol.r file (and optionally a user.r file) that is shared by all users on multiuser systems.

2.4 How Scripts are Started

The method used to start initial scripts has been modified.

This change will have no affect on your programs, but it will produce a better error message on script errors by eliminating the error line that refers to the problem being "Near do-boot". That line was confusing to new users, because do-boot was not part of their script file.

2.5 Command Line Terminator (CR)

REBOL no longer shows Usage info if a CR char is passed from a Unix shell on the shell command line.

Although REBOL scripts properly read and load with respect to the CR (ASCII 13) character, some operating systems will pass the CR as a command line argument, causing older versions of REBOL to print its usage help information.

For example, you might see a problem with a CGI shell script starting with:

```
#!/home/user/rebol -cs
REBOL [...]
```

that was written on a Windows system and transferred to Linux without converting line terminators.

This case has been fixed.

2.6 Changes to SECURE

Beta V2.5.4 did not set SECURE prior to running scripts (in other words, it functioned like REBOL/Base).

With V2.5.5 security is being set again, although with a slight change: by default you can now write to a script's target directory. For example, if you run:

```
REBOL /home/test/script.r
```

REBOL will startup with WRITE access to /home/test. All other directories will be set to READ only (with WRITE ask).

This change makes it easier for users to write scripts that write and update local data files without the need to provide a -S command line option, but it still protects the rest of your files.

Note that the +S command line option has also been changed. Previously, +S meant SECURE QUIT, which was almost useless because your REBOL would QUIT as soon as it tried to read its rebol.r and user.r files. Now +S means SECURE ASK, so running with:

```
rebol +s script.r
```

Will prompt the user for all READ/WRITE operations for files and networking. A safe mode of operation, but less strict than before.

In addition, the --secure options should also work again now:

```
rebol --secure allow script.r
rebol --secure ask script.r
```

2.7 MOLD/Flat Refinement

The /flat refinement allows you to MOLD code and data without line indentation, making the resulting string smaller (for sending over a network, storing as compressed data, encapping, etc.).

For example:

```
blk: [
    name [
        first "Bob"
        last "Smith"
    ]
    options [
        colors [
            red
            green
            blue
        ]
    ]
]

>>print mold blk
[
    name [
        first "Bob"
        last "Smith"
    ]
    options [
        colors [
            red
            green
            blue
        ]
    ]
]

>> print mold/flat blk
[
name [
first "Bob"
last "Smith"
]
options [
colors [
red
green
blue
]
]
]
```

2.8 Truncation of Series Index (when Past Tail)

The handling of an "past the tail" series index has been changed.

In prior versions of REBOL, referring to a series past its tail would cause an error:

```
>> a: "ABC"
== "ABC"
>> b: next a
== "BC"
>> clear a
== ""
>> insert b "X"
** Script Error: Out of range or past end
** Near: insert b "X"
```

In some situations, this problem could become difficult to detect, because nearly any reference to B would cause the error.

```
>> tail? b
** Script Error: Out of range or past end
** Near: tail? b
```

As a result, the "past end" handling has been relaxed. For many functions, the index will now truncate to the current tail of the series:

```
>> a: "ABC"
== "ABC"
>> b: next a
== "BC"
>> clear a
== ""
>> tail? b
== true
```

2.9 AT on PORTs

The AT function (similar to the SKIP function) now works for the Port datatype.

2.10 Added SIXTH Through TENTH Functions

Five new ordinal functions have been added:

```
sixth
seventh
eighth
ninth
tenth
```

The benefit of these function becomes more clear when you consider using them to pick values from a series (instead of using an object field).

```
name-of: :first
age-of:  :second
```

```
date-of: :third
...
product-of: :ninth
...
if empty? product-of record [print "missing product"]
```

The line above is much more readable than:

```
if empty? ninth record [print "missing product"]
```

Here's a handy utility function for creating such named accessors:

```
ordain: func [
    "Defines ordinal accessors (synonyms)."
    words [block!] /local ords
][
    ords: [first second third fourth fifth
        sixth seventh eighth ninth tenth]
    foreach word words [
        set word get first ords
        ords: next ords
    ]
]
```

The above example would then be simplified to:

```
ordain [name-of age-of date-of ... product-of]
...
if empty? product-of record [print "missing product"]
```

Question: Would you like to see ordain become a regular function in REBOL? Tell us via Feedback.

2.11 Lines Longer Than 4 KB in Direct/Lines Mode

Earlier versions of REBOL did not properly expand their internal line buffers beyond 4 KB for files opened in direct/lines mode. As a result, lines longer than 4 KB (no line terminators for more than 4 KB) would not be read correctly and would cause an end of file. This problem has been fixed in the current release.

2.12 DECODE-CGI Handles Duplicate Names

With increased use of REBOL for CGI scripts, we've expanded the DECODE-CGI function to handle duplicate name fields.

For example, checklists used in web forms may return multiple values for a single variable. DECODE-CGI will now make these multivalued returns into blocks:

```
>> cgi-str: {name=bob&option=view&option=email&}
>> decode-cgi cgi-str
== [name: "bob" option: ["view" "email"]]
```

2.13 ALTER Fixed

A serious error in the ALTER function (a data set flagging function, see updated docs in The REBOL Dictionary) caused it to malfunction. This problem has been fixed.

2.14 Network Errors in TRY (Net-Error Throw)

The NET-ERROR function used THROW for errors, rather than just letting the error throw itself. This caused some types of network errors to bypass a higher level catch when using TRY.

This meant that if you used a TRY around sections of your code, certain network errors (such as an SMTP problem) may be thrown past it, which would cause the error to be passed all the way out to user at the console. This was a problem for SDK scripts, and has been fixed.

2.15 Extended FTP Directory Paths

When using FTP URLs, there was no easy way to provide a full path from the root directory.

For example, your logon directory may be:

```
/home/luke/
```

but you want to refer to:

```
/var/log/
```

In the new release, you can use a double slash URL to provide the absolute directory path:

```
read ftp://user:pass@host//var/log/messages
```

(Please tell us if you have any problems using this format.)

2.16 STATS for SYSTEM/STATS

The STATS function is now directly accessible. You no longer need to use SYSTEM/STATS.

In addition, the STATS function now more accurately computes memory usage.

2.17 PURL Captured

The variable PURL used in DECODE-URL is now a local variable rather than a global.

3. Core 2.5.3

3.1 Credits

Thanks go to these contributors for their feedback, suggestions, or solutions:

```
Andrew Martin
"Anton"
Cal Dixon
Carl Sassenrath
Ernie van der Meer
Gregg Irwin
Ladislav Mecir
Maarten Koopmans
"Mr. Martin Mr. Saint"
Reichart Von Wolfsheild
```

Did we miss your feedback or contribution to fixes or improvements? If so, tell us at: http://www.rebol.com/feedback.html.

It is not always possible to get every change into each new release, but if you have found a bug that's causing you problems, please let us know and we'll do our best to fix it. Provide a very short, repeatable example of the problem, and if you have a fix to the problem, send that along as well. (You'll see the fix happen much faster that way.)

3.2 MAKE-DIR Rewritten

The MAKE-DIR function has been rewritten and now works correctly. In addition, it has been relaxed to not cause an error when the specified directory exists. Use the EXISTS? function if you need to check that.

3.3 New Bitset Functions: CLEAR, LENGTH?, EMPTY?

Three new functions (actions) have been added to bitsets:

The CLEAR function quickly clears all bits to zero.

The LENGTH? function returns the number of bits in the bitset (always multiple of 8).

The EMPTY? function returns TRUE if any bit is set, otherwise it returns FALSE. (Note that EMPTY? is the same function as TAIL?; therefore, TAIL? also returns the same results, but the word has no meaning for bitsets.)

Bitsets are used as high performance logic arrays for character sets and hashes.

Examples:

```
>> items: make bitset! 40
== make bitset! #{0000000000}

>> length? items
== 40

>> empty? items
== true

>> insert items 10
== make bitset! #{0004000000}

>> empty? items
== false

>> clear items
```

```
== make bitset! #{0000000000}

>> empty? items
== true

>> empty? negate items
== false
```

3.4 Changes to SKIP Function

There are three changes to SKIP that you should note:

1. SKIP now handles decimal offsets correctly. Values are truncated down to lower integer value. (Note that decimal offsets should be used with caution because 1.9999999 is not the same as 2.0 when it comes to indexing.)

```
>> skip "abc" 1.9999999
== "bc"
>> skip "abc" 2.0
== "c"
```

2. SKIP can have a logic offset. It is made consistent with PICK and AT when used with logic offsets.

```
>> pick [red green] true
== red
>> skip [red green] true
== [red green]
>> at [red green] true
== [red green]

>> pick [red green] false
== green
>> skip [red green] false
== [green]
>> at [red green] false
== [green]
```

For logic offsets, SKIP is identical to AT.

3. SKIP on images will offset pixels, not RGBA bytes. This is consistent with AT, PICK, and POKE. If you are currently using skip on images, this change will affect your code.

3.5 ARRAYs Initialized with Block Values

If a block value is provided as the initial element value in the ARRAY function, each element should be the block, not the contents of the block.

In addition, if the initial value is a SERIES of any type (e.g. string, block, email, url) it will be deep copied into each array element (that is, the value of each element will be unique, not shared).

As an example:

```
>> a: array/initial [2 3] [1 2]
```

```
== [[[1 2] [1 2] [1 2]] [[1 2] [1 2] [1 2]]]
>> append a/1/2 3
== [1 2 3]
>> a
== [[[1 2] [1 2 3] [1 2]] [[1 2] [1 2] [1 2]]]
```

3.6 Added PARSE BREAK Word

Within some types of PARSE loops such as ANY and SOME, it can sometimes be difficult to write rules that terminate the loop. This happens primarily when using general rules that try to capture "all other cases" within the loop. For example, code such as:

```
parse item [
    any [
        "word" (print "got word")
        | copy value [to "abc" | to end]
            (print value)
    ]
]
```

will end up looping forever because the second rule within the ANY will always succeed, and the ANY will never terminate.

To help solve this problem, the BREAK keyword was added to the parse dialect. Its use is simple. When the BREAK word is encountered within a rule block, the block is immediately terminated regardless of the current input pointer. Expressions that follow the BREAK within the same rule block will not be evaluated.

The above example can now be written as:

```
parse item [
    any [
        "word" (print "got word")
        | copy value [to "abc" | to end]
            (print value) break
    ]
]
```

The ANY loop will be terminated after the second rule.

Generally, for ANY rule blocks that need to terminate at the end of the input stream, you can add an END check followed by the BREAK keyword, such as in this example:

```
parse item [
    any [
          end break
        | "word" (print "got word")
        | copy value [to "abc" | to end]
            (print value)
    ]
]
```

3.7 Fix to OPEN on Network Ports

OPEN on network port under some versions of Windows32 would fail due to an incorrect error code returned from WinSock library (as documented by Microsoft). Fixed.

3.8 Fixed Crash on Modified Functions

Fixed the crash that happened when modifying a function's value while evaluating its arguments. For example, the code below:

```
a: func [x] [print x]
b: func [] [a: 42]
a b
```

no longer causes a crash.

Note that modifying a function while it is evaluating may produce odd results that may vary between implementation versions and should generally be avoided.

