TABLE DES MATIERES

[Introduction 2](#_gjdgxs)

[1.](#_30j0zll) Chemins et manipulations autour du système de fichiers 2

[1.1.](#_1fob9te) Généralités 2

[1.2.](#_3znysh7) Commandes utiles pour naviguer dans le système de fichier 2

[2.](#_tyjcwt) Variables 3

[2.1.](#_3dy6vkm) Généralités 3

[2.1.1.](#_1t3h5sf) Cas des nombres 3

[2.1.2.](#_4d34og8) Cas des « séries de valeurs » 3

[2.1.3.](#_2s8eyo1) Autres opérations sur les chaînes de caractères 4

[2.2.](#_17dp8vu) Expansion des variables 4

[2.3.](#_3rdcrjn) Tableaux - structures de données plus complexes 5

[2.4.](#_26in1rg) Scoping 6

[2.5.](#_lnxbz9) Assignation de variables directement par l’utilisateur (*user input*) 6

[3.](#_35nkun2) Commandes 6

[3.1.](#_1ksv4uv) Commutateurs (switches) 7

[3.1.1.](#_44sinio) Extensions des commandes 7

[3.2.](#_2jxsxqh) Syntaxe & interprétation 7

[3.2.1.](#_z337ya) Caractères génériques 7

[3.3.](#_3j2qqm3) Code sortie 8

[3.4.](#_1y810tw) Sorties standards et redirections 8

[3.5.](#_4i7ojhp) Syntaxe pour la combinaison de plusieurs commandes 9

[3.5.1.](#_2xcytpi) Rediriger la sortie d’une commande vers l’entrée d’une autre : le *pipeline* 9

[3.5.2.](#_1ci93xb) Conditionner l’exécution d’un groupe de commandes au succès d’un autre : le *compound* 9

[3.5.3.](#_3whwml4) Récupérer la sortie d’un groupe de commandes : *parenthesized compound* 9

[3.6.](#_2bn6wsx) Appeler un script comme une commande 9

[4.](#_3o7alnk) Structures de contrôle 10

[4.1.](#_23ckvvd) IF 10

[4.2.](#_ihv636) FOR 10

[5.](#_qsh70q) Fonctions 11

[5.1.](#_3as4poj) Arguments 11

[5.1.1.](#_1pxezwc) Reference variables 12

[5.2.](#_49x2ik5) Contexte local d’une fonction et valeur retour 13

[5.3.](#_2p2csry) Template de fonction DOS 13

[5.4.](#_147n2zr) De la fonction au script 13

Fusionner avec Batch.docx Normalement ça parle de ce qu’est un script. On en a pas encore parler donc intégrer ça dans un paragraphe type qu’est ce qu’un script (au début des fonctions ?)

# Introduction

MS-DOS (Disk Operating System) est un système d’exploitation mono-tâche de Microsoft qui n’est aujourd’hui plus maintenu. Les différentes versions de Windows offrent une interface en ligne de commande[[1]](#footnote-1) donnant accès au DOS. Les commandes permettent la manipulation de fichiers et de répertoires, l'exécution d'applications, et l'exécution de scripts. Attention, les commandes (souvent regroupées sous le nom de Batch) sont très loin de constituer un langage complet. Les possibilités en batch sont assez limitées même par rapport à la plupart des shells. Pour scripter avec Windows, préférer Powershell sinon un shell classique type Bash offrant largement plus de fonctionnalités. **Ne recourir à Batch que pour des opérations simples autour du système de fichiers.**

Il convient de bien distinguer le fait de passer directement des commandes à la console de la rédaction d’un script batch. Un script n’est qu’un simple fichier texte ASCII (attention à l’encodage) où est consignée une séquence de commandes telles qu’on les aurait tapées dans la console. L’extension d’un script batch est .cmd ou .bat sur d’anciennes versions de Windows. La principale précaution à prendre quant à leur nommage est de ne pas leur donner le même nom que des *built-in commands*, *system binaries* ou de programmes fréquemment utilisés afin d’éviter tout collision dans le *search path*.

# Chemins et manipulations autour du système de fichiers

## Généralités

En DOS et contrairement au monde UNIX, les différentes parties d’un chemin sont séparées par un **backslash «\».** Le nom du lecteur est toujours suivi de «:». DOS est insensible à la casse mais on choisit dans cette note d’écrire les commandes en lettres majuscules.

Les chemins peuvent être donnés de façon **absolue**, le chemin désigne alors un unique fichier quel que soit le contexte, soit de manière **relative** par rapport au répertoire courant (*current directory*), le fichier désigné dépend alors du contexte courant.

**Le *current directory* est aussi nommé «.»** et peut être accédé via la variable dynamique %CD% (dynamique car par construction sa valeur dépend du contexte et doit être recalculée à chaque fois qu’on veut y accéder). Le ***parent directory*** du *current directory* est désigné par **«..»**.

