|  |
| --- |
| norme d’écriture des scripts ksh |
| Version : |

|  |
| --- |
| Romain Vassallo |

Sommaire

[1 Introduction 3](#_Toc419070427)

[1.1 Objectifs 3](#_Toc419070428)

[1.2 Conventions de rédaction 3](#_Toc419070429)

[1.3 Versions de ksh testées 4](#_Toc419070430)

[2 Règles générales 4](#_Toc419070431)

[2.1 Emploi du langage 4](#_Toc419070432)

[2.2 Nommage des scripts ksh 4](#_Toc419070433)

[2.3 Permissions 5](#_Toc419070434)

[2.4 Organisation générale des scripts ksh 5](#_Toc419070435)

[2.5 Règles de style 8](#_Toc419070436)

[3 Règles de programmation 12](#_Toc419070437)

[3.1 Généralités 12](#_Toc419070438)

[3.2 Codes d’exit 13](#_Toc419070439)

[3.3 Utilisation des variables 14](#_Toc419070440)

[3.4 Contrôle des arguments 19](#_Toc419070441)

[3.5 Gestion des options 19](#_Toc419070442)

[3.6 Détection des erreurs 24](#_Toc419070443)

[3.7 Gestion des erreurs par le shell 26](#_Toc419070444)

[3.8 Chemins d’accès aux commandes et aux fichiers 31](#_Toc419070445)

[3.9 Fichiers de traces 34](#_Toc419070446)

[3.10 Utilisation des fonctions 36](#_Toc419070447)

[3.11 Utilisation des tableaux 43](#_Toc419070448)

[3.12 Utilisation des traps 46](#_Toc419070449)

[4 Bonnes pratiques 51](#_Toc419070450)

[4.1 Lancer moins de sous-processus 51](#_Toc419070451)

[4.2 Externaliser les paramètres de configuration 53](#_Toc419070452)

[4.3 Variables PATH et LD\_LIBRARY\_PATH 56](#_Toc419070453)

[5 Récapitulatif 59](#_Toc419070454)

[6 Différences entre ksh et pdksh 62](#_Toc419070455)

[6.1 Commandes built-in à droite d’un pipe 62](#_Toc419070456)

[6.2 Traitement des erreurs d’exécution 63](#_Toc419070457)

[6.3 Gestion des traps 64](#_Toc419070458)

[6.4 Autres différences 66](#_Toc419070459)

[7 Références 70](#_Toc419070460)

[7.1 Généalogie de AT&T ksh et de pdksh 70](#_Toc419070461)

[7.2 ksh 70](#_Toc419070462)

[7.3 ksh93 71](#_Toc419070463)

[8 Historique des modifications 71](#_Toc419070464)

# Introduction

## Objectifs

Ce document décrit une norme de programmation des scripts utilisant le langage Korn Shell (ksh).

Cette norme permet de standardiser l’utilisation du Korn Shell afin de :

* améliorer la fiabilité des scripts ksh
* uniformiser les pratiques et faciliter la maintenance
* tirer profit des fonctionnalités disponibles dans ksh [[1]](#footnote-1)

Important : ce document traite uniquement de la version « 1988 » de Korn Shell, qui est celle habituellement utilisée jusqu’à présent. Il existe une version enrichie de Korn Shell, appelée ksh93, qui introduit de multiples améliorations ainsi que des incompatibilités ; cette nouvelle version n’est pas abordée dans ce document [[2]](#footnote-2).

## Conventions de rédaction

Les règles sont énoncées dans des cadres présentés comme ci-dessous :

|  |  |  |
| --- | --- | --- |
| REF-01 | Importance de la règle | Énoncé de la règle |

Phrase décrivant l’exigence à respecter

Cf. ci-dessous

Référence de la règle

Niveau d’importance : une règle peut être :

* Obligatoire : règle à respecter impérativement
* Recommandée : règle à observer en général, avec des exceptions possibles dans des cas particuliers
* Conseillée : bonne pratique, à observer si possible.

Remarque : certaines règles peuvent concerner une implémentation spécifique de ksh ; en pareil cas l’implémentation en question est mentionnée sous la référence de la règle.

## Versions de ksh testées

Les exemples dans ce document ont été testés avec les versions dans le tableau ci-après.

|  |  |  |
| --- | --- | --- |
| Interpréteur | Système d’exploitation | Remarques |
| ksh | Solaris 10 x86 32 bits  (SunOS 5.10 Generic\_142910-17 i86pc i386 i86pc) | Implémentation de AT&T ksh (1988) pour Solaris 10. |
| pdksh | Cygwin 1.7 (version de pdksh : v5.2.14 99/07/13.2) | Cf. section 6, page 62. |

Avertissement : les 2 implémentations ci-dessus ont servi de référence lors de la rédaction de ce document ; les implémentations de AT&T ksh sur d’autres Unix commerciaux (HP-UX, AIX, etc.), tout comme d’autres versions de pdksh le cas échéant, peuvent présenter des différences non décrites dans ce document.

# Règles générales

## Emploi du langage

|  |  |  |
| --- | --- | --- |
| GEN-01 | Obligatoire | Les scripts ksh sont utilisés dans les chaînes batchs pour réaliser des fonctions techniques *élémentaires* de manipulation de fichiers et de lancement unitaire de traitements.  Les scripts shells ne sont utilisés ni pour implémenter des fonctions métier, ni pour réaliser des enchaînements complexes de traitements. |

Explication : les fonctions métier doivent être implémentées dans des langages adaptés aux besoins, que cela soit pour des raisons de performance, de fiabilité, ou de coûts de maintenance. De ce point de vue, le langage shell n’est pas le plus adapté.

En ce qui concerne l’enchaînement des traitements, il doit être géré dans le système d’ordonnancement (Control-M, Autosys, etc.), qui est le plus adapté à cette tâche ; cela permet de limiter l’emploi du shell à des fonctions techniques simples et bien découpées.

## Nommage des scripts ksh

|  |  |  |
| --- | --- | --- |
| NOM-01 | Conseillé | Les noms des scripts sont en lettres minuscules uniquement (pas de mélange de casses majuscules et minuscules) ; le tiret de soulignement « \_ » peut être utilisé comme séparateur de mots. |

|  |  |  |
| --- | --- | --- |
| NOM-02 | Conseillé | Les scripts ksh portent l’extension .ksh |

|  |  |  |
| --- | --- | --- |
| NOM-03 | Conseillé | Les noms de scripts peuvent être préfixés par le trigramme du projet, éventuellement suivi de l’identification (trigramme) du module ou du lot. |

Exemple conforme : xyz\_mod\_faireqqch.ksh

## Permissions

|  |  |  |
| --- | --- | --- |
| PERM-01 | Obligatoire | Le bit d’exécution nécessaire au user applicatif est activé sur les scripts ksh. |

Remarque : ce n’est pas une règle de sécurité, c’est simplement nécessaire pour que le user concerné puisse utiliser le script comme une commande. Sans le droit d’exécution, le lancement du script échoue avec le message d’erreur « cannot execute » ; on peut contourner ce problème en appelant le script via la commande : ksh monscript.ksh [[3]](#footnote-3), cependant généraliser cette solution serait fastidieux.

Si possible, le droit d’exécution est positionné et conservé dans le gestionnaire de versions des sources ; à défaut, ou en complément, il est activé au moment du déploiement (dans ce cas il est préférable que cela soit automatisé).

Note : en fonction de l’utilisateur susceptible d’utiliser le script, on active typiquement le bit d’exécution soit pour le user propriétaire, soit à la fois pour le propriétaire et pour le groupe [[4]](#footnote-4).

|  |  |  |
| --- | --- | --- |
| PERM-02 | Conseillé | Le bit d’exécution n’est pas activé sur les fichiers *inclus* dans les scripts ksh via la commande . *fichier* |

Activer le bit d’exécution sur les fichiers inclus dans les scripts ksh n’est pas gênant, mais demeure superflu. Au contraire, l’omission de ce bit souligne que les fichiers concernés ne sont pas conçus comme des scripts exécutables de manière autonome, et qu’ils sont faits pour être joués dans le contexte de scripts appelants.

|  |  |  |
| --- | --- | --- |
| PERM-03 | Obligatoire | L’utilisation des bits setuid et setgid est interdite. |

Motif : lorsque qu’un utilisateur exécute un script dont le bit setuid est activé, il emprunte l’identité du propriétaire du script (*effective uid*) ; idem avec le bit setgid : l’utilisateur emprunte l’identité du groupe propriétaire (*effective gid*). Ce mécanisme octroie aux utilisateurs possédant un simple droit d’exécution sur le script la possibilité de l’exécuter avec *tous les privilèges du user et/ou du groupe propriétaire*. En pratique cela crée un risque de sécurité dans la mesure où un utilisateur mal intentionné peut détourner, le plus souvent assez aisément, la fonctionnalité initiale du script afin d’usurper ces privilèges.

## Organisation générale des scripts ksh

|  |  |  |
| --- | --- | --- |
| ORG-01 | Obligatoire | La première ligne des scripts ksh exécutables est obligatoirement :  #! /bin/ksh |

Explication : au démarrage d’un programme exécutable la fonction système exec(2) lit les premiers octets du programme ; la chaîne #! indique que le programme est un script interprété par le binaire dont le chemin suit immédiatement #! [[5]](#footnote-5).

Important : si la règle ci-dessus n’est pas respectée (omission de #!, 1ère ligne blanche avant #!, etc.) vous n’avez aucune garantie que le script sera réellement exécuté par l’interpréteur Korn shell [[6]](#footnote-6) ; or dans le cas contraire, le risque d’erreur d’exécution est très élevé.

Remarque : le chemin /bin/ksh n’est pas forcément universel, cependant il fonctionne avec la plupart des systèmes Unix qui implémentent ksh.

Bien sûr, la règle précédente ne concerne que les scripts exécutables : la ligne #! ne sert à rien sur les fichiers non exécutables inclus dans les scripts ksh par la commande . *fichier*.

|  |  |  |
| --- | --- | --- |
| ORG-02 | Recommandé | Organisation générale des scripts ksh  Les scripts ksh suivent habituellement le découpage ci-après :   1. 1ère ligne : #! /bin/ksh (cf. règle ) 2. Commentaires d’en-tête 3. Initialisations : définitions de variables, déclarations de fonctions, inclusion de scripts communs, etc. 4. [Facultatif] traitement des options de la ligne de commande 5. Lecture des arguments non optionnels de la ligne de commande 6. Contrôle de la syntaxe d’appel du script (sortie en erreur si non ok) 7. [Facultatif] création d’un fichier de log spécifique 8. Traces préalables au traitement (récapitulatif des arguments, etc.) 9. Traitement principal du script 10. [Si nécessaire] Nettoyage post-traitement : suppression des fichiers de travail inutiles après traitement, etc. 11. Traces post-traitement : bilan d’exécution |

Les commentaires d’en-tête suivent la ligne #! /bin/ksh. Ils incluent notamment :

* Le nom du script (sans le chemin)
* Une brève description de la fonction accomplie par le script
* La description de la syntaxe d’appel : cette description est plus ou moins détaillée en fonction du besoin, cependant elle doit toujours être complète et suffisante pour les utilisateurs et les futurs mainteneurs du script [[7]](#footnote-7)
* La description des codes d’exit du script
* Le cas échéant, des explications sur les points structurants de l’implémentation.

Exemple :

# APPLICATION

# XYZ -- the XYZ project

#

# PROGRAM

# xyz\_do\_something.ksh

#

# DESCRIPTION

# This program computes the "something" function on its input.

#

# SYNTAX

# xyz\_do\_something.ksh [ options ] file\_in file\_out

#

# ARGUMENTS

# file\_in

# path of input file

# file\_out

# path of output file. Use '-' to emit output on stdout.

#

# OPTIONS

# -v verbose mode

# -t dry run, only check validity of input

#

# RETURNS

# 0 if success

# > 0 if failure (2 if input file was not found, 1 in all other cases)

Les initialisations viennent ensuite, de préférence dans l’ordre indiqué :

1. Positionnement d’options particulières du shell, par exemple :

# prohibit undefined parameters [[8]](#footnote-8)

set -o nounset

1. Inclusion des scripts communs : ce sont les scripts (non exécutables) qui définissent les paramètres d’environnement applicatifs, les fonctions ksh couramment utilisées, les codes d’exit, etc.

. ${APP\_HOME}/shells/xyz\_profile.ksh # common env. var. for the XYZ project [[9]](#footnote-9)  
. ${APP\_HOME}/shells/xyz\_exit\_codes.ksh # common exit codes  
. ${APP\_HOME}/shells/xyz\_functions.ksh # common ksh functions

. ${APP\_HOME}/config/xyz\_config.ksh # parameters set by the XYZ admin.

1. Déclaration de variables spécifiant les commandes utilisées dans le script :

# Commands used in this script  
CP=/usr/bin/cp  
GREP=/usr/xpg4/bin/grep  
PRINTF=/usr/bin/printf  
RM=/usr/xpg4/bin/rm

Remarque : une meilleure solution pour alléger les scripts consisterait à placer ces dernières dans un script commun xyz\_unix\_cmds.ksh inclus à l’étape (b) ci-dessus ; en effet, les commandes utilisées dans les scripts sont toujours plus ou moins les mêmes.

1. Initialisations de variables utilisées localement dans le script.

L’étape de traitement des options consiste à analyser la ligne de commande pour détecter les arguments facultatifs correspondant à des options. Pour cela la commande getopts – cf. règle page 20 – est généralement utilisée. Les arguments identifiés comme des options sont ensuite retirés de la liste des arguments du script ($@) à l’aide de la commande shift, afin de ne conserver dans cette liste que les arguments non optionnels. Cela permet de trouver ces arguments à une position prévisible ($1, $2, etc.) quelque soit le nombre d’arguments optionnels présents initialement sur la ligne de commande.

Les arguments principaux du script sont lus une fois les options analysées. Cette opération consiste simplement à affecter les arguments $1, $2, etc. à des paramètres nommés.

# Mandatory args  
file\_in="$1"   
file\_out="$2"

En effet, la suite du script sera nettement plus claire si des paramètres avec des noms explicites sont utilisés en lieu et place de $1, $2, etc.

À ce stade, une première validation de la syntaxe d’appel du script est nécessaire, portant sur le nombre d’arguments sur la ligne de commande ($#) ; sinon les paramètres $1, $2, etc. peuvent ne pas être définis.

Au-delà de la simple vérification du nombre d’arguments ($#), les arguments lus sur la ligne de commande doivent être vérifiés de manière aussi complète que possible :

* si l’argument est le chemin d’un fichier existant, vérifier que ce fichier existe ;
* si l’argument attendu est un nombre, vérifier que la valeur lue est bien une suite de chiffres ;
* etc.

Ces contrôles servent à détecter le plus tôt possible les erreurs commises dans l’appel au script : il est largement préférable de détecter les erreurs *au niveau de l’interface* du script plutôt qu’au moment de l’utilisation des paramètres réels dans le cœur du traitement. Cela permet d’arrêter le script au plus tôt, sans traitement inutile, et d’émettre des messages d’erreurs précis.

Une fois les arguments de la ligne de commande entièrement analysés et validés, le traitement proprement dit peut s’exécuter. Les étapes n° 7 à 11 ci-dessus sont indicatives.

Remarque : on ne saurait trop insister sur la nécessité de concevoir les traces avec soin – cf. section 3.9 page 34 – pour répondre à 2 impératifs :

1. Fournir à l’utilisateur des messages aussi clairs que possible afin de faciliter l’utilisation du script et le suivi du traitement.
2. Tracer toutes les actions accomplies par le script susceptibles de modifier l’état de l’application ou des données, et ce quelque soit le contexte dans lequel le script est lancé : il faut autant de traces en mode « batch » (ordonnancé) qu’en mode « interactif » (lancement manuel).

## Règles de style

|  |  |  |
| --- | --- | --- |
| STY-01 | Recommandé | Longueur des lignes : la longueur des lignes est limitée à 80 caractères, sauf cas particuliers des messages pour lesquels un dépassement est tolérable. |

Explication : les lignes de commandes trop longues étant difficiles à lire, il est préférable d’écrire les commandes longues sur plusieurs lignes, en utilisant le caractère de continuation « \ » pour indiquer au shell que la commande se poursuit sur la ligne suivante.

Exemple :

java -Xmx1024m -classpath ${APP\_CLASSPATH} –Djava.io.tmpdir=${APP\_TMP\_DIR} ${APP\_DIR}/app.jar "$@"

est moins lisible que :

java \

–Xmx1024m \  
 -classpath ${APP\_CLASSPATH} \

–Djava.io.tmpdir=${APP\_TMP\_DIR} \  
 ${APP\_DIR}/app.jar \

"$@"

qui est donc plus facile à maintenir.

L’inconvénient de cette forme est qu’elle est moins compacte verticalement, c’est pourquoi il est tolérable de s’en écarter quand on ne peut pas faire autrement, ou quand une certaine compacité est souhaitable sur le plan vertical, à condition toutefois de rester dans les limites du raisonnable : le seuil des 110 caractères (soit 37,5% de dépassement) devrait être considéré comme une limite absolue.

|  |  |  |
| --- | --- | --- |
| STY-02 | Obligatoire | Largeur d’indentation : les scripts sont indentés avec une tabulation logique de 4 caractères ; les éditeurs de texte sont configurés pour indenter :   * soit uniquement avec des blancs * soit avec des blancs *et* des tabulations, mais à condition que la largeur des tabulations physiques reste à 8 caractères. |

Explication : en pratique, l’indentation logique avec un retrait de 4 caractères est le meilleur compromis.

Idéalement, l’indentation est réalisée avec des blancs seulement, car de cette manière le rendu sera identique sur tous les terminaux et éditeurs de texte. Alternativement un mélange de blancs et de tabulations est acceptable, à condition que l’indentation soit correctement restituée sur un terminal affichant les tabulations avec une largeur de 8 caractères [[10]](#footnote-10).

Exemple :

function log\_warning {

typeset ts=$($DATE "+%Y%m%d%H%M%S")

print –u2 "[$ts][WARNING] $@"

}

idx=0

max=100

while ((idx < max)) ; do

file\_to\_move="${tab\_file\_in[idx]}"  
 dest\_dir="${tab\_dir\_out[idx]}"  
 if [ -f "${file\_to\_move}" ]; then

$MV "${file\_to\_move}" "${dest\_dir}/."  
 rc=$?

if [ $rc –ne 0 ]; then

log\_warning "$MV \"${file\_to\_move}\" \"${dest\_dir}/.\" failed (exit code: $rc)"  
 fi  
 fi

((idx += 1))  
done

Remarque : l’utilisation de structures de contrôle avec un niveau d’imbrication trop élevé – par exemple, trop de boucles imbriquées – est le signe certain d’un programme insuffisamment structuré : il peut être utile de le restructurer en utilisant des fonctions (cf. section 3.10 page 36).

|  |  |  |
| --- | --- | --- |
| STY-03 | Obligatoire | Les paramètres positionnels $1, $2, etc., ne doivent être utilisés que pour copier leurs valeurs dans des paramètres nommés. Toute référence à ces paramètres en dehors de cet usage est interdite. |

Justification : le nom des paramètres positionnels : $1, $2, etc., n’est porteur d’aucune information, si ce n’est sur la position de l’argument dans la liste des arguments. Les scripts qui référencent ces paramètres dans le corps du traitement sont, de ce fait, difficiles à comprendre. De plus, ces scripts sont délicats à maintenir en cas d’évolution de la syntaxe d’appel (modification de l’ordre des arguments) ; enfin ces scripts sont particulièrement fragiles : que se passe-t-il si la liste des arguments est modifiée accidentellement ? Pour toutes ces raisons, le seul emploi admis des paramètres positionnels est l’affectation de leurs valeurs à des paramètres nommés.

Exemple non conforme :

# !!! A NE PAS FAIRE !!!

$CP -p "$1" "$2/${1##\*/}.tmp"

Version conforme :

input\_file="${1:-}"

tmp\_dir="${2:-}"

# NB: contrôles non détaillés [[11]](#footnote-11)  
[...]

work\_file="${tmp\_dir}/$(input\_file##\*/}.tmp"

$CP -p "${input\_file}" "${work\_file}"

Le code ainsi réécrit est certes moins compact mais il est nettement plus facile à comprendre – sur des scripts de taille réelle la différence devient écrasante – et bien plus facile à maintenir.

|  |  |  |
| --- | --- | --- |
| STY-04 | Obligatoire | Utilisation de la casse : les paramètres locaux, ainsi que les fonctions, sont nommés en casse minuscule ; les majuscules sont réservées aux constantes et aux paramètres exportés (variables d’environnement). Le tiret de soulignement (\_) peut être employé comme séparateur de mots dans les noms de variables et de fonctions. |

Cette convention facilite considérablement la lecture des scripts et leur maintenance.

Exemples :

PATH=/bin:/usr/bin # variable d’environnement (exportée) => majuscules

export PATH

LD\_LIBRARY\_PATH=$ORACLE\_HOME/lib # variable d’environnement (exportée) => majuscules

export LD\_LIBRARY\_PATH # Noter le « \_ » utilisé comme séparateur de mots.

RM=/usr/xpg4/bin/rm # paramètre constant (chemin complet d’une commande) => majuscules

DATE=/usr/bin/date # paramètre constant (chemin complet d’une commande) => majuscules

input\_file="$1" # paramètre local (argument du script) => minuscules

output\_file="$2" # paramètre local (argument du script) => minuscules

date\_trt=$($DATE "+%Y%m%d%H%M%S") # paramètre local => minuscules

log\_file="${input\_file%.\*}\_${date\_trt}.log # paramètre local => minuscules

A contrario, l’utilisation systématique des majuscules, ou encore les noms en casse mixte (de la forme *camelBackNotation* préconisée en Java) sont à éviter impérativement [[12]](#footnote-12).

|  |  |  |
| --- | --- | --- |
| STY-05 | Recommandé | Les paramètres dont le nom contient un tiret de soulignement (\_) doivent être évalués en les inscrivant dans une paire d’accolades. |

Explication :

echo "File number #${input\_file\_number} out of ${max\_input\_file\_count}..."

est plus facile à lire que :

echo "File number #$input\_file\_number out of $max\_input\_file\_count..."

Par ailleurs l’utilisation des accolades élimine les ambiguïtés et permet d’éviter des erreurs :

filein="$1"  
filelog="$filein\_log" # <-- ERREUR, l’auteur voulait écrire "${filein}\_log"

Exception : en contexte d’évaluation arithmétique (expressions let "…", (( … )), indices de tableaux, etc.), si un paramètre est référencé sans le faire précéder du symbole $ [[13]](#footnote-13), il n’est pas non plus entre accolades :

line\_count=0

max\_lines=10  
while ((line\_count < max\_lines)) ; do # <-- évaluation arithmétique entre (( et ))

...

((line\_count += 1)) # <-- évaluation arithmétique entre (( et ))

done

|  |  |  |
| --- | --- | --- |
| STY-06 | Recommandé | Expressions arithmétiques : la syntaxe recommandée pour les expressions arithmétiques est l’écriture de ces expressions entre (( et )) [[14]](#footnote-14). |

Explication : cette forme est plus lisible que la forme *let* "…" qui lui est équivalente. Par ailleurs elle est implémentée nativement dans ksh (i.e. c’est une commande built-in) contrairement à expr qui est une commande externe.

Exemple :

idx=0

max=10

while ((idx < max)) # NB : l’expression (( ... )) est une commande, c’est pourquoi elle n’est

do # pas précédée du symbole $.

echo "idx au carre = $((idx \* idx))" # On met le symbole $ devant (( ... )) pour substituer

# le résultat de l’expression sur la ligne de commande.

...

((idx += 1)) # Incrémente idx

done

|  |  |  |
| --- | --- | --- |
| STY-07 | Recommandé | Ne pas utiliser les backquotes ! La syntaxe recommandée pour substituer le résultat d’une commande est la forme $(*commande*) |

En ksh la syntaxe `*commande*` est obsolète ; il faut utiliser la syntaxe $(*commande*) qui est équivalente fonctionnellement mais sans les inconvénients [[15]](#footnote-15), et qui est nettement plus lisible par ailleurs.

Important : la syntaxe $(*commande*) n’est pas magique (pas plus que `*commande*`) : la commande invoquée peut échouer, c’est il pourquoi il est recommandé de contrôler son code d’exit ($?) afin d’être certain qu’elle s’est bien exécutée comme attendu.

# Règles de programmation

## Généralités

|  |  |  |
| --- | --- | --- |
| PRG-01 | Obligatoire | La duplication de code par « copier-coller » est interdite. |

Cette règle est valable dans tous les langages de programmation avancés, qui offrent des fonctionnalités suffisantes pour permettre de *factoriser* et de *réutiliser* le code commun. Le Korn shell ne fait pas exception.

La duplication de code par « copier-coller » consiste à créer un nouveau script en dupliquant un script existant, que l’on altère ensuite ici et là pour obtenir le résultat souhaité. Lorsque ce résultat est une simple variante du comportement du script initial, cette pratique conduit à un niveau très élevé de redondance du code, ce qui alourdit considérablement la maintenance : si un changement (correction de bug ou autre) doit être apporté dans une section de code reproduite dans plusieurs scripts, tous ces scripts doivent être modifiés, testés et revalidés ! La multiplication de code commun par « copier-coller » entraîne donc l’explosion des coûts de maintenance ultérieurs, c’est pourquoi elle doit absolument être évitée.

Ksh offre plusieurs mécanismes qui permettent la réutilisation de code :

* Le découpage des scripts : le script A peut appeler le script B qui appelle le script C, etc.
* L’inclusion de fichiers par la commande : . *fichier*
* Les listes d’arguments de la ligne de commande : $1, $2, …
* Le support des options sur la ligne de commande (cf. section 3.5 page 19)
* Les fonctions (cf. section 3.10 page 36).

Ces mécanismes favorisent un bon découpage des scripts et permettent d’éviter le « copier-coller ».

## Codes d’exit

|  |  |  |
| --- | --- | --- |
| PRG-02 | Obligatoire | Les codes d’exit à utiliser dans les scripts ksh sont :   * 0 : succès * 1 : échec – code général, sans notion de causalité particulière * de 2 à 125 : échecs consécutifs à des causes spécifiques   Les codes d’exit négatifs (-1, -2, etc.) sont interdits [[16]](#footnote-16). |

Explication : la règle ci-dessus découle de la convention en vigueur sous Unix : le code d’exit 0 signifie succès, ou encore « vrai » dans les enchaînements logiques ; les codes d’exit > 0 correspondent aux cas d’échec, ou « faux » dans les enchaînements logiques.

Il est tentant de définir de multiples codes d’exit pour caractériser différents cas d’échecs. Cependant ce type de nomenclature s’avère souvent peu pratique à l’usage ; un code d’exit « fourre-tout » est finalement retourné dans un grand nombre de situations non formellement prévues, notamment les cas d’échecs techniques. Par convention, on réserve le code d’exit 1 à ces situations.

Dans la plupart des scripts, les deux codes d’exit : 0 (succès) et 1 (échec, toutes causes confondues) sont suffisants. Utiliser des codes d’exit spécifiques (entre 2 et 125) ne se justifie que si le programme appelant doit traiter spécialement certains cas d’échecs, ou si l’on souhaite distinguer plusieurs cas de succès. Par exemple, un script de vérification d’arrivée de fichier pourra retourner :

* 0 si le fichier attendu est présent (succès)
* 2 si le fichier attendu est absent (en l’absence d’échec technique)
* 1 en cas d’échec technique : répertoire non trouvé, erreur de syntaxe d’appel, etc.

Cette distinction permettra au script appelant, ou à l’ordonnanceur, d’agir en fonction de la présence ou de l’absence du fichier attendu.

Les codes d’exit ≥ 126 sont réservés par le shell et ne doivent pas être utilisés à des fins applicatives :

* 126 : la commande appelée n’est pas un fichier exécutable.
* 127 : la commande appelée n’est pas trouvée.
* ≥ 128 : terminaison du processus par un signal ; le code d’exit est 128 + le numéro du signal. Exemple : 139 = 128 + 11, i.e. terminaison par le signal SIGSEGV (segmentation fault).

|  |  |  |
| --- | --- | --- |
| PRG-03 | Obligatoire | Les codes retournés par les fonctions ksh sont soumis aux mêmes règles que les codes d’exit des scripts – cf. règle ci-dessus. |

La discussion concernant les codes d’exit des scripts vaut également pour les valeurs de retour des fonctions, sachant que la commande return fonctionne pratiquement comme la commande exit, et qu’il n’y a pas de différence (aux effets de bords près) entre un appel de commande et un appel de fonction.

|  |  |  |
| --- | --- | --- |
| PRG-04 | Obligatoire | Les codes d’exit des scripts et les valeurs de retour des fonctions sont documentés dans leurs en-têtes respectifs. |

Exemple : en-tête de script ksh incluant la description des codes d’exit : cf. règle page 6.

|  |  |  |
| --- | --- | --- |
| PRG-05 | Conseillé | Les codes d’exit des scripts, ainsi que les valeurs de retour des fonctions, sont standardisés et définis dans des scripts de définitions communes. |

Cette pratique vise à rendre plus lisibles les scripts et à uniformiser l’utilisation des codes d’exit : un fichier définissant les codes d’exit est créé, et appelé dans tous les scripts.

Exemple :

Fichier de définition des codes d’exit :

# xyz\_exit\_codes.ksh

#

# Exit codes / return codes for the XYZ project

# Success

xyz\_success=0

# Non-failure codes

xyz\_file\_not\_found=2 # Expected file is not there  
 # Note: this is not a technical failure

# General failure code

xyz\_failed=1 # The script failed for some reason

# (to be diagnosed using accompanying messages)

# Specific failure codes

xyz\_syntax\_error=5 # Syntax error upon script invocation / bad arguments

xyz\_config\_error=10 # Error detected in configuration files

Script utilisant les codes d’exits communs :

# xyz\_do\_something.ksh  
#

# [*Note*: *en-tête omis pour simplifier*...]

# common exit codes

. ${APP\_HOME}/shells/xyz\_exit\_codes.ksh

# Check program call syntax

#

if [ $# -lt 2 ]; then

log\_error "Missing argument(s)."

print\_usage

exit ${xyz\_syntax\_error}

fi

[...]

Important : attention à ne pas commettre d’erreur de frappe dans le nom des paramètres définissant les codes d’exit ; ce type de bug doit être éliminé par relecture et par un test systématique des cas d’échecs.

## Utilisation des variables

|  |  |  |
| --- | --- | --- |
| VAR-01 | Recommandé | Si possible, utiliser l’option nounset afin de traiter l’évaluation d’un paramètre non défini comme une erreur d’exécution. |

Syntaxe :

set -o nounset

ou bien (synonyme) :

set -u

ou encore : invoquer ksh avec l’argument -u.

L’option nounset traite l’évaluation d’un paramètre non défini comme une erreur d’exécution. Étant donné que les erreurs d’initialisation de variables (ou plutôt d’absence d’initialisation) sont une source majeure de bugs dans les scripts shells, utiliser cette option est vivement recommandé.

Attention : quand l’option set -o nounset (ou set -u) est active, évaluer un paramètre non défini provoque un message d’erreur « *nom\_param*: parameter not set » sur la sortie des erreurs (stderr), suivi de l’abandon de l’unité d’exécution dans laquelle l’erreur s’est produite. Autrement dit, sauf cas particuliers [[17]](#footnote-17) :

* À l’intérieur d’une fonction, cette erreur provoque l’abandon de la fonction : le contrôle est retourné à l’appelant avec un code de retour égal à 1.
* Dans le script principal, cette erreur entraîne l’exit du script, avec un code d’exit égal à 1.

Important : l’option set -u devrait être considérée comme obligatoire pour tous les nouveaux scripts ksh. En revanche, appliquer cette option a posteriori sur des scripts écrits avec des règles moins strictes est assez risqué, car cela peut entraîner des échecs triviaux alors que les scripts concernés sont parfaitement fonctionnels sans cette option. À moins de pouvoir effectuer des tests de non-régression avec une couverture de tests de 100% (ce qui est pratiquement impossible), il vaut mieux éviter d’introduire cette option a posteriori sur des scripts déjà opérationnels.

Rappel : avec l’option nounset, pour substituer un paramètre susceptible de ne pas être défini, il faut utiliser la syntaxe ${*param*:-*default\_value*}, où *default\_value* est la valeur de remplacement si le paramètre *param* est non défini ou nul [[18]](#footnote-18).

Variante : ${*param*-*default\_value*} : substitue *default\_value* uniquement si le paramètre *param* est non défini ; substitue nul si *param* est nul.

Attention : avec AT&T ksh, les substitutions avec élimination de préfixe (${*param*#*pattern*} ou ${*param*##*pattern*}) ou de suffixe (${*param*%*pattern*} ou ${*param%%pattern*}) ne provoquent pas d’erreur si le paramètre *param* n’est pas défini, même quand l’option nounset est activée.

|  |  |  |
| --- | --- | --- |
| VAR-02 | Obligatoire | Les paramètres susceptibles de contenir des blancs doivent être protégés par des guillemets doubles afin qu’ils ne soient pas considérés comme une suite d’arguments distincts.  De la même façon, les paramètres susceptibles d’être nuls ou non définis, lorsqu’ils sont utilisés en tant qu’arguments, doivent être protégés par des guillemets doubles. |

Explication : le fait d’écrire "${*param*}" a pour conséquence d’empêcher que la partie entre guillemets doubles soit découpée après l’évaluation du paramètre *param* ; de plus si *param* est nul ou non défini, le résultat de l’évaluation sera égal à "" (i.e. un argument vide), alors que sans les guillemets l’évaluation ne produirait aucun résultat (exactement comme si on avait omis ${*param*} sur la ligne de commande).

Dans le même ordre d’idée, si un paramètre peut être interprété comme un motif de recherche de noms de fichiers (exemple : \*.???), il doit être mis entre guillemets si on souhaite éviter que le shell ne lui substitue les noms de fichiers correspondants.

Exemple : renommage d’un fichier : l’ancien nom et le nouveau nom sont désignés respectivement par les paramètres *old\_name* et *new\_name* ; si des blancs sont possibles dans les noms, il faut écrire :

mv "${old\_name}" "${new\_name}"

Les guillemets doubles garantissent que la commande mv recevra bien 2 arguments, ni plus ni moins. En l’absence de guillemets, le nombre d’arguments n’est pas garanti, et la commande risque de provoquer une erreur ou de produire un résultat différent de celui attendu par l’utilisateur.

|  |  |  |
| --- | --- | --- |
| VAR-03 | Recommandé | Pour tester si un paramètre est défini et non nul, utilisez :  if [ -n "${*param*:+X}" ]; then …  ou bien :  if [[ -n ${*param*:+X} ]]; then … (NB : avec les crochets doublés) |

Explications :

1. Dans la première variante ci-dessus (avec les crochets simples), les guillemets doubles sont obligatoires, cf. règle  ; au contraire, ils sont superflus dans la variante avec crochets doubles [[19]](#footnote-19).
2. A priori, il serait plus simple d’écrire : if [ "$*param*" != "" ]; then … ou : if [ -n "$*param*" ]; then …   
   Cependant ces formes peuvent provoquer des erreurs de syntaxe pour certaines valeurs de *param* dans certaines implémentations (anciennes) de ksh [[20]](#footnote-20).
3. Si l’option nounset est utilisée (cf. règle ), l’évaluation de $*param* provoque une erreur lorsque *param* n’est pas défini ; la forme ${*param*:+X} (si *param* est défini et non nul, remplace *param* par X, sinon remplace *param* par nul) évite cette erreur.
4. On peut aussi écrire : if [ "X${*param*:+X}" != "X" ]; then … Cependant cette forme est plus lourde et on lui préfère les 2 variantes recommandées car l’opérateur -n exprime plus clairement l’intention d’effectuer un test de non-nullité.

|  |  |  |
| --- | --- | --- |
| VAR-04 | Recommandé | *(Par analogie à la règle )*  Pour tester si un paramètre est nul ou non défini, utilisez :  if [ -z "${*param*:+X}" ]; then …  ou bien :  if [[ -z ${*param*:+X} ]]; then … [[21]](#footnote-21) |

|  |  |  |
| --- | --- | --- |
| VAR-05 | Recommandé | Pour tester si 2 paramètres de type chaîne de caractères sont égaux, utilisez :  if [ "X${*param1*:-}" = "X${*param2*:-}" ]; then … [[22]](#footnote-22)  ou bien :  if [[ "${*param1*:-}" = "${*param2*:-}" ]]; then … [[23]](#footnote-23) |

|  |  |  |
| --- | --- | --- |
| VAR-06 | Recommandé | Pour tester si un paramètre est conforme à un motif, utilisez :  if [[ ${*param*} = *motif* ]]; then … |

Notez l’emploi de la forme [[ … ]] pour le test ci-dessus ; cette forme permet de comparer une chaîne, à gauche du signe =, avec un motif, à droite du signe =, exprimé dans le langage des motifs de recherche des noms de fichiers. Notez que *motif* ne doit pas être entre guillemets, sinon la comparaison devient un simple test d’égalité de 2 chaînes de caractères. L’opérateur != inverse le test.

Exemples :

1) Comparaison d’un nom de fichier à un motif :

if [[ ${filename} != ${prefix}\_[0-9][0-9][0-9].@(txt|csv) ]]; then

# Nom de fichier non conforme !

echo "Erreur, \"${filename}\" n'est pas un fichier ${prefix}\_NNN.txt ou .csv"  
 ...

fi

2) Contrôle de numéricité : cf. fonction page 37.

Remarque : le motif objet du test peut inclure des substitutions de paramètres, telles que ${prefix} dans l’exemple ci-dessus, mais uniquement si leurs valeurs ne contiennent pas de parenthèse. Cette limitation empêche de variabiliser les motifs qui contiennent des parenthèses [[24]](#footnote-24).

### ATTENTION: ne marche pas avec ksh88

motif="${prefix}\_[0-9][0-9][0-9].@(txt|csv)"

if [[ $filename = $motif ]]; then ### <-- Ce test ne marche pas en ksh88 !!!

# Nom de fichier conforme ### (car $motif contient une parenthese...)

...

fi

Attention : la syntaxe du test de conformité à un motif est quasiment identique à celle du test d’égalité : seule l’absence des guillemets doubles autour de l’argument à droite de l’opérateur fait que celui-ci est traité comme un motif et non comme une chaîne. C’est pourquoi il peut être utile d’expliciter l’intention du programmeur par un commentaire précisant le test effectué, notamment si le motif est variabilisé.

|  |  |  |
| --- | --- | --- |
| VAR-07 | Recommandé | Pour tester si 2 paramètres de type entier, ou pouvant être interprétés en tant qu’entiers, sont égaux, utilisez :  if ((*param*1 == *param*2)); then … |

Comme indiqué à la règle STY-06, les expressions arithmétiques sont écrites en utilisant de préférence la syntaxe entre (( et )) ; cela inclut les opérateurs de comparaison usuels (<, <=, ==, !=, >=, >).

Le cas échéant les paramètres de type chaîne de caractères sont convertis en entier pour la comparaison :

typeset –Z5 foo=33 # Right-justify with leading zeros (largeur : 5 caractères)  
typeset –L5 bar=33 # Left-justify with trailing blanks (largeur : 5 caractères)

print "foo=[$foo] bar=[$bar]" # Affiche : foo=[00033] bar=[33 ]

if [[ "$foo" != "$bar" ]]; then # Les chaînes de caractères "00033" et "33 " sont distinctes

... # donc le test a la valeur logique « vrai »

fi

if ((foo == bar)); then # Les conversions en entier donnent 2 valeurs égales, donc le

... # test a la valeur logique « vrai »

fi

Attention : la confusion entre == (comparaison) et = (affectation) est une erreur classique :

foo=42  
bar=0

if ((foo = bar)); then ### <-- ERREUR : affectation et non pas comparaison !!!

print "égalité: $foo == $bar"

else  
 print "inégalité: $foo != $bar"

fi

Résultat : affiche inégalité: 0 != 0 ce qui n’est sans doute pas ce que le programmeur attendait.

|  |  |  |
| --- | --- | --- |
| VAR-08 | Conseillé | Les paramètres de type entier utilisés en tant qu’indices ou dans des expressions arithmétiques sont déclarés au moyen de la syntaxe :  typeset -i *param* (variante : typeset -i *param*=*expression*)  ou bien :  integer *param* (variante : integer *param*=*expression*) |

La commande typeset -i déclare que l’argument *param* doit être représenté en mémoire en tant qu’entier signé et non en tant que chaîne de caractères ; cette représentation accélère l’évaluation des expressions arithmétiques car elle évite des conversions de chaîne vers entier.

Le mot clef integer est un alias pré-défini de ksh pour typeset -i ; les 2 formes sont donc équivalentes.

Remarque : si le paramètre est et déclaré et initialisé simultanément, l’*expression* à droite du signe = est évaluée comme une expression arithmétique [[25]](#footnote-25).

## Contrôle des arguments

|  |  |  |
| --- | --- | --- |
| PRG-06 | Obligatoire | Les scripts vérifient, avant tout traitement, que leurs arguments sont valides et conformes à la syntaxe d’appel documentée. Si les arguments ou la syntaxe d’appel ne sont pas valides, aucun traitement n’est effectué : un message d’erreur explicite est émis, suivi de la fin immédiate du script avec un code d’exit correspondant à une erreur. |

La vérification de la syntaxe d’appel, ainsi que les contrôles de validité des arguments, sont indispensables pour les raisons suivantes :

1. S’il y a une possibilité d’erreur, vous pouvez être certain qu’elle sera commise, tôt ou tard.
2. Une erreur ne devrait pas entraîner de conséquence fâcheuse, ou le moins possible.
3. Les erreurs doivent être diagnostiquées au plus tôt, d’une part afin d’éviter qu’elles ne se propagent en cascade, d’autre part pour faciliter les analyses et les corrections le cas échéant.

Pour ces 3 raisons, il convient d’adopter une approche défensive vis-à-vis des arguments d’appel du script : dans la mesure où ces arguments sont reçus de l’*extérieur* ils ne doivent pas être considérés comme valides *a priori* ; au contraire, il faut « blinder » le script à tous points de vue contre les arguments erronés, manquants, ou en surnombre ; on doit donc vérifier :

* La présence et le nombre des arguments
* La conformité de la syntaxe d’appel au protocole documenté
* La validité des arguments : si un nombre est attendu, vérifier que l’argument est numérique, si un fichier est attendu, vérifier que le chemin est valide, etc.

Si une erreur quelconque est détectée, un message de diagnostic clair doit être émis à l’intention de l’utilisateur (ou du programmeur), puis le script doit se terminer avec un code d’exit signalant sans ambiguïté une situation d’erreur [[26]](#footnote-26).

Il est important que ces vérifications soient effectuées avant tout traitement, pour 2 raisons :

* D’une part, afin d’éviter (ou de limiter) les opérations de reprise nécessaires le cas échéant : toute erreur qui est détectable *a priori* doit être signalée et traitée *a priori*.
* D’autre part, parce que séparer les contrôles de présence et de validité des arguments de leur utilisation ultérieure contribue à un bon découpage des tâches.

## Gestion des options

|  |  |  |
| --- | --- | --- |
| OPT-01 | Conseillé | Si une nouvelle fonctionnalité peut être implémentée soit en créant un nouveau script, soit en enrichissant un script existant par de nouvelles options dans sa syntaxe d’appel, en général on privilégiera la 2ème solution, dans la mesure où elle favorise la réutilisation du code existant. |

A l’inverse, créer un nouveau script par « copier-coller » d’un script existant à seule fin d’introduire une variante dans ses paramètres, sa syntaxe d’appel ou son comportement, est une pratique à proscrire – cf. règle page 12.

Finalement, la seule situation dans laquelle l’ajout de nouvelles options à un script existant est peu souhaitable concerne les scripts déjà complexes que l’on préférerait éviter de compliquer davantage. En pareil cas, sachant que la duplication de code par « copier-coller » n’est pas une solution, généralement un *refactoring* s’impose : soit on simplifiera le script existant avant de lui ajouter de nouvelles options, soit on en extraira les parties réutilisables dans des fonctions (cf. section 3.10 page 36) qui seront communes à la nouvelle version et au script à créer.

|  |  |  |
| --- | --- | --- |
| OPT-02 | Recommandé | La commande built-in getopts [[27]](#footnote-27) est recommandée pour interpréter les options sur la ligne de commande. |

La commande getopts, qui est une commande built-in dans ksh, facilite le traitement de paramètres optionnels de la forme -*c* ou +*c* [[28]](#footnote-28) éventuellement suivis d’un argument.

Une discussion complète de la commande getopts dépasse l’ambition de ce document, cependant l’exemple ci-après illustre un cas d’utilisation typique.

Exemple : script acceptant les options -v (verbose), -t (test) et -l *logfile* (chemin du fichier de traces)

Remarque : le traitement des options est une *étape* dans le contrôle de la syntaxe d’appel ; pour la brièveté de l’exemple, seule cette étape est présentée ci-dessous.

001| #------------------------

002| # Traitement des options

003|

004| # Specification des options acceptees

005| #

**006|** opt\_spec=":tvl:"

007|

**008|** # Indicateur de presence / d’absence des options

009| #

010| opt\_v="" # verbose mode ("" / true <=> not set / set)

011| opt\_t="" # test mode ("" / true <=> not set / set)

012| opt\_l="" # path of log file

013|

014| opt\_err="" # error(s) detected ("" / true <=> no error / error(s))

015|

016| # Boucle de traitement des options

017| #

**018|** while getopts "${opt\_spec}" optname ; do

019| case $optname in

**020|** "v") opt\_v="true"

021| ;;

022| "t") opt\_t="true"

023| ;;

**024|** "l") opt\_l="$OPTARG"

025| ;;

**026|** "?") echo "ERROR, unknown option: $OPTARG"

027| opt\_err="true"

028| ;;

**029|** ":") echo "ERROR, argument is missing for option ${OPTARG:-}"

030| opt\_err="true"

031| ;;

032| esac

033| done

034|

**035|** shift $((OPTIND - 1))

036|

037| if [ -n "${opt\_err}" ]; then

038| # print usage and exit with an error status

039| usage\_exit 1

040| fi

Ligne 6 : le paramètre *opt\_spec*, utilisé à la ligne 18 en argument de getopts, définit les options acceptées ; le symbole deux-points (:) initial indique à getopts que le script ksh prend en charge la gestion des erreurs, c’est le mode de fonctionnement recommandé [[29]](#footnote-29) ; le symbole deux-points (:) qui suit la lettre l signifie que l’option -l attend un argument ; les lettres t et v ne sont pas suivies d’un tel symbole, ce qui indique que les options correspondantes ne prennent pas d’argument.

Ligne 8 et suivantes : on déclare les paramètres *opt\_xxx* qui seront utilisés comme indicateurs dans la boucle de traitement des options ; un paramètre *opt*\_c (avec *c* = t, v, ou l) est déclaré pour chaque lettre dans *opt\_spec*: ce paramètre indiquera si l’option correspondante est activée ou non ; dans le cas du paramètre *opt\_l*, si l’option -l *logfile* est activée ce paramètre sera valorisé par l’argument (*logfile*) de l’option -l [[30]](#footnote-30) ; on déclare également le paramètre *opt\_err*, indicateur de présence d’erreur(s) [[31]](#footnote-31).

Remarque : déclarer explicitement ces paramètres est non seulement utile si l’option set -o nounset est utilisée [[32]](#footnote-32), mais permet également de documenter ces paramètres, ce qui rend le script plus clair. Naturellement, si getopts est utilisée pour traiter les options d’appel d’une fonction (cf. section 3.10 page 36), toutes ces déclarations utiliseront la commande typeset, afin de déclarer des paramètres locaux à la fonction ; le paramètre *optname* à la ligne 18 sera également déclaré de cette manière.

Ligne 18 : la commande getopts prend 2 arguments : la chaîne définissant les options à accepter et le nom du paramètre à affecter en fonction de l’option rencontrée, *optname* dans notre exemple ; getopts, qui est toujours utilisée dans une boucle comme ci-dessus, analyse successivement les paramètres positionnels $1, $2, etc., et retourne les options trouvées l’une après l’autre.

Sont considérés comme des options (attendues ou non) tous les paramètres positionnels situés au début de la liste des paramètres positionnels et dont le 1er caractère est le signe moins (-) ou plus (+) ; getopts reconnaît également les paramètres positionnels utilisés en tant qu’arguments des options déclarées comme attendant un argument. La recherche s’arrête au premier paramètre positionnel qui ne commence pas par (‑) ou (+) et qui n’est pas non plus un argument d’une option attendant un argument [[33]](#footnote-33) ; getopts retourne alors une valeur logique fausse, ce qui termine la boucle de traitement des options.

Le corps de la boucle de traitement des options contient essentiellement une instruction *case*, où l’on gère à la fois les options acceptées et les erreurs.

Ligne 20 : getopts a reconnu l’option -v, qui n’attend pas d’argument ; on positionne le paramètre indicateur correspondant.

Ligne 24 : cas d’une option attendant un argument : ici getopts a reconnu l’option -l et son argument ; dans ce cas l’argument est disponible via le paramètre OPTARG.

Remarque : si l’option -l avait été reconnue mais non son argument (cas où l’argument est omis), le paramètre *optname* contiendrait le symbole deux-points (:) – cf. ligne 29.

Ligne 26 : lorsque getopts rencontre une option non reconnue, c-à-d. un argument qui serait une option valide si le caractère correspondant était mentionné dans la liste des options attendues, getopts affecte un point d’interrogation (?) à *optname* et le caractère concerné à OPTARG [[34]](#footnote-34).

Ligne 29 : lorsque que getopts reconnaît une option attendant un argument, si cet argument est omis getopts affecte le symbole deux-points (:) à *optname* et la lettre de l’option à OPTARG.

Ligne 35 : getopts affecte à OPTIND le numéro du prochain paramètre positionnel à examiner ; de sorte que, quand la boucle de traitement des options est terminée, OPTIND désigne le 1er paramètre qui n’est pas une option ; on utilise alors la commande shift pour « dépiler » les paramètres correspondant aux options, ce qui a pour effet de renuméroter à partir de $1 les paramètres positionnels correspondant aux arguments restant, s’il y en a. Le contrôle de la syntaxe d’appel peut alors prendre en compte à la fois les options et les arguments non optionnels.

Précisions :

1. Les options acceptées par getopts peuvent être introduites non seulement par le signe moins (-) mais aussi par le signe plus (+), auquel cas la valeur affectée à *optname* est aussi préfixée par ce caractère. Cela n’apparaît pas dans l’exemple ci-dessus, car nous avons fait des choix très simplificateurs, à savoir : (i) les options introduites par le signe plus ne font que désigner le comportement par défaut, exactement comme si elles étaient omises, et (ii) si les options +*c* et -*c* sont présentes simultanément, alors -*c* l’emporte quelque soit l’ordre dans lequel les options apparaissent. Il va de soi que dans d’autres situations, il faudrait traiter spécifiquement les options introduites par le signe plus [[35]](#footnote-35).
2. L’exemple ci-dessus ne gère pas les répétitions d’une même option : -l *logfile1* -l *logfile2* est traitée comme si seulement -l *logfile2* était spécifiée. Dans un script réel, il faudrait peut-être traiter cette situation comme un cas d’erreur.
3. La commande getopts reconnaît les options concaténées : ainsi, -tv est équivalent à -t -v ; les formes ‑l logfile et -llogfile (sans blanc intermédiaire) sont également équivalentes ; en revanche la forme ‑l=logfile n’est pas directement supportée [[36]](#footnote-36).

Attention : comme -l attend un argument, getopts interprète tout élément consécutif à -l, avec ou sans blanc intermédiaire, comme l’argument de -l ; de telle sorte que -l -t n’est pas traité comme l’option -l, avec omission de son argument, suivie de l’option ‑t, mais bien comme l’option -l avec ‑t  comme argument ; de la même façon, -lt est traité comme l’option -l avec la lettre t comme argument [[37]](#footnote-37). Ce point constitue une des principales limitations de getopts : la notation supportée est intrinsèquement ambiguë, sans que getopts permette de gérer cette ambiguïté. Selon les besoins, c’est-à-dire en tenant compte des conséquences potentielles en cas d’erreur de saisie, cette limitation sera considérée comme un inconvénient mineur ou rédhibitoire, nécessitant alors une autre syntaxe ou un autre mécanisme de reconnaissance des options.

Note : dans l’implémentation de Solaris 10, la commande getopts contient une extension permettant de définir des options « longues » (multi-caractères) introduites sur la ligne de commande par un double signe moins (--) ou un double signe plus (++). Dans l’exemple ci-dessus, il suffit de changer la définition du paramètre *opt\_spec* à la ligne 6 en :

opt\_spec=":t(test)v(verbose)l:(log)"

pour supporter des chaînes d’options telles que : --test --verbose --log=logfile (qui équivaut à : -t -v -l logfile). Aussi séduisante que cette fonctionnalité puisse paraître, il s’agit d’une extension propre à Solaris 10, donc non portable a priori, c’est pourquoi son utilisation n’est pas recommandée : si on souhaite supporter des options « longues » il semble préférable de les implémenter au moyen d’un mécanisme standard [[38]](#footnote-38).

|  |
| --- |
| Finalement, à quoi sert vraiment la commande getopts ?  Réponse : getopts simplifie l’écriture des boucles de traitement des options grâce aux fonctionnalités ci‑après :   * Affectation automatique des paramètres OPTIND et OPTARG * Détection de la chaîne « -- »  (double signe moins) qui termine la séquence d’options * Support de la concaténation des options : -xyz équivaut à -x -y -z   Il est tout-à-fait possible de programmer une boucle de traitement des options sans utiliser getopts, cependant cette boucle sera certainement un peu plus longue à écrire et un peu plus complexe ; cette solution doit donc être réservée aux rares situations dans lesquelles les limitations de getopts ne sont pas considérées comme acceptables. |

|  |  |  |
| --- | --- | --- |
| OPT-03 | Conseillé | Les scripts dont la syntaxe d’appel est non triviale acceptent l’option -h (*help*). Si cette option est utilisée, les scripts n’effectuent aucun traitement, mais affichent un résumé de leur syntaxe d’appel sur la sortie standard. |

L’affichage d’une aide en ligne si l’option -h est utilisée est une convention relativement répandue. Dans ce cas les scripts doivent bien sûr ignorer tous les autres arguments sur la ligne de commande.

## Détection des erreurs

|  |  |  |
| --- | --- | --- |
| PRG-07 | Obligatoire | Le code d’exit de chaque commande appelée est systématiquement contrôlé afin de s'assurer qu’elle s’est exécutée avec succès et d’agir en conséquence dans le cas contraire. Cette règle vaut également pour tous les programmes, fonctions, et autres scripts appelés dans les scripts ksh. |

Le code d’exit de la dernière commande exécutée par le shell (ou de la dernière fonction appelée s’il s’agit d’un appel de fonction) est stocké dans la variable $? ; cette variable doit être contrôlée *systématiquement* après chaque appel, selon la technique suivante :

*commande* # Appel à une fonction ou à une commande (built-in ou externe)

rc=$? # Affectation du code d’exit ($?) dans un paramètre local (rc) [[39]](#footnote-39)

if [ $rc –ne 0 ]; then

# echec

log\_error "Echec de *commande* avec le code d'exit $rc"

# action correctrice (ici, arrêt du script)

exit ${xyz\_failed}

fi

Important : pour construire des scripts robustes il faut partir du principe que *toutes* les commandes appelées peuvent échouer, y compris les plus basiques. Les scripts ksh doivent donc être assemblés en reproduisant la séquence ci-dessus pour toutes les commandes ou fonctions appelées ; en résumé :

1. Appel de commande ou de fonction
2. Copie du code d’exit ($?) dans un paramètre local
3. Test du code d’exit ; action correctrice en cas d’échec.

L’action correctrice à mettre en œuvre doit être déterminée en répondant à la question : à quoi sert de continuer l’exécution du script si la commande considérée a échoué ? Bien sûr, il n’y a pas de réponse systématique à cette question, toutefois dans au moins 90% des cas la réponse est que cela ne sert à rien ; il est alors préférable d’arrêter le script au plus tôt, tout en fournissant les messages d’erreur les plus complets et les plus détaillés possibles sur l’anomalie à l’origine de l’arrêt du script [[40]](#footnote-40). La réponse la plus simple à mettre en œuvre – l’arrêt du script aussitôt qu’une anomalie est détectée – est donc celle qui convient dans le plus grand nombre de situations. A contrario, omettre de vérifier les codes d’exit est une recette très sûre pour ne détecter les anomalies qu’après une cascade d’échecs, ce qui peut rendre le diagnostic en cas d’incident difficile voire impossible.

|  |  |  |
| --- | --- | --- |
| PRG-08 | Obligatoire | Les redirections d’I/O pouvant échouer, les codes d’exit des commandes redirigées sont systématiquement testés. |

Cette règle peut paraître redondante avec la précédente, mais il convient d’insister sur ce point : les redirections d’I/O sont des *ouvertures* ou des *créations* de fichiers, et peuvent donc échouer comme n’importe quelle autre commande.

|  |  |  |
| --- | --- | --- |
| PRG-09 | Recommandé | Les erreurs susceptibles de survenir dans les commandes lancées dans des pipelines doivent être prises en considération. |

Après une séquence de type pipeline telle que :

*commande\_1* | *commande\_2* | ... | *commande\_N*

le paramètre $? contient le code d’exit de la dernière commande dans le pipeline (*commande\_N*) ; il n’y a aucun moyen de récupérer les codes d’exit des autres commandes.

Cette limitation doit être prise en compte si des commandes en tête ou intermédiaires dans le pipeline peuvent échouer avec une probabilité non négligeable (suite à une erreur d’input, etc.) dans le cadre de l’exploitation normale du script. Le contrôle du code d’exit de ces commandes étant impossible, d’éventuelles erreurs risquent de passer inaperçues [[41]](#footnote-41).

Le cas échéant, ce type de considération peut conduire à remplacer un pipeline par une exécution séquentielle des commandes concernées :

*commande\_1* >*fichier\_tmp1*  
rc1=$?

# contrôle de $rc1 ...

*commande\_2* <*fichier\_tmp1* >*fichier\_tmp2*  
rc2=$?

# contrôle de $rc2 ...

...

*commande\_N <fichier\_tmp\_N-1*  
rc=$?

Bien que moins élégante, voire moins performante en raison du recours aux fichiers intermédiaires, cette solution offre un meilleur contrôle sur le déroulement des commandes impliquées dans le traitement.

Alternativement, si les commandes utilisées produisent des messages d’erreur sur la sortie stderr, on peut envisager la solution ci-après :

*commande\_1* 2>*fic\_err\_1* | *commande\_2* 2>*fic\_err\_2* | ... | *commande\_N* 2>*fic\_err\_N*

if [ -s *fic\_err\_1* ] || [ -s *fic\_err\_2* ] || ... || [ -s *fic\_err\_N* ] ; then  
 # Traiter les erreurs  
 ...  
fi

Dans cette solution, le pipeline est conservé mais la sortie stderr de chaque commande est redirigée vers un fichier. On vérifie a postériori que ces fichiers sont vides pour s’assurer qu’aucune erreur n’est survenue.

Important : cette solution n’est pas parfaite : en cas de saturation du système de fichiers, il peut très bien arriver que les fichiers *fic\_err\_1*, *fic\_err\_2* … *fic\_err\_N* soient créés mais demeurent vides, non parce qu’aucune erreur ne s’est produite, mais bien parce que la saturation du système de fichiers a empêché l’écriture des messages d’erreurs (cf. encadré concernant les commandes [echo et print](#Builtin_echo_print_et_erreurs), page 30).

## Gestion des erreurs par le shell

NB : cette section ne définit aucune règle, mais vise à rappeler le comportement de ksh dans certaines situations d’erreur. Il est important de bien connaître ces situations afin de prévenir et de traiter les erreurs, dans la mesure du possible.

### Paramètres non définis (rappels)

Si l’option set -o nounset (ou set -u, qui est équivalente) n’est pas utilisée, évaluer un paramètre non défini n’est pas une erreur : l’évaluation ne produit aucun résultat. Exemples :

*commande* *arg1* ${*param\_non\_defini*} *arg3* # équivaut à : *commande* *arg1 arg3*

*commande* *arg1* "${*param\_non\_defini*}" *arg3* # équivaut à : *commande* *arg1* "" *arg3*

# (avec un 2ème argument nul)

Si l’option set -o nounset est activée (recommandé), évaluer un paramètre non défini entraîne, sauf cas particuliers décrits ci-après :

1. un message « *param*: parameter not set » sur la sortie des erreurs (stderr), puis :
2. l’abandon immédiat de l’unité d’exécution courante, avec un code d’exit égal à 1.

Cas particuliers : avec AT&T ksh, si cette erreur affecte la définition du fichier de redirection d’une commande built-in, le traitement dépend de la commande concernée : si c’est une commande built-in *spéciale* [[42]](#footnote-42), l’erreur provoque un exit du script ; si c’est une built-in ordinaire, l’erreur entraîne un simple échec de la commande. Autrement dit, AT&T ksh traite cette erreur comme un échec de redirection. Au contraire, pdksh ne fait pas cette distinction et traite toujours cette erreur de la même façon.

Le tableau ci-dessous récapitule les différents cas et leur traitement par AT&T ksh et pdksh.

| Cas d’erreur : paramètre *param* non défini (avec l’option nounset activée) | Traitement de l’erreur par le shell | |
| --- | --- | --- |
| AT&T ksh | pdksh |
| Cas général : évaluation de ${*param*} | Abandon de l’unité d’exécution courante. | Abandon de l’unité d’exécution courante. |
| Cas particulier 1 : paramètre non défini affectant le fichier de redirection dans un appel d’une commande built-in *spéciale* :  Exemple : exec 2>"${*param*}" | Exit du script | Abandon de l’unité d’exécution courante. |
| Cas particulier 2 : paramètre non défini affectant le fichier de redirection dans un appel d’une commande built-in *régulière* :  Exemple : print -u2 "toto" 2>"${*param*}" | Échec de la commande print, sans abandon de l’unité d’exécution courante. | Abandon de l’unité d’exécution courante. |
| Cas particulier 3 : évaluation d’un des opérateurs :  ${#*param*} ${*param*%*word*} ou ${*param*%%*word*}  ${*param*#*word*} ou ${*param*##*word*} | AT&T ksh ne traite pas ce cas comme une erreur : ${#*param*} s’évalue à 0, les 4 autres substitutions produisent un résultat nul. | Abandon de l’unité d’exécution courante. |

En contexte d’évaluation arithmétique, l’évaluation d’un paramètre non défini est traitée comme une erreur par AT&T ksh dans tous les cas – indépendamment de l’option nounset – contrairement à pdksh qui évalue silencieusement les paramètres non définis à 0.

| Cas d’erreur : évaluation d’un paramètre non défini dans un contexte arithmétique (NB : même résultat avec ou *sans* l’option nounset.) | Traitement de l’erreur par le shell | |
| --- | --- | --- |
| AT&T ksh | pdksh |
| Le paramètre *param* n’étant pas défini, évaluation d’une expression telle que :  x=$((*param* + 1)) ou : let x=*param*+1 | Message « *param* + 1: bad number » puis abandon de l’unité d’exécution courante. | Pas d’erreur : l’expression est évaluée à 1. |

Rappel : évaluer un paramètre non défini avec un des opérateurs dans le tableau ci-dessous, qui effectuent un test sur ce paramètre, n’entraîne pas d’erreur :

| Expression | Résultat de l’expression | | |
| --- | --- | --- | --- |
| Paramètre défini et non nul | Paramètre défini et nul | Paramètre non défini |
| ${*param*:-*word*} | Valeur de *param* | Valeur de *word* | Valeur de *word* |
| ${*param*-*word*} | Valeur de *param* | Résultat nul | Valeur de *word* |
| ${*param*:+*word*} | Valeur de *word* | Résultat nul | Résultat nul |
| ${*param*+*word*} | Valeur de *word* | Valeur de *word* | Résultat nul |

### Commandes built-in spéciales

Il existe 2 catégories de commandes built-in [[43]](#footnote-43) dans ksh :

1. Les commandes built-in ordinaires (ou régulières), qui se comportent exactement comme des commandes ordinaires
2. Les commandes built-in « spéciales », ainsi nommées car elles diffèrent des autres commandes par certains aspects de leur comportement, notamment le traitement des erreurs.

Le tableau ci-après liste les commandes built-in couramment utilisées.

|  |  |
| --- | --- |
| Commandes built-in spéciales  En cas d’erreur d’exécution :  - AT&T ksh : exit du script  - pdksh : échec de l’unité d’exécution courante | Commandes built-in ordinaires  Même traitement des erreurs que pour les commandes ordinaires. |
| **.** *file* [ *arg* ]  break [ *n* ]  continue [ *n* ]  eval [ *arg* … ]  exec [ *arg* … ]  exit [ *n* ]  export [ *name*[=*value*] ] …  return [ *n* ]  shift [ *n* ]  trap [ *arg* *sig* … ]  typeset [ *options* ] [ *name*[=*value*] ] …  wait [ *job* ] [[44]](#footnote-44) | cd [ -L ] [ -P ] [ *arg* ]  cd *old* *new*  echo [ *arg* … ]  kill [ -*sig* ] *pid* …  print [ *arg* … ]  pwd  read [ -prsu[ *n* ] ] [ *name* ... ]  set [ ±*flags* ] [ ±o *option* ] ... [ ±A *name* ] [ *arg* ... ]  test *expression* |

Les erreurs lors des appels de commandes built-in spéciales sont traitées de façon particulière :

* AT&T ksh : le manuel précise sobrement : « *errors cause a script that contains them to abort.* », ce qui signifie qu’en cas d’erreur d’exécution le script sera terminé purement et simplement, sans pouvoir ni détecter l’erreur a posteriori, ni la traiter, puisqu’il sera arrêté immédiatement.
* pdksh : la formule du manuel : « *an error during their execution can cause a non-interactive shell to exit* » est plus ambiguë. Cependant, en général pdksh traite ces erreurs par un abandon de l’unité d’exécution courante, non par un exit du script.

L’emploi des commandes built‑in spéciales nécessite donc des précautions, tout spécialement avec AT&T ksh, afin d’éviter les erreurs *a priori* (dans la mesure du possible).

Exemple : la commande typeset est une commande built-in spéciale, c’est pourquoi (avec AT&T ksh) il est recommandé d’écrire :

typeset -i num # déclare le paramètre num de type entier signé

num=$param # affecte à num la valeur du paramètre param

plutôt que :

typeset -i num=$param # déclare le paramètre num de type entier signé et lui  
 # affecte (simultanément) la valeur du paramètre param

En effet, si l’affectation entraîne une erreur de conversion de la valeur de *param*, dans le 1er cas cette erreur a lieu au moment de l’affectation, alors que dans le 2nd cas elle a lieu lors de l’appel de la commande typeset. C’est pourquoi, dans le 1er cas cette erreur entraîne seulement l’abandon de l’unité d’exécution courante, alors que dans le 2nd cas (avec AT&T ksh) elle entraîne l’arrêt immédiat du script.

### Traitement des erreurs par les commandes built-in eval et exec

Les commandes built-in eval et exec sont des commandes built-in spéciales ce qui entraîne (rappel) qu’une erreur lors de leur exécution peut provoquer un arrêt du script. L’emploi de ces commandes nécessite une grande vigilance :

* L’argument de eval doit être une expression ksh valide sur le plan syntaxique et sémantique.
* L’argument de exec doit être une commande valide.
* Les fichiers de redirection éventuels doivent pouvoir être ouverts sans erreur.

Faute de quoi l’échec de la commande eval ou exec pourra entraîner un exit du script. Ici encore, AT&T ksh et pdksh ne se comportent pas de la même façon, comme précisé dans le tableau ci-après :

| Cas d’erreur | Traitement de l’erreur par le shell | |
| --- | --- | --- |
| AT&T ksh | pdksh |
| Erreur de syntaxe dans l’argument de eval :  Exemple : eval 'if true ; then echo "True."'  (il manque «  ; fi » pour terminer l’expression) | Exit du script. (\*) | Abandon de l’unité d’exécution courante. |
| La commande eval entraîne l’évaluation d’un paramètre non défini (avec l’option nounset activée) :  Exemple : eval 'echo ${parametre\_non\_defini}' | Exit du script. (\*) | Abandon de l’unité d’exécution courante. |
| Commande exec avec argument non accessible :  Exemple : exec /nosuchdir/bin/nosuchcmd | Exit du script. (\*\*) | Exit du script. (\*\*\*) |
| (\*) Les traps sur EXIT sont déclenchés.  (\*\*) Les traps sur EXIT sont déclenchés, à l’exception de celui de la fonction dans laquelle l’erreur se produit.  (\*\*\*) Le trap sur EXIT du script n’est pas déclenché. | | |

### Erreurs dans les expressions arithmétiques

AT&T ksh traite les erreurs en contexte d’évaluation arithmétique en émettant un message d’erreur sur la sortie stderr, suivi d’un abandon de l’unité d’exécution courante. Le comportement de pdksh est un peu moins clair, car il dépend de l’erreur rencontrée.

| Cas d’erreur : évaluation d’un paramètre non numérique dans un contexte arithmétique | Traitement de l’erreur par le shell | |
| --- | --- | --- |
| AT&T ksh | pdksh |
| foo=42zz  ((bar = foo)) | Message « 42zz: bad number » puis abandon de l’unité d’exécution courante. (\*) | Message « 42zz: bad number `42zz' » puis abandon de l’unité d’exécution courante. (\*) |
| foo=42zz  ((bar = $foo)) | Message « 42zz: bad number » puis abandon de l’unité d’exécution courante. (\*\*) | Message « bar = 42zz: bad number `42zz' » sans abandon de l’unité d’exécution ($? vaut 2). (\*) |
| (\*) Le paramètre bar n’est pas modifié.  (\*\*) Le paramètre bar reçoit la valeur 0. | | |

| Cas d’erreur : division par zéro | Traitement de l’erreur par le shell | |
| --- | --- | --- |
| AT&T ksh | pdksh |
| div=0  ((x = 42 / *div*)) | Message « x = 42 / *div*: divide by zero » puis abandon de l’unité d’exécution courante. (\*) | Message « x = 42 / div: zero divisor » sans abandon de l’unité d’exécution ($? vaut 2). (\*\*) |
| (\*) Le paramètre x reçoit la valeur 0.  (\*\*) Le paramètre x n’est pas modifié. | | |

| Cas d’erreur : évaluation d’un paramètre non défini dans un contexte arithmétique [rappel] (NB : même résultat avec ou *sans* l’option nounset.) | Traitement de l’erreur par le shell | |
| --- | --- | --- |
| AT&T ksh | pdksh |
| Avec le paramètre *param* non défini :  ((x = *param* + 1)) | Message « x = *param* + 1: bad number » puis abandon de l’unité d’exécution courante. | Pas d’erreur : l’expression est évaluée à 1. |

### Traitement des erreurs de redirection

En cas d’erreur de redirection, c’est-à-dire si le fichier spécifié ne peut pas être *ouvert*, les commandes ordinaires échouent, i.e. elles retournent un code d’exit non nul, mais l’unité d’exécution en cours (fonction ou script principal) continue.

Le cas des commandes built-in spéciales[[45]](#footnote-45) est particulier :

* Avec AT&T ksh, un échec de redirection au moment de l’appel d’une commande built-in spéciale entraîne un exit du script.
* Avec pdksh, un échec de redirection lors de l’appel d’une commande built-in spéciale entraîne seulement l’abandon de l’unité d’exécution courante.

Ce comportement est propre aux commandes built-in spéciales : un échec de redirection au moment de l’appel d’une commande built-in régulière est traité comme dans le cas d’une commande ordinaire.

Les échecs de redirection survenant lors des appels de fonction, ou lors d’appels de commandes composées de la forme { *liste* }, sont également particuliers : avec AT&T ksh, un échec de redirection lors de tels appels provoque l’abandon de l’unité d’exécution courante. Au contraire, pdksh traite ces commandes comme des commandes ordinaires.

Exemple : commande composée de la forme { *liste* } avec redirection de stdin :

# Fonction de lecture d’un fichier

function read\_file {  
 typeset input\_file="$1"

typeset rc  
 { while read line ; do

# faire qqch avec $line

...

done  
 } <"${input\_file}" # AT&T ksh : abandon de la fonction en cas d'erreur d’ouverture !  
 # pdksh : la fonction continue même si l’ouverture du fichier échoue.

rc=$?

if [ $rc -ne 0 ]; then

# traitement des erreurs :

# - AT&T ksh : ce point n’est pas atteint si l’ouverture du fichier a échoué.

# - pdksh : ce point est atteint en cas d’échec d’ouverture du fichier.

...

fi

}

Important : une erreur d’*écriture* dans un fichier ouvert par redirection n’est pas un échec de redirection : une fois le fichier de redirection *ouvert* avec succès par le shell, son utilisation par les commandes appelées est transparente pour le shell ; le traitement des erreurs d’écriture dans le fichier de redirection dépend donc entièrement de la façon dont les commandes appelées traitent les erreurs d’entrées/sorties. En pratique cela signifie que l’écriture dans un fichier de redirection (ouvert avec succès par le shell) peut échouer sans qu’il soit possible de détecter cette erreur au niveau du script ksh !

|  |
| --- |
| Exemple des commandes echo et print : les commandes echo et print renvoient toujours 0, sauf en cas d’échec de redirection, c’est-à-dire si le fichier de redirection ne peut pas être *ouvert* ; dans ce cas la commande échoue avec un code d’exit non nul. Dans tous les autres cas, ces commandes retournent 0, même si l’écriture échoue – cela se produit typiquement en cas de saturation du système de fichiers [[46]](#footnote-46).  Conséquence : si un fichier est généré par redirection des sorties d’un script ksh, même si le code d’exit du script est égal à 0 (succès), il n’est pas garanti à 100% que le fichier produit soit complet. En effet des commandes echo ou print appelées par le script peuvent avoir échoué « silencieusement », sans qu’il soit possible de détecter ces erreurs au niveau du script ksh. Si l’intégrité du fichier généré est importante, il faut obligatoirement mettre en place des vérifications complémentaires, telles qu’un contrôle a posteriori du nombre de lignes dans le fichier généré [[47]](#footnote-47). |

## Chemins d’accès aux commandes et aux fichiers

|  |  |  |
| --- | --- | --- |
| PRG-10 | Recommandé | Les fichiers lus / écrits par les scripts shells sont désignés par des chemins complets ; l’utilisation de chemins relatifs est fortement déconseillée. |

L’objectif à atteindre est de rendre les scripts shells indépendants :

* Du répertoire courant du processus (*working directory*)
* De l’organisation des répertoires utilisés dans l’application, ainsi que de leurs noms.

Ceci afin de ne pas créer de contrainte d’exploitation :

* Les scripts doivent fonctionner quels que soient les répertoires depuis lesquels ils sont appelés.
* L’organisation des répertoires de l’application doit pouvoir évoluer, sans que cela impose de modifications conjointes des scripts.

Cette indépendance ne peut être atteinte que si les chemins désignant les fichiers et les répertoires sont systématiquement :

1. Des chemins complets, et
2. Spécifiés au moyen de variables d’environnement.

La définition de ces variables doit être centralisée dans un fichier d’environnement applicatif, qui est soit inclus directement par tous les scripts, soit inclus dans le fichier $HOME/.profile du user applicatif.

Exemple de fichier d’environnement applicatif :

# APPLICATION

# XYZ -- the XYZ project

#

# FILE

# profile.xyz

#

# DESCRIPTION

# This file exports the environment variables which are needed

# by scripts and programs in the XYZ application.

#

# NOTE: this file deals only with application-specific variables.

# Environment variables which pertain to base software components

# (e.g. ORACLE, WEBLOGIC, etc.) are defined elsewhere [[48]](#footnote-48).

#

# USAGE

# This file is meant to be included (sourced) from the .profile

# file of the application user. Usually it is not necessary to

# source it again from individual scripts [[49]](#footnote-49).

#

# IMPORTANT

# DO NOT USE KSH-SPECIFIC FEATURES IN THIS SCRIPT!

# This script must remain compatible with Bourne shell as it may

# be sourced from Bourne shells scripts [[50]](#footnote-50).

#

# Root directory for the XYZ application

#

XYZ\_APP\_ROOT\_DIR=/apps/xyz

export XYZ\_APP\_ROOT\_DIR

# Root directory for XYZ software components

#

XYZ\_APP\_INST\_DIR=${XYZ\_APP\_ROOT\_DIR}/current\_version

export XYZ\_APP\_INST\_DIR

# Directory of XYZ scripts

#

XYZ\_APP\_SCRIPT\_DIR=${XYZ\_APP\_INST\_DIR}/scripts

export XYZ\_APP\_SCRIPT\_DIR

[...]

# Directories for log files

#

XYZ\_LOG\_DIR\_MAIN=${XYZ\_LOG\_ROOT\_DIR}/last # Today's log files

export XYZ\_LOG\_DIR\_MAIN

[...]

# Directory for just-received daily files

#

XYZ\_DATA\_RECV\_DIR=${XYZ\_DATA\_ROOT\_DIR}/recv/daily

export XYZ\_DATA\_RECV\_DIR

[...]

Le principe de base est de définir avec précision la fonction de chaque répertoire de l’application, puis de créer une variable d’environnement pour chacune des fonctions ainsi définies, désignant le répertoire qui remplit cette fonction. Ainsi les répertoires applicatifs, comme /apps/xyz/scripts, /apps/xyz/data/recv/daily ou encore /logs/xyz/last, ne sont jamais référencés directement dans les scripts, mais uniquement via les variables d’environnement correspondantes, dans notre exemple : ${XYZ\_APP\_SCRIPT\_DIR}, ${XYZ\_DATA\_RECV\_DIR} et ${XYZ\_LOG\_DIR\_MAIN}. Les scripts conformes à ce principe sont indépendants aussi bien du répertoire courant que de la structure de répertoires de l’application : si cette structure change, seul le fichier d’environnement applicatif doit être modifié.

|  |  |  |
| --- | --- | --- |
| PRG‑11 | Obligatoire | Les commandes externes ou les scripts invoqués dans les scripts ksh sont toujours appelés en utilisant leur chemin complet. |

Utiliser systématiquement le chemin complet des commandes externes présente un avantage essentiel : les scripts qui respectent cette règle sont indépendants du PATH, donc non susceptibles de voir leur comportement modifié en cas de modification du PATH. Ce point est très important sur un système tel que Solaris où, pour des raisons de compatibilité, certaines commandes existent en plusieurs implémentations dans des spécifications différentes ; ces variantes sont installées dans des répertoires distincts (/usr/bin, /usr/xpg4/bin, etc.) mais leurs noms sont identiques [[51]](#footnote-51). Un simple changement de l’ordre des répertoires du PATH peut donc avoir pour conséquence que les scripts qui n’utilisent pas les chemins complets (et ne définissent pas eux-mêmes le PATH) basculent d’une implémentation à une autre [[52]](#footnote-52). Étant donné que la variable PATH est héritée du processus parent, à moins de redéfinir soi-même cette variable il est risqué de faire des suppositions concernant sa valeur : rien ne garantit qu’elle sera identique en production et en développement, ni qu’elle sera stable dans le temps. Finalement, il n’y a que 2 solutions :

* Soit redéfinir la variable PATH
* Soit utiliser systématiquement les chemins complets.

Redéfinir la variable PATH n’est pas la solution recommandée car :

* D’une part ce n’est pas toujours faisable : si votre script appelle d’autres scripts ou des programmes tiers, pouvez-vous prédire quel sera l’effet de la modification du PATH sur ces programmes ?
* D’autre part une telle modification affecte toutes les commandes appelées dans le script [[53]](#footnote-53).

Utiliser les chemins complets des commandes n’améliore évidemment pas la lisibilité des scripts, mais il est très facile d’y remédier en définissant des paramètres constants pour chacune des commandes utilisées :

CP=/usr/bin/cp

GREP=/usr/xpg4/bin/grep

PRINTF=/usr/bin/printf

RM=/usr/xpg4/bin/rm

Ces définitions sont à ajouter aux initialisations en début de script (cf. règle page 6) ou dans un fichier de définitions communes inclus en début de script.

Dans la suite du script, tous les appels de commandes sont faits en utilisant les paramètres correspondants :

$CP -p "$fich" "${fich\_tmp}"

$GREP -v '^[[:space:]]\*$' "${fich\_tmp}" >"$fich"

$RM -f "${fich\_tmp}"

Cette solution préserve la lisibilité du script, tout en offrant une certaine flexibilité pour des changements ultérieurs dans le choix de telle ou telle implémentation d’une commande donnée.

Attention : l’emploi des chemins complets ne concerne bien sûr que les commandes externes. Au contraire, les appels de commandes built-in ne doivent jamais mentionner de chemin – sinon, ce sont bien des commandes externes qui seront utilisées, à la place des commandes built-in !

Remarque : à la casse majuscule près (utilisée car il s’agit de paramètres constants), les noms utilisés sont identiques aux noms des commandes référencées : c’est la seule situation où l’emploi de noms identiques à ceux de commandes existantes du système est préconisé (tout autre usage de ces noms est source de confusion et donc très fortement déconseillé).

En ce qui concerne les appels de commandes ou de scripts applicatifs, la pratique recommandée est de les appeler par leurs noms, en indiquant leur chemin complet au moyen d’une variable désignant le répertoire dans lequel ils sont installés [[54]](#footnote-54) :

# Appel du script "xyz\_do\_something.ksh" de l’application XYZ,

# installé dans le répertoire ${XYZ\_APP\_SCRIPT\_DIR}

${XYZ\_APP\_SCRIPT\_DIR}/xyz\_do\_something.ksh

|  |  |  |
| --- | --- | --- |
| PRG-12 | Obligatoire | Le lancement de commandes ou de scripts via des chemins relatifs est interdit. |

Les chemins relatifs doivent impérativement être évités, car :

1. Les scripts qui utilisent des chemins relatifs dépendent pour leur bon fonctionnement du répertoire courant du processus ($PWD) : si ce répertoire n’est pas celui attendu, de tels scripts risquent de ne pas fonctionner correctement. Pour éviter cet écueil, les scripts sensibles au répertoire courant doivent bien sûr changer eux-mêmes le répertoire courant – cela ne doit pas être au programme appelant de le faire !
2. Les chemins relatifs dépendent implicitement de l’organisation des fichiers et des répertoires dans l’application. Si cette organisation est modifiée, les chemins relatifs risquent de plus être valides. Autrement dit les chemins relatifs sont des constructions fragiles et difficiles à maintenir.

Comme développé dans la règle précédente, la bonne pratique consiste à utiliser systématiquement des chemins complets spécifiés indirectement via des paramètres du shell ; cette solution est non seulement robuste mais également flexible et maintenable.

## Fichiers de traces

|  |  |  |
| --- | --- | --- |
| TRC-01 | Recommandé | Les traces d’exécution des scripts ksh sont écrites sur la sortie standard (stdout), ou sur la sortie des erreurs (stderr) dans le cas des commandes produisant leurs résultats (*output*) sur stdout.  L’utilisation simultanée de stdout et de stderr pour les traces est déconseillée. |

Cette règle est essentiellement une question de simplicité : ce n’est pas parce que Unix offre 2 sorties standards, une pour l’*output* usuel, l’autre pour les erreurs, qu’il faut obligatoirement utiliser les deux. Si votre script n’affiche rien d’autre que des traces d’exécution, le plus simple est d’émettre tous les messages, y compris les erreurs, sur stdout ; de sorte que l’utilisation de stderr pour les messages d’erreurs n’est justifiée que pour les scripts qui produisent un *résultat* sur stdout, auquel cas il est bien évident que ce résultat ne doit pas être pollué par des messages d’erreurs.

|  |  |  |
| --- | --- | --- |
| TRC-02 | Recommandé | Les scripts ksh n’effectuent pas de redirection de stdout / stderr pour renvoyer les traces applicatives vers des fichiers.  Cas particulier : les redirections peuvent être utilisées pour générer des fichiers de traces *en* *complément* des traces écrites sur stdout / stderr. |

Par principe, les traces applicatives doivent être écrites sur les sorties standards (stdout / stderr) afin de laisser l’appelant, ou l’utilisateur en cas d’utilisation interactive du script, libre de choisir où ces traces doivent être écrites : sur l’écran du terminal, dans un fichier, ou les deux à la fois, ou vers /dev/null, etc. Au contraire, un script qui effectue une redirection (avant d’appeler un autre script ou une commande) supprime toute possibilité de choix : les sorties vont là où l’auteur du script l’a décidé et nulle part ailleurs. Cela peut être utile voire nécessaire dans certains cas, mais c’est souvent un mauvais choix en ce qui concerne les traces applicatives étant donné l’absence de flexibilité qui en résulte.

Cette règle admet toutefois une exception notable : si un fichier de log doit être créé *systématiquement*, quelle que soit la façon dont le script est lancé (que cela soit par un ordonnanceur ou en mode interactif), dans ce cas il peut être nécessaire d’émettre les traces applicatives *à la fois* vers stdout / stderr et vers un fichier de log créé automatiquement par le script. Les redirections pourront bien sûr être utilisées pour écrire les messages dans ce fichier de log.

|  |  |  |
| --- | --- | --- |
| TRC-03 | Recommandé | Les traces d’exécution des scripts sont aussi verbeuses que nécessaire. En particulier, le début et la fin des étapes de traitement sont tracées, ainsi que toutes les erreurs rencontrées. |

On ne saurait trop insister sur l’importance des traces d’exécution pour le support applicatif, en particulier en cas d’incident ou d’analyse d’anomalie. Elles doivent être aussi complètes que possible, ce qui inclut :

* Messages de pré-traitement : rappels des paramètres d’appel et des options utilisées ; ce rappel permet de vérifier que les paramètres ont été correctement saisis ou transmis, et que les options ont été correctement employées par l’appelant et interprétées par le script.
* Messages en cours de traitement : le début et la fin de toutes les étapes significatives sont tracés, afin de renseigner sur le déroulement du traitement ; cela permet non seulement de connaître les actions réellement accomplies par le traitement (très utile en cas de dépannage ou de débogage), mais également d’en vérifier la progression (détecter les traitements gelés ou en boucle infinie).
* Messages d’erreur ou d’avertissement : toute erreur ou circonstance inattendue doit être tracée – cf. règle ci-dessous.
* Messages de fin de traitement : bilan d’exécution, plus ou moins détaillé en fonction du besoin : succès / échec, nombre de fichiers traités, présence d’erreurs ou d’avertissements, etc.

|  |  |  |
| --- | --- | --- |
| TRC-04 | Recommandé | Toute erreur ou circonstance inattendue, telle que l’échec d’une commande, l’absence d’un fichier en entrée, etc., doit être détectée et tracée. |

Ce point est très important pour la résolution des incidents et le débogage : si le traitement est arrêté prématurément à cause d’une erreur, il est indispensable que toutes les informations utiles à la compréhension de l’incident (et si possible à sa résolution) aient été tracées au préalable.

Le programmeur ksh doit anticiper les situations d’erreur, en partant du principe que toute opération est susceptible d’échouer ; il faut donc détecter systématiquement les erreurs et les traiter, *a minima* en s’assurant que des messages précis et complets sont enregistrés avant tout autre action [[55]](#footnote-55) [[56]](#footnote-56).

Dans le même esprit, dès qu’un problème mineur est détecté, il est recommandé d’enregistrer un message d’avertissement : même si de tels messages sont souvent anodins par définition, dans certaines situations ils pourront s’avérer très utiles pour comprendre d’éventuelles erreurs survenant en aval.

|  |  |  |
| --- | --- | --- |
| TRC-05 | Recommandé | Les messages de début et de fin des étapes de traitement sont horodatés. |

L’horodatage des messages facilite le contrôle de la progression du traitement, ainsi que, le cas échéant, la détection des étapes trop longues ou gelées [[57]](#footnote-57).

|  |  |  |
| --- | --- | --- |
| TRC-06 | Recommandé | Utilisez autant de fichiers de traces que nécessaire pour permettre une exploitation facile des traces dans le cadre du support applicatif.  Créer un fichier de traces distinct pour chaque lancement de chaque job ou script est généralement considéré comme adéquat. |

Si un fichier de traces distinct est utilisé pour chaque *itération* d’un job, ou pour chaque lancement d’un script, il est facile d’accéder aux traces d’une itération particulière si les fichiers de traces sont nommés de manière explicite, par exemple d’après le nom du script et le numéro d’itération (ou l’horodatage [[58]](#footnote-58)). Au contraire, si les traces des itérations successives sont enregistrées dans un seul fichier, l’analyse des traces est plus difficile car il faut d’abord distinguer les traces de l’itération que l’on souhaite analyser.

Le principal inconvénient de ce système est qu’il peut conduire à créer de très nombreux fichiers dans les répertoires contenant les fichiers de traces ; outre la mise en place, indispensable, d’une purge des fichiers de traces, il peut être utile d’organiser les traces dans plusieurs répertoires [[59]](#footnote-59).

## Utilisation des fonctions

|  |  |  |
| --- | --- | --- |
| FUN-01 | Recommandé | Découpez les traitements en utilisant des fonctions. |

Le Korn shell supporte la programmation structurée à travers le mécanisme des fonctions, qui ont leur propre liste de paramètres positionnels ($1, $2, etc.), peuvent déclarer des variables locales à l’aide de la commande typeset, et retournent un code d’exit à l’appelant, implicitement ou explicitement via la commande return. Les fonctions peuvent être invoquées comme des commandes dans n’importe quelle fonction (y compris récursivement), ou dans le script principal à partir de leur point de définition [[60]](#footnote-60) [[61]](#footnote-61).

Exemple de fonction ksh :

# FUNCTION

# check\_isnum

#

# DESCRIPTION

# Tests whether the argument is numeric, i.e. is

# a string of digits with an optional sign character

# (+|-), possibly with leading and trailing blanks.

#

# ARGUMENTS

# $1: argument to be checked

#

# RETURNS

# 0 if the argument is numeric, as defined above

# > 0 if the argument is not numeric, as defined above

function check\_isnum {

# arg #1 = value to be tested

typeset arg="$1"

[[ "$arg" = \*([$IFS])?([-+])+([0-9])\*([$IFS]) ]]

}

La fonction ci-dessus détermine si son argument est numérique, c’est-à-dire composé uniquement d’une suite de chiffres précédée d’un signe facultatif, éventuellement suivie ou précédée de blancs [[62]](#footnote-62) ; ci-après nous définissons une 2ème fonction afin de tester la première.

function test\_check\_isnum {

typeset arg="$1"

if check\_isnum "$arg" ; then

echo "Argument \"${arg}\" is numeric."

else

echo "Argument \"${arg}\" is NOT numeric!"

fi

}

On peut ainsi valider la fonction check\_isnum à l’aide d’une suite de tests telle que :

test\_check\_isnum "0" # numeric

test\_check\_isnum "1234567890" # numeric

test\_check\_isnum " -123" # numeric with leading blanks and sign

test\_check\_isnum "+123

" # numeric with trailing blanks, including "\n"

test\_check\_isnum "#!/bin/ksh" # not numeric (obviously)

test\_check\_isnum " " # not numeric: only blanks

test\_check\_isnum "" # not numeric: empty string

test\_check\_isnum "-" # not numeric: no digits

test\_check\_isnum " + " # not numeric: no digits

test\_check\_isnum "123.4" # not numeric: non-digit char

test\_check\_isnum "123 456" # not numeric: non-digit char

Résultats :

Argument "0" is numeric.

Argument "1234567890" is numeric.

Argument " -123" is numeric.

Argument "+123

" is numeric.

Argument "#!/bin/ksh" is NOT numeric!

Argument " " is NOT numeric!

Argument "" is NOT numeric!

Argument "-" is NOT numeric!

Argument " + " is NOT numeric!

Argument "123.4" is NOT numeric!

Argument "123 456" is NOT numeric!

Cet exemple illustre à quel point les fonctions permettent de structurer et ainsi de simplifier les scripts ksh.

En résumé, les fonctions ksh peuvent être utilisées pour :

* Simplifier les scripts en décomposant les tâches complexes
* Masquer la complexité au programme appelant, en la confinant à l’intérieur des fonctions
* Éviter la duplication de code et créer des modules réutilisables.

Inutile de préciser que leur utilisation est plus qu’encouragée.

|  |  |  |
| --- | --- | --- |
| FUN-02 | Obligatoire | Les fonctions sont déclarées en utilisant le mot-clef *function*. Les fonctions déclarées dans la syntaxe Bourne Shell / POSIX sont interdites. |

Il existe 2 syntaxes distinctes pour définir une fonction :

1. La syntaxe ksh, introduite par le mot-clef function :

function *nom\_fonction* {

*commande\_composée*

}

1. La syntaxe compatible avec le Bourne shell, également appelée syntaxe POSIX :

*nom\_fonction*() {

*commande\_composée*

}

La syntaxe Bourne shell / POSIX ne doit pas être utilisée, pour les raisons suivantes :

1. Elle est moins lisible que la syntaxe ksh
2. Sur un plan sémantique, des différences subtiles existent entre les fonctions définies dans la syntaxe Bourne shell / POSIX et celles définies dans la syntaxe ksh, notamment :
   1. Les traps sur EXIT définis au niveau des fonctions (AT&T ksh uniquement [[63]](#footnote-63)) ne sont supportés que dans les fonctions non-POSIX – cf. règle TRAP-05 page 49.
   2. Dans pdksh, les paramètres OPTARG et OPTIND ne sont spéciaux que dans les fonctions définies dans la syntaxe ksh ; dans les fonctions Bourne shell / POSIX, ces paramètres perdent leur caractère spécial, ce qui interdit d’utiliser la commande built-in getopts dans ces fonctions (cf. section 6.4.5 page 66).

Bien que les différences soient subtiles et limitées à des points de fonctionnalité précis, ces limitations des fonctions déclarées dans la syntaxe Bourne shell / POSIX conduisent à proscrire cette syntaxe [[64]](#footnote-64).

Remarque : dans la suite de ce document, la syntaxe ksh (utilisant le mot-clef function) est appelée syntaxe « non-POSIX » ; de même les fonctions définies selon cette syntaxe sont dites « non-POSIX ».

|  |  |  |
| --- | --- | --- |
| FUN-03 | Recommandé | Utilisez la commande typeset pour créer des variables locales aux fonctions. |

La commande typeset, quand elle est utilisée à l’intérieur d’une fonction, définit des variables locales, dont l’existence cesse lors du retour à l’appelant. Si ces variables sont nommées avec des noms déjà utilisés ailleurs, les définitions locales masquent les définitions antérieures jusqu’au retour de la fonction.

Exemple :

001| function foo {

002|  typeset var1="FOO"

003|  typeset var2

004|  echo "In function foo, var1 = $var1"

005|  echo "In function foo, var2 = ${var2-undefined}"

006|  bar

007| }

008|

009| function bar {

010|  echo "In function bar, var1 = $var1"

011|  echo "In function bar, var2 = ${var2-undefined}"

012| }

013|

014| var1="MAIN"

015| var2="main"

016|

017| foo

018| bar

019|

020| echo "In main script, var1 = $var1"

021| echo "In main script, var2 = $var2"

Aux lignes 2 et 3 ci-dessus, la fonction foo définit 2 paramètres locaux, var1 et var2, qui masquent les paramètres de mêmes noms définis aux lignes 14 et 15 dans le script principal. Il est important de noter que la portée de ces définitions est dynamique : les paramètres var1 et var2 définis aux lignes 2 et 3 existent jusqu’au retour de la fonction foo ; ils sont donc visibles dans la fonction bar lorsque celle-ci est appelée depuis la fonction foo ; quand la fonction bar est appelée directement depuis le script principal, les définitions en vigueur sont celles des lignes 14 et 15.

Résultats :

001| In function foo, var1 = FOO

002| In function foo, var2 = undefined

003| In function bar, var1 = FOO

004| In function bar, var2 = undefined

005| In function bar, var1 = MAIN

006| In function bar, var2 = main

007| In main script, var1 = MAIN

008| In main script, var2 = main

Utiliser systématiquement des variables locales à l’intérieur des fonctions est une pratique recommandée, afin que les variables utilisées à l’intérieur des fonctions ne « polluent » pas l’ensemble du script.

Cette pratique admet 2 principales exceptions :

1. Les fonctions qui doivent transférer un résultat à l’appelant : comme une fonction ne peut renvoyer qu’un code d’exit (par la commande return), la seule solution pour transférer un résultat est de l’affecter à un paramètre non local.
2. Les fonctions dont le comportement est *influencé* par des paramètres non locaux (soit des variables d’environnement, soit des paramètres définis ailleurs dans le script) que l’on ne souhaite pas propager en tant qu’arguments des fonctions afin de ne pas surcharger les listes d’arguments (l’accès à ces paramètres est uniquement en lecture).

En dehors de ces 2 cas, l’utilisation de la commande typeset pour définir des variables locales doit constituer un véritable réflexe.

Attention (rappel) : la commande typeset est une commande built-in spéciale, c’est pourquoi on évitera (avec AT&T ksh) de regrouper en une seule instruction la *déclaration* avec typeset et l’*affectation*, lorsque cette dernière est susceptible de provoquer une erreur d’exécution (cf. page 28).

|  |
| --- |
| Avertissement : en ksh93, les variables définies par la commande typeset à l’intérieur des fonctions ont une portée statique (et non plus dynamique). Ce point constitue une différence majeure entre ksh88 et ksh93 [[65]](#footnote-65). |

|  |  |  |
| --- | --- | --- |
| FUN-04 | Obligatoire | La règle page 10 s’applique également à l’intérieur des fonctions : les paramètres positionnels ($1, $2, etc.) seront systématiquement copiés dans des variables locales, utilisées en lieu et place de ces paramètres. |

A l’intérieur des fonctions, il faut utiliser la commande typeset afin de déclarer un paramètre local pour chaque argument, et lui affecter la valeur du paramètre positionnel correspondant. Les paramètres positionnels sont ainsi remplacés par des variables locales nommées, ce qui rend le code plus lisible.

Exemple :

function do\_something {

typeset srcdir="$1" # arg 1: source directory

typeset dstdir="$2" # arg 2: destination directory

typeset fpattern="$3" # arg 3: pattern for matching filenames

# ...

}

Dans le cas de fonctions réutilisables faisant partie d’un framework, l’affectation des paramètres positionnels est généralement précédée d’un contrôle de présence des arguments et suivie de vérifications de la validité des arguments. Ainsi, la fonction ci-dessus s’écrirait plutôt sous la forme :

function do\_something {  
 typeset funnam="do\_something" # name of this function

# Check call syntax, return 1 if not ok

if (($# != 3)); then

echo "ERRROR, $funnam: expecting 3 arguments, got $#"

return 1

fi

typeset srcdir="$1" # arg 1: source directory

typeset dstdir="$2" # arg 2: destination directory

typeset fpattern="$3" # arg 3: pattern for matching filenames

# Check arguments

if [ ! -d "$srcdir" ]; then

echo "ERROR, $funnam: source dir not found: \"$srcdir\""

return 1

fi

if [ ! -d "$dstdir" ]; then

echo "ERROR, $funnam: destination dir not found: \"$dstdir\""

return 1

fi

# Make sure we can create files in destination dir.

if [ ! –w "$dstdir" ]; then

echo "ERROR, $funnam: missing write permission: \"$dstdir\""

return 1

fi

# ...

}

Ici encore, le code est incomparablement plus lisible sous cette forme que si l’on avait écrit partout $1 à la place de $srcdir et $2 à la place de $dstdir.

|  |  |  |
| --- | --- | --- |
| FUN-05 | Recommandé | Favorisez la réutilisation des fonctions en créant un *framework* de fonctions communes réutilisables. |

L’expérience montre que les développeurs shell sont confrontés de manière récurrente à des besoins technico-fonctionnels basiques auxquels ksh ne répond pas nativement. En particulier, l’obligation de concevoir des scripts robustes, incluant une détection des erreurs et des traces d’exécution adéquates, fait qu’un script ksh bien écrit ne se réduit pas à une simple suite d’appels de commandes : même une fonctionnalité élémentaire nécessite quelques lignes de code pour les besoins du traitement des erreurs et de la gestion des traces. C’est ici qu’interviennent les fonctions réutilisables : certains besoins étant entièrement récurrents, faute de fonctions réutilisables les mêmes fonctionnalités de base sont sans cesse ré-implémentées, au prix d’un fort allongement des scripts dont la maintenabilité diminue en proportion. Il est donc indispensable d’identifier ces fonctionnalités de base, puis de les implémenter dans un ensemble de fonctions réutilisables, ayant vocation à servir de *framework* pour l’ensemble de l’application. Cette démarche est particulièrement utile en début de projet.

Les domaines susceptibles d’être traités dans un framework de fonctions réutilisables sont multiples :

* Le contrôle des arguments : tests de numéricité, de valeurs logiques, de dates, etc.
* La gestion des traces (remplacement des commandes built-in echo et print)
* Les opérations usuelles sur les fichiers (sélection du plus récent, copie, déplacement, purge, etc.)
* La gestion des fichiers de configuration (cf. section 4.2 page 53)
* L’interrogation des bases de données ou l’exécution de requêtes SQL.

Attention : le succès d’un framework dépend de plusieurs facteurs :

* Adéquation aux besoins
* Simplicité d’utilisation
* Robustesse de l’implémentation
* Qualité de la documentation

Les exigences et le niveau de qualité requis font que le développement de fonctions réutilisables destinées à un framework (alternativement, le choix d’un framework, si on s’oriente vers la reprise d’un framework déjà existant) ne doit pas être entrepris à la légère : il est facile d’écrire une fonction ad-hoc utilisée plusieurs fois dans un script. En tirer une fonction réutilisable dans un cadre plus large impose de traiter un nombre bien plus élevé de questions relatives au design, à la gestion des traces, au traitement des erreurs, sans oublier le contrôle des arguments en entrée.

Un autre point à ne pas perdre de vue est la stabilité, indispensable, des interfaces : une fois qu’une fonction commune est disponible dans un framework, il peut devenir très difficile d’en changer les spécifications ou le comportement sans affecter la compatibilité avec les scripts clients du framework. C’est pourquoi les fonctions réutilisables doivent être spécifiées et décrites avec beaucoup de rigueur, en ayant à l’esprit que ces spécifications constituent un contrat potentiellement irrévocable avec les scripts appelants.

|  |  |  |
| --- | --- | --- |
| FUN-06 | Recommandé | Les fonctions communes réutilisables sont documentées méticuleusement. |

La documentation des fonctions réutilisables est un facteur clef pour la qualité des développements utilisant un framework de fonctions réutilisables. Nous proposons ci-après un exemple de documentation d’une fonction pouvant servir de modèle.

# FUNCTION

# get\_file\_time

#

# DESCRIPTION

# This function retrieves the last modification time

# of the file specified as argument (or optionally,

# its creation time).

#

# OPTIONS

# -c get the file creation time

# -m get the file modification time (default)

# -f format

# format string to use as output

# (default: "%Y-%m-%d %H:%M:%S")

# -q quiet mode: don't log any message if the file time

# cannot be retrieved.

#

# ARGUMENTS

# $1 path of the argument file

#

# RETURNS

# 0 if the file time was successfully retrieved

# >0 if the file time was not retrieved succesfully

#

# GLOBAL PARAMETERS

# \_\_get\_file\_time\_seconds

# The file time as seconds since Epoch  
# (unset if not successful)

# \_\_get\_file\_time\_str

# The file time as a formatted string

# (unset if not successful)

#

La description de chaque fonction doit comporter les mentions ci-après :

1. Le nom de la fonction.
2. Une brève description du traitement accompli par la fonction. Cette description doit être précise et concise ; elle n’est pas nécessairement exhaustive, sachant qu’elle est généralement complétée par les autres rubriques. Si la complexité du traitement l’exige, elle pourra être précisée par une description plus détaillée, placée après la présentation des arguments et des options.
3. La liste et la description des arguments optionnels supportés par la fonction [[66]](#footnote-66).
4. La liste et la description détaillée des paramètres d’appel de la fonction.
5. L’énoncé des pré-requis, s’il y en a [[67]](#footnote-67).
6. La description des valeurs de retour de la fonction
7. Le détail des paramètres globaux créés, modifiés, ou simplement lus par la fonction [[68]](#footnote-68)
8. Le cas échéant, toute autre mention utile concernant d’éventuels effets de bord de la fonction (événements loggués, possibilité d’exit du script en cas d’échec, etc.).

|  |  |  |
| --- | --- | --- |
| FUN-07 | Conseillé | Utilisez la commande built-in getopts pour gérer les options dans la syntaxe d’appel des fonctions. |

Comme l’illustre l’exemple de la règle précédente, il est assez fréquent d’introduire des arguments optionnels dans la syntaxe d’appel des fonctions afin de supporter des variantes dans le traitement qu’elles accomplissent. C’est une pratique recommandée dans la mesure où elle favorise la réutilisabilité.

La commande built-in getopts peut être utilisée pour traiter ces options, de la même façon que pour traiter les options du script principal – cf. règle page 20. De plus le paramètre OPTIND est un paramètre spécial de ksh [[69]](#footnote-69) : sa valeur est sauvegardée automatiquement avant chaque appel de fonction et est restaurée au retour (sans qu’il soit nécessaire de le déclarer avec typeset). On peut ainsi utiliser getopts aussi bien dans les fonctions appelées que dans le script principal, sans risque d’interférence [[70]](#footnote-70).

## Utilisation des tableaux

|  |  |  |
| --- | --- | --- |
| ARR-01 | Obligatoire | Les scripts qui utilisent des tableaux doivent tenir compte des limites sur la taille des tableaux en ksh. |

Dans AT&T ksh sous Solaris, la taille des tableaux est limitée à 4096 éléments ; dans pdksh, cette limite est de 1024 éléments seulement [[71]](#footnote-71) ; si la compatibilité avec pdksh est recherchée, on devra donc se contenter de tableaux de 1024 éléments au maximum.

En pratique, cette limite pose rarement problème, mais elle ne peut pas toujours être ignorée. Le plus fréquemment, de par la nature du problème à traiter, le nombre d’éléments attendus est très inférieur à 1024 éléments ; dans ce cas (si on est vraiment certain que cette limite ne sera jamais approchée) il n’est pas nécessaire de mettre en place un contrôle de limite. En revanche, s’il existe une possibilité d’atteindre ou de dépasser cette limite, un contrôle sur le nombre d’éléments s’impose : un message applicatif explicite est toujours préférable à une erreur « *subscript out of range* » en cours de traitement [[72]](#footnote-72). Par ailleurs, si un dépassement de capacité peut se produire autrement que dans des circonstances exceptionnelles, il faut se demander si l’algorithme choisi est adéquat, voire si un langage autre que ksh ne serait pas mieux indiqué.

|  |  |  |
| --- | --- | --- |
| ARR-02 | Conseillé | Éviter les tableaux « creux ». |

En ksh, les tableaux sont :

* Soit déclarés et initialisés avec l’instruction set -A
* Soit déclarés implicitement lorsqu’un indice de tableau est utilisé.

set -A tabfoo 'Hello' 'World !' # déclaration du tableau tabfoo, avec initialisation

echo "${tabfoo[1]}" # affiche : World !

set -A tabfoo # ré-initialisation de tabfoo à vide (pas de liste de valeurs)

tabbar[0]='Hello' # affectation dans tabbar[0], déclare implicitement le tableau tabbar

tabbar[1]='World !' # affectation dans tabbar[1]

Cependant rien n’interdit d’utiliser des indices non consécutifs. Exemple :

set -A tabfoo

tabfoo[13]='Hello'

tabfoo[42]='World !'

echo "${tabfoo[13]} ${tabfoo[42]}" # affiche : Hello World !

echo "${#tabfoo[\*]}" # affiche : 2 (nombre d’éléments définis dans le tableau)

Un tableau est dit « creux » si les indices de ses éléments ne sont pas consécutifs. L’utilisation des tableaux creux en ksh est peu conseillée, pour les raisons suivantes :

* Ce type de tableau est inhabituel : vous risquez de surprendre les futurs mainteneurs [[73]](#footnote-73)
* Dans un tableau creux, la relation entre la plage d’indices valides et le nombre d’éléments est perdue : dans un tableau normal, les indices des éléments vont de 0 à ${#*tableau*[\*]} - 1 ; dans un tableau creux, cette relation n’est pas valide.

La boucle de parcours ci-dessous ne fonctionne qu’avec des tableaux normaux :

# Boucle de parcours des éléments d’un tableau

# ATTENTION, cette boucle ne marche pas avec les tableaux creux !

typeset -i idx # indice de boucle

idx=0

while ((idx < ${#tabfoo[\*]})); do # ${#tabfoo[\*]} est le nombre d’éléments dans tabfoo

# faire quelque chose avec l’élément ${tabfoo[idx]} [[74]](#footnote-74)

...

((idx += 1))

done

Un parcours itératif des tableaux creux demeure possible, mais sans relation avec les indices :

# Boucle de parcours des éléments d’un tableau

#

# Cette boucle fonctionne aussi bien avec les tableaux creux qu’avec

# les tableaux normaux, mais la relation aux indices est perdue.

for elemt in "${tabfoo[@]}" ; do

# faire quelque chose avec l’élément $elemt

...

done

Cette boucle exploite le fait que l’expression "${*tableau*[@]}" – attention, entre guillemets doubles ! – est spéciale dans la mesure où elle est équivalente à la *liste* "${*tableau*[*n*1]}" "${*tableau*[*n*2]}" … "${*tableau*[*nN*]}", où *n*1, *n*2, … *nN* sont les indices des éléments du tableau [[75]](#footnote-75).

Exemple :

set -A tabfoo

tabfoo[13]='Hello World!'

tabfoo[42]='How do you do?'

for elemt in "${tabfoo[@]}" ; do

echo "== $elemt =="

done

affiche les 2 lignes ci-après :

== Hello World! ==

== How do you do? ==

## Utilisation des traps

Avertissement : les traps constituent un sujet « avancé » et fortement dépendant de l’implémentation. En particulier, des différences significatives existent entre AT&T ksh et pdksh.

|  |
| --- |
| A quoi servent les traps ?  Les traps servent à exécuter des commandes ksh en réaction :   * Soit à la réception de signaux Unix : HUP, TERM, etc. * Soit à des événements propres à l’exécution du script ksh : pseudo-signaux DEBUG, ERR et EXIT.   La commande : trap *command* *signal* [*signal…*] permet de définir la commande à exécuter lors de l’occurrence de l’événement (cf. man ksh(1)).  Les traps couramment utilisés sont :   1. Les traps sur EXIT au niveau des fonctions dans AT&T ksh (cf. infra) 2. La commande : trap "" HUP pour ignorer le signal HUP (hangup) 3. Les traps sur signaux (TERM, USR1, USR2, etc.) dans certains cas particuliers [[76]](#footnote-76).   Les cas 1 et 2 sont de loin les plus fréquents. |

|  |  |  |
| --- | --- | --- |
| TRAP-01 | Recommandé | Les scripts qui utilisent des traps et doivent fonctionner aussi bien avec pdksh qu’avec AT&T ksh doivent tenir compte des différences significatives entre les deux implémentations. |

Le tableau ci-après synthétise le comportement des traps dans AT&T ksh et pdksh, ainsi que leurs principales différences ; d’autres différences sont listées à la section 6.3 page 64.

|  | AT&T ksh | pdksh |
| --- | --- | --- |
| Traps et fonctions | Le script principal et les fonctions définissent des traps distincts, propres à chaque niveau dans la pile d’appel.  Les traps définis par l’appelant ne sont pas hérités par l’appelé, à l’exception des signaux ignorés par l’appelant qui le sont également par l’appelé. | Les traps sont globaux : un trap défini reste effectif pour la durée du script, qu’il soit défini dans une fonction ou non (tant qu’il n’est pas remplacé par une nouvelle définition).  Les traps restent effectifs dans les fonctions appelées, mais non dans les sous-processus (à l’exception des signaux Unix ignorés). |
| Traps sur EXIT | Les traps sur EXIT du script principal et des fonctions sont distincts (propres à chaque niveau), mais uniquement si les fonctions non-POSIX sont utilisées ; sinon, le trap sur EXIT est global et déclenché uniquement lors de l’exit du script [[77]](#footnote-77).  Les traps sur EXIT sont déclenchés en cas de sortie sur erreur. | Le trap sur EXIT est global et déclenché uniquement lors de l’exit du script.  Le trap sur EXIT n’est pas déclenché en cas de sortie consécutive à une erreur d’exécution. |
| Traps sur ERR | Les traps sur ERR du script principal et des fonctions sont distincts. | Le trap sur ERR est global ; il reste effectif dans les fonctions appelées. |
| Traps sur DEBUG |  | Non implémenté : erreur d’exécution si utilisé. |
| Traps sur signaux Unix, appels de fonctions et sous-processus | Signaux Unix traités par l’appelant : l’action par défaut est rétablie dans les fonctions appelées et dans les sous-processus.  Signaux Unix ignorés par l’appelant : les fonctions appelées et les sous-processus ignorent également ces signaux ; on ne peut pas définir de traps pour traiter des signaux déjà ignorés par l’appelant ou au démarrage du processus ksh. | Signaux Unix traités : le trap est global et effectif pour la durée du script, y compris dans les fonctions appelées. L’action par défaut est rétablie dans les sous-processus.  Signaux Unix ignorés : le trap est global et défini pour la durée du script (ou jusqu’à une nouvelle définition). Les signaux ignorés sont hérités par les sous-processus. Un script ne peut pas définir de trap sur un signal déjà ignoré lors du démarrage du processus pdksh. |
| Traitement effectué lors de la réception d’un signal Unix | En fonction du trap en place dans le contexte (script / fonction) en cours d’exécution :  - Signal traité par le niveau en cours d’exécution : le trap est exécuté dès que possible [[78]](#footnote-78) dans le contexte courant.  - Action par défaut : la fonction en cours d’exécution est interrompue, dès que possible  et le signal est propagé à l’appelant ; et ainsi de suite jusqu’à atteindre un niveau dans lequel le signal est traité (le script