REBOL

Pourquoi (et où) vous procurer Rebol ?

Rebol est un langage interprété, que l'on peut utiliser sur de nombreuses plate-formes différentes. Son interpréteur tient en quelques kilo-octets à peine, ce qui simplifie grandement son déploiement en comparaison de langages de scripts plus répandus comme Perl ou Python.

Rebol, un langage simple et moderne

Comme je l'ai dit en introduction, Rebol est un langage beaucoup plus facile à utiliser que les langages de bas niveau comme C ou C++ pour tout ce qui touche à la manipulation de données. Il implémente en effet des outils puissants (dont nous aurons l'occasion de reparler) pour cela, et, ce qui est pratique, permet de profiter d'une certaine abstraction pour les tâches habituellement fastidieuses : il peut par exemple gérer à votre place les allocations mémoire (dites adieu à `malloc` et aux pointeurs !) ou les connexions au réseau (nous verrons cela au chapitre 5).

Si vous faites une recherche sur le web, Rebol vous semblera peu connu, et la version actuelle, Rebol 2, est de plus en fin de vie. Tout ceci nuit un peu à l'image du langage, mais c'est dû au développement de la version 3 du langage, qui a pris beaucoup plus de temps que prévu. N'ayez craintes, la version 2 est mature et utilisable en production, pour des projets sérieux.

Parmi ces projets sérieux, beaucoup concernent le web. Les créateurs de Rebol prennent très au sérieux le développement croissant du web ainsi que les technologies comme AJAX, et citent sur les pages communautaires du langage le projet Qtask, un système de gestion de projets via un site web moderne, entièrement écrit en Rebol, ou encore AltME, un logiciel de messagerie décentralisé lui aussi développé en Rebol (l'équipe de Rebol est d'ailleurs accessible via ce logiciel).

Vous l'aurez donc compris, Rebol est conçu pour simplifier le développement d'applications utilisant le réseau, qu'elles soient centralisées ou non, qu'elles tournent dans un navigateur web, comme Qtask, ou non, comme AltME. Rebol intègre en effet son propre moteur d'affichage, VID, conçu pour être simple et néanmoins très souple à la fois, et qui est intégré à l'interpréteur Rebol dans un programme nommé RebView - qui tient dans moins d'un mo lui aussi (il est donc facile à déployer).

Un langage interprété

Pourquoi s'embêter à utiliser un langage interprété ?

Rebol, comme la plupart des langages de scripts, nécessite la présence sur la machine cible d'un interpréteur, qui est un programme (souvent écrit en C ou dans un langage de bas niveau) traduisant les programmes écrits dans le langage interprété? pour qu'ils soient réellement exécutables. Cela veut dire que Rebol, comme Python, Perl, PHP? sera plus lent que le C (entre autres), et qu'il nécessitera un logiciel pré-installé.

C'est a priori assez gênant. Cependant, comme je l'ai dit en introduction, l'interpréteur de Rebol peut être embarqué avec vos programmes, qui eux sont livrés sous forme de fichiers source (et non de binaires), et il pèse finalement assez peu. Vous pourrez donc le fournir avec vos programmes sans que ça soit vraiment un problème. D'autre part, pour ce qui est des performances, Rebol est effectivement à la traîne. Mais ce serait

problématique si vous comptiez utiliser ce langage pour faire un jeu vidéo - ce qui n'est pas le cas. Clairement, Rebol ne remplacera jamais les langages bas niveau là où ils sont utiles.

En revanche, le côté interprété de Rebol a permis à ses développeurs de donner au langage une certaine flexibilité qui serait difficile à décrire pour l'instant, mais dont nous profiterons bien vite. Rebol n'est pas fait pour être rapide, il est fait pour faciliter le développement en utilisant des concepts de haut niveau. Ainsi, il introduit un concept nouveau, l'utilisation de langages spécialisés pour différentes tâches. C'est comme si vous manipuliez des objets qui comprennent chacun un langage qui ressemble à Rebol lui-même, mais avec de nouveaux mots (une espèce de jargon).

Par exemple, si vous vous adressez à un robot Lego pilotable par ordinateur, vous écrivez quelque chose comme

```
insert lego [motor left power 10 start]
```

où des mots comme `motor`, `left` etc. n'ont de sens que dans ce contexte (notez le mot `lego` qui précède). Utilisez des dialectes se fait simplement en Rebol, grâce à ses outils comme la fonction `parse` - mais nous verrons tout cela plus tard . Notez que VID, l'outil d'interface graphique de Rebol, est aussi un dialecte, qui lui définit de nouveaux mots comme `button` ou `layout`.

Une archive contenant Rebol

Nous allons travailler avec la version 2.7.6 de Rebol, plus particulièrement avec une version spéciale un peu plus lourde (11 méga-octets quand même !) qui contient tous les outils pratiques pour découvrir le langage. Vous la trouverez sans problème sur cette page de téléchargement. Récupérez une archive adaptée à votre système.

Dans cette archive se trouve un répertoire "tools", qui contient normalement un exécutable nommé "rebol" : c'est lui que nous utiliserons dans un premier temps, car il nous permettra de tester en ligne de commande tout ce dont nous avons besoin.