Exemple : CD ..\Some\_other\_folder

L’espace étant utilisé comme séparateur entre les arguments d’une commande, celui devrait être en principe échappé si jamais il se trouvait dans le nom d’un fichier/répertoire dont on passe le chemin à une commande. Ce n’est pas de cas comme pour les autres caractères spéciaux utiles à la syntaxe des commandes et échappés avec **«^»**. **La présence d’espaces dans le chemin demande que celui-ci soit passé entre *double quotes*.**

## Commandes utiles pour naviguer dans le système de fichier

**CD** ou **CHDIR** (*Change Directory*) : Change de répertoire courant. Remarque : CD \ remonte à la racine lecteur du répertoire courant. **PUSHD** peut aussi être utilisé pour cette tâche (en association avec **POPD**).

**COPY** : Permet de copier un ou plusieurs fichiers vers un autre répertoire.

**MOVE** : Permet de déplacer («Couper») un ou plusieurs fichiers et de renommer des répertoires.

**DELETE** ou **ERASE** : Supprime un ou plusieurs fichiers.

**DIR** : (*DIRectory listing*) Liste le contenu du current directory (commande ls sous UNIX).

**MD** ou **MKDIR** (*Make Directory*) : Crée un nouveau répertoire.

**RD** ou **RMDIR** (*Remove Directory*) : Supprime un répertoire (devant être vide par défaut).

**REN** (*REName*) : Renomme des fichiers ou des répertoires.

**TYPE** : Permet d’afficher le contenu d’un fichier. Peut être utilisé de pair avec **EDIT** qui permet d’éditer un fichier texte dans une console DOS (qui peut donner lieu à des problèmes d’encodages, les fichiers cmd par exemple étant encodés en ASCII mais l’affichage de la console peut utiliser un autre encodage comme en France).

**TREE** : Affiche une représentation graphique de la structure de fichiers.

**CLS** (*CLear Screen*) : Supprime les commandes tapées antérieurement.

Pour les commandes ci-dessus pouvant prendre en charge plusieurs fichiers à la fois, on utilise les **caractères génériques** (*wildcards*) \* et ? pour leur passer ces groupes de fichiers.

Exemple : RM \*.txt

Remarque : Pour commenter un script, précéder le commentaire de la commande REM ou « :: ». La première forme est toujours valable, la seconde est parfois source d’erreurs (notamment dans les boucles FOR).

# Variables

## Généralités

En batch, il n’y a pas de types de variables. **Toutes les variables sont des chaînes de caractères.** Ces dernières n’ont ainsi pas besoin d’être signalées en étant entourées de *single/double quotes* comme dans de nombreux langages sauf dans certaines conditions (typiquement lorsqu’une chaîne avec caractères spéciaux est passée en argument d’une commande). Sauf lors de son assignation, une variable est signalée par le caractère %.

**Les variables sont définies via la commande SET et on peut accéder à leur valeur via ECHO %my\_var%.** L’appel de SET sans arguments retourne l’ensemble des variables actuellement définies dans l’environnement (sauf les variables dynamiques).

Pour supprimer une variable il faut l’assigner à la chaîne de caractère vide : SET var=

Inversement, la même commande permet aussi de déclarer une variable sans l’initialiser.

Remarque sur ECHO : Lors de l’exécution d’un script ses commandes sont exécutées une à une comme si elles étaient tapées dans l’invite. Pour éviter d’alourdir la console en laissant la commande s’afficher à chaque fois, on peut précéder son nom de « @ » (ex : @SET var =1). Pour éviter d’avoir à faire ça pour chaque commande, on débute souvent les scripts par @ECHO OFF (affichage qu’on rétablit juste avant de sortir du script avec ECHO ON). « @ » est particulièrement utile pour les boucles FOR tapées directement dans la console : FOR %i IN (ensemble) DO @command.

### Cas des nombres

Batch n’évalue les expressions arithmétiques que dans le cadre de l’assignation d’une valeur à une variable. Il faut indiquer à DOS avec le switch /A de SET que l’expression passée à la commande SET n’est pas une chaîne de caractères à prendre telle quel mais une expression arithmétique à évaluer.

Batch fournit aussi des opérateurs d’assignation type +=, -=, \*=, /= et %=.

Exemple :

SET /A a=3

SET /A a+=7

ECHO %a% ; retourne 10

### Cas des « séries de valeurs »

DOS semble fournir des structures de données itérables ressemblant à des vecteurs ou des listes. Ce n’est pas le cas. Les itérables sont en fait des chaînes de caractères comportant des caractères qui permettront à des commandes comme FOR de les fragmenter pour réaliser une opération sur chacun des éléments issus du découpage. Les caractères séparateurs autorisés étant multiples (,;:= <espace> ou <tab>), ils peuvent même être mélangés.

Exemple : SET var=1,2,3

Si passé à FOR, la commande va couper au niveau de chaque virgule. Comme l’espace ou encore le point-virgule sont aussi des séparateurs, les séquences 1 2 3 ou 1;2;3 ou 1; 2; 3 marchent aussi. Tout comme 1; 2, 3.