3.9 CHANGE Accepts Any Type Value

To be consistent with INSERT, the CHANGE function allows a new value to be any datatype (ANY-TYPE!).

Users should be aware that a missing value argument will result in a CHANGE value of UNSET!, not an error.

For example:

```
>> a: [10]
== [10]
>> do [change a]
== []
>> probe a
[unset]
== [unset]
```

3.10 Unset Object Variables (on Exit)

If exit occurs during object evaluation, unassigned variables not copied from parent object are unset. Fixes the "end" bug where the first variable was set to the END! datatype.

```
>> probe make object! [exit a: b:]
== make object! [
    a: unset
    b: unset
]
```

3.11 Added BUILD-MARKUP Function

This function was inspired by the EREBOL concept of Maarten Koopmans and Ernie van der Meer. Essentially, the idea is that REBOL makes a powerful PHP style markup processor for generating

web pages and other markup text.

The BUILD-MARKUP function has been added to support this operation, and will become a standard part of every REBOL implementation.

The BUILD-MARKUP function takes markup text (e.g. HTML) that contains tags that begin with "<%" and end with "%>". It evaluates the REBOL code within each tag (as if it were a REBOL block), and replaces the tag with the result. Any REBOL expression can be placed within the tag. As PHP has shown, this is a very useful technique.

For example:

```
== build-markup "<%1 + 2%>"
>> "3"

== build-markup "<B><%1 + 2%></B>"
>> "<B>3</B>"

== build-markup "<%now%>"
>> "2-Aug-2002/18:01:46-7:00"

== build-markup "<B><%now%></B>"
>> "<B>2-Aug-2002/18:01:46-7:00</B>"
```

Supplying a <%now%> tag to BUILD-MARKUP inserts the current date/time in the output.

Here's a short example that generates a web page from a template and a custom name and email address:

```
template: {<HTML>
    <BODY>
    Hi <%name%>, your email is <i><%email%></i>.<P>
    </BODY>
    </HTML>
}

name: "Bob"
email: bob@example.com
page: build-markup template
```

Don't forget the two % characters within the tag. It's a common mistake.

The value that is returned from the tag code is normally "joined" into the output. You can also use FORM or MOLD on the result to get the type of output you require. The example below loads a list of files from the current directory and displays them three different ways:

```
Input: "<PRE><%load %.%></PRE>"
Result: {<PRE>build-markup.rchanges.txt</PRE>}

Input: "<PRE><%form load %.%></PRE>"
Result: {<PRE>build-markup.r changes.txt</PRE>}

Input: "<PRE><%mold load %.%></PRE>"
Result: {<PRE>[%build-markup.r %changes.txt]</PRE>}
```

If the evaluation of a tag does not return a result (for example using code such as print "test"), then nothing is output. In this case, the output of PRINT will be sent to the standard output device.

```
Input: {<NO-TEXT><%print "test"%></NO-TEXT>}
test
Result: "<NO-TEXT></NO-TEXT>"
```

The BUILD-TAG function can be used within the tag for converting REBOL values into markup output:

```
Input: {<%build-tag [font color "red"]%>}
Result: {<font color="red">}
```

Tags can set and use variables in the same way as any REBOL script. For example, the code below loads a list of files from the current directory, saves it in a variable, then prints two file names:

```
Input: "<PRE><%first files: load %.%></PRE>"
Result: {<PRE>build-markup.r</PRE>}

Input: "<PRE><%second files%></PRE>"
Result: {<PRE>changes.txt</PRE>}
```

Note that variables used within tags are always global variables.

If an error occurs within a tag, an error message will appear as the tag's result. This allows you to see web page errors from any HTML browser.

```
Input: "<EXAMPLE><%cause error%></EXAMPLE>"
Result: {<EXAMPLE>***ERROR no-value in: cause error</EXAMPLE>}
```

If you do not want error messages to be output, use the /QUIET refinement. The example above would result in:

```
Result: "<EXAMPLE></EXAMPLE>"
```

3.12 Revised BUILD-TAG Function

The previous version of BUILD-TAG generated poor results for most input combinations. It has been replaced by a new function that was contributed by Andrew Martin. This function produces better results, but ones that are different from the previous function. If you are currently using BUILD-TAG, you will need to adjust your code.

```
In:  [input "checked" type "radio" name "Favourite" value "cat"]
Old: {<input="checked" type="radio" name="Favourite" value="cat">}
New: {<input checked type="radio" name="Favourite" value="cat">}

In:  [html xml:lang "en"]
```

```
Old: {<html="xml:lang"="en">}
New: {<html xml:lang="en">}

In:  [body text #FF80CC]
Old: {<body text="FF80CC">}
New: {<body text="#FF80CC">}

In:  [a href %Test%20File%20Space.txt]
Old: {<a href="Test File Space.txt">}
New: {<a href="Test%20File%20Space.txt">}

In:  [/html gibber %Froth.txt]
Old: {</html gibber="Froth.txt">}
New: "</html>"

In:  [?xml version "1.0" encoding "UTF-8"]
Old: {<?xml version="1.0" encoding="UTF-8">}
New: {<?xml version="1.0" encoding="UTF-8"?>}

In:  [html xmlns http://w3.org/xhtml xml:lang "en" lang "en"]
Old: {<html xmlns="http://w3.org/xhtml"="xml:lang"="en" lang="en">}
New: {<html xmlns="http://w3.org/xhtml" xml:lang="en" lang="en">}

In:  [html xmlns http://w3.org/xhtml/ xml:lang "en" lang "en"]
Old: {<html xmlns="http://w3.org/xhtml/"="xml:lang"="en" lang="en">}
New: {<html xmlns="http://w3.org/xhtml/" xml:lang="en" lang="en">}
```

3.13 Revised DECODE-CGI Function

Several bugs have been fixed in DECODE-CGI. More specifically, the function handles empty attribute assignments. Here are some examples:

```
Input:   "name=val1&name=val2"
Decoded: [name: "val1" name: "val2"]

Input:   "name1=val1&name2&name3=&name4=val3"
Decoded: [name1: "val1" name2: "" name3: "" name4: "val3"]

Input:   "name1="
Decoded: [name1: ""]

Input:   "name2&"
Decoded: [name2: ""]

Input:   "name3=&"
Decoded: [name3: ""]

Input:   "name4=val"
Decoded: [name4: "val"]

Input:   "name5=val&"
Decoded: [name5: "val"]
```

The new function is based on Andrew Martin's contribution. Thanks!

3.14 UNPROTECT Fixed

Attempting to UNPROTECT a block containing any value other than a word could cause a crash.

Now non-word values are ignored.

3.15 ALTER added to Core

The ALTER function found only in View is general purpose and has been made available in all version of REBOL.

3.16 SYSTEM Word Protected

To help prevent an accidental beginner mistake that is difficult to debug, the SYSTEM and REBOL words are now protected. If you need to change them, use UNPROTECT first.

3.17 Error Message for Word Context

The error message "word not defined in this context" has been changed to "word has no context" to better indicate that the word has never been defined (more precisely, never been bound) in the context of an object, function, or global environment.

3.18 Fixed Crash on Future Dates

Prevents startup crash caused by error in Microsoft Windows time functions when running with system date set to greater than 2036.

4. Core 2.5.2

4.1 CONSTRUCT Object Creator

We've added a "light-evaluation" object creator to provide a higher level of security for imported data objects. The function is called CONSTRUCT and it makes new objects, but without a normal evaluation of the object's specification (as is done in the MAKE and CONTEXT functions).

When you CONSTRUCT an object, only literal types are accepted. Functional evaluation is not performed. This allows your code to directly import objects (such as those sent from unsafe external sources such as email, cgi, etc.) without concern that they may include "hidden" side effects using executable code.

CONSTRUCT is used in the same way as the CONTEXT function:

```
obj: construct [
    name: "Fred"
    age: 27
    city: "Ukiah"
]
```

but, no evaluation takes place. That means object specifications like:

```
obj: construct [
    name: uppercase "Fred"
    age: 20 + 7
    time: now
]
```

do not produce their evaluated results.

The CONSTRUCT function is useful for importing external objects. For example, loading preference settings from a file can done with:

```
prefs: construct load %prefs.r
```

Similarly, you can use CONSTRUCT to load a CGI or email response.

To provide a template object that contains default variable values (similar to MAKE), use the /WITH refinement. The example below would use an existing object called standard-prefs as the template.

```
prefs: construct/with load %prefs.r standard-prefs
```

The CONSTRUCT function will perform evaluation on the words TRUE, FALSE, NONE, ON, and OFF to produce their expected values. Literal words and paths will also be evaluated to produce their respective words and paths. For example:

```
obj: construct [
    a: true
    b: none
    c: 'word
]
```

The obj/a value would be logical TRUE, obj/b would be NONE, and obj/c would be WORD.

4.2 LOAD Change (Important)

Script header objects are now created by the CONSTRUCT function and they are no longer evaluated.

This change provides a greater level of default security when the LOAD function is used to load REBOL data. The change should affect very few scripts (only those that depend on evaluated header variables -- rare.)

The LOAD function can be used safely without the need to use LOAD/ALL to inspect a script header prior to evaluation.

4.3 HELP Expanded

Help can now be used to explore the fields of an object in a user-friendly format. For example, typing this line at the prompt:

```
help system
```

will produce this result:

```
SYSTEM is an object of value:
   version         tuple!    1.0.4.3.1
   build           date!     1-May-2002/13:31:11-7:00
```

```
   product          word!     Link
   components       block!    length: 45
   words            object!   [unset! error! datatype! context! native
   license          string!   {REBOL End User License Agreement IMPORT
   options          object!   [home script path boot args do-arg link-
   user             object!   [name email home words]
   script           object!   [title header parent path args words]
   console          object!   [history keys prompt result escape busy
   ports            object!   [input output echo system serial wait-li
   network          object!   [host host-address]
   schemes          object!   [default Finger Whois Daytime SMTP POP I
   error            object!   [throw note syntax script math access co
   standard         object!   [script port port-flags email face sound
   view             object!   [screen-face focal-face caret highlight-
   stats            native!   System statistics. Default is to return
   locale           object!   [months days]
   user-license     object!   [name email id message]
```

This works for other types of objects as well as user-created objects. The code below would display the contents of the text-face object created by a View layout:

```
out: layout [text-face: text "test"]

help text-face
```

Sub-objects can also be viewed. They are specified as paths. For example, to see the options object of the system object:

```
help system/options
```

The result would look something like this:

```
SYSTEM/OPTIONS is an object of value:
   home             file!     %/d/rebol/link/
   script           file!     %/d/rebol/link/ranch/utilities/console.r
   path             file!     %/d/rebol/link/ranch/
   boot             file!     %/d/rebol/link/rebol-link.exe
   args             none!     none
   do-arg           none!     none
   link-url         string!   "tcp://localhost:4028"
   server           none!     none
   quiet            logic!    true
   trace            logic!    false
   help             logic!    false
   install          logic!    false
   boot-flags       integer!  2064
   binary-base      integer!  16
   cgi              object!   [server-software server-name gateway-int
```

In addition, the search feature in help now provides more information about matches that were made. If you type:

```
help "to-"
```