Lancez-le (chez moi, double-cliquer dessus suffit). Vous aurez alors une ligne de commande, dans laquelle vous pourrez entrer des expressions Rebol pour avoir leur résultat. Par exemple, Rebol peut afficher

```
REBOL/Core 2.7.6.2.5 (31-Jul-2008)
Copyright 2008 REBOL Technologies
REBOL is a Trademark of REBOL Technologies
All rights reserved.

>>
```

Les >> sont le signe que l'interpréteur attend vos commandes. Nous allons lui en donner !

Maintenant que Rebol est installé, à l'abordage !

Premiers contacts avec le langage

Nous allons découvrir Rebol et sa syntaxe (assez inhabituelle pour les débutants) à l'aide de quelques exemples d'expressions dans ce langage. Il suffit de les taper en ligne de commande et nous avons le résultat .

Des calculs

La première chose que nous pouvons faire en Rebol est de vérifier si nos opérateurs mathématiques habituels fonctionnent encore :

```
>> 40 + 2
== 42
>> 545 - 2342 + 12
== -1785
>> 2 * 3
== 6
>> 2 ** 16
== 65536.0
```

À chaque fois, Rebol nous donne le résultat sur la ligne suivante, précédé par un "==". Notez l'opérateur ** qui sert à élever un nombre à une certaine puissance. Le résultat est un nombre flottant : ils se notent comme en C, à l'aide d'un point pour la virgule.

Testez l'exemple suivant : `12 + 3 * 2`. Surprise ! Rebol ne gère pas la priorité des opérateurs ! En effet, et c'est peut-être un peu dommage, Rebol ne tient pas compte de règles de priorité pour les opérateurs mathématiques (ou autres) et exécute de gauche à droite tout ce qu'il trouve. Il faudra faire avec .

Bien que Rebol soit faiblement typé (nous reviendrons sur ce terme), il possède une vague notion de types de données. De même qu'en C vous utilisiez des nombres de types int ou long pour les entiers, float ou double pour les flottants, ici les types Rebol sont appelés `Integer!` et `Decimal!`. Avec le point d'exclamation, oui.

Si vous essayez vos propres calculs, vous écrirez peut-être à un moment une opération mathématique sans espace de chaque côté des opérateurs : Rebol vous signalera une erreur ! Les opérateurs doivent être séparés des opérandes par un espace.

Des variables

Ceci ne devrait pas trop vous dépayser : vous connaissez certainement les variables, pour les avoir déjà rencontrées dans d'autres langages. Elles servent à stocker des résultats intermédiaires, et peuvent être déclarées en ligne de commande comme les autres. Regardez ces exemples pour comprendre quelle est la syntaxe :

```
>> a: 42
== 42
>> foo-bar: a
== 42
>> a + foo-bar
== 84
>> c: d: e: 14
== 14
>> a + (z: 32)
== 74
```

Il faut donc utiliser deux points pour définir une variable (et pas de caractère étrange pour en utiliser une). Des notations assez riches sont autorisées, par exemple `foo-bar` qui ne sera pas compris comme étant une soustraction (cf. fin de la partie précédente) mais comme un nom. Notez qu'une définition de variable renvoie la valeur stockée, d'où l'exemple de la dernière ligne. Suivant l'ordre des parenthèses, z est défini à 32, puis la valeur 32 est renvoyée, et l'opération est effectuée. Je ne vous conseille pas d'abuser de cette possibilité, ça ne facilite en rien la lecture du code.

Profitons-en pour découvrir quelques fonctions (ici encore, vous connaissez le concept). Commençons par la fonction print : elle affiche ce que vous lui donnez en argument, et finit par un retour à la ligne.

```
>> print "Bonjour les zéros !"
Bonjour les zéros !
```

Ici, pas de symbole "==" : ça n'est pas une valeur de retour que Rebol nous a affichée, mais bien du texte imprimé sur l'écran par la commande `print`. Il existe aussi une fonction `prin` qui se comporte comme print mais sans retour à la ligne.

Dans certains langages (comme Python avant la version 3), l'instruction `print` permet d'afficher plusieurs éléments à la suite si on les sépare par des virgules. Par exemple on pourrait avoir en Python

```
>> print "a vaut", a, "!"
a vaut 42 !
```

Pour faire ceci en Rebol, pas besoin d'utiliser une syntaxe spéciale : il suffit d'utiliser un type de données que nous aborderons plus tard, les blocs. Les blocs contiennent plusieurs éléments, séparés par des espaces, entre des crochets. Ils permettent par exemple d'écrire `print ["a vaut" a "!"]` pour avoir un résultat équivalent au précédent .

Des mots

Voici un concept important, sur lequel nous devrons pourtant nous montrer brefs pour l'instant : Rebol sait manipuler des objets d'un type étrange, les mots. Ce sont des expressions a priori anodines, et elles pourraient vous rappeler les chaînes de caractères. Grossière erreur ! Les mots sont plus que ça.

Contrairement aux blocs et aux valeurs littérales comme les chaînes de caractères ou les nombres, ils n'ont de valeur que quand vous les définissez. Par exemple, avant d'être une fonction, `print` est un mot? qui a été défini pour être la fonction `print` que nous connaissons. Un peu tordu non ?

Les mots sont un nom avant d'être une valeur. Cela veut dire qu'on peut les manipuler en tant que tels, en faisant précéder ce nom par un ' (apostrophe). Par exemple, il est correct d'écrire

```
>> variable: 'print
== print
>> print variable
== print
```

La variable variable contient le mot print. Nous aurions pu écrire `print 'print`. Ici, le mot print apparaît deux fois, mais la deuxième fois (avec l'apostrophe) il a un sens littéral, tandis que la première fois, il est évalué comme étant une fonction. C'est simple : précéder un mot d'une apostrophe empêche son évaluation. Nous verrons plus tard à quoi cela peut servir.