Ces séries de valeurs ne sont donc pas des objets indexées. On ne peut pas faire var[1] pour obtenir la première valeur par exemple. Conséquence du fait que les séries de valeurs ne sont en fait que des chaînes de caractères comportant des séparateurs, retirer un élément de la «liste» demande en fait d’utiliser les techniques permettant de remplacer un élément d’une chaîne par un autre.

Exemple :

SET var=abc

SET var=%var:b=d%

ECHO %var% renvoie adc

**Pour supprimer un élément d’une « série », il faut le remplacer par la chaîne vide.**

SET var=1,2,3

SET var=%var:2,=%

ECHO %var% renvoie 1,3

Attention: il ne faut pas oublier la virgule dans %var:2,=% sinon la nouvelle valeur serait 1,,3.

Cette technique est notamment utile pour retirer un chemin de PATH une fois qu’on en a plus usage.

**De même, pour ajouter un élément à une série de valeurs, il faut en fait recourir à la concaténation de chaînes de caractères.** Il n’existe pas d’opérateur pour cela, on exploite en fait l’expansion des variables faite avant évaluation.

Exemple :

SET var=1,2,3

SET var=%var%,4 (%var% est remplacée par sa valeur avant évaluation)

ECHO %var% renvoie 1,2,3,4

Cette technique est notamment utile pour ajouter un chemin à PATH (SET PATH=%PATH%;%CD%).

### Autres opérations sur les chaînes de caractères

On peut ne renvoyer qu’une sous-chaîne d’une chaîne de caractères stockés dans une variable. La syntaxe est la suivante : %var:~x,y% où x est l’index du caractère de départ (commence à 0) et y la longueur de la chaîne à retourner. Si y n’est pas précisé, toute la chaîne sera retournée à partir du xe caractère. Si l’une des valeurs passées est négative, elle est ajoutée à la longueur de la chaîne puis interprétée comme dans le cas normal où les paramètres sont positifs. Ainsi :

%var:~-2% va renvoyer les deux derniers caractères.

%var:~2,-1% va renvoyer tous les caractères sauf le dernier et les deux premiers.

### Création & édition de variables d’environnement

Pour créer ou modifier la valeur de variables d’environnement (et pour que cette valeur persiste une fois la session terminée), il faut utiliser la commande SETX. Par la défaut le scope de la variable sera l’environnement utilisateur (HKEY\_LOCAL\_USER), pour la spécifier dans l’environnement système (HKEY\_CURRENT\_USER) il faut ajouter le commutateur /M. SETX se distingue aussi de SET dans le sens où on utilise pas de signe = pour l’assignation.

Exemple :

SETX PATH «%PATH:C:\Program Files (x86)\Common Files\Oracle\Java\javapath;=%» \M

SETX PATH «C:\Program Files (x86)\Java\jdk1.8.0\_171\bin;%PATH%» \M

SETX JAVA\_HOME «C:\Program Files (x86)\Java\jdk1.8.0\_171\bin» \M

## Expansion des variables

Lorsque DOS interprète une commande, il substitue à chaque variable présente avant l’évaluation de la commande (signalée par %) sa valeur dans l’environnement courant. Ainsi, si la variable var est assignée à la valeur Hello et qu’on tape ECHO %var% dans l’invite, ce n’est pas ECHO %var% qui est finalement évaluée mais ECHO Hello. Cela peut être très embêtant comme par exemple dans les boucles FOR.

Exemple :

SET var=0

FOR %%i in (1,1,5) do SET var=%var%,%%i

Au lieu d’avoir la valeur 0,1,2,3,4,5 à la fin de la boucle, var aura la valeur 0,5 à cause de l’expansion des variables. En effet, la commande réellement exécutée par DOS après expansion est FOR %%i in (1,1,5) do SET var=0,%%i

Pour pallier à ce problème il faut activer l’**expansion retardée des variables**, désactivée par défaut. Deux possibilités :

* Ouvrir une nouvelle instance de l’interpréteur avec la commande : CMD /V:ON
* Exécuter ses commandes dans un environnement local ouvert avec SETLOCAL ENABLEDELAYEDEXPANSION. SETLOCAL a priorité sur les options de CMD.

Les variable qu’on ne souhaite pas voir étendue avant l’évaluation doivent être signalées par un «!» (ne fonctionnera que si l’expansion retardée a été préalablement activée).

Dans un environnement avec expansion retardée, var aura bien la valeur 0,1,2,3,4,5 après exécution de la commande : FOR %%i in (1,1,5) do SET var=!var!,%%i

L’expansion des variables est également valable pour les références aux arguments d’une fonction/d’un script (de la forme %1 ou %\* qui est replacé par l’ensemble des arguments de la commande sauf %0), les *reference variables* qu’on peut construire sur ces derniers comme par exemple %~p1 et comme nous l’avons vu, pour les variables muettes des commandes FOR.

