Référence des commandes Tcl
Tcl/Tk, Apprentissage et Référence

http://www.vuibert.fr

Table des matières

Ce chapitre de référence fournit une documentation aussi complète que possible concernant toutes les commandes de base de Tcl. Elle est établie à partir de la documentation officielle qui accompagne les distributions de Tcl/Tk. Il s'agit ici de la documentation correspondant à la toute récente version 8.4 de Tcl mais nous précisons dans le texte les commandes ou sous-commandes qui sont apparues dans cette version et qui n'existaient pas antérieurement.

Pour chaque commande, les sous-commandes disponibles sont listées par ordre alphabétique. Il en va de même des options.

Par convention les noms de commandes et sous-commandes sont écrits en caractères gras. Les options (introduites par un tiret) sont également en caractères gras. Les arguments des commandes sont indiqués par un nom en italiques et leur signification est indiquée dans l'explication qui suit les commandes. Un argument optionnel sera entouré de points d'interrogations, comme ceci : *?Script?*. Ces points d'interrogation sont bien entendu une convention d'écriture et ne doivent pas figurer dans les scripts. Des points de suspension à la suite d'un argument indiquent qu'il peut y avoir un nombre arbitraire d'arguments de ce type.

Sous Unix, on peut accéder à l'information (en anglais) relative à chaque commande Tcl au moyen de la commande man à partir de n'importe quelle fenêtre de terminal. Par exemple :

```
bash-2.04# man n llength
```

Pour les valeurs booléennes les termes suivants sont équivalents et peuvent être utilisés indifféremment (y compris sous forme abrégée et en lettre majuscules) :

- *0*, *false*, *no* et *off* pour faux.
- *1*, *true*, *yes* et *on* pour vrai.

Le fichier ReferenceTk.pdf du CD-ROM répertorie de même les commandes de base de Tk.

after

La commande **after** permet de provoquer l'exécution d'une commande après un délai spécifié. Elle est utilisée pour retarder l'exécution d'un programme ou pour provoquer l'exécution d'une commande plus tard comme tâche d'arrière-plan. Elle possède plusieurs sous-commandes qui correspondent à divers modes d'utilisation :

after *ms*
: L'argument *ms* est un entier indiquant le temps en millisecondes. La commande dort pendant *ms* millisecondes et retourne. Pendant ce temps là l'application ne répond plus aux événements qu'elle reçoit.

after *ms ?Script Script Script...?*
: Sous cette forme, la commande retourne immédiatement mais fait en sorte qu'une commande Tcl soit exécutée *ms* millisecondes plus tard comme procédure de rappel. La commande sera exécutée une fois seulement au moment indiqué. Elle est obtenue en concaténant tous les arguments à la manière de la commande **concat** (cf. p. 41). L'exécution se fait au niveau global, en dehors du contexte de toute procédure.
Si une erreur se produit pendant l'exécution de la commande différée, le mécanisme d'erreurs de la commande **bgerror** est utilisé pour rendre compte de l'erreur. La commande **after** renvoie un identificateur qui peut servir à annuler la commande différée avec une instruction **after cancel**.

after cancel *Id*
: Annule l'exécution d'une commande dont l'exécution a été programmée pour plus tard. L'argument *Id* indique quelle commande doit être annulée ; c'est une valeur qui aura été obtenue précédemment par une commande **after**. Si la commande en question a déjà été exécutée entre temps, l'annulation est bien entendu sans effet.

after cancel *Script Script...*
: Cette commande, comme la précédente, sert à annuler l'exécution d'une commande différée. Les arguments sont concaténés en une chaîne unique en les séparant avec une espace à la manière de la commande **concat** : s'il y a une commande en suspens qui correspond à cette chaîne, c'est elle qui est annulée. Si aucune commande ne correspond, l'annulation sera sans effet.

after idle *Script ?Script Script...?*
: Concatène les arguments *Script* en une chaîne unique en les séparant avec une espace à la manière de la commande **concat** et fait en sorte que le script qui en résulte soit exécuté plus tard comme une fonction de rappel pendant un temps mort de l'application (*idle callback*). Le script sera évalué exactement une fois la prochaine fois que la boucle d'événements sera parcourue et qu'il n'y aura plus d'événements en attente.
Cette commande renvoie un identificateur qui peut servir à annuler la commande différée avec une instruction **after cancel**. Si une erreur se produit

pendant l'exécution de la commande différée, le mécanisme d'erreurs de la commande **bgerror** est utilisé pour rendre compte de l'erreur.

after info *?Id?*
Cette commande renvoie des informations concernant les procédures de rappel existantes (*event handlers*). Si aucun argument *Id* n'est fourni, la commande renvoie une liste de tous les identificateurs de gestionnaires d'événements créés avec la commande **after** dans cet interpréteur. Si l'argument *Id* est spécifié, il désigne une procédure existante ; *Id* est une valeur qui aura été obtenue précédemment par un appel à la commande **after** et la commande différée ne doit pas encore avoir été déclenchée, ni avoir été annulée. Dans ce cas la commande **after info** renvoie une liste de deux termes : le premier est le script associé et le second est l'un des mots **idle** ou **timer** pour indiquer le type de gestionnaire dont il s'agit.

Les formes de syntaxe **after** *ms* et **after idle** supposent bien sûr que l'application soit pilotée par événements : le mécanisme ne peut fonctionner que si l'application entre dans la boucle d'événements. Pour des applications qui ne sont pas normalement pilotées par événements (comme par exemple tclsh), on devra utiliser les commandes **vwait** et **update** pour entrer dans la boucle d'événements (cf. p. 203 et 197).

append

La commande **append** adjoint de nouvelles valeurs à la valeur d'une variable.

Syntaxe

append *NomVar?Valeur Valeur Valeur...?*

Tous les arguments *Valeur* sont adjoints par concaténation à la valeur actuelle de la variable *NomVar*. Si cette variable n'existe pas déjà, elle est créée avec pour valeur la concaténation de tous les arguments *Valeur*. Cette commande fournit un moyen efficace pour construire des variables très longues.

AppleScript

La commande **AppleScript** permet d'exécuter des scripts en langage AppleScript depuis Tcl.

Syntaxe

AppleScript *sous-commande ?arg arg ...?*

Description

Cette commande est invoquée sur Macintosh pour utiliser les services de la composante OSA (*Open Scripting Architecture*) appelée AppleScript. Le langage AppleScript permet de communiquer avec le système lui-même et avec les autres applications : il permet en particulier de piloter les autres applications et de leur faire échanger des données. En faisant exécuter des scripts AppleScript depuis Tcl, on accède ainsi à toute la puissance du système de communication entre applications sous MacOS. La commande **AppleScript** n'a donc de signification que dans un environnement Macintosh. Elle ne fait pas partie des commandes de base de Tcl et n'est disponible que si l'on utilise l'extension TclAppleScript. On la charge au moyen d'une instruction :

```
package require Tclapplescript
```

Avec cette extension, il est possible de compiler des scripts, d'exécuter des scripts compilés, d'exécuter directement des données de scripts ou inversement de décompiler un script. On peut aussi charger un script compilé à partir d'une ressource de type *scpt* ou inversement stocker un script compilé sous forme d'une ressource *scpt*.

Les sous-commandes reconnues par la commande **AppleScript** sont les suivantes :

AppleScript compile *?-option valeur? donnéesScript1 ?donnéesScript2 ...?*

Les éléments *donnéesScript* sont concaténés avec un espace entre chacun d'eux et envoyés à AppleScript pour être compilés. Il n'y a pas de limitation concernant la taille du script *donnéesScript* si ce n'est la mémoire disponible dans l'interpréteur Wish lui-même.

Si la compilation réussit, la commande renvoie un identificateur que l'on peut passer ensuite à la commande **run** pour exécuter le script compilé. En cas d'échec au contraire, la valeur de retour sera le message d'erreur émis par AppleScript : la ligne de code concernée contiendra un symbole de soulignement _ pour indiquer l'endroit exact où a été détectée l'erreur.

La compilation est contrôlée par des paires de type option/valeur. Les options disponibles sont les suivantes :

-augment *booléen*
Cette option doit être utilisée conjointement avec l'option **-context**. Dans le cas de l'exécution d'un script dans un contexte (voir la documentation d'AppleScript pour de plus amples détails et l'option **-context** ci-dessous), si cette option a la valeur *yes* ou *true*, les données de l'argument *donnéesScript* seront ajoutées au script de contexte ; si cette option a la valeur *no* ou *false*, les données de l'argument *donnéesScript* remplacent celles qui sont contenues dans le script de contexte.
La valeur par défaut est *yes*.

-context *booléen*
Cette option permet que le script désigné par l'argument *donnéesScript* soit compilé dans un « contexte ». Dans le vocabulaire d'AppleScript, un contexte est l'équivalent de la création d'un nom d'espace dans Tcl. Dans ce cas, la sous-commande **compile** renvoie le nom du contexte plutôt que le nom d'un script compilé.
Dans un contexte, on peut stocker des données et des procédures (appelées *handlers* en anglais dans le jargon d'AppleScript). Par la suite, on peut exécuter d'autres scripts dans ce contexte : ceux-ci verront et auront accès aux données et aux procédures stockées dans le contexte. Pour exécuter un script dans un contexte, on se sert de l'option **-context**, disponible aussi avec les sous-commandes **execute** et **run**, en lui transmettant le nom du contexte.
À la différence d'une commande **compile** ordinaire, le code compilé avec l'option **-context** est exécuté immédiatement afin de mettre en place tout de suite le contexte.

-name *nom*
L'argument *nom* servira à nommer le script ou le contexte compilés par la commande **compile**. S'il existe déjà un script portant ce nom, il sera ignoré. Il en va de même avec les contextes de script à moins que l'option **-augment** n'ait la valeur *true* (voir plus haut). Si cette option n'est pas spécifiée, un nom unique par défaut sera créé à la place. Dans tous les cas, le nom est renvoyé par la commande **compile**.

-parent *nomContexte*
Cette option doit être utilisée conjointement avec l'option **-context**. L'argument *nomContexte* doit être le nom d'un script compilé comme script de contexte. Dans ce cas le nouveau script de contexte qui est compilé héritera des données et des procédures stockées dans ce script parent.

AppleScript decompile *nomScript*
Cette commande décompile les données compilées dans le script *donnéesScript* et renvoie le code source.

AppleScript delete *donnéesScript*
Cette commande détruit les données compilées dans le script *donnéesScript* et libère toutes les ressources en mémoire qui lui sont associées.

AppleScript execute *?options valeur? donnéesScript1 ?donnéesScript2...?*
Cette commande compile *et* exécute le script désigné par l'argument *données-Script* (après concaténation) et renvoie le résultat de cette exécution. C'est l'équivalent des commandes **compile** et **run** successivement, si ce n'est que, dans ce cas, le script compilé est oublié aussitôt qu'il a été exécuté.

AppleScript info *typeInfo*
Cette commande donne des informations sur la connexion établie avec la composante OSA qui est la composante système chargée de gérer les langages tels que AppleScript. L'argument *typeInfo* peut prendre les valeurs suivantes :

contexts ?motif?
Cette constante permet d'obtenir la liste des scripts de contexte qui ont été compilés. Si l'argument *motif* est spécifié, seuls les contextes dont le nom correspond au motif seront listés.

scripts ?motif?
Cette constante permet d'obtenir la liste des scripts qui ont été compilés au cours de la session. Si l'argument *motif* est spécifié, seuls les scripts dont le nom correspond au motif seront listés.

AppleScript load *?option valeur? nomFichier*
Cette commande charge en mémoire des données compilées stockées dans une ressource de type *scpt* du fichier désigné par l'argument *nomFichier* et renvoie un identificateur pour ces données. Le script est chargé mais n'est pas exécuté.

On notera que des scripts compilés au moyen de l'application Éditeur de Scripts (*Script Editor*) fournie par Apple, sont compilés comme des scripts de contexte mais, à la différence de la commande **compile -context**, la commande **load** ne les exécute pas automatiquement : on devra explicitement invoquer une commande **run** pour mettre en place les données et les procédures qu'ils contiennent.

La commande **load** accepte les options suivantes :

-rsrcname *nomRessource*
Permet de charger une ressource *scpt* en la désignant par son nom *nom-Ressource*.

-rsrcid *ressourceID*
Permet de charger une ressource *scpt* en la désignant par son numéro d'identification *ressourceID*.

Si aucune des deux options **-rsrcname** ou **-rsrcid** n'est spécifiée, la commande **load** recherchera la ressource 128. C'est le numéro de ressource qu'utilise aussi l'éditeur de scripts *Script Editor*.

AppleScript run *?option valeur? donnéesScript*
Cette commande exécute un script préalablement compilé dans *données-Script*. Si l'exécution réussit, la commande renvoie la valeur de retour du script sous forme d'une chaîne. Sinon elle renvoie l'erreur émise par l'in-

terpréteur AppleScript (la composante OSA). La sous-commande **run** accepte les options suivantes :

-context *nomContexte*
: L'argument *nomContexte* désigne un nom de contexte résultant d'un appel préalable à la commande **compile** utilisée avec son option **-context**. Dans ce cas, le code représenté par l'argument *donnéesScript* sera exécuté dans ce contexte : il aura accès à toutes les données et les procédures qui ont été définies dans ce contexte.

AppleScript store *?option valeur? donnéesScript nomFichier*
: Cette commande stocke un script compilé ou un script de contexte dans une ressource de type 'scpt' du fichier désigné par l'argument *nomFichier*. La commande **store** accepte les options suivantes :

-rsrcname chaîne
: Stocke la ressource par son nom.

-rsrcid nbEntier
: Stocke la ressource par son numéro d'identification.

Si aucune des deux options **-rsrcname** ou **-rsrcid** n'est spécifiée, la commande **store** utilise traditionnellement la ressource 128. C'est le numéro de ressource qu'utilise aussi l'éditeur de scripts *AppleScript Editor* que l'on trouve sur les systèmes MacOS.

array

La commande **array** permet la manipulation des variables de type tableau associatif.

Syntaxe

array *Sous-Commande Tableau ?Arg Arg...?*

Description

En fonction de la sous-commande utilisée, la commande **array** permet d'accomplir toutes sortes de tâches concernant les variables tableau. Dans tout ce qui suit l'argument *Tableau* désigne une variable valide de ce type. Les sous-commandes reconnues sont les suivantes :

array anymore *Tableau IdRecherche*
: Renvoie 1 s'il reste des éléments à examiner dans une recherche au sein d'un tableau, 0 si tous les éléments ont été renvoyés. L'argument *IdRecherche* est une valeur obtenue antérieurement grâce à la commande **array startsearch** et indique quelle recherche effectuer dans le tableau. Cette option est utile lorsqu'un tableau possède un élément avec un nom vide car la valeur obtenue par une instruction **array nextelement** n'indique pas si la recherche a abouti.

array donesearch *Tableau IdRecherche*
: Cette commande met fin à une recherche (c'est-à-dire un parcours à travers tous les éléments du tableau) et détruit l'état associé à cette recherche ; elle renvoie une chaîne vide. L'argument *IdRecherche* est une valeur obtenue antérieurement grâce à la commande **array startsearch** et indique quelle recherche détruire dans le tableau.

array exists *Tableau*
: Renvoie 1 si *Tableau* est effectivement une variable de type tableau, 0 s'il n'y a pas de variable de ce nom ou bien s'il s'agit d'une variable de type scalaire.

array get *Tableau ?Motif?*
: Renvoie une liste constituée de paires d'éléments. Le premier élément est le nom d'un élément de *Tableau* et le second élément est la valeur qui lui est associée. L'ordre des paires est indéfini. Si l'argument *Motif* n'est pas spécifié, tous les éléments du tableau figurent dans le résultat, sinon, seulement les éléments dont le nom correspond au motif y sont inclus. Les règles concernant la syntaxe du motif sont les mêmes qu'avec la commande **string match** (cf. p. 174). Si *Tableau* n'est pas le nom d'une variable de type tableau ou si celui-ci n'a aucun élément alors une liste vide est renvoyée.

array names *Tableau ?Mode? ?Motif?*
: Renvoie une liste constituée des noms de tous les éléments du tableau qui correspondent au motif *Motif*. L'argument optionnel *Mode* a été introduit avec la version 8.4 de Tcl : il peut avoir l'une des valeurs **-exact**, **-glob** ou

-regexp. Ce mode concerne le type de règle à appliquer pour chercher des correspondances entre les noms et le motif : avec les options **-glob** ou **-regexp** le motif peut contenir des métacaractères selon la syntaxe reconnue respectivement par les commandes **string match** (cf. p. 174) ou bien **regexp** (cf. p. 142). Avec l'option **-exact**, il doit y avoir correspondance littérale entre le nom et le motif. L'option par défaut est **-glob**. Si *Tableau* n'est pas le nom d'une variable de type tableau ou si aucun élément ne correspond au motif alors une liste vide est renvoyée.

array nextelement *Tableau IdRecherche*

Renvoie le nom de l'élément suivant dans le tableau *Tableau*, ou une chaîne vide si tous les éléments du tableau ont déjà été renvoyés par une recherche de ce type. L'argument *IdRecherche* est une valeur obtenue antérieurement grâce à la commande **array startsearch** et identifie la recherche.

Si des éléments sont ajoutés ou supprimés du tableau, toutes les recherches sont immédiatement interrompues comme si une commande **array donesearch** avait été exécutée ; les opérations **array nextelement** échoueront alors sur ces recherches.

array set *Tableau Liste*

Fixe les valeurs d'un ou plusieurs éléments du tableau *Tableau*. L'argument *Liste* doit avoir la forme d'une liste contenant un nombre pair d'éléments analogue à celles retournées par la commande **array get** vue précédemment. Chaque élément d'indice impair sera considéré comme une clé du tableau et l'élément qui le suit sera considéré comme sa valeur. Si la variable tableau n'existe pas déjà, elle est créée ; si en outre la liste *Liste* est vide, un tableau vide est créé.

array size *Tableau*

Renvoie une valeur décimale représentant le nombre d'éléments du tableau. Si *Tableau* n'est pas le nom d'une variable tableau existante, la valeur 0 est renvoyée.

array startsearch *Tableau*

Cette commande initialise une recherche élément par élément dans le tableau *Tableau*, de telle sorte que la commande **array nextelement** puisse être employée ensuite afin de parcourir un à un tous les noms des éléments du tableau. Une fois cette recherche terminée, il est nécessaire d'appeler la commande **array donesearch**. Cette sous-commande **startsearch** renvoie une valeur qui servira d'identificateur utilisé par **nextelement** et **donesearch** : ceci permet d'effectuer plusieurs parcours ou recherches en parallèle parmi tous les éléments d'un tableau.

array statistics *Tableau*

Renvoie des informations statistiques concernant la distribution des données au sein du tableau : nombre d'entrées, nombre de pointeurs de la table de hachage, distance moyenne d'un élément. Cette commande n'existe que depuis la version 8.4 de Tcl.

array unset *Tableau ?Motif?*

Supprime tous les éléments du tableau qui correspondent à un certain motif. Si le motif n'est pas spécifié, c'est le tableau entier qui est supprimé. Les règles de correspondance avec le motif sont les mêmes que pour **string match** (cf. p. 174). S'il n'y a pas de variable de ce nom ou bien si aucune correspondance n'est trouvée, aucune erreur n'est générée. La commande renvoie toujours une chaîne vide.

bgerror

La commande **bgerror** est une procédure invoquée pour traiter les erreurs en arrière-plan.

Syntaxe

bgerror *Message*

Description

La commande **bgerror** n'est pas une commande de base de Tcl. C'est une procédure qui est appelée par Tcl en cas d'erreurs pendant l'exécution de certaines opérations comme tâche d'arrière-plan. Elle devra être définie, comme procédure Tcl, par chaque application individuelle ce qui permet une très grande flexibilité dans le traitement des erreurs.

Une erreur d'arrière-plan (en anglais *background error* dont *bgerror* est l'abréviation) est une erreur qui se produit dans un procédure de rappel ou qui provient d'une commande extérieure à l'application. Par exemple, si une erreur se produit pendant l'exécution d'une commande spécifiée avec la commande **after**, ce sera une erreur d'arrière-plan.

Pour une erreur ordinaire, les mécanismes internes de Tcl permettent de faire remonter cette erreur depuis la procédure où elle a eu lieu à travers la série des appels de procédures jusqu'au sommet d'où l'application peut rendre compte de l'erreur comme elle le souhaite. Pour une erreur d'arrière-plan, le déroulement des appels aboutit à la bibliothèque Tcl et il n'y a pas de moyen évident de la récupérer.

Lorsque Tcl détecte une telle erreur, il l'enregistre et invoque la commande **bgerror** plus tard dès qu'un temps mort se produit dans l'application. Avant d'invoquer **bgerror**, Tcl recharge dans les variables **errorInfo** et **errorCode** l'information sauvegardée sur l'erreur puis il appelle **bgerror** avec le message d'erreur comme unique argument. Cela suppose que l'application a implémenté une commande **bgerror** et que celle-ci a été définie de manière à rendre compte de l'erreur dans le cadre de l'application. Tcl ignore tout résultat renvoyé par la commande **bgerror** (à moins qu'il ne s'agisse encore d'une erreur !).

Si une autre erreur se produit à l'intérieur de la procédure alors Tcl rend compte de cette erreur lui-même en envoyant un message sur le canal d'erreurs *stderr*.

Si plusieurs erreurs d'arrière-plan s'accumulent avant que **bgerror** ne soit invoquée, cette procédure sera ensuite appelée une fois pour chacune des erreurs dans l'ordre dans lequel elles sont apparues.

Il n'y a pas de définition par défaut de la procédure **bgerror** dans Tcl mais il y en a une dans Tk : elle affiche une fenêtre de dialogue contenant le message d'erreur et proposant de montrer une trace de la pile indiquant où l'erreur s'est produite. Cette fenêtre de dialogue possède un bouton supplémentaire qui peut être configuré par l'application et qui, par défaut, propose de sauvegarder l'extrait de la trace dans un fichier sur disque. Le comportement de ce bouton peut être

personnalisé par l'application en réglant certaines valeurs dans la base de ressources (cf. section **??**) : on utilise l'expression **ErrorDialog.function.text* pour spécifier le nom du bouton et **ErrorDialog.function.command* pour désigner la commande à exécuter. Le texte de la trace est ajouté comme argument à la commande lorsque celle-ci est évaluée. Si l'une de ces options est une chaîne vide alors le bouton ne sera pas affiché.

binary

La commande **binary** permet d'insérer ou d'extraire des parties ou des champs d'une chaîne de données binaires.

Syntaxe

binary format *ChaîneFormat ?Arg Arg...?*
binary scan *Chaîne ChaîneFormat ?NomVar NomVar...?*

Description

La commande **binary** procure toutes les facilités pour la manipulation de données binaires. Sous la forme **binary format** elle permet de créer des données binaires à partir de valeurs Tcl ordinaires. Sous la forme **binary scan** elle effectue l'opération inverse : extraire des données d'une chaîne binaire et renvoyer des valeurs Tcl ordinaires.

La commande *binary format*

La chaîne binaire générée par la commande **binary format** est modelée selon un format fourni par l'argument *ChaîneFormat*. Cet argument est constitué d'une série de 0 ou plusieurs spécificateurs de champ séparés par 0 ou plusieurs espacements. Chaque spécificateur de champ est une lettre-type unique éventuellement suivie d'un nombre *Nb*. La plupart de ces spécificateurs utilisent un des arguments *Arg* pour obtenir la valeur qu'ils doivent formater. Le spécificateur a pour rôle d'indiquer comment elle doit être formatée.

Le nombre optionnel *Nb* sert à indiquer combien d'éléments du type spécifié sont pris dans la valeur spécifiée par l'argument *Arg*. On l'exprime soit au moyen d'un nombre entier positif ou nul, soit au moyen du symbole $*$ qui indique normalement que tous les éléments de la valeur passée doivent être utilisés. Si le nombre d'arguments fournis ne correspond pas au nombre des spécificateurs de champs, une erreur est générée.

La chaîne binaire renvoyée est construite progressivement en parcourant les paires *lettre-type/nombre* de gauche à droite et en ajoutant à chaque fois les nouveaux octets formatés.

Les lettres-types admises par la commande **binary format** sont les suivantes :

a

Produit en sortie une chaîne de caractères de longueur *Nb*. Si l'argument *Arg* correspondant a moins de *Nb* octets, des octets nuls sont ajoutés automatiquement pour compléter. Si l'argument *Arg* est plus long que la longueur spécifiée, les caractères en trop sont ignorés. Si l'argument *Nb* est un astérisque

∗, tous les octets de *Arg* sont utilisés. Enfin si *Nb* est omis, seul le premier caractère sera retenu. Par exemple,

```
binary format a7a*a Unix Macintosh Windows
```

renvoie une chaîne d'octets équivalente à

```
Unix\000\000MacintoshW
```

A

Cette lettre-type se comporte comme **a** à la seule différence que ce sont des espaces et non des octets nuls qui sont utilisés pour compléter une chaîne trop courte. Par exemple,

```
binary format A6A*A alpha bravo charlie
```

renvoie

␣␣␣␣Unix␣␣MacintoshW

avec deux espaces qui séparent les deux mots.

b

Produit en sortie une chaîne de *Nb* nombres binaires dont les bits sont ordonnés du plus faible au plus fort. L'argument *Arg* correspondant doit être constitué d'une série de 0 et de 1 : les octets résultants sont produits du premier au dernier mais dans chacun d'eux l'ordre des bits va du plus faible au plus fort. Si l'argument *Arg* a moins de *Nb* chiffres, des zéros sont ajoutés pour compléter. Si l'argument *Arg* est plus long que la longueur spécifiée, les chiffres en trop sont ignorés. Si l'argument *Nb* est un astérisque ∗, alors tous les chiffres de l'argument *Arg* seront formatés. Si *Nb* est omis, un seul chiffre est formaté. Enfin si le nombre de bits formatés ne correspond pas à la fin d'un octet, les bits restants du dernier octets seront des zéros. Par exemple,

```
binary format b5b* 11100 111000011010
```

renvoie une chaîne d'octets équivalente à

```
\x07\x87\x05
```

B

Cette lettre-type se comporte comme **b** à la seule différence que les bits sont ordonnés du plus fort au plus faible dans chaque octet. Par exemple,

```
binary format B5B* 11100 111000011010
```

renvoie une chaîne d'octets équivalente à

```
\xe0\xe1\xa0
```

h

Produit en sortie une chaîne de *Nb* nombres hexadécimaux dont les bits sont ordonnés du plus faible au plus fort. L'argument *Arg* correspondant doit être constitué de nombres hexadécimaux (0123456789abcdefABCDEF) : les octets résultants sont produits du premier au dernier mais dans chacun des nombres hexadécimaux produits l'ordre des bits va du plus faible au plus fort. Si l'argument *Arg* a moins de *Nb* chiffres, des zéros sont ajoutés pour compléter. Si l'argument *Arg* est plus long que le nombre de chiffres spécifié, les chiffres en trop sont ignorés. Si l'argument *Nb* est un astérisque ∗, alors tous les

chiffres de l'argument *Arg* sont formatés. Si *Nb* est omis, un seul chiffre est formaté. Enfin si le nombre des chiffres formatés ne correspond pas à la fin d'un octet, les bits restants du dernier octets seront des zéros. Par exemple,

```
binary format h3h* AB def
```

renvoie une chaîne d'octets équivalente à

```
\xba\x00\xed\x0f
```

H

Cette lettre-type se comporte comme **h** à la seule différence que les bits sont ordonnés du plus fort au plus faible dans chaque octet. Par exemple,

```
binary format H3H* ab DEF
```

renvoie une chaîne d'octets équivalente à

```
\xab\x00\xde\xf0
```

c

Produit en sortie une ou plusieurs valeurs entières sur 8-bit. Si aucun nombre *Nb* n'est spécifié, alors l'argument *Arg* correspondant doit être une valeur entière ; sinon *Arg* doit être une liste comportant au moins *Nb* nombres entiers. Les huit bits de poids faible de chaque entier sont formatés comme une valeur sur un seul octet. Si l'argument *Nb* est un astérisque $*$, alors tous les entiers de la liste sont formatés. Si le nombre d'éléments dans la liste est inférieur à *Nb*, une erreur est générée. Si le nombre d'éléments dans la liste est supérieur à *Nb*, les éléments excédentaires sont ignorés. Par exemple,

```
binary format c3cc* {3 -3 128 1} 260 {2 5}
```

renvoie une chaîne d'octets équivalente à

```
\x03\xfd\x80\x04\x02\x05
```

tandis que

```
binary format c {2 5}
```

provoque une erreur.

s

Cette lettre-type se comporte comme **c** à la différence que les nombres sont formatés sur 16 bits, c'est-à-dire sur deux octets, et dans l'ordre dit « *little endian* » autrement dit en plaçant l'octet de poids faible avant l'octet de poids fort.[1] Par exemple,

```
binary format s3 {3 -3 258 1}
```

renvoie une chaîne d'octets équivalente à

```
\x03\x00\xfd\xff\x02\x01
```

1. Si l'on a besoin de connaître le type de représentation utilisé par le système en vigueur sur une certaine machine, il suffit d'interroger la variable globale tcl_platform avec l'instruction suivante :

set tcl_platform(byteOrder)

Le résultat sera soit *bigEndian*, soit *littleEndian*.

S

Cette lettre-type se comporte comme **s** à la différence que les nombres formatés sur 16 bits sont produits dans l'ordre dit « *big endian* » autrement dit en plaçant l'octet de poids fort avant l'octet de poids faible. Par exemple,

```
binary format S3 {3 -3 258 1}
```

renvoie une chaîne d'octets équivalente à

```
\x00\x03\xff\xfd\x01\x02
```

i

Cette lettre-type se comporte comme **c** à la différence que les nombres sont formatés sur 32 bits, c'est-à-dire sur quatre octets, et dans l'ordre dit « *little endian* » autrement dit en commençant par l'octet le moins significatif. Par exemple,

```
binary format i3 {3 -3 65536 1}
```

renvoie une chaîne d'octets équivalente à

```
\x03\x00\x00\x00\xfd\xff\xff\xff\x00\x00\x01\x00
```

I

Cette lettre-type se comporte comme **i** à la différence que les nombres formatés sur 32 bits sont produits dans l'ordre dit « *big endian* » autrement dit en commençant par l'octet le plus significatif. Par exemple,

```
binary format I3 {3 -3 65536 1}
```

renvoie une chaîne d'octets équivalente à

```
\x00\x00\x00\x03\xff\xff\xff\xfd\x00\x01\x00\x00
```

f

Cette lettre-type se comporte comme **c** à la différence que les nombres sont formatés comme des nombres en virgule flottante en simple précision. Le format de représentation de ces nombres dépend du système d'exploitation utilisé. Il est donc déconseillé d'utiliser ce format pour communiquer des valeurs sur d'autres plates-formes ou à travers un réseau étant donné que la machine qui les recueillera pourrait avoir une représentation différente. La taille des nombres en virgule flottante n'étant pas la même d'une architecture à l'autre, le nombre d'octets produits par ce type de formatage peut varier. Si la valeur excède la valeur maximale acceptée sur une plate-forme particulière, la valeur FLT_MAX définie par le système sera utilisée à la place. D'un autre côté, Tcl utilise des nombres en double précision pour sa représentation interne : il peut donc y avoir une perte de précision lors de la conversion en simple précision. Par exemple, sous un système Windows tournant sur un processeur Pentium, l'instruction

```
binary format f2 {1.6 3.4}
```

renvoie une chaîne d'octets équivalente à

```
\xcd\xcc\xcc\x3f\x9a\x99\x59\x40
```

d

Cette lettre-type est analogue à **f** à la différence que les nombres sont formatés comme des nombres en virgule flottante en double précision. Les mêmes

remarques que précédemment concernant la taille des résultats produits s'appliquent. Par exemple, sous un système Windows tournant sur un processeur Pentium, l'instruction

```
binary format d1 {1.6}
```

renvoie une chaîne d'octets équivalente à

```
\x9a\x99\x99\x99\x99\x99\xf9\x3f
```

x

Produit en sortie *Nb* octets nuls. Cette lettre-type n'est pas associée à un argument *Arg*. Si l'argument *Nb* n'est pas spécifié, un seul octet nul est produit. Si *Nb* est un astérisque (∗), une erreur est générée. Par exemple,

```
binary format a3xa3x2a3 abc def ghi
```

renvoie une chaîne d'octets équivalente à

```
abc\000def\000\000ghi
```

X

Déplace le curseur en arrière de *Nb* octets dans la chaîne produite. Si l'argument *Nb* est un astérisque (∗) ou bien s'il est supérieur à la position courante, le curseur est renvoyé à la position 0 de telle sorte que le prochain octet produit sera le premier de la chaîne. Si *Nb* est omis, le curseur remonte d'une position. Cette lettre-type n'est pas associée à un argument *Arg*. Par exemple,

```
binary format a3X*a3X2a3 abc def ghi
```

renverra

```
dghi
```

@

Déplace le curseur dans la chaîne produite à la position absolue spécifiée par le nombre *Nb*. La position 0 correspond au premier octet. Si *Nb* correspond à une position au-delà du dernier octet, alors toutes les positions manquantes seront remplies par des octets nuls et le curseur se retrouvera ainsi à la position spécifiée. Si l'argument *Nb* est un astérisque (∗), le curseur est envoyé à la fin de la chaîne produite. Si *Nb* est omis, une erreur est générée. Cette lettre-type n'est pas associée à un argument *Arg*. Par exemple,

```
binary format a5@2a1@*a3@10a1 abcde f ghi j
```

renverra

```
abfdeghi\000\000j
```

La commande *binary scan*

binary scan *Chaîne ChaîneFormat ?NomVar NomVar...?*

La commande **binary scan** analyse les champs d'une chaîne binaire et renvoie à la fin le nombre de conversions opérées. L'argument *Chaîne* désigne la chaîne à analyser et l'argument *ChaîneFormat* indique comment l'analyser. Chacun des

arguments *NomVar* indique le nom d'une variable : quand un champ de la chaîne *Chaîne* est examiné, le résultat est assigné à la variable correspondante.

Comme avec la commande **binary format**, l'argument *ChaîneFormat* est constitué d'une série de 0 ou plusieurs spécificateurs de champ séparés par 0 ou plusieurs espacements. Chaque spécificateur de champ est une lettre-type unique éventuellement suivie d'un nombre *Nb*. La plupart de ces spécificateurs utilisent un des arguments *NomVar* pour obtenir le nom d'une variable dans laquelle stocker le résultat de la conversion opérée : l'association entre un spécificateur et une variable se fait de gauche à droite. Le spécificateur a pour rôle d'indiquer comment les données binaires doivent être interprétées.

Le nombre optionnel *Nb* sert à indiquer combien d'éléments du type spécifié sont à prendre dans la chaîne binaire. On l'exprime soit au moyen d'un nombre entier positif ou nul, soit au moyen du symbole $*$ qui indique normalement que tous les éléments des données doivent être utilisés. S'il n'y a pas assez d'octets après la position courante pour satisfaire le spécificateur courant alors la variable correspondante est laissée inchangée et la commande **binary scan** s'interrompt et renvoie immédiatement le nombre de variables fixées jusque là. S'il n'y a pas assez de variables pour tous les champs de la chaîne *ChaîneFormat* qui en ont besoin, une erreur est générée.

La chaîne renvoyée est construite progressivement en parcourant les paires *lettre-type/nombre* de gauche à droite dans la chaîne binaire fournie en entrée et en ajoutant à chaque fois les nouveaux octets convertis.

Les lettres-types admises par la commande **binary scan** sont les suivantes :

a

Les données sont une chaîne de caractères de longueur *Nb*. Si l'argument *Nb* est un astérisque (*), tous les octets restant de la chaîne *Chaîne* seront traités et envoyés dans la variable correspondante. Si l'argument *Nb* est omis, un seul caractère est examiné. Par exemple,

```
binary scan abcde\000fghi a6a10 var1 var2
```

renverra 1 et la chaîne équivalente à `abcde\000` sera stockée dans la variable *var1* tandis que la variable *var2* restera inchangée.

A

Cette forme est analogue au spécificateur **a** à la différence que les espaces en fin de chaîne et les octets nuls sont supprimés de la chaîne traitée avant qu'elle soit stockée dans la variable associée. Par exemple,

```
binary scan "abc efghi  \000" A* var1
```

renverra 1 et la variable *var1* contiendra *abc efghi*.

b

Les données sont converties en une chaîne de *Nb* chiffres binaires (séquences de 0 et de 1) énumérés dans l'ordre allant du poids faible au poids fort. Les octets sont traités de gauche à droite et dans chacun d'entre eux les bits sont placés du plus faible au plus fort. Tout bit excédentaire dans le dernier octet sera ignoré. Si l'argument *Nb* est un astérisque (*), alors tous les bits restants

de l'argument *Chaîne* seront traités. Si *Nb* est omis, un bit seulement est traité. Par exemple,

```
binary scan \x07\x87\x05 b5b* var1 var2
```

renverra 2 car deux conversions sont opérées: la variable *var1* contiendra la valeur `11100` et la variable *var2* contiendra la valeur `1110000110100000`.

B

Cette forme est analogue au spécificateur **b** à la différence que les bits sont énumérés en allant du poids fort au poids faible. Par exemple,

```
binary scan \x70\x87\x05 B5B* var1 var2
```

renverra 2 car deux conversions sont opérées: la variable *var1* contiendra la valeur `01110` et la variable *var2* contiendra la valeur `1000011100000101`.

h

Les données sont converties en une chaîne de *Nb* chiffres hexadécimaux (pris parmi la série « 0123456789abcdef ») ordonnés du poids faible au poids fort. Les octets sont traités de gauche à droite et dans chacun d'entre eux les chiffres hexadécimaux sont placés du plus faible au plus fort. Tout bit excédentaire dans le dernier octet sera ignoré. Si l'argument *Nb* est un astérisque (∗), alors tous les chiffres hexadécimaux restants de l'argument *Chaîne* seront traités. Si *Nb* est omis, un chiffre hexadécimal seulement est traité. Par exemple,

```
binary scan \x07\x86\x05 h3h* var1 var2
```

renverra 2 car deux conversions sont opérées: la variable *var1* contiendra la valeur `706` et la variable *var2* contiendra la valeur `50`.

H

Cette forme est analogue au spécificateur **h** à la différence que les chiffres sont énumérés en allant du poids fort au poids faible. Par exemple,

```
binary scan \x07\x86\x05 H3H* var1 var2
```

renverra 2 car deux conversions sont opérées: la variable *var1* contiendra la valeur `078` et la variable *var2* contiendra la valeur `05`.

c

Les données sont converties en *Nb* entiers signés sur 8 bits et stockées sous forme de liste Tcl dans la variable correspondante. Si l'argument *Nb* est un astérisque (∗), tous les octets restants de l'argument *Chaîne* seront traités. Si *Nb* est omis, un octet seulement est traité. Par exemple,

```
binary scan \x07\x86\x05 c2c* var1 var2
```

renverra 2 car deux conversions sont opérées: la variable *var1* contiendra la liste {`7 -122`} et la variable *var2* contiendra la valeur `5`.
On notera que les entiers renvoyés sont signés. On peut toutefois les convertir en entiers 8-bit non signés en utilisant une expression telle que :

```
expr ( $num + 0x100 ) % 0x100
```

s

Les données sont interprétées comme *Nb* entiers signés sur 16 bits et représentés dans l'ordre dit « *little endian* » autrement dit en plaçant l'octet de poids faible avant l'octet de poids fort. Les entiers sont stockés sous forme de liste Tcl dans la variable correspondante. Si l'argument *Nb* est un astérisque (∗), tous les octets restants de l'argument *Chaîne* seront traités. Si *Nb* est omis, un entier 16-bit seulement est traité. Par exemple,

```
binary scan \x05\x00\x07\x00\xf0\xff s2s* var1 var2
```

renverra 2 car deux conversions sont opérées: la variable *var1* contiendra la liste {`5 7`} et la variable *var2* contiendra la valeur `-16`.
On notera que les entiers renvoyés sont signés. On peut toutefois les convertir en entiers 16-bit non signés en utilisant une expression telle que :

```
expr ( $num + 0x10000 ) % 0x10000
```

S

Cette forme est analogue au spécificateur **h** à la différence que les données sont interprétées comme des entiers signés sur 16 bits et représentés dans l'ordre dit « *big endian* » autrement dit en plaçant l'octet de poids fort avant l'octet de poids faible. Par exemple,

```
binary scan \x05\x00\x07\x00\xf0\xff S2S* var1 var2
```

renverra 2 car deux conversions sont opérées: la variable *var1* contiendra la liste {`1280 1792`} et la variable *var2* contiendra la valeur `-3841`.

i

Les données sont interprétées comme *Nb* entiers signés sur 32 bits et représentés dans l'ordre dit « *little endian* » autrement dit en plaçant l'octet de poids faible avant l'octet de poids fort. Les entiers sont stockés sous forme de liste Tcl dans la variable correspondante. Si l'argument *Nb* est un astérisque (∗), tous les octets restants de l'argument *Chaîne* seront traités. Si *Nb* est omis, un entier 32-bit seulement est traité. Par exemple,

```
set val \x05\x00\x00\x00\x07\x00\x00\x00\xf0\xff\xff\xff
binary scan $val i2i* var1 var2
```

renverra 2 car deux conversions sont opérées: la variable *var1* contiendra la liste {`5 7`} et la variable *var2* contiendra la valeur `-16`.
On notera que les entiers renvoyés sont signés et qu'il n'est pas possible pour Tcl de les représenter comme des valeurs non signées.

I

Cette forme est analogue au spécificateur **i** à la différence que les données sont interprétées comme des entiers signés sur 32 bits et représentés dans l'ordre dit « *big endian* » autrement dit en plaçant l'octet de poids fort avant l'octet de poids faible. Par exemple,

```
set val \x00\x00\x00\x05\x00\x00\x00\x07\xff\xff\xff\xf0
binary scan $val I2I* var1 var2
```

renverra 2, la variable *var1* contiendra la liste `5 7` et la variable *var2* contiendra la valeur `-16`.

f

Les données sont interprétées comme *Nb* nombres en virgule flottante en simple précision. Les nombres produits sont stockés sous forme de liste Tcl dans la variable correspondante. Si l'argument *Nb* est un astérisque (∗), tous les octets restants de l'argument *Chaîne* seront traités. Si *Nb* est omis, un seul nombre en virgule flottante et simple précision est traité. La taille des nombres en virgule flottante n'étant pas la même d'une architecture à l'autre, le nombre d'octets traités par ce type de spécificateur peut varier. Si les données traitées ne représentent pas un nombre en virgule flottante et simple précision qui soit valide, le résultat est imprévisible et dépend du compilateur. Par exemple, sous un système Windows tournant sur un processeur Pentium, l'instruction

```
binary scan \x3f\xcc\xcc\xcd f var1
```

renverra 1 et la variable *var1* contiendra la valeur `1.6000000238418579`.

d

Cette forme est analogue au spécificateur **f** à la différence que les données sont interprétées comme des nombres en virgule flottante en double précision conformément à la représentation adoptée sur le système d'expoitation utilisé. Par exemple, sous un système Windows tournant sur un processeur Pentium, l'instruction

```
binary scan \x9a\x99\x99\x99\x99\x99\xf9\x3f d var1
```

renverra 1 et la variable *var1* contiendra la valeur `1.6000000000000001`.

x

Déplace le curseur vers l'avant de *Nb* octets dans la chaîne *Chaîne*. Si l'argument *Nb* est un astérisque (∗) ou bien s'il est supérieur au nombre d'octets situés après la position courante, le curseur est renvoyé après le dernier octet de *Chaîne*. Si *Nb* est omis, le curseur avance d'une position. Ce spécificateur n'est pas associé à un nom de variable *NomVar*. Par exemple,

```
binary scan \x01\x02\x03\x04 x2H* var1
```

renverra 1 et la variable *var1* contiendra la valeur `0304`.

X

Déplace le curseur en arrière de *Nb* octets dans la chaîne *Chaîne*. Si l'argument *Nb* est un astérisque (∗) ou bien s'il est supérieur à la position courante, le curseur est renvoyé à la position 0 de telle sorte que le prochain octet produit sera le premier de la chaîne. Si *Nb* est omis, le curseur remonte d'une position. Ce spécificateur n'est pas associé à un nom de variable *NomVar*. Par exemple,

```
binary scan \x01\x02\x03\x04 c2XH* var1 var2
```

renverra 2, la variable *var1* contiendra la liste {`1 2`} et la variable *var2* contiendra la valeur `020304`.

@

Déplace le curseur dans la chaîne de données *Chaîne* à la position absolue spécifiée par le nombre *Nb*. La position 0 correspond au premier octet. Si l'argument *Nb* est un astérisque (∗) ou bien s'il correspond à une position

au-delà de la fin de la chaîne, alors le curseur est positionné *après* le dernier octet. Si *Nb* est omis, une erreur est générée. Ce spécificateur n'est pas associé à un nom de variable *NomVar*. Par exemple,

```
binary scan \x01\x02\x03\x04 c2@1H* var1 var2
```

renverra 2, la variable *var1* contiendra la liste {`1 2`} et la variable *var2* contiendra la valeur `020304`.

▷ Il faut faire attention au fait que les spécificateurs **c**, **s** et **S** (de même que **i** et **I** sur les systèmes 64-bit) seront traités dans des valeurs longues. Il en résulte que si le bit fort de ces valeurs (0x80 pour les caractères, 0x8000 pour les nombres de type *short*, 0x80000000 pour les nombres de type *int*) se trouve mis à 1, les valeurs seront considérées comme signées. Par exemple, après les deux instructions suivantes

```
set signShort [binary format s1 0x8000]
binary scan $signShort s1 val
```

la variable *val* contiendra la valeur `0xFFFF8000`, c'est-à-dire -32768. Pour obtenir une valeur non signée, on peut procéder comme ceci en utilisant un masque :

```
set val [expr {$val & 0xFFFF}]
```

Dans ce cas, la variable *val* contiendra la valeur `0x8000`, c'est-à-dire 32768 en décimal.

break

La commande **break** interrompt l'exécution d'une structure itérative.

Description

Cette commande est utilisée exclusivement à l'intérieur d'une instruction répétitive installée par une commande **for**, **foreach** ou **while**. L'exécution de la boucle est interrompue immédiatement et le programme se poursuit à la suite de la commande itérative qui contient cette commande **break**. La valeur interne renvoyée par cette commande est TCL_BREAK. Pour interrompre uniquement l'exécution d'une itération particulière et passer à l'itération suivante, voir la commande **continue**.

catch

La commande **catch** permet d'évaluer une série d'instructions et d'intercepter les erreurs qui pourraient s'y produire afin de permettre au programme de poursuivre son exécution malgré tout.

Syntaxe

catch *Script ?NomVar?*

Description

La commande **catch** sert à empêcher des erreurs éventuelles d'interrompre l'exécution d'un programme. L'argument *Script* est un script, c'est-à-dire un ensemble d'instructions adressées à l'interpréteur Tcl, qui est susceptible de produire des erreurs : si l'on considère que les erreurs produites éventuellement par ce script ne devraient pas empêcher le programme de se poursuivre, on peut les intercepter au moyen de la commande **catch**. Le script est en général délimité par une paire d'accolades. La commande **catch** se terminera toujours sans générer d'erreur : quoi qu'il arrive **catch** adresse à l'interpréteur, en cas d'erreur dans le script, une valeur interne non nulle qui évite toute interruption.

On peut cependant récupérer le message d'erreur qu'aura produit le script si l'on précise le nom *NomVar* d'une variable comme argument optionnel : cette variable contiendra le texte du message d'erreur s'il y a lieu ou bien simplement la valeur de retour du script si aucune exception ne s'est produite au cours de son exécution.

La commande **catch** renvoie la valeur 0 (la constante interne TCL_OK) si le script n'a généré aucune erreur et 1 le cas échéant. Il ne faut pas confondre la valeur renvoyée par **catch** avec la valeur renvoyée par le script.

Cette commande est capable de capturer toutes les exceptions ainsi que la plupart des erreurs. Seules les erreurs de syntaxe découvertes au moment de la compilation du script ne sont pas interceptées : en effet **catch** intercepte les erreurs qui se produisent en cours d'exécution alors qu'une erreur de syntaxe sera détectée par l'interpréteur au moment où le script sera interprété donc avant que **catch** n'ait pu intervenir.

cd

La commande **cd** permet de changer le répertoire courant.

Syntaxe

cd *?NomRépertoire?*

Description

Si l'argument *NomRépertoire* est spécifié, il désignera le nouveau répertoire courant (comme on peut le vérifier avec la commande **pwd**) ; autrement, c'est le répertoire HOME de l'utilisateur qui deviendra le répertoire courant. La commande **cd** retourne une chaîne vide.

clock

La commande **clock** est utilisée pour l'obtention ou la manipulation de dates dans différents formats.

Syntaxe

clock *Sous-commande?Arg Arg...?*

Description

En fonction de la sous-commande utilisée, on pourra soit obtenir des valeurs concernant le temps écoulé, soit formater des valeurs temporelles obtenues à partir d'autres commandes. Les durées sont représentées au moyen de valeurs décimales correspondant au temps écoulé depuis une certaine date de référence. Ce temps peut être mesuré de diverses façons : en secondes ou fractions de secondes ou bien en battements d'horloge (en anglais *ticks*).

clock clicks?**-milliseconds**?
: La sous-commande **clicks** renvoie une valeur temporelle obtenue auprès du système d'exploitation lui-même : cette valeur décimale est exprimée dans une unité dépendante du système qui correspond en principe à une résolution maximale (cycles d'horloge par exemple). On utilise la valeur obtenue pour mesurer et comparer des temps écoulés autrement dit pour des évaluations relatives pour lesquelles l'unité utilisée n'a pas à être connue précisément. Cependant on peut préciser l'option **-milliseconds** afin d'obtenir une valeur qui soit garantie en millisecondes.

clock format *ValeurHorloge*?**-format** *Chaîne?*?**-gmt***Booléen?*
: Permet de convertir une valeur temporelle obtenue au moyen de commandes telles que **clock seconds** ou **clock scan** ou bien des sous-commandes **atime** ou **mtime** de la commande **file** en un format particulier. Le format désiré est spécifié en utilisant l'option **-format** grâce à une chaîne descriptive : cette chaîne est constituée de descripteurs de champs correspondants aux divers éléments de la date que l'on souhaite faire apparaître. Tous ces descripteurs sont constitués d'un symbole pourcentage (%) suivi d'une lettre dont la signification est indiquée dans le tableau 1. Le tableau 2 indique des descripteurs complémentaires disponibles uniquement sur les systèmes Unix et sous MacOS.
  Si jamais l'option **-format** n'est pas utilisée alors le format par défaut est équivalent à la chaîne suivante :
  ```
  %a %b %d %H:%M:%S %Z %Y
  ```
  Par ailleurs l'option **-gmt** permet d'indiquer l'heure selon le temps moyen de Greenwich (gmt est l'abréviation de *Greenwich Mean Time*). Cette option prend l'une des valeurs **true** ou **false** suivant que l'on veut l'heure de Greenwich ou bien l'heure du fuseau horaire local.

%%	Insère un %.
%a	Nom du jour abrégé (Mon, Tue, etc.).
%A	Nom du jour complet (Monday, Tuesday, etc.).
%b	Nom du mois abrégé (Jan, Feb, etc.).
%B	Nom du mois complet.
%c	Date et heure selon la *locale* en vigueur.
%d	Quantième (01 - 31).
%H	Heure en format 24H (00 - 23).
%I	Heure en format 12H (00 - 12).
%j	Numéro du jour dans l'année (001 - 366).
%m	Numéro du mois (01 - 12).
%M	Minute (00 - 59).
%p	Indication AM/PM.
%S	Secondes (00 - 59).
%U	Numéro de semaine (00 - 52) avec dimanche premier jour.
%w	Numéro du jour (dimanche = 0).
%W	Numéro de semaine (00 - 52) avec lundi premier jour.
%x	Date selon la *locale* en vigueur.
%X	Heure selon la *locale* en vigueur.
%y	Année sans le siècle (00 - 99).
%Y	Année entière (e.g. 1990)
%Z	Nom du fuseau horaire.

TAB. 1 – *Les descripteurs de l'option* **-format** de la commande *clock*

clock scan *ChaîneDate*?**-base** *ValeurHorloge*??**-gmt** *Booléen*?

Cette sous-commande est l'opposé de **clock format** : elle part d'une date exprimée sous une forme à peu près quelconque et la convertit en une valeur décimale selon le format interne adopté pour exprimer le temps. Si seulement l'heure est donnée en argument alors il est sous-entendu que la date est la date courante. Si l'argument ne contient pas d'indication de fuseau horaire alors c'est le fuseau local qui est utilisé à moins que l'option **-gmt** n'ait été spécifiée. Cette option a la même signification qu'avec la sous-commande **clicks** (voir ci-dessus).

Si l'option **-base** est spécifiée, la valeur *ValeurHorloge* doit être une valeur décimale représentant une date valide. Seule la date correspondant à cette valeur de base est utilisée (pas l'heure) : cela permet de déterminer l'heure pour un jour particulier ou de faire des conversions relativement à une certaine date. L'option **-gmt** n'interfère pas avec **-base** puisqu'elle affecte seulement l'heure et non la date.

Il reste à expliquer la syntaxe admise pour l'argument *ChaîneDate* représentant la date à convertir au format interne. Cette chaîne peut comporter un certain nombre d'éléments obéissant à l'un des modèles suivants :

%D	Équivalent à %m/%d/%y.
%e	Quantième (1 - 31) sans zéros.
%h	Nom du mois abrégé.
%n	Insère un saut de ligne.
%r	Équivalent à %I:%M:%S %p.
%R	Équivalent à %H:%M.
%t	Insère une tabulation.
%T	Équivalent à %H:%M:%S.

TAB. 2 – *Descripteurs complémentaires pour* clock *sous Unix et MacOS.*

heure

Une heure de la journée exprimée sous les formes :

hh?:mm?:ss??? méridien?? fuseau?
hhmm? méridien?? fuseau?

Par méridien on entend l'une des deux valeurs AM ou PM pour désigner respectivement la première ou la deuxième moitié de la journée.

date

Un mois et un jour particuliers, l'année étant optionnelle. Les formats admis sont les suivants (*a* pour année, *m* pour mois, *j* pour jour, *S* pour siècle) :

mm/jj?/aa?
nom_du_mois jj?aa?
jj nom_du_mois?aa?
day jj nom_du_mois aa
?SS?aammjj
?SS?aa-mm-jj
jj-nom_du_mois-?SS?aa

Dans la quatrième forme, c'est le mot anglais *day* (signifiant *jour*) lui-même qui est utilisé. On notera effectivement d'autre part que si seulement les deux derniers chiffres de l'année sont spécifiés, la commande suppose que des valeurs entre 00 et 68 correspondent aux années 2000-2068 tandis que des valeurs entre 69 et 99 correspondent aux années 1969-1999. Les années 38-70 étant ignorées par certaines plates-formes, il peut résulter une erreur.

notation ISO-8601

Une date peut être spécifiée selon la notation de la norme ISO-8601 (où la lettre T représente elle-même) :

SSaammjjThhmmss
SSaammjj hhmmss
SSaammjjThh:mm:ss

date relative

Un certain nombre de mots-clés sont admis pour permettre de représenter des dates relatives à la date courante. Il s'agit d'un nombre entier (un nombre d'unités) suivi de l'un des termes suivants : **year**, **fortnight**, **month**, **week**, **day**, **hour**, **minute** (ou **min**), **second** (ou **sec**). Chacun de ces termes peut être utilisé indifféremment au singulier ou au pluriel. On peut leur adjoindre des modificateurs exprimés par les termes suivants : **tomorrow**, **yesterday**, **today**, **now**, **last**, **this**, **next**, **ago**. On pourra ainsi écrire par exemple :

```
clock scan "2 days ago"
```

clock seconds

Renvoie un nombre décimal exprimant la date et l'heure en nombre de secondes écoulées depuis une certaine date de référence dépendante du système d'exploitation. Cette commande est utilisée pour faire des comparaisons de dates et non pour avoir une valeur absolue car la date de référence peut varier d'un système à l'autre.

close

La commande **close** referme un canal ouvert.

Syntaxe

close *Canal*

Description

L'argument *Canal* doit être un numéro d'identification de canal valide obtenu au moyen d'une commande **open** ou **socket**. La commande **close** vide le contenu de tout tampon de sortie associé au canal mais ignore tout tampon d'entrée. Une fois fermé, le canal ne peut plus être utilisé.

Si le canal est en mode bloquant, la commande ne retournera pas tant que tout le contenu du tampon n'aura pas été vidé. Si le canal est en mode non-bloquant, et qu'il reste des données dans le tampon de sortie, la commande **close** retourne immédiatement mais le canal reste en fait ouvert afin de vider ce tampon en arrière-plan ; une fois ce tampon vidé, le canal est effectivement fermé.

Lorsque le canal correspond à une série de commandes chaînées (*pipeline*) en mode non-bloquant, la commande **close** attend que tous les processus enfants soient terminés avant de s'exécuter.

Si ce canal est partagé entre plusieurs interpréteurs, **close** le rend indisponible pour l'interpréteur qui l'a invoquée mais n'a aucun effet vis-à-vis des autres interpréteurs tant que ceux-ci ne cherchent pas à le fermer à leur tour. Lorsque le dernier interpréteur auprès duquel le canal est enregistré invoque la commande **close** alors toutes les opérations de nettoyage des tampons décrites plus haut peuvent avoir lieu.

La commande retourne une chaîne vide mais génère une erreur si une erreur se produit pendant le nettoyage des tampons.

concat

La commande **concat** permet de fusionner plusieurs listes ensemble.

Syntaxe

concat *?Arg Arg...?*

Description

Chacun des arguments *Arg* est considéré comme une liste et les espaces qui l'entourent éventuellement sont éliminés. Toutes ces listes sont fusionnées par **concat** en une liste unique, chacun des *Arg* étant séparé des autres par une espace. Si aucun argument n'est spécifié, le résultat sera une chaîne vide. Cette commande a pour effet de supprimer un niveau d'accolades. Par exemple

```
concat a b {c d e} {f {g h}}
```

renvoie

```
a b c d e f {g h}
```

continue

La commande **continue** permet de passer directement à l'itération suivante dans une boucle sans exécuter la fin de la boucle courante.

Syntaxe

continue

Description

Cette commande est utilisée exclusivement à l'intérieur d'une structure itérative installée par une commande **for**, **foreach** ou bien **while**. Si une instruction **continue** est rencontrée pendant l'exécution d'une itération particulière alors le reste des instructions contenues dans la boucle ne sera pas exécuté et on passera directement à l'itération suivante. La valeur interne renvoyée par cette commande est TCL_CONTINUE. Pour interrompre définitivement l'exécution d'une structure itérative, voir la commande **break**.

dde

La commande **dde** exécute une commande *Dynamic Data Exchange* sous Windows.

Syntaxe

dde *NomServeur ?Sujet?*
dde?**-async**? *commande nomService sujet ?Données?*

Description

Cette commande permet à une application d'envoyer des commandes *Dynamic Data Exchange* (DDE) sous Windows. *Dynamic Data Exchange* est un mécanisme par lequel des applications échangent des données brutes. Chaque transaction DDE requiert un *nom de service* et un *sujet* qui sont tous les deux définis par l'application. Par exemple Tcl utilise le nom de service **TclEval** et le sujet est le nom de l'interpréteur fourni par la commande **dde servername**. Les autres applications ont leurs propres noms de service et sujets. Par exemple, Microsoft Excel a pour nom de service **Excel**. Cette commande n'est pas une commande de base de Tcl. Pour l'utiliser, il faut spécifier le module dde comme ceci :

```
package require dde 1.1
```

La seule option de la commande **dde** est **-async** qui requiert une invocation asynchrone. Ceci n'est valide que pour la sous-commande **execute**. Normalement, la sous-commande **execute** attend jusqu'à ce que la commande s'achève, en renvoyant les messages d'erreur appropriés. Si l'option **-async** est utilisée, la commande retourne immédiatement et aucune information d'erreur n'est disponible.

Les commandes DDE

Les sous-commandes suivantes sont définies :

dde servername *?Sujet?*
: Cette commande enregistre l'interpréteur Tcl comme serveur DDE avec nom de service **TclEval** et sujet spécifié par l'argument *Sujet*. Si aucun argument *Sujet* n'est indiqué, la commande renvoie le nom du sujet courant ou une chaîne vide s'il n'y a pas de sujet couramment enregistré.

dde execute *nomService Sujet Données*
: Cette commande prend les données spécifiées par l'argument *Données* et les envoie au serveur *nomService* avec le sujet indiqué par l'argument *Sujet*. Typiquement, *nomService* est le nom d'une application, et *Sujet* est un fichier sur lequel travailler. Le champ *Données* est fourni à l'application qui les traite en général comme un script qu'elle exécute. La commande renvoie une erreur si le script n'a pas été exécuté. Si l'option **-async** est utilisée, la commande retourne immédiatement sans message d'erreur.

dde poke *nomService Sujet Élément Données*
Cette commande passe les *Données* au serveur *nomService* avec les arguments *Sujet* et *Élément* spécifiés. L'argument *Élément* est spécifique à l'application et souvent n'est pas utilisé, mais il doit toujours être non nul. Le champ *Données* est retransmis à l'application.

dde request *nomService Sujet Élément*
Cette commande est typiquement utilisée pour obtenir la valeur de quelque chose (par exemple une cellule dans Microsoft Excel ou le texte d'une sélection dans Microsoft Word). La commande renvoie la valeur de *Élément* telle qu'elle est définie par l'application.

dde services *nomService Sujet*
Cette commande renvoie une liste des paires de type *service-sujet* qui existent couramment sur la machine. Si *nomService* et *Sujet* sont tous les deux des chaînes vides, alors toutes les paires couramment disponibles sur le systéme sont renvoyées. Si *nomService* est vide et pas *Sujet*, tous les services avec le sujet spécifié sont renvoyés. Si *nomService* est non vide mais que *Sujet* l'est, tous les sujets pour le service en question sont renvoyés. Si les deux sont non vides et que la paire existe couramment, elle est renvoyée par la commande sinon la valeur de retour est nulle.

dde eval *Sujet Commande ?Arg Arg...?*
Cette commande évalue une commande et ses arguments en utilisant l'interpréteur spécifié par l'argument *Sujet*. Le serveur DDE doit être **TclEval**. Cette commande peut remplacer la commande **send** sous Windows.

DDE et TCL

Un interpréteur Tcl a toujours pour nom de service **TclEval**. Chacun des interpréteurs des applications Tcl en cours d'exécution doit avoir un nom unique spécifié par une commande **dde servername**. Chacun d'eux sera disponible comme sujet DDE seulement si la commande **dde servername** a été utilisée pour fixer le nom de sujet de chaque interpréteur. Une commande **dde services TclEval {}** renverra une liste des paires service-sujet dans laquelle chaque interpréteur en cours figurera comme sujet.

Lorsque Tcl exécute une commande **dde execute**, les données sont exécutées comme un script dans l'interpréteur désigné dans le sujet de la commande **dde execute**.

Lorsque Tcl exécute une commande **dde request**, il renvoie la valeur de la variable spécifiée dans le contexte de l'interpréteur désigné dans le sujet. Tcl reserve la variable $TCLEVAL$EXECUTE$RESULT pour son usage interne et des commandes **dde request** appliquées à cette variable donneront des résultats imprévisibles.

Une application externe qui souhaite exécuter un script dans Tcl devrait faire stocker le résultat dans une variable, exécuter la commande **dde execute** puis la commande **dde request** pour récupérer la valeur de la variable.

Avant d'utiliser DDE, on s'assurera que la file d'attente d'événements de Tcl a été vidée au moyen d'une commande **update** ou **vwait**. C'est ce qui se pro-

duit par défaut avec Wish à moins qu'une commande bloquante n'ait été appelée (comme par exemple une commande **exec** sans ajouter & pour placer un processus en arrière-plan). Si pour une raison quelconque la file d'attente d'événements n'est pas vidée, des commandes DDE pourront rester bloquées jusqu'à ce qu'elle se vide. Cela peut créer des blocages définitifs.

encoding

La commande **encoding** manipule les encodages.

Syntaxe

encoding *Sous-commande ?Arg Arg...?*

Introduction

Les chaînes Tcl sont encodées au moyen de caractères Unicode 16-bit, autrement dit sur deux octets. Des systèmes d'expoitation ou des applications externes peuvent produire des chaînes dans d'autres encodages tels que Shift-JIS pour le japonais. La commande **encoding** permet de faire le pont entre Unicode et les autres formats.

Description

Plusieurs types d'opérations peuvent être effectués en fonction de la sous-commande utilisée. Les sous-commandes admises sont les suivantes :

encoding convertfrom *?Encodage? Données*
: Convertit les données de l'argument *Données* depuis l'encodage *Encodage* à Unicode. Les caractères dans l'argument *Données* sont traités comme des données binaires où les 8 bits de poids faible de chaque caractère sont pris comme un simple octet. La séquence d'octets qui en résulte est traitée comme une chaîne dans l'encodage *Encodage* spécifié. Si *Encodage* n'est pas spécifié, l'encodage système courant est utilisé.

encoding convertto *?Encodage? Chaîne*
: Convertit l'argument *Chaîne* depuis Unicode à l'encodage *Encodage* spécifié. Le résultat est une séquence d'octets qui représente la chaîne convertie. Chaque octet est stocké dans les 8 bits faibles d'un caractère Unicode. Si *Encodage* n'est pas spécifié, l'encodage système courant est utilisé.

encoding names
: Renvoie une liste de tous les encodages disponibles. Le tableau 3 indique ceux que l'on trouve dans la version 8.4 de Tcl.

encoding system *?encodage?*
: Fixe l'encodage système à l'encodage *Encodage*. L'encodage système est utilisé chaque fois que Tcl passe des chaînes à des appels systèmes. Si l'argument *encodage* est omis, cette commande renvoie le nom de l'encodage courant.

Exemple

Un utilisateur japonais écrivant un script au moyen d'un éditeur qui produit de l'encodage euc-jp sera confronté au problème suivant : les caractères Ascii sont

ascii	cp857	euc-jp	iso8859-4	macIceland
big5	cp860	euc-kr	iso8859-5	macJapan
cp1250	cp861	gb12345	iso8859-6	macRoman
cp1251	cp862	gb1988	iso8859-7	macRomania
cp1252	cp863	gb2312	iso8859-8	macThai
cp1253	cp864	identity	iso8859-9	macTurkish
cp1254	cp865	iso2022	jis0201	macUkraine
cp1255	cp866	iso2022-jp	jis0208	shiftjis
cp1256	cp869	iso2022-kr	jis0212	symbol
cp1257	cp874	iso8859-1	koi8-r	tis-620
cp1258	cp932	iso8859-10	koi8-u	ucs-2be
cp437	cp936	iso8859-13	ksc5601	unicode
cp737	cp949	iso8859-14	macCentEuro	utf-8
cp775	cp950	iso8859-15	macCroatian	X11ControlChars
cp850	dingbats	iso8859-16	macCyrillic	
cp852	ebcdic	iso8859-2	macDingbats	
cp855	euc-cn	iso8859-3	macGreek	

TAB. 3 – *Encodages reconnus par Tcl 8.4*

représentés sur un seul octet tandis que les caractères japonais le seront sur deux octets. Or la commande **source** lit toujours les fichiers qu'elle doit interpréter au moyen de l'encodage ISO-8859-1 et donc Tcl traitera chaque octet comme un caractère *séparé* appartenant à la page 00 de l'encodage Unicode, ce qui n'est pas le souhait de l'utilisateur. La commande **encoding** devrait être utilisée dans ce cas-là pour convertir elle-même les caractères japonais dans le code Unicode qui convient. Par exemple,

```
set s [encoding convertfrom euc-jp "\xA4\xCF"]
```

renverra la chaîne Unicode `\u306F` qui représente le caractère Hiragana HA.

eof

La commande **eof** (*end of file*, fin de fichier) permet de tester si la condition de fin de fichier est atteinte sur un canal particulier.

Syntaxe

eof *Canal*

Description

La valeur renvoyée sera 1 si une condition de fin de fichier a été rencontrée au cours de la plus récente opération d'entrée effectuée sur le canal *Canal* (au moyen d'une commande **gets** par exemple) et 0 sinon.

error

La commande **error** permet de générer une erreur.

Syntaxe

error *Message ?Info? ?Code?*

Description

Cette commande renvoie un code interne de valeur TCL_ERROR qui interrompt l'interprétation des commandes. L'argument *Message* est une chaîne qui sera renvoyée à l'application pour indiquer ce qui s'est passé. Si l'argument optionnel *Info* est spécifié et est non vide, il est utilisé pour initialiser la variable globale de Tcl **errorInfo**. Cette variable interne est utilisée pour accumuler une trace dans la pile de ce qui était en cours d'exécution lorsque l'erreur s'est produite. En déroulant toutes les commandes imbriquées de la pile, l'interpréteur ajoute des informations à la variable **errorInfo**. La commande qui contient une instruction **error** n'apparaît pas dans la variable **errorInfo** : à sa place on trouvera la chaîne représentée par l'argument *Info*. On utilise ce mécanisme en conjonction avec la commande **catch** comme ceci :

```
catch {...} errMsg
set infoPrec $errorInfo
...
error $errMsg $infoPrec
```

De la sorte, le code fautif représenté par `{...}` provoque une erreur qui est capturée par **catch**. La commande **catch** place le message de l'erreur capturée dans la variable *errMsg*, le contenu actuel de la variable interne **errorInfo** est stocké dans une variable provisoire *infoPrec* puis l'instruction avec la commande **error** a pour effet de cumuler les messages contenus dans les variables *errMsg* et *infoPrec* et de les transmettre via la variable **errorInfo**. De fil en aiguille toute la série des fonctions qui ont pu être appelées est parcourue en sens inverse jusqu'à ce que le message parvienne à l'utilisateur.

Si l'argument *Code* est spécifié, sa valeur sera stockée dans la variable globale interne **errorCode** qui est destinée à contenir des descriptions de l'erreur compréhensibles par la machine pour le cas où une telle information serait disponible. En l'absence de l'argument *Code* la variable **errorCode** est automatiquement remise à la valeur NONE par l'interpréteur Tcl.

eval

La commande **eval** évalue un script Tcl.

Syntaxe

eval *Arg?Arg...?*

Description

Cette commande admet un ou plusieurs arguments qui, mis ensemble, constituent un script Tcl comprenant une ou plusieurs commandes. La commande concatène ses arguments de la même façon que la commande **concat** puis les passe à l'interpréteur et transmet le résultat de leur évaluation ou une erreur éventuelle.

exec

La commande **exec** permet de faire exécuter des sous-processus depuis un script Tcl.

Syntaxe

exec *?Commutateurs? Arg?Arg...?*

Description

Cette commande traite ses arguments comme une ligne de commande du *shell* décrivant un ou plusieurs processus à faire exécuter. Les arguments représentent donc une série de commandes chaînées (*pipeline*) dont chaque élément est un sous-processus.

Cette ligne de commandes peut être précédée par des options commençant par un tiret et qui ne font pas partie de la définition du pipeline. Les options admises sont les suivantes :

-keepnewline
: Avec cette option les symboles de fin de ligne ne seront pas éliminés de la sortie produite par le pipeline. Normalement ils le sont.

--
: Indique la fin des options. Tout ce qui suit constitue la partie des arguments *Arg*.

Une autre convention veut que si un argument (ou une paire d'arguments) prend l'une des formes suivantes il sera utilisé par la commande **exec** pour contrôler le flux des entrées et des sorties entre les processus chaînés par le pipeline. Ce sont les symboles de redirection habituels sur les systèmes de type Unix :

—
: Sépare des commandes distinctes dans le pipeline. La sortie standard de la commande qui précède la barre verticale est branchée sur l'entrée standard de la commande qui suit.

—&
: Sépare des commandes distinctes dans le pipeline. À la fois la sortie standard et le canal d'erreurs standard de la commande qui précède la barre verticale seront branchés sur l'entrée standard de la commande qui suit. Cette forme de redirection a la précédence sur des instructions de la forme 2> et >&.

< *NomFichier*
: Le fichier nommé *NomFichier* est ouvert et utilisé comme entrée standard pour la première commande du pipeline.

<@ *IdFichier*
: L'argument *IdFichier* doit être le numéro d'identification tel qu'on l'obtient à

l'ouverture d'un fichier avec la commande **open**. Ce fichier sera utilisé comme entrée standard pour la première commande du pipeline. Il faut que le fichier soit ouvert en lecture évidemment.

<< *Valeur*
L'argument *Valeur* est passé à la première commande du pipeline pour lui servir d'entrée standard.

> *NomFichier*
La sortie standard de la dernière commande du pipeline est dirigée vers le fichier nommé *NomFichier* dont le contenu sera écrasé par les nouvelles données.

2> *NomFichier*
Le canal d'erreurs standard de chacune des commandes du pipeline est redirigé vers le fichier nommé *NomFichier* dont le contenu sera écrasé par les nouvelles données.

>& *NomFichier*
À la fois le canal de sortie et le canal d'erreurs de toutes les commandes du pipeline sont redirigés vers le fichier nommé *NomFichier* dont le contenu sera écrasé par les nouvelles données.

>> *NomFichier*
La sortie standard de la dernière commande du pipeline est dirigée vers le fichier nommé *NomFichier* et les nouvelles données sont ajoutées au contenu déjà existant plutôt que de l'écraser.

2>> *NomFichier*
La sortie d'erreurs standard de chacune des commandes du pipeline est redirigée vers le fichier nommé *NomFichier* et les nouvelles données sont ajoutées au contenu déjà existant plutôt que de l'écraser.

>>& *NomFichier*
À la fois le canal de sortie et le canal d'erreurs de toutes les commandes du pipeline sont redirigés vers le fichier nommé *NomFichier* et les nouvelles données sont ajoutées au contenu déjà existant plutôt que de l'écraser.

>@ *IdFichier*
L'argument *IdFichier* doit être le numéro d'identification tel qu'on l'obtient à l'ouverture d'un fichier avec la commande **open**. La sortie standard de la dernière commande est redirigée vers le fichier d'identificateur *IdFichier* qui devra bien entendu être ouvert en écriture.

2>@ *IdFichier*
L'argument *IdFichier* doit être le numéro d'identification tel qu'on l'obtient à l'ouverture d'un fichier avec la commande **open**. La sortie d'erreurs standard de toutes les commandes du pipeline est redirigée vers le fichier d'identificateur *IdFichier* qui devra bien entendu être ouvert en écriture.

>&@ *IdFichier*
L'argument *IdFichier* doit être le numéro d'identification tel qu'on l'obtient à l'ouverture d'un fichier avec la commande **open**. À la fois le canal de sortie et la sortie d'erreurs de toutes les commandes du pipeline sont redirigés vers

le fichier d'identificateur *IdFichier* qui devra bien entendu être ouvert en écriture.

Dans toutes les commandes de la forme < *NomFichier*, l'argument *NomFichier* peut être indifféremment accolé au symbole < ou bien séparé par des espaces.

Si la sortie standard n'a pas été redirigée alors la valeur de retour de la commande **exec** sera précisément la sortie standard de la dernière commande du pipeline.

Si l'un quelconque des processus chaînés dans le pipeline est tué ou suspendu ou bien s'interrompt anormalement, alors la commande **exec** renverra une erreur et le message d'erreur contiendra la sortie du pipeline suivie des messages d'erreur décrivant l'incident. La variable **errorCode** contiendra des informations supplémentaires concernant la dernière anomalie rencontrée. D'autre part si l'un quelconque des processus écrit sur sa sortie d'erreur standard et que celle-ci n'a pas été redirigée alors **exec** renverra une erreur ; le message d'erreur contiendra la sortie standard produite par le pipeline suivie de messages concernant l'anomalie puis de la sortie d'erreurs.

Si le dernier caractère du résultat est un symbole de fin de ligne, il sera éliminé comme c'est le cas en général avec les valeurs de retour de Tcl. On peut inverser ce comportement en utilisant l'option **-keepnewline**.

Si l'entrée standard n'est pas dirigée au moyen d'un des symboles <, << ou <@ alors elle sera prise depuis l'entrée standard de l'application elle-même.

Si le dernier argument du pipeline est & alors celui-ci sera exécuté en tâche d'arrière-plan. Dans ce cas, la valeur de retour de la commande **exec** sera la liste des identificateurs de processus de tous les processus du pipeline. La sortie standard du dernier processus ira à la sortie standard de l'application si elle n'a pas été redirigée, de même pour la sortie d'erreur.

Le premier terme de chaque commande est pris comme le nom de la commande. Il y a substitution des symboles tilde éventuels et s'il n'y a aucun séparateur / alors ce sont les répertoires contenus dans la variable d'environnement PATH qui seront visités pour rechercher un exécutable de ce nom. Si le nom contient un séparateur / alors il doit faire référence à un exécutable du répertoire courant. Aucun autre type de substitution n'est opéré.

Portabilité

Les différentes plate-formes présentent quelques particularités relatives à la commande **exec**.

Windows (toutes versions)
La lecture ou l'écriture depuis un canal de connexion (*socket*) en utilisant la notation @*IdFichier* ne marche pas. La console Tk n'a pas de réelles capacités d'entrée et de sortie : depuis l'entrée standard les applications reçoivent uniquement un symbole de fin de fichier et l'information redirigée en sortie est perdue.

Aussi bien les barres obliques que les contre-obliques sont acceptées comme séparateurs de noms de chemin par les commandes Tcl. Mais il faut tenir compte de ce que, pour la plupart des applications Windows, les barres obliques sont utilisées comme séparateurs d'options et les contre-obliques commes séparateurs dans les noms de fichier. Il faut donc veiller à utiliser les bons séparateurs attendus par les processus eux-mêmes.

Avec des applications MS-DOS 16-bit ou bien les anciens systèmes Windows 3.x, tous les chemins doivent utiliser le format 8.3 de noms de fichiers et de répertoires.

Deux barres obliques ou bien contre-obliques successives désignent des adresses de réseau. C'est une source potentielle d'erreurs : la simple concaténation du répertoire racine c:/ avec un sous-répertoire /windows/system conduirait à c://windows/system qui désigne le point de montage nommé *system* sur une machine nommée *windows* (et du coup le c:/ sera ignoré) et non pas le répertoire c:/windows/system, de la machine locale. On aura intérêt à utiliser plutôt la commande Tcl **file join** pour réunir des composantes de nom de fichier.

Windows NT

Pour exécuter une application sous WindowsNT, la commande **exec** recherche d'abord avec le nom spécifié. Puis ensuite, si elle ne trouve pas d'exécutable, elle ajoute dans l'ordre les extensions *.com*, *.exe* et *.bat*. Si un nom de répertoire n'a pas été spécifié la recherche se fera dans l'ordre dans les répertoires suivants :

- le répertoire depuis lequel l'application Tcl a été lancée
- le répertoire courant
- le répertoire système Windows NT 32-bit
- le répertoire système Windows NT 16-bit
- le répertoire *home*
- les répertoires listés dans le chemin

Pour exécuter des commandes *shell* internes telles que *dir* ou *copy*, il faudra les préfixer avec l'expression `cmd.exe /c`.

Windows 95

Pour exécuter une application sous Windows 95, la commande **exec** recherche d'abord avec le nom spécifié. Puis ensuite, si elle ne trouve pas d'exécutable, elle ajoute dans l'ordre les extensions *.com*, *.exe* et *.bat*. Si un nom de répertoire n'a pas été spécifié la recherche se fera dans l'ordre dans les répertoires suivants :

- le répertoire depuis lequel l'application Tcl a été lancée
- le répertoire courant
- le répertoire système Windows 95
- le répertoire *home*
- les répertoires listés dans le chemin

Pour exécuter des commandes *shell* internes telles que *dir* ou *copy*, il faudra les préfixer avec l'expression `command.exe /c`.

Si une application MS-DOS 16-bit a lu depuis l'entrée standard et a ensuite quitté, toutes les applications MS-DOS 16-bit qui suivent trouveront le canal d'entrée fermé. Les applications 32-bits n'auront pas ce problème et fonctionneront correctement.

Une redirection entre le périphérique NUL: et une application 16-bit ne marche pas toujours. Les résultats peuvent être imprévisibles et de toutes façons désastreux.

Toutes les applications 16-bit sont exécutées de manière synchrone. Toute entrée standard envoyée par un tube (*pipe*) à une application 16-bit est collectée dans un fichier temporaire ; l'autre extrêmité du tube doit être fermée avant que l'application 16-bit ne commence son exécution. Inversement, toute sortie standard ou d'erreur depuis une application 16-bit vers un tube est collectée dans des fichiers temporaires ; l'application 16-bit doit avoir terminé avant que les fichiers temporaires ne soient envoyés vers la suite du pipeline. Ceci est dû à un bogue dans l'implémentation des tubes avec Windows 95 et c'est d'ailleurs ainsi que le *shell* DOS de Windows 95 manipule les tubes.

Certaines applications telles que **command.com** ne devraient pas être exécutées interactivement. Des applications qui accèdent directement à la console plutôt que de lire depuis une entrée standard et d'écrire sur une sortie standard peuvent échouer, figer Tcl ou le système lui-même.

Macintosh

La commande **exec** n'est pas implémentée sur les versions du système Macintosh antérieures à MacOSX car le concept de pipeline n'y existe pas.

Unix

La commande **exec** est pleinement fonctionnelle sur les systèmes Unix sans restrictions.

exit

La commande **exit** a pour effet de terminer l'application en cours d'exécution et de quitter le programme tclsh ou wish.

Syntaxe

exit *?CodeRetour?*

Description

On met fin inconditionnellement à l'exécution d'un script ou d'une application au moyen de la commande **exit** et on peut éventuellement transmettre une valeur de retour *CodeRetour* à l'attention du système (ou du *shell* sur des systèmes Unix). Par défaut la valeur renvoyée est 0.

expr

La commande **expr** évalue une expression.

Syntaxe

expr *Arg?Arg Arg...?*

Description

Les arguments *Arg* sont réunis en une seule expression, séparés par une espace les uns des autres, et cette expression est évaluée: elle renvoie presque toujours une valeur numérique (entière ou en virgule fottante). L'expression contient en principe des opérateurs, des opérandes et éventuellement des parenthèses. La syntaxe des expressions autorisées est analogue à celle des expressions du langage C mais Tcl supporte en plus des opérandes non numériques et peut faire des comparaisons de chaînes. Pour qu'une valeur ou une opération soient traitées en virgule flottante, il faut que l'un des nombres soit noté avec un point décimal. Par exemple, les trois expressions suivantes

```
expr 8 / 3
expr 8.0 / 3
expr 8.2 + 3
```

renvoient les valeurs 2, 2.66667 et 11.2 respectivement. Les valeurs en virgule flottante sont toujours renvoyées avec un point même si le résultat est entier.

Opérandes

Les opérandes, opérateurs et parenthèses peuvent être séparés par des espaces: celles-ci seront de toutes façon ignorées par l'interpréteur. Partout où c'est possible les opérandes sont interprétés comme des nombres entiers. Les entiers peuvent être exprimés en notation décimale (c'est le cas ordinaire), en notation octale (si le premier caractère est un 0) ou en notation hexadécimale (les deux premiers caractères sont 0x).

Dans les autres cas, l'opérateur sera traité, si possible, comme nombre en virgule flottante. Les notations usuelles du langage C norme ANSI sont acceptées (les suffixes f, F, l et L ne sont toutefois pas autorisés). Par exemple: 2.1, 3., 6e4, 7.91e+16.

Si aucune des interprétations précédentes n'est possible alors l'opérande est considéré comme une chaîne de caractères et seulement un nombre limité d'opérations peuvent lui être appliquées.

On peut spécifier les opérandes de plusieurs façons:

- comme valeur numérique entière ou en virgule flottante comme vu précédemment.
- comme une variable Tcl en utilisant la notation standard avec le symbole dollar $ pour désigner sa valeur.

- comme une chaîne placée entre guillemets doubles. L'interpréteur effectuera les interpolations de variables et substitutions de commandes de la chaîne passée entre les guillemets et utilisera le résultat comme opérande.
- comme une chaîne placée entre accolades { }. L'interpréteur n'effectue aucune interpolation ou substitution.
- comme une instruction Tcl placée entre crochets. Cette commande imbriquée sera exécutée et c'est son résultat qui sera utilisé comme opérande.
- comme une fonction mathématique dont les arguments prennent une des formes qui viennent d'être mentionnées comme par exemple sin($\$x$).

Opérateurs

Les opérateurs admis par la commande **expr** sont :

- + ~ !
Opérateurs unaires moins, plus, NON bit-à-bit, NON logique. Aucun de ces opérandes ne peut être appliqué à des chaînes et l'opérateur de NON bit-à-bit ne s'applique qu'à des opérandes entiers.

*** / %**
Multiplications, divisions et restes. Aucun de ces opérandes ne peut être appliqué à des chaînes et l'opérateur de reste ne s'applique qu'à des opérandes entiers. Le reste a toujours le même signe que le diviseur.

+ -
Addition et soustraction. Applicable seulement à des opérandes numériques.

¡¡ ¿¿
Décalage des bits à gauche ou à droite. Applicable seulement à des opérandes numériques. Le décalage à droite propage toujours le bit de signe.

¡ ¿ ¡= ¿=
Inférieur, supérieur, inférieur ou égal, supérieur ou égal. Ils fonctionnent comme des opérateurs logiques et renvoient 1 si la relation d'ordre est vérifiée, 0 sinon. On peut également les appliquer à des chaînes : les comparaisons de chaînes sont alors employées.

== !=
Égalité et différence. Ils renvoient 1 si la relation d'égalité est vérifiée, 0 sinon. Applicables à tous les types.

eq ne
Égalité et différence pour des chaînes. ils renvoient 1 si la relation d'égalité est vérifiée, 0 sinon. Les opérandes ne peuvent être que des chaînes. Ces opérateurs ont été introduits avec la version 8.4 de Tcl.

&
ET bit-à-bit. Applicable uniquement à des entiers.

^
OU exclusif bit-à-bit. Applicable uniquement à des entiers.

—
OU bit-à-bit. Applicable uniquement à des entiers.

abs	**ceil**	**floor**	**log**	**sin**
acos	**cosh**	**fmod**	**pow**	**sqrt**
asin	**cos**	**hypot**	**rand**	**srand**
atan2	**double**	**int**	**round**	**tanh**
atan	**exp**	**log10**	**sinh**	**tan**

TAB. 4 – *Fonctions mathématiques supportées par la commande* expr *de Tcl.*

&&
: ET logique. Renvoie 1 si les deux opérandes sont non-nuls, 0 sinon. Applicable uniquement à des valeurs numériques ou booléennes.

——
: OU logique. Renvoie 0 si les deux opérandes sont nuls, 1 sinon. Applicable uniquement à des valeurs numériques ou booléennes.

*x***?** *y*:*z*
: C'est l'instruction if-then-else du langage C. Si *x* est évalué à une valeur non nulle, alors le résultat sera la valeur de *y*. Sinon ce sera celle de *z*. L'opérande *x* doit avoir une valeur numérique.

Les règles de précédence sont les mêmes que celles qui sont en usage dans le langage C. Les opérateurs **&&**, —— et **?:** sont dits paresseux : leurs opérandes ne sont évalués que si nécessaire. Par exemple, dans l'expression

```
expr {$v ? [a] : [b]}
```

seulement la commande imbriquée **[a]** ou **[b]** sera effectivement évaluée selon la valeur de $v. Ceci n'est vrai cependant que si l'expression est entourée d'accolades, sinon l'interpréteur Tcl lui-même aura déjà substitué `[a]` et `[b]` avant même qu'ils ne soient traités par **expr**.

Fonctions mathématiques

Tcl supporte les fonctions mathématiques réunies dans le tableau 4. Leur signification est la suivante :

abs(*Arg*)
: Renvoie la valeur absolue de *Arg*. *Arg* peut être un entier ou un nombre en virgule flottante et le résultat est renvoyé dans le même type.

acos(*Arg*)
: Renvoie l'arc cosinus de *Arg* dans l'intervalle $[0,\pi]$ radians. *Arg* doit appartenir à l'intervalle $[-1,1]$.

asin(*Arg*)
: Renvoie l'arc sinus de *Arg* dans l'intervalle $[-\pi/2,\pi/2]$ radians. *Arg* doit appartenir à l'intervalle $[-1,1]$.

atan(*Arg*)
: Renvoie l'arc tangente de *Arg* dans l'intervalle $[-\pi/2,\pi/2]$ radians.

atan2(*x, y*)
Renvoie l'arc tangente de y/x dans l'intervalle $[-\pi,\pi]$ radians. x et y doivent être non nuls.

ceil(*Arg*)
Renvoie le plus petit entier supérieur ou égal à *Arg*.

cos(*Arg*)
Renvoie le cosinus de *Arg* (*Arg* est exprimé en radians).

cosh(*Arg*)
Renvoie le cosinus hyperbolique de *Arg*. Si le résultat provoque un dépassement de capacité, une erreur est générée.

double(*Arg*)
Si *Arg* est une valeur en virgule flottante, renvoie *Arg*, sinon convertit *Arg* en virgule flottante et renvoie la valeur obtenue.

exp(*Arg*)
Renvoie l'exponentielle de *Arg* définie comme e^{Arg}. Si le résultat provoque un dépassement de capacité, une erreur est générée.

floor(*Arg*)
Renvoie le plus grand entier inférieur ou égal à *Arg*.

fmod(*x, y*)
Renvoie le reste en virgule flottante de la division de x par y. Si y est nul une erreur est générée.

hypot(*x, y*)
Calcule la longueur de l'hypoténuse d'un triangle rectangle de côtés x et y (norme $\sqrt{x^2+y^2}$ du vecteur (x,y)).

int(*Arg*)
Si *Arg* est un entier, renvoie *Arg*, sinon convertit *Arg* à une valeur entière en le tronquant et renvoie la valeur obtenue. Attention, ce n'est pas la fonction partie entière: `expr int(-3.5)` renvoie -3 et non pas -4.

log(*Arg*)
Renvoie le logarithme népérien de *Arg*. *Arg* doit être strictement positif.

log10(*Arg*)
Renvoie le logarithme en base 10 de *Arg*. *Arg* doit être strictement positif.

pow(*x, y*)
Calcule la valeur x^y de x élevé à la puissance y. Si x est négatif alors y doit être entier.

rand()
Renvoie un nombre aléatoire dans l'intervalle [0,1[(1 exclu). Le germe de cette fonction aléatoire est pris à l'horloge interne de la machine ou peut être fixé au moyen de la fonction **srand**.

round(*Arg*)
Si *Arg* est entier, renvoie *Arg*, sinon arrondit *Arg* à une valeur entière et renvoie cette valeur. Par exemple,

```
expr round(3.4999)
```

```
expr round(3.5)
```

renvoient respectivement 3 et 4.

sin(*Arg***)**
Renvoie le sinus de *Arg* (*Arg* est exprimé en radians).

sinh(*Arg***)**
Renvoie le sinus hyperbolique de *Arg*. Si le résultat provoque un dépassement de capacité, une erreur est générée.

sqrt(*Arg***)**
Renvoie la racine carrée de *Arg*. *Arg* doit être négatif ou nul.

srand(*Arg***)**
L'argument *Arg* doit être entier et sera utilisé pour servir de germe à la fonction **rand()** qui engendre des nombres aléatoires. La valeur de retour est le premier nombre aléatoire engendré à partir de cette valeur.

tan(*Arg***)**
Renvoie la tangente de *Arg* (*Arg* est exprimé en radians).

tanh(*Arg***)**
Renvoie la tangente hyperbolique de *Arg*.

Types, capacité et précision

Tous les calculs internes mettant en jeu des valeurs entières sont effectués avec le type *long* du langage C ; ceux qui mettent en jeu des nombres en virgule flottante sont faits avec le type *double*. Lors de la conversion d'une chaîne à une valeur en virgule flottante les dépassements d'exposant sont détectés et provoquent une erreur. Pour la conversion de chaînes en valeurs entières, la détection des dépassements arithmétiques dépend du comportement de certaines routines de la bibliothèque C locale, elle n'est donc pas fiable. Dans tous les cas, les dépassements ou les valeurs trop faibles dans les calculs intermédiaires en valeurs entières ne sont pas non plus détectés de manière fiable. Dans le cas des calculs en virgule flottante, ils sont détectés au même degré qu'ils le sont au niveau du matériel ce qui est en général suffisamment fiable.

Les conversions entre les représentations internes des entiers, nombres en virgule flottante et chaînes se font automatiquement en fonction des besoins. De manière générale, les calculs sont menés en nombres entiers tant qu'un nombre en virgule flottante n'est pas rencontré; à partir de là ils se poursuivront en virgule flottante. Par exemple,

```
expr 5 / 4
```

renvoie 1, tandis que les expressions

```
expr 5 / 4.0
expr 5 / ( [string length "abcd"] + 0.0 )
```

renvoient toutes les deux 1,25. Les valeurs calculées en virgule flottante sont toujours renvoyées avec un point décimal ou bien avec une lettre **e** d'exposant pour qu'on ne les confonde pas avec des valeurs entières. Par exemple,

```
expr 20.0/5.0
```

renvoie 4,0 et non pas simplement 4.

Opérations sur les chaînes

Des chaînes peuvent être utilisées comme valeurs pour les opérandes des opérateurs de comparaison bien que l'interpréteur essaie de faire les comparaisons en nombres entiers ou en virgule flottante chaque fois qu'il le peut. La seule exception est le cas des opérateurs **eq** et **ne**. Si l'un des opérandes est une chaîne et l'autre un nombre, l'opérande numérique est reconverti en chaîne en utilisant le spécificateur **%d** de la fonction C *sprintf* pour les entiers et le spécificateur **%g** pour les nombres en virgule flottante.

Étant donnée la propension de Tcl à traiter les valeurs comme des nombres chaque fois que c'est possible, il n'est pas conseillé d'utiliser des opérateurs comme == lorsque l'on souhaite réellement une comparaison de chaînes et que les valeurs des opérandes sont arbitraires ; il vaut mieux dans ce cas se servir des opérateurs **eq** et **ne**, ou bien invoquer la commande **string** directement.

Optimisation

Pour accélérer les calculs et limiter les besoins de stockage des nombres, il est recommandé de placer les expressions entre accolades surtout si ces expressions impliquent des substitutions de commandes ou des interpolations de variables. C'est ainsi que le générateur de code compilé de Tcl fournira le code le plus efficace.

Il faut savoir, d'autre part, que les expressions subissent en fait une double substitution : une première fois par l'interpréteur Tcl lorsqu'il rencontre l'instruction **expr** suivie de ses arguments, puis une deuxième fois par la commande **expr** elle-même. L'exemple suivant illustre bien ce phénomène :

```
set a 3
set b {$a + 2}
expr $b*4
```

Le résultat produit est curieusement 11 et non pas un multiple de 4 comme pourrait le laisser croire l'expression `$b*4`. La deuxième ligne n'attribue pas à la variable b la valeur 5 ($= 3 + 2$) mais en fait la valeur littérale `$a + 2`. Dans la troisième ligne, l'interpréteur Tcl substitue cette valeur de la variable b et la commande **expr** devient alors

```
expr $a + 2*4
```

La commande **expr** opère alors la substitution de `$a` et calcule $3 + 2 * 4 = 11$.

fblocked

La commande **fblocked** teste si la dernière opération d'entrée a épuisé toutes les données disponibles.

Syntaxe

fblocked *Canal*

Description

La commande **fblocked** renvoie la valeur 1 si la dernière opération d'entrée effectuée sur le canal désigné par l'argument *Canal* a fourni moins d'informations que ce qui était requis faute de données suffisantes ou parce qu'il n'y a plus de données susceptibles d'être recueillies en entrée.

Par exemple, si la commande **gets** est invoquée alors qu'il n'y a que quelques caractères disponibles et pas de symbole de fin de ligne, elle renvoie une chaîne vide et un appel à la commande **fblocked** renverra la valeur 1.

fconfigure

La commande **fconfigure** fixe et lit les options de réglage d'un canal.

Syntaxe

fconfigure *Canal*
fconfigure *Canal Nom*
fconfigure *Canal Nom Valeur?Nom Valeur...?*

Description

La commande **fconfigure** permet de récupérer ou de fixer les options pour les canaux. L'argument *Canal* est l'identificateur du canal dont on veut manipuler les options. Si aucun des arguments *Nom* ou *Valeur* n'est fourni, la commande renvoie une liste contenant une alternance de noms et de valeurs des diverses options. Si l'argument *Nom* est fourni mais pas *Valeur* alors la commande renvoie la valeur correspondant à ce nom. Si une ou plusieurs paires de type *Nom / Valeur* sont spécifiées, la commande a pour effet d'attribuer ces valeurs aux noms correspondants.

Les options qui suivent sont supportées par tous les types de canaux mais certains canaux, en particulier les canaux de connexion, peuvent avoir des options spécifiques supplémentaires (cf. la commande **socket** à la page 166).

-blocking *Booléen*
: Cette option indique si les opérations d'entrée/sortie sur le canal peuvent bloquer le processus pour une durée indéterminée. La valeur passée dans l'argument *Booléen* doit être une valeur booléenne valide (*true* ou 1 pour vrai, *false* ou 0 pour faux).

  Habituellement les canaux sont en mode non-bloquant. Si un canal est placé en mode non-bloquant les commandes **gets**, **read**, **puts**, **flush** et **close** s'en trouveront affectées. On se reportera à la documentation des commandes en question pour plus de précision. Pour que le mode non-bloquant fonctionne proprement l'application Tcl doit avoir installé une boucle d'événements au moyen de la commande **vwait** la plupart du temps ou, dans du code C, par invocation de la commande **Tcl_DoOneEvent**.

-buffering *Valeur*
: Si la valeur attribuée à l'argument *Valeur* est *full* alors le système des entrées et sorties remplira le tampon jusqu'à ce qu'il soit plein ou qu'une commande **flush** soit invoquée. Si la valeur de *Valeur* est *line* le tampon sera automatiquement vidé à chaque caractère de fin de ligne. La valeur par défaut est *full* sauf pour les canaux qui connectent à des périphériques de type terminal ; dans ce cas la valeur initiale est *line*. Si la valeur attribuée à l'argument *Valeur* est *none* le tampon sera vidé à la suite de chaque opération de sortie.

Les canaux *stdin* et *stdout* sont initialement réglés sur *line* et le canal *stderr* sur *none*.

-buffersize *Taille*

L'argument *Taille* doit être un nombre entier dont la valeur représente la taille en octets du tampon alloué au canal pour stocker les opérations d'entrée et sortie. La valeur doit être comprise entre dix et un million.

-encoding *Nom*

Cette option permet de spécifier l'encodage du canal afin que les données puissent être correctement converties depuis et vers Unicode qui est l'encodage utilisé en interne par Tcl. Voir les encodages disponibles dans le tableau 3 à la page 47.

Si le fichier contient des données purement binaires (une image JPEG par exemple) l'encodage du canal devrait être fixé à la valeur *binary*. Dans ce cas, Tcl ne traduira pas les données mais les lira et les écrira comme des données brutes. La commande Tcl **binary** (à ne pas confondre avec la valeur d'encodage *binary* qui vient d'être mentionnée) est utile pour la manipulation de données binaires (cf. p. 23).

L'encodage par défaut pour un canal nouvellement ouvert est le même que l'encodage système (cf. la commande **encoding** à la page 46).

-eofchar *Caractère*

-eofchar *CarEntrée CarSortie*

Cette option supporte les systèmes de fichiers MS-DOS qui utilisent le caractère Control-z (de code hexadécimal `\x1a`) comme marqueur de fin de fichier. Si l'argument *Caractère* n'est pas la chaîne vide, c'est lui qui signalera les fins de fichier dans le flot d'entrée. Pour les sorties, le caractère de fin de fichier est produit lorsque le canal est refermé. Si l'argument *Caractère* est la chaîne vide alors il n'y a pas de marqueur spécifique de fin de fichier.

Pour les canaux ouverts en lecture et en écriture, une liste de deux éléments *CarEntrée CarSortie* permet de déclarer des symboles de fin de fichier différents pour les entrées et pour les sorties. Lorsque l'on récupère la valeur du symbole de fin de fichier pour un canal ouvert en lecture et en écriture, c'est toujours un couple de deux valeurs qui est renvoyé même si elles sont identiques.

La valeur par défaut est la chaîne vide dans tous les cas sauf pour Windows où c'est la valeur Control-z (`\x1a` en hexadécimal) pour la lecture et la chaîne vide pour les opérations d'écriture.

-translation *Mode*

-translation {*modeEntrée modeSortie*}

Dans les scripts Tcl, les fins de lignes sont toujours marquées en utilisant un unique caractère de saut de ligne (*newline*, noté symboliquement `\n`). Néanmoins dans les fichiers et les périphériques les fins de lignes peuvent être marquées différemment selon les plates-formes et les systèmes d'exploitation : sous Unix, ce sont les sauts de ligne (dits *linefeed* ou *lf*), sous MacOS ce sont les retours-chariots (*cr*) et sous Windows les couples de retours-chariots et sauts de ligne (*crlf*).

Pendant les opérations d'entrée avec des commandes telles que **gets** et **read**,

Tcl convertit automatiquement les fins de lignes qu'il reçoit en caractères de saut de ligne (lf). Pendant les opérations de sortie avec une commande telle que **puts**, Tcl traduit ses sauts de lignes internes en la représentation attendue normalement sur le système sur lequel il se trouve. Le mode de traduction par défaut s'appelle *auto* et gère automatiquement les principaux cas qui ont été mentionnés. Mais si l'on veut forcer d'autres types de traduction de caractères de fin de ligne, on utilisera l'option **-translation**.
Comme avec l'option **-translation**, la valeur associée à l'option est un terme unique pour les canaux ouverts seulement en lecture ou en écriture et une liste de deux termes pour les canaux ouverts à la fois en lecture et en écriture, d'où les deux formes indiquées pour la syntaxe de cette option. On peut en fait utiliser la première forme même pour des canaux en lecture/écriture : dans ce cas le caractère sera valable pour les deux types d'opérations.
Les termes autorisés pour spécifier l'option **-translation** sont les suivants :

auto
: Pour les opération d'entrée, le mode *auto* peut traiter n'importe quel type de symbole de fin de lignes (lf, cr ou crlf) qui seront tous convertis en lf. Il peut même y avoir plusieurs sortes mélangées dans un même fichier.
Pour les opération de sortie, le mode *auto* choisit le mode de représentation approprié en fonction du contexte : pour des connexions de réseaux (*sockets*) ce sera crlf pour toutes les plates-formes, sous Unix ce sera lf, sous MacOS ce sera cr et sous les diverses versions de Windows ce sera crlf.
auto est la valeur par défaut de l'option **-translation** aussi bien en entrée qu'en sortie.

binary
: Aucune conversion des symboles ne sera effectuée avec cette option.

cr
: L'option cr (abréviation de *carriage-return*, retour-chariot) est utilisée typiquement sur les Macintosh. En entrée la traduction se fait de cr à lf et en sortie de lf à cr.

lf
: L'option lf (abréviation de *linefeed*, saut de ligne) est utilisée typiquement sur les plates-formes Unix. Aucune traduction n'est nécessaire puisque lf est aussi la représentation utilisée en interne par Tcl.

crlf
: L'option crlf est utilisée typiquement sur les plates-formes Windows. En entrée la traduction se fait de crlf à lf et en sortie de lf à crlf.

Options de configuration du port série

Si l'argument *Canal* se réfère à un port série de nouvelles options de configuration ont été définies, depuis la version 8.4 de Tcl, sur les systèmes Unix et Windows

avec une interface série POSIX :

-mode	**-queue**	**-ttystatus**
-handshake	**-sysbuffer**	**-xchar**
-lasterror	**-timeout**	
-pollinterval	**-ttycontrol**	

On se reportera à la documentation de Tcl 8.4 pour une description détaillée de ces options.

Signaux et erreurs

De nombreuses erreurs sont susceptibles de se produire lors des opérations de lecture sur le port série ou du traitement d'événements en arrière-plan : le périphérique externe peut avoir été éteint, les données contenir du bruit, les tampons être saturés, les réglages être erronés etc. Il est donc important, de manière générale, d'exécuter les opérations concernant le port série dans une instruction **catch**. En cas d'erreur Tcl renvoie une erreur générale d'accès de fichier. La commande `fconfigure -lasterror` peut alors aider à localiser le problème. On se reportera à la documentation de Tcl 8.4 qui comporte une liste décrivant les signaux du port série (dans le standard RS-232) et les codes d'erreurs qui sont susceptibles de se produire pendant les opérations de lecture.

fcopy

La commande **fcopy** effectue des copies de données d'un canal à l'autre.

Syntaxe

fcopy *CanalEntrée CanalSortie*?**-size** *Taille??*?**-command** *CmdRappel?*

Description

La commande **fcopy** copie les données d'un canal d'entrées/sorties, *CanalEntrée* à un autre, *CanalSortie*. La commande s'efforce de réguler le système de tampons de Tcl afin d'éviter de faire des copies inutiles ou de stocker trop de données en mémoire.

Elle transfère les données depuis le canal d'entrée *CanalEntrée* jusqu'à ce que la fin de fichier soit atteinte ou bien qu'un certain nombre d'octets correspondant à la taille du tampon ait été traité. L'option **-size** sert précisément à fixer la taille du tampon : si elle n'est pas précisée alors la copie ira jusqu'à la fin du fichier. Toutes les données lues depuis le canal *CanalEntrée* sont copiées vers le canal *CanalSortie*.

En l'absence de l'option **-command**, la commande **fcopy** bloque le processus jusqu'à ce que l'opération s'achève et renvoie le nombre d'octets qui ont été copiés.

L'option **-command** permet à **fcopy** de travailler en tâche de fond. Dans ce cas la commande retourne immédiatement et la commande nommée *CmdRappel* est invoquée plus tard lorsque la copie s'achève. Cette fonction *CmdRappel* sera invoquée avec un ou deux arguments supplémentaires indiquant combien d'octets ont été copiés vers *CanalSortie*. Si une erreur s'est produite pendant l'exécution en arrière-plan le second argument devra représenter le message d'erreur associé.

Pour une copie en tâche de fond, il n'est en fait pas nécessaire de mettre les canaux d'entrée et de sortie en mode non-bloquant car la commande **fcopy** s'en charge automatiquement. En revanche, si le script est exécuté dans tclsh il faut installer une boucle d'événements en utilisant la commande **vwait**. Avec wish ce n'est pas nécessaire car il a sa propre boucle.

Par ailleurs quelques précautions sont à observer avec les copies en tâche de fond :

- il ne faut surtout pas exécuter d'autres opérations sur les canaux *CanalEntrée* ou *CanalSortie* pendant qu'une copie s'exécute en arrière-plan : si jamais une opération venait à fermer l'un de ces canaux, la copie en cours s'interromprait et la fonction ne serait pas appelée.
- il faut désactiver d'éventuels gestionnaires (*handlers*) créés avec la commande **fileevent** afin qu'ils ne risquent pas d'interférer avec la copie. Toute tentative se solderait par une erreur indiquant que le canal est occupé (*channel busy*).

Les symboles de fin de ligne sont traduits aussi bien pour *CanalEntrée* que pour *CanalSortie* suivant les réglages effectués avec la commande **fconfigure -translation** (voir la documentation à la page **??**). Cette traduction des symboles de fin de ligne

peut faire que le nombre d'octets lus sur *CanalEntrée* peut être différent de celui écrit sur *CanalSortie*. Seul le nombre d'octets écrits sur le canal de sortie sera renvoyé, soit comme valeur de retour de la commande **fcopy** elle-même en cas de copie synchrone, soit comme second argument de la fonction de rappel pour une copie asynchrone comme expliqué ci-dessus.

Exemple

L'exemple qui suit montre comment la fonction de rappel, installée par l'option **-command**, reçoit le nombre d'octets transférés. Il utilise la commande **vwait** pour installer une boucle d'attente d'événements.

```
proc Cleanup {in out bytes {error {}}} {
    global total
    set total $bytes
    close $in
    close $out
    if {[string length $error] != 0} {
 error occurred during the copy
   }
}
set in [open $file1]
set out [socket $server $port]
fcopy $in $out -command [list Cleanup $in $out]
vwait total
```

file

La commande **file** permet d'interagir avec le système de fichiers et de manipuler les divers attributs des fichiers. Certains de ces attributs peuvent être spécifiques à un système d'exploitation et ne pas exister avec d'autres.

Syntaxe

file *Sous-commande NomFichier?Arg Arg...?*

Description

L'argument *NomFichier* désigne le nom du fichier. Les conventions concernant les noms de fichiers ont été expliquées en détail au chapitre *Les fichiers* : en particulier, si le nom du fichier commence par un tilde ~, une substitution du tilde est opérée avant l'exécution de la commande.

Le type d'action à effectuer sur le nom du fichier est déterminé par la sous-commande. Les sous-commandes disponibles sont les suivantes [2] :

file atime *NomFichier?Temps?*
Retourne une valeur décimale indiquant le moment où s'est produit le dernier accès à ce fichier si l'argument *Temps* n'est pas spécifié, ou bien dans le cas contraire sert à fixer l'instant d'accès à cette valeur. Le temps est mesuré en nombre de secondes à partir d'une certaine date [3]. Si cet attribut n'est pas accessible, comme c'est le cas sous Windows, une erreur est générée: il vaudra mieux inclure cette commande dans une instruction **catch**.

file attributes *NomFichier*
file attributes *NomFichier?option?*
file attributes*NomFichier?option valeur option valeur...?*
La sous-commande **attributes** renvoie une série d'attributs concernant un fichier ou permet de fixer leur valeur suivant celle des trois syntaxes qui est utilisée. Les attributs en question varient en fonction du système d'exploitation. Sous la première forme, la sous-commande renvoie la liste de toutes les options disponibles sur une plate-forme particulière ainsi que leurs valeurs. La seconde forme renvoie la valeur de l'option spécifiée en argument tandis que la troisième permet de fixer la valeur d'une ou plusieurs options.

- sous Unix, les options **-owner** et **-group** permettent de manipuler le propriétaire et le groupe auxquels appartient le fichier : la valeur passée peut être le numéro d'identification ou bien le nom ; la valeur renvoyée sera toujours le nom du propriétaire ou du groupe. Avec l'option **-permissions** on peut manipuler les droits d'accès du fichier selon le système octal utilisé par la commande Unix chmod : pour fixer les valeurs, il est possible

2. Il est toujours possible d'abréger le nom d'une sous-commande pour peu que l'abréviation ne donne lieu à aucune ambiguïté.
3. En général le 1er janvier 1970. Sous MacOS, le 1er janvier 1904.

d'utiliser, outre le système octal, les notations symboliques acceptées par chmod telles que `u+s,go-wx` ou bien `rwxr-xr--`.

– sous MacOS, les options **-creator** et **-type** concernent respectivement le créateur et le type du fichier. L'option **-hidden** régit la visibilité du fichier. L'option **-readonly** (lecture seule) régit les droits d'écriture du fichier en cas de partage de fichier (ceci ne fonctione que si le partage de fichiers est activé sur l'ensemble du système au moyen du tableau de bord correspondant). Par exemple la commande suivante rend le fichier essai.tex invisible :

```
file attributes "HD:essai.tex" -hidden 1
```

– sous Windows, l'option **-archive**, **-hidden**, **-system** permettent de renvoyer ou de fixer la valeur des attributs *archive*, *visibilité* et *système* d'un fichier. L'option **-readonly** régit les droits d'accès en écriture du fichier. L'option **-longname** développe chaque élément d'un chemin d'accès sous sa forme longue tandis que **-shortname** effectue l'opération inverse et ramène chaque élément à sa forme courte de type "8.3". Ces deux attributs ne peuvent pas être fixés.

file channels *?Motif?*

Si l'argument *Motif* n'est pas spécifié, la commande renvoie la liste de tous les canaux ouverts dans l'interpréteur. Sinon ne sont indiqués que les canaux dont le nom correspond au *motif* selon les mêmes règles que celles qui régissent la commande **string match** (cf. p. 174).

filecopy?**-force**??**--**? *Source Cible*

file copy?**-force**??**--**? *Source ?Source...? Cible*

Utilisée sous la première forme, cette commande exécute une copie du fichier ou du répertoire source sous le nom spécifié par l'argument *Cible*. Lorsque le répertoire cible existe déjà, on utilise alors la seconde forme : tous les fichiers source seront copiés dans ce répertoire ; si un répertoire est indiqué comme source, ce sont alors tous les fichiers contenus dans ce répertoire qui sont copiés récursivement dans le répertoire cible.

S'il existe déjà des fichiers du même nom dans le répertoire de destination, ils ne seront pas écrasés à moins que l'option **-force** n'ait été précisée. Il n'est pas permis cependant d'écraser un répertoire non vide, d'écraser un fichier par répertoire ou bien un répertoire par un fichier : cela provoque une erreur de la part de l'interpréteur. Comme toujours, le double tiret **--** indique la fin des options éventuelles.

file delete?**-force**??**--**? *NomChemin ?NomChemin...?*

Supprime tous les fichiers ou répertoires spécifiés en arguments. On utilise l'option **-force** pour forcer la suppression de répertoires non vides.

file dirname *NomFichier*

Renvoie une chaîne composée de tous les éléments du chemin d'accès *NomFichier* à l'exception du dernier, ce qui dans le cas d'un chemin complet revient à donner le chemin du répertoire contenant le fichier. Dans le cas où l'argument désigne un chemin relatif constitué d'un seul élément, c'est un simple

point qui est renvoyé (ou bien un deux-points dans le cas de Macintosh). Dans le cas où il désigne un répertoire racine, c'est ce répertoire qui est renvoyé. Voici quelques exemples :

Commande	*Résultat*
file dirname /usr/lib/fichier	/usr/lib
file dirname /root	/
fle dirname	/home
file dirname c:/	c:/ *(sous Windows)*

Sous Unix, le cas du tilde est traité de manière particulière :
appliquée au nom de fichier `~/src/foo.c` la commande renverra `~/src` mais, si on l'applique à un tilde tout seul, celui-ci sera d'abord remplacé par le répertoire qu'il représente (par exemple /home) et c'est avec ce dernier que la commande sera exécutée.

file executable *NomFichier*
Renvoie 1 si le fichier est exécutable pour l'utilisateur courant et 0 sinon.

file exists *NomFichier*
Renvoie 1 si le fichier existe et, dans le cas des systèmes Unix, si l'utilisateur a le droit d'accès en lecture au répertoire qui contient ce fichier. Autrement la valeur renvoyée sera 0.

file extension *NomFichier*
Renvoie tous les caractères du nom du fichier situés après le dernier point du dernier élément (le point lui-même est inclus) : cela revient en principe à renvoyer l'extension éventuelle du nom du fichier. S'il n'y a pas d'extension, la commande renvoie la chaîne vide.

file isdirectory *Nom*
Renvoie 1 si l'argument *Nom* désigne un répertoire, 0 sinon.

file isfile *Nom*
Renvoie 1 si l'argument *Nom* désigne un fichier, 0 sinon.

file join *NomFichier?Nom...?*
Cette commande sert à reconstituer un chemin d'accès à partir d'éléments séparés : elle combine les éléments en les joignant au moyen du séparateur de chemin d'accès correspondant à la plate-forme sur laquelle on se trouve (la barre oblique / sous Unix, la contre-oblique \ sous Windows, le deux-points sous MacOS). Ceci facilite la portabilité des scripts puisqu'un chemin d'accès aura nécessairement la syntaxe correcte s'il est construit avec cette commande.
Les chemins relatifs donnent lieu à un traitement particulier : si un élément est relatif (c'est-à-dire commence par un séparateur), il sera combiné à celui qui précède seulement s'il s'agit réellement d'un chemin, autrement dit s'il contient un séparateur. Autrement les éléments qui précèdent sont purement et simplement ignorés. Le tableau 5 rassemble diverses situations pour mieux comprendre le comportement de cette commande : les résultats obtenus sous MacOS diffèrent de ceux que l'on obtient sous Unix car les conventions concernant les chemins de fichiers ne sont pas les mêmes sur ces deux

TAB. 5 – *La commande* file join

Sous Unix	
file join a	a
file join a b/	a/b
file join /a b	/a/b
file join a b	a/b
file join a /b	/b
file join a// b	a/b
file join a b c d	a/b/c/d
file join /a b c d	/a/b/c/d
file join a/b c d	a/b/c/d
file join a/b c/d	a/b/c/d
file join a b /c d	/c/d
file join a/b/ /c d	/c/d
file join a b c d/	a/b/c/d
file join a b c /d	/d
Sous MacOS	
file join a	:a
file join a:	a:
file join a b	:a:b
file join a b:	b:
file join :a b	:a:b
file join a b	a:b
file join a :b	:a:b
file join a:: b	a::b
file join a:: :b	a::b
file join a:: ::b	a:::b
file join a b c d	:a:b:c:d
file join :a b c d	:a:b:c:d
file join a:b c d	a:b:c:d
file join a:b c:d	c:d
file join a b :c d	:a:b:c:d
file join a:b: :c d	a:b:c:d
file join a b c d:	d:
file join a b c :d	:a:b:c:d

systèmes (sur Macintosh un double séparateur :: indique que l'on remonte de deux répertoires).

file lstat *Nom NomVar*

Cette commande est analogue à la commande **file stat** expliquée ci-dessous. La différence concerne les fichiers qui sont des liens symboliques : l'information renvoyée dans la variable *NomVar* concernera le lien et non pas le fichier auquel il fait référence. Sur les systèmes qui ne supportent pas les liens sym-

boliques, la sous-commande **lstat** se comporte comme **stat**.

file mkdir *Répertoire ?Répertoire...?*
La sous-commande **mkdir** (abréviation de *make directory*) permet de créer chacun des répertoires spécifiés. Cette commande créera tous les répertoires intermédiaires qui figurent entre le nom du répertoire et le répertoire racine si ceux-ci n'existent pas. Si le répertoire à créer existe déjà, rien ne se passe ; en revanche, si l'on tente de créer un répertoire alors qu'un fichier du même nom existe déjà à cet endroit alors une erreur est renvoyée par l'interpréteur et l'exécution de la commande s'interrompt.

file mtime *NomFichier ?Temps?*
Retourne une valeur décimale indiquant la date de dernière modification du fichier *NomFichier* si l'argument *Temps* n'est pas spécifié, ou bien permet de fixer la date de modification de ce fichier à la valeur *Temps*. Le temps est mesuré en nombre de secondes à partir d'une certaine date comme expliqué ci-dessus avec la sous-commande **atime**. Si cet attribut n'est pas accessible ou si le fichier n'existe pas, une erreur est générée.

file nativename *NomFichier*
Renvoie le nom de fichier spécifique à la plate-forme sur laquelle on se trouve. On utilisera cette commande lorsque l'on doit passer un nom de fichier à une fonction qui n'existe que sur une plate-forme particulière comme **exec** sous Windows ou **AppleScript** sous MacOS.

file owned *NomFichier*
Renvoie 1 si le fichier *NomFichier* appartient à l'utilisateur courant, 0 sinon.

file pathtype *NomFichier*
Renvoie l'un des termes *absolute*, *relative*, *volumerelative*. Ils correspondent respectivement à l'une des trois situations suivantes :

- l'argument *NomFichier* se réfère à un fichier particulier d'un volume particulier ;
- il se réfère à un fichier relatif au répertoire de travail courant ;
- il se réfère à un fichier relatif au répertoire de travail courant sur un volume spécifié ou bien à un fichier spécifique sur le volume courant.

file readable *NomFichier*
Renvoie 1 si l'utilisateur courant a les droits en lecture sur le fichier *NomFichier*, 0 sinon.

file readlink *NomLien*
Renvoie le nom du fichier sur lequel pointe un lien symbolique *NomLien*. Si *NomLien* n'est pas un lien ou s'il ne peut être lu, l'interpéteur génère une erreur.

file rename?-**force**??**--**? *Source Cible*
file rename?-**force**??**--**? *Source ?Source...? Cible*
Cette commande peut être utilisée selon l'une des deux syntaxes ci-dessus. Utilisée sous la première forme, elle renomme le fichier ou le répertoire *Source* sous le nom *Cible* et d'autre part déplace le fichier si la *Cible* désigne un nom dans un autre répertoire.

Si l'argument *Cible* désigne un répertoire existant, on utilisera la seconde forme : dans ce cas, chaque fichier ou répertoire *Source* est déplacé dans le répertoire désigné par *Cible*. Des fichiers déjà existants ne seront écrasés que si l'option **-force** est spécifiée. Comme avec la sous-commande **copy**, toute tentative d'écraser un répertoire non vide, d'écraser un fichier par un répertoire ou bien un répertoire par un fichier générera une erreur.

file rootname *NomFichier*
Renvoie tous les caractères de *NomFichier* jusqu'au dernier point éventuel figurant dans le dernier élément de ce nom (à l'exclusion de ce point). S'il n'y a pas de point, la commande renvoie l'argument *NomFichier* entier.

file size *NomFichier*
Renvoie une valeur décimale correspondant à la taille du fichier *NomFichier* en octets. Si le fichier n'existe pas ou si l'information n'est pas disponible, une erreur est générée.

file split *NomFichier*
Cette commande réalise l'opération inverse de **file join**. Elle permet de décomposer un nom de fichier en la liste de ses éléments constitutifs. Les séparateurs sont omis à moins qu'il n'y ait un risque d'ambiguïté : sous Unix la commande

```
file split /foo/~bar/baz
```

renverra

```
/  foo  ./~bar  baz
```

afin que le tilde ne risque pas par la suite d'être interprété comme un chemin relatif.

file stat *Nom NomVar*
Invoque l'appel au noyau **stat** sur le fichier *NomFichier* et utilise la variable désignée par l'argument *NomVar* pour recueillir l'information renvoyée par le noyau. Cette commande renvoie une chaîne vide. *NomVar* est une variable de type tableau avec les clés suivantes : *atime*, *ctime*, *dev*, *gid*, *ino*, *mode*, *mtime*, *nlink*, *size*, *type*, *uid*. Chaque élément (à l'exception de *type*) est une chaîne décimale représentant la valeur du champ correspondant dans la structure *stat* renvoyée. On se reportera à la documentation de la commande Unix stat (en tapant *man stat* depuis une fenêtre de terminal par exemple) pour plus de précisions sur la signification de ces champs. L'élément *type* indique le type sous la même forme que celle qui est produite par la commande **file type**.

file tail *NomFichier*
Renvoie tous les caractères de l'argument *NomFichier* figurant après le dernier séparateur de fichier. S'il n'y en a pas, l'argument *NomFichier* est renvoyé inchangé.

file type *NomFichier*
Renvoie le type du fichier *NomFichier*. Ce sera l'un des termes suivants : **file**, **directory**, **caractèreSpecial**, **blockSpecial**, **fifo**, **link**, ou **socket**.

file volume
Renvoie une liste des volumes montés sur le système. Sous MacOS on obtient

la liste à la fois des volumes locaux et des volumes distants. Sous Windows, seuls les volumes locaux sont listés. Sous Unix la question ne se pose pas puisque tous les volumes sont montés localement.

file writable *NomFichier*

Renvoie 1 si l'utilisateur courant a les droits en écriture sur le fichier *NomFichier*, 0 sinon.

fileevent

La commande **fileevent** exécute un script lorsqu'un canal devient disponible pour la lecture ou pour l'écriture.

Syntaxe

fileevent *Canal* **readable** *?Script?*
fileevent *Canal* **writable** *?Script?*

Description

Cette commande est utilisée pour créer des gestionnaires d'événements de fichiers, c'est-à-dire pour établir un lien entre un canal et un script de telle sorte que le script soit exécuté au moment où le canal en question deviendra accessible soit en lecture, soit en écriture. Ces gestionnaires sont utilisés couramment pour permettre de recevoir des données depuis un autre processus en fonction de certains événements de telle sorte que l'application puisse continuer de fonctionner en attendant que les données commencent à arriver. Si le canal se trouve en mode non-bloquant et que des commandes **gets** ou **read** sont invoquées alors qu'aucune donnée en entrée n'est encore disponible, le processus se bloquera temporairement et ne pourra pas traiter les événements qui pourraient intervenir en attendant ; l'utilisateur aura l'impression que le processus est gelé. Si l'on utilise la commande **fileevent**, le processus sera en mesure de détecter quand une donnée devient disponible, ce qui lui permet de n'invoquer les commandes **gets** ou **read** qu'à ce moment-là.

L'argument *Canal* de la commande **fileevent** est un identificateur de canal qui aura été obtenu auparavant comme valeur de retour d'une commande **open** ou **socket** utilisées pour ouvrir un canal ou établir une connexion de réseau. Si l'argument *Script* est spécifié, alors **fileevent** crée un nouveau gestionnaire d'événements : le script *Script* sera évalué dès que le canal deviendra accessible en lecture ou en écriture suivant le second argument de la commande. Dans ce cas, **fileevent** renverra une chaîne vide. Les options d'accès **readable** et **writable** permettent de créer des gestionnaires d'événements qui sont indépendants et peuvent être ensuite détruits séparément. En revanche il ne peut y avoir plus d'un gestionnaire **readable** et un gestionnaire **writable** pour un fichier à un moment donné dans un interpréteur donné. Si une commande **fileevent** est invoquée alors qu'un gestionnaire existe déjà, le nouveau script remplace l'ancien.

Si l'argument *Script* n'est pas spécifié, **fileevent** renvoie le script courant pour le canal *Canal* ou une chaîne vide s'il n'y en a pas. En revanche si le script est expressément défini au moyen d'une chaîne vide, le gestionnaire d'événements sera détruit. Un gestionnaire est aussi détruit automatiquement lorsque son canal est refermé ou que l'interpréteur lui-même est détruit.

Un canal est encore considéré lisible s'il reste des données non lues disponibles sur le périphérique sous-jacent ou bien également dans un tampon d'entrée sauf dans le cas où la dernière tentative de lecture a été faite avec la commande **gets** et que celle-ci n'a pas trouvé dans le tampon de quoi terminer une ligne. Ceci permet de lire un fichier ligne par ligne en mode non-bloquant en utilisant des événements.

Un canal est encore considéré lisible aussi si le fichier ou le périphérique sous-jacents contiennent un symbole de fin de fichier ou bien une condition d'erreur. Il est important que le script teste ces situations et les traite en conséquence sinon il peut se créer des boucles infinies dans lesquelles le script n'arrive à lire aucune donnée, retourne et puis est immédiatement invoqué à nouveau.

Un canal est considéré disponible en écriture si au moins un octet de donnée peut être écrit sur le fichier ou le périphérique sous-jacents sans bloquer, ou bien si une condition d'erreur y est présente.

En principe les entrées/sorties pilotées par des événements fonctionnent mieux pour les canaux qui ont été placés en mode non-bloquant au moyen de la commande **fconfigure**. En mode non-bloquant, la commande **puts** bloque le processus si on lui transmet plus de données que ce que le fichier ou le périphérique sous-jacents peuvent accepter et une commande **gets** ou **read** bloquera si l'on tente de lire plus de données qu'il n'est disponible : aucun événement ne sera traité tant qu'une commande bloquera. Au contraire en mode non-boquant les commandes **puts**, **read** et **gets** ne bloqueront jamais.

Le script pour un événement de fichier est exécuté au niveau global dans l'interpréteur dans lequel la commande **fileevent** a été invoquée. Si une erreur se produit, le gestionnaire d'événement est détruit, ceci afin d'éviter qu'un code défectueux ne provoque des boucles infinies.

flush

La commande **flush** permet de vider le tampon de sortie d'un canal.

Syntaxe

flush *Canal*

Description

L'argument *Canal* doit être un numéro d'identification de canal valide obtenu au moyen d'une commande telle que **open** ou **socket** et il faut que ce canal ait été ouvert en écriture. La commande **flush** vide le contenu d'un tampon contenant les sorties produites par le canal.

Si le canal est en mode bloquant, la commande ne retournera pas tant que tout le contenu du tampon n'aura pas été vidé. Si le canal est en mode non-bloquant, la commande pourra retourner avant que tout le contenu n'ait été vidé : le restant sera traité en arrière-plan.

for

La commande **for** installe une boucle itérative.

Syntaxe

for *Initialisation Test Continuation Instructions*

Description

La commande **for** permet de construire des structures répétitives. Les arguments *Initialisation*, *Continuation* et *Instructions* doivent être des instructions Tcl valides et l'argument *Test* est une expression. Le processus est le suivant pour exécuter une instruction **for** : elle commence par invoquer l'interpréteur Tcl pour qu'il exécute l'instruction incluse dans l'argument *Initialisation* à la suite de quoi il évalue l'argument *Test* comme une expression : si le résultat est non nul, l'interpréteur Tcl exécute les instructions contenues dans la partie *Instructions* et qui peut être toute une suite d'instructions Tcl c'est-à-dire un script, que l'on désigne souvent comme le corps de la boucle. Dès que ce script est terminé, l'interpréteur exécute l'instruction représentée par l'argument *Continuation* et recommence le corps de la boucle.

Ce processus itératif s'achèvera lorsque l'expression *Test* renverra une valeur nulle. Cet enchaînement répétitif peut cependant, lorsque c'est nécessaire, être interrompu ou modifié :

- si une commande **continue** est rencontrée à un moment donné dans le corps de la boucle, l'interpréteur ne termine pas cette itération de la boucle et passe directement à l'argument *Continuation* et puis à l'itération suivante ;
- si une commande **break** est rencontrée dans le corps de la boucle, la commande **for** s'achève immédiatement et l'on retourne à l'instruction qui l'avait appelée.

La commande **for** renvoie toujours une chaîne vide. Il est prudent d'entourer les quatre arguments au moyen d'accolades d'une part afin d'améliorer la lisibilité du script, d'autre part afin d'éviter que des substitutions de variables ou des interpolations ne se produisent trop tôt. Si l'expression *Test* est évaluée trop tôt, elle ne le sera plus ensuite, ce qui risque de la rendre inutile puisqu'elle ne verrait plus si une certaine variable a changé de valeur : il peut ainsi en résulter des boucles infinies. L'exemple suivant est symptomatique :

```
for {set x 0} {$x < 10} {incr x} {
    puts "x a la valeur $x"
}
```

Tel qu'il est écrit il est tout à fait correct. Si jamais l'instruction `$x < 10` n'était pas entourée d'accolades, l'interpréteur Tcl, lors de sa première lecture de la commande **for**, remplacerait $x par sa valeur, à savoir 0. La condition de sortie de la boucle serait alors `0 < 10` qui sera toujours vérifiée, d'où la boucle infinie.

foreach

La commande **foreach** permet d'itérer des instructions en parcourant les éléments d'une ou de plusieurs listes.

Syntaxe

foreach *NomVar Liste Instructions*
foreach *ListeVar1 Liste1 ?ListeVar2 Liste2...? Instructions*

Description

La commande **foreach** implémente une boucle dont les valeurs de certaines variables peuvent être prises parmi les éléments d'une ou plusieurs listes. Dans la première forme, on spécifie le nom d'une variable, *NomVar*, et une liste, *Liste* : cette liste est une liste de valeurs qui seront assignées successivement à la variable. Pour chacune de ces valeurs, les instructions *Instructions* constituent un script qui sera exécuté en remplaçant le contenu de la variable par la valeur en question.

Dans la deuxième forme, il peut y avoir plusieurs variables et plusieurs listes de valeurs à affecter à chacune de ces variables. L'avancement dans ces listes se fait parallèlement pour chaque variable : à chaque itération, chaque valeur reçoit la valeur suivante dans la liste qui lui est associée. Si jamais une liste ne contient pas assez de valeurs, la variable correspondante recevra alors comme valeur une chaîne vide. L'itération se termine donc lorsque la liste la plus longue aura été épuisée.

On peut modifier le déroulement répétitif d'une commande **foreach** en utilisant les commandes **break** et **continue** qui ont exactement la même signification qu'avec la commande **for** (cf. p. 80). La valeur de retour de **foreach** est une chaîne vide.

format

La commande **format** permet de formater une chaîne de caractères selon un schéma spécifié.

Syntaxe

format *Schéma ?Arg Arg...?*

Introduction

Cette commande fonctionne d'une manière très semblable à la fonction *sprintf* du langage C (norme ANSI). L'argument *Schéma* permet de décrire, au moyen de spécificateurs qui seront expliqués plus loin, un format modèle auquel les arguments qui suivent devront se conformer : le résultat sera une chaîne formatée.

Le *Schéma* contient toutes les instructions de formatage. La commande **format** l'analyse en le lisant de gauche à droite : tant qu'elle lit des caractères autres que le signe pourcentage (%), elle se contente d'ajouter ces caractères à la chaîne qu'elle est en train de construire. En revanche un signe pourcentage indique une spécification de format. Tous les spécificateurs de format commencent par un % et les quelques signes (en nombre variable) qui le suivent prennent une signification particulière. Tous ces spécificateurs s'appliquent à un des arguments *Arg* de la commande : le premier spécificateur s'applique au premier argument *Arg* et ainsi de suite jusqu'à épuisement de tous les spécificateurs (il doit y avoir au moins autant d'arguments *Arg* qu'il y a de spécificateurs).

Description des spécificateurs

Chaque spécificateur est suivi de caractères ayant une signification ou une fonction particulière et que l'on appelle champs. Il existe six types différents de champs dont certains peuvent être omis :

- un spécificateur de position ;
- un ensemble de drapeaux ;
- un nombre désignant une largeur minimale ;
- un nombre désignant une précision ;
- un modificateur de longueur ;
- un caractère de conversion

Seul le caractère de conversion (sixième champ) est obligatoire ; tous les autres champs sont facultatifs. Les champs doivent être indiqués dans l'ordre qui vient d'être mentionné. C'est la commande **format** qui se charge de les reconnaître et de les interpréter.

Spécificateur de position

Ce premier champ est de loin le plus compliqué. Si le symbole % est suivi d'un nombre entier n puis du symbole dollar $ (comme par exemple `%2$d`) alors l'argument à convertir ne sera pas pris à partir du suivant dans la liste des arguments *Arg* de la commande **format** mais à partir du n-ième argument de cette liste. Un spécificateur comportant cette syntaxe est dit *spécificateur de position* et les conventions exigent que si un tel spécificateur est utilisé dans le schéma *Schéma* alors tous les autres spécificateurs doivent aussi être de ce type.

Si jamais d'autre part le symbole $*$ est utilisé dans une spécification (voir ci-dessous les troisième et quatrième champs), alors les arguments successifs sont utilisés en partant justement de celui qui est désigné par le nombre n.

Ensemble de drapeaux

Le deuxième champ peut contenir n'importe lequel des symboles suivants et dans n'importe quel ordre :

- **-** permet de spécifier, pour l'argument converti, une justification à gauche à l'intérieur de son champ (les nombres sont usuellement justifiés à droite, précédés d'espaces si nécessaire) ;
- \+ impose qu'un nombre soit toujours précédé d'un signe (même positif) ;
- ˽ fait en sorte qu'une espace (ici représentée par ˽ par convention) soit ajoutée au début du nombre s'il ne comporte pas de signe + ou -.
- **0** fait en sorte que le nombre soit précédé de zéros si nécessaire.
- \# permet de préciser des options d'affichage. Pour des conversions utilisant **o** ou **O** (cf. les caractères de conversion ci-dessous), le **0** garantit que le nombre octal commencera par 0. Pour des conversions utilisant **x** ou **X**, le **0** garantit que le nombre hexadécimal commencera par 0x ou 0X respectivement (sauf le nombre 0). Pour des conversions de nombres en virgule flottante (**e**, **E**, **f**, **g** et **G**), il garantit que le résultat contiendra toujours un point décimal. Pour **g** et **G**, il spécifie que des zéros comme dernières décimales ne doivent pas être supprimés.

Largeur minimale

Ce troisième paramètre de spécification permet d'imposer une largeur minimale pour la chaîne formatée. On l'utilise pour obtenir des alignements en colonnes. Si l'argument converti comporte moins de caractères que la largeur requise, il sera complété en général par des espaces (à moins qu'un modificateur ne soit spécifié. Voir ci-dessous). Si le symbole $*$ est utilisé pour désigner la largeur minimale au lieu d'un nombre alors c'est l'argument *Arg* suivant dans la série des arguments de la commande **format** qui sera utilisé : il faut bien sûr que cet argument désigne un nombre entier.

Nombre désignant une précision

Le quatrième champ d'un schéma de spécification de format sert à indiquer une précision : il consiste en un point suivi d'un nombre. Ce nombre est interprété

différemment en fonction du type de conversion opéré (voir ci-dessous les caractères de conversion) :

- avec les caractères de conversion **e**, **E** et **f**, il désigne le nombre de décimales ;
- avec les caractères de conversion **g** et **G**, il indique le nombre total de chiffres aussi bien à gauche qu'à droite du point décimal (toutefois les dernières décimales sont ignorées si elles sont nulles à moins que le drapeau 0 n'ait été déclaré) ;
- pour des conversions d'entiers, il désigne un nombre minimal de chiffres à produire. Au besoin des zéros seront insérés en préfixe ;
- avec le caractère de conversion **s**, il indique le nombre *maximal* de caractères à retenir ;
- si le symbole $*$ est utilisé au lieu d'un nombre alors c'est l'argument *Arg* suivant dans la série des arguments de la commande **format** qui sera utilisé pour indiquer la précision : il faut bien sûr que cet argument désigne un nombre entier.

Modificateur de longueur

Il est indiqué par l'une des lettres **h** ou **l**. La lettre **h** spécifie que la valeur à convertir devra être tronquée à un nombre 16-bit : cela correspond en interne à une conversion au type *short*. La lettre **l** (valide avec la fonction *sprintf* du langage C) est ignorée : elle correspond à une conversion en interne au type double ce qui est justement toujours le cas avec Tcl.

Caractère de conversion

Les caractères suivants sont reconnus pour indiquer le type de conversion à effectuer sur l'argument correspondant :

d convertit un entier en une chaîne décimale signée.

u convertit un entier en une chaîne décimale non signée.

i convertit en une chaîne décimale signée ; l'entier peut se présenter sous formes soit décimale, soit octale (avec un 0 initial), soit hexadécimale (commençant par 0x).

o convertit un entier en une chaîne octale non signée.

x convertit un entier en une chaîne hexadécimale non signée, utilisant des minuscules pour les hexadécimaux (abcdef).

X convertit un entier en une chaîne hexadécimale non signée, utilisant des majuscules pour les hexadécimaux (ABCDEF).

c convertit un entier en le caractère Unicode qu'il représente.

s n'effectue pas de conversion mais sert simplement à insérer la chaîne définie par l'argument *Arg* correspondant.

f convertit un nombre en virgule flottante en une chaîne décimale signée de la forme $xx.yyy$ où le nombre de y est déterminé par la précision (par défaut 6). Une précision de 0 revient à ne garder que la partie située avant le point.

e ou **E** convertissent un nombre en virgule flottante en notation scientifique selon le modèle $x.yyye\pm zz$ où le nombre de y est déterminé par la précision (par défaut 6). Une précision de 0 revient à ne garder que la partie située avant le point. Avec le caractère **E** le nombre sera noté $x.yyyE\pm zz$.

g ou **G** convertissent des nombres en virgule flottante : si l'exposant est inférieur à -4 ou bien supérieur ou égal à la précision, cela correspond à **%e** ou **%E**. Autrement cela correspond à **%f**. Les dernières décimales nulles sont omises.

% permet d'insérer un signe de pourcentage ordinaire.

Pour que les conversions s'opèrent, il faut évidemment que l'argument *Arg* correspondant soit bien un nombre du type qui est attendu. On notera que les spécificateurs **%p** et **%n** de la fonction *sprintf* ne sont pas supportés.

gets

La commande **gets** lit des lignes en entrée depuis un canal ouvert.

Syntaxe

gets *Canal?NomVar?*

Description

Cette commande permet de lire depuis le canal désigné par l'argument *Canal* la prochaine ligne et renvoie l'intégralité de la ligne jusqu'au caractère de fin de ligne non compris. Elle ignore les caractères de fin de ligne. Si l'argument *NomVar* est omis, la ligne est la valeur de retour de la commande **gets**. Sinon la ligne est placée dans cette variable *NomVar* et la valeur de retour est le nombre de caractères ainsi renvoyés.

Trois cas particuliers peuvent se produire :

- si la fin du fichier lu en entrée est rencontrée avant d'avoir atteint un symbole de fin de ligne, la commande lit jusqu'à la fin du fichier.
- si le canal est en mode non-bloquant et qu'il n'y a pas une ligne entière disponible en entrée, la commande renvoie une chaîne vide et laisse les caractères en entrée intacts.
- si un nom de variable a été spécifié et qu'une chaîne vide est renvoyée, le décompte du nombre de caractères sera mis à -1.

On peut se servir des commandes **eof** et **fblocked** (cf. p. 48 et 63 respectivement) pour distinguer ces trois cas particuliers.

glob

La commande **glob** renvoie les noms de fichiers correspondant à un certain motif descriptif.

Syntaxe

glob *?Commutateurs? Motif?Motif...?*

Description

Cette commande permet de dresser des listes de fichiers dont les noms correspondent à un ou plusieurs motifs dits parfois « motifs globalisants ».

Syntaxe des motifs

La syntaxe servant à décrire de manière abstraite des noms de fichiers utilise quelques métacaractères dont voici la signification :

?
: Le point d'interrogation sert à désigner n'importe quel caractère unique.

*****
: L'astérisque sert à désigner n'importe quelle série de zéro ou plus caractères. Il s'agit d'un quantificateur.

[*Caractères*]
: Les paires de crochets entourant des caractères permettent de désigner n'importe lequel des caractères ainsi spécifiés. Ces caractères peuvent être indiqués au moyen d'intervalles du type x-y qui représentent en fait tous les caractères dont le code ASCII est compris inclusivement entre celui de x et celui de y.

\x
: Représente simplement le caractère x lui-même. Cela n'a d'intérêt que pour les caractères qui autrement ont valeur de métacaractères.

{ a, b, . . . }
: Représente n'importe laquelle des chaînes a, b etc. Ici a, b représentent des chaînes et non pas des caractères uniques.

Il ne faut pas confondre ces métacaractères de motifs globalisants avec les métacaractères des expressions régulières: l'astérisque en particulier n'a pas du tout la même signification dans les deux cas.

Comme avec le *shell* csh, un point « . » au début d'un nom de fichier ou bien après une barre oblique / doit correspondre exactement.

Si le premier caractère d'un motif est un tilde ~ il fait référence, comme avec le shell, au répertoire personnel (home) de l'utilisateur dont le nom suit le tilde. Si aucun nom ne suit le tilde mais que c'est une barre oblique alors le tilde est remplacé par la valeur de la variable d'environnement HOME. Ces remarques concernent évidemment les systèmes de type Unix.

À la différence de csh, les motifs de la commande **glob** ne trient pas la liste des fichiers qu'elle renvoie. On peut toujours utiliser la commande Tcl **lsort** si on veut trier la liste. Par ailleurs, la commande **glob** ne renvoie que les noms de fichiers existants alors qu'avec csh l'existence des fichiers n'est pas vérifiée à moins que le motif ne contienne un symbole ?, ∗, [ou].

Options de la commande glob

La commande **glob** admet un certain nombre d'options :

-directory *répertoire*
: Recherche des fichiers correspondant aux motifs en partant du répertoire *répertoire* spécifié. Cette option permet de rechercher dans des répertoires dont le nom contient certains des métacaractères mentionnés ci-dessus. Les options **-directory** et **-path** s'excluent mutuellement.

-join
: Les arguments suivants sont traités comme un motif unique obtenu en les joignant au moyen de séparateurs à la manière de la commande **file join**.

-nocomplain
: Normalement une erreur est générée si jamais la liste en retour est vide. Cette option empêche l'erreur.

-path *PrefixeChemin*
: Recherche des fichiers dont le nom commence par *PrefixeChemin* et dont la suite correspond aux motifs spécifiés. Cette option permet de rechercher des fichiers dont le nom est analogue à un fichier donné même si leur nom contient certains des métacaractères mentionnés ci-dessus. Les options **-directory** et **-path** s'excluent mutuellement.

-types *typeListe*
: Ne retient que les fichiers ou les répertoires correspondant à l'argument *typeListe* qui est une liste dont les éléments peuvent prendre deux formes :
  - La première forme est analogue à l'option *-type* de la commande Unix find. Il s'agit de trouver les fichiers dont le type est spécifié au moyen d'une ou plusieurs des lettres suivantes : *b* (fichier bloc spécial), *c* (fichier caractère spécial), *d* (répertoire), *f* (fichier ordinaire), *l* (lien symbolique), *p* (*pipe* nommé), ou *s* (*socket*),.
  - la seconde forme spécifie des types qui doivent tous correspondre. Les valeurs possibles sont : *r*, *w*, *x* pour les droits d'accès, *readonly* et *hidden*. Sous MacOS on peut également indiquer le type ou le créateur : il faut écrire *macintosh type* XXXX ou *macintosh creator* XXXX, où XXXX est la valeur souhaitée (quatre lettres au maximum).

  Ces deux formes peuvent être mélangées. Par exemple

  ```
  -types d f r w
  ```

  rechercherait tous les fichiers ordinaires *ou* les répertoires dont les conditions d'accès sont r *et* w.

--
Un double tiret marque la fin des options.

Les noms de ces options peuvent être abrégés si cela ne présente pas d'ambiguïté.

Portabilité

La commande **glob** admet uniquement des styles de noms natifs, pas de noms dans le style des noms de réseau.

Windows
Pour des noms de type UNC sur Windows, les composantes *servername* et *sharename* du chemin ne doivent pas contenir les métacaractères ?, ∗, [ou]. Sur WindowsNT, si le motif est de la forme

```
~NomUtilisateur@Domaine
```

il fait référence au répertoire personnel de l'utilisateur *NomUtilisateur* dont l'information de compte se trouve sur le serveur de domaine *Domaine* spécifié. Sinon, l'information de compte est obtenue à partir de la machine locale. Sur Windows 95 et 98 la commande **glob** accepte des motifs tels que .../ ou ..../ pour désigner des répertoires parents.

Macintosh
Lorsque les options **-dir**, **-join** ou **-path** sont utilisées, la commande **glob** utilise le deux-points « : » au lieu de la barre oblique / comme séparateur pour les versions du système antérieures à MacOSX.

global

La commande **global** permet de donner accès aux variables globales.

Syntaxe

global *NomVar?NomVar...?*

Description

Cette commande n'a de sens qu'à l'intérieur d'une procédure définie par la commande **proc**. Tous ses arguments sont des noms de variables qui sont déclarées comme globales et non comme locales et limitées à la portée de la procédure exécutée. Les variables globales sont définies dans l'espace global d'interprétation : en les déclarant au moyen de la commande **global** dans une procédure, on les rend accessibles depuis l'intérieur de cette procédure. Cette déclaration peut se placer à n'importe quel endroit dans la procédure (mais bien entendu avant que la variable ainsi déclarée soit utilisée pour la première fois).

history

La commande **history** permet de manipuler l'historique des commandes exécutées par l'interpréteur Tcl.

Syntaxe

history *?Sous-commande? ?Arg Arg...?*

Description

Cette commande permet d'obtenir des informations et d'effectuer certaines opérations sur l'historique des commandes, c'est-à-dire la liste des commandes les plus récentes exécutées par l'interpréteur Tcl. Ces commandes sont désignées sous le vocable *événements*. Pour spécifier, dans les commandes qui suivent, un tel événement plusieurs formes sont admises :

- un nombre positif fait référence à la commande ayant ce numéro, les événements étant numérotés à partir de 1 dans l'historique. Un nombre négatif désigne un événement relativement à l'événement courant : 0 est l'événement courant, -1 l'événement précédent, etc.
- une chaîne désigne le plus récent événement dont le nom correspond à cette chaîne, soit parce que les premiers caractères coïncident, soit parce qu'il y a correspondance au sens de la commande **string match** (cf. p. **??**).

La commande **history** est utilisée avec l'une des sous-commandes suivantes :

history
: Synonyme de **history info**.

history add *Commande ?exec?*
: Ajoute en tant que nouvel événement l'argument *Commande* à l'historique. Si l'argument **exec** est spécifié alors la commande sera aussi exécutée et son résultat renvoyé. Si *exec* n'est pas spécifié la valeur de retour est une chaîne vide.

history change *Valeur ?Commande?*
: Remplace la valeur enregistrée pour un événement par *Valeur*. L'argument *Commande* désigne l'événement à remplacer qui est, par défaut, l'événement courant. Cette commande peut être utilisée dans des procédures qui installent de nouvelles formes d'historiques et entendent remplacer l'événement courant (celui qui invoque une substitution) par la commande créée par cette substitution. La valeur de retour est une chaîne vide.

history clear
: Efface l'historique. La taille courante de l'historique est retenue. Les numéros d'événements sont remis à un.

history event *?Commande?*
Renvoie la valeur de l'événement *Commande*. Par défaut c'est l'événement -1, c'est-à-dire celui qui précède l'événement courant.

history info *?combien?*
Renvoie une chaîne formatée indiquant le numéro et le contenu de chaque événement de l'historique, hormis l'événement courant. L'argument *combien* est un nombre entier n : s'il est spécifié seulement les n derniers événements sont renvoyés.

history keep *?nombre?*
Cette commande permet de changer la taille de l'historique, c'est-à-dire le nombre maximal de commandes à conserver. L'argument indique la nouvelle taille. Initialement, cette valeur est fixée à 20. Si l'argument n'est pas spécifié, la valeur de la taille courante est renvoyée.

history nextid
Renvoie le numéro du prochain événement qui sera enregistré dans l'historique.

history redo *?Commande?*
Réexécute la commande désignée par l'argument *Commande* et renvoie son résultat. Par défaut *Commande* est l'événement -1, c'est-à-dire celui qui précède l'événement courant.

Cette commande provoque une révision de l'historique. Les versions de Tcl antérieures à 8.0 avaient un mécanisme complexe pour la révision de l'historique. Désormais les anciennes sous-commandes **substitute** et **words** n'existent plus. Lorsque **redo** est invoquée l'événement le plus récent est modifié afin d'éliminer la commande **history** et de la remplacer par son résultat. Si l'on veut réexécuter un événement avec **redo** sans modifier l'historique, il faudra utiliser la sous-commande **event** pour récupérer un événement puis la sous-commande **add** pour l'ajouter à l'historique et l'exécuter.

if

La commande **if** sert à l'exécution conditionnelle des instructions.

Syntaxe

if *Expr1***?then?** *Script1* **elseif** *Expr2***?then?** *Script2* **elseif**...**?else?** *?ScriptN?*

Description

La commande *if* évalue l'expression *Expr1* : celle-ci doit retourner une valeur booléenne c'est-à-dire soit une valeur numérique entière où 0 est synonyme de *faux* et n'importe quelle autre valeur est synonyme de *true*, soit l'un des mots *true* ou *yes* pour signifier *true* et *false* ou *no* pour signifier *faux*. Si la valeur renvoyée par l'évaluation de *Expr1* est *true*, le script *Script1* est exécuté par l'interpréteur Tcl ; sinon l'expression *Expr2* à son tour est évaluée et si elle renvoie la valeur *true*, le script *Script2* est exécuté et ainsi de suite. Si aucune expression n'est évaluée comme vraie alors c'est finalement le script *ScriptN* qui est exécuté.

Les termes **then** et **else** sont entièrement facultatifs : ils permettent d'améliorer la lisibilité d'une instruction **if** mais ne sont pas requis.

Il peut y avoir autant de clauses **elseif** qu'on le souhaite (éventuellement aucune). Le dernier argument *ScriptN* peut être omis : dans ce cas le **else** est omis lui aussi.

L'instruction **if** renvoie la valeur résultant de celui des scripts qui aura finalement été exécuté ou bien une chaîne vide si aucun n'est exécuté.

incr

La commande **incr** permet d'incrémenter ou de décrémenter la valeur d'une variable.

Syntaxe

incr *NomVar ?Incrément?*

Description

Si la variable *NomVar* contient une valeur entière, la commande **incr** permet de l'augmenter d'une unité si aucun incrément n'est précisé ou bien de la valeur de l'argument *Incrément*. Ce dernier peut être positif ou négatif. La nouvelle valeur est contenue dans la variable *NomVar* et est aussi renvoyée par la fonction.

info

La commande **info** fournit des informations concernant l'état de l'interpréteur Tcl.

Syntaxe

info *Sous-commande ?Arg Arg...?*

Description

Cette commande permet d'obtenir des informations concernant diverses variables internes de Tcl en fonction de la sous-commande qui lui est associée. Les sous-commandes disponibles sont les suivantes :

info args *Nomproc*
Renvoie une liste contenant les noms des arguments de la procédure *Nomproc* dans l'ordre où ils sont déclarés dans la définition de cette procédure.

info body *Nomproc*
Renvoie le corps de la procédure *Nomproc*.

info cmdcount
Renvoie le nombre total des commandes qui ont été invoquées dans cet interpréteur Tcl.

info commands *?Motif?*
Si l'argument *Motif* n'est pas spécifié, la commande renvoie une liste de toutes les commandes dans l'espace de noms courant, aussi bien les commandes de base que les procédures définies au moyen de la commande **proc**.
Si l'argument *Motif* est spécifié, seulement les noms de commandes correspondant au motif seront listés. Les règles de correspondance avec le motif sont les mêmes que pour la commande **string match** (cf. p. 174). Le motif *Motif* peut être un nom qualifié (comme `abc::def*`), c'est-à-dire désigner un espace de noms particulier et donc comporter une suite de termes séparés par un double deux-points (**::**). Il peut même se terminer par des métacaractères afin de désigner d'un seul coup toute une série de commandes dans cet espace de noms. Les commandes renvoyées seront elles aussi désignées par leur nom qualifié (cf. la commande **namespace** à la page 124).

info complete *Commande*
Renvoie 1 si *Commande* est une commande Tcl complète, autrement dit ne contient pas d'erreurs de syntaxe telles que paires de délimiteurs non refermées etc. Sinon la valeur de retour est 0. Cette commande est utile pour des saisies interactives de commandes : elles permettent qu'un utilisateur rentre une série de commandes sur plusieurs lignes et d'attendre que la commande soit bien complète avant de l'envoyer à l'interpréteur Tcl. Si l'utilisateur frappe la touche RETOUR et que la commande n'est pas encore complète, l'application peut ainsi retarder son exécution.

info default *Nomproc Arg NomVar*
Nomproc doit être le nom d'une procédure Tcl et l'argument *Arg* doit désigner le nom de l'un des arguments de cette procédure. Si cet argument *Arg* n'a pas de valeur par défaut dans la définition de la procédure alors la valeur de retour sera 0. Autrement ce sera 1 et la valeur par défaut en question sera placée dans la variable de nom *NomVar* (qui sera créée si elle n'existe pas déjà).

info exists *NomVar*
Renvoie 1 si la variable *NomVar* existe dans le contexte courant (aussi bien en tant que variable globale ou locale) et s'est vu attribuer une valeur, 0 sinon.

info globals *?Motif?*
Si l'argument *Motif* n'est pas spécifié, la commande renvoie la liste de toutes les variables globales couramment définies. Si l'argument *Motif* est spécifié, seulement les noms de variables correpondant au motif seront listés. Les règles de correspondance avec le motif sont les mêmes que pour la commande **string match** (cf. p. 174).

info hostname
Renvoie le nom de l'ordinateur sur lequel cette instruction est invoquée.

info level *?Nombre?*
Si l'argument *Nombre* n'est pas spécifié, cette commande renvoie le niveau de la procédure appelante dans la pile ou bien 0 si cette commande est invoquée au niveau global.
Si l'argument *Nombre* est spécifié, le résultat est une liste constituée du nom et des arguments de l'appel de procédure correspondant au niveau *Nombre* de la pile. Un nombre *Nombre* strictement positif sélectionne un niveau particulier de la pile (1 désigne la procédure sommet, 2 la procédure appelée depuis 1, et ainsi de suite) ; un nombre négatif ou nul désigne un niveau relativement au niveau courant, autrement dit 0 correspond au niveau courant, -1 au niveau de la procédure appelante etc. Voir la commande **uplevel** à la page 198.

info library
Renvoie le nom du répertoire sur le système qui contient les bibliothèques Tcl standards. C'est en fait la valeur de la variable **tcl_library** dont le contenu peut être modifié à tout moment dans un script.

info loaded *?Interpréteur?*
Renvoie une liste décrivant tous les modules qui ont été chargés par le script au moyen de la commande **load** dans l'interpréteur éventuellement spécifié par l'argument *Interpréteur*. Chaque élément de cette liste est lui-même une liste de deux termes : le premier désigne le nom du fichier sur le système à partir duquel le module a été chargé, le second désigne le nom de ce module. Ces modules peuvent être des bibliothèques statiques ou partagées. Pour un module chargé statiquement, le nom de fichier sera vide.
Si l'argument *Interpréteur* est omis alors l'information renvoyée concerne tous les interpréteurs du processus en cours. Pour obtenir une liste des modules chargés dans l'interpréteur courant, on peut spécifier une chaîne vide `{}` pour l'argument *Interpréteur*.

info locals *?Motif?*
Si l'argument *Motif* n'est pas spécifié, la commande renvoie la liste de toutes les variables locales couramment définies, y compris celles qui sont les arguments de la procédure courante. Si l'argument *Motif* est spécifié, seulement les noms de variables correspondant au motif seront listés. Les règles de correspondance avec le motif sont les mêmes que pour la commande **string match** (cf. p. 174).

info nameofexecutable
Renvoie le chemin complet du fichier exécutable à partir duquel l'application courante a été lancée. En cas d'échec, Tcl renvoie une chaîne vide.

info patchlevel
Renvoie la valeur de la variable globale **tcl_patchLevel**.

info procs *?Motif?*
Si l'argument *Motif* n'est pas spécifié, la commande renvoie la liste de toutes les procédures définies dans l'espace de noms courant. Si l'argument *Motif* est spécifié, seulement les noms de procédures correspondant au motif seront listés. Les règles de correspondance avec le motif sont les mêmes que pour la commande **string match** (cf. p. 174).

info script *?NomFichier?*
Si un fichier de script Tcl est en cours d'évalution, cette commande renvoie le nom du fichier inclus en cours d'exécution. Si l'argument *NomFichier* est spécifié, la valeur de retour de cette commande sera modifiée pendant la durée de l'invocation active afin de renvoyer ce nom. Sinon la commande renvoie une chaîne vide.

info sharedlibextension
Renvoie l'extension utilisée sur une plate-forme pour désigner les bibliothèques partagées (par exemple **.so** sur Solaris, **.shlb** sur MacOS, **.dll** sur Windows). Si la plate-forme ne supporte pas ce type de bibliothèques, alors la valeur de retour est une chaîne vide.

info tclversion
Renvoie la valeur de la variable globale **tcl_version**.

info vars *?Motif?*
Si l'argument *Motif* n'est pas spécifié, la commande renvoie la liste de toutes les variables couramment visibles : cela comporte les variables locales et les variables globales rendues visibles. Si l'argument *Motif* est spécifié, seulement les noms de variables correspondant au motif seront listés. Les règles de correspondance avec le motif sont les mêmes que pour la commande **string match** (cf. p. 174). Le motif *Motif* peut être un nom qualifié (comme `abc::def*`), c'est-à-dire désigner un espace de noms particulier et donc comporter une suite de termes séparés par un double deux-points (**::**). Il peut même se terminer par des métacaractères afin de désigner d'un seul coup toute une série de commandes dans cet espace de noms. Les commandes renvoyées seront elles aussi désignées par leur nom qualifié (concernant les espaces de noms, voir la commande **namespace** à la page 124).

interp

La commande **interp** permet de créer et de manipuler des interpréteurs Tcl.

Syntaxe

interp *Option ?Arg Arg...?*

Description

Il est possible avec cette commande de créer simultanément plusieurs interpréteurs cohabitant avec l'interpréteur qui les a créés. L'interpréteur créateur est aussi appelé interpréteur maître et les nouveaux interpréteurs sont des interpréteurs esclaves. Un maître peut créer un nombre arbitraire d'esclaves et chaque esclave à son tour peut créer des interpréteurs dont il sera maître. Il peut ainsi y avoir toute une hiérarchie d'interpréteurs.

Chaque interpréteur est indépendant des autres : il a son propre espace de noms pour les commandes, les procédures, les variables globales. Un interpréteur maître peut établir des liens entre ses esclaves et lui-même par un mécanisme d'alias. Un alias est une commande d'un interpréteur esclave qui, lorsqu'elle est invoquée, provoque l'appel d'une commande dans l'interpréteur maître ou dans un des autres esclaves. Les seuls autres liens qu'il puisse y avoir entre les interpréteurs se font par les variables d'environnement (la variable **env**) qui sont normalement partagées entre tous les interpréteurs.

La commande **interp** introduit par ailleurs la notion d'interpréteurs sécurisés (*safe*). Un interpréteur sécurisé est un interpréteur maître dont les commandes internes ont été restreintes : certaines commandes ont été supprimées afin de pouvoir exécuter sans risque des scripts d'origine inconnue et douteuse qui pourraient contenir des instructions dangereuses ou des attaques contre le système. Par exemple, toutes les commandes qui permettent d'établir des canaux d'entrée/sortie sont inaccessibles dans un interpréteur sécurisé. En réalité les commandes ne sont pas retirées de l'interpréteur, elles sont seulement cachées de telle sorte que seuls des interpréteurs autorisés puissent y avoir accès (voir ci-dessous). Le mécanisme des alias peut servir aussi à créer des communications protégées entre un interpréteur esclave et son maître.

Les noms des interpréteurs sont relatifs à l'interpréteur dans lequel ils sont utilisés. Un nom d'interpréteur qualifié est une *liste* Tcl contenant un sous-ensemble de ses ancêtres dans la hiérarchie de l'interpréteur. Par exemple, si **a** est un esclave de l'interpréteur courant et a lui-même un esclave **a1** ayant un esclave **a11**, le nom qualifié de **a11** dans **a** est la liste **{a1 a11}**.

La commande **interp** accepte des noms d'interpréteur qualifiés comme argument. On peut toujours se référer à l'interpréteur dans lequel la commande est évaluée avec le symbole **{}** (une liste vide). Il existe deux restrictions motivées par des raisons de sécurité :

– il n'est pas possible de se référer au maître par son nom depuis un esclave,

sauf par le biais des alias ;
– il n'y a pas de nom global pour se référer à l'interpréteur initial créé par l'application.

Les sous-commandes

La commande **interp** permet de créer, manipuler ou détruire des interpréteurs et de partager entre eux des canaux. Les diverses actions sont obtenues grâce aux sous-commandes suivantes :

interp alias *InterpSource CommandeSource*
Renvoie une liste Tcl dont les éléments sont les arguments *CommandeCible* et *Arg* associés à l'alias nommé *CommandeSource*. Ce sont des valeurs qui ont été spécifiées au moment où l'alias a été créé. Il est possible que la commande source dans l'alias soit différente de *CommandeSource* si jamais elle a été renommée.

interp alias *InterpSource CommandeSource* {}
Détruit l'alias pour *CommandeSource* dans l'interpréteur maître désigné par *InterpSource*. L'argument *CommandeSource* se réfère au nom sous lequel l'alias a été créé ; si la commande source a été renommée, c'est la commande renommée qui est détruite.

interp alias *InterpSource CmdSource InterpCible CmdCible ?Arg Arg...?*
Cette commande crée un alias entre deux interpréteurs esclaves (voir ci-dessous la méthode **alias** applicable à une instance d'un interpréteur pour créer un alias entre un esclave et son maître). Chacun des esclaves peut se trouver n'importe où dans la hiérarchie des interpréteurs sous l'interpréteur qui a invoqué la commande.

Les arguments *InterpSource* et *CmdSource* identifient la source de l'alias : *InterpSource* est une liste Tcl dont les éléments sélectionnent un interpréteur particulier. Par exemple, **{a b}** correspond à un interpréteur **b**, esclave de l'interpréteur **a**, lui-même esclave de l'interpréteur appelant. Une liste vide {} désigne l'interpréteur qui invoque la commande. L'argument *CmdSource* indique le nom de la nouvelle commande qui sera créée dans l'interpréteur source.

Les arguments *InterpCible* et *CmdCible* désignent un interpréteur et une commande cibles et les arguments *Arg*, s'il y en a, désignent des arguments supplémentaires qui seront ajoutés devant d'éventuels arguments spécifiés dans l'invocation de *CmdSource*. *CmdCible* peut ne pas être défini au moment de l'appel ou bien exister déjà ; il n'est pas créé par cette commande.

L'alias fait en sorte que la commande cible soit invoquée dans l'interpréteur cible chaque fois que la commande source est invoquée dans l'interpréteur source.

interp aliases *?nomInterp?*
Cette commande renvoie une liste Tcl des noms de toutes les commandes

sources pour tous les alias définis dans l'interpréteur désigné par l'argument *nomInterp*.

interp create?-**safe**?**--**? *?nomInterp?*
Cette commande crée un interpréteur maître identifié par *nomInterp* ainsi qu'une nouvelle commande dite *commande esclave*. Le nom de cette commande esclave est le dernier terme de *nomInterp*. Le nouvel interpréteur esclave et la commande esclave sont créés dans l'interpréteur identifié par l'argument *nomInterp* duquel on retire le dernier terme. Par exemple, si l'argument *nomInterp* est **{a b c}** alors un nouvel interpréteur esclave et une commande esclave appelés **c** sont créés dans l'interpréteur identifié par le chemin **{a b}**. Si l'argument *nomInterp* est omis, Tcl crée un nom unique de la forme interp*n* où *n* est un nombre entier et l'utilise à la fois pour l'interpréteur et la commande esclaves.
Avec l'option **-safe** (ou si l'interpréteur maître est sécurisé), le nouvel interpréteur esclave sera sécurisé. Le double tiret **--** marque la fin des options en cas d'ambiguïté.
Le résultat de la commande est le nom du nouvel interpréteur. Le nom de l'interpéteur esclave ne doit pas exister déjà parmi les autres interpréteurs du même maître sinon une erreur est générée.

interp delete *?nomInterp...?*
Détruit zéro ou plus interpréteurs spécifiés par les arguments *nomInterp* et, pour chaque interpréteur, détruit aussi ses interpréteurs esclaves. Les commandes esclaves sont détruites également. Si un argument *nomInterp* désigne un interpréteur qui n'existe pas, une erreur est générée.

interp eval *nomInterp Arg?Arg...?*
Cette commande concatène tous les arguments *Arg* à la manière de la commande **concat** (cf. p. 41) puis évalue la chaîne résultante comme un script dans l'interpréteur désigné par *nomInterp*. Le résultat de cette évaluation est renvoyé à l'interpréteur appelant.

interp exists *nomInterp*
Renvoie 1 si l'interpréteur maître spécifié par *nomInterp* existe dans le maître, 0 sinon. Si *nomInterp* est omis, l'interpréteur appelant est utilisé.

interp expose *nomInterp NomCaché ?CommandeExposée?*
Révèle une commande cachée *NomCaché*, lui attribuant éventuellement le nouveau nom *CommandeExposée* (qui doit être un nom valide dans l'espace de noms global) dans l'interpréteur désigné par *nomInterp*. S'il existe déjà une commande de ce nom, l'instruction échouera.

interp hide *nomInterp CommandeExposée ?CommandeCachée?*
Cache, dans l'interpréteur désigné par *nomInterp*, la commande *CommandeExposée* en lui donnant le nom de la commande cachée *CommandeCachée*, ou en gardant son nom si l'argument *CommandeCachée* n'est pas spécifié. S'il existe déjà une commande cachée de ce nom, l'instruction échouera. Ni *CommandeExposée*, ni *CommandeCachée* ne peuvent contenir de qualificateurs d'espaces de noms sinon une erreur est générée. Les commandes à cacher

sont recherchées dans l'espace de noms global même si l'espace de noms courant est différent. Cela empêche que des interpréteurs esclaves ne trompent l'interpréteur maître afin de lui faire cacher une mauvaise commande en rendant l'espace de noms courant différent de l'espace global.

interp hidden *nomInterp*
Renvoie une liste des noms de toutes les commandes cachées dans l'interpréteur identifié par *nomInterp*.

interp invokehidden *nomInterp*?**-global**? *CommandeCachée ?Arg...?*
Invoque la commande cachée *CommandeCachée* avec les arguments fournis dans l'interpréteur désigné par *nomInterp*. Aucune substitution ou évaluation ne sont appliquées aux arguments. Si le drapeau **-global** est présent, la commande cachée est invoquée au niveau global dans l'interpréteur cible ; sinon elle est invoquée dans le cadre courant de la pile et peut accéder aux variables locales dans ce cadre et les cadres extérieurs.

interp issafe *?nomInterp?*
Renvoie 1 si l'interpréteur identifié par l'argument *nomInterp* est sécurisé, 0 sinon.

interp marktrusted *nomInterp*
Marque l'interpréteur identifié par l'argument *nomInterp* comme sûr (*trusted*). N'expose pas les commandes cachées. Cette commande peut être invoquée seulement à partir d'un interpréteur réputé sûr. La commande est sans effet si l'interpréteur identifié par *nomInterp* est déjà réputé sûr.

interp share *InterpSource Canal InterpDestination*
Fait du canal entrée/sortie identifié par *Canal* un canal partagé entre l'interpréteur identifié par *InterpSource* et l'interpréteur identifié par *InterpDestination*. Les deux interpréteurs ont les mêmes permissions sur le canal. Chacun des deux interpréteurs doit invoquer une commande **close** pour fermer le canal sous-jacent. Par ailleurs les canaux accessibles depuis un interpréteur sont automatiquement refermés lorsque cet interpréteur est détruit.

interp slaves *?nomInterp?*
Renvoie une liste des noms de tous les interpréteurs esclaves associés à l'interpréteur identifié par *nomInterp*. Si *nomInterp* est omis, l'interpréteur appelant est utilisé.

interp target *nomInterp Alias*
Renvoie une liste Tcl décrivant l'interpréteur cible pour un alias. L'alias est spécifié avec un nom d'interpréteur et un nom de commande source, exactement comme avec la commande **interp alias** vue ci-dessus. Le nom de l'interpréteur cible est renvoyé comme chemin d'interpréteur relativement à l'interpréteur invoquant. Si l'interpréteur cible pour l'alias est l'interpréteur invoquant alors une liste vide est renvoyée. Si l'interpréteur cible pour l'alias n'est pas l'interpréteur invoquant ou un de ses descendants alors une erreur est générée. La commande cible n'a pas à être définie au moment de cette invocation.

interp transfer *InterpSource Canal InterpDestination*
Rend le canal entrée/sortie identifié par *Canal* disponible dans l'interpréteur identifié par *InterpDestination* et indisponible dans l'interpréteur identifié par *InterpSource*.

Méthodes applicables aux interpréteurs esclaves

Pour chaque interpréteur maître créé au moyen de la commande **interp**, une nouvelle commande Tcl est créée dans l'interpréteur maître avec le nom de ce nouvel interpréteur. Cette commande peut être utilisée pour effectuer diverses opérations sur l'interpréteur. La forme générale est la suivante :

Esclave **Sous-Commande** *?Arg Arg...?*

Esclave est le nom de l'interpréteur (donc de la commande) et l'argument *Sous-Commande* et les arguments *Arg* déterminent le comportement exact de la commande. Ces sous-commandes peuvent être vues, pour reprendre la terminologie de la programmation orientée objets, comme des *méthodes* applicables à chaque *instance* d'un interpréteur esclave. Les sous-commandes admises sont les suivantes :

Esclave **aliases**
Cette commande renvoie une liste Tcl des noms de tous les alias dans l'interpréteur *Esclave*. Les noms renvoyés sont les valeurs *CommandeSource* utilisées à la création des alias et qui peuvent ne pas être les mêmes que les noms actuels des commandes si celles-ci ont été renommées.

Esclave **alias** *CommandeSource*
Renvoie une liste Tcl dont les éléments sont les arguments *CommandeCible* et *Arg* associés à l'alias nommé *CommandeSource*. Ce sont des valeurs qui ont été spécifiées au moment où l'alias a été créé. Il est possible que la commande source dans le maître soit différente de *CommandeSource* si jamais elle a été renommée.

Esclave **alias** *CommandeSource* **{}**
Détruit l'alias de *CommandeSource* dans l'interpréteur esclave. *CommandeSource* se réfère au nom sous lequel l'alias a été créé. Si la commande source a été renommée, c'est la commande renommée qui est détruite.

Esclave **alias** *CommandeSource CommandeCible ?Arg ..?*
Crée un alias de telle sorte que chaque fois que *CommandeSource* est invoquée dans l'interpréteur *Esclave*, *CommandeCible* est invoquée dans le maître. Les arguments *Arg* seront passés à *CommandeCible* comme arguments supplémentaires, devant d'autres arguments éventuels passés à l'invocation de *CommandeSource*.

Esclave **eval** *Arg ?Arg ..?*
Cette commande concatène tous les arguments *Arg* à la manière de la commande **concat** (cf. p. 41) puis évalue la chaîne résultante comme un script dans l'interpréteur *Esclave*. Le résultat de cette évaluation est renvoyé à l'interpréteur appelant.

Esclave **expose** *NomCaché?CommandeExposée?*
Révèle une commande cachée *NomCaché*, lui attribuant éventuellement le nouveau nom *CommandeExposée* (qui doit être un nom valide dans l'espace de noms global), dans l'interpréteur *Esclave*. S'il existe déjà une commande de ce nom, l'instruction échouera.

Esclave **hide** *CommandeExposée?CommandeCachée?*
Cache, dans l'interpréteur *Esclave*, la commande *CommandeExposée* en lui donnant le nom de la commande cachée *CommandeCachée*, ou en gardant son nom si l'argument *CommandeCachée* n'est pas spécifié. S'il existe déjà une commande cachée de ce nom, l'instruction échouera. Ni *CommandeExposée*, ni *CommandeCachée* ne peuvent contenir des qualificateurs d'espaces de noms sinon une erreur est générée. Les commandes à cacher sont recherchées dans l'espace de noms global même si l'espace de noms courant est différent.

Esclave **hidden**
Renvoie une liste des noms de toutes les commandes cachées dans l'interpréteur *Esclave*.

Esclave **invokehidden**?**-global**? *NomCaché?Arg ..?*
Invoque la commande cachée *CommandeCachée* avec les arguments fournis dans l'interpréteur *Esclave*. Aucune substitution ou évaluation n'est appliquée aux arguments. Si l'option **-global** est présente, la commande cachée est invoquée au niveau global dans l'interpréteur cible ; sinon elle est invoquée dans le cadre courant de la pile et peut accéder aux variables locales dans ce cadre et les cadres extérieurs.

Esclave **issafe**
Renvoie 1 si l'interpréteur est sécurisé, 0 sinon.

Esclave **marktrusted**
Marque l'interpréteur *Esclave* comme sûr (*trusted*). Cette commande n'expose pas les commandes cachées dans l'interpréteur maître. Elle peut être invoquée seulement à partir d'un interpréteur réputé sûr. La commande est sans effet si l'interpréteur esclave est déjà réputé sûr.

Interpréteurs sécurisés

Un interpréteur sécurisé est un interpréteur dans lequel certaines commandes et variables ont été retirées afin de permettre d'exécuter un script d'origine inconnue ou douteuse et de se protéger contre des attaques éventuelles contre l'application et le système en général. Par exemple, la commande **exec** est indisponible dans un interpréteur sécurisé car elle pourrait être utilisée pour lancer des instructions dévastatrices à l'encontre du système au moyen de sous-processus.

Les alias permettent néanmoins d'avoir un accès limité à certaines des fonctions désactivées après vérification de leurs arguments par l'interpréteur maître. Par exemple, la création de fichiers pourrait être autorisée dans certains répertoires particuliers spécifiés d'avance ou la commande **exec** pourrait être autorisée pour exécuter certains processus dont la liste est prédéterminée.

Un interpréteur sécurisé est créé en spécifiant **-safe** l'option dans la commande

interp create. La liste suivante indique quelles sont les commandes disponibles dans un interpréteur sécurisé:

after	eval	incr	namespace	split
append	expr	info	package	string
array	fblocked	interp	pid	subst
binary	fcopy	join	proc	switch
break	fileevent	lappend	puts	tell
case	flush	lindex	read	trace
catch	for	linsert	regexp	unset
clock	foreach	list	regsub	update
close	format	llength	rename	uplevel
concat	gets	lrange	return	upvar
continue	global	lreplace	scan	variable
eof	history	lsearch	seek	vwait
error	if	lsort	set	while

Les commandes suivantes sont cachées dans un interpréteur sécurisé:

cd	fconfigure	load	socket
exec	file	open	source
exit	glob	pwd	vwait

Ces commandes peuvent être recréées comme procédures Tcl ou comme alias, ou encore réexposées au moyen de la commande **interp expose** vue ci-dessus.

En outre, la variable **env** est indisponible dans un interpréteur sécurisé donc il ne pourra partager de variables d'environnement avec les autres interpréteurs. Les variables d'environnement présentent un risque potentiel d'atteinte à la sécurité puisqu'elles sont parfois utilisées pour stocker des informations sensibles.

Si des extensions ou des modules sont chargés par la commande **load** dans un interpréteur sécurisé, ils peuvent aussi restreindre eux-mêmes leurs fonctionnalités pour éliminer les commandes dangereuses (cf. p. 113) s'ils ont été programmés en conséquence.

Utilisation des alias

Le mécanisme des alias donne accès à des commandes cachées de façon à garantir qu'elles ne présentent pas de danger et que l'on puisse exécuter un script potentiellement dangereux dans un interpréteur sécurisé avec des alias dont la cible est dans un interpréteur réputé sûr (en général le maître). Le point délicat est de garantir que l'information passée par l'interpréteur esclave au maître ne soit pas évaluée ou interpolée dans le maître.

Lorsque la source d'un alias est invoquée dans l'interpréteur esclave, les substitutions habituelles de Tcl sont effectuées au moment de l'analyse de la commande. Ces substitutions ont lieu dans l'interpréteur source (c'est-à-dire l'interpréteur esclave sécurisé) comme c'est le cas pour toutes les autres commandes appelées dans

cet interpréteur. Les arguments de la commande source sont ajoutés aux arguments *arg* de *CommandeCible* déclarés à la création de l'alias : par exemple, si la commande source a la syntaxe suivante

```
CommandeSource arg1 arg2... argN
```

alors la nouvelle commande sera :

```
CommandeCible arg arg... arg arg1 arg2... argN
```

La commande *CommandeCible* est en principe une commande existant dans l'interpréteur cible sinon il y aura une erreur. Aucune substitution nouvelle n'est effectuée : la commande cible est évaluée directement sans repasser par un cycle d'interpolations dans l'interpréteur cible. Les arguments *CommandeCible* et *args* ont donc été interpolés au moment de l'analyse de la commande qui avait créé l'alias et les arguments *arg1* - *argN* au moment où la commande source de l'alias a été analysée dans l'interpréteur source.

Il est donc important lorsqu'on écrit la procédure *CommandeCible* que les arguments de cette commande ne soient ni substitués, ni évalués dans le cadre de la définition de la procédure.

Commandes cachées

Il arrive cependant que l'on ait besoin d'autoriser l'utilisation de certaines commandes qui ne sont en principe pas disponibles dans des interpréteurs sécurisés. La commande **interp** dispose d'un mécanisme pour cela : plutôt que de retirer complètement les commandes dangereuses, celles-ci sont en réalité cachées ce qui les rend indisponibles pour des scripts Tcl exécutés dans cet interpréteur. Néanmoins ces commandes peuvent être invoquées dans un parent considéré comme sûr de l'interpréteur sécurisé en utilisant la commande **interp invoke**. Les commandes cachées et les commandes exposées résident dans des espaces de noms séparés. Il est possible de définir une commande cachée et une commande exposée de même nom au sein d'un même interpréteur. Des commandes cachées d'un interpréteur esclave peuvent être invoquées dans le corps de définition de procédures appelées dans le maître au cours de l'invocation d'un alias. À titre d'exemple, supposons que l'on ait créé un alias pour la commande **source** dans un interpréteur esclave. Lorsque cet alias est invoqué dans l'interpréteur esclave, une procédure correspondante est appelée dans l'interpréteur maître : cette procédure pourrait consister à vérifier que le fichier à sourcer est un fichier autorisé et donc demander qu'il soit sourcé dans l'interpréteur esclave. Il y a ici deux commandes appelées **source** dans l'interpréteur esclave : l'alias et la commande cachée.

Les interpréteurs sécurisés ne sont pas autorisés à invoquer des commandes cachées à l'intérieur d'eux-mêmes ou de leurs descendants, ceci afin d'empêcher des interpréteurs esclaves d'avoir accès à leurs fonctionnalités cachées.

L'ensemble des commandes cachées dans un interpréteur peut être manipulé par un interpréteur sûr en utilisant les commandes **interp expose** ou bien **interp hide**. La première déplace une commande cachée dans l'ensemble des commandes exposées de l'interpréteur désigné par l'argument *nomInterp*, en la renommant au passage (si une commande de ce nom existe déjà dans l'interpréteur cible, il y aura

erreur). Inversement **interp hide** déplace une commande exposée dans l'ensemble des commandes cachées de l'interpréteur. Les interpréteurs sécurisés n'ont pas le droit de déplacer des commandes entre les commandes cachées et les commandes exposées, ni à l'intérieur d'eux-mêmes, ni de leurs descendants.

Les noms de commandes cachées ne peuvent contenir de qualificateurs d'espaces de noms et il faut renommer une commande dans l'espace de noms courant avant de pouvoir en faire une commande cachée. Les commandes que l'on cache au moyen de **interp hide** sont recherchées dans l'espace global même si l'espace de noms courant est différent.

join

La commande **join** crée une chaîne en joignant ensemble des éléments de liste.

Syntaxe

join *Liste ?Jonction?*

Description

L'argument *Liste* doit représenter une liste Tcl valide. Cette commande renvoie une chaîne formée en joignant les divers éléments de la liste au moyen de l'élément désigné par l'argument *Jonction*. Si l'argument *Jonction* n'est pas spécifié, c'est une espace qui est utilisée par défaut. Si l'on veut qu'il n'y ait pas d'élément de jonction entre les éléments, on indiquera explicitement une chaîne vide `""` comme argument *Jonction*. Par exemple :

```
set l [list a b c d e f]
join $l
join $l "-*-"
join $l ""
```

produiront respectivement

```
a b c d e f
a-*-b-*-c-*-d-*-e-*-f
abcdef
```

lappend

La commande **lappend** adjoint des éléments de liste à une variable.

Syntaxe

lappend *NomVar ?Valeur Valeur Valeur...?*

Description

Cette commande traite la variable *NomVar* comme une liste et lui adjoint chacun des arguments *Valeur* comme nouveaux éléments séparés les uns des autres par une espace. Si la variable *NomVar* n'existe pas déjà, elle est créée en tant que liste ayant pour éléments les arguments *Valeur*. Il ne faut pas confondre cette commande **lappend** avec la commande **append** (cf p. 13) qui adjoint des valeurs à une variable de type chaîne.

lindex

La commande **lindex** permet de récupérer les éléments d'une liste.

Syntaxe

lindex *Liste ?indice... ?*

Description

Cette commande traite l'argument qui lui est passé comme une liste Tcl. Si aucun argument *indice* n'est spécifié, la commande renvoie simplement le nom de la liste. Si l'on spécifie un indice de valeur n alors la commande renvoie le n-ième élément de la liste, l'indice 0 correspondant au premier élément de la liste. En extrayant un élément, la commande **lindex** observe les mêmes règles concernant les accolades, guillemets et contre-obliques que l'interpréteur Tcl lui-même mais n'effectue aucune substitution ou interpolation de variables ou de commandes. Chaque élément est traité textuellement.

Si n est négatif ou supérieur ou égal au nombre d'éléments de la liste alors une chaîne vide est renvoyée. On peut utiliser la constante symbolique *end* pour désigner le dernier élément de la liste et appliquer à cette constante des opérations arithmétiques de la forme end-n pour désigner un décalage par rapport à la fin de la liste. `end-1` sera ainsi l'avant-dernier élément de la liste et ainsi de suite. Ce décalage peut aussi être la valeur d'une autre variable : si la variable a vaut 1, on peut ainsi utiliser la syntaxe `end-$a` pour désigner l'avant-dernier élément.

Si des indices supplémentaires sont passés en argument, chaque argument est utilisé à tour de rôle pour sélectionner un sous-élément de l'élément précédemment extrait s'il est lui-même une liste. La commande

```
lindex $a {1 2 3}
```

est synonyme de

```
lindex [lindex [lindex $a 1] 2] 3
```

La possibilité de spécifier plus d'un indice en argument a été introduite avec la version 8.4 de Tcl. En voici quelques exemples :

Commande	*Résultat*
`lindex {a b c}`	a b c
`lindex {a b c} {}`	a b c
`lindex {a b c} 0`	a
`lindex {a b c} 2`	c
`lindex {a b c} end`	c
`lindex {a b c} end-1`	b
`lindex {{a b c} {d e f} {g h i}} 2 1`	h
`lindex {{a b c} {d e f} {g h i}} {2 1}`	h
`lindex {{{a b} {c d}} {{e f} {g h}}} 1 1 0`	g
`lindex {{{a b} {c d}} {{e f} {g h}}} {1 1 0}`	g

linsert

La commande **linsert** insère des éléments dans une liste.

Syntaxe

linsert *Liste Indice Élément?Élément Élément...?*

Description

Cette commande crée une nouvelle liste à partir de la liste *Liste* en insérant tous les arguments *Élément* juste avant l'élément d'indice *Indice*. Chaque argument devient un élément séparé de la nouvelle liste. Si l'indice *Indice* est négatif ou nul, les éléments sont insérés au début de la liste ; s'il est supérieur ou égal au nombre d'éléments de la liste ou bien s'il est représenté par la constante symbolique *end*, alors les éléments sont ajoutés en fin de liste. Les mêmes règles arithmétiques qu'avec la commande **lindex** concernant la constante *end* s'appliquent ici.

list

La commande **list** crée une liste Tcl.

Syntaxe

list *?Arg Arg...?*

Description

Cette commande renvoie une liste constituée de tous les arguments *Arg*, ou une chaîne vide si aucun argument *Arg* n'est spécifié. Des paires d'accolades et des contre-obliques peuvent éventuellement être ajoutées dans certains cas afin de préserver les espaces figurant dans certains éléments et de pouvoir ré-extraire ces éléments par la suite en utilisant la commande **lindex**. Les listes sont construites aussi de telle sorte que la commande **eval** puisse être appliquée à la liste en considérant le premier élément de la liste comme la commande à exécuter et les suivants comme les arguments de cette commande (cf. p. 50).

La commande **list** produit des résultats différents de la commande **concat** pour ce qui concerne les éléments groupés : **concat** retire un niveau de parenthésage en formant une liste à partir d'autres listes, tandis que **list** réserve les éléments originaux. Par exemple, l'instruction suivante

```
list a b {c d e} {f {g h}}
```

renverra

```
a b {c d e} {f {g h}}
```

tandis que

```
concat a b {c d e} {f {g h}}
```

renverrait

```
a b c d e f {g h}
```

llength

La commande **llength** compte le nombre d'éléments d'une liste.

Syntaxe

llength *Liste*

Description

Cette commande traite l'argument qui lui est passé comme une liste et renvoie une valeur décimale représentant le nombre d'éléments de cette liste. Une variable de type chaîne peut ainsi être considérée comme liste. Ainsi par exemple :

```
set txt "a b c d e f"
llength $txt
```

renverra la valeur 6 tandis que

```
llength {$txt}
```

renverrait 1.

load

La commande **load** charge du code compilé et installe de nouvelles commandes dans l'interpréteur Tcl.

Syntaxe

load *NomFichier*
load *NomFichier NomExtension*
load *NomFichier NomExtension Interpréteur*

Description

Cette commande charge dans l'espace mémoire de l'application du code binaire contenu dans un fichier (une bibliothèque statique ou partagée par exemple) et appelle une procédure d'initialisation qui incorpore ce code dans l'interpréteur sous la forme de nouvelles commandes Tcl qui s'ajoutent donc aux commandes de base. L'argument *NomFichier* est le nom du fichier contenant le code binaire. Sur la plupart des systèmes il prend la forme d'une bibliothèque partagée telle qu'un fichier .so (*shared object*) sous Unix, un fichier .dll sous Windows ou une extension de type shlb (*shared library*) sous MacOS. L'argument *NomExtension* est le nom interne de l'extension chargée et est utilisé pour définir le nom de la procédure d'initialisation. L'argument *Interpréteur* est le nom de l'interpréteur dans lequel l'extension est chargée (cf. la commande **interp** à la page 98); si cet argument est omis, l'interpréteur sera celui au sein duquel la commande **load** a été invoquée.

Le rôle de la procédure d'initialisation est typiquement d'ajouter de nouvelles commandes à l'interpréteur Tcl. Pour un interpréteur normal (non sécurisé), le nom de la procédure d'initialisation sera de la forme *extn_Init* où *extn* est le nom de l'extension spécifiée par l'argument *NomExtension* à la seule différence près que la première lettre est convertie en majuscule et les lettres suivantes en minuscules. Par exemple, si *NomExtension* est *lisa* ou *LISA*, la procédure d'initialisation s'appelle *Lisa_Init*.

Si l'interpréteur est sécurisé, le nom de la procédure d'initialisation sera de la forme *extn_SafeInit* au lieu de *extn_Init*. La fonction *extn_SafeInit* doit être écrite avec précautions pour initialiser l'interpréteur uniquement avec des commandes qui sont considérées comme sûres face à un script potentiellement dangereux (voir les questions de sécurité à la page 159).

Syntaxe de la procédure d'initialisation

La procédure d'initialisation doit adopter la syntaxe suivante dans le fichier source en code C qui définit l'extension :

```
typedef int Tcl_PackageInitProc(Tcl_Interp  *interp);
```

L'argument *Interpréteur* identifie l'interpréteur dans le cadre duquel l'extension doit être chargée. La procédure d'initialisation doit renvoyer les valeurs TCL_OK ou TCL_ERROR pour indiquer si elle réussit ou pas. En cas d'erreur, elle devra renvoyer un pointeur vers un message d'erreur. Le résultat de la commande **load** sera la résultat renvoyé par la procédure d'initialisation. On en trouvera des exemples à l'annexe *Extensions de Tcl*.

Le chargement effectif dans l'application d'un fichier de code binaire n'aura lieu qu'une fois pour chaque argument *NomFichier*. Si un fichier *NomFichier* particulier est chargé dans des interpréteurs multiples, seulement le premier appel à la commande **load** chargera le code et invoquera la procédure d'initialisation ; les appels successifs invoqueront eux aussi la procédure d'initialisation mais ne rechargeront pas l'extension. Il n'est pas possible de décharger (d'éliminer de la mémoire) un fichier chargé afin de le recharger. Par ailleurs si un même fichier binaire est appelé plusieurs fois par la commande **load** sous des noms différents, ils seront tous installés dans l'espace mémoire de l'application : cette situation constitue un bogue que l'on cherchera à éviter car les conséquences sont imprévisibles.

La commande **load** supporte aussi les bibliothèques statiques liées à l'application si ces extensions ont été enregistrées par un appel à la fonction **Tcl_StaticPackage**.

Si l'argument *NomExtension* est omis ou spécifié sous la forme d'une chaîne vide, Tcl essaie lui-même de deviner le nom de l'extension. Les règles qu'il applique pour cela peuvent varier d'une plate-forme à l'autre. Sous Unix seul le dernier élément de l'argument *NomFichier* est conservé et un éventuel préfixe *lib* est supprimé puis seuls les caractères alphabétiques (et caractères de soulignement) qui suivent sont conservés. Par exemple, la commande

```
load libxyz4.2.so
```

recherchera une extension nommée *xyz* et la commande

```
load bin/last.so {}
```

recherchera l'extension nommée *last*.

Si l'argument *NomFichier* est une chaîne vide, l'argument *NomExtension* doit impérativement être spécifié. D'autre part la commande **load** recherche d'abord un fichier chargé de manière statique parmi ceux qui ont été enregistrés avec la procédure **Tcl_StaticPackage** puis, si aucun n'est trouvé, recherche un fichier chargé de manière dynamique.

Portabilité

Sous Windows, une commande de chargement peut échouer avec un message d'erreur signalant qu'une bibliothèque n'a pas été trouvée (*library not found*). Cela ne correspond pas nécessairement à la bibliothèque que l'on tente de charger mais à des bibliothèques dépendantes. Pour connaître la liste des bibliothèques dépendantes il suffit d'utiliser la commande suivante dans une console MS-DOS :

```
dumpbin -imports bibl.DLL
```

où `bibl.DLL` est à remplacer par le nom de la bibliothèque à charger avec la commande **load**. D'autre part, au cours du chargement d'une bibliothèque partagée

(DLL), Windows ignore les éléments de type `./` dans la spécification du chemin. Pour éviter cela on utilisera l'instruction sous la forme suivante :

```
load [file join [pwd] bibl.DLL]
```

lrange

La commande **lrange** renvoie un ou plusieurs éléments d'une liste.

Syntaxe

lrange *Liste Premier Dernier*

Description

L'argument *Liste* doit être une liste Tcl valide. Cette commande renvoie une nouvelle liste constituée des éléments de la liste *Liste* dont l'indice est compris inclusivement entre *Premier* et *Dernier*.

On peut utiliser la constante symbolique *end* pour désigner l'indice du dernier élément de la liste. Si l'indice *Premier* est négatif, il est traité comme 0. Si l'indice *Dernier* est supérieur ou égal au nombre d'éléments de la liste initiale il est traité comme synonyme de *end*. Si *Premier* est supérieur à *Dernier*, une chaîne vide est renvoyée.

lreplace

La commande **lreplace** remplace des éléments d'une liste par de nouveaux éléments. Elle est utilisée également pour supprimer des éléments d'une liste.

Syntaxe

lreplace *Liste Premier Dernier?Élément Élément...?*

Description

Cette commande renvoie une nouvelle liste constituée en remplaçant les éléments de la liste *Liste* dont l'indice est compris inclusivement entre les valeurs *Premier* et *Dernier* par les arguments *Élément*. S'il n'y a aucun argument *Élément*, le résultat de la commande **lreplace** est simplement de supprimer les éléments d'indices compris entre *Premier* et *Dernier*. Chaque argument *Élément* devient un élément séparé de la nouvelle liste. L'indice 0 correspond au premier élément de la liste et la constante *end* peut être utilisée pour désigner l'indice du dernier élément.

Si la liste *Liste* est vide, les indices *Premier* et *Dernier* sont ignorés. Si l'indice *Premier* est négatif, il est traité comme 0. L'élément désigné par l'indice *Premier* doit exister effectivement dans la liste sinon une erreur est générée.

Si l'indice *Dernier* est négatif mais supérieur à *Premier*, alors tous les éléments spécifiés sont insérés en début de liste. Si *Dernier* est inférieur à *Premier*, aucun élément n'est supprimé et les nouveaux éléments sont insérés devant l'élément d'indice *Premier*.

lsearch

La commande **lsearch** permet de rechercher un élément particulier dans une liste.

Syntaxe

lsearch *?Options? Liste Motif*

Description

Cette commande recherche parmi les éléments de la liste *Liste* ceux qui correspondent au motif indiqué par l'argument *Motif*. Si un tel élément est trouvé, la commande renvoie son indice dans la liste (le premier élément ayant l'indice 0). Autrement la valeur de retour est -1.

Les arguments optionnels servent à spécifier le mode de recherche à utiliser. Ils peuvent prendre les valeurs suivantes :

-ascii
: Les éléments de la liste sont examinés comme des chaînes ASCII. Cette option n'a de sens que conjointement avec les options **-exact** ou **-sorted**.

-decreasing
: Les éléments de la liste sont triés en ordre décroissant. Cette option n'a de sens qu'avec l'option **-sorted**.

-dictionary
: Les éléments de la liste seront comparés de façon lexicographique. Cette option n'a de sens que conjointement avec les options **-exact** ou **-sorted**.

-exact
: Un élément de la liste doit correspondre au *Motif* caractère par caractère.

-glob
: Cette option signifie que le motif utilise des métacaractères selon la syntaxe décrite avec la commande **string match** (cf. p. 174).

-increasing
: Les éléments de la liste sont triés en ordre croissant. Cette option n'a de sens qu'avec l'option **-sorted**.

-integer
: Les éléments de la liste seront comparés comme des nombres entiers. Cette option n'a de sens que conjointement avec les options **-exact** ou **-sorted**.

-real
: Les éléments de la liste seront comparés comme des nombres en virgule flottante. Cette option n'a de sens que conjointement avec les options **-exact** ou **-sorted**.

-regexp
: Cette option signifie que le motif utilise des métacaractères selon la syntaxe

des expressions régulières décrite avec les commandes **regexp** et **regsub** (cf. p. 142 et p. 148).

-sorted
Cette option suppose que les éléments de la liste sont triés. On l'indique ainsi à la commande **lsearch** qui peut dans ce cas utiliser un algorithme de recherche plus efficace. Si aucune autre option n'est précisée, la liste *Liste* est supposée triée en ordre croissant et faite de chaînes ASCII. Cette option ne peut pas être utilisée pour des recherches avec métacaractères (donc avec les options **-glob** ou **-regexp**).

Les versions de Tcl jusqu'à la version 8.3 n'acceptaient que les options **-exact**, **-glob** et **-regexp** : toutes les autres options ont été introduites à partir de la version 8.4 de Tcl.

Si aucune option n'est précisée, la recherche se fera avec l'option **-glob** par défaut. Si des options contradictoires sont indiquées (comme par exemple -exact, -glob, -regexp et -sorted ou bien encore -ascii, -dictionary, -integer et -real), c'est la dernière trouvée qui est utilisée.

lsort

La commande **lsort** trie les éléments d'une liste.

Syntaxe

lsort *?Options? Liste*

Description

Cette commande trie les éléments de la liste *Liste* et renvoie une nouvelle liste constituée des éléments triés. Par défaut, elle opère un tri selon les codes Ascii des caractères mais on peut employer l'une des options suivantes (ou une abréviation s'il n'y a pas d'ambiguïté) pour modifier le mode de tri :

-ascii
Opère les comparaisons selon l'ordre des codes Ascii C'est la valeur par défaut.

-dictionary
Opère des comparaisons lexicographiques. Ce mode diffère de l'option **-ascii** sur les points suivants :

- il n'y a pas de distinction entre majuscules et minuscules
- si deux chaînes comportent des nombres imbriqués, ces nombres sont comparés selon leur valeur et non pas comme des chaînes de caractères. Par exemple x10y vient entre x9y et x11y.

-integer
Convertit les éléments de la liste en nombres entiers et opère une comparaison numérique. Cela n'a de sens que si tous les éléments de la liste sont des chaînes numériques.

-real
Convertit les éléments de la liste en nombres en virgule flottante et opère une comparaison numérique.

-command *Commande*
Indique une commande Tcl appelée *Commande* qui devra être utilisée pour effectuer les comparaisons. Cette commande *Commande* devra être définie par ailleurs et avoir la syntaxe suivante : elle admet deux paramètres qui représentent les deux éléments à comparer et elle renvoie une valeur négative, nulle ou positive (par exemple -1, 0, 1) selon que le premier élément doit être considéré comme antérieur, identique ou postérieur au second dans la classification.

-increasing
Trie les éléments dans l'ordre croissant. C'est la situation par défaut.

-decreasing
Trie les éléments dans l'ordre décroissant.

-index *nb*

Cette option considère que chaque élément de la liste à trier est lui-même une liste de plusieurs termes : le tri sera fait en comparant seulement le terme d'indice *nb* de chaque élément de la liste de départ (comme toujours le premier terme a pour indice 0). La constante symbolique *end* peut aussi être utilisée à la place du nombre *nb* afin de désigner le dernier terme de chaque élément. Par exemple :

```
lsort -integer -index 1 {{Flaubert 24} {Hugo 18} {Stendhal 12}}
```

renverra la liste

```
{Stendhal 12} {Hugo 18} {Flaubert 24}
```

car la comparaison est faite sur les seconds terme de chacun des trois éléments (option `-index 1`) et de manière numérique (option `-integer`).

-unique

Si cette option est utilisée et que la liste comporte des doublons, seulement un exemplaire de chaque doublon sera retenu. La notion de doublon dépend du mode de tri utilisé. Par exemple, en spécifiant `-index 0`, un tri sera opéré en examinant uniquement le premier terme de chaque élément et considérera donc que `1 a` et `1 b` sont « identiques ». Seulement {1 b} sera alors retenu.

memory

La commande **memory** contrôle les capacités de débogage de la mémoire de Tcl.

Syntaxe

memory *Sous-commande ?Arg Arg...?*

Description

La commande **memory** permet de contrôler l'usage de la mémoire par Tcl. Elle ne fonctionne en réalité que si Tcl lui-même a été compilé en activant l'option de débogage de la mémoire (la constante de compilation TCL_MEM_DEBUG doit être définie au moment de la compilation). La commande **memory** ne sera utile qu'aux programmeurs qui ont besoin d'avoir une information de bas niveau afin de déboguer une application. Elle fonctionne avec les sous-commandes suivantes :

memory info
: Fournit une liste contenant le nombre total d'allocations et de libérations de mémoires effectuées depuis le lancement de Tcl, les paquets couramment alloués [4], les octets alloués et le nombre maximal de paquets et d'octets alloués.

memory trace [**on**—**off**]
: Permet d'activer ou désactiver le traçage de la mémoire. Lorsqu'il est activé, tout appel à la fonction *ckalloc* envoie une ligne d'information au canal stderr comportant le mot ckalloc suivi de l'adresse obtenue pour le bloc de mémoire, de la quantité de mémoire allouée, et le nom du fichier source C avec le numéro de ligne exact d'où provient la demande d'allocation. Par exemple :

```
ckalloc 40e478 98 tclProc.c 1406
```

Les appels à *ckfree* sont tracés de la même manière.

memory validate [**on**—**off**]
: Permet d'activer ou désactiver la validation de la mémoire. Lorsque la validation est activée, à chaque appel à *ckalloc* ou *ckfree*, les zones de sécurité sont inspectées pour tout bloc de mémoire couramment alloué afin de rechercher des problèmes de dépassement de zone et d'écriture au-delà des limites allouées. Cela a pour effet de ralentir considérablement Tcl et ne devrait être utilisé que lorsque de tels problèmes sont réellement suspectés. Un problème de dépassement sera détecté dès le prochain appel à *ckalloc* ou *ckfree* émis après qu'il se sera produit.

memory trace_on_at_malloc *Nb*
: Active le traçage de la mémoire après que *Nb* appels à *ckalloc* ont été invoqués. Cela permet de réduire la quantité d'information produite si on a une

4. Cela correspond au nombre courant d'appels à *ckalloc* qui n'ont pas été annulés par un appel à *ckfree*. *ckalloc* et *ckfree* sont les équivalents dans le code Tcl des traditionnels *malloc* et *free* du langage C.

idée du nombre d'allocations qui précèdent un problème que l'on cherche à analyser.

memory break_on_malloc *Nb*

Après *Nb* allocations de mémoire, les appels à *ckalloc* ne seront plus tracés et un message annonce que le programme essaie d'entrer dans le débogueur C. Tcl émet un signal *SIGINT* adressé à lui-même : si Tcl a été lancé depuis un débogueur, il devrait alors entrer en mode débogage.

memory display *Fichier*

Ecrit une liste de toute la mémoire couramment allouée pour le fichier spécifié.

namespace

La commande **namespace** permet de créer et de manipuler des espaces de noms pour les commandes et les variables.

Syntaxe

namespace *?Option??Arg...?*

Description

La notion d'espace de noms a été étudiée au paragraphe **??** à la page **??**. La commande **namespace** comporte toutes les sous-commandes nécessaires permettant de créer et d'utiliser des espaces de noms et des domaines de portée pour des variables et pour des procédures. Les sous-commandes admises sont les suivantes :

namespace children *?EspaceDeNom??Motif?*
: Renvoie une liste de tous les espaces de noms descendants de l'espace *EspaceDeNom*. Si l'argument *EspaceDeNom* n'est pas spécifié, la commande renvoie les descendants de l'espace courant. Si l'argument *Motif* est spécifié, seuls sont renvoyés les descendants dont le nom correspond au motif ; les règles de correspondance sont les mêmes que celles de la commande **glob** (cf. p. 87). À cela s'ajoutent les conventions suivantes : si le motif commence par **::**, il est utilisé tel quel ; autrement l'espace de noms *EspaceDeNom* (ou le nom qualifié de l'espace courant si cet argument n'est pas indiqué) sera inséré au début du motif. Les noms renvoyés sont eux-mêmes entièrement qualifiés.

namespace code *Script*
: Cette commande permet de reporter à plus tard l'exécution d'un code et de faire que celle-ci ait lieu dans l'espace de noms courant : normalement une procédure de rappel (*callback*) comme celles que l'on définit avec les commandes **bind** ou **button** dans Tk sont exécutées dans l'espace global. La commande **namespace code** n'exécute pas elle-même le code : elle génère une commande qui encapsule l'argument *Script* dans une commande **namespace inscope** (voir ci-dessous) : cette commande **inscope** peut être évaluée dans n'importe quel espace de noms et a pour effet que *Script* est bien évalué dans l'espace de noms courant (c'est-à-dire celui dans lequel **namespace code** a été invoquée).
: D'autre part, des arguments supplémentaires peuvent être ajoutés au script encapsulé, comme c'est souvent le cas avec les procédures de rappel, et ces arguments seront transmis à *Script* comme arguments additionnels.

namespace current
: Renvoie le nom entièrement qualifié de l'espace de noms courant. Bien que le nom de l'espace global soit en principe une chaîne vide, cette commande renvoie néanmoins le symbole **::** dans ce cas particulier.

namespace delete *?EspaceDeNom EspaceDeNom...?*
Chaque espace de noms *EspaceDeNom* est détruit par cette commande avec toutes les procédures, variables et espaces de noms enfants qu'il contient. Si une procédure est en cours d'exécution dans un espace de nom que l'on veut détruire, celui-ci est maintenu en vie jusqu'à ce que la procédure s'achève mais il est marqué de telle sorte que plus aucun code ne puisse chercher à l'utiliser. Si un espace de noms n'existe pas, cette commande renvoie une erreur. Si aucun espace de noms n'est spécifié, la commande ne fait rien.

namespace eval *EspaceDeNom Arg ?Arg...?*
Rend actif l'espace de noms *EspaceDeNom* et évalue le code dans ce contexte. Si l'espace de noms n'existe pas déjà, il est créé. Si plusieurs arguments *Arg* sont spécifiés, ils sont concaténés entre eux à la manière de la commande **eval** et le résultat est évalué dans le contexte de cet espace de noms.

namespace exists *EspaceDeNom*
Renvoie 1 si *EspaceDeNom* est un espace de noms valide dans le contexte courant, 0 sinon. Cette sous-commande n'existe que depuis la version 8.4 de Tcl.

namespace export ?**-clear**? *?Motif Motif...?*
Spécifie quelles commandes sont exportées à partir d'un espace de noms. Ce sont les commandes que, par la suite, on pourra importer dans un autre espace de noms au moyen de la commande **namespace import**. À la fois les commandes qui sont définies dans un espace de noms et celles qui ont pu être importées antérieurement dans cet espace peuvent être exportées par un espace de noms.
Les arguments optionnels *Motif* permettent de désigner toutes les commandes correspondant à l'un de ces motifs (en un sens analogue à celui vu précédemment avec la sous-commande **children**) mais les motifs ne doivent pas contenir de qualificateurs venant d'autres espaces de noms : seulement les commandes de l'espace courant peuvent être exportées. Cette procédure est entièrement formelle : les commandes n'ont pas besoin d'avoir été définies au moment où elles sont exportées ; ce sont en fait leurs *noms* qui sont exportés.
L'option **-clear** permet de remettre à zéro la liste des procédures à exporter avant d'appliquer la commande **namespace export** : autrement les procédures sont ajoutées à la liste de celles qui auraient pu être déjà exportées par des instructions antérieures.
Si aucun *Motif* n'est spécifié et que l'option **-clear** n'est pas utilisée, la commande renvoie la liste des procédures couramment exportées.

namespace forget *?Motif Motif...?*
Cette commande permet de supprimer des commandes importées depuis un espace de noms par la commande **namespace import**. Chaque *Motif* est un nom qualifié (comme *xyz::t* ou *a::b::p**) au moyen d'un espace de noms qui a exporté des procédures et peut utiliser la syntaxe et les métacaractères reconnus par la commande **glob** (cf. p. 87) *à la fin* d'un nom qualifié : on ne peut pas utiliser ces métacaractères au milieu d'un nom d'espace.

namespace import ?-**force**? *?Motif Motif...?*

Cette commande importe des procédures dans un espace de noms. Ces procédures devront avoir été exportées auparavant depuis un autre espace de noms par une commande **namespace export** : l'importation consiste à créer, dans l'espace de noms courant, une nouvelle commande pointant vers la commande dans son espace de noms original. Si la commande importée est en conflit avec une commande déjà existante dans l'espace courant, une erreur est en principe générée: on peut éviter cela en utilisant l'option **-force** qui a pour effet de forcer le remplacement d'une commande déjà existante par une commande importée de même nom.

Les règles concernant les motifs éventuels spécifiés avec la commande **import** sont les mêmes que pour la commande **namespace forget** vue précédemment.

namespace inscope *EspaceDeNom Arg?Arg...?*

Exécute un script dans le contexte d'un espace de noms particulier. Les appels à cette commande sont faits en principe indirectement depuis la commande **namespace code** : celle-ci transforme un script dont l'exécution est différée (*callback*) en une commande **inscope** qui garde la trace de l'espace de nom courant et assure que le code sera bien exécuté, le moment venu, dans l'espace souhaité et non pas dans l'espace global. Cette commande est comparable à la commande **eval** mais elle traite différemment les arguments supplémentaires qui sont éventuellements ajoutés au script différé: le premier argument est traité comme une liste et les arguments supplémentaires sont ajoutés comme des éléments propres de cette liste (comme avec la commande **lappend**). Ce point est important car les procédures de rappel sont souvent des préfixes auxquels on ajoute des arguments avant de les évaluer.

Cette commande ne devrait pas être appelée directement : on utilisera **namespace code** si l'on veut qu'une procédure de rappel s'exécute dans un espace de nom particulier.

namespace origin *Commande*

Renvoie le nom entièrement qualifié de la commande à laquelle l'argument *Commande* fait référence. Le mécanisme des procédures importées qui peuvent elles-mêmes avoir été importées depuis d'autres espaces de noms conduit à des imbrications dont cette commande permet de retrouver la trace : elle renvoie dans tous les cas le nom complet de la commande depuis le premier espace de nom où elle a été définie.

Si l'argument *Commande* ne se réfère pas à une commande importée alors c'est simplement le nom entièrement qualifé de cette commande qui est renvoyé.

namespace parent *?EspaceDeNom?*

Renvoie le nom entièrement qualifié de l'espace parent de celui qui est désigné par l'argument *EspaceDeNom*. Si cet argument *EspaceDeNom* n'est pas spécifié, c'est le nom entièrement qualifé de l'espace courant qui sera renvoyé.

namespace qualifiers *Chaîne*

Cette commande renvoie la partie de la chaîne qui se trouve avant le dernier symbole *::* qu'elle contient éventuellement. Si la chaîne est *::abc::def::x*, la commande renvoie *::abc::def* ; si c'est *::* elle renvoie une chaîne vide. Il s'agit simplement d'une manipulation de chaîne et la commande ne vérifie pas si les éléments composant cette chaîne correspondent effectivement à des espaces de noms existants. Elle est le complément de la commande **namespace tail** .

namespace tail *Chaîne*

Cette commande renvoie la partie de la chaîne qui se trouve après le dernier symbole *::* qu'elle contient éventuellement. Si la chaîne est *::abc::def::x*, la commande renvoie *x* ; si c'est *::* elle renvoie une chaîne vide. Il s'agit simplement d'une manipulation de chaîne et la commande ne vérifie pas si les éléments composant cette chaîne correspondent effectivement à des espaces de noms existants. Elle est le complément de la commande **namespace qualifiers** .

namespace which?**-command**?**?-variable**? *Nom*

Examine l'argument *Nom* soit comme une commande, soit comme une variable et renvoie son nom entièrement qualifié. Les règles suivantes s'appliquent :

- si l'argument *Nom* n'existe pas dans l'espace courant mais existe dans l'espace global, la commande renvoie le nom entièrement qualifié dans l'espace global ;
- si l'argument *Nom* n'existe pas du tout (ni en tant que commande, ni en tant que variable), une chaîne vide est renvoyée ;
- s'il s'agit d'une variable qui a été créée mais qui n'est pas encore définie (par exemple à la suite d'une déclaration avec la commande **variable** ou par utilisation de la commande trace sur cette variable), c'est le nom entièrement qualifié de la variable qui est renvoyé.

Si aucune des options **-command** ou **-variable** n'est spécifiée, l'argument *Nom* est considéré comme une variable.

open

La commande **open** permet d'ouvrir un canal de type fichier ou une suite de processus chaînés (pipelines).

Syntaxe

open *NomFichier*
open *NomFichier Accès*
open *NomFichier Accès Permissions*

Description

Cette commande ouvre un canal représentant un fichier, un port série ou bien une suite de commandes chaînées (*pipeline* en anglais) et fournit une valeur numérique qui servira d'identificateur pour désigner ce canal dans toutes les opérations que l'on souhaite effectuer avec lui (comme par exemple les commandes **read**, **puts** ou **close**).

Si le premier caractère de l'argument *NomFichier* n'est pas une barre verticale (—) alors la commande ouvre un fichier portant le nom *NomFichier*.

L'argument *Accès* permet de spécifier les conditions d'accès au canal ouvert par la commande **open**. Il peut prendre l'une des valeurs suivantes :

- **r** Ouvre le fichier en lecture seulement. C'est la valeur par défaut.
- **r+** Ouvre le fichier en lecture et en écriture. Le fichier doit déjà exister.
- **w** Ouvre le fichier en écriture seulement. Si le fichier n'existe pas, il est créé. Le contenu éventuel de ce fichier sera écrasé par les nouvelles opérations d'écriture.
- **w+** Ouvre le fichier en lecture et en écriture. Si le fichier n'existe pas, il est créé. Le contenu éventuel de ce fichier sera écrasé par les nouvelles opérations d'écriture.
- **a** Ouvre le fichier en écriture seulement. Si le fichier n'existe pas, il est créé. Les écritures se font en fin de fichier, autrement dit, s'ajoutent au contenu déjà existant.
- **a+** Ouvre le fichier en lecture et en écriture. Si le fichier n'existe pas, il est créé. Les écritures se font en fin de fichier, autrement dit, s'ajoutent au contenu déjà existant.

Une autre manière d'écrire les droits d'accès à un fichier que l'on ouvre utilise les termes suivants : on peut en spécifier plusieurs mais l'un d'eux doit obligatoirement être RDONLY, WRONLY ou RDWR.

RDONLY Ouvre le fichier en lecture seulement.

WRONLY Ouvre le fichier en écriture seulement.

RDWR Ouvre le fichier en lecture et en écriture.

APPEND Place le pointeur de position courante à la fin du fichier.

CREAT Crée le fichier s'il n'existe pas déjà.

EXCL Si l'option CREAT est aussi spécifiée, une erreur est générée lorsque le fichier existe déjà.

NOCTTY Si le fichier est un terminal, cette option l'empêche de devenir le terminal contrôlant le processus.

NONBLOCK Empêche le processus de bloquer au cours de l'ouverture du fichier et des opérations d'entrée et de sortie qui peuvent s'ensuivre. Le comportement de cette option est variable selon les systèmes et son usage n'est pas recommandé. Il est préférable d'utiliser la commande **fconfigure** pour placer un fichier en mode non-bloquant.

TRUNC Place le pointeur de position courante au début du fichier, ce qui aura pour effet d'écraser son contenu éventuel.

Enfin l'argument *permissions* est un nombre qui permet de fixer les droits du fichier en liaison avec le masque de mode de création du processus. Par défaut il vaut 0666.

Si le premier caractère de l'argument *NomFichier* est une barre verticale (—) alors tous les caractères suivants de l'argument sont traités comme une série de commandes chaînées (*pipeline*) à invoquer comme avec la commande Tcl **exec**. L'identificateur qui est renvoyé par la commande **open** peut servir alors à écrire sur l'entrée de cette connexion au port série ou à lire sur sa sortie, compte-tenu des conditions d'accès qui ont été spécifiées. Si le canal est ouvert seulement en écriture avec l'option d'accès **w**, les sorties du pipeline sont dirigées sur la sortie standard (stdout) de même qu'avec un canal ouvert seulement en lecture avec l'option d'accès **r** l'entrée du pipeline se fera à partir de l'entrée standard (stdin).

Enfin si l'argument *NomFichier* fait référence à un port série alors ce port est ouvert et initialisé de manière variable suivant le système d'exploitation (voir ci-dessous).

Options de configuration

La commande **fconfigure** peut être invoquée pour fixer ou obtenir certaines options de configuration concernant un port série ouvert par la commande **open** :

fconfigure -mode *bauds*, *parité*, *bits_données*, *bits_arret*

Cette option comporte une série de quatre valeurs séparées par des virgules et qui correspondent au débit en bauds, à la parité, au nombre de bits de données et au nombre de bits d'arrêt pour ce port série. Le débit de bauds indique la vitesse de connexion. La parité est une des lettres n, o, e, m et s qui correspondent respectivement aux options *none*, *odd*, *even*, *mark* et *space*. *bits_données* est un entier compris entre 5 et 8 et *bits_arret* un entier compris entre 1 et 2.

fconfigure -pollinterval *msec*
Cette option est disponible uniquement pour les ports série sous Windows et contrôle l'intervalle de temps maximal entre des événements de fichiers (cf. *fileevent* p. 77). Cette valeur se répercute sur l'intervalle entre l'examen de deux événements par l'interpréteur Tcl : c'est la plus petite valeur qui l'emporte. La valeur par défaut est de 10 millisecondes.

fconfigure -lasterror
Cette option est disponible uniquement pour les ports série sous Windows et permet de récupérer la dernière erreur éventuelle. En cas d'erreur les commandes **read** ou **puts** renvoient seulement une erreur d'entrée/sortie. Avec **fconfigure lasterror** on peut récupérer les détails de cette erreur.

Plates-formes particulières

Concernant l'exécution d'applications externes, on se reportera à la commande **exec** pour un certain nombre de particularités rencontrées sur certains systèmes d'exploitation. L'information qui suit concerne plus spécifiquement le cas des pipelines.

- sous Windows, l'argument *NomFichier* concernant un port série à ouvrir est de la forme **com***X***:**, où X est un nombre entier généralement entre 1 et 4. Ce nombre peut aller jusqu'à 9. Toute tentative d'ouvrir un port inexistant ou bien de numéro supérieur à 9 échouera. Une autre notation consiste à utiliser un nom de fichier de la forme `\\.\comX` où X désigne un numéro de port. Cette dernière méthode est beaucoup plus lente sous Windows 95 et Windows 98.
- sous Windows NT, si Tcl est utilisé interactivement, il peut se produire des interactions indésirables avec la vraie console (s'il y en a une ouverte) : aussi bien en lecture qu'en écriture des caractères peuvent s'égarer d'un canal à l'autre car à la fois l'interpréteur Tcl et les processus enfants sollicitent la console simultanément. Ce problème ne se produit pas si le pipeline est démarré à partir d'un script plutôt que depuis la ligne de commande de Tclsh ou bien entendu s'il ne fait pas usage de l'entrée et de la sortie standard.
- sous Windows 95, un pipeline qui exécute une application MS-DOS 16-bit ne peut être ouvert à la fois en lecture et en écriture car les applications DOS qui reçoivent leur entrée depuis un tube et envoient leur sortie vers un autre tube fonctionnent de manière synchrone. Les pipelines qui n'exécutent pas des applications 16-bit fonctionnent de façon asynchrone et peuvent être ouverts à la fois en lecture et en écriture.
  Le problème concernant l'exécution d'un pipeline depuis la ligne de commandes de Tclsh mentionné au paragraphe précédent reste vrai avec des applications 32-bit.
  Le problème de synchronicité a aussi une autre conséquence : si un pipeline est ouvert pour lire depuis une application DOS 16-bit, la commande **open** ne retournera pas tant que le pipeline n'aura pas reçu une instruction de fin de

fichier de la part de sa sortie. De même si un pipeline est ouvert en écriture pour écrire vers une application DOS 16-bit, aucune donnée ne sera envoyée tant que le pipeline n'aura pas été effectivement refermé.

– sous MacOS, l'ouverture d'un port série n'est pas implémentée actuellement de même que l'ouverture de pipelines. Cela provient du fait que le système MacOS ne connaît pas la notion d'entrées et de sorties standard. Cette situation pourrait changer avec l'arrivée du système MacOSX fondé sur un noyau Unix

– sous Unix, la syntaxe de l'argument *NomFichier* pour l'ouverture d'un port série est en général de la forme `/dev/tty`X, où X est l'une des lettres `a` ou **b**. Le nom de n'importe quel pseudo-fichier qui pointe vers un port série peut être utilisé aussi bien.

  En cas d'exécution d'un pipeline en utilisant Tcl interactivement, c'est-à-dire depuis la ligne de commande de tclsh, on rencontre également les problèmes d'instabilité décrits ci-dessus pour Windows.

package

La commande **package** accomplit les tâches de gestion de modules externes en particulier le chargement des extensions et le contrôle de leurs versions.

Syntaxe

package *Sous-commande Args*

Description

Cette commande maintient à jour, dans une base de données interne, les informations nécessaires concernant les modules disponibles pour l'interpréteur courant : elle peut gérer des versions multiples d'un même module et charger la version correcte requise par une application. Les deux commandes de base utilisées dans les scripts sont **package require** et **package provide**, les autres étant plus spécialement destinées aux tâches de gestion interne de la base de données des extensions. Les sous-commandes reconnues sont les suivantes :

package forget *?Module Module...?*
: Supprime dans l'interpréteur toutes les informations concernant le module spécifié, y compris celles fournies par les commandes **package ifneeded** et **package provide**.

package ifneeded *Module Version ?Script?*
: Cette commande n'est utilisée que dans des scripts de configuration d'un système. Elle indique qu'une version particulière d'un certain module est disponible au besoin et que ce module peut être ajouté à la base de données par exécution du script défini par l'argument *Script*. Le script est stocké en attendant l'invocation future d'une commande **package require**. Le plus souvent ce script installe le chargement automatique (*auto-loading*) des commandes contenues dans le module ou bien invoque directement les commandes **load** ou **source** et ensuite exécute **package provide** pour indiquer la présence du module.

  La base de données peut contenir déjà des informations pour différentes versions du module : dans ce cas le nouveau script remplacera un script déjà existant. Si l'argument *Script* est omis, la commande renvoie le texte du script actuellement défini pour la version *Version* du module *Module* ou bien une chaîne vide si aucune commande **package ifneeded** n'a été invoquée antérieurement pour ce module dans cette version.

package names
: Renvoie la liste des noms de tous les modules de l'interpréteur pour lesquels une version a été fournie au moyen d'une commande **package provide** ou bien un script a été déclaré par une commande **package ifneeded**. L'ordre des éléments dans la liste est arbitraire.

package present ?**-exact**? *Module ?Version?*
Cette commande est équivalente à **package require** à la différence qu'elle n'essaie pas de charger le module lorsqu'il n'est pas déjà chargé.

package provide *Module ?Version?*
Cette commande est utilisée pour indiquer que la version *Version* du module *Module* est rendue disponible dans l'interpréteur. On l'utilise usuellement dans un script déclaré avec la commande **package ifneeded** ou dans un module à la fin de son chargement. Une erreur est générée si une version différente du module a été déclarée par une commande **package provide** antérieure.
Si l'argument *Version* est omis, la commande renvoie le numéro de la version couramment disponible ou bien une chaîne vide lorsqu'aucune commande **package provide** n'a été invoquée pour ce module dans l'interpréteur.

package require ?**-exact**? *Module ?Version?*
Cette commande demande qu'un certain module soit utilisé avec un numéro de version particulier. Les arguments permettent de spécifier le module désiré et la commande assure qu'une version adéquate du module soit chargée dans l'interpréteur. Si la commande réussit, elle renvoie le numéro de la version du module qui est chargée, sinon elle génère une erreur.
Si à la fois l'option **-exact** et l'argument *Version* sont spécifiés, alors seulement cette version précise du module peut être chargée. Si l'option **-exact** est omise, il est possible de charger une version ayant le même numéro majeur [5] et postérieure à celle qui est demandée si celle-ci n'est pas disponible. Si l'option **-exact** et l'argument *Version* sont omis, alors n'importe quelle version du module peut être chargée.
Si une version du module a déjà été fournie par une commande **package provide**, son numéro de version doit impérativement satisfaire aux critères spécifiés par l'option **-exact** et l'argument *Version*: la commande **package require** retourne alors immédiatement. Autrement, la commande recherche dans la base de données interne contenant les informations qui ont pu être fournies par des commandes **package ifneeded** antérieures afin de voir si une version acceptable du module est disponible. Si c'est le cas, c'est le script correspondant au numéro de version le plus élevé qui est invoqué. Si la base de données ne contient pas de version acceptable et si d'autre part une commande **package unknown** a été définie pour l'interpréteur courant, alors cette commande est invoquée. Une fois qu'elle a été exécutée, l'interpréteur examine à nouveau si un module acceptable est maintenant disponible ou s'il existe maintenant un script spécifié par une commande **package ifneeded**.
Si aucune des étapes décrites précédemment n'a pu aboutir, la commande renvoie une erreur.

package unknown *?Commande?*
Cette commande est une commande de la dernière chance si aucune version acceptable d'un module n'a pu être trouvée dans la base de données constituée par des instructions **package ifneeded**. Si l'argument optionnel *Commande*

5. Voir ci-dessous les conventions sur les numéros de version.

est spécifié, il contient la première partie d'une commande ; si cette commande est invoquée à la suite d'une instruction **package require**, Tcl la complète en lui ajoutant deux arguments supplémentaires indiquant le nom et le numéro de version du module désiré.

Si aucun numéro de version n'est fourni à la commande **package require**, alors le deuxième argument supplémentaire sera une chaîne vide. Si la commande **package unknown** est invoquée sans que soit précisé l'argument *Commande*, le script **package unknown** courant est renvoyé s'il y en a un, ou une chaîne vide autrement. Enfin, si l'argument *Commande* est spécifiquement déclaré comme une chaîne vide alors le script **package unknown** courant est supprimé : c'est le seul moyen par lequel on peut supprimer un script existant.

package vcompare *Version1 Version2*

Compare les deux numéros de versions indiqués par les arguments *version1* et *version2*. Renvoie -1 si *version1* est antérieur à *version2*, 0 s'ils sont égaux, et 1 si *version1* est postérieur à *version2*.

package versions *Module*

Renvoie une liste de tous les numéros de versions du module pour lesquels une information a été fournie au moyen d'une commande **package ifneeded**.

package vsatisfies *Version1 Version2*

Renvoie 1 si des scripts écrits pour *verVersion2sion2* fonctionnent sans modification avec *Version1* autrement dit si :

- *Version1* est égale ou postérieure à *Version2* ;
- les deux versions ont le même numéro majeur (voir ci-dessous).

La commande renvoie la valeur 0 dans le cas contraire.

Numérotation des versions d'un module

Un numéro de version est constitué de un ou plusieurs nombres séparés par des points comme par exemple 2 ou 3.1416 ou 3.1.13.1. Le premier nombre est dit numéro de version majeur. Plus il est élevé et plus la version est récente. Les nombres les plus à gauche ont le degré de signification le plus important. Par exemple, une version 2.1 est postérieure à une version 1.3, de même que 3.4.6 est postérieure à 3.3.5.

Si un champ est absent, on considère qu'il est nul. Ainsi une version 1.3 est équivalente à une version 1.3.0 ou 1.3.0.0. Une version particulière est censée être compatible rétrospectivement avec une version antérieure si toutes les deux ont le même numéro majeur. Un changement de numéro majeur signifie donc l'introduction de changements incompatibles.

Indexation des modules

La commande **pkg_mkIndex** vient compléter le mécanisme de gestion des modules et des bibliothèques de procédures au moyen des instructions **package require** et **package provide** qui vient d'être décrit. Cette commande crée dans chacun des

répertoires où elle est appliquée un fichier qui indexe toutes les procédures disponibles et permet le chargement automatique des modules dès que la commande **package require** est invoquée.

pid

La commande **pid** recherche le numéro d'identification des processus en cours.

Syntaxe

pid *?Identificateur?*

Description

Si l'argument *Identificateur* est spécifié, il fait référence à une séquence chaînée de processus (*process pipeline*) créée avec la commande **open**. Dans ce cas, la commande renvoie une liste de tous les numéros d'identification des processus qui constituent la séquence chaînée, dans l'ordre. Ces numéros sont des valeurs décimales. La liste sera vide si *Identificateur* se réfère à un fichier ouvert qui n'est pas une séquence chaînée.

Finalement, si aucun argument n'est spécifié, c'est le numéro d'identification du processus Tcl courant (attribué par le noyau) qui est renvoyé.

proc

La commande **proc** permet de créer des procédures Tcl.

Syntaxe

proc *Nom Args Corps*

Description

La commande **proc** crée une nouvelle procédure Tcl appelée *Nom* et remplace toute commande ou procédure de ce nom qui pourrait exister déjà. L'argument *Corps* contient la définition de la procédure, c'est-à-dire le script qui sera exécuté par l'interpréteur Tcl lorque la procédure sera invoquée.

Habituellement le nom *Nom* n'est pas qualifié (n'est pas précédé d'un espace de noms relatif ou absolu) mais si jamais il l'est alors la procédure *Nom* sera créée dans l'espace de noms correspondant.

L'argument *Args* spécifie les arguments formels de la procédure. C'est une liste éventuellement vide dont chaque terme peut servir d'argument formel dans la définition de la procédure : ces termes peuvent se présenter sous la forme d'un simple nom de variable ou bien d'une liste de deux termes dont le premier désigne une variable formelle et le deuxième une valeur par défaut qui sera attribuée à cette variable si jamais elle n'est pas spécifiée dans une invocation de la procédure.

Lorsque la procédure *Nom* est appelée, une variable locale sera créée pour chacun des arguments formels de la procédure : sa valeur sera la valeur qui lui est transmise par l'instruction appelante ou bien, si aucune valeur n'est spécifiée, la valeur déclarée par défaut. Donc les arguments sans valeur par défaut sont obligatoires et les autres sont facultatifs. Il ne doit donc en principe pas y avoir d'arguments en plus de ceux qui sont déclarés. Il y a toutefois une exception qui permet de créer des procédures avec un nombre d'arguments variable et non connu d'avance : si le *dernier* des arguments *Args* déclarés dans la commande **proc** est précisément nommé **args**, tous les arguments qui seront passés à la procédure à partir de celui-ci seront combinés en une liste et c'est cette liste qui sera transmise comme valeur de l'argument formel **args**. Il appartiendra ensuite à la procédure, dans sa définition, de savoir quoi faire de la liste d'arguments qu'elle reçoit ainsi.

Lorsque le script *Corps* définissant la procédure est exécuté, les noms de variables se réfèrent en principe à des variables locales : d'ailleurs ces variables ne sont créées qu'au moment de l'exécution même de la procédure et sont détruites une fois celle-ci achevée. Elles peuvent donc porter le même nom que des variables existant en-dehors de cette procédure : ces dernières sont en quelque sorte masquées. Toutefois la définition de la procédure peut aussi faire référence, au besoin, à des variables définies en-dehors mais il faut pour cela qu'elle fasse appel aux commandes **global** ou bien **upvar** (cf. p. 90 et 200). D'autre part des variables définies dans un espace de noms particulier ne peuvent être utilisées à l'intérieur de la définition

d'une procédure qu'au moyen des commandes **variable** (et non **global**) ou bien **upvar**.

La commande **proc** renvoie une chaine vide. Pour définir la valeur de retour que doit transmettre une procédure définie avec **proc**, on se sert de la commande **return** (cf. p. 153). En l'absence de commande **return**, la procédure renverra la valeur de retour de la dernière commande qu'elle aura exécutée dans le corps de sa définition. Si une erreur se produit au cours de son exécution alors la procédure dans son ensemble renvoie cette erreur.

puts

La commande **puts** permet d'écrire sur un canal ouvert.

Syntaxe

puts?**-nonewline**? *?Canal? Chaîne*

Description

La commande écrit les caractères spécifiés par l'argument *Chaîne* sur le canal *Canal*. *Canal* doit être un identificateur de canal valide qui aura été obtenu auparavant au moyen des commandes **open** ou **socket**. Il faut bien entendu qu'il ait été ouvert en écriture. Si l'argument n'est pas spécifié alors c'est la sortie standard *stdout* qui est utilisée.

Par défaut la commande **puts** ajoute un saut de ligne à la fin de la chaîne copiée : l'option **-nonewline** permet d'annuler ce comportement. Les symboles de saut de ligne insérés dépendent de la plate-forme sur laquelle on travaille : sous MacOS il s'agit de retours-chariots (CR), sous Unix de sauts de ligne (LF) et sous Windows de la combinaison des deux (CRLF). Il s'agit en fait de la valeur courante spécifiée au moyen de la commande **fconfigure -translation**.

Tcl utilise un tampon interne pour stocker provisoirement les données transmises par la commande **puts**. Tcl ne vide le tampon sur le canal de destination que lorsque celui-ci est plein ou bien si l'on cherche à le fermer. Au besoin, on peut forcer Tcl à vider ce tampon en utilisant la commande **puts** (cf. p. 79). Au moment de vider le tampon, la commande normalement bloque le canal jusqu'à ce que l'opération de transfert soit terminée. Si le canal est en mode non-bloquant l'écriture se fait en arrière-plan dès que le fichier ou le périphérique correspondant est en état de l'accepter. Cela implique que l'application doit utiliser la boucle d'événements de Tcl car c'est seulement à travers cette boucle que Tcl peut déceler si un fichier ou un périphérique est disponible pour recevoir des données. On utilise donc des entrées/sorties en mode non-bloquant dans un contexte d'événements en liaison avec la commande **fileevent** : c'est par un événement de fichier (*fileevent*) que l'on peut être averti que le canal est prêt.

pwd

La commande **pwd** retourne le chemin complet du répertoire courant. Elle n'admet aucun argument, ni aucune option.

Syntaxe

pwd

read

La commande **read** permet de lire depuis un canal ouvert.

Syntaxe

read?**-nonewline**? *Canal* **read** *Canal NbCars*

Description

Cette commande peut être utilisée de deux manières différentes :

– sous la première forme, elle lit toutes les données en provenance du canal jusqu'à la fin du fichier. Si l'option **-nonewline** est utilisée, le dernier caractère lu sera ignoré si jamais il s'agit d'un symbole de fin de ligne. Cette option est ignorée si la commande termine avant d'avoir atteint la fin du fichier.

– sous la deuxième forme, elle lit uniquement le nombre de caractères spécifié par l'argument supplémentaire *NbCars*. S'il y a moins de caractères à lire que la valeur demandée, la lecture s'arrête au dernier caractère restant. Si le canal est configuré pour lire des caractères codés sur plusieurs octets, le nombre *NbCars* correspond au nombre effectif de caractères (et non pas au nombre d'octets utilisés pour les représenter).

Si le canal est en mode non-bloquant, la commande peut ne pas arriver à lire autant de caractères que demandé : une fois lues les données disponibles la commande renverra ces données plutôt que de bloquer en attendant la suite. Si le canal est configuré pour recevoir des caractères multi-octets et qu'il reste dans le tampon quelques octets qui ne constituent pas un caractère valide, la commande **read** ne les renverra que lorsqu'un caractère reconnu aura été complété ou bien si la fin du fichier a été atteinte.

La commande **read** traduit les symboles de saut de ligne suivant les options fixées pour ce canal par la commande **fconfigure -translation**. Les sauts de lignes ne sont pas représentés de la même façon selon les divers systèmes d'exploitation (voir la commande **puts** à la page 139).

regexp

La commande **regexp** compare une chaîne à une expression régulière.

Syntaxe

regexp *?Options? Expression Chaîne ?VarCorresp? ?SousVarCorresp SousVarCorresp...?*

Description

Cette commande détermine si l'expression régulière *Expression* correspond en tout ou en partie à *Chaîne* et renvoie 1 si c'est le cas, 0 sinon.[6] La syntaxe des expressions régulières reconnues par Tcl est expliquée au chapitres *Les expressions régulières*.

Si des arguments additionnels sont spécifiés à la suite de *Chaîne* ils sont traités comme les noms de variables destinées à contenir l'information sur les portions de chaînes capturées par l'expression *Expression*. Le premier de ces arguments, *VarCorresp*, contiendra toute la partie de *Chaîne* qui a été trouvée par l'expression régulière. Les arguments suivants, *SousVarCorresp*, contiendront les portions de *Chaîne* capturées par les sous-motifs entre parenthèses de l'expression.

La commande **regexp** supporte les options suivantes que l'on peut spécifier avant l'expression elle-même :

-about
: Au lieu d'essayer de trouver l'expression régulière, cette option renvoie une liste contenant des informations sur l'expression régulière. Le premier élément de la liste est le nombre de sous-expressions. Le second élément est une liste de propriétés décrivant divers attributs de l'expression régulière. Cette option est utilisée à des fins de débogage.

-all
: Permet que l'expression régulière soit trouvée autant de fois qu'il est possible dans la chaîne, la commande renvoyant alors le nombre de correspondances trouvées. Si la commande est utilisée avec des variables capturantes, celles-ci contiendront les valeurs associées à la dernière correspondance uniquement.

-expanded
: Autorise l'utilisation de la syntaxe développée des expressions régulières dans laquelle les espaces et les commentaires sont ignorés ce qui permet justement d'en introduire afin de rendre l'expression plus lisible et plus compréhensible. C'est l'équivalent de l'option `(?x)` que l'on peut placer directement dans une expression.

-indices
: Modifie ce qui sera stocké dans les variables *SousVarCorresp*. Au lieu de

6. À moins que l'option **-inline** ne soit spécifiée. Voir ci-dessous.

stocker les caractères trouvés, chaque variable contiendra une liste de deux valeurs entières représentant les indices du premier et du dernier caractère de la correspondance dans la chaîne *Chaîne*.

-inline
Permet que la commande renvoie sous forme d'une liste les données qui normalement seraient stockées dans des variables capturantes. Avec cette option on peut donc ne pas spécifier de noms de variables capturantes. Si elle est utilisée conjointement avec l'option **-all**, les résultats obtenus à chaque itération sont agrégés à la liste. Pour chaque itération, la commande ajoute à la liste la correspondance entière puis les éléments correspondant aux sous-motifs éventuels. Par exemple :

regexp `-inline -- {\w(\w)} " script "`

⇝ *sc c*

regexp `-inline` −**all** `-- {\w(\w)} " script "`

⇝ *sc c ri i pt t*

-line
Permet de rendre la recherche sensible aux symboles de saut de ligne. Par défaut, un caractère de saut de ligne est traité comme un caractère ordinaire. Avec cette option, le métacaractère « . » et les expressions négatives entre crochets (expressions commençant par `[^`) ne représenteront plus le caractère de saut de ligne. Cela permet d'empêcher que des correspondances s'étendent sur plusieurs lignes. Les symboles d'ancrage sont alors traités différemment : le début de ligne symbolisé par un circonflexe `^` désignera une position juste après un saut de ligne et la fin de ligne symbolisée par un dollar `$` désignera une position juste avant un saut de ligne.
Cette option revient à spécifier à la fois les options **-linestop** et **-lineanchor** et est l'équivalent de la directive `(?n)` que l'on peut placer directement dans une expression.

-lineanchor
Modifie le comportement des symboles d'ancrage circonflexe `^` et dollar `$` afin qu'ils correspondent à de véritables commencement ou fin de ligne respectivement. C'est l'équivalent de la directive `(?w)` que l'on peut placer directement dans une expression.

-linestop
Modifie le comportement du métacaractère « . » et des expressions négatives entre crochets (expressions commençant par `[^`) afin qu'elles s'arrêtent aux caractères de saut de ligne. C'est l'équivalent de la directive `(?p)` que l'on peut placer directement dans une expression.

-nocase
Supprime la distinction entre majuscules et minuscules.

-start *Index*
Spécifie l'indice d'un caractère de la chaîne à partir duquel doit commencer la recherche. Avec cette option, la symbole d'ancrage circonflexe ^ ne marquera pas le début d'une ligne mais la séquence d'échappement `\A` désignera toujours le début de la chaîne à la position désignée par l'argument *Index*. Si l'option **-indices** est spécifiée, les positions seront indexées à partir du début absolu de la chaîne.

--
Marque la fin des options. L'argument qui suit un double tiret est traité comme une expression régulière même si celle-ci commence par un tiret.

registry

La commande **registry** ser,t à manipuler le registre de Windows.

Syntaxe

package require registry 1.0
registry *Sous-commande NomClé ?Arg Arg...?*

Description

Le module **registry** procure un ensemble d'opérations de base permettant d'effectuer des opérations sur le registre de Windows. Il implémente une commande Tcl appelée **registry**. Cette commande est supportée uniquement sur les plates-formes Windows. Elle doit être utilisée avec prudence car un registre corrompu rendra le système inutilisable. Ce n'est pas une commande de base de Tcl. Pour la rendre disponible, il faut charger l'extension appelée *registry* qui se présente sous la forme d'une bibliothèque partagée (comme tclreg10.dll pour la version 1.0).

L'argument *NomClé* est le nom d'une clé du registre. Les clés du registre peuvent prendre l'une des formes suivantes :

*nomHote**nomRacine**nomClés*
*nomRacine**nomClés*
nomRacine

où *nomHote* spécifie le nom de n'importe quel hôte Windows qui exporte son registre. La composante *nomRacine* doit être l'une des valeurs :

HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_CURRENT_CONFIG, HKEY_PERFORMANCE_DATA ou HKEY_DYN_DATA.

La composante *nomClés* peut être un ou plusieurs noms de clés de registre séparés par des contre-obliques \.

Les sous-commandes définies pour la commande **registry** sont :

registry delete *NomClé ?NomValeur?*
: Si l'argument optionnel *NomValeur* est présent, la valeur spécifiée par *NomClé* sera supprimée du registre. Si l'argument optionnel *NomValeur* est omis, la clé spécifiée ainsi que toutes les sous-clés et valeurs en-dessous d'elle dans la hiérarchie du registre seront détruites. Si la clé n'a pas pu être supprimée, une erreur est générée. Si la clé n'existe pas, la commande est sans effet.

registry get *NomClé NomValeur*
: Renvoie les données associées à la valeur *NomValeur* sous la clé *NomClé*. Si soit la clé soit la valeur n'existent pas, une erreur est générée. Pour plus de détails sur le format des données renvoyées, voir le paragraphe *Types supportés* ci-dessous.

registry keys *NomClé ?Motif?*
Si *Motif* n'est pas spécifié, la commande renvoie une liste des noms de toutes les sous-clés de *NomClé*. Si *Motif* est spécifié, seuls les noms correspondant au motif sont renvoyés. Les correspondances avec le motif sont déterminées selon les mêmes règles que pour la commande **string match**. Si l'argument *NomClé* spécifié n'existe pas, une erreur est générée.

registry set *NomClé ?NomValeur Données ?Type??*
Si *NomValeur* n'est pas spécifié, la commande crée la clé *NomClé* si celle-ci n'existe pas déjà. Si *NomValeur* est spécifié, elle crée la clé *NomClé* et la valeur *NomValeur* si nécessaire. Le contenu de *NomValeur* est fixé à *Données* avec le type indiqué dans l'argument *Type*. Si *Type* n'est pas spécifié, le type par défaut sera *sz*. Pour plus de détails sur le format des données renvoyées, voir le paragraphe *Types supportés* ci-dessous.

registry type *NomClé NomValeur*
Renvoie le type de la valeur *NomValeur* dans la clé *NomClé*.

registry valeurs *NomClé ?Motif?*
Si *Motif* n'est pas spécifié, la commande renvoie une liste des noms de toutes les valeurs de *NomClé*. Si *Motif* est spécifié, seuls les noms correspondant au motif sont renvoyés. Les correspondances avec le motif sont déterminées selon les mêmes règles que pour la commande **string match**.

Types supportés

Chaque valeur associée à une clé du registre contient des données d'un type particulier dans une représentation spécifique à ce type. La commande **registry** effectue la conversion entre cette représentation interne et une représentation manipulable par des scripts Tcl. Dans la plupart des cas, les données sont renvoyées simplement comme une chaîne Tcl. Le type indique l'usage prévu pour les données mais ne modifie pas réellement la représentation. Pour certains types, la commande **registry** renvoie les données sous une forme différente pour les rendre plus faciles à manipuler. Les types suivants sont reconnus par la commande **registry** :

binary
La valeur de registre contient des données binaires arbitraires. Les données sont représentées telles quelles dans Tcl en incluant les octets nuls éventuels.

none
La valeur de registre contient des données binaires arbitraires sans type défini. Les données sont représentées telles quelles dans Tcl en incluant les octets nuls éventuels.

sz
La valeur de registre contient une chaîne terminée par un octet nul. Les données sont représentées dans Tcl comme une chaîne.

expet_sz
La valeur de registre contient une chaîne terminée par un octet nul contenant des références à des variables d'environnement dans le style Windows habituel

(par exemple "%PATH%"). Les données sont représentées dans Tcl comme une chaîne.

dword
: La valeur de registre contient un nombre 32-bits dans l'ordre *little-endian*. Les données sont représentées dans Tcl comme une chaîne décimale.

dword_big_endian
: La valeur de registre contient un nombre 32-bits dans l'ordre *big-endian*. Les données sont représentées dans Tcl comme une chaîne décimale.

link
: La valeur de registre contient un lien symbolique. Les données sont représentées telles quelles dans Tcl en incluant les octets nuls éventuels.

multi_sz
: La valeur de registre contient un tableau de chaînes terminées par un octet nul. Les données sont représentées dans Tcl comme une liste de chaînes

resource_list
: La valeur de registre contient une liste de ressources de pilote de périphérique. Les données sont représentées telles quelles dans Tcl en incluant les octets nuls éventuels.

En plus des noms symboliques ci-dessus, les types inconnus sont identifiés en utilisant un entier 32-bits qui correspond au code renvoyé par l'interface système. Dans ce cas, les données sont représentées telles quelles dans Tcl en incluant les octets nuls éventuels.

regsub

La commande **regsub** opère des substitutions basées sur des motifs d'expressions régulières.

Syntaxe

regsub *?Options? Expression Chaîne Substitution NomVar*

Description

Cette commande compare l'expression régulière *Expression* avec la chaîne *Chaîne*, et copie *Chaîne* dans la variable désignée par l'argument *NomVar*: si une correspondance est trouvée, elle est remplacée au cours de la copie par la chaîne de substitution désignée par l'argument *Substitution*. Celui-ci doit être écrit selon la syntaxe des motifs de substitution expliquée à la section **??**. La commande renvoie le nombre de portions de la chaîne qui ont été trouvées et remplacées.

La commande **regsub** supporte les options suivantes que l'on peut spécifier avant l'expression elle-même :

-all	**-linestop**	**-start**
-expanded	**-line**	**--**
-lineanchor	**-nocase**	

Ces options ont exactement la même signification que les options de même nom appartenant à la commande **regexp** (cf. p 142) à l'exception de l'option **-all**. Celle-ci, dans le cas de **regsub**, permet que toutes les correspondances possibles soient trouvées et remplacées par le motif de substitution. Si on ne spécifie pas cette option, seule la première correspondance rencontrée est traitée. Si **-all** est spécifiée, les séquences `&` et $\backslash n$ sont évaluées pour chaque correspondance avec l'information tirée de cette correspondance.

rename

La commande **rename** sert à renommer ou à détruire une commande.

Syntaxe

rename *AncienNom NouveauNom*

Description

Renomme la commande antérieurement appelée *AncienNom* et lui attribue le nouveau nom *NouveauNom*. Si l'argument *NouveauNom* est une chaîne vide alors la commande *AncienNom* est détruite. Les arguments *AncienNom* et *NouveauNom* peuvent comporter des qualificateurs d'espaces de noms (cf. la commande **namespace** à la page 124). Si une commande est renommée dans un nouvel espace de noms, les prochaines invocations de cette commande se feront par rapport à cet espace. La commande **rename** retourne toujours une chaîne vide.

resource

La commande **resource** permet de manipuler la branche de ressources des fichiers Macintosh.

Syntaxe

resource *Sous-commande ?Arg Arg...?*

Description

Cette commande Tcl n'a de sens que pour les plates-formes Macintosh. Tout fichier sous MacOS possède deux branches : la branche de données et la branche de ressources. Les commandes usuelles pour la manipulation des fichiers (**open**, **close** etc.) s'appliquent en fait à la branche de données. Pour avoir accès à la branche de ressources, il faut utiliser la commande **resource**. Les ressources du Macintosh peuvent être de types divers : les différents types sont désignés au moyen d'un identificateur en quatre lettres, comme par exemple DLOG pour les dialogues, MENU pour les menus etc. Dans un même type, il peut y avoir plusieurs ressources : elles sont identifiées au moyen d'un numéro unique d'identification ou éventuellement au moyen d'un nom.

Les diverses opérations sur les ressources d'un fichier sont effectuées au moyen de sous-commandes. La syntaxe obéit à la même logique que les commandes **open** : à l'ouverture d'une ressource, un numéro de référence est attribué qui sera utilisé par la suite pour les autres opérations que l'on veut effectuer sur cette ressource. Ce numéro de référence ne doit pas être confondu avec le numéro d'identification d'une ressource dans un type particulier. Le numéro de référence est attribué par Tcl pendant la durée d'ouverture d'une ressource tandis que le numéro d'identification appartient en propre à la ressource.

Les sous-commandes valides sont les suivantes (n'importe quelle abréviation unique de ces sous-commandes est acceptée) :

resource close *RefRessource*
: Ferme la ressource ayant le numéro de référence *RefRessource*. Les ressources de ce fichier de ressources ne sont alors plus disponibles.

resource delete *?Options? TypeRessource*
: Supprime la ressource spécifiée par l'argument *Options* et le type *TypeRessource*. L'argument *Options* fournit plusieurs méthodes pour désigner la ressource à supprimer. Les options disponibles sont :

  -id *IdRessource*
  : avec l'option **-id**, la ressource ayant le numéro d'identification *IdRessource* sera détruite.

  -name *NomRessource*
  : avec l'option **-name**, la ressource nommée *NomRessource* est détruite.

Si l'option **-id** est elle aussi spécifiée, il doit y avoir une ressource correspondant à la fois à ce nom et à cet identificateur.

-file *RefRessource*
avec l'option **-file**, la ressource sera détruite dans le fichier pointé par l'argument *RefRessource*. Autrement la première ressource rencontrée correspondant à un nom *NomRessource* ou à un identificateur *IdRessource* dans la branche de ressources sera détruite. Pour connaître le chemin complet du fichier, on utilisera la commande **resource files**.

resource files *?RefRessource?*
Si l'argument *RefRessource* n'est pas fourni, la commande renvoie une liste Tcl des numéros de référence de tous les fichiers de ressources qui ont été ouverts antérieurement par une commande **resource open**. Cette liste est dans l'ordre séquentiel naturel dans lequel le Macintosh a stocké les ressources. Si l'argument *RefRessource* est spécifié, la commande renvoie le chemin complet du fichier dont la branche de ressources est représentée par l'identificateur *RefRessource*.

resource list *TypeRessource ?RefRessource?*
Liste tous les identificateurs de ressources de type *TypeRessource* existantes. Si le numéro de référence *RefRessource* est spécifié, la commande limite ses recherches à ce fichier de ressources particulier. Autrement tous les fichiers de ressources couramment ouverts par l'application sont examinés. La valeur de retour est une liste Tcl de tous les noms ou les identificateurs des ressources trouvées: les noms sont toujours recherchés ou renvoyés de préférence aux nombres, autrement dit la commande renverra des noms si ceux-ci existent (ne sont pas une chaîne vide) et sinon des numéros d'identification.

resource open *NomFichier ?Accès?*
Cette commande ouvre la branche de ressources du fichier *NomFichier*. Les autorisations d'accès standard peuvent être spécifiées (cf. la commande **open** p. 128). Un numéro de référence est renvoyé: c'est ce numéro qui est utilisé par un certain nombre de sous-commandes de la commande **resource**. Une erreur peut se produire si le fichier spécifié n'existe pas ou bien s'il ne possède pas de branche de ressources (branche vide). Néanmoins si on ouvre le fichier avec les droits en écriture le fichier et/ou la branche de ressources seront créés sans provoquer d'erreur.

resource read *TypeRessource IdRessource ?RefRessource?*
Cette commande lit intégralement la ressource ayant le type *TypeRessource* et le nom ou l'identificateur *IdRessource* et renvoie le résultat. Si l'argument *RefRessource* est spécifié, la recherche se limite à ce fichier, sinon elle s'étend à tous les fichiers ouverts dans l'application. La plupart des ressources du Macintosh sont des données binaires (pouvant contenir des octets nuls ou des caractères non-ASCII) et il peut être nécessaire d'utiliser la commande **binary** (cf. p. 23) pour les traiter.

resource types *?RefRessource?*
Cette commande renvoie une liste Tcl de tous les types de ressources trouvés

dans le fichier désigné par l'argument *RefRessource*. Si *RefRessource* n'est pas spécifié, elle renvoie les types de ressources trouvés dans tous les fichiers couramment ouverts dans l'application.

resource write *?Options? TypeRessource Données*

Cette commande écrit les données passées dans l'argument *Données* comme une nouvelle ressource ayant le type *TypeRessource*. Plusieurs options sont disponibles pour indiquer où et comment la ressource doit être stockée :

-id *IdRessource*

Avec l'option **-id**, l'identificateur *IdRessource* est utilisé pour la nouvelle ressource, sinon un identificateur unique est créé pour éviter des conflits avec des ressources déjà existantes.

-name *NomRessource*

Si l'option **-name** est spécifiée, la ressource sera nommée*NomRessource*, autrement son nom sera une chaîne vide.

-file *RefRessource*

Avec l'option **-file**, la ressource sera écrite dans le fichier désigné par l'argument *RefRessource*, sinon la ressource la plus récemment ouverte sera utilisée.

-force

Si la ressource cible existe déjà, Tcl par défaut émet une erreur plutôt que de l'écraser avec la nouvelle ressource. Cette option au contraire permet de forcer l'écrasement d'une ressource déjà existante.

return

La commande **return** s'utilise dans le cadre de la définition d'une procédure et provoque la fin de l'exécution de cette procédure et le retour à l'instruction qui l'appelait.

Syntaxe

return ?**-code** *Code??*?**-errorinfo** *Info??*?**-errorcode** *Code??Chaîne?*

Description

Cette commande provoque le retour immédiat et inconditionnel depuis la procédure couramment exécutée (ou depuis une commande **source**). L'argument optionnel *Chaîne* permet de transmettre une valeur de retour : il peut s'agir de n'importe quel type de variable Tcl, y compris une liste.

Il ne faut pas confondre cette valeur renvoyée avec l'option **-code** utilisée pour générer des codes de retour exceptionnels. L'option **-code** sert à implementer de nouvelles structures de contrôle et permet de transmettre des conditions exceptionnelles aux instructions appelantes. Habituellement la commande **return** s'achève en transmettant à l'interpréteur la valeur interne TCL_OK. Si l'on veut modifier cette valeur, on utilise l'option **-code** en lui attribuant l'une des valeurs symboliques *Code* suivantes :

ok Retour normal avec code interne TCL_OK (comme lorsque l'option n'est pas utilisée) ;

error Retour avec code interne TCL_ERROR comme lorsque l'on utilise la commande **error** si ce n'est que les variables **errorInfo** et **errorCode** sont traitées différemment (voir ci-dessous) ;

return Retour avec code interne TCL_RETURN ce qui provoque le retour également de la procédure appelante ;

break Retour avec code interne TCL_BREAK ce qui a pour effet de provoquer la sortie d'une éventuelle boucle itérative dans la procédure appelante (cf. la commande **break**) ;

continue Retour avec code interne TCL_CONTINUE ce qui a pour effet de provoquer l'interruption de l'exécution d'une éventuelle itération dans la procédure appelante (voir la commande **continue**) ;

valeur *valeur* doit être un nombre entier qui sera une valeur renvoyée à l'interpréteur et dont la signification devra être gérée par le programmeur.

Les deux autres options que l'on peut ajouter sur la ligne de la commande **return** sont **-errorinfo** et **-errorcode**. Elles n'ont de sens que si l'option **-code** est utilisée avec la valeur de code *error* vue ci-dessus :

– l'option **-errorinfo** spécifie une valeur initiale de la trace de la pile pour la variable **errorInfo**. Si elle n'est pas spécifiée, la trace de la pile laissée

dans la variable **errorInfo** incluera les appels à la procédure et aux niveaux supérieurs sur la pile d'exécution mais ne comportera aucune information concernant le contexte de l'erreur à l'intérieur de la procédure. Typiquement la valeur attribuée à cette option est fournie à partir de l'information placée dans la variable **errorInfo** lorsqu'une commande **catch** intercepte une erreur à l'intérieur de la procédure.

– si l'option **-errorcode** est spécifiée alors la valeur attribuée à cette option servira de valeur pour la variable **errorCode**. Si l'option n'est pas spécifiée, la variable **errorCode** contiendra la valeur NONE.

Base sécurisée

Ce module, écrit en code Tcl, définit un mécanisme pour créer et manipuler des interpréteurs sécurisés.

Syntaxe

::safe::interpCreate *?Esclave? ?options...?*
::safe::interpInit *Esclave ?options...?*
::safe::interpConfigure *Esclave ?options...?*
::safe::interpDelete *Esclave*
::safe::interpAddToAccessPath *Esclave répertoire*
::safe::interpFindInAccessPath *Esclave répertoire*
::safe::setLogCmd *?cmd arg...?*

Description

Safe Base est une extension écrite en Tcl qui définit des procédures permettant d'exécuter de manière sécurisée des scripts dont la fiabilité n'est pas assurée et pour fournir un accès sûr à des commandes pontentiellement dangereuses. Il s'appuie sur les fonctionnalités de base de Tcl concernant le contrôle des interpréteurs: l'intérêt de ce module est de fournir un mécanisme prêt à l'emploi garantissant un fonctionnement correct.

Safe Base empêche des scripts non fiables de corrompre l'état de la machine qui les reçoit et les exécute, d'avoir accès à l'information stockée sur cette machine ou dans l'application qui les héberge.

Safe Base permet à un interpréteur maître de créer des interpréteurs restreints contenant un ensemble d'alias prédéfinis pour des commandes telles que **source**, **load**, **file**, **encoding** et **exit** qui sont normalement entièrement retirées d'un interpréteur créé avec l'option -safe.

Aucune fuite concernant la structure du système de fichiers ne pourra se produire à partir de l'interpréteur sécurisé car celui-ci n'a accès qu'à un chemin virtuel. Lorsque l'interpréteur sécurisé demande qu'un fichier soit sourcé, il utilise la donnée fournie par le chemin virtuel comme partie du chemin du fichier à sourcer. C'est l'interpréteur maître qui le traduit de manière transparente en un véritable nom de répertoire et exécute l'opération requise.

Différents niveaux de sécurité peuvent être sélectionnés au moyen des options des commandes décrites ci-dessous. Toutes les commandes fournies dans l'interpréteur maître par le module Safe Base résident dans un espace de noms appelé *safe*.

Commandes

Les commandes suivantes sont définies dans l'interpréteur maître:

::safe::interpCreate *?Esclave? ?options...?*
Crée un interpréteur sécurisé, installe les alias décrits dans la section *Alias* et

initialise le mécanisme du module en fonction des options spécifiées (voir la section *Options* ci-dessous). Si l'argument *Esclave* est omis, un nom sera généré automatiquement. La commande renvoie toujours le nom de l'interpréteur créé.

::safe::interpInit *Esclave ?options...?*
Cette commande est analogue à **interpCreate** sauf qu'elle ne crée pas l'interpréteur sécurisé. *Esclave* doit avoir été créé par un autre moyen (par exemple par une commande `interp create -safe`).

::safe::interpConfigure *Esclave ?options...?*
Si aucune option n'est indiquée, la commande renvoie la liste de toutes les valeurs d'options dans cet interpréteur sécurisé *Esclave* sous forme d'une liste de paires *options-valeurs*. Si un seul argument additionnel est fourni, la commande renvoie une liste de deux éléments *Nom* et *Valeur* où *Nom* est le nom complet de cette option et *Valeur* sa valeur courante dans l'interpréteur *Esclave*. Si plus de deux arguments additionnels sont fournis, la commande reconfigure dans l'interpréteur sécurisé les options indiquées. Par exemple :

```
# Créer un nouvel interp i1 avec la même configuration qu'un interp i0 :
set i1 [eval safe::interpCreate [safe::interpConfigure $i0]]
# Récupérer le script de destruction courant :
set dh [safe::interpConfigure $i0 -del]
# Modifier uniquement les deux options −statics et −delete :
safe::interpConfigure $i0 -delete {foo bar} -statics 0 ;
```

::safe::interpDelete *Esclave*
Détruit l'interpréteur sécurisé et les structures associées dans l'interpréteur maître. Si un script a été spécifié avec l'option **-deleteHook** pour cet interpréteur, il est évalué avant la destruction de l'interpréteur, avec le nom de l'interpréteur comme argument additionnel.

::safe::interpFindInAccessPath *Esclave repert*
Cette commande trouve et renvoie la donnée pour le véritable répertoire *repert* dans le chemin d'accès virtuel courant de l'interpréteur sécurisé. Une erreur est générée si le répertoire n'est pas trouvé. Par exemple :

```
$esclave eval [list set tk\_library \
        [::safe::interpFindInAccessPath $name $tk\_library]]
```

::safe::interpAddToAccessPath *Esclave repert*
Cette commande ajoute *repert* au chemin virtuel maintenu par l'interpréteur sécurisé dans le maître, et renvoie la donnée qui peut être utilisée dans l'interpréteur sécurisé pour accéder aux fichiers de ce répertoire. Si le répertoire est déjà sur le chemin virtuel, la commande renvoie seulement la donnée sans ajouter à nouveau le répertoire au chemin. Par exemple :

```
$esclave eval [list set tk\_library \
        [::safe::interpAddToAccessPath $name $tk\_library]]
```

::safe::setLogCmd *?cmd arg...?*

Cette commande installe un script qui sera appelé lorsque certains événements se produisent dans un interpréteur sécurisé. Le but de cette commande est d'aider à déboguer les interpréteurs sécurisés. Elle permet d'obtenir des messages d'erreurs complets lorsqu'une erreur se produit, tout en ne laissant transparaître que des messages génériques à l'intérieur de l'interpréteur sécurisé afin que des informations sensibles telles que les noms réels de certains répertoires ne puissent être vus par l'interpréteur esclave. Si cette commande est invoquée sans argument, elle renvoie le script couramment installé. Si l'argument est une chaîne vide, le script courant est retiré et les informations ne sont plus consignées. Le script est invoqué avec un argument additionnel décrivant l'événement qui l'a déclenché.

Exemple d'utilisation :

```
::safe::setLogCmd puts stderr
```

Voici maintenant l'information que l'on recueillerait si l'interpréteur sécurisé avait tenté de sourcer un fichier qui ne se trouve pas sur son chemin virtuel :

```
NOTICE for slave interp10 : Created
NOTICE for slave interp10 : Setting accessPath=(/foo/bar)
                            staticsok=1 nestedok=0 deletehook=()
NOTICE for slave interp10 : auto\_path in interp10 has been
                            set to {$p(:0:)}
ERROR for slave interp10 : /foo/bar/init.tcl:
                            no such file or directory
```

Dans le même temps l'interpréteur sécurisé lui-même n'aura reçu qu'un message disant que le fichier n'a pas été trouvé.

Options

Les options suivantes sont communes aux commandes *::safe::interpCreate*, *::safe::interpInit* et *::safe::interpConfigure*. Elles peuvent être abrégées sans distinction entre minuscules et majuscules.

-accessPath *listeRepert*

Cette option établit la liste des répertoires à partir desquels l'interpréteur sécurisé peut sourcer et charger des fichiers. Si cette option n'est pas spécifiée, ou si c'est une liste vide, l'interpréteur sécurisé utilisera les mêmes répertoires que son maître pour l'autochargement.

-statics *bool*

Cette option spécifie si l'interpréteur sécurisé est autorisé à charger des modules liés statiquement (comme `load {} Tk`). La valeur par défaut est **true**.

-noStatics

Cette option est simplement une abréviation pour `-statics false` et spécifie que l'interpréteur sécurisé n'est pas autorisé à charger des modules liés statiquement.

-nested *bool*
Cette option spécifie si l'interpréteur sécurisé est autorisé à charger des modules dans ses propres sous-interpréteurs. La valeur par défaut est **false**.

-nestedLoadOk
Cette option est simplement une abréviation pour `-nested true` et spécifie que l'interpréteur sécurisé est autorisé à charger des modules dans ses propres sous-interpréteurs.

-deleteHook *Script*
Si l'argument *Script* n'est pas vide, il sera évalué dans le maître avec le nom de l'interpréteur sécurisé comme argument additionnel juste avant que l'interpréteur sécurisé soit effectivement détruit. Une valeur vide a pour effet de désinstaller un script préalablement déclaré. La valeur par défaut est justement {}.

Alias

Les alias suivants sont prédéfinis dans l'interpréteur sécurisé:

source *NomFichier*
Le fichier requis, fichier source Tcl, est sourcé dans l'interpréteur sécurisé s'il est trouvé. L'alias **source** peut sourcer uniquement des fichiers qui se trouvent dans les répertoires du chemin virtuel de l'interpréteur sécurisé. Cet alias exige que l'interpréteur sécurisé utilise l'un des jetons symboliques qui figurent dans son chemin virtuel pour désigner le répertoire dans lequel le fichier à sourcer peut être trouvé. Voir la section **Sécurité** ci-dessous pour une discussion des restrictions concernant les noms de fichiers valides.

load *NomFichier*
Le fichier requis, une bibliothèque partagée, est chargé dynamiquement dans l'interpréteur sécurisé s'il est trouvé. Le nom du fichier doit comporter l'un des jetons symboliques qui figurent dans le chemin virtuel de cet interpréteur sécurisé pour qu'il puisse être trouvé. En outre, la bibliothèque partagée doit contenir un point d'entrée sécurisé, c'est-à-dire défini par une procédure *module*_SafeInit pointant sur un interpréteur sécurisé plutôt qu'une fonction *module*_Init (voir la commande **load** p.113 et l'annexe *Extensions de Tcl*).

file *?SousCommande Args...?*
L'alias **file** donne accès à un ensemble de sous-commandes sûres de la commande **file**. Il autorise uniquement les sous-commandes **dirname**, **join**, **extension**, **root**, **tail**, **pathname** et **split**.

encoding *?SousCommande Args...?*
L'alias **encoding** donne accès à un ensemble de sous-commandes sûres de la commande **encoding**. Il interdit de fixer l'encodage système mais autorise toutes les autres sous-commandes y compris la sous-commande **system** pour obtenir la valeur actuelle.

exit
Avec cet alias, l'interpréteur qui invoque cette commande est détruit mais le

processus Tcl dans lequel il réside n'est pas interrompu.

Sécurité

Le module Safe Base ne tente pas d'éliminer complètement les attaques de nuisance et de déni de service. Ces sortes d'attaques empêchent temporairement l'utilisateur ou l'application d'utiliser l'ordinateur pour accomplir des tâches usuelles, par exemple, en accaparant le microprocesseur ou bien le domaine de l'écran entier. Ces attaques, bien que gênantes, sont considérées comme moins graves en général que des atteintes à l'intégrité du système et au secret des données qui sont celles que Safe Base cherche à contrer.

Les commandes disponibles dans un interpréteur sécurisé, en plus de l'ensemble sécurisé défini lorsqu'un interpréteur est créé avec l'option -safe, comportent des alias pour les commandes **source**, **load** et **exit** et autorisent un sous-ensemble d'opérations pour les commandes **file** et **encoding**. L'interpréteur sécurisé peut aussi auto-charger du code lui-même et demander que des extensions soient chargées.

Du fait que certaines de ces commandes ont accès au système de fichiers local, il y a un risque potentiel de fuite d'information concernant la structure des répertoires. Pour éviter cela, les commandes qui prennent des noms de fichier comme argument utilisent des jetons symboliques à la place des noms véritables des répertoires. C'est l'interpréteur maître qui contrôle et opère la traduction des jetons symboliques en chemins véritables. Ce système de chemins virtuels est maintenu dans le maître pour tous les interpréteurs sécurisés créés par la commande **::safe::interpCreate** ou initialisés par la commande **::safe::interpInit** : les interpréteurs sécurisés n'obtiennent donc aucune information sur la structure réelle du système de fichiers de la machine hôte sur laquelle l'interpréteur maître est exécuté.

Les seuls noms de fichier valides servant d'arguments aux alias des commandes **source** et **load** et passés à l'interpréteur esclave seront de la forme :

[**file join** *jeton nomFichier*]

où *jeton* représente l'un des répertoires de la liste des chemins d'accès et où *NomFichier* est un fichier de ce répertoire (les accès aux sous-répertoires ne sont pas autorisés).

Lorsqu'un jeton symbolique est utilisé dans un interpréteur sécurisé pour sourcer ou charger un fichier, il est vérifié et traduit en un nom véritable par l'interpréteur maître mais l'interpréteur sécurisé ne connaît jamais le chemin réel.

Par mesure de précaution supplémentaire les fichiers auxquels l'alias de la commande **source** a accès sont encore soumis aux restrictions suivantes : le nom doit comporter au plus quatorze caractères, ne peut contenir qu'un seul point et doit avoir une extension *.tcl* ou bien être tclIndex.

Chaque élément de la liste des chemins d'accès initiale se verra attribuer un jeton symbolique qui sera installé dans la variable *auto_path* de l'interpréteur esclave et le premier élément de cette liste sera stocké dans la variable *tcl_library* de cet esclave.

Si l'argument chemin d'accès n'est pas spécifié ou bien si c'est une liste vide, le comportement par défaut consiste à donner accès pour l'interpréteur esclave aux

mêmes modules que ceux auxquels le maître a accès : pour être plus précis, il s'agit uniquement des modules écrits en Tcl (ce qui ne peut pas être dangereux puisque de toute façon ils seront exécutés dans l'interpréteur sécurisé) et aux extensions compilées dont le point d'entrée est de type sécurisé (*Safe_Init*). Pour cela, la variable *auto_path* du maître sera utilisée pour construire le chemin d'accès de l'esclave. Afin que l'esclave réussisse à charger les fichiers de bibliothèques Tcl (ce qui inclut le mécanisme d'autochargement lui-même), la valeur de la variable *tcl_library* sera ajoutée ou bien déplacée, si besoin est, en première position dans le chemin d'accès de l'esclave de telle sorte que la variable *tcl_library* de l'esclave soit la même que celle du maître (son nom réel sera toujours invisible pour l'esclave néanmoins). Pour que le mécanisme d'autochargement fonctionne dans l'esclave comme dans le maître dans cette situation par défaut, les sous-répertoires de premier niveau de chacun des répertoires de la variable *auto_path* du maître seront ajoutés au chemin d'accès de l'esclave. Il est toujours possible de spécifier un chemin plus restrictif dont certains sous-répertoires ne seront jamais visités en désignant explicitement la liste des répertoires au moyen de l'option **-accessPath** plutôt que de se reposer sur ce mécanisme par défaut.

Lorsque le chemin d'accès est modifié, après la création ou l'initialisation, en utilisant une commande
interpConfigure -accessPath *Liste*
une commande **auto_reset** est automatiquement évaluée dans l'interpréteur sécurisé pour synchroniser sa variable **auto_index** avec la nouvelle liste.

scan

La commande **scan** analyse des chaînes au moyen de spécificateurs de conversion.

Syntaxe

scan *Chaîne Format ?NomVar NomVar...?*

Introduction

Cette commande analyse les champs d'une chaîne passée en entrée, à la manière de la fonction *sscanf* du langage C : elle renvoie le nombre de conversions opérées ou bien la valeur -1 si la fin de la chaîne a été atteinte sans qu'aucune conversion n'ait pu être effectuée.

L'argument *Chaîne* désigne la chaîne à analyser et l'argument *ChaîneFormat* indique, aux moyens de spécificateurs introduits par le signe pourcentage (%), comment l'analyser. Chacun des arguments *NomVar* fournit le nom d'une variable : lorsqu'un champ de la chaîne est traité le résultat est transformé en une chaîne qui est affectée comme valeur de la variable correspondante. Si aucun nom de variable *NomVar* n'est spécifié, la commande **scan** produit directement en sortie les données qui seraient autrement stockées dans des variables. Dans ce cas, si aucune conversion n'a été opérée, la commande renvoie une chaîne vide.

Syntaxe de la commande *scan*

La commande **scan** opère simultanément sur les arguments *Chaîne* et *ChaîneFormat*. Si le prochain caractère rencontré dans *ChaîneFormat* est un caractère d'espacement (espace ou tabulation), il représente en réalité n'importe quel nombre d'espacements (éventuellement aucun) : autrement dit des espaces multiples seront assimilés à un espace unique. Si c'est un caractère autre que le signe pourcentage, il doit correspondre exactement au même caractère dans *Chaîne*. Lorsqu'un signe % est rencontré, il introduit un spécificateur de conversion. Un tel spécificateur peut contenir jusqu'à quatre champs différents :

- un astérisque indiquant que la valeur convertie doit être écartée au lieu d'être assignée à une variable ;
- un spécificateur de position ;
- un nombre entier indiquant une largeur de champ maximale ;
- un caractère de conversion.

Les trois premiers champs sont optionnels : seul le caractère de conversion est obligatoire. Lorsqu'ils sont présents, les champs doivent apparaître dans l'ordre qui vient d'être énoncé. Lorsque la commande **scan** trouve un spécificateur de conversion dans l'argument *ChaîneFormat*, il commence par ignorer les caractères d'espacement de la chaîne *Chaîne* (à moins que le spécificateur ne soit [ou **c**). Puis

il convertit les caractères suivants selon le spécificateur de conversion : le résultat est stocké dans la variable *NomVar*.

Si le symbole % est suivi à la fois d'un nombre entier n et d'un signe dollar ($) (comme par exemple `%2$d`), la variable à utiliser n'est pas prise séquentiellement dans la série des arguments *NomVar*. Il s'agit au contraire de la n-ième variable de cette liste. Si un spécificateur de position est utilisé dans le format *ChaîneFormat* alors tous les autres spécificateurs doivent aussi être de ce type. Chaque variable *NomVar* de la liste des arguments doit correspondre à exactement un spécificateur de conversion sous peine d'erreur ou bien, dans le cas où la commande est utilisée sans aucun nom de variable, une position ne doit être spécifiée qu'une fois au plus et les positions vides sont remplies par des chaînes vides.

Les caractères de conversion admis sont les suivants :

d
: Le champ en entrée doit être un nombre décimal. Il est lu et la valeur est stockée dans la variable comme une chaîne décimale.

o
: Le champ en entrée doit être un nombre octal. Il est lu et la valeur est stockée dans la variable comme une chaîne décimale.

x
: Le champ en entrée doit être un nombre hexadécimal. Il est lu et la valeur est stockée dans la variable comme une chaîne décimale.

u
: Le champ en entrée doit être un nombre décimal. Il est lu et la valeur est stockée dans la variable comme une chaîne d'entier décimal non signé.

i
: Le champ en entrée doit être un entier. La base (décimale, octale ou hexadécimale) est déterminée de la même manière qu'avec la fonction **expr** (cf. p. 57). La valeur est stockée dans la variable comme une chaîne décimale.

c
: Un caractère unique est lu et sa valeur binaire est stockée dans la variable comme chaîne décimale. Un éventuel espacement initial sera pris en compte, faisant que le champ en entrée peut être un caractère d'espacement. Cette conversion diffère du standard ANSI, le caractère en entrée doit être unique et aucune largeur de champ ne peut être spécifiée.

s
: Le champ en entrée est constitué de tous les caractères jusqu'au prochain caractère d'espacement. Les caractères sont copiés dans la variable.

e ou **f** ou **g**
: Le champ en entrée doit être un nombre en virgule flottante représenté par un éventuel signe suivi de chiffres contenant un éventuel point décimal et enfin d'un exposant optionnel qui est désigné par la lettre **e** ou **E** suivie d'un signe éventuel et d'une chaîne de nombres décimaux. Il est lu et stocké dans la variable comme une chaîne en virgule flottante.

[*Caractères*]
: Le champ en entrée comporte un nombre quelconque de caractères pris dans la série désignée par l'argument *Caractères*. La chaîne trouvée est stockée dans la variable. Si le premier caractère dans la paire de crochets est un crochet fermant, il est considéré comme faisant partie de la suite des caractères de l'argument *Caractères* et non pas comme un crochet qui referme la paire de crochets. Si l'argument *Caractères* contient une série de la forme *a*-*b* alors tout caractère dont le code se situe entre celui du caractère désigné par *a* et celui du caractère désigné par *b* pourra correspondre. Si le premier ou le dernier caractère de la paire de crochets est un tiret -, celui-ci sera traité comme un caractère ordinaire et non comme la marque d'un intervalle comme dans *a*-*b*.

[^ *Caractères*]
: Le champ en entrée comporte un nombre quelconque de caractères *n'appartenant pas* à la série désignée par l'argument *Caractères*. La chaîne trouvée est stockée dans la variable. Les mêmes règles particulières vues dans le cas précédent s'appliquent.

n
: Avec ce spécificateur aucun caractère de la chaîne en entrée n'est utilisé. C'est le nombre total de caractères traités dans la chaîne en entrée qui est stocké dans la variable.

▷ Le nombre de caractères lus pour une conversion sera toujours le maximum possible : le plus grand nombre de valeurs décimales si le spécificateur est **%d**, le plus grand nombre de valeurs octales si le spécificateur est **%o** etc. Le champ en entrée se termine soit avec un caractère d'espacement, soit lorsqu'une largeur maximale de champ a été spécifiée et est atteinte. Si un astérisque ∗ est utilisé dans le spécificateur alors aucune variable n'est utilisée et l'argument qui suit normalement dans la commande **scan** reste disponible pour les spécificateurs suivants.

seek

Modifie l'indice d'accès à un canal ouvert.

Syntaxe

seek *Canal Décalage?Origine?*

Description

Permet de modifier la position d'accès courante du canal *Canal*. La valeur *Canal* doit être un numéro d'identification de canal valide obtenu au moyen d'une commande telle que **open** ou **socket**.

L'argument *Décalage* spécifie la position où aura lieu le prochain accès en lecture ou en écriture sur ce canal. Il s'agit d'un nombre entier positif ou négatif correspondant à un nombre d'octets (et non pas de caractères comme c'est le cas avec la commande **read**). Ce décalage peut être calculé par rapport à une position particulière grâce à l'argument optionnel *Origine* qui prend l'une des valeurs suivantes :

start
: Décalage calculé par rapport au début. C'est la valeur par défaut.

current
: Décalage (positif ou négatif) calculé par rapport à la position courante.

end
: Décalage calculé par rapport à la fin. Dans ce cas, une valeur positive placera la position d'accès au-delà de la fin du fichier.

La commande **seek** renvoie une chaîne vide. Elle provoque une erreur si elle est appliquée à un fichier ou à un périphérique qui ne supporte pas le positionnement. Elle a par ailleurs pour effet de vider tout tampon de sortie du canal avant de retourner, même si le canal est en mode non-bloquant. Elle ignore en revanche tout tampon d'entrée qui n'aurait pas encore été lu.

set

Permet de lire ou de fixer la valeur d'une variable.

Syntaxe

set *NomVar?Valeur?*

Description

Si l'argument *Valeur* est précisé, la commande **set** renvoie la valeur de la variable *NomVar*, sinon elle fixe la valeur de la variable *NomVar* à *Valeur*, créant une nouvelle variable si celle-ci n'existe pas déjà et renvoyant sa valeur.

On peut appliquer cette commande de la même façon à une variable de type tableau (*array*).

Si aucun qualificateur d'espace de nom n'est spécifié, la variable que l'on crée ou dont on fixe la valeur est définie dans l'espace courant. Dans le cas contraire, il s'agira d'une variable de ce nom dans l'espace désigné. La portée d'une telle variable est limitée à la procédure dans laquelle elle est déclarée, à moins qu'elle n'ait été déclarée globale au moyen de la commande **global** ou bien qu'elle ait été étendue à tout un espace de noms grâce à la commande **variable**.

socket

La commande **socket** établit une connexion de réseau TCP.

Syntaxe

socket *?Options? Hôte Port*
socket -server *Commande?Options? Port*

Description

Cette commande ouvre une connexion de réseau et renvoie un identificateur de canal c'est-à-dire un nombre qui permettra par la suite de se référer, dans des appels à certaines sous-commandes ou commandes telles que **read**, **puts** et **flush**, au canal en question.

Actuellement seul le protocole de communication TCP est reconnu. La commande **socket** peut être utilisée pour ouvrir aussi bien le côté client que le côté serveur de la connexion, suivant que l'option **-server** est spécifiée ou pas.

Connexions client

Si l'option **-server** n'est pas spécifiée, c'est le côté client d'une connexion de réseau qui est ouvert et la commande **socket** renvoie un identificateur qui peut être utilisé à la fois pour la lecture et pour l'écriture.

Les arguments *Port* et *Hôte* spécifient le port auquel se connecter ; il doit exister un serveur acceptant des connexions sur ce port. L'argument *Port* est un nombre entier et *Hôte* est soit un nom de domaine tel que *www.hugo.com*, soit une adresse IP telle que *123.45.67.890*. On peut aussi utiliser le mot *localhost* pour désigner l'hôte sur lequel la commande a été invoquée.

Des informations supplémentaires concernant la connexion peuvent être spécifiées par les options suivantes :

-myaddr *adr*
: L'argument *adr* donne le nom de domaine ou l'adresse IP de l'interface de réseau côté client à utiliser pour cette connexion. Cette option est utile si la machine cliente a plusieurs interfaces réseau. Si cette option est omise, l'interface côté client sera choisie par le système.

-myport *port*
: L'argument *port* désigne un numéro de port à utiliser pour le côté client de la connexion. Si cette option est omise, un numéro de port sera choisi de façon aléatoire par le système.

-async
: L'option **-async** permet que la connexion client se fasse de manière asynchrone. Cela signifie que le canal de connexion (*socket*) sera créé immédiatement mais que la connexion au serveur peut ne pas être encore établie au

moment où la commande **socket** retourne. Lorsqu'une commande **gets** ou **flush** est invoquée sur le canal avant que la tentative de connexion effective ait réussi ou échoué, si le canal est en mode bloquant, l'opération patientera jusqu'à ce que la tentative de connexion se termine (que ce soit par un succès ou par un échec). En mode non-bloquant, si une commande **gets** ou **flush** concernant le canal est invoquée avant que la tentative de connexion ne s'achève, l'opération retourne immédiatement : dans ce cas, une commande **fblocked** sur le canal renverra la valeur 1 (cf. p 63).

Connexions serveur

Si l'option **-server** est spécifiée, c'est le côté serveur d'une connexion qui est ouvert sur le port désigné par l'argument *Port*. Tcl acceptera automatiquement les connexions à ce port. Pour chaque connexion, Tcl crée un nouveau canal qui peut être utilisé pour communiquer avec le client. Tcl invoque alors la commande *Commande* avec trois arguments additionnels : le nom du nouveau canal, l'adresse de réseau de l'hôte du client et le numéro de port du client.

On peut par ailleurs spécifier les options suivantes avant l'argument *Hôte* :

-myaddr *adr*
: L'argument *adr* donne le nom de domaine ou l'adresse IP de l'interface de réseau côté serveur à utiliser pour cette connexion. Cette option est utile si la machine serveur a plusieurs interfaces réseau. Si cette option est omise, le canal serveur sera lié à l'adresse spéciale INADDR_ANY de façon à pouvoir accepter des connexions depuis n'importe quelle interface.

Les canaux serveurs ne peuvent pas être utilisés pour l'entrée et la sortie ; leur seule fonction est d'accepter de nouvelles connexions clients. Les canaux créés pour chaque connexion client sont ouverts en entrée et en sortie. La fermeture du canal serveur conduit à l'extinction du serveur de telle sorte qu'aucune nouvelle connexion ne peut plus être acceptée ; en revanche les connexions déjà établies ne sont pas affectées.

Les canaux serveurs dépendent du mécanisme de gestion des événements de Tcl pour savoir si de nouvelles connexions sont ouvertes. Si l'application n'entre pas dans la boucle d'événements (par utilisation de la commande **vwait** ou par invocation de la fonction **Tcl_DoOneEvent** dans du code C), aucune connexion ne sera acceptée.

Options de configuration

La commande **fconfigure** (cf. p. 64) peut être utilisée pour s'informer sur un certain nombre d'options de configuration concernant les canaux de connexion. Les options définies sont les suivantes :

-error
: Cette option lit l'état courant des erreurs pour un canal donné. Cela peut être utile pour déterminer si une opération de connexion asynchrone a réussi.

S'il y a eu une erreur, le message d'erreur est renvoyé par cette option. Sinon la valeur de retour est une chaîne vide.

-sockname
Cette option renvoie une liste de trois éléments: l'adresse, le nom d'hôte et le numéro de port du canal. Si le nom d'hôte ne peut être établi, le second élément sera identique au premier, c'est-à-dire à l'adresse.

-peername
Cette option n'est pas supportée par les canaux serveurs. Pour les canaux clients, l'option renvoie une liste de trois éléments: l'adresse, le nom d'hôte et le numéro de port auquel le canal (*peer socket*) est connecté ou lié. Si le nom d'hôte ne peut être établi, le second élément sera identique au premier, c'est-à-dire à l'adresse.

source

La commande **source** évalue un fichier ou une ressource en tant que script Tcl.

Syntaxe

source *NomFichier*
source -rsrc *NomRessource ?NomFichier?*
source -rsrcid *IdRessource ?NomFichier?*

Description

Cette commande passe le contenu du fichier ou de la ressource spécifiés à l'interpréteur Tcl afin qu'il l'exécute. La valeur de retour de la commande **source** est celle de la dernière instruction exécutée dans le script. Si une erreur se produit au cours de l'exécution, elle est transmise par la commande. Si une commande **return** est rencontrée, les commandes suivantes sont ignorées et la valeur de retour de la commande **source** est celle de cette commande **return**.

Le caractère de fin de fichier est `\32` (`^Z`) pour toutes les plates-formes. La commande lit le fichier jusqu'à ce caractère. Si le caractère `^Z` est nécessaire dans le script, il faudra le représenter comme `\032` ou bien `\u001a`. Cette restriction n'existe pas avec les commandes **read** et **gets** ce qui permet de lire des fichiers contenant du code binaire aussi bien que des données.

Les syntaxes avec les options **-rsrc** et **-rsrcid** sont disponibles uniquement sur MacOS où tout fichier possède à la fois une branche dite de données et une branche dite de ressources. Elles permettent de sourcer des scripts stockés sous forme de ressources de type TEXT : une telle ressource peut être désignée soit par son nom, soit par son numéro d'identification. L'argument *NomFichier* désigne le fichier qui possède la ressource en question ; s'il n'est pas spécifié, Tcl recherchera dans tous les fichiers de ressources ouverts, ce qui inclut l'application courante et toutes les bibliothèques C éventuellement chargées.

split

La commande **split** scinde une chaîne en une liste Tcl.

Syntaxe

split *Chaîne ?Scission?*

Description

Cette commande renvoie une liste créée en scindant la chaîne *Chaîne* en chacun des caractères indiqués dans l'argument *Scission*. Les caractères où se fait la scission disparaissent et la liste est donc constituée de ceux qui se trouvent entre deux points de scission successifs. Des éléments de liste vides seront créés si la chaîne *Chaîne* contient des points de scission adjacents ou si elle commence ou finit par un tel point. Si l'argument est une chaîne vide alors chaque caractère de la chaîne *Chaîne* devient un élément de la liste ainsi créée. Si l'argument *Scission* n'est pas spécifié, la chaîne sera scindée en chaque symbole d'espacement.

string

La commande **string** permet d'accomplir toutes les opérations de manipulation de chaînes de caractères.

Syntaxe

string *Sous-commande Arg ?Arg...?*

Description

Les différentes opérations à exécuter sont déterminées par la *sous-commande* utilisée. Ces sous-commandes peuvent être abrégées si cela ne présente pas d'ambiguïté. On dispose ainsi des sous-commandes suivantes :

string bytelength *Chaîne*
: Renvoie une valeur décimale correspondant au nombre d'octets utilisés pour représenter la chaîne *Chaîne* en mémoire. Cette valeur ne coïncide pas forcément avec la longueur de la chaîne en caractères étant donné que l'encodage Unicode UTF-8 code certains caractères sur plusieurs octets. Pour avoir la longueur de la chaîne en nombre de caractères, on utilisera la commande **string length**.

string compare?-**nocase**?-**length** *nb*? *Chaîne1 Chaîne2*
: Effectue une comparaison caractère par caractère des chaînes *Chaîne1* et *Chaîne2*. Renvoie -1, 0, ou 1, suivant que la chaîne *Chaîne1* est lexicographiquement inférieure, égale ou supérieure à la chaîne *Chaîne2*.
Si l'option **-length** est spécifiée, seulement les *nb* premiers caractères sont pris en compte dans la comparaison. Si l'option **-nocase** est spécifiée, la comparaison se fait indépendamment de la casse des caractères autrement dit sans distinction de minuscules et de majuscules.

string equal?-**nocase**?-**length** *nb? Chaîne1 Chaîne2*
: Effectue une comparaison caractère par caractère des chaînes *Chaîne1* et *Chaîne2*. Renvoie 1 si les chaînes *Chaîne1* et *Chaîne2* sont identiques, 0 sinon. Les options **-nocase** et **-length** ont la même signification qu'avec la commande **string compare** vue précédemment.

string first *Chaîne1 Chaîne2 ?IndexDébut?*
: Recherche dans la chaîne *Chaîne2* une suite de caractères correspondant exactement aux caractères de *Chaîne1*. Si une correspondance est trouvée, l'indice du premier caractère trouvé dans la chaîne *Chaîne2* est renvoyé. Autrement la valeur de retour sera -1.
Si l'argument*IndexDébut* est spécifié (sous l'une des formes acceptées par la commande **index**), la recherche démarre au caractère de la chaîne *Chaîne2* situé à cet indice. Par exemple,
`string first a 0a23456789abcdef 5`

renverra la valeur 10 tandis que

```
string first a 0123456789abcdef 11
```

renverra -1.

string index *Chaîne n*

Renvoie le n-ième caractère de la chaîne *Chaîne*. L'indice 0 correspond au premier caractère de la chaîne. L'indice n peut être spécifié de diverses manières :

- au moyen d'un simple nombre entier ;
- au moyen du mot-clé *end* pour désigner le dernier caractère de la chaîne ;
- au moyen d'une expression « end-n » pour désigner un décalage de n caractères à partir de la fin. Par exemple end-1 correspond à la lettre *c* dans *abcd*.

Si n est négatif ou bien supérieur à la longueur de la chaîne (en nombre de caractères), la valeur de retour est la chaîne vide.

string is *Classe*?**-strict**??**-failindex** *NomVar*? *Chaîne*

Renvoie 1 si *Chaîne* est un membre valide de la classe de caractères spécifiée, sinon 0.

Avec l'option **-strict** une chaîne vide renverra la valeur 0, sinon 1. Si l'option **-failindex** est utilisée et si la commande renvoie 0, l'indice de la chaîne où la classe n'est plus valide sera stocké dans la variable *NomVar* ; si la fonction renvoie 1, la variable *NomVar* ne recevra aucune valeur. Les classes reconnues sont rassemblées dans le tableau 6.

Pour les classes **double** et **integer**, en cas de dépassement (valeur trop grande ou trop petite) la commande renvoie 0 et la variable *NomVar* contiendra -1.

Pour les classes **boolean**, **true** et **false**, si la valeur de retour de la commande est 0, la variable *NomVar* sera mise à zéro 0 étant données les différentes façons de représenter une valeur booléenne.

string last *Chaîne1 Chaîne2 ?IndexDébut?*

Recherche dans la chaîne *Chaîne2* une suite de caractères correspondant exactement aux caractères de *Chaîne1*. En cas de succès, la valeur de retour sera l'indice du premier caractère de la *dernière* correspondance trouvée dans *Chaîne2*. Autrement la valeur de retour sera -1.

Si l'argument*IndexDébut* est spécifié (sous l'une des formes acceptées par la commande **index**), la recherche se fera uniquement sur les caractères situés avant cette position (position incluse). Par exemple,

```
string last a 0a23456789abcdef 15
```

renverra la valeur 10 tandis que

```
string last a 0a23456789abcdef 9
```

renverra 1.

string length *Chaîne*

Renvoie le nombre de caractères dans la chaîne *Chaîne*. Cette valeur peut différer du nombre d'octets servant à représenter cette chaîne étant donné que l'encodage Unicode UTF-8 code certains caractères sur plusieurs octets.

alnum	Tout caractère Unicode alphanumérique
alpha	Tout caractère Unicode alphabétique
ascii	Tout caractère de valeur inférieure à u0080 (de 0 à 127)
boolean	Tout terme admis pour désigner une valeur booléenne
control	Tout caractère de contrôle Unicode
digit	Tout caractère numérique Unicode
double	Toutes valeurs de type *double* dans Tcl
false	Tout terme admis pour désigner une valeur booléenne fausse
graph	Tout caractère imprimable Unicode à l'exception des espacements
integer	Toutes valeurs de type *entier* dans Tcl
lower	Tout caractère Unicode minuscule
print	Tout caractère imprimable Unicode y compris les espacements
punct	Tout caractère Unicode de ponctuation
space	Tout caractère Unicode d'espacement
true	Tout terme admis pour désigner une valeur booléenne vraie
upper	Tout caractère Unicode majuscule
wordchar	Tout caractère Unicode de mot (alphanumérique ou soulignement)
xdigit	Tout caractère hexadécimal ([0-9A-Fa-f])

TAB. 6 – *Classes de caractères reconnues par Tcl.*

string map?**-nocase**? *TableCars Chaîne*

Remplace les caractères de la chaîne *Chaîne* suivant les correspondances clé-valeur contenues dans la table *TableCars*. Cette table est une liste de type *clé valeur clé valeur...* analogue à celles obtenues avec la commande **array get**. Les clés et les valeurs peuvent être des caractères multiples. Chaque instance d'une clé est remplacée par la valeur correspondante. Si l'option **-nocase** est spécifiée, la comparaison se fait indépendamment de la casse des caractères autrement dit sans distinction de minuscules et de majuscules.

Les remplacements se font de manière ordonnée: la première clé de la liste sera examinée d'abord et ainsi de suite. La chaîne est parcourue une fois à chaque itération ce qui fait que des remplacements de clés antérieurs ne sont pas affectés par de nouvelles correspondances. Par exemple,

```
string map {abc 1 ab 2 a 3 1 0} 1abcaababcababababc
```

renverra la chaîne 01321221.

string match?**-nocase**? *Motif Chaîne*
Vérifie si le motif *Motif* correspond à la chaîne *Chaîne*. La valeur de retour est 1 s'il y a correspondance, sinon 0. Avec l'option **-nocase**, la correspondance sera recherchée sans distinction de minuscules et de majuscules.
Le motif peut contenir des métacaractères. Ceux-ci sont expliqués en détail à la section *Syntaxe des motifs* p. .

string range *Chaîne Premier Dernier*
Renvoie les caractères de la chaîne *Chaîne* situés entre l'indice *Premier* et l'indice *Dernier*. L'indice correspond au premier caractère de la chaîne. Ces indices *Premier* et *Dernier* peuvent être spécifiés selon la syntaxe admise par la commande **index**. Si *Premier* est négatif il est considéré comme 0, et si *Dernier* excède la longueur de la chaîne il est assimilé à **end**. Si *Premier* est supérieur à *Dernier* une chaîne vide est renvoyée.

string repeat *Chaîne nb*
Renvoie la chaîne *Chaîne* dupliquée *nb* fois.

string replace *Chaîne Premier Dernier ?NouvelleChaîne?*
Supprime les caractères de la chaîne *Chaîne* situés entre l'indice *Premier* et l'indice *Dernier*. Si une chaîne *NouvelleChaîne* est spécifiée, elle est substituée aux caractères supprimés.
Les règles concernant les indices *Premier* et *Dernier* sont les mêmes que pour la commande **string range** vue précédemment. Si *Premier* est supérieur à *Dernier* ou à la longueur de la chaîne initiale, ou bien si *Dernier* est négatif, la chaîne initiale est renvoyée inchangée.

string tolower *Chaîne ?Premier? ?Dernier?*
Renvoie une chaîne dans laquelle toutes les majuscules de la chaîne *Chaîne* ont été remplacées par des minuscules. Si l'argument *Premier* est spécifié, il désigne l'indice à partir duquel opérer cette modification. De même si l'indice *Dernier* est spécifié, il désigne inclusivement le caractère jusqu'où opérer la modification. *Premier* et *Dernier* peuvent être spécifiés selon la syntaxe admise par la commande **index**.

string totitle *Chaîne ?Premier? ?Dernier?*
Renvoie une chaîne dans laquelle la première lettre de la chaîne *Chaîne* a été remplacée par sa variante Unicode *titre* (ou mise en majuscule s'il n'y a pas de variante *titre*) et toutes les lettres suivantes sont mises en minuscules. Les règles concernant les indices *Premier* et *Dernier* sont les mêmes que pour la commande **string tolower** vue précédemment.

string toupper *Chaîne ?Premier? ?Dernier?*
Renvoie une chaîne dans laquelle toutes les minuscules de la chaîne *Chaîne* ont été remplacées par des majuscules. Les règles concernant les indices *Premier* et *Dernier* sont les mêmes que pour la commande **string tolower** vue précédemment.

string trim *Chaîne ?Caractères?*
Renvoie une chaîne dans laquelle tous les caractères désignés par l'argument *Caractères* et figurant à gauche et à droite de la chaîne *Chaîne* ont été sup-

primés. Si l'argument *Caractères* n'est pas spécifié, tous les caractères d'espacement sont supprimés à gauche et à droite de la chaine (espaces, tabulations, sauts de lignes et retours-chariots).

string trimleft *Chaîne ?Caractères?*
Renvoie une chaîne dans laquelle tous les caractères désignés par l'argument *Caractères* et figurant à gauche uniquement de la chaîne *Chaîne* ont été supprimés. Si l'argument *Caractères* n'est pas spécifié, tous les caractères d'espacement sont supprimés à gauche de la chaine (espaces, tabulations, sauts de lignes et retours-chariots).

string trimright *Chaîne ?Caractères?*
Renvoie une chaîne dans laquelle tous les caractères désignés par l'argument *Caractères* et figurant à droite uniquement de la chaîne *Chaîne* ont été supprimés. Si l'argument *Caractères* n'est pas spécifié, tous les caractères d'espacement sont supprimés à droite de la chaine (espaces, tabulations, sauts de lignes et retours-chariots).

string wordend *Chaîne IndexCaractère*
Renvoie l'indice du caractère qui suit le mot contenant le caractère d'indice *IndexCaractère* dans la chaîne *Chaîne*. L'indice *IndexCaractère* peut être spécifié selon les conventions admises par la commande **index**.
Par convention, un mot est toute suite contiguë de caractères alphanumériques (au sens Unicode) ou de caractères de soulignement _ (désigné comme *ponctuation de connexion* dans la base Unicode), ou bien n'importe quel autre caractère isolé.

string wordstart *Chaîne IndexCaractère*
Renvoie l'indice du premier caractère du mot contenant le caractère d'indice *IndexCaractère* dans la chaîne *Chaîne*. L'indice *IndexCaractère* peut être spécifié selon la syntaxe admise par la commande **index**. La notion de mot est la même qu'avec la commande **string wordend** vue précédemment.

subst

La commande **subst** effectue des substitutions de variables, de commandes et de contre-obliques dans une chaîne.

Syntaxe

subst?-**nobackslashes**?-**nocommands**?-**novariables**? *Chaîne*

Description

Cette commande effectue toutes les substitutions possibles de commandes, de variables et de contre-obliques dans l'argument *Chaîne*. Elle renvoie la chaîne résultant de toutes ces substitutions. Les substitutions sont opérées comme avec les commandes Tcl, la chaîne *Chaîne* subissant deux séries de substitutions : celle de l'interpréteur Tcl puis celle de la commande **subst**. La commande renvoie une erreur en cas d'échec.

Si les options **-nobackslashes**, **-nocommands** et **-novariables** sont utilisées alors respectivement les substitutions de contre-obliques, de commandes et de variables respectivement ne sont pas opérées. L'option **-nocommands** par exemple aura pour effet que les paires de crochets seront traitées comme des caractères ordinaires et non pas comme les délimiteurs d'une commande imbriquée.

Les guillemets et les accolades n'ont pas de signification particulière pour la commande **subst**. Par exemple, le script

```
set a 44
subst {xyz {$a}}
```

renverra `xyz {44}` et non pas `xyz {$a}`.

On notera qu'une substitution d'une sorte peut inclure des substitutions d'une autre sorte. Par exemple, même si l'option **-novariables** est spécifiée, la substitution de commandes est opérée sans restrictions. Cela signifie que toute substitution de variables nécessaire pour que la substitution de commande puisse se faire sera effectuée. De la même manière, toute substitution de commandes nécessaire pour exécuter une substitution de variables sera effectuée même si l'option **-nocommands** a été spécifiée.

Si une erreur se produit au cours d'une substitution, la commande **subst** renvoie cette erreur. Si une exception de type *break* se produit pendant une substitution de variables ou de commandes, le résultat de toute la substitution sera la chaîne substituée obtenue jusqu'au moment où l'exception a été levée. Si une exception de type *continue* se produit pendant l'évaluation d'une substitution de variables ou de commandes, une chaîne vide sera substituée pour cette substitution de commande ou de variable. Si une exception de type *return* se produit ou si un code de retour est renvoyé pendant une substitution, c'est la valeur renvoyée qui est substituée pour cette substitution. Tous les codes de retour exceptionnels sont ainsi capturés

par la commande **subst**. Cela permet un contrôle plus fin des substitutions. En voici quelques exemples :

```
subst {abc,[break],def}
```

renvoie "abc," et non pas "abc,,def"

```
subst {abc,[continue;expr 1+2],def}
```

renvoie "abc,,def" et non pas "abc,3,def"

```
subst {abc,[return xyz;expr 1+2],def}
```

renvoie "abc,xyz,def" et non pas "abc,3,def"

```
subst {abc,[return -code 10 xyz;expr 1+2],def}
```

renvoie "abc,xyz,def" et non pas "abc,3,def".

switch

La commande **switch** évalue différents scripts en fonction de la valeur d'une certaine variable.

Syntaxe

switch *?Options? Chaîne Motif Corps?Motif Corps...?*
switch *?Options? Chaîne* {*Motif Corps?Motif Corps...?*}

Description

La commande compare son argument *Chaîne* successivement à chacun des arguments *Motif*: si elle trouve une correspondance alors elle exécute le script associé au motif trouvé. Une fois ce script terminé la commande **switch** s'interrompt et transmet comme valeur de retour la valeur de retour de ce script. Elle ne cherche pas de correspondance entre la chaîne *Chaîne* et les motifs qui suivent éventuellement. Si aucun motif ne correspond à la chaîne et si le dernier motif s'appelle **default**, alors le script qui lui est associé est exécuté. S'il n'y a pas de terme **default**, la commande se termine en renvoyant une chaîne vide.

On peut préciser des options qui permettent de spécifier la manière dont doit se faire la recherche de correspondance. Les options disponibles sont les suivantes:

-exact
: L'argument *Chaîne* doit correspondre littéralement à l'argument *Motif*, caractère par caractère.

-glob
: Cette option signifie que le motif utilise des métacaractères selon la syntaxe décrite avec la commande **string match** (cf. p. 174).

-regexp
: Cette option signifie que le motif utilise des métacaractères selon la syntaxe des expressions régulières décrite avec les commandes **regexp** et **regsub** (voir au chapitre *Les expressions régulières*).

-- Un double tiret signale la fin des options.

On peut associer plusieurs motifs à un même script en les séparant par de simples tirets.

Il existe deux syntaxes différentes, et non équivalentes, pour les arguments *Motif* et *Corps*. La première utilise un argument séparé pour chacun des motifs et chacune des commandes associées: cette forme autorise les substitutions et interpolations dans les motifs et les commandes. La seconde forme réunit tous les motifs et commandes en un unique argument qui doit être une liste Tcl valide dont les éléments seront précisément les motifs et les commandes alternativement. Dans cette forme, les accolades qui entourent la liste entière empêchent que des substitutions ne soient opérées par l'interpréteur.

Enfin on notera que les commentaires ne sont pas admis à l'intérieur d'une commande **switch** entre les motifs et les blocs de commandes et provoqueront des erreurs de la part de l'interpréteur. On peut placer des commentaires uniquement au sein des blocs de commandes.

Variables globales

Description

Les variables globales suivantes sont créées et gérées automatiquement par Tcl. Il est en général prudent, sauf mention contraire, de ne pas modifier les valeurs de ces variables et de ne les utiliser qu'en lecture seule.

env

Il s'agit d'une variable de type *array* dont les éléments sont les variables d'environnement pour le processus. La valeur de cette variable donne la valeur de la variable d'environnement correspondante. Par exemple :

```
set env(SHELL)
```

⇝ */bin/bash*

Le fait de modifier la valeur d'un élément de **env** modifiera en conséquence la variable d'environnement correspondante ou la créera si elle n'existe pas déjà. Supprimer un élément du tableau **env** supprimera la variable d'environnement correspondante. Si le tableau **env** entier est supprimé par une commande **unset**, Tcl cessera de le gérer et de tenir à jour les variables d'environnement correspondantes.
L'instruction suivante permet d'obtenir, sur la console de tclsh ou de wish, les clés et les valeurs des éléments du tableau **env** :

```
foreach var [array names env] {
    puts "$var: $env($var)"
}
```

Les résultats de cette commande seront bien entendu différents suivant la plate-forme utilisée. Sous MacOS, il n'y a pas de variables d'environnement et le tableau *env* est construit et géré directement par Tcl. On peut créer ses propres variables d'environnement sur Macintosh en procédant de la manière suivante : il faut créer un fichier appelé *Tcl Environment Variables* que l'on place dans le sous-dossier Préférences du dossier Système. Chaque ligne de ce fichier définira une variable d'environnement selon le modèle suivant :
Nom_var=valeur
Une alternative, si l'on sait manipuler les branches de ressources des fichiers, consiste à créer pour l'application une ressource de type STR# que l'on nommera *Tcl Environment Variables* et donc le contenu suivra le même modèle que le fichier de même nom mentionné précédemment.

errorCode

À la suite d'une erreur, cette variable sera fixée pour contenir de l'information complémentaire sur cette erreur sous une forme qui soit facile à utiliser dans les programmes. La variable **errorCode** consiste en une liste d'un ou

plusieurs éléments : le premier identifie une classe générale pour l'erreur qui détermine le format de la suite de la liste. Les formats suivants sont définis par défaut (mais des applications particulières peuvent aussi définir leurs propres formats) :

ARITH *code msg*
: Ce format est utilisé lorsqu'une erreur arithmétique se produit (par exemple une division par 0 dans la commande **expr**). L'argument *code* identifie l'erreur précise et *msg* fournit une description circonstanciée de l'erreur. L'argument *code* sera DIVZERO (pour une tentative de division par 0), DOMAIN (si un argument est en-dehors du domaine de définition d'une fonction, comme par exemple acos(-3)), IOVERFLOW (pour des dépassements de capacité en nombres entiers), OVERFLOW (pour des dépassements de capacité en virgule flottante), ou UNKNOWN (en cas de cause indéterminée).

CHILDKILLED *pid nomSignal msg*
: Ce format est utilisé lorsqu'un processus enfant a été terminé à la suite d'un signal. Le second élément de **errorCode** sera l'identificateur décimal du processus. Le troisième élément sera le nom symbolique du signal qui a terminé le processus comme par exemple SIGPIPE. Le quatrième élément sera un message bref décrivant le signal.

CHILDSTATUS *pid code*
: Ce format est utilisé lorsqu'un processus enfant a quitté avec une valeur de retour non nulle. Le second élément sera l'identificateur décimal du processus. Le troisième élément sera le code décimal de sortie renvoyé par le processus.

CHILDSUSP *pid nomSignal msg*
: Ce format est utilisé lorsqu'un processus enfant a été suspendu à la suite d'un signal. Le second élément de **errorCode** sera l'identificateur décimal du processus. Le troisième élément sera le nom symbolique du signal qui a terminé le processus comme par exemple SIGTTIN. Le quatrième élément sera un message bref décrivant le signal.

NONE
: Ce format est utilisé pour les erreurs pour lesquelles aucune information supplémentaire n'est disponible. Dans ce cas la liste de **errorCode** contiendra uniquement le mot **NONE**.

POSIX *nomErreur msg*
: Si le premier élément de **errorCode** est **POSIX**, l'erreur s'est produite au cours d'un appel POSIX au noyau. Le second élément de la liste contiendra le nom symbolique de l'erreur, comme par exemple ENOENT. Le troisième élément sera une brève description de cette erreur.

Pour fixer la variable **errorCode** elle-même, les applications Tcl devront invoquer la commande **error** (cf. p 49) ou bien, dans du code C, utiliser des procédures telles que **Tcl_SetErrorCode** et **Tcl_PosixError**.

errorInfo

À la suite d'une erreur, cette variable contiendra une ou plusieurs lignes identifiant les procédures et les commandes qui ont été exécutées lorsque l'erreur la plus récente s'est produite. Cette information prend la forme d'une trace de la pile d'exécution montrant les divers appels de commandes imbriquées invoqués au moment de l'erreur.

tcl_library

Cette variable contient le nom du répertoire contenant les scripts de la bibliothèque standard de Tcl comme par exemple ceux qui sont utilisés par le mécanisme de chargement automatique de procédures. La valeur de cette variable peut être obtenue au moyen de la commande **info library**.

Très souvent les applications ou les modules d'extension possèdent leur propre bibliothèque de scripts. Ils devraient normalement définir une variable globale portant un nom de la forme **$***app***_library** (où *app* est le nom de l'application) afin de contenir le nom de fichier de réseau pour le répertoire contenant cette bibliothèque. La valeur initiale de **tcl_library** est fixée, lorsqu'un interpréteur est créé, en recherchant à travers divers répertoires un script de démarrage approprié. Si la variable d'environnement TCL_LIBRARY existe, le répertoire qu'elle désigne est visité en premier. Si cette variable n'est pas fixée ou que son répertoire ne convient pas, Tcl visite quelques autres répertoires (qui varient en fonction de la plate-forme), en particulier le répertoire contenant l'application et le répertoire de travail courant.

tcl_patchLevel

Quand un interpréteur est créé, Tcl initialise cette variable avec une chaîne indiquant le numéro de développement courant de Tcl, comme par exemple *7.3p2* pour Tcl 7.3 avec les deux premiers rectificatifs (*patches*) officiels ou bien *7.4b4* qui désigne la quatrième version bêta de Tcl 7.4. La valeur de cette variable peut être obtenue au moyen de la commande **info patchlevel**.

tcl_pkgPath

Cette variable contient une liste de répertoires dans lesquels les modules d'extension sont normalement installés. Elle n'est pas utilisée sous Windows. Elle contient typiquement un ou deux éléments. S'il y a deux éléments, le premier est normalement un répertoire pour les extensions qui dépendent de la plate-forme (comme les bibliothèques partagées) et le second est un répertoire pour les extensions qui ne dépendent pas du système d'exploitation (comme les bibliothèques de scripts).

Typiquement une extension est installée comme sous-répertoire de l'un de ces deux répertoires. Par défaut, ces répertoires sont inclus dans la variable **auto_path** de telle sorte qu'ils soient visités, de même que leurs sous-répertoires immédiats, lors de l'exécution d'une commande **package require**.

La variable **tcl_pkgPath** ne devrait pas être modifiée par une application. Sa valeur est ajoutée à **auto_path** au moment du démarrage et des modifications ultérieures ne seront pas prises en compte. Si l'on veut ajouter des répertoires à visiter, il vaut mieux les ajouter directement à la variable **auto_path**.

tcl_platform

Il s'agit d'une variable de type *array*, tableau associatif dont les éléments contiennent des informations concernant la plate-forme sur laquelle l'application est exécutée. Les éléments mentionnés ci-dessous seront toujours présents mais leurs valeurs peuvent être parfois des chaînes vides si Tcl n'a pas réussi à obtenir l'information. Les extensions et les applications peuvent ajouter des éléments qui leur sont propres à ce tableau. Les éléments prédéfinis sont :

byteOrder
L'ordre des octets de poids faible et fort sur cette machine : soit *littleEndian*, soit *bigEndian*.

debug
Si cette variable existe, l'interpréteur a été compilé pour générer des symboles de débogage. Cette variable existe uniquement sous Windows afin que les développeurs puissent spécifier quel module charger en fonction de la bibliothèque C utilisée au moment de l'exécution.

machine
Le jeu d'instructions exécuté par cette machine, comme par exemple *intel*, *PPC*, *68k* ou *sun4m*. Sous Unix, c'est la valeur renvoyée par la commande `uname -m`.

os
Le nom du système d'exploitation en vigueur sur la machine, comme par exemple *Windows 95*, *Windows NT*, *MacOS* ou *SunOS*. Sur les machines Unix, c'est la valeur renvoyée par la commande `uname -s`. Sur Windows 95 et Windows 98, la valeur renvoyée sera toujours Windows 95 pour des raisons de compatibilité ; pour faire le distinguo entre les deux, il faut regarder la valeur correspondant à la clé *osVersion*.

osVersion
Le numéro de version du système d'exploitation en vigueur sur la machine. Sur les machines Unix, c'est la valeur renvoyée par la commande `uname -r`. Sur Windows 95, la version sera 4.0 et sur Windows 98, ce sera 4.10.

platform
Renvoie l'une des valeurs *windows*, *macintosh* ou *unix*. Cela permet d'identifier le système d'exploitation général utilisé sur la machine.

threaded
Si cette variable existe, l'interpréteur a été compilé en autorisant les *threads*.

user
Cette variable identifie l'utilisateur courant, en se basant sur l'information de login disponible sur la plate-forme. Sa valeur provient des variables d'environnement USER ou LOGNAME sous Unix et de la fonction GetUserName sur MacOS et Windows.

tcl_precision
Cette variable contrôle le nombre de chiffres à générer pour convertir des

valeurs en virgule flottante en chaînes. La valeur par défaut est 12. Une valeur de 17 chiffres est une valeur parfaite pour les virgules flottantes IEEE, elle permet que les valeurs en double précision soient converties en chaînes et réciproquement sans perte d'information. Néanmoins, l'utilisation de 17 chiffres empêche de faire des arrondis et produit des résultats plus longs et moins intuitifs. Par exemple, `expr 1.4` renvoie 1.3999999999999999 si **tcl_precision** vaut 17 et 1.4 si elle vaut 12.

Tous les interpréteurs dans un même processus partagent une unique valeur pour **tcl_precision** : si on la change dans un interpréteur, tous les autres seront aussi bien affectés. Cependant, les interpréteurs sécurisés ne sont pas autorisés à modifier cette variable.

tcl_rcFileName

Cette variable est utilisée au cours de l'initialisation pour indiquer le nom d'un fichier de démarrage contenant les préférences spécifiques d'un utilisateur. Si elle est fixée par une initialisation spécifique à une application le code Tcl vérifiera l'existence de ce fichier et en chargera le code s'il le trouve. Par exemple, pour wish cette variable a la valeur ~/.wishrc sous Unix et ~/wishrc.tcl sous Windows. Sous MacOS, elle n'est pas définie.

tcl_rcRsrcName

Cette variable est utilisée sur les systèmes Macintosh, au cours de l'initialisation, pour indiquer le nom d'une ressource de type TEXT située dans la branche de ressources de l'application ou de l'extension. Si elle est fixée par une initialisation spécifique à une application alors le code Tcl vérifiera l'existence de cette ressource et en chargera le code s'il la trouve. Par exemple, avec wish pour Macintosh cette variable a la valeur *tclshrc*.

tcl_traceCompile

La valeur de cette variable peut être fixée pour déterminer la quantité d'information dont il faut garder la trace durant la compilation du code binaire (*bytecode*). Par défaut, **tcl_traceCompile** a la valeur 0 et aucune information n'est affichée. Si on lui donne la valeur 1, une trace d'une ligne est envoyée à la console stdout chaque fois qu'une procédure ou une commande de premier niveau est compilée. Avec la valeur 2, un listing détaillé des instructions de code binaire émises pendant la compilation est envoyé à la console. Cette variable est utile pour traquer des problèmes que l'on suspecte de la part du compilateur de code binaire.

tcl_traceExec

La valeur de cette variable peut être fixée pour contrôler la quantité d'information tracée qu'il faut afficher au moment de l'exécution du code binaire compilé. Par défaut, **tcl_traceCompile** a la valeur 0 et aucune information n'est affichée. Si on lui donne la valeur 1, un résumé d'une ligne est envoyé à la console stdout chaque fois qu'une procédure Tcl est appelée. Si on lui donne la valeur 2, une trace d'une ligne est envoyée à la console stdout chaque fois qu'une commande Tcl contenant le nom de la commande et de ses arguments est invoquée. Avec une valeur de 3, on obtient une trace détaillée montrant

le résultat de l'exécution de chaque instruction compilée. Néanmoins, des instructions telles que *set* et *incr* entièrement remplacées par une séquence d'instructions binaires ne sont pas montrées.

tcl_wordchars
La valeur de cette variable est une expression régulière qui peut être fixée pour définir ce qu'on doit considérer comme un caractère constitutif d'un « mot ». C'est cette valeur qui détermine comment un « mot » est sélectionné par double-clic dans Tk. Sur Windows, cette expression est fixée à \S, ce qui désigne n'importe quel caractère autre que les caractères d'espacement Unicode. Ailleurs, la valeur par défaut est \w, ce qui désigne n'importe quel caractère de mot Unicode (nombre, lettre ou caractère de soulignement).

tcl_nonwordchars
La valeur de cette variable est une expression régulière qui peut être fixée pour définir ce qu'on doit considérer comme un caractère n'appartenant pas à un « mot » comme lorsqu'on sélectionne un mot par double-clic dans Tk. Sur Windows, cette expression est fixée à \s, ce qui désigne n'importe quel caractère d'espacement Unicode. Ailleurs, la valeur par défaut est \W, ce qui désigne n'importe quel caractère Unicode autre qu'un nombre, une lettre ou le caractère de soulignement.

tcl_version
Quand un interpréteur est créé, Tcl initialise cette variable avec une chaîne indiquant le numéro de version de Tcl, sous la forme *x.y*. Un changement de valeur de x correspond à des changements majeurs et peut-être des incompatibilités avec les versions antérieures. Un changement de valeur de y représente de petites améliorations et des corrections de bogues qui ne sont pas cause d'incompatibilités. La valeur de cette variable peut être obtenue au moyen de la commande **info tclversion**.

tell

La commande **tell** renvoie la position d'accès courante dans un canal ouvert.

Syntaxe

tell *Canal*

Description

La valeur renvoyée est une valeur numérique représentant un décalage en nombre d'octets. Cette valeur peut ensuite être utilisée avec la commande **seek** afin de se positionner à un endroit particulier dans le canal. Si le canal ne supporte pas le positionnement, la valeur renvoyée est -1.

time

La commande **time** permet d'évaluer le temps d'exécution d'un script.

Syntaxe

time *Script ?nb?*

Description

Cette commande provoque l'évaluation du script *Script* par l'interpréteur Tcl *nb* fois (une seule fois si le nombre *nb* n'est pas spécifié en argument). La valeur renvoyée est le temps moyen, mesuré en microsecondes, d'exécution du script au cours de ces *nb* évaluations. La réponse pourra ressembler à ceci :

```
503 microseconds per iteration
```

trace

La commande **trace** permet de contrôler le déroulement des appels de commandes et des accès de variables.

Syntaxe

trace *Sous-commande ?Arg Arg...?*

Description

Le mécanisme est très général: cette commande provoque l'exécution de certaines procédures Tcl dès que certaines opérations sont invoquées. Les différentes opérations sont spécifiées au moyen de sous-commandes. La commande **trace** a subi de profondes modifications avec la version 8.4 de Tcl. C'est cette nouvelle syntaxe qui est décrite ici. Les sous-commandes reconnues sont les suivantes :

trace add *Type Nom Opérations ?Args?*
Cette commande permet d'ajouter une trace. Sa syntaxe est différente suivant le type de l'objet tracé. L'argument *Type* peut être **command**, **execution** ou bien **variable**, ce qui conduit aux trois cas suivants :

trace add command *Nom Opérations Commande*
Fait en sorte que la procédure *Commande* soit exécutée dès que la commande *Nom* est modifiée d'une des manières indiquées par la liste représentée par l'argument *Opérations*. La commande *Nom* sera recherchée dans l'espace de noms courant. Si elle n'existe pas, une erreur est générée. L'argument *Opérations* est une liste comportant un ou plusieurs des termes suivants :

rename
Invoque la procédure *Commande* chaque fois que la commande tracée est renommée. Dans le cas où elle est renommée avec une chaîne vide, elle ne sera pas tracée car cela revient en fait à la détruire (cf. la commande **rename** à la page 149).

delete
Invoque la procédure *Commande* lorsque la commande tracée est détruite. Cela se produit lorsque la commande est renommée comme chaîne vide ou bien lorsque l'interpréteur lui-même est détruit mais, dans ce dernier cas, il n'y aura pas de trace car il n'y a plus d'interpréteur où exécuter la trace.

Lorsqu'une trace est déclenchée, trois arguments sont ajoutés à la procédure *Commande*. Celle-ci doit donc être définie avec ces trois arguments et sera appelée sous la forme :// **commande** *ancienNom nouveauNom op*

Les arguments *ancienNom* et *nouveauNom* désignent respectivement le nom de la variable avant que l'on cherche à la renommer ou à la détruire et son nouveau nom (c'est-à-dire la chaîne vide dans le cas d'une destruction). L'argument *op* indique celle des deux opérations qui était demandée (*rename* ou *delete*).
La procédure ne peut pas arrêter la destruction d'une commande : une fois la trace achevée la commande sera bien détruite. Par ailleurs le processus n'est pas récurrent : si la procédure elle aussi renomme ou détruit la commande, une autre trace ne sera pas déclenchée.

trace add execution *Nom Opérations Commande*
Fait en sorte que la procédure *Commande* soit exécutée dès que la commande *Nom* est modifiée d'une des manières indiquées par la liste représentée par l'argument *Opérations*. La commande *Nom* sera recherchée dans l'espace de noms courant. Si elle n'existe pas, une erreur est générée.
L'argument *Opérations* est une liste comportant un ou plusieurs des termes suivants :

enter
Invoque la commande *Commande* chaque fois que la procédure *Nom* est exécutée, juste avant que l'exécution ait lieu.

leave
Invoque la commande *Commande* chaque fois que la procédure *Nom* est exécutée, juste après que l'exécution ait eu lieu.

enterstep
Invoque la commande *Commande* pour toutes les commandes Tcl qui sont exécutées à l'intérieur de la procédure *Nom*, juste avant que l'exécution ait lieu. Par exemple si l'on a une procédure

```
proc abc {} { puts "Hello" }
```

une trace de type *enterstep* serait invoquée juste avant que l'instruction `puts "hello"` soit exécutée. Installer une trace de type *enterstep* sur une commande de base ne provoque pas d'erreur et est simplement ignoré.

leavestep
Invoque la commande *Commande* pour toutes les commandes Tcl qui sont exécutées à l'intérieur de la procédure *Nom*, juste après que l'exécution ait eu lieu. Installer une trace de type *leavestep* sur une commande de base ne provoque pas d'erreur et est simplement ignoré.

Lorsque la trace est déclenchée, suivant l'opération, un certain nombre d'arguments sont ajoutés à la commande *Commande*. Celle-ci sera donc appelée sous la forme suivante :

– Pour des opérations *enter* et *enterstep* :
commande *chaîneCmd op*

L'argument *chaîneCmd* contient la commande courante complète qui est exécutée (c'est-à-dire la commande tracée pour une opération *enter* ou une commande arbitraire pour une opération *enterstep*) avec tous ses arguments sous une forme entièrement développée. L'argument *op* indique l'opération (*enter* ou *enterstep*). L'opération de traçage peut être utilisée pour stopper l'exécution de la commande en détruisant la commande en question. Bien sûr lorsque la commande est exécutée ensuite, une erreur « commande inconnue » se produira.

– Pour des opérations *leave* et *leavestep* :
commande *chaîneCmd code résultat op* L'argument *chaîneCmd* contient la commande courante complète qui est exécutée (c'est-à-dire la commande tracée pour une opération *leave* ou une commande arbitraire pour une opération *leavestep*) avec tous ses arguments sous une forme entièrement développée. L'argument *code* indique le code résultant de l'exécution et l'argument *résultat* contient le résultat de cette exécution. L'argument *op* indique l'opération (*leave* ou *leavestep*). L'opération de traçage peut être utilisée pour stopper l'exécution de la commande en détruisant la commande en question. Bien sûr lorsque la commande est exécutée ensuite, une erreur « commande inconnue » se produira.

La création de plusieurs traces de type *enterstep* ou *leavestep* peut aboutir à des résultats complexes car les commandes invoquées pour une trace peuvent elles-mêmes conduire à des invocations de commandes pour d'autres traces. La commande *Commande* est exécutée dans le même contexte que celui du code qui a invoqué l'opération de trace : la commande *Commande*, si elle est invoquée depuis une procédure, aura accès aux mêmes variables locales que le code de la procédure. Ce contexte peut être différent de celui dans lequel la trace a été créée. Si la commande *Commande* invoque une procédure (ce qui est normalement le cas), cette procédure devra utiliser les commandes **upvar** ou **uplevel** si elle veut avoir accès aux variables locales du code qui a invoqué l'opération de trace. Pendant que la commande *Commande* s'exécute, les traces sur la procédure *Nom* sont désactivées : cela permet à *Commande* d'invoquer *Nom* sans déclencher à nouveau de trace de manière récurrente. Si une erreur se produit pendant l'exécution des instructions de *Commande* alors la procédure *Nom* renverra elle-même la même erreur.

Si des traces multiples sont installées sur la procédure *Nom* les commandes tracées sont invoquées dans l'ordre inverse de celui dans lequel les traces ont été créées s'il s'agit d'opération *enter* ou *enterstep*, et dans l'ordre original s'il s'agit d'opération *leave* ou *leavestep*. Le comportement des traces d'exécution est indéfini pour des procédures qui sont importées depuis un autre espace de noms.

trace add variable *Nom Opérations Commande*

Fait en sorte que la procédure *Commande* soit exécutée dès que la variable *Nom* est soumise à une des opérations indiquées par la liste représentée par l'argument *Opérations*. La commande *Nom* peut désigner une variable ordinaire, une variable de tableau ou un élément de ce tableau. Si elle désigne le nom d'un tableau, la trace sera déclenchée dès qu'un élément quelconque du tableau est modifié. Si la variable n'existe pas, elle est créée mais sans valeur attribuée : elle sera visible avec la commande **namespace which** mais pas avec la commande **info exists**.

L'argument *Opérations* est une liste comportant un ou plusieurs des termes suivants :

array
: Invoque la procédure *Commande* chaque fois que l'on accède à la variable ou qu'on la modifie au moyen de la commande **array**. Si l'argument *Nom* désigne en fait une variable scalaire, l'accès par la commande **array** ne déclenche pas de trace.

read
: Invoque la procédure *Commande* chaque fois que la variable est lue.

write
: Invoque la procédure *Commande* chaque fois que la variable est écrite.

unset
: Invoque la procédure *Commande* chaque fois que la variable est détruite. Les variables peuvent être détruites explicitement par la commande **unset**, ou bien lorsque la procédure au niveau de laquelle elles sont localement définies retourne ou encore si l'interpréteur courant lui-même est détruit. Dans ce dernier cas cependant, il n'y aura pas de trace car il n'y aurait plus d'interpréteur où exécuter la trace.

Lorsqu'une trace est déclenchée, trois arguments sont ajoutés à la procédure *Commande*. Celle-ci doit donc être définie pour admettre ces trois arguments et sera appelée sous la forme :
commande *Nom1 Nom2 op*
Les arguments *Nom1* et *Nom2* indiquent les noms de la variable à laquelle on accède : si la variable est scalaire, *Nom1* représente son nom et *Nom2* est vide ; si c'est un élément de tableau, *Nom1* indique le nom du tableau et *Nom2* l'élément du tableau ; si la trace porte sur un tableau entier, *Nom1* indique le nom du tableau et *Nom2* est vide. *Nom1* et *Nom2* ne sont pas nécessairement les mêmes que ceux utilisés dans la commande **trace variable** : la commande **upvar** (cf. p. 200) permet de lier par référence une variable à un autre nom. L'argument *op* indique celle des opérations qui était demandée (*read*, *write* ou *unset*).
La procédure *Commande* est exécutée dans le même contexte d'évaluation que le code qui a invoqué la trace. Si une variable que l'on trace

a été manipulée dans le cadre d'une procédure Tcl alors la procédure *Commande* aura accès aux mêmes variables qu'elle. Ce contexte peut être différent de celui dans lequel la trace a été créée avec la commande **trace**. Si *Commande* invoque une procédure, celle-ci devra faire appel aux commandes **upvar** ou **uplevel** si elle souhaite avoir accès à la variable tracée.

Pour des traces de type *read* et *write*, *Commande* peut modifier la variable afin d'affecter le résultat de l'opération tracée. Dans ce cas-là, la nouvelle valeur sera renvoyée comme résultat de l'opération tracée. La valeur de retour de *Commande* est ignorée sauf dans le cas où une erreur se sera produite : l'opération tracée renverra elle aussi une erreur contenant le même message d'erreur. C'est un mécanisme qui permet d'implémenter des variables « en lecture seule » par exemple. Pour une trace de type *write*, la procédure *Commande* est appelée après que la valeur de la variable aura été modifiée par l'opération d'écriture. Si l'on implémente des variables en lecture seule, il faudra que *Commande* soit capable de rétablir l'ancienne valeur de la variable pour rendre nul l'effet d'une écriture.

Pendant que *Commande* s'exécute dans le cadre d'une trace de type *read* ou *write*, les traces sur la variable sont temporairement désactivées : des lectures et des écritures de la variable dans le cadre de *Commande* ne déclencheront pas une trace à nouveau afin d'éviter des appels récurrents. Néanmoins si *Commande* cherche à détruire la variable avec la commande **unset**, alors des traces de type *unset*, s'il y en a, seront déclenchées.

Lorsqu'une trace de type *unset* est invoquée, la variable a déjà été détruite et elle apparaîtra comme indéfinie. Si la destruction résulte de la sortie de la procédure dans laquelle la variable tracée était locale, la trace sera invoquée dans le contexte de la procédure appelante puisque la procédure qui vient de terminer n'existe déjà plus dans la pile. Les traces ne sont pas désactivées pendant une trace de type *unset* : si *Commande* crée une nouvelle trace **unset** et accède à la variable, la trace sera invoquée. Toute erreur sur une trace **unset** est ignorée.

Si plusieurs traces ont été créées sur une variable, elles sont invoquées dans l'ordre inverse de leur création, la plus récente en premier. Si l'une d'elles renvoie une erreur, les suivantes ne sont pas invoquées. Enfin si une trace a été définie sur un élément de tableau et qu'il existe aussi une trace sur le tableau dans son ensemble, cette dernière est invoquée avant celle qui porte sur l'élément particulier.

Une trace reste valide tant qu'elle n'a pas été explicitement annulée par une commande **trace remove variable** ou détruite par une commande **unset** ou bien que l'interpréteur lui-même n'a pas été détruit. Détruire une trace sur un élément de tableau n'annule pas les traces portant sur le tableau entier. La commande renvoie une chaîne vide.

trace remove *Type Nom Opérations Commande*

L'argument *Type* peut être **command**, **execution** ou bien **variable** :
trace remove command *Nom Opérations Commande*
trace remove execution *Nom Opérations Commande*
trace remove variable *Nom Opérations Commande*

Dans chacun de ces trois cas, si une trace a été créée sur la commande *Nom* avec les opérations contenues dans la liste *Opérations* et la procédure *Commande*, la trace est supprimée, ce qui fait que *Commande* ne sera plus jamais invoquée. Cette commande renvoie une chaîne vide. Dans le cas de **command** et **execution**, si *Nom* n'existe pas, une erreur est générée.

trace info *Type Nom*
L'argument *Type* peut être **command**, **execution** ou bien **variable** :
trace info command *Nom*
trace info execution *Nom*
trace info variable *Nom*

Dans chacun de ces trois cas, la commande renvoie une liste contenant un élément pour chaque trace couramment installée sur l'argument *Nom*. Chaque élément de la liste est lui-même une liste de deux termes qui sont respectivement la liste des opérations et la procédure *Commande* associée. Si l'argument *Nom* n'est pas soumis à une trace, le résultat de la commande sera une chaîne vide. Dans le cas de **command** et **execution**, s'il n'existe pas, une erreur est générée.

Pour des raisons de compatibilité rétroactive, trois autres sous-commandes anciennes sont encore disponibles mais devraient être remplacées désormais par leurs équivalents :

trace variable *Nom Opérations Command*
Équivalent à **trace add variable** *Nom Opérations Commande*.

trace vdelete *Nom Opérations Command*
Équivalent à **trace remove variable** *Nom Opérations Commande*.

trace vinfo *Nom*
Équivalent à **trace info variable** *Nom*.

Elles sont en voie d'abandon. Elles utilisent une syntaxe ancienne dans laquelle les types *array*, *read*, *write*, *unset* sont remplacés par **a**, **r**, **w** et **u** respectivement, l'argument *Opérations* n'est pas une liste mais la concaténation des lettres symbolisant les opérations, comme par exemple **rwua**.

unknown

La commande **unknown** gère les invocations de commandes inexistantes.

Syntaxe

unknown *NomCommande ?Arg Arg...?*

Description

Cette commande est invoquée par l'interpréteur Tcl chaque fois qu'un script appelle une procédure qui n'existe pas. La commande **unknown** n'est pas une commande interne de base de Tcl : il s'agit d'une procédure dont la bibliothèque standard de Tcl donne une définition par défaut mais que chaque application ou script peut redéfinir pour en modifier le comportement et l'adapter à ses besoins.

S'il ne trouve aucune commande **unknown**, l'interpréteur renvoie une erreur. Si elle existe, elle est invoquée avec pour arguments le nom complet et les arguments d'origine de la commande *NomCommande* inexistante qui a déclenché ce mécanisme.

Les tâches typiques que la commande **unknown** peut accomplir consistent par exemple à rechercher dans des répertoires contenant des bibliothèques de code Tcl une procédure appelée *NomCommande*, ou à développer des noms de commandes abrégés ou encore exécuter les commandes inconnues comme des sous-processus. Chaque application installera les mécanismes qu'elle souhaite. Le résultat de la commande est utilisé comme résultat de la commande inexistante d'origine.

La bibliothèque standard fournit une définition par défaut pour une commande **unknown** qui peut servir de modèle :

- elle commence par appeler la procédure **auto_load** pour charger la commande inconnue ;
- en cas d'échec, elle appelle alors **auto_execok** pour voir s'il existe un fichier exécutable de ce nom. Si c'est le cas la commande **exec** est invoquée pour l'exécuter ;
- si elle ne peut être auto-exécutée, **unknown** cherche si la commande a été appelée au niveau supérieur en dehors de tout script. Elle regarde alors si la commande a l'une des trois formes suivantes :
  !!
  !*Événement*
  ˆancienˆnouveau?ˆ?
  Si c'est le cas, **unknown** effectue des substitutions avec l'historique des commandes à la manière du *shell* csh. Ce cas particulier est illustré à la section **??** : il correspond aux raccourcis que l'on peut utiliser sur la ligne de commande pour naviguer dans l'historique (cf. p **??**).
- finalement, **unknown** examine si la commande inconnue est une abréviation unique pour une commande Tcl existante. Si oui elle la remplace par le nom

entier et l'exécute avec les arguments d'origine.

Si aucune des tentatives n'aboutit, la commande **unknown** génère une erreur. Si la variable globale **auto_noload** est définie alors la première étape est ignorée. Si la variable globale **auto_noexec** est définie alors la deuxième étape est ignorée. Normalement, la valeur de retour de la commande **unknown** est la valeur de retour de la commande qui aura effectivement été exécutée.

unset

La commande **unset** détruit des variables existantes.

Syntaxe

unset ?--??**-nocomplain**? *?Nom Nom Nom...?*

Description

Cette commande permet de supprimer une ou plusieurs variables dont les noms lui sont passés en argument. Si l'argument se réfère à un élément d'un tableau associatif, cet élément est supprimé du tableau et les autres éléments restent intacts. Mais si l'argument est le nom d'un tableau, c'est le tableau entier qui est supprimé. La valeur de retour de la commande est toujours une chaîne vide.

Si l'option **-nocomplain** est spécifiée, aucune erreur ne sera rapportée. Les erreurs peuvent se produire dans les cas suivants :

- si l'on essaye de supprimer une variable qui n'existe pas ;
- si l'on essaye de supprimer un élément de tableau alors que le nom du tableau désigne en fait une variable scalaire ;
- si l'on se réfère à une variable d'un espace de noms qui n'existe pas (cf. p. 124).

Comme toujours on peut utiliser un double tiret -- pour séparer une éventuelle option **-nocomplain** des autres arguments.

update

La commande **update** exécute les événements en suspens et les fonctions de rappel.

Syntaxe

update?**idletasks**?

Description

Cette commande est utilisée pour mettre à jour une application, c'est-à-dire lui faire exécuter toutes les tâches qu'elle pourrait avoir en retard. Elle force l'application à rentrer de manière répétitive dans la boucle d'événements jusqu'à ce que tous les événements en suspens aient été traités, y compris les fonctions de rappel (*callbacks*) dont l'exécution est repoussée aux temps de repos c'est-à-dire lorsque la file d'attente des événements est vide.

Si la sous-commande **idletasks** est ajoutée, aucun événement dans la file d'attente, ni aucune erreur ne seront traités, seules le seront les tâches de repos, ce qui comprend principalement les rafraîchissements d'écran et les calculs de mise en page des fenêtres.

Le choix entre les deux formes de cette commande dépend du type d'événement que l'on veut mettre à jour :

- la commande **update idletasks** est utile dès qu'un script a modifié l'état d'une application et que l'on veut que ces modifications soient immédiatement répercutées sur l'écran plutôt que d'attendre la fin de l'exécution du script. La plupart des mises à jour et rafraichissements d'écran sont exécutés par des fonctions de rappel : celles-ci seront déclenchées immédiatement par une commande **update idletasks**. Cependant certaines mises à jour n'ont lieu qu'en réponse à certains événements, comme par exemple celles qui résultent de modifications des dimensions d'une fenêtre : celles-ci ne seront pas déclenchées par **update idletasks**.
- la commande **update** toute seule est utile dans les scripts qui exécutent un long calcul et où l'on souhaite que l'application continue de répondre aux événements susceptibles d'être occasionnés par l'utilisateur. Il suffit d'appeler épisodiquement la commande **update** : les données reçues seront traitées lors du prochain appel à cette commande.

uplevel

La commande **uplevel** exécute un script dans un autre niveau de la pile.

Syntaxe

uplevel *?Niveau? Arg ?Arg...?*

Description

Cette commande a pour effet de concaténer tous les arguments *Arg* à la manière de la commande **concat** puis de provoquer l'évaluation du résultat obtenu dans le contexte de variable indiqué par l'argument *Niveau*. La valeur de retour est le résultat de cette évaluation.

Il y a plusieurs syntaxes différentes pour cet argument *Niveau* :

- si c'est un nombre entier, il indique la distance (en termes de niveaux d'imbrication dans la pile) dont l'interpréteur devra se déplacer afin d'exécuter la commande ;
- si *Niveau* est indiqué au moyen d'un symbole dièse # suivi d'un nombre, ce nombre désigne un niveau absolu dans la pile à partir du sommet ;
- s'il est omis, par défaut il vaudra 1. Il ne peut dependant pas prendre de valeur par défaut lorsque le premier argument *Commande* commence par un chiffre ou par un symbole #.

Prenons l'exemple d'une procédure **a** invoquée depuis le niveau global, qui appelle une procédure **b** qui à son tour appelle une procédure **c**. Si la procédure **c** invoque la commande **uplevel** en spécifiant un *Niveau* 1 ou, ce qui ici revient au-même `#2`, alors la commande sera exécutée dans le même contexte de variable que la procédure **b**. Si le niveau était 2 (ou `#1`) alors la commande serait exécutée dans le même contexte de variable que la procédure **a**. Avec un *Niveau* 3 ou `#0` la commande sera exécutée au niveau global.

La procédure appelante (ici **c**) disparaît en quelque sorte de la pile d'exécution pendant l'exécution de la commande **uplevel**. Si la procédure **c** contient le code suivant

```
uplevel 1 {set x 43; d}
```

où **d** est encore une autre procédure Tcl, la commande **set** modifiera la variable x dans le contexte de **b** et **d** sera exécutée au niveau 3 *comme si elle avait été appelée depuis* **b** et non pas depuis **c**. Si **d** à son tour contient le code suivant :

```
uplevel {set x 42}
```

cette commande **set** modifiera la *même* variable x dans le contexte de **b**. La procédure **c** semble donc avoir disparu de la pile pendant que **d** s'exécute. Pour savoir à quel niveau d'exécution l'on se trouve dans la pile, on peut utiliser la commande **info level**.

La commande **namespace eval** constitue une autre façon de modifier le contexte d'évaluation de Tcl. Elle ajoute un niveau d'appel à la pile pour représenter le

contexte de l'espace de noms : ce niveau devra être comptabilisé lorsque l'on utilise les commandes **uplevel** et **upvar**.

Inversement, l'instruction `info level 1` renvoie une liste décrivant ou bien la commande la plus extérieure dans une série d'appels ou bien la commande **namespace eval** la plus extérieure. L'instruction `uplevel 0` provoque l'évaluation des arguments qui suivent au niveau global.

upvar

La commande **upvar** établit un lien avec une variable qui se trouve à un niveau différent dans la pile d'exécution.

Syntaxe

upvar *?Niveau? AutreVar1 maVar1 ?AutreVar2 maVar2...?*

Description

Cette commande permet qu'une ou plusieurs variables locales de la procédure courante se réfèrent à des variables définies à des niveaux supérieurs dans des procédures appelantes ou à des variables globales. Le principe n'est pas sans analogie avec les passages de variables par référence du langage C++.

Le niveau est désigné par l'argument *Niveau* selon les mêmes règles que pour la commande **uplevel** (cf. p. 198). Ce niveau peut être omis (sa valeur par défaut étant 1) mais à condition que l'argument suivant ne commence pas par un chiffre ou par un symbole # pour qu'il n'y ait pas de risque de mésinterprétation. La commande **upvar** rend chaque variable *AutreVar* définie au niveau indiqué accessible dans la procédure courante sous le nom spécifié par l'argument *maVar*. La valeur de retour de **upvar** est une chaîne vide.

La variable *AutreVar* n'a pas besoin forcément d'exister au moment de l'appel : elle sera créée la première fois que la variable *maVar* sera référencée. Il ne doit pas déjà exister de variable nommée *maVar* lorsque **upvar** est invoquée. Cette variable sera toujours traitée comme une variable ordinaire et pas comme une variable associative de type *array*. En revanche la variable *AutreVar* peut faire référence à une variable simple, un tableau ou un élément de tableau.

Cette commande permet de transmettre des noms de variables dans des appels de procédures et d'utiliser la valeur de cette variable dans le corps de la procédure. Par exemple supposons que l'on veuille définir une procédure *mult2* pour multiplier la valeur d'une variable par 2 de telle sorte que l'on puisse par la suite écrire des instructions telles que : `mult2 nombre` où *nombre* désigne donc le *nom* d'une variable numérique quelconque. Il faudra procéder comme ceci :

```
proc mult2 nom {
    upvar $nom x
    set x [expr $x*2]
}
```

La commande **upvar** permet d'accéder à la variable *nom* du niveau supérieur en la liant localement à une variable x.

La même remarque qu'avec la commande **uplevel** s'applique concernant les espaces de noms éventuels et le niveau supplémentaire d'évaluation introduit par une instruction telle que **namespace eval** (cf. p. 198).

Si l'on cherche à détruire, par une commande **unset**, une variable créée par **upvar** (comme par exemple la variable x de l'exemple précédent) c'est en fait la variable à laquelle elle est liée qui sera détruite. Il n'existe pas de moyen de détruire x elle-même : c'est de toute façon une variable purement locale qui disparaît dès que l'on sort de la procédure. Il est possible néanmoins de lier x à une autre variable par une autre instruction **upvar**.

▷ Un autre point délicat concerne les traces de variables faites sur des variables liées (voir la commande **trace** à la page 188). Si l'on cherche à tracer *AutreVar*, cette trace sera déclenchée par toute opération effectuée sur *maVar* plutôt que sur le nom de *AutreVar*. Par exemple le code suivant donnera comme résultat **VarLocale** et non pas **VarOriginale** :

```
proc traceProc {nom index op} {
    puts $nom
}
proc setAvecUpvar {nom valeur} {
    upvar $nom VarLocale
    set VarLocale $valeur
}
set VarOriginale 1
trace variable VarOriginale w traceProc
setAvecUpvar VarOriginale 2
```

Si *AutreVar* se réfère à un élément d'un tableau, un traçage défini sur l'ensemble du tableau ne sera pas invoqué lors d'un accès à *maVar* mais des traces sur un élément particulier le seraient. En particulier, dans le cas du tableau interne **env** (cf. p 180), des changements opérés sur une variable locale *maVar* qui lui serait liée, ne seront pas transmis correctement aux sous-processus.

variable

La commande **variable** crée et initialise une variable dans un espace de noms.

Syntaxe

variable *?Nom Valeur...? Nom?Valeur?*

Description

Cette commande est normalement utilisée à l'intérieur d'une commande **namespace eval** afin de créer une ou plusieurs variables au sein de l'espace de noms. Chaque variable *Nom* est initialisée à la valeur représentée par l'argument *Valeur* correspondant. Cette *Valeur* est optionnelle pour la dernière variable.

Si une des variables *Nom* n'existe pas, elle est créée et si l'argument *Valeur* est spécifié, il lui est attribué, autrement elle est laissée indéfinie. Si la variable existe déjà, elle est fixée à la valeur de l'argument *Valeur* s'il est spécifié ou laissée inchangée dans le cas contraire. Normalement, *Nom* n'est pas qualifié (en ce sens qu'il n'inclut pas les noms des espaces de noms qui contiennent cette variable), et la variable est créée dans l'espace de noms courant.

Si cependant *Nom* inclut des qualificateurs d'espaces de noms, la variable sera créée dans l'espace indiqué. Si la variable n'est pas définie, elle sera visible avec la commande **namespace which** mais pas avec la commande **info exists**.

Si la commande **variable** est exécutée à l'intérieur d'une procédure Tcl, elle crée des variables locales qui sont liées aux variables correspondantes de l'espace de noms. De la sorte, la commande est analogue à la commande **global** quoique celle-ci lie uniquement à des variables de l'espace global. Si des valeurs ont été données, elles modifient les valeurs des variables de l'espace de noms associées. Si une variable de l'espace de noms n'existe pas, elle est créée et optionnellement initialisée.

Il existe une restriction concernant les variables déclarées par la commande **variable** : l'argument *Nom* ne peut faire référence à un élément d'un tableau. À la place, *Nom* devrait faire référence à un tableau entier et la valeur d'initialisation *Valeur* devrait être omise. Une fois la variable ainsi déclarée, les éléments du tableau peuvent être fixés au moyen des commandes usuelles **set** ou **array**.

vwait

Exécute des événements jusqu'à ce qu'une variable soit fixée.

Syntaxe

vwait *nomVar*

Description

Cette commande force l'interpréteur à entrer dans une boucle d'événements: Tcl traitera les événements et instructions qui arrivent jusqu'à ce qu'une certaine variable *nomVar* dont le nom a été donné en argument de la commande reçoive une valeur ou bien voie sa valeur modifiée. La commande **vwait** met alors fin à la boucle d'événements aussitôt que la procédure ou la série d'instructions qui ont modifié la variable s'achèvent. La variable *nomVar* doit être une variable globale, soit au niveau global de l'application, soit au niveau d'un espace de noms et dans ce cas elle doit être désignée par son nom pleinement qualifié.

Dans certains cas, la commande **vwait** ne termine pas immédiatement après que la variable associée ait été modifiée. Cela se produit si la procédure qui fixe une valeur à la variable ne s'achève pas immédiatement : si par exemple cette procédure fixe une valeur à la variable et appelle à son tour une autre commande **vwait** associée à une autre variable, alors la première commande **vwait** se retrouve bloquée et ne pourra retourner que lorsque la seconde sera elle-même terminée.

while

La commande **while** exécute un script de manière répétitive tant qu'une certaine condition est remplie.

Syntaxe

while *Test Corps*

Description

La commande **while** évalue l'argument *Test* comme une expression, de la même manière que la commande **expr** évalue son argument (cf. p. 57). La valeur renvoyée par ce test doit être une valeur booléenne valide : si celle-ci représente la valeur *true* alors le corps de la boucle est exécuté par l'interpréteur Tcl, à la suite de quoi la condition de test est évaluée à nouveau. Tant qu'elle renvoie la valeur *true* (*true* ou 1) le corps est exécuté ; dès qu'elle renvoie *faux* (*false* ou 0), l'instruction **while** s'interrompt. La valeur de retour de **while** est toujours une chaîne vide.

On peut modifier le déroulement répétitif d'une commande **foreach** en utilisant les commandes **break** et **continue** qui ont exactement la même signification qu'avec la commande **for** (cf. p. 80).

On notera qu'il est prudent, sinon même indispensable, d'entourer l'instruction *Test* d'accolades afin d'éviter que des évaluations et interpolations de variables n'interviennent trop tôt. La raison est la même que pour la commande **for** (voir les explications à la page 80).

Index

SYMBOLES

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y