You will now see:

```
Found these words:
   caret-to-offset native!   Returns the offset position relative to
   offset-to-caret native!   Returns the offset in the face's text co
   to-binary       function! Converts to binary value.
   to-bitset       function! Converts to bitset value.
   to-block        function! Converts to block value.
   to-char         function! Converts to char value.
   to-date         function! Converts to date value.
   to-decimal      function! Converts to decimal value.
   to-email        function! Converts to email value.
   to-event        function! Converts to event value.
   to-file         function! Converts to file value.
   to-get-word     function! Converts to get-word value.
   to-hash         function! Converts to hash value.
   to-hex          native!   Converts an integer to a hex issue!.
   to-idate        function! Returns a standard Internet date string.
   to-image        function! Converts to image value.
   to-integer      function! Converts to integer value.
   to-issue        function! Converts to issue value.
   to-list         function! Converts to list value.
   to-lit-path     function! Converts to lit-path value.
   to-lit-word     function! Converts to lit-word value.
   to-local-file   native!   Converts a REBOL file path to the local
   to-logic        function! Converts to logic value.
   to-money        function! Converts to money value.
   to-none         function! Converts to none value.
   to-pair         function! Converts to pair value.
   to-paren        function! Converts to paren value.
   to-path         function! Converts to path value.
   to-rebol-file   native!   Converts a local system file path to a R
   to-refinement   function! Converts to refinement value.
   to-set-path     function! Converts to set-path value.
   to-set-word     function! Converts to set-word value.
   to-string       function! Converts to string value.
   to-tag          function! Converts to tag value.
   to-time         function! Converts to time value.
   to-tuple        function! Converts to tuple value.
   to-url          function! Converts to url value.
   to-word         function! Converts to word value.
```

4.4 SUFFIX? Function Added

The SUFFIX? function is a helper function that returns the file "extension" portion of a filename or URL. For example:

```
>> suffix? %into.txt
== %.txt

>> suffix? http://www.rebol.com/docs.html
== %.html
```

```
>> suffix? %test
== none

>> suffix? %test.it/file
== none
```

SUFFIX? can be useful when selecting files from a directory. The example below will only select html and htm files:

```
foreach file load %. [
    if find [%.html %.htm] suffix? file [
        browse file
    ]
]
```

4.5 SIGN? Function Added

The SIGN? function returns a positive, zero, or negative integer based on the sign of its argument.

```
print sign? 1000
print sign? 0
print sign? -1000
```

The sign is returned as an integer to allow it to be used as a multiplication term within an expression:

```
new: 2000 * sign val

if size > 10 [xy: 10x20 * sign num]
```

4.6 COMPONENT? Function Added

COMPONENT? is a helper function that checks for the existence of a named REBOL component.

```
if component? 'crypt [print "Encryption available."]

if not component? 'sound [print "No sound."]
```

4.7 SEND Function Updated

The SEND function now includes a /SHOW refinement that will show all TO recipients in the header. By default, recipients are not normally shown.

```
send/show [bob@example.com fred@example.com] message
```

In addition the FROM field will include the sender's name as well as an email address. The string in system/user/name is used as the from address. For example, if:

```
```

```
system/user/name: "Fred Reboller"
```

Then when the email is sent, the from field will now appear as:

```
From: Fred Reboller <fred@example.com>
```

If system/user/name is NONE or empty, it will not be used.

Finally, some problems in the /ATTACH refinement have been fixed. Attachments of the form:

```
send/attach user content [[%file.txt "text message"]]
```

should work properly now. See 2.5.1 notes on SEND for more information about /ATTACH.

4.8 Miscellaneous Fixes

4.8.1 TYPE? (As used in PARSE)

Fixed the problem reported in the TYPE? function that caused the datatype error in PARSE. Code of the form below will work properly now:

```
parse [34] reduce [type? 34]
parse [34.5] reduce [type? 34.5]
```

4.8.2 MOLD/ALL

Fixed the problem with function molding when /ALL refinement is provided.

```
load mold/all context [test: func [arg] [print arg]]
```

4.8.3 FTP

Two fixes:

When connecting to a Microsoft IIS based FTP server Rebol did not recognize folders correctly. They were marked as type 'directory but no slash was appended to the file name as done with other types of FTP servers.

In FTP soft-links can now be identified. They can now return an file type of 'LINK.

Thanks: Cal and Reichart at Prolific.com for submitting the above FTP fixes.

5. Core 2.5.1

5.1 Source Code Form for Values of NONE, TRUE, etc.

To better support persistent values between SAVE, MOLD, and LOAD, a new syntactic element has been added to REBOL.

In the past, the source code representation of values like NONE and TRUE required evaluation in order to convert those words into their actual values. For example:

```
load "NONE"
```

would return the word NONE, not the value none. This required that your code either evaluate the result (with DO, REDUCE, or TRY) or manually convert the NONE word with code such as:

```
if value = 'none [value: none]
```

Thus, if you were storing REBOL data using MOLD or SAVE functions, when you read the data back in, you would need to either evaluate it (e.g. with REDUCE) or add checks for a few specific types of words (as shown above).

To better handle this situation with the literal expression of NONE, TRUE, FALSE, and other values that require evaluation, a new syntactic form has been added to the language:

```
#[datatype value]
```

Where datatype indicates the datatype and value provides its value. For simple datatypes such as NONE, TRUE, and FALSE the shorthand form:

```
#[word]
```

is also allowed. For example:

```
#[none]
```

expresses the value NONE, not the word NONE, even with unevaluated code or data. Similarly,

```
#[true]
#[false]
```

express the values of TRUE and FALSE within non-evaluated code.

The LOAD, DO, and other source translating functions have been expanded to accept this new syntax. The example below helps show the change:

```
block: load " none #[none] true #[true] "

foreach value block [print type? value]

word
none
word
logic
```

The literal expression of REBOL objects is also allowed by this new format. For example the non-evaluated expression:

```
#[object! [name: "Fred" skill: #[true]]]
```

is equivalent evaluating:

```
make object! [
    name: "Fred"
    skill: true
]
```

Thus, a persistent (mold/load symmetric) form of objects now becomes available to programmers.

To create or save source code or data using these new datatype expressions, the MOLD and SAVE functions have been given a new refinement called /ALL. This refinement will output the new #[] format for necessary values. For example:

```
mold/all reduce ["Example" true none time]
```

word return the string:

```
["Example" #[true] #[none] 5-May-2002/9:08:04-7:00]
```

Similarly, using SAVE/ALL would store the above expression to a file.

5.2 Less Aggressive Evaluation (Important!)

Variables are no longer aggressively evaluated.

In pervious versions of REBOL, variables that held set-words, parens, paths, and other types of evaluative datatypes would be evaluated on reference. That is no longer the case.

For example, in earlier versions, if you wrote:

```
a: first [b/c/d]
print a
```

the result would be an evaluation of b/c/d. This evaluation caused problems because it is more common to reference such data, not evaluate it. For instance, you want to be able to write:

```
data: ["test" A: mold/only (10 + 2.5)]

foreach value data [print value]
```

and smoothly traverse the elements of the data block as simple values. You would not want those values to self evaluate.

This change to evaluation has some deeper implications. For example, it affects any function that specifies literal values within arguments. For example, code such as:

```
f: func ['word] [print word]
```

The REBOL design definition of a literal function argument is to accept the argument always as a literal value and NOT evaluate it. However, in the actual implementation it turned out that an evaluation would normally occur when the variable was referenced. (As part of the PRINT in this case.)

For instance, if you passed:

```
f a/b/c
```

within the function, the reference to the word would cause the path to be evaluated at that point. Hence, if you referred to the word argument three times, each time it would be evaluated.

This implicit and "aggressive" evaluation, while interesting, was not intuitive and created to hard to find errors with difficult to interpret error messages.

Consider the case where the set-word datatype might be passed as the argument:

```
f test:
```

The set word is then evaluated where WORD appeared (in the PRINT line), and it would require an argument. The code would produce an error message that would tell you that the set required an argument. Yet, you would see no set within your code. It would look like an interpreter flaw, when in fact it was doing precisely what you asked.

Since the vast majority of REBOL programs do not rely on this type of evaluation, we've taken it out. This cleans up the semantics of literal arguments, but it also means that where programs depend on passing non-literals and assume that evaluation occurs, they do not. That will break some code, but it is rare. (For example, in the entire source code base of REBOL Technologies, there was only one such case.)

5.3 COMPOSE/ONLY Inserts Blocks

The COMPOSE function now has an /ONLY refinement similar to the INSERT/ONLY refinement. When /ONLY is specified, blocks that are inserted are inserted only as blocks, not as the elements of the block (the default). A REDUCE on the block is not required.

For example:

```
>> block: [1 2 3]

>> compose [example (block)]
== [example 1 2 3]

>> compose [example (reduce [block])]
```

```
== [example [1 2 3]]

>> compose/only [example (block)]
== [example [1 2 3]]
```

5.4 REMOVE-EACH - Easy Series Element Removal

The new function REMOVE-EACH works in a similar fashion to FOREACH and makes removing values from strings, blocks, and other series much easier.

Also, in most cases REMOVE-EACH is many times faster than using a WHILE loop and the REMOVE function.

The format of REMOVE-EACH is identical to FOREACH except that it uses the result of the block expression to determine removal. REMOVE-EACH iterates over one or more elements of a series and evaluates a block for each. If the block returns FALSE or NONE, nothing is removed, but if the block returns any other value, the items or items are removed.

```
remove-each value series [expression]
```

Similar to FOREACH, REMOVE-EACH can operate on multiple elements from the series (and uses the appropriate skip to get to the next set of elements):

```
remove-each [val1 val2 val3] series [expression]
```

Note that in addition to modifying the series, REMOVE-EACH also returns the series as its result.

Here are a number of examples.

To remove the odd numbers from block:

```
>> blk: [1 22 333 43 58]
>> remove-each item blk [odd? item]
== [22 58]
>> probe blk
== [22 58]
```

To remove all numbers greater than 10 from block:

```
>> blk: [3 4.5 20 34.5 6 50]
>> remove-each item blk [item > 10]
== [3 4.5 6]
>> probe blk
== [3 4.5 6]
```

To remove all directories from a directory listing:

```
>> dir: load %.
>> remove-each file dir [#"/" = last file]
```

```
== [%calculator.r %clock.r %console.r %find-file.r...]
```

To keep only the directories in the listing:

```
remove-each file load %. [#"/" <> last file]
```

To remove keep only the .doc files from the listing:

```
remove-each file load %. [not suffix? ".doc"]
```

Here is an example of a block that uses multiple values:

```
blk: [
    1 "use it"
    2 "rebol"
    3 "great fun"
    4 "the amazing bol"
]

remove-each [num str] blk [odd? num]

remove-each [num str] blk [not find str "bol]
```

REMOVE-EACH also removes characters from a string:

```
>>  str: "rebol is fun to use"

>>  remove-each chr str [chr = #"e"]
== "rbol is fun to us"

>>  remove-each chr str [find "uno " chr]
== "rblisfts"
```

5.5 ATTEMPT for Error Handling

The ATTEMPT function is a shortcut for the common REBOL idiom:

```
error? try [block]
```

The format for ATTEMPT is:

```
attempt [block]
```

ATTEMPT is useful in the cases where you either do not care about the error result or you want to make simple types of decisions based on the error.

```
attempt [make-dir %fred]
```

ATTEMPT returns the result of the block if an error did not occur. If an error did occur, a NONE is returned.