## Tableaux - structures de données plus complexes

Batch ne propose pas de structure de données autre que des chaînes de caractères simples assignées à une variable. Néanmoins, il existe des façons d’émuler le comportement d’un tableau en Batch dont une méthode est présentée ci-après. Cela permet de rappeler que ce langage n’est pas conçu pour des applications très sophistiquées et que pour des manipulations plus complexes, d’autres langages doivent être préférés.

La méthode pour émuler un tableau en batch se fonde sur le fait que les caractères spéciaux #$'()\*+,-.?@[]\_`{}~ ainsi que les chiffres sont autorisés dans les noms de variables et en particulier []. Il est donc parfaitement correct d’écrire SET my\_tab[1]=1 et ECHO %my\_tab[1]% renvoie 1.

On voit immédiatement que la méthode a le désavantage que les éléments du «tableau» doivent être assignés un à un. De plus il n’existe pas d’objet my\_tab en tant que tel avec un ensemble de méthodes rendant sa manipulation aisée. Itérer sur la «table» pour la remplir ou y accéder demande que l’expansion retardée des variables soit active.

Exemple :

SET i=2

ECHO %my\_tab[%i%]%

L’expression ci-dessus ne va pas marcher car l’interpréteur va chercher une variable %my\_tab[%. L’expression suivante en revanche, fonctionne :

SET i=2

ECHO !my\_tab[%i%]!

Exemple :

SET i=0

FOR %%d IN (Sunday Monday Tuesday Wednesday Thrusday Friday Saturday) DO (

SET /A i=i+1

SET day[!i!]=%%d

)

* 1. Scoping

**DOS implémente à la fois des éléments de scoping lexical et dynamique.** Le scope est simple puisqu’il n’existe que deux niveaux, le scope global (niveau auquel on se trouve par défaut) et le scope local créé notamment entre SETLOCAL et ENDLOCAL. Pour ce qui est du scoping lexical, toute variable définie entre SETLOCAL et ENDLOCAL ou à l’intérieur d’un script appelé depuis une commande n’est plus accessible une fois le programme sorti de SETLOCAL ou du script respectivement. DOS implémente aussi le scoping dynamique : un script appelé par un autre script aura accès aux éléments de l’environnement de ce dernier que ceux-ci soient à l’intérieur d’un ensemble SETLOCAL/ENDLOCAL ou non. SETLOCAL est la seule façon de créer un environnement local, sinon tout s’exécute par défaut dans l’environnement global (ou dans un *calling environment* qui peut être local).

On appelle **variables d’environnement** les variables utiles au fonctionnement de la console et à l’exécution des commandes. Cela a pour corollaire que changer ces variables peut modifier la façon dont l’interpréteur opère. **Les principales variables d’environnement sont PATH et PATHEXT.** On trouve également des variables faisant office de raccourcis (*short-cuts*) vers des répertoires d’intérêt comme USERPROFILE, TEMP, etc.

Plus généralement et en cohérence avec l’implémentation du scoping dynamique, l’interpréteur de commandes hérite des variables d’environnement du process qui l’a créée (qui a lancé la commande CMD qui a elle-même pour effet de lancer une nouvelle instance de l’invite). Par exemple, si on ouvre une l’invite depuis le bureau, c’est Windows Explorer qui est le process qui l’a créée.

Exemple 1 :

CMD ; si PATH a été modifié avant, ce PATH va s’appliquer dans cette nouvelle instance et non celui du premier démarrage de l’invite lancée depuis Explorer.

EXIT ; il semble ne rien se passer. La commande a fermé l’instance ouverte par la ligne du dessus.

EXIT ; ferme l’invite de commandes.

Exemple 2 :

CMD /C « EXIT /B 10» ; il ne se passe rien d’apparent

ECHO %ERRORLEVEL% ; retourne 10

Certaines variables d’environnement sont dites **dynamiques** (il arrive qu’on appelle les autres «statiques» par opposition). Leur valeur peut changer en fonction du contexte dans lequel elles sont accédées. **Leur valeur est recalculée à chaque évaluation.** Il s’agit par exemple des variables CD, TIME, DATE, RANDOM, ERRORLEVEL, etc.

## Assignation de variables directement par l’utilisateur (*user input*)

Il est possible de laisser à l’utilisateur le choix des valeurs à donner à une variable. Cela est permis par les commandes SET /P et CHOICE.

# Commandes

Une commande DOS consiste en :

* un nom de commande
* aucun, un ou plusieurs arguments
* aucun, un ou plusieurs commutateurs (*switches*) servant à modifier le comportement de la commande

Chacun de ces éléments est séparé par un espace (mais ,;:= ou <tab> sont aussi des séparateurs autorisés). On distingue les commandes externes (sous-entendu à l’interpréteur Windows) qui consistent en des exécutables séparés de cmd.exe et fournis avec l’OS Windows. Remplacer ces fichiers modifie le comportement de ces commandes. À l’opposé, on trouve les commandes intégrées à l’interpréteur (*built-in*) qui ne sont pas modifiables (CALL, CD, CLS, DIR, ECHO, FOR, SET, etc.).

## Commutateurs (switches)

Il s’agit d’une séquence d’un ou plusieurs caractères précédés d’un slash «/» (hyphen «-» sous UNIX) permettant de modifier le comportement de la commande. DOS est insensible à la casse mais cela peut ne pas être le cas de tous les shells. Il est préférable de séparer d’éventuels switches multiples par des espaces.

Exemples :

DIR liste l’ensemble des fichiers et répertoires du répertoire courant

DIR /A:D liste uniquement les répertoires contenus dans le répertoire courant

Si une commande d’un autre OS est «portée» sur Windows (*ported to Windows*) elle peut garder la syntaxe de son OS d’origine.

Exemple de commande à la syntaxe différente mais fonctionnant dans DOS : 7z e \*.7z -o..\Dst

Cette commande va utiliser 7zip pour extraire le contenu de l’ensemble des archives d’extension .7z contenues dans le répertoire courant vers le répertoire ..\Dst, le switch -o permettant de préciser le répertoire de destination des fichiers.

### Extensions des commandes

Certaines commandes proposent des options supplémentaires si l’extension des commandes est activée (inactive par défaut). L’activation peut se faire de deux façons :

* Ouvrir une nouvelle instance de l’interpréteur avec la commande : CMD /E:ON
* Exécuter ses commandes dans un environnement local ouvert avec SETLOCAL ENABLEEXTENSIONS. SETLOCAL a priorité sur les options de CMD.

## Syntaxe & interprétation

Dans le cas où une variable serait passée en argument d’une commande, sa valeur est substituée à son nom avant l’exécution de la commande (cf. Extension des variables). Dans le cas d’arguments intégrant des caractères spéciaux (&()[]{}^=;!'+,`~ et <espace>), ces derniers doivent être précédés du caractère d’échappement «^» ou l’expression soit être entourées de guillemets. L’échappement ne suffit pas dans le cas où l’un de ces caractères serait un espace auquel cas les guillemets doivent être utilisés.

Exemple :

ECHO «Johnson & Sons» ou ECHO Johnson ^& Sons mais pas ECHO Johnson & Sons

SET "%~1=some\_value"

ECHO est un mauvais exemple pour les espaces mais l’idée ici est surtout d’échapper «&» qui est sinon interprété comme opérateur de conjonction inconditionnelle (cf. plus bas).

### Caractères génériques

Les caractères génériques (*wildcards*) «\*» et «?» sont souvent utilisés en arguments de commandes (DIR, FOR, COPY, etc.). Dans un contexte normal, ils symbolisent une chaîne arbitrairement longue de caractères pour «\*» ou un ou aucun caractère pour «?». Ainsi, ???.txt va correspondre à tous les fichiers textes dont le nom comporte entre zéro et trois caractères autre que «.» (\*.txt attrape l’ensemble des fichiers textes). À son évaluation, une expression comportant des caractères génériques est remplacée par une longue chaîne de caractères juxtaposant toutes les chaînes correspondant au motif. Chaque élément étant isolé par des caractères séparateurs, on peut notamment itérer sur la valeur retournée. Contrairement à d’autres shells, CMD.exe ne procède pas à l’expansion des caractères génériques avant évaluation (*wildcard expansion*), c’est à chaque programme de le faire pendant leur exécution.

## Code sortie

Toutes les commandes DOS retourne un entier appelé code sortie. La valeur du code sortie (*exit code, return code, exit status*) de la dernière commande exécutée est passée à la variable d’environnement ERRORLEVEL. La valeur du code sortie permet de savoir si l’exécution est un succès et sinon peut donner des indications sur la nature de l’échec de la commande. Par convention, un succès correspond à 0, un échec à une autre valeur souvent positive.

Il est encouragé d’intégrer le retour d’un code sortie à ses scripts. Cela s’implémente via la commande EXIT.

Exemple : EXIT /B 1

Cette commande ferme le script en cours (sans fermer l’invite CMD.exe qui l’a appelé grâce au switch /B) en fixant ERRORLEVEL à 1. Les valeurs prises par ERRORLEVEL sont souvent associées à des éléments de contrôle du flux du programme.

## Sorties standards et redirections

Les entrées et sorties des différentes commandes sont appelées entrées et sorties standards. Il y a une entrée standard (*standard input stdin*) et deux sorties : la sortie standard (*standard output stdout*) et la sortie d’erreur (*standard error stderr*). Ces entrées/sorties peuvent être alimentées/redirigées par/via des fichiers grâce aux opérateurs « < », « > » et « >> » qui prennent en paramètres quelle(s) sortie(s) doit être redirigée (stdout par défaut) :

* Commande > some\_file.txt

Stdout est écrite dans some\_file.txt écrasant son contenu.

* Commande >> some\_file.txt

Stdout est écrite dans some\_file.txt à la suite du contenu déjà présent.

* Commande < some\_file.txt

Stdin est alimentée par la lecture du contenu de some\_file.txt

Il est également possible de se référer à stdin, stdout et stderr par les chiffres 0, 1 et 2 respectivement. On peut aussi rediriger un flux vers un autre à l’aide de « >& » pour par exemple les écrire dans un même fichier.

Exemples :

DIR \*.txt > NUL && ECHO Success ; Redirige la sortie standard vers le périphérique NUL (il en existe d’autres comme CON, COM1, etc. mais ils servent peu ou ont vieilli), seul Success est imprimé à l’écran. À l’inverse, DIR \*.txt && ECHO Success aurait imprimé la sortie standard de DIR puis Success.

DIR \*.txt 2 > listing\_err.log ; Redirige stderr et non stdin vers listing\_err.log.

DIR \*.txt > listing.log 2 > listing\_err.log ; Redirige chacune des sorties vers un fichier différent.

DIR \*.txt > listing.log 2>&1 ; Redirige stderr vers stdout, flux ensuite écrit dans listing.log

Remarque : CON est le périphérique correspondant à la console dont la sortie standard correspond aux commandes qu’on y a tapé. On peut ainsi directement créer un script directement dans la console en écrivant TYPE CON > my\_script.txt suivi de toutes nos commandes, sortir de ce mode demande d’insérer le caractère EOF avec CTRL+Z avant d’appuyer sur ENTREE.

## Syntaxe pour la combinaison de plusieurs commandes

### Rediriger la sortie d’une commande vers l’entrée d’une autre : le *pipeline*

Deux commandes peuvent être enchaînées pour former un *pipeline* à l’aide de « | » (*pipe operator*).

Exemple : command1 | command2

La sortie standard de 1 est alors passée à l’entrée standard de 2.

### Conditionner l’exécution d’un groupe de commandes au succès d’un autre : le *compound*

Il est fréquent de vouloir conditionner l’exécution d’un groupe de commande au succès ou à l’échec de celle d’un autre groupe de commandes. Plutôt que d’utiliser un IF/ELSE avec ERRORLEVEL, DOS fournit des opérateurs syntaxiques équivalents appelés conjonctions :

* & *unconditionnal conjunction* : Pipeline1 & Pipeline2

Pipeline2 est toujours exécuté quelle qu’est été l’issue de l’exécution de Pipeline1

* && *positive conditionnal conjunction* : Pipeline1 && Pipeline2

Pipeline2 n’est exécuté que si l’exécution de Pipeline1 est un succès (ERRORLEVEL = 0)

* || *negative conditionnal conjunction* : Pipeline1 || Pipeline2

Pipeline2 n’est exécutée que si l’exécution de Pipeline1 est un échec (ERRORLEVEL > 0)

Exemples :

some\_command && ECHO Succes est équivalent à IF %ERRORLEVEL% EQU 0 ECHO Succes

some\_command || ECHO Failure est équivalent à IF %ERRORLEVEL% NEQ 0 ECHO Failure

### Récupérer la sortie d’un groupe de commandes : *parenthesized compound*

Un compound placé entre parenthèse va s’exécuter comme une commande simple dont on va pouvoir rediriger les sorties standards.

Exemple : (compound) > some\_file.txt

Exemples :

ECHO Hello | ECHO World > some\_file.txt ; Le fichier va contenir World

(ECHO Hello | ECHO World) > some\_file.txt ; Le fichier va contenir Hello World

FOR %i in (1,2,3) do @ECHO %i > some\_file.txt ; Le fichier va contenir 3

(FOR %i in (1,2,3) do @ECHO %i) > some\_file.txt ; Le fichier va contenir 1,2,3

## Appeler un script comme une commande

Pour pouvoir exécuter un script en ne tapant que son nom (sans son extension), il faut satisfaire à deux conditions :

* Le script doit pouvoir être trouvé dans PATH ou CD (il doit en fait être dans le scope, visible)
* Son extension doit pouvoir être trouvée dans PATHEXT qui regroupe toutes les extensions de fichier script dont on peut s’économiser l’écriture dans l’invite de commandes (.cmd, .exe, .bat, etc).

À l’intérieur d’un script il faut ajouter une condition supplémentaire : l’appel à un autre script doit obligatoirement avoir recours à CALL. On peut aussi utiliser CALL pour appeler un script depuis la console mais ce n’est pas obligatoire.

Ainsi si my\_script.cmd est dans CD par exemple, les commandes passées dans l’invite sont équivalentes : Test ; Test.cmd ; CALL Test ; CALL Test.cmd

Si on ajoute .py à PATHEXT par exemple (PATHEXT=%PATHEXT% ;.py), plus besoin de taper l’extension des scripts Python visibles de la console pour les exécuter.

# Fonctions

Il n’y a pas de fonctions à proprement parler en Batch mais des sous-routines peuvent être émulées à l’aide de CALL, SETLOCAL, ENDLOCAL et de *labels* (appelés étiquettes en français). CALL peut être utilisé aussi bien pour appeler une fonction définie à l’intérieur du script, on utilise alors son *label* (:func\_name), qu’un autre script, on utilise alors son chemin, absolu ou relatif.