Il y a une autre opération que l'on peut faire sur les mots, qui empêche leur évaluation mais ne donne pas pour autant leur sens littéral. En ajoutant : avant leur nom, on permet de copier ces mots sans les évaluer, mais sans se contenter de stocker leur nom pour autant. Petit exemple :

```
>> foo: 'print
== print
>> bar: rint
>> baz: print
** Script Error: print is missing its value argument
** Near: baz: print
>> foo 3
== 3
>> bar 3
3
```

Nous tentons de définir trois variables. L'une d'elle contient le mot littéral print. L'autre copie la fonction print. La dernière? ne sera pas définie, car la syntaxe est incorrecte. En effet, quand nous écrivons `baz: print`, Rebol croit que nous souhaitons stocker dans baz le résultat qui se trouve à droite des deux points, or `print` tout seul n'est pas une opération correcte (l'erreur nous le dit : il manque des arguments).

Ensuite, nous utilisons nos deux variables en tentant de passer 3 en argument. Dans le premier cas, on pourrait croire que cela réussit, mais non : rappelez-vous de ce que je vous ai dit ! Si c'était print qui affichait un résultat à l'écran, il n'y aurait pas de "==" placé avant le 3 . Rebol se contente donc d'évaluer la dernière expression et de la renvoyer. En revanche, dans le deuxième cas, la variable bar agit comme print, et affiche son argument à l'écran.

Pour l'instant, les mots vous semblent probablement inutiles. Nous verrons cependant qu'ils permettent de définir au sein de Rebol des micro-langages adaptés en fonction du contexte, et qu'ils autorisent une grande souplesse dans la syntaxe du langage. Contrairement au C (et à la grande majorité des langages) où il existe des mots réservés, Rebol vous autorise par exemple? à redéfinir le mot `if`, dont nous verrons au chapitre suivant qu'il a pourtant le même sens qu'en C, à savoir de permettre les instructions conditionnelles. Mais l'intérêt de ceci est également encore assez flou pour vous, je le sens bien .

N'ayez pas peur de Rebol. Je vous ai présenté les mots pour vous déstabiliser un petit peu, mais maintenant que c'est fait plus rien de compliqué ne nous attend .

Les blocs

Rebol, en tant que langage dynamique, vous permet de manipuler des blocs de code qui peuvent être exécutés quand bon vous semble. Mais ces blocs sont aussi la principale structure de données, et pourront également vous servir à contenir des données diverses.

Les blocs sont des conteneurs

Construire des blocs

Nous en avons un tout petit peu parlé précédemment. Les blocs servent à contenir des objets Rebol comme des chaînes, des nombres ou autres (pourquoi pas d'autres blocs ?), tous rangés dans un certain ordre. On écrit le bloc avec des crochets, en séparant les objets par un espace. Par exemple `foo: [1 2 3 4]` stocke un bloc contenant ces 4 chiffres dans la variable foo.

Il existe plusieurs fonctions pour manipuler les blocs, dont les plus simples sont `empty?`et `length?`. La première fonction renvoie `true` quand son argument est un bloc vide, la seconde renvoie la longueur de son argument quand c'est un bloc :

```
>> empty? []
== true
>> empty? [foo]
== false
>> length? foo    ; notre variable foo: [1 2 3 4]
== 4
>> length? [a b [foo bar]]    ; Notez que le mot bar n'a pas besoin d
'être défini ; nous y reviendrons.
== 3
```

Naturellement, il nous faut aussi des fonctions pour construire des blocs (qui seront par exemple utilisées dans des boucles ou des fonctions comme nous le verrons prochainement). La fonction la plus élémentaire pour ça est la fonction `append`. Elle prend en argument un bloc et une valeur à ajouter. Par exemple :

```
>> append [1 2 3] 4
== [1 2 3 4]
```

Cependant, observez l'exemple suivant :

```
>> append [1 2 3] [unbloc]
== [1 2 3 unbloc]
```

Voici un comportement singulier. Quand nous utilisons `append` sur deux blocs, la fonction ajoute le contenu du deuxième au premier. Mais si nous voulions ajouter le bloc [unbloc] comme *élément* du premier ?

Il faut utiliser ce que l'on appelle un raffinement. Les raffinements sont des options que l'on peut passer à certaines fonctions. Ici, `append` peut recevoir le raffinement `/only`, ce qui donne

```
>> append/only [1 2 3] [unbloc]
== [1 2 3 [unbloc]]
```

Bien sûr dans les cas précédents, `append` continue d'avoir le même comportement. Nous utiliserons cette particularité lorsque nous voudrons construire des blocs de blocs.

Cela ne s'est pas vu ici, mais la fonction `append` modifie les blocs sur lesquels elle travaille. En d'autres termes, si nous écrivons `append unbloc unevaleur`, le bloc `unbloc` est modifié, `append` ne se contente pas de renvoyer le résultat.

Altérer des blocs déjà existant

Une autre fonction intéressante est `insert`. Cette fonction prend en argument un bloc et une valeur et modifie ce bloc pour insérer le résultat au début. Voici des exemples :

```
>> insert [1 2 3] 0
== [1 2 3]
>> foo: [1 2 3]
== [1 2 3]
>> insert foo 0
== [1 2 3]
>> print foo
0 1 2 3
```

Il est important que vous compreniez qu'ici, ce qui compte ça n'est pas le résultat de `insert`, qu'on pourrait à la limite ignorer. Ce qui compte, c'est la modification que cette fonction réalise sur le bloc.