In the line:

```
value: attempt [load %data]
```

the value may be set to NONE if the %data file cannot be loaded (e.g. it's missing or contains an error). This allows you to write conditional code such as:

```
if not value: attempt [load %data] [alert "Problem"]
```

Or, even simpler:

```
value: any [attempt [load %data] 1234]
```

In this line, if the file cannot be loaded, then the value is set to 1234.

5.6 EXTRACT Function Updated

The EXTRACT function now includes an /INDEX refinement to allow you to specify what "column" you want to extract. For example:

```
data: [
    10 "fred" 1.2
    20 "bob" 2.3
    30 "ed" 4.5
]

>> probe extract/index data 3 2
== ["fred" "bob" "ed"]
```

If index is not supplied, then the first column is extracted.

In addition, the extract function has been fixed to properly extract blocks from a block series.

5.7 SAVE to Memory

The SAVE function can now write its results into memory. This is useful because the SAVE function contains formatting refinements that are found no where else.

To use SAVE to write to memory, specify a binary value rather than a file name. For example:

```
bin: make binary! 20000 ; (auto expands if necessary)

image: load %fred.gif
```

```
save/png bin image
```

Now the BIN binary holds the PNG formatted image. It was not necessary to write it to a file and read it back in.

5.8 SEND Refinements /Subject /Attach

The SEND function used for sending email has been extended to easily allow a subject line to be specified with a /SUBJECT refinement. This is a handy shortcut. For example:

```
letter: read %letter.txt

send/subject luke@rebol.com letter "Here it is!"
```

The above example will send the text of the letter with a subject line as specified. If the /SUBJECT refinement is not given, then the first line of the letter would be used as the subject.

In addition, the SEND function can now include attachment files by providing an /ATTACH refinement. A single file or multiple files can be attached. The specified files can be read from disk or can be provide as a filename and data. The formats are:

```
send/attach user letter file

send/attach user letter [file1 file2 ...]

send/attach user letter [[%filename data] ...]
```

You can also combine the last two above formats.

Example attachments are:

```
user: luke@rebol.com
letter: read %letter.txt

send/attach user letter %example.r

send/attach user letter [%file1.txt %file2.jpg]

send/attach user letter compose/deep [
    [%options.txt (mold system/options)]
]
```

5.9 Difference for Date/time

The DIFFERENCE function will now provide the time difference (hours, minutes, seconds) between two date/time values.

```
>> difference now 1-jan-2000
== 20561:20:26
```

Normally, when you subtract date/time the result is the number of days, not time:

```
>> now - 1-jan-2000
== 856
```

5.10 System Port Added

The system port component was added. This component allows direct access to various OS-specific features. For example, the system port can catch Unix and Linux termination signals and perform a controlled shutdown. Under Win32, this port can be used to access win-messages.

More information about the System Port component will be provided as a separate document.

6. Core 2.5.0

6.1 New Sort Function

Several changes and additions have been made to SORT to add functionality, including reverse sorting, hierarchical sorting (sorting on more than one field), sorting of only part of a series, stable sorting (items that are "equal" are not swapped during sorting), and easier specification of sort criteria (without the need for a custom comparator function). The new SORT function is fully backward-compatible.

6.1.1 Terminology

Record	A single logical item in the series to sort. Usually a character if the series to sort is a string, or a value if the series to sort is a block. If the /skip refinement is used then a record consists of multiple, consecutive elements in the series.
Field	A part of a record. With the /skip refinement and a skip length of n a field is one of n elements in the record. If the /skip refinement is not used and the series to sort is a block which contains sub-blocks, then a field is one item of a sub-block (NOT the complete sub-block). This allows field-wise sorting of blocked records. In all other cases a field is identical to a record (i.e. each record has exactly one field).
Field offset	An integer specifying the offset of a field within a record. For a record consisting of n fields a field offset can be between 1 and n.

6.1.2 Arguments

In addition to the series to sort, the new sort action accepts the following refinements. New behavior is marked with [NEW].

/case	Sort case-sensitive. This only has an effect for fields of type string or character.
/skip size	Treat series as records of fixed size. size is of type integer.
/all	Used in combination with the /skip refinement. [NEW] By default only a single field in a record is used for comparison. If the /all refinement is used then all fields in a record are used for comparison.
/compare comparator	Specify a custom comparator. This can be a field offset (type integer [NEW]), a comparator function, or a comparator block [NEW]. See below.
/reverse	Sort in reverse. [NEW]

/part size	Sort only part of a series. (Similar to the use of the /part refinement in copy or change [NEW]).

6.1.3 Comparators

Sorting is performed by comparing and swapping elements, i.e. comparators define the sort order. The following comparators are supported:

- No comparator (i.e. no /compare refinement). In this case the first field in each record is used for comparison. If the /all refinement is used then all fields in each record are used for comparison. /case and /reverse are observed normally.
- Field offset (integer) [NEW]. Specifies the field to use for comparison. If the /all refinement is used as well then the specified field is the first field to use, and all subsequent fields are used as well. /case and /reverse are observed normally.
- Function. Function called by the sort action to compare two records. The records are passed as arguments, and the function needs to return -1, 0 or 1, if the first record is smaller than, equal to or greather than the second record, respectively. [NEW] For backward compatibility the function may also return true or false, where true indicates that the first record is less than or equal to the second record, and false indicates that the first record is greater than the second record. /case is ignored. /reverse is observed normally, i.e. /reverse inverts the meaning of the comparator function. If the /skip refinement is used then the argument passed to the comparison function only consists of the first field in the record, not the complete record. This is for backward compatibility with the old sort function. In order to pass the complete record, in a block, use the /all refinement.
- Block [NEW]. Comparison dialect specifying the type of comparison in more detail. See below.

Comparison Refinements

The following items can appear in a comparison block:

field offset (integer)	Specifies the offset of the next field to compare. Fields are compared in the order specified.
reverse (word)	Sets the comparison for all subsequently specified fields to reverse.
forward (word)	Sets the comparison for all subsequently specified fields to forward (opposite of reverse).
case (word)	Makes the comparison for all subsequently specified fields case-sensitive.
no-case (word)	Makes the comparison for all subsequently specified fields case-insensitive (opposite of case).
to (word) field offset (integer)	Specifies a range of fields to compare, e.g. [1 to 5] compares fields one to five.

With comparison blocks the /all, /case and /reverse refinements slightly change their behaviors: /all is equivalent to adding "to max-field" to the end of the comparison block, i.e. when the end of the comparison block is reached comparison continues until the end of the record is reached. /case specifies that the default case mode (until a case or no-case word is reached) is case-sensitive. Otherwise it is case-insensitive. /reverse reverses the resulting list as a whole, and is independent of the reverse/forward words in the comparison block.

Examples

```
a: [10 "Smith" "Larry" 20 "Smith" "Joe" 80 "Brown" "Harry" 50 "Wood" "Jim"]
```

```
sort/skip a 3     ; sorts on the first field (number)
[10 "Smith" "Larry" 20 "Smith" "Joe" 50 "Wood" "Jim" 80 "Brown" "Harry"]
sort/skip/compare a 3 2     ; sorts on the second field (last name)
[80 "Brown" "Harry" 10 "Smith" "Larry" 20 "Smith" "Joe" 50 "Wood" "Jim"]
sort/skip/compare/all a 3 2
; sorts on the second field and following fields (last name and first name)
[80 "Brown" "Harry" 20 "Smith" "Joe" 10 "Smith" "Larry" 50 "Wood" "Jim"]
sort/skip/reverse a 3     ; sorts on the first field (number), in reverse
[80 "Brown" "Harry" 50 "Wood" "Jim" 20 "Smith" "Joe" 10 "Smith" "Larry"]
sort/skip/compare a 3 [2 reverse 3]
; sorts on the last name forward, and the first name in reverse
[80 "Brown" "Harry" 10 "Smith" "Larry" 20 "Smith" "Joe" 50 "Wood" "Jim"]
```

6.2 File Modes

GET-MODES and SET-MODES functions have been added for file and network ports. Two new port actions are introduced:

get-modes	Return current modes for an open port.
set-modes	Change modes for an open port.

The get-modes function has the following syntax:

```
get-modes: native [
    {Return mode settings for a port}
    target [file! url! block! port!]
    modes [word! block!]
]
```

The block being passed in consists of words defining which modes should be queried. Each word corresponds to one mode. get-modes returns a block which contains pairs of mode names and current mode settings.

Example:

```
>> get-modes someport [direct binary]
== [direct: true binary: false]
```

indicating that someport is opened in direct and non-binary (text) mode.

Alternatively a single word can be passed in, in which case get-modes returns the value directly, without putting it into a name-value block.

Example:

```
>> get-modes someport 'binary
== false
```

As another alternative a name/value-paired block can be passed in, of the same format as the block get-modes returns. In that case the values are ignored.

Example:

```
>> get-modes someport [direct: none binary: none]
== [direct: true binary: false]
```

The set-modes function has the following syntax:

```
set-modes: native [
    {Change mode settings for a port}
    target [file! url! block! port!]
    modes [block!]
]
```

The block being passed in consists of name-value pairs describing the modes to be changed. A block returned by get-modes can be passed as an argument to set-modes. set-modes returns the port that was passed as an argument.

Example:

```
>> set-modes someport [direct: false binary: false]
```

The mode block accepted by set-modes is actually an object-style initialization block and allows multiple names to reference the same value.

Example:

```
>> set-modes someport [direct: binary: false]
```

6.2.1 Getting Lists of Modes

get-modes supports a few "special" modes, which do not return mode settings for a specific port, but rather a set of modes that is applicable for a file (or directory, socket etc.). These are "file-modes", "copy-modes", "network-modes" and "port-modes". If any of these modes are specified in a get-modes request then the response contains a block of matching modes which are available on the current filesystem.

```
>> get-modes somefileordir 'port-modes
== [direct binary lines no-wait (...a few more...) ]

>> get-modes somefileordir 'file-modes
== [file-note creation-date archived script (...a few more...) ]

>> get-modes someudpsocket 'network-modes
== [broadcast multicast-groups type-of-service]
```

The above example is for an Amiga. Note that the mode block returned by any of these special requests is identical for all files and directories within a filesystem (and all sockets within a scheme), i.e. it varies per platform and possibly per filesystem and scheme, but not per file or socket. The purpose of this feature is only to provide a mechanism to obtain a list of supported modes, not to

obtain the actual mode settings. To obtain the actual mode settings for a port the returned block has to be used in another call to get-modes:

```
>> get-modes somefileordir get-modes somefileordir 'file-modes
== [file-note: "cool graphics" creation-date: 1-Jan-2000
    archived: true script: false]
```

The difference between "file-modes" and "copy-modes" is:

file-modes	Return all modes of the underlying data (typically a file), regardless of how it was opened.
copy-modes	Same as file-modes, but only return those modes which are safe to copy, i.e. which can (and should) be included in a set-modes call when creating a new file, in order to create an exact clone of an existing file. If a platform provides file properties which are not safe to copy or which necessarily vary on copied files (e.g. Unix inodes) then those modes would show up in the block returned by file-modes, but not by copy-modes.