On se place ici dans le contexte de la rédaction d’un script (créer des fonctions dans la console n’a pas de sens).

## Arguments

Les arguments passés à une fonction sont ajoutés à la suite de son nom et séparés par des espaces ou tout autre séparateur valable (,;= ou <tab>). On y accède via les références de la forme %i avec i entre 1 et 9, i se référant à la position de l’argument en partant du nom de la fonction. Dans l’éventualité d’un nombre d’arguments supérieur à 10, les récupérer va demander de s’aider de la commande SHIFT.

L’appel d’une fonction prend ainsi la forme : CALL :my\_function arg1 arg2

Les corps de fonctions sont placés à la fin du script. Au moment où le programme rencontre le CALL :label, il saute à la ligne labellisée correspondante en fin de programme[[2]](#footnote-2).

Exemple :

CALL :my\_function arg1 arg2

… Instructions exécutées après la fonction

GOTO :EOF

:my\_function

… Corps de la fonction

GOTO :EOF

Comme EXIT, GOTO :EOF permet de sortir d’un script ou d’une sous-routine (fonction). D’un point de vue purement syntaxique, cette fonction permet aussi de délimiter le corps d’une fonction. Contrairement à EXIT, GOTO :EOF ne permet pas d’assigner une valeur à ERRORLEVEL. Utiliser EXIT peut ainsi apparaitre plus propre. EXIT est souvent utilisée avec /B qui permet de sortir d’un script sans fermer CMD.exe. L’exemple suivant est aussi valable :

Exemple :

@ECHO OFF

CALL :setError

ECHO %ERRORLEVEL%

EXIT /B 0

:setError

EXIT /B 5

Dans cet exemple, EXIT /B 5 ne sort pas du script mais de la fonction. ECHO %ERRORLEVEL% sera bien exécutée et retournera 5. Taper la même commande après la fin de l’exécution du script retourne 0 grâce à EXIT /B 0.

En fait l’appel d’une fonction à l’intérieur d’un script via CALL :label créé un nouveau « contexte de fichier de commandes » comme si on avait en fait lancé un script séparé (alors que le texte de la fonction et celui de la commande qui l’appelle sont concrètement écrites dans le même fichier). Comme pour n’importe quel script, il faut pouvoir le quitter avec EXIT ou EOF. Terminer sa fonction par EXIT ou GOTO :EOF est donc indispensable. On comprend aussi que le EXIT en queue de fonction ne quitte pas tout mais seulement le contexte de fichier de commande dans lequel le corps de la fonction est exécuté. Il faut voir et écrire sa fonction comme s’il s’agissait d’un script séparé, à part entière.

### Reference variables

Dans le cas où l’argument (de la fonction, du script ou d’une commande comme FOR) est un nom de fichier, il est possible de récupérer des métadonnées sur le fichier et pas simplement son nom grâce à des références spéciales. Celles-ci sont de la forme suivante (exemple pour le 2e argument %2 - cf. l’aide de FOR pour l’ensemble de ces commandes) :

%~f2 : renvoie le chemin absolu du fichier (full path)

%~p2 : renvoie le chemin du parent directory du fichier

%~dp2 : renvoie le chemin du parent directory du fichier avec le lecteur

%~z2 : renvoie la taille du parent directory du fichier

Etc.

Dans le cas d’un script, les références comme %~f0 ou %~dp0 donnent des informations sur le contexte du script lui-même et peuvent être utiles.

Dans le contexte d’une boucle FOR (itérant par exemple sur l’output de DIR), ces références fonctionnent sur la variable muette. On peut utiliser des références type %%~dpi dans la séquence de commandes à exécuter à chaque itération.

Les *reference variables* sont sujettes à l’expansion des variables.

## Contexte local d’une fonction et valeur retour

Une fonction ne retourne pas de valeur en Batch. Ce n’est pas forcément un problème si celle-ci s’exécute dans l’environnement global dont une des variables peut être utilisée pour le résultat. En revanche, dans le cas où l’utilisateur aurait choisi d’émuler un environnement d’exécution local en plaçant SETLOCAL/ENDLOCAL dans le corps de la fonction, sortir le résultat est plus compliqué.

Exemple avec variable globale :

CALL :myGetFunc var1

ECHO var1 after : %var1%

:myGetFunc

SET "%~1=DosTips"

GOTO :EOF

Dans le cas où la fonction opère dans un contexte local, on fait une utilisation astucieuse de l’expansion des variables pour sortie une ou plusieurs valeurs d’un contexte local.

Exemple :

SET var=1

SETLOCAL

SET foo=2

ENDLOCAL & Set var=%foo%

ECHO %var%

Après ENDLOCAL, le contexte local est détruit et la variable foo perdue et Set var=%foo% ne devrait pas fonctionner comme attendu après ENDLOCAL. En revanche, comme Set var=%foo% est dans la même commande que ENDLOCAL, la commande qui est réellement exécutée après expansion des variables est ENDLOCAL & Set var=2. On a ainsi réussi à ramener dans le contexte global une valeur qui n’était assignée que dans un contexte local.

## Template de fonction DOS

:myFunctionName -- function description here

:: -- %~1: argument description here

SETLOCAL

REM --function body here

set LocalVar1=...

set LocalVar2=...

(ENDLOCAL & REM -- RETURN VALUES

IF "%~1" NEQ "" SET %~1=%LocalVar1%

IF "%~2" NEQ "" SET %~2=%LocalVar2%

)

GOTO:EOF

SETLOCAL est souvent accompagné de ENABLEDELAYEDEXPANSION et ENABLEEXTENSIONS.

## De la fonction au script

Un script s’apparente à une fonction dont le corps correspond au texte du fichier exécuté. Tout ce qui a été dit pour les fonctions quant aux arguments et aux *reference variables* reste valable. Il faut veiller à garder l’hygiène de faire retourner à son script un code de sortie. Mis à part le retour de valeurs, le modèle de fonction donné précédemment reste valable pour un script batch.

Il est fréquent de créer un environnement local pour l’exécution en commençant le script par SETLOCAL. Là aussi cela peut être vu comme une bonne pratique dans le sens où notre script ne peut ainsi venir modifier des variables de l’environnement global (ex : PATH).

# Structures de contrôle

Batch ne propose que deux structures de contrôle rudimentaires : IF et FOR. IF est élémentaire dans le sens où il ne propose qu’un unique ELSE.

## IF

IF est de la forme IF [NOT] condition commande. En fonctionnement normal les conditions sont de trois formes :

* ERRORLEVEL nombre
* string1==string2 ; sensible à la casse par défaut
* EXIST file\_name

Dans la même veine qu’EXIST pour les fichiers, on peut contrôler si une variable existe avec DEFINED.

Exemple : IF NOT DEFINED var (SET var = default\_value)

Exemple : sur l’utilisation des guillemets

SET var = Hello World!

IF «%var%» == «Hello World!» (commande)

Le LHS exige d’ajouter des guillemets à cause de l’espace, la chaîne de caractère ne sera plus rigoureusement identique au contenu de la variable, il faut donc ajouter ces mêmes guillemets au RHS.

Si les extensions de commandes sont activées, on peut se servir des opérateurs de comparaison (EQU, NEQ, LSS, LEQ, GTR, GEQ) pour des commandes de la forme IF string1 NEQ string2 commande.

Si la commande est entre parenthèses, la parenthèse ouvrante doit se trouver sur la ligne du IF. Idem pour ELSE.

Exemple :

IF condition (

commande1

) ELSE (

commande2

)

## FOR

Le FOR de base est très limité et de la forme : FOR %variable IN (ensemble) DO commande [args]

Dans un script de commandes, la variable muette doit être précédée de deux % : %%variable. Les *reference variables* peuvent être utilisées notamment si on itère sur un ensemble de noms de fichiers (Ex : %%~dpi pour une variable muette %%i). L’ensemble sur lequel on va itérer est globalement formé des éléments issus du fractionnement d’une chaîne de caractères au niveau de ses caractères séparateurs. L’ensemble peut être produit à l’aide de caractères génériques.

Exemple :

FOR %%i IN (1,2,3) DO @ECHO %%i ; va retourner 1 2 3, ne pas oublier “@” si tapé directement dans l’invite

FOR %%i IN (\*.txt) DO @ECHO %%~zi ; va retourner la taille de chaque fichier texte du CD

Comme pour IF, si la commande est entre parenthèses et sur plusieurs lignes, la parenthèse ouvrante doit être sur la même ligne que FOR.

Si les extensions de commandes sont activées, FOR propose différents modes de fonctionnement potentiellement intéressants. Les plus intéressants correspondent aux switchs :

* /D : L’expansion de caractères génériques dans (ensemble) ne va retourner que des répertoires et non des fichiers. Utile pour itérer sur les répertoires contenus dans le CD.
* /R : L’expansion de caractères génériques dans (ensemble) va retourner l’ensemble des fichiers satisfaisant la condition et présents dans l’arborescence du répertoire passé en paramètre (CD par défaut). Si (ensemble) = (.), on va itérer sur tous les sous-répertoires de l’arborescence.
* /L : Permet de créer une séquence de nombres pour (ensemble) qui est de la forme (début, pas, fin).
* /F : Ce mode permet plusieurs choses dont d’itérer sur les lignes ou les mots d’un fichier (pas forcément très utile). Sa principale utilité est de pouvoir itérer sur la sortie standard d’une commande passée dans (ensemble). La commande doit être entourée de single quotes.

Ex : FOR /F %%i IN ('DIR /B /A:D') do commande

1. “A command-line interface is a means of interacting with a computer program where the user (or client) issues commands to the program in the form of successive lines of text (command lines). A program which handles the interface is called a command language interpreter or shell.” [↑](#footnote-ref-1)
2. Même principe que GOTO :label qui renvoie le programme vers une ligne labellisée (*labelled line*). [↑](#footnote-ref-2)