Pour modifier une valeur dans un bloc, utilisez `replace bloc valeur1 valeur2`. Pour supprimer une valeur, utilisez `remove bloc` (supprime l'élément en tête). Un petit exemple vaut mieux qu'un long discours :

```
>> foo: [1 1 3 1 1]
== [1 1 3 1 1]
>> replace foo 1 2
== [2 1 3 1 1]
>> replace/all foo 1 2
== [2 2 3 2 2]
>> foo
== [2 2 3 2 2]
>> remove foo
== [2 3 2 2]
>> foo
== [2 3 2 2]
>> remove/part foo length? foo
== []
```

Notez le raffinement `replace/all` qui remplacera les valeurs dans le bloc entier (sans ce raffinement seule la première valeur qui correspond à ce que l'on cherche est remplacée). Notez également le raffinement `remove/part` qui ajoute un argument : le nombre d'éléments à supprimer. Ici, on passe en argument `length? foo` soit le nombre d'éléments total contenu dans le bloc.

Une question d'index

Les blocs sont particulièrement importants en Rebol. Si vous connaissez le langage Lisp, ils ont autant d'importance que les listes dans ce langage. Toutefois, les blocs Rebol possèdent une caractéristique importante que les listes n'ont pas : ils contiennent un index interne, qui sert à faire des opérations.

C'est une notion qui se révèlera utile, comme toutes les autres, mais qui est pleine de pièges. On peut voir cet index comme une sorte de curseur qui indique où l'on est en train de travailler, sans pour autant détruire le bloc. En quelques sortes, on "cache" une partie du bloc pour faire des opérations sur le reste. Cependant, on peut récupérer à tout moment le bloc original, en profitant des modifications qui on été réalisées.

La seule fonction que nous présenterons pour l'instant est `next`, qui permet d'avancer d'un cran le curseur dans un bloc. Il en existe d'autres comme `back` et `skip`, au sujet desquelles je vous invite à vous renseigner, en tapant `help back` dans la ligne de commande Rebol par exemple.

La fonction `next` va en fait renvoyer le bloc privé de son premier élément (sauf si le bloc est vide bien sûr), et il faut alors le stocker dans une variable (éventuellement la même) ou réaliser des opérations directement dessus. Par exemple :

```
>> foo: [1 2 3 4 5]
== [1 2 3 4 5]
>> foo: next next foo
== [3 4 5]
>> foo
== [3 4 5]
```

À ce stade on dirait que foo a été modifiée de façon irréversible? On peut stocker une nouvelle valeur dans cette "nouvelle" variable foo, par exemple :

```
>> insert foo 0
== [3 4 5]
>> foo
== [0 3 4 5]
```
Mais, à l'aide de la fonction `head`, nous pouvons alors afficher le bloc depuis le début (ce qui ne déplace pas le curseur, cela produit juste un affichage) :

```
>> head foo
== [1 2 0 3 4 5]
```
Surprise ! Ni le 1 ni le 2 n'ont été perdus, ils étaient simplement mis de côté pendant que nous travaillions plus loin dans le bloc? Nous reviendrons sur les index lorsque nous utiliserons des boucles pour modifier des blocs de données.

Des blocs de code

Outre le fait qu'ils contiennent des données, les blocs en Rebol servent surtout à contenir? du code ! Rebol permet en effet de manipuler du code, et d'utiliser certaines fonctions sur les blocs pour exécuter ce code dans un contexte particulier. Si vous n'avez utilisé pour le moment que des langages de bas niveau, cela peut vous sembler étrange et contre nature.

Pourtant, cela peut se montrer très utile. On a alors la possibilité d'écrire des fonctions (dites "d'ordre supérieur") qui manipulent ce code pour le faire s'exécuter dans des contextes particuliers. Imaginons que vous commandiez un robot : vous pourrez alors définir une fonction d'évaluation des commandes du robot qui reconnaît, en plus des mots habituels, des instructions pour que le robot avance, recule, tourne? Vous écririez par exemple

```
robot [

    avance 5

    tourne 90

    avance 10

]
```
et le sens de ces instructions serait compris par la fonction `robot`. Plus intéressant encore, puisque ce qui se trouve entre les blocs n'est pas directement évalué, vous pouvez utiliser des langages spécialisés dont vous pouvez contrôler la syntaxe.

Cependant, toute ce côté "conception de micro-langages" (on les appelle "dialectes") ne nous intéressera que plus tard. Pour l'instant, étudions un peu comment Rebol stocke les blocs de code, et quelles opérations nous pouvons effectuer dessus.

Évalutation retardée

Le gros avantage de ces blocs est, comme je l'ai dit, d'attendre sagement qu'on les évalue. Nous pouvons stocker dans un bloc un code qui afficherait du texte à l'écran, en faisant `a: [print "Bonjour les Zéros !"]`, mais ce code n'est pas évalué (rien n'est pour l'instant affiché à l'écran), puisque pour cela il faut une fonction adaptée.