Copying all file properties from file1 to file2 can be done with the following call:

```
>> set-modes file2 get-modes file1 get-modes file1 'copy-modes
```

6.2.2 Modes Available

File-modes

- status-change-date, modification-date, access-date, backup-date, creation-date: REBOL date. modification-date is available on all platforms. status-change-date is available in Unix. access-date is available in Unix and Windows. creation-date is available in Windows and MacOS. backup-date is available in MacOS. All modes are settable and gettable, except that modification-date and access-date are not settable in Elate.
- owner-name, group-name: REBOL string (user/group name). Available in Unix.
- owner-id, group-id: REBOL integer (user/group id). Available in Unix and AmigaOS (>=V39).
- owner-read, owner-write, owner-delete, owner-execute, group-read, group-write, group-delete, group-execute, world-read, world-write, world-delete, world-execute: REBOL logic. All -read, -write and -execute modes are available in Unix, BeOS, QNX and Elate. owner-read, owner-write, owner-execute and owner-delete are available in AmigaOS. All group- and world- modes are available in AmigaOS >=V39. owner-write is also available in Windows (mapped to the inverse of the Windows "read-only" bit).
- comment: REBOL string (file comment). Available in AmigaOS only.
- script, archived, system, hidden, hold, pure: REBOL logic (additional flags). Script, hold and pure are available in AmigaOS only. Archived is available in AmigaOS and Windows. System and hidden are available in Windows only.
- type, creator: REBOL string. Available in MacOS only.

Platforms which only support a single file date export it via modification-date. Similarly, platforms which do not support multi-user file access modes make the file access modes available via owner-read/write/delete/execute.

Port-modes

- read, write, binary, lines, no-wait, direct: REBOL logic. Binary, lines and no-wait are settable.

Network-modes

- broadcast: REBOL logic. UDP only. Setting this to true allows sending broadcasts from a socket. All platforms except BeOS.
- multicast-groups: REBOL block of blocks. UDP only, describes which multicast groups a socket has joined. Each sub-block consists of two IP addresses (tuples): the multicast group and the IP address of the interface on which the multicast group was joined. Typically available in MacOS, Windows, most Unix platforms, QNX and AmigaOS (Miami/MiamiDx only). Availability is determined at runtime and varies with the OS version and TCP/IP protocol stack version.
- type-of-service: REBOL integer. UDP and TCP. This is the value of the 8-bit "TOS" field in IP headers. Typical values are 0 (default), 2 (minimize cost), 4 (maximize reliability), 8 (maximize throughput) and 16 (minimize latency), but the interpretation of this field is up to the TCP/IP stack and intermediate routers, and not enforced by REBOL. All platforms except BeOS.
- keep-alive: REBOL logic. TCP only. Setting this to true forces TCP to send "keep-alive" packets after a certain period of time (typically 4 hours). All platforms except BeOS.
- receive-buffer-size, send-buffer-size: REBOL integer. UDP and TCP. Size of receive and send buffer within the TCP/IP stack. Default values and allowed ranges vary widely between platforms. Primarily useful for UDP. Increasing this values usually does NOT improve performance. All platforms except BeOS.
- multicast-interface: REBOL string. UDP only. The default interface to use for multicasting. Same platforms as multicast-groups.
- multicast-ttl: REBOL integer. UDP only. The time-to-live value (maximum propagation distance) of multicasts. Same platforms as multicast-groups.
- multicast-loopback: REBOL logic. UDP only. If set to true sent multicasts are looped back to the local socket. Same platforms as multicast-groups.
- no-delay: REBOL logic. TCP only. Disables the Nagle algorithm. Most protocols perform better if this is set to false. It should only be set to true if a protocol is interactive in nature AND relies on precise event timing in combination with queueing (e.g. X11). All platforms except BeOS.
- interfaces: REBOL block of objects. Each object represents one network interface and currently contains the following fields: "name": interface name (on some platforms this is a dummy name). "address": local IP address. "netmask": subnet mask. "broadcast": broadcast address. "remote-address": remote IP address for point-to-point interfaces. "flags": a block of words describing interface properties, currently supported words are "broadcast" (interface supports broadcasting), "multicast" (interface supports multicasting), "loopback" (interface is the loopback interface) and "point-to-point" (interface is a point-to-point interface, as opposed to a multi-drop interface). Some values may be none (e.g. "remote-address" on a multi-drop interface). In Windows all interfaces appears as multi-drop (including PPP), with dummy netmasks. Available in all platforms, except BeOS, Elate, WinCE. Not settable.

6.2.3 Using File Forks

The /custom refinement to read, open and write allows access to different forks of a file (currently useful for MacOS only).

```
open/custom %myfile [fork "name"] ; Specify which fork to open.
```

The specified fork name defines which fork to open:

If no fork is specified (no fork specifier in the custom block or none is passed instead of "name"), then for non-Mac platforms the file is opened normally. For Mac the data fork is opened. If the /new refinement is used then the file size is reset to zero bytes. For Mac this means that all forks are reset to zero bytes. (Note: this behavior has changed from Core 2.3. Previously open/new on Macintosh would only reset the size of the data fork, and there was no way to reset the size of the resource fork.)

If a fork is specified by name then that fork is opened. If the /new refinement is used then the size of only that fork is reset to zero bytes. If the fork does not exist (and, for write access, cannot be

created) on the current filesystem then an error is thrown.

Non-Mac platforms only have a single fork named "data". Mac platforms have two forks, named "data" and "resource".

6.2.4 Finding All Forks

This is done using the mode mechanism, in a way similar to finding all supported modes.

```
>> get-modes somefile 'forks
== ["data" "resource"]
```

Example above on Mac. For all other platforms only ["data"] would be returned.

6.3 Serial Port Access

This specification describes the creation and operation of serial communication ports within REBOL. Serial ports were supported in version 2.3 but were not documented.

6.3.1 Specifying a Serial Port

Serial ports are created in the same manner as other ports within REBOL. The scheme name for serial ports is "serial:". URL's are encoded with the different fields separated by slashes. For example,

```
port: open serial://port1/9600/8/none/1
```

The order of values in the serial URL scheme is not significant, as the type of field can be determined by the content. (Note, you can also use a MAKE PORT object rather than a URL to specify a serial port.)

The specification of a serial port may include the device number, the communication speed, the number of data bits, the parity and number of stop bits. The specification information can be specified directly by setting the appropriate fields within the port specification object or by creating a URL which contains the same information. Any field not specified will be given a default value.

The default serial port settings are:

```
device: port1
speed: 9600
data-bits: 8
parity: none
stop-bits: 1
```

Within a port specification, the various parameters are stored in the following object fields:

```
host:           device
speed:          speed
data-bits:      data bits
parity:         parity
stop-bits:      stop bits
```

The portN specification is an indirect reference to the communication device. It refers to the Nth device defined in the System/ports/serial block. This block is initialized by default depending on the system being used and can be modified in user.r to reflect any local requirements.

For example, on Windows the block might be defined as:

```
System/ports/serial: [ com1 com2 ]
```

or if COM1 is being used by the mouse, it might just be:

```
System/ports/serial: [ com2 ]
```

On unix-style systems, the block might be defined as:

```
System/ports/serial: [ ttyS0 ttyS1 ]
```

or if the first device should correspond to COM2:

```
System/ports/serial: [ ttyS1 ttyS0 ]
```

Thus, the ports can be specified in a machine indpendent manner while the machine specific definition can be controlled using the serial definition block in System/ports/serial.

6.3.2 Operation

Serial ports are always opened as direct ports in much the same way as console and network ports. They may be opened as either /STRING or /BINARY with the default being /STRING. They are opened by default as asynchronous, but may be made synchronous by using the /WAIT refinement. When operating asynchronously, any attempts to copy data from the port will return NONE if no data is present. When operating synchronously, the copy will block until data is available.

Data can be written to the port using the INSERT native. Data can be read from the port using the PICK, FIRST or COPY natives with the usual ramifications. As usual with direct ports, the REMOVE, CLEAR, CHANGE and FIND functions are not supported.

The UPDATE function can be used to change port parameters. For example, to change the speed after an initial connection has been established, you could just do:

```
ser-port: open serial://9600
ser-port/speed: 2400
update ser-port
```

Changing the device number or the System/ports/serial and calling UPDATE will have no effect. Once the port has been opened with a particular device, the device can't be changed.

There are two additional port fields that can't be set with a URL, but can be set in a port specification block or by manually changing them. The RTS-CTS field specifies whether hardware handshaking should be used on the port. The default is ON. To change the default, do:

```
ser-port/rts-cts: off
update ser-port
```

A timeout value can be specified by modifying the timeout field in the port structure. The timeout value only applies to serial ports that are opened with the /wait refinement. When the timeout expires, a serial port timeout error will be generated. To set the timeout value do:

```
ser-port/timeout: 10        ; 10 second timeout
```

Serial ports work properly with the WAIT native.

6.4 Objects

- SYSTEM/LOCALE object added. Currently has month and day names. *Weekdays are separate strings in system/locale/days.
- CONTEXT function added as a shortcut to MAKE object!.
- You can now QUERY an object. QUERY will return a block containing the fields that have been modified in an object since its creation or since it was last passed to QUERY/clear. QUERY will return none if there have not been any changed fields. QUERY/clear will clear the modified state of all the fields in an object.
- Objects now accept LOGIC! values as arguments for to pick (for consistency with blocks).

6.4.1 Make Object Object

You can now make an object out of two other objects, a "template object" and a "spec object". When making a new object out of two other objects, the bindings of words coming only from the "spec object" will be maintained, words shared by both the template object and the spec object will take their values from the spec object. This will cause some differences in behavior because the block spec approach is lexical and the object spec approach is definitional.

6.4.2 Third Object

You can use THIRD on the object to get back a block of set-word value pairs from the object:

```
z: make object! [a: 99 b: does [a + 1]]
t: third z
== [a: 99 b: func [][a + 1]]
```

The block returned from THIRD on an object is like an object spec block, however the set-words are bound to their object. This block is like a snapshot of the object's values at that time. You can also use the returned block to set values in the object:

```
set first t 199
z/b
== 200
```

6.5 Mold and Load Changes

- MOLD/only does not produce the outermost [] in the resulting string.
- Molding a recursive block or object will print [...] when on the second instance of the recursive

item. So, you can now PRINT MOLD SYSTEM !
- Molding strings greater that 50 characters containing unbalanced "{" characters are now reloadable.
- Added /all refinement to LOAD. LOAD/all of a script does not evaluate the REBOL header. LOAD/all always returns a block.
- LOAD/next/header results in a block with the evaluated header followed by the rest of the script as a string.
- Loading strings with CTRL chars greater than CTRL-Z now allowed.
- Added script HEADER/CONTENT field. When set TRUE, the script's entire source text can be accessed from SYSTEM/SCRIPT/HEADER/CONTENT (when evaluated with DO) or from the header object (when loaded with LOAD/header or LOAD/next/header). This allows your script to access other data that may be part of its script file (e.g. the REBOL archive format, RIP).

LOAD function refinement combinations:

LOAD/Refinement	Eval Header?	Returns
load	yes	returns script (data). If file contains a single value (like 1234), then this returns just that value. If there are multiple values, then it returns a block.
load/next	no	[first-val { rest of script }] -- returns a value followed by a string. This refinement lets you parse REBOL files, one value at a time (for operations like pretty printing and colorizing.)
load/next/header	yes	[header-obj {rest of script}] -- same as load/next but includes the evaluated header object.
load/header	yes	[header-obj rest of script] -- returns the script, starting with the REBOL header as the first value.
load/all	no	[all of script as data] -- returns the script, always as a block, even if it's just a single value (very handy for cases where you always want data as a block, even if it's a single value).
load/next/all	no	[first-val { rest of script }]
load/header/all	yes	[header-obj rest of script]
load/next/header/all	yes	[header-obj {rest of script}]

The ALL refinement will be ignored when other refinements are present.

6.6 File and Port Changes

- Added asynchronous wait-able DNS for Unix and Windows. For example: open dns:///async, then insert/wait/copy.
- Asynchronous HTTP better supported. You can open/direct/no-wait an HTTP request and use WAIT and COPY to receive the results as they arrive.
- WAIT [0 port] now returns the port if it has data, none otherwise.
- Added WAIT/all refinement, which causes wait to return a block of all ports that have data.
- The /no-wait refinement allows a port to be opened in non-blocking mode. COPY on a non-blocking port returns an empty string unless the end of the port has been reached, in which

case it returns none.
- WAIT now supports port handlers (http, tcp etc.) in /direct mode.
- Added AWAKE field in port-specs and root protocol to specify a block or function called when wait is about to wake up. Used to implement background processing.
- WAIT works correctly on /lines ports opened without /with.
- TO-LOCAL-FILE and TO-REBOL-FILE functions can be used to convert to and from the local OS file path formats.
- Added local-port, remote-ip and remote-port fields to port spec for consistent values independent of port creation.
- MAKE-DIR/deep creates all needed directories in a long path.
- CONNECTED? native for most platforms.
- Fixed incorrect port-id assignment on accepted sockets.
- Fixed bug that would prevent UDP listen sockets from responding to packets from different origins.
- SPLIT-PATH was modified to properly split the path and target of a file path.

6.7 Network Protocol Change (APOP, IMAP)

- Added APOP - Authentication for pop:// that does not send passwords to the server in clear text.
- Added imap:// protocol. URL format and behavior identical to pop://, plus additional URL formats specified in RFC 2192.
- Fixed SEND when /header and no FROM address is given
- Fixed do-send to change email lines with only a single "." to have two periods so those emails will not blow out anymore.

6.8 Data Series Changes

- HASH! and LIST! datatypes now offer more complete, consistent, reliable, and faster operation.
- The SELECT, FIND, UNION, INTERSECT, EXCLUDE, DIFFERENCE, and UNIQUE functions accept a /skip refinement to specify data size.
- UNIQUE accepts the /case refinement.
- TO-BINARY of tuples now uses the tuple values instead of forming the tuple.
- JOIN on binary values now joins as binaries, not FORMed strings.
- FIND now works on all types of functions properly.
- FIND/REVERSE works for bitsets.
- Pick on objects and functions with negative number fixed.
- Fixed bug in ENBASE that could cause invalid characters to be inserted into a base-64-encoded string.
- Removed /only refinement from DIFFERENCE (EXCLUDE is equivalent).
- MINIMUM-OF and MAXIMUM-OF series. You can use MINIMUM-OF and MAXIMUM-OF on a single series argument which will return the series offset to the least or greatest contained value respectively. Eg:

```
minimum-of reduce [pi .099 10 * 100]
== [0.099 1000]

maximum-of reduce [pi .099 10 * 100]
== [1000]
```

6.9 Math Related Changes

- Added CHECKSUM/secure and RANDOM/secure, producing cryptographically secure checksums and random numbers respectively.
- Added CHECKSUM/hash and CHECKSUM/method (same as checksum/secure, but with a selectable algorithm: 'md5 or 'sha1), and checksum/key (calculates a keyed message digest).

MD5 added as an alternative checksum algorithm.
- Subtraction of a date! value months bug fixed.
- Time! values can convert to integers and decimals.
- NOW/precise returns greater precision on time. Useful for timing events.

6.10 Command Line Changes

- REBOL startup command line now accepts "--" to signal the end of startup switches. Remaining items on the command line are arguments to be passed to the REBOL script. This also allows REBOL to start without a script but with arguments passed.
- When starting from the command line, system/script/args == string of all args, system/options/args == block of separate args. All items following the specification of the script are considered arguments.
- REBOL now deals with arbitrary amount of command line arguments-- no fixed size.

6.11 Console

- Console tab completion now completes as much of the defined word or file path as possible.
- Added proper console reinitialization code when REBOL is woken up after Ctrl-Z followed by "fg" (for Unix and shells with job control).
- File name completion in the console is now case-insensitive for operating systems that have case-insensitive file systems (Windows, AmigaOS etc.)
- Screen now gets cleared on first console output, not signon message.

6.12 Control

- A BREAK used in the first block of a WHILE will cause a BREAK from the while loop.
- CATCH and THROW work more reliably in more cases.
- FOREACH returns correct value on empty series.

6.13 Interpreter

- Binding speed greatly improved for top-level words.
- Indefinite extent fixed for many cases (the USE function).

6.14 Other Changes

- HAS function added as a shortcut to define functions with local variables but no arguments.
- Fixed CONFIRM to throw error on bad pick operation.

7. Interpreter Fixes

The following internal interpreter problems have been fixed.

1. Series Expansion: Significant performance and memory problem has been fixed resulting in many times greater speed in some types of uses. This change speeds up MOLD and SAVE by many times.
2. Lit-path! comparison works properly
3. Time zone bug in date comparison fixed
4. Handling of "--" in command line arguments
5. Tab expansion of words in console
6. Component version numbers added to the boot message
7. Increased the global word limit to 8000
8. Potential crash during expansion and recycling of hash! values
9. Potential crash when recycling open, unreferenced ports
10. Port cleanup to close ports in the correct order

8. Networking Fixes

The following fixes were made to networking:

1. DNS - Unix only: fixed lockup when DNS helper process dies unexpectedly.
2. DNS - Unix only: DNS helper process no longer remains a "zombie" if the main process is killed.
3. FTP - conflicts with generic proxy setups fixed
4. FTP - allow more response codes from CWD command in FTP
5. HTTP - proxy authentication added
6. HTTP - POST linefeed conversion problem fixed
7. HTTP - forwarding fixed where host name in header was not updated
8. POP - will try APOP if port/algorithm is set to 'apop (workaround for bug in some POP servers).
9. TCP - On Windows only: TCP listen sockets no longer allow multiple listeners on the same port. Workaround for bug in Windows TCP/IP stack regarding SO_REUSEADDR.

9. Other Function Fixes

Fixes to natives and functions include:

1. COMPRESS/DECOMPRESS - series offsets are now allowed
2. DEBASE - potential crash fixed
3. DEBASE - series offsets are now allowed
4. FIND - when using bitset with chars greater than 127
5. FIND/MATCH/CASE - fixed
6. LOAD/MARKUP - potential crash fixed
7. MAKE - potential crash with make object! object! fixed
8. MAKE-DIR/DEEP - fixed bug
9. SET-MODES - fixed file date setting when no time value is provided
10. SORT - handles hashed blocks now, and potential crashes fixed
11. UNION, INTERSECT, etc. - series offsets are now allowed on the second argument

10. Summary of New Features in 2.3

The following table shows a summary of new features in REBOL/Core 2.3:

- load/markup refinement - returns a block of tags and text
- repend function - reduces argument before appending
- replace/case - allows case sensitive replace
- ?? function - prints useful debugging function
- to-pair function - converts values into pairs
- pair datatype - new datatype that represents vectors and coordinates
- offset? function - returns the offset between series positions
- does function - shortcut for creating functions without arguments
- unique function - returns a series with duplicates removed
- functions accepting pair! - many existing functions now accept the pair value
- make-dir/deep refinement - allows creation of multiple depth directories

11. Summary of Enhancements in 2.3

The following table shows a summary of enhancements made to existing functions and global objects in REBOL/Core 2.3:

- help function - information is formatted differently. Help can now perform searches over internal documentation
- http user-agent - Http handler now allows setting the http user-agent
- value/selector return - returns the set value, not the aggregate value

- random function - now operates on negative values and tuples
- system/options/path - indicates directory the running rebol script is in
- what function - changed the way functions are listed; they are sorted now
- build-tag - accepts a wider range of values
- parse - to parse blocks and strings
- if, any and all functions - changed false return values to none
- pop handler - contains information on messages sizes
- and, or, xor - AND, OR, and XOR now work on binary types. The resulting binary will be as long as the longest of the two binary values. The bitwise operations use the binary value's offset to determine where to start.
- functions accepting pair! - many functions now accept the pair value
- query function - changed to allow the querying of objects to determine changed words
- load/next - changed behavior with respect to script header
- tab completion - changed to show a list of possibilities upon hitting tab twice

12. Summary of Fixes in 2.3

The following table shows a summary of fixes incorporated into REBOL/Core 2.3:

- to-block function - fixed to work properly on path literals
- to-word function - fixed to properly convert set-word values into word values
- trim/lines - fixed so that it removes all extra white space
- Networking - Network activity can now be interrupted with the escape key
- split-path function - fixed so that the lowest level path splits correctly

13. New Functions and Refinements in 2.3

The following new functions and refinements are incorporated in to REBOL/Core 2.3:

13.1 Pair Datatype

Pairs are datatypes used to describe the (x,y) coordinates of a point. This datatype is most commonly used to describe graphical objects and their screen positions, sizes offsets, etc. A pair datatype consists of two values separated by an upper or lowercase 'x' as in 10x20. Example: `A: 10x20`, defines `A` as a pair datatype consisting of the coordinates `x=10`, `y=20`. These values can be individually accessed using the refinements **`/x`** and **`/y`** as in: `print A/x`, which would print the value `10`.

13.2 Make-dir/deep

Using the **`/deep`** refinement for **`make-dir`** will cause all directories in the supplied path to be created if those directories do not already exist.

The following example uses **`make-dir/deep`** to create a long directory structure:

```
probe dir? %/c/abc
false
make-dir/deep %/c/abc/def/ghi/jkl/mno/pqr/stu/vwx/yz/
probe dir? %/c/abc
true
probe dir? %/c/abc/def/ghi/jkl/mno/pqr/stu/vwx/yz/
true
```

If **`make-dir/deep`** fails any directories that were created before failing will be removed.

13.3 Unique

Unique accepts a series argument and returns another series. The result of **unique** is a copy of the passed in series with out any duplicate items.

Here are some examples demonstrating the behavior of **unique:**

```
probe unique "abcabcabc"
"abc"
probe unique [1 2 3 3 2 1 2 3 1]
[1 2 3]
```

The **/case** refinement may be used with **unique** to make the operation case sensitive.

13.4 Does

Use **does** to create functions with out arguments or locally defined words. For example:

```
get-script-dir: does [probe system/options/path]
probe get-script-dir
%/usr/local/rebol/
```

13.5 Offset?

Offset? takes two series as arguments and returns the distance between their two indexes.

The following example shows **offset?** being used on the same series:

```
colors: ["red" "green" "blue" "yellow" "orange"]
colors-2: skip colors 3
probe offset? colors colors-2
3
```

The example below shows how **offset?** can also be used with two different series:

```
str: "abcdef"
blk: skip [a b c d e f] 4
probe offset? str blk
4
```

13.6 Load/Markup

Using the /markup refinement for **load** causes the string, file or URL argument to be treated as markup data. (Files and URLs are first read in as strings by **load**.) All tags found in the argument to **load/markup** will be converted to a tag! value, and all other non tag data is left as a string. The result of **load/markup** is a block of tag! values and strings.

The next example demonstrates the behavior of **Load/markup:**

```
probe load/markup {<head> 123 abc <head>}
[<head> { 123 abc  } </head>]
```

13.7 Repend

Repend is short for "reduce append". **Repend** takes two arguments. The first argument is a series. The second argument to **repend** is reduced before it is appended to the first argument.

Here are some examples of using **Repend:**

```
a: "AA" b: "BB" c: "CC"
probe repend [] [a b c]
["AA" "BB" "CC"]
probe repend "" [a b c]
"AABBCC"
```

13.8 Replace/case

The **replace** function now has a **/case** refinement that forces case sensitive replacements. By default **replace** is case insensitive.

This example demonstrates using **replace** on a block without **/case:**

```
probe replace/all [A a B b A a B b] 'A 'C
[C C B b C C B b]
```

This example demonstrates replace on a block with **/case:**

```
probe replace/all/case [A a B b A a B b] 'A 'C
[C a B b C a B b]
```

This example demonstrates using **replace** on a string without **/case:**

```
probe replace/all "ABCabcDEFdefABCabcDEFdef" "abc" "xxx"
"xxxxxxDEFdefxxxxxxDEFdef"
```

This example demonstrates using **replace** on a string with **/case:**

```
probe replace/all/case "ABCabcDEFdefABCabcDEFdef" "abc" "xxx"
"ABCxxxDEFdefABCxxxDEFdef"
```

13.9 ??

?? accepts one argument of any data type.

As an aid to debugging, if a word is supplied to **??** that word will printed out along with the mold of its value. Like the behavior of **probe**, **??** returns the unevaluated value of its argument so that **??** can be used transparently within arbitrary REBOL expressions.

When **??** is used directly on a non word value the molded value is printed and the unevaluated value is returned.

The next example demonstrates using **??** on a word defined to a block:

```
blk: ["a" block 'of [values]]
?? blk
blk: ["a" block 'of [values]
```

The next example demonstrates using **??** on a word defined as a function:

```
?? probe
probe: func [
    {Prints a molded, unevaluated value and returns the same value.}
    value
][
    print mold :value :value
]
```

The next example demonstrates using **??** on a non word value:

```
?? 12:30pm
12:30
```

13.10 To-pair

To-pair accepts one argument of any data type.

To-pair attempts to convert the supplied argument into a pair value.

The next example demonstrates using **to-pair** with a block:

```
probe to-pair [11 11]
11x11
```

The next example demonstrates using **to-pair** with a string:

```
probe to-pair "11x11"
11x11
```

14. Enhancements in 2.3

14.1 Parse Block

Under REBOL/Core 2.3 **parse** can be used on blocks. The ability to create dialects in REBOL is enhanced through the ability to parse blocks.

Refer to the User's Guide for more information on using **parse** with blocks.

14.2 POP handler change

The POP mail handler has been enhanced to store the total size of messages and the sizes of individual messages. The total size of messages can be found stored as an integer representing bytes in the POP port's locals object under the total-size field. Individual message sizes are stored as a block of message numbers and byte sizes in the port's locals object under the sizes field. Examples:

Total size of all messages in a POP connection:

```
pop: open pop://login:pass@mail.site.com/
size: probe pop/locals/total-size
735907
print join to-integer size / 1024 "K"
718K
```

Sizes of individual messages:

```
pop: open pop://login:pass@mail.site.com/
sizes: pop/locals/sizes
foreach [mesg-num size] sizes [
    print [
        "Message" mesg-num "is" join to-integer size / 1024 "K"
    ]
]
Message 1 is 2K
Message 2 is 2K
Message 4 is 1K
Message 5 is 9K
Message 6 is 1K
Message 7 is 678K
Message 8 is 1K
```

14.3 Tab Completion

Tab completion for the REBOL console has been expanded to show a list of possible matches when pressing tab a second time.

To see the behavior of REBOL's tab completion, type `fun` at the REBOL prompt and then hit the tab key. Tab completion will expand `fun` to `func`. Hitting the tab key once more will show the following: `function!`, `function?`, `func` and `function` as possible matches for the word you are typing in.

14.4 Load's /Next Refinement

The behavior has changed for **load's /next** refinement. **Load/next** will not skip over a REBOL script header. To load a script header as an object using the **/next** refinement, include the **/header** refinement. Examples:

Saving a simple script to the file simple.r:

```
save %simple.r [
    REBOL [
        title: "simple"
        file:  %simple.r
        ]
    print "simple script"
]
```

Using **load/next** on a simple script:

```
script: probe load/next %simple.r
[
    REBOL { [
    title: "simple"
    file: %simple.r
]
print "simple script"}]
probe load/next second script
[[
        title: "simple"
        file: %simple.r
    ] {
print "simple script"}]
```

Using **load/next/header** on a simple script:

```
script: probe load/next/header %simple.r
[
    make object! [
        Title: "simple"
        Date: none
        Name: none
        Version: none
        File: %simple.r
        Home: none
        Author: none
        Owner: none
        Rights: none
        Needs: none
        Tabs: none
        Usage: none
        Purpose: none
        Comment: none
        History: none
        Language: none
    ] {
print "simple script"}]
probe load/next second script
[
    print { "simple script"}]
```

14.5 Query Function

Query now accepts objects, returning a block of modified words in that object or **none**.

The block returned from using **query** on an object contains the words which have been modified since the object's creation or since it was last passed to **query/clear**.

The words in the returned block are bound to the context of the object. Using **query**, changes to an object can be tracked.

The following example illustrates the use of **query** on objects:

Defining an object:

```
obj: make object! [
    field-one: "string"
    field-two: 123
    field-three: 11:11:01
]
```

Using **query** on obj immediately after the object was created will return **none:**

```
probe query obj
none
```

Using **query** after a field within obj has changed:

```
obj/field-two: 456
obj/field-three: 12:12:02
mod: query obj<br>
probe get mod/1
456
probe get mod/2
12:12:02
```

Query returned a block of modified words bound to obj.

Use **query/clear** to reset the modified state of the object.

The next example demonstrates clearing the modified state of the fields in obj:

```
query/clear obj<br>
probe query obj
none
```

Query on **obj** returned **none** because the modified state was reset using **query/clear**.

14.6 Functions Accepting Pair Values

Following is a list of the functions that have been extended to accept pair values:

- Addition Functions + and Add

```
probe 12x12 + 12x12
24x24
probe add 11x11 11
22x22
```

- Subtraction Functions - and Subtract

```
probe 24x24 - 12x20
12x4
probe subtract 24x24 12
12x12
```

- Multiplication Functions * and Multiply

```
probe 12x12 * 12x12
144x144
probe multiply 12x12 24
288x288
```

- Division Functions / and Divide

```
probe 12x12 / 2x4
6x3
probe divide 24x24 4
6x6
```

- Remainder Functions // and Remainder

```
probe 24x24 // 5x3
4x0
probe remainder 24x24 10
4x4
```

- Minimum Functions Min and Minimum

```
probe min 12x12 12x11
12x11
probe minimum 23x24 24x24
23x24
```

- Maximum Functions Max and Maximum

```
```

```
probe max 12x12 12x11
12x12
probe maximum 23x24 24x24
24x24
```

- Negate Function

```
probe negate 12x12
-12x-12
probe negate 12x0
-12x0
```

- Absolute Functions Abs and Absolute

```
probe abs -12x-24
12x24
probe absolute -12x24
12x24
```

- Zero? Function

```
probe zero? 12x0
false
probe zero? 0x0
true
```

- Pick, First and Second Functions

```
probe pick 24x12 1
24
probe pick 24x12 2
12
probe pick 24x12 3 ; anything beyond 2 returns NONE
none
probe first 24x12
24
probe second 24x12
12
```

- Reverse Function

```
probe reverse 12x24
24x12
```

14.7 Random Function

The **`random`** function now takes negative values. When a negative value is used, **`random`** returns a number between -1 and the negative value. Examples:

```
loop 5 [print random -5]
-5
-5
-3
-3
-5
loop 5 [print random -$5]
-$4.00
-$5.00
-$3.00
-$5.00
-$3.00
```

14.8 System/options/path

The word path has been added to the system/options object. System/options/path is the directory where the current REBOL script was executed from. Example:

```
print system/options/path
/C/REBOL/
```

14.9 What Function

What now alphabetically sorts the list of defined functions printed.

14.10 If, Any, and All Functions

If, **any** and **all** now return **none** as a "fail" value where they had previously returned **false**.

Here are some examples of the new return value:

```
val: 10
probe if val = 100 [print "matched"]
none
val: 10
probe any [val = 100 val = 1000 val = 10000]
none
set [val1 val2 val3] [10 100 2000]
probe all [val1 = 10 val2 = 100 val3 = 1000]
none
```

14.11 Help

Use the **help** function to get the description, arguments, and refinements for all functions, or to inspect other REBOL values. Type **help** and the function name at the prompt:

```
>> help cosine
USAGE:
    COSINE value /radians
DESCRIPTION:
```

```
     Returns the trigonometric cosine in degrees.
     COSINE is a native value.
ARGUMENTS:
     value -- (Type: number)
REFINEMENTS:
     /radians -- Value is specified in radians.
```

Help can also be used to search REBOL's internal documentation. Here is an example of using **help** with a word:

```
>> help path
Found these words:
    clean-path     (function)
    inst-path      (object)
    lit-path!      (datatype)
    lit-path?      (action)
    make-dir-path  (function)
    path!          (datatype)
    path?          (action)
    set-path!      (datatype)
    set-path?      (action)
    split-path     (function)
    to-lit-path    (function)
    to-path        (function)
    to-set-path    (function)
```

15. Fixes in 2.3

Here are the fixes made since version 2.3.

15.1 Split-path Function

When splitting the lowest level of a file path (%. or %/), the second value in the returned block will be **none**. When splitting a file (%file), the first value in the returned block will be the current directory (%./) and the second value the file itself (%file).

Here are some examples of using **split-path:**

```
probe split-path %.
[%./ none]
probe split-path %./
[%./ none]
probe split-path %/
[%/ none]
probe split-path %file.txt
[%./ %file.txt]
```

15.2 Network activity interruptible

Network activity may now be interrupted by pressing the escape key (esc).

A known exception is network activity on the BeOS version of REBOL/core which currently is not

interruptible with the escape key.

15.3 Trim/lines

Trim/lines now removes all extra white space in a string, compressing all occurrences of whitespace into single spaces.

The following example demonstrates the behavior of **trim/lines:**

```
str: {
    line one
        line two
    line five
    line eight
}
probe trim/lines str
"line one line two line five line eight"
```

15.4 To-word Function

To-word now accepts **set-word** values as an argument.

The behavior of **to-word** with a **set-word** as an argument is demonstrated in the following example:

```
probe to-word first [data: "string of data"]
data
probe type? to-word first [data: "string of data"]
word!
```

15.5 To-block Function

When a path literal is used as an argument to **to-block**, a block is returned containing the path components as words.

The following example demonstrates the use of **to-block** with a literal path value:

```
probe to-block 'system/options/path
[system options path]
```

16. Other Changes (From Addendum Document) in 2.3

- REBOL 2.3 has an additional +q (non-quiet) command line flag
- DIFFERENCE renamed EXCLUDE, though DIFFERENCE is retained as a synonym for EXCLUDE
- IF has a new /else refinement
- PARSE now accepts any series instead of just a string
- TRACE has a new /function refinement which allows you to trace function calls
- SLASH and BACKSLASH are pre-defined values
- DIRIZE is a function which turns a file into a directory if the file is not a directory already
- APPEND has new /only refinement which behaves like INSERT/only

- The system object contains a COMPONENTS block for future additions
- The systen object now contains user/home which holds the location of the $HOME environment variable
- These fields have been added to ports: localip, localservice, remoteservice, lastremoteservice, direction, key, strength, algorithm, block-chaining, init-vector. Some of these fields are necessary for security features of other REBOL products.
- The words action! and unset! no longer are unset.
- system/options/home/public/ is writeable by default
- The following other datatypes exist in REBOL/core 2.3 although they are not all usable: event! library! struct! routine! image!
- when a path is used to set a value in a datatype, and the datatype is a scalar, it sets it in the scalar that was used in the path. i.e., date/minute: 1 now data/minute will give 1
- Read or load of %/ now returns drives on WIN32 and Amiga.
- Molding an object! with a field set to a literal word value is now handled correctly
- UDP listen ports work now
- Support has been added for COPY using UDP ports
- Scripts can now specify the version and product required to run the script in the REBOL header NEEDS field
- RANDOM now works on tuples
- Poking a tuple is now clipped at 0 and 255
- New native connected? attempts to determine if the computer is connected to the internet
- The following are functions and natives found in REBOL 2.3 which are disabled: FREE BROWSE LAUNCH

REBOL Interim Builds

Updates for Expert REBOL Developers
Updated: 8-Sep-2005

These directories hold raw builds not distribution packages. They include the most recent executable files for a variety of REBOL developer products. Many of these builds are not tested. Use at your own risk.

Note: SDK and Command products require your valid license.key file in order to enable the extended product features. Purchasing REBOL products helps us to make further changes and improvements. If you have not purchased products in the past, please consider doing so in the future.

Download Directories

Locate the directory below that matches your platform. Within that directory, the files are listed as **raw binary executables** (not zip or tg) using version encoded filenames. Click the "last modified" link to sort file dirs by release date. Note that you may need to right-click to download (because some files have no extensions) or use REBOL (see bottom of page).

- 031 - Windows 32 (All x86 versions)
- 024 - Mac OS X
- 042 - Linux x86 Libc6
- 092 - OpenBSD x86 elf
- 101 - Sun Solaris

Releases

Posted	Class	Platform	Product	Description
8-Sep-2005	Beta	Linux PPC 044	Core 2.6.0a	This is an initial test build of REBOL for the Genesi PPC running Debian and Gentoo Linux. This is our first non-OSX PPC build for quite some time. Please let us know how it works for you.
3-Sep-2005	Beta	Win32 031	Core 2.6.0d View 1.3.1d	Fixes RAMBO 3885, 3895, 3896. (Rare bind and GC bugs)
1-Sep-2005	Beta	OpenBSD 092	Core 2.6.0c	Open BSD version of **/seek** refinement.
30-Aug-2005	Beta	Linux 042	Core 2.6.0c	Linux version of **/seek** refinement.

30-Aug-2005	Beta	Win32 031	`Core 2.6.0c` `View 1.3.1c`	A Beta test version of REBOL/Core with new **`/seek`** refinement added to **open**. Allows fast random access to large files. See Seek mode summary for more comments.
11-Aug-2005	Beta-2	Win32 031	`SDK 2.6.0`	A Beta test version of the SDK with the 1.3.1 fixes and changes is now available for current SDK developers. Contact REBOL via feedback or email Cindy if you would like to try it.
9-Aug-2005	Beta	Linux 042	`View 1.3.1`	**First beta test version of View for Linux.** Requires C++ libraries to be in your lib directory (pref: stdc++ 5.0). (Please tell us if you encounter execution problems.) This release does not do installation. View will run in the dir you put it in. But, it does use the standard "~/.rebol/" area for the data cache. Note that DRAW (AGG) does not support fonts on Linux at this time due to lack of X11 support for font glyph queries.
8-Aug-2005	Alpha	OSX 024	`View 1.3.1`	**This is a crude prototype of REBOL/View running on OSX. It generally works and runs REBOL/View scripts.** The viewtop opens properly, and you can launch various reblets, even some of the games. However, there are a few things to watch out for (to be fixed): • Not yet an app bundle. So, sorry, you must **chmod +x** rebview and run it from the shell. (We will bundle it pretty soon). • **No security requestors!** That means it is like running without normal REBOL security in place. Use caution in what you decide to run. • No installation. View will run in the dir you put it in. But, it does use the standard OSX "~/Library/Application Support" area for the data cache. • No timers. So scripts that use timers (like a few of the demos) do not yet work. • Still uses CTRL keys for apps, not yet APPLE CMD key. • DRAW (AGG) does not support

				fonts. • Also missing: menus, file requestor (native), clipboard port, icons, and much more. But, despite all that, it still does ok for an early prototype. *We would like to thank Jaime Vargas for his encouragement and help in getting REBOL/View for OSX moving again!*

7-Jul-2005	Beta	OSX
		024
			`Core 2.6.0`
			`Base`
			`Pro`
			`Command`
			`2.5.125`
				Same as below.

6-Jul-2005	Beta	OpenBSD
		092
			`Core 2.6.0`
			`Base`
			`Pro`
			`Command`
			`2.5.125`
				Here are builds of the most recent betas for OpenBSD. They include all fixes below, **but** they may not include all necessary fixes for final release. We need OpenBSD users to test these and tell us how they stand. Note: The normal graphics functions that are part of Command are not included.

17-Jun-2005	Release	Win32 031	`View 1.3.1`	Various bug fixes. Mostly minor, but one major problem with Launch has been fixed. For list, search RAMBO with 1.3.1 string.

15-Jun-2005	Beta	Win32
		031
			`Core 2.6.0`
			`Base`
			`Pro`
			`Command`
			`2.5.125`
				New releases based on 1.3.0 changes. Quite a few changes and additions so we are doing one beta release before moving them to 2.6.0 version. See 1.3 changes doc (linked below) for details.

10-Jun-2005	Release	Win32 031	`View 1.3.0`	New release. See REBOL/View Changes doc for more information. Other platforms to be supported soon.
May-2005	Internal	Win32 031 Linux 042	`View 1.2.100+`	**Baseline releases for REBOL/View 1.3 project development only.** See detailed View 1.3 Change notes for more information.

23-Mar-2005	Test	Mac OS X 024	`Pro 2.5.8cb1`	Experimental build of OS X with library callbacks enabled for selected developers to try. Absolutely untested.

11-Jan-2005	Alpha	OpenBSD	`2.5.8`
			`2.5.58`
			`(as .tg)`
				Update to OpenBSD 3.6.
				
				Note that the 58 release does not include FACE or VIEW related products due to recent changes that prevented those builds (DRAW AGG compatiblity). But does include fixes for TCP port, system port, and CALL function interference and changes in termination handling changes, shell CALL function added back, security request message when CALL is used.

4-Jan-2005	Alpha	Windows	`Core`
			`2.5.58.3.1`
			`2.5.58.4.2`
			`2.5.58.9.2`
				Most changes for this release are fixes in Linux TCP port, system port, and CALL function interference. Other changes: minor termination handling changes, shell CALL function added back, security request message when CALL is used. Search RAMBO for 2.5.58 for list.

8-Nov-2004	Alpha	Linux x86	`Command` `2.5.55.4.2 a1`	Command with async core for Linux. This is a test build. Unlike the Windows version below, this one does not include graphics functions. We had to remove them because the AGG DRAW changes are not yet ready for compiling on Linux.
8-Nov-2004	Alpha	Win32	`Command` `2.5.55.3.1 a1`	Alpha build of Command with async core.
18-Oct-2004	Alpha	OSX	`Pro, Command` `2.5.8.2.4 a2`	Alpha build of OSX that should fix dynamic library problem. However, DLL's have not been tested (it's alpha).
18-Oct-2004	Alpha	Win32	`View` `1.2.55.3.1 a1`	This is a test merge of all prior changes related to /View, including async, a functioning Desktop, AGG (vector graphic) enhancements. It may not be totally compatible with prior DRAW effects but should not crash.

18-Oct-2004	Alpha	Win32	`Base, Core 2.5.55.3.1 a1`	New alpha release of async kernel version. Adds a few new PATH forms. Various other fixes. See Carl's blog for details.
15-Oct-2004	Alpha	OSX	`Base, Core 2.5.8.2.4`	Alpha release of REBOL console executables for Mac OS X. These are "very alpha". Use at your own risk.
07-Oct-2004	Alpha	Linux x86	`Core 2.5.8.4.2`	Alpha release of Linux changes for redirection and shell exit status.
07-Oct-2004	Alpha	Solaris Sparc	`Core 2.5.8.10.1`	Alpha release of Solaris changes for redirection and shell exit status.
07-Oct-2004	Alpha	OpenBSD x86	`reb* en* 2.5.8.9.2`	Alpha release of OpenBSD changes for redirection and shell exit status.
05-Oct-2004	Alpha	Win32	`Core 2.5.54.3.1 Base 2.5.54.3.1 View 1.2.54.3.1`	Fixes async kernel IO problem (rugby and read-thru should now work ok).
04-Oct-2004	Alpha	Linux x86	`Core 2.5.53.4.2 Base 2.5.53.4.2`	Linux Alpha 4 Internal Test Build (Async Kernel)
30-Jul-2004	Alpha	Win32	`Core 2.5.8.3.1 View 1.2.48.3.1 All Products`	Beta Base Build prior to Async Kernel source merge.

Earlier experimental releases can be found here.

Build Class

The build "Class" field above means:

Internal	An internal build for project developers.
Release	An official release of REBOL.
Beta	Required functionality is present, but waiting for feedback from developers regarding problems. Not supported. Use at your own risk.

Alpha	Created for testing purposes to try new ideas or check for stability of changes. Not tested. Not supported. Use at your own risk.
Test	A specific version built for developers who need a specific change or intermediate version. Not tested. Not supported. Use at your own risk.

How to Download Using REBOL

To avoid problems many browsers have with downloading a file that has no suffix, you can use this REBOL code:

```
write/binary %rebol
read/binary http://www.rebol.net/builds/092/rebol2600092
```

Put this in a file or type it as one line in the console.

Do not forget the /binary or the file will be corrupted.