La fonction la plus simple est `do`, qui exécute le bloc passé en argument et renvoie la dernière expression évaluée. Par exemple

```
>> do [ print "Salut salut !"
[    4 + 3 ]
Salut salut !
== 7
```

Ceci est parfaitement autorisé : nous écrivons deux instructions, une sur chaque ligne, et Rebol exécute le print puis calcule 4 + 3, qui est la dernière expression du bloc (le retour est donc 7). L'interpréteur affiche un [au début de la seconde ligne pour nous rappeler que nous sommes dans un bloc au moment où nous écrivons .

Citons également la fonction `reduce`, qui, à partir d'un bloc, évalue toutes les expressions qu'il contient puis renvoie le bloc formé des résultats :

```
>> reduce [ 3 + 4 5 * 2 17 / 3]
== [7 10 5.66666666666667]
```

Nous nous servirons de cette fonction dans deux chapitres à peine, quand nous traiterons des listes de fichiers . Notez que ce que nous avons appris dans la partie précédente tient toujours : la fonction `replace` par exemple peut très bien être utilisée sur du code, comme dans

```
do replace [print "foo"] 'print 'prin
foo>>
```

J'ai fait exprès de compliquer l'expression, pour vous forcer à réfléchir un peu . Comme toujours, elle se lit plus ou moins de gauche à droite. La première fonction exécutée est en fait `replace`, qui remplace le mot print par prin (rappelez-vous, cette fonction affiche le texte à l'écran mais ne revient pas à la ligne). Ensuite `do` exécute le bloc alors renvoyé (qui est `[prin "foo"]`.

Ça n'est pas si compliqué, c'est une expression typiquement rebollienne .

Si certains concepts vous échappent encore, ça ne fait rien. Nous avons tout le temps de nous familiariser avec eux. Nous allons commencer à faire de vrais programmes dès le chapitre suivant !

Structures de contrôle et fonctions

Nous allons retrouver les blocs, mais sous une forme que vous connaissez mieux, car les structures de contrôle en Rebol ressemblent à toutes celles de la plupart des langages impératifs .

Vous n'êtes pas obligés de tout retenir ; sachez juste revenir à ce chapitre si vous trouvez un jour une structure de contrôle qui vous est inconnue, vous l'avez peut-être juste loupée ici.

Test conditionnel

Il y a quelques structures de contrôle qui permettent d'exécuter du code sous certaines conditions. L'instruction if du C est plus ou moins présente, et elle porte le même nom en Rebol, mais elle s'en distingue quand même par quelques petites subtilités.

Les valeurs booléennes

Si nous voulons faire des tests conditionnels (Si? alors? sinon?) en Rebol, il nous faut déjà savoir ce qui vaut vrai, et ce qui ne le vaut pas. En fait, en Rebol, tout ce qui est différent de `false` et `none` est considéré comme vrai. Même l'entier 0, qui équivaut pourtant à faux en C?

On dispose des opérateurs de comparaison classiques, comme = pour l'égalité, <> pour la différence, et bien sûr >, <, >= et <=. Toutefois, nous utiliserons parfois =? (correspondant à la fonction same?) pour tester l'égalité des valeurs, pour des raisons que nous préciserons.

De même, on dispose de 4 opérateurs pour combiner des valeurs booléennes, qui sont NOT pour la négation, AND pour la conjonction, OR pour la disjonction et XOR pour la disjonction exclusive.

Enfin, citons deux fonctions qui agissent sur des blocs, `any` et `all`, qui testent toutes les valeurs du bloc passé en argument. Alors `all` équivaut à vrai si elles équivalent toutes à vrai, et `any` équivaut à vrai si au moins une équivaut à vrai. Par exemple `any [false none 2]` équivaut à `true`.

Le mot if

Il s'emploie simplement, comme en C, suivant la syntaxe `if <condition> <bloc>` et renvoie le résultat du bloc si ce dernier s'exécute, sinon `none`. Considérons la fonction `ask` qui pose une question à l'utilisateur et renvoie sa réponse, et le code suivant :

```
>> reponse: ask "Est-ce votre anniversaire ?"
Est-ce votre anniversaire ? Oui
== "Oui"
>> if reponse = "Oui" [print "Joyeux anniversaire !"]
Joyeux anniversaire !
```

Rien de bien sorcier donc.

Nous pouvons spécifier ce que Rebol doit faire dans le cas où la condition n'est pas remplie : il suffit de rajouter le raffinement `/else` ! Par exemple

```
rep: ask "Est-ce votre anniversaire ? "

if/else rep = "Oui" [

        print "Incroyable, c'est votre anniversaire !" ][

        print "Un autre jour peut-être ?"

]
```

Nous verrons prochainement comment faire de ce code un script stocké dans un fichier à part, gardez patience . Mais déjà, nos programmes deviennent plus concrets !

Le mot either

Le mot `either` fait exactement la même chose que `if/else`, et le second bloc est donc obligatoire .

Boucles

On trouve naturellement quelques boucles en Rebol. Certaines servent à répéter une instruction sous certaines conditions, d'autres à parcourir un bloc?

Boucles conditionnelles

La première (et la plus classique) est la fameuse boucle `while`, qui s'exécute tant qu'une condition est vraie. Elle prend deux blocs en argument, le premier étant la condition à tester, et le deuxième le code à exécuter. Par exemple

```
>> i: 0
== 0
>> while [i < 10] [i: i + 1
[    print i ]
1
2
3
4
5
6
7
8
9
10
```

Sans grandes surprises donc

Un peu plus particulière, la boucle `until` qui exécute le bloc passé en argument tant que la dernière expression est équivalente à `true`. Nous l'utiliserons plus rarement, dans des cas où son utilisation fait gagner du temps.

Il arrive que l'on utilise l'instruction `break` qui, comme en C, interrompt le calcul d'une boucle. Cette instruction se révèle notamment utile dans une boucle qui ne s'arrête jamais, `forever` (exécute inlassablement son bloc de code).

Itérations

Comme dans beaucoup de langages, on trouve en Rebol une boucle `for`, qui a le même comportement que d'habitude. Elle prend en argument un mot (la variable "compteur"), les bornes de départ et d'arrivée, et le pas d'itération. L'exemple précédent se réécrirait `for i 1 10 1 [print i]`.

Mais il y a d'autres boucles accessibles. La boucle `loop` exécute un bloc un nombre fixé de fois (passé en argument).

La boucle `foreach` sert à parcourir un bloc (ou une autre structure de données) élément après élément. Nous l'utiliserons par exemple pour appliquer un traitement à tous les fichiers d'un même répertoire.

La boucle `repeat` est équivalente à `for` avec un départ fixé à 1 et un pas fixé de 1.

N'hésitez pas à vous entraîner un petit peu à utiliser ces boucles !

Fonctions

Tant que nous y sommes, parlons de la définission de nos propres fonctions. Comme les boucles, elles sont déclarées à l'aide d'un mot clef et de plusieurs blocs : un pour lister les arguments, et un pour donner le code de la fonction. Ce mot clef est le mot `func`. Le second bloc peut renvoyer un résultat, qui sera, comme dans le cas du mot `do`, le résultat de la dernière expression évaluée.

On peut alors bien sûr stocker la fonction générée dans une variable qui formera un nouveau mot utilisable comme les autres : en utilisant `append` sur des chaînes de caractère, on peut ainsi faire une fonction comme

```
>> bonjour: func [ nom ] [ append "Bonjour " nom ]

>> bonjour "Zozor"

== "Bonjour Zozor"
```

Comme vous pouvez ici le voir, nous utilisons la fonction `append` sur deux chaînes. En fait, beaucoup de fonctions qui marchent sur les blocs peuvent être utilisées sur des chaînes, je vous laissez faire vos propres expériences.
Nous reviendrons sur les fonctions petit à petit. Nous avons certes appris plein de nouvelles structures, mais nos connaissances sont encore assez faibles : nous sommes pour l'instant incapables d'utiliser Rebol pour de vrais problèmes. Nous allons corriger cette lacune .

Enregistrer nos programmes

Comme nous allons maintenant faire de vrais programmes de plusieurs lignes, il serait plus agréable de les enregistrer dans des fichiers réutilisables. Si ça n'est pas déjà fait, enregistrez l'exécutable rebol dans un répertoire où il pourra être appelé facilement en ligne de commande (par exemple /usr/bin ou éventuelleemnt mieux sous un Unixoïde). Sous Microsoft Windows, il faudrait associer l'extension .r aux codes Rebol, ce que je ne sais pas faire, mais je suis ouvert à toute proposition d'aide .

On peut alors écrire la source quasi-directement. En fait, au début du fichier doit se trouver un bloc particulier, de la forme

```
REBOL [?informations?]
```

Voici un petit exemple :

```
REBOL [

  title: "Hello world"

  author: "Linkoln"

  date: 27-Jun-2009
```

```
  version: 1.0.0

]



nom: ask "Entrez votre nom : "

print ["Bonjour" nom]
```

Si vous ne voulez pas entrer ces informations, laissez un bloc vide. Enregistrez-le sous le nom "hello.r" par exemple. Pour exécuter le programme, tapez alors "rebol hello.r" en ligne de commande en vous plaçant dans le répertoire où vous avez sauvegardé le programme .

Assez perdu de temps avec les bases du langage ! Apprenons maintenant à scripter comme des grands .

Des types de données bien pratiques

Outre les types de données que j'ai déjà plus ou moins abordés, Rebol nous propose nativement (c'est à dire sans bibliothèque extérieure) quelques types très pratiques pour nos scripts, comme par exemple le support des URL, des fichiers et des dates. Je ne sais pas vous, mais moi j'en ai assez de la théorie, et ces types de données nous permettront de passer au concret .

Les fichiers

Chemin vers un fichier

Rebol simplifie la gestion des fichiers en la rendant uniforme d'un système à l'autre. En effet, que vous utilisiez Microsoft Windows ou un Unixoïde, les chemins seront notés à l'aide de slashs (avec dans le premier cas un éventuel identifiant de disque au début). En Rebol, on place juste un % au début de ce chemin. Par exemple `%/home/linkoln/fichier.txt`, comme sous Unix, ou `%/c/program%20files/` sont des chemins valides. Notez que les espaces sont remplacés par les "%20" que l'on trouve dans les URL .

Bon, pour les gens que ça rebute, il est également possible d'écrire %"/c/program files/", je l'avoue. Le tout est de préciser à Rebol que le nom doit être lu en entier, sans quoi il s'arrête au premier espace. Naturellement, il n'est pas nécessaire de donner le chemin entier du fichier, un simple %fichier.txt suffit par exemple à désigner le fichier du même nom contenu dans le répertoire courant. Les fichiers spéciaux %. et %.. désignant le répertoire courant et le répertoire parent sont également accessibles.

Lire un fichier

On peut lire très facilement le contenu d'un fichier à l'aide de la commande `read`. Par exemple, `text: read %fichier.txt` crée (ou met à jour) une variable `text` qui contient sous forme de chaîne ce qu'il y avait dans fichier.txt. Cependant, toujours pour des raisons de portabilité, lors d'une telle opération les retours à la ligne sont transformés en le caractère Rebol nommé `newline` et comptent pour un seul

caractère, même sous Windows. Cela permet de ne pas se préoccuper du retour à la ligne, particulier à chaque système.

Cette conversion est pratique pour des fichiers texte, mais pas pour des images, des sons ou autre. Il est donc possible d'utiliser le raffinement `/binary` pour lire à la place une chaîne binaire sans conversion. Nous reviendrons sur les chaînes binaires, mais pour le moment contentons-nous de les utiliser quand le stockage d'une image ou d'un autre document élaboré est nécessaire. Enfin, le raffinement `/lines` renvoie un bloc contenant toutes les lignes du fichiers sous forme de chaînes.

Il est possible de "lire" un répertoire, ce qui renvoie un bloc contenant les fichiers de ce répertoire. Essayez par exemple `read %.`. Attention cependant, les autres répertoires que . et .. ont leur nom qui se termine par un / : ainsi, `read %Images` ne produira pas le même résultat que `read %Images/`

À titre d'exercice, essayons d'écrire une fonction Rebol qui liste tous les fichiers texte d'un répertoire. Pour cela, utilisons la fonction `suffix?` qui renvoie l'extension d'un fichier. Nous allons construire un bloc contenant tous les fichiers concernés. Pour cela, il suffit de le déclarer initialement vide, puis d'utiliser la fonction `append` dès que nous trouvons un fichier texte.

Voici le code qui construit r à partir du répertoire courant :

```
r: copy []

foreach file read %. [

    if (suffix? file) = %.txt [append r file]

]
```

Nous utilisons à la première ligne une fonction qui vous est encore inconnue, `copy`. Celle-ci sera importante lorsque nous définirons la fonction proprement dite, afin que la variable r soit nouvelle à chaque appel? Nous reviendrons sur l'importance de `copy`, retenez pour l'instant juste que cette fonction est à utiliser quand on veut des valeurs locales modifiables, comme les blocs.

Notez que la fonction `suffix?` renvoie ordinairement une chaîne quand elle est utilisée sur des chaînes, mais que sur des fichiers, elle renvoie? un fichier . Même s'il n'existe pas de fichier nommé ".txt" chez vous (chez moi non plus), la comparaison doit se faire entre deux noms de fichiers, et pas entre un nom de fichier et une chaîne. Le code est très lisible : si le nom du fichier se termine par .txt, on l'ajoute à r, et ce pour tous les fichiers contenus dans le répertoire courant.

Je vous laisse maintenant écrire une fonction qui attend en argument le répertoire à analyser. N'oubliez pas de renvoyer un résultat à la fin?

Voici la fonction

```
f: func [d] [
```

```
    r: copy []

    foreach file read d [

        if (suffix? file) = %.txt [append r file]

    ]

    r

]
```

Écrire dans un fichier

Pour écrire à l'intérieur d'un fichier, on utilise la fonction `write` qui prend en argument le fichier et une chaîne à y écrire. Comme vous le savez peut-être déjà, écrire dans un fichier provoque par défaut sa réinitialisation, i.e. la suppression des données qu'il contenait précédemment. Pour pallier cela, on peut utiliser le raffinement `write/append`.

Par exemple pour cloner le contenu d'un fichier :

```
txt: read %monfichier.txt

write %monfichier.txt txt

write/append %monfichier.txt txt
```

De même que précédemment, les éventuels caractères `newline` seront convertis en le ou les caractère(s) utilisé(s) pour cela sur votre machine. On peut ici encore utiliser `write/binary` pour écrire une chaîne binaire plutôt qu'une chaîne de caractères. Et ici encore on peut utiliser `write/lines` pour écrire un bloc de chaînes en écrivant une chaîne par ligne dans le fichier.

Il existe de nombreuses opérations possibles sur les fichiers et les répertoires (suppression, récupération de leur taille ou d'autres informations?) que je listerai plus tard. Pour l'instant les opérations de lecture et d'écriture sont suffisantes.

Les URL

Comme je l'ai dit, Rebol supporte nativement les URL, ce qui permet de définir des interfaces assez simples à utiliser vers différents protocoles. Ainsi, lorsque nous utiliserons une extension permettant de gérer les bases de données MySQL, nous passerons par une bibliothèque écrite en Rebol qui définit des URL de type mysql:// .

Mais en attendant, apprenons déjà à nous servir de quelques protocoles. Rebol en supporte un certain nombre, dont certains vous seront probablement inconnus (saviez-vous qu'il existe un protocole chargé de vous donner la date ? connaissez-vous NNTP ?). Aussi nous nous limiterons pour l'instant aux plus simples ou aux plus courants.

DNS

Le protocole DNS fait correspondre une adresse IP à un nom, ce qui permet de retenir des adresses comme "www.siteduzero.com" plutôt que "80.248.219.123" lorsque vous voulez vous rendre sur ce site. En Rebol, ce protocole est très simple à utiliser : il suffit de se servir de la fonction `read`.

```
>> read dns://www.siteduzero.com
== 80.248.219.123
```

La valeur de retour n'est pas une chaîne, mais ce qu'on appelle un tuple de 4 éléments : c'est juste un regroupement de 4 valeurs, séparées par des points. Ça n'est pas un bloc dans la mesure où ça ne peut pas être modifié.

Il est également parfois possible de réaliser l'opération inverse :

```
>> read dns://80.248.219.123
== "lisa.siteduzero.com" ; ceci est bien une chaîne, pas un tuple !
```

Toutefois l'opération qui associe un nom à une adresse IP n'est pas une fonction dans la mesure où il peut y avoir plusieurs résultats (ici on trouve un nom différent de www.siteduzero.com). De plus, il existe beaucoup d'adresses IP qui n'ont pas de nom. Vous pouvez récupérez votre nom local en utilisant l'URL `dns://` tout court .

HTTP

Moins surprenant, mais très efficace, le protocole HTTP définit en Rebol vous permet d'écrire simplement des opérations qui seraient assez complexes à faire à la main. Tout d'abord, ne nous étonnons pas de pouvoir afficher le code source d'une page en écrivant `print read http://www.siteduzero.com/` par exemple. Il est alors possible de faire `write %index.html read http://www.siteduzero.com/`, ce qui ne récupérera bien sûr ni les images ni les feuilles de style, juste le code.

Nous verrons prochainement comment analyser le contenu d'une telle page. Il est également possible de vous faire passer pour un navigateur connu, en faisant précéder vos requêtes de la commande `system/schemes/http/user-agent:"Mozilla/4.0"`par exemple. Par défaut, une version du REBOL Core sera envoyée (chez moi, "REBOL Core 2.7.6.2.5").

Il est possible d'envoyer des requêtes aux formulaires à l'aide du raffinement `read/custom` qui accepte un bloc supplémentaire de données à envoyer sous la forme de données CGI. Par exemple, http://hroch486.icpf.cas.cz/formpost.html est un formulaire que nous pouvons remplir. Si nous regardons sa source, l'adresse du script traitant est http://hroch486.icpf.cas.cz/cgi-bin/echo.pl . Nous pouvons alors regarder les options qu'attend ce script et lui envoyer par exemple que nous nous appelons Zozor, et que nous aimons les bananes.

```
page: read/custom http://hroch486.icpf.cas.cz/cgi-bin/echo.pl [

    post "your_name=Zozor&fruit=banana"
```

```
]
```

Nous stockons le résultat dans la variable page pour pouvoir l'afficher après en écrivant `print page`. C'est aussi simple que ça !

FTP

Les URL FTP ont la forme ftp://utilisateur:motdepasse@hote/repertoire/fichier . Vous l'aurez compris, ces URL là seront accessibles en lecture et en écriture.

Tout d'abord, si vous avez besoin d'utiliser FTP en mode passif, utilisez `system/schemes/ftp/passive: true`.

Ensuite, utilisez les mêmes commandes que pour les fichiers : vous pouvez lire un fichier avec `read` et ses raffinements, et en écrire avec `write` et ses raffinements. Par exemple, cet exemple à peine modifié provient de la documentation en ligne de Rebol : supposons que nous voulions envoyer dans un répertoire pages sur notre site toutes les pages du répertoire courant. On écrirait

```
site: ftp://wwwuser:secret@www.site.dom/pages

files: read %.

foreach file files [write site/:file read file]
```

On retrouve ici plusieurs concepts que nous avons déjà appris. Les deux premières lignes sont compréhensibles grâce à ce chapitre, mais dans la dernière, outre la boucle `foreach` que vous connaissez déjà, on retrouve les deux points précédant un mot, qui servent à récupérer son contenu. Si nous n'avions pas mis les deux points, toutes les pages auraient été enregistrées sous le même nom, à savoir "file". Ici, nous copions le nom de chacune de nos pages dans le répertoire pages, ce qui nous permet de mettre notre site très rapidement à jour.

Bien sûr, nous pourrions améliorer ce script pour qu'il ne mette à jour que les fichiers modifiés depuis une certaine date (et enregistrer après la mise en ligne la date de façon à s'y référer la prochaine fois). Pour cela, nous pouvons utiliser la commande `modified?`qui prend en argument un fichier et renvoie la date de dernière modification, et la commande now qui renvoie la date actuelle.

Mais pour cela il faut que nous apprenions à nous servir des dates .

Les dates

Le dernier objet de "haut niveau" dont nous apprendrons à nous servir pour l'instant sera la date. Les dates en Rebol ont le format JJ-MMM-AAAA/hh:mm:ss, ou éventuellement l'une de ces deux parties (date et heure) seule.

Elles peuvent être utilisées avec quelques opérations que vous connaissez déjà. On peut par exemple leur ajouter une heure (sous la forme hh:mm:ss) ou un nombre de jours (représenté par un entier), ou leur soustraire une autre date (ce qui donne le

nombre de jours séparant les deux dates) ou encore un nombre de jours. Vous pouvez également les comparer en utilisant les opérateurs habituels.

Les mois à utiliser sont au format anglo-saxon de trois lettres : Jan, Feb, Mar, Apr, May, Jun, Jul, Sep, Oct, Nob et Dec. Vous pouvez récupérer la date actuelle à tout moment en faisant `now`. Cette fonction accepte plusieurs raffinements selon la partie de la date qui vous intéresse, et je vous invite à consulter l'aide en tapant `help now`.

Vous voilà maintenant parés à faire de vrais utilitaires. Amusez-vous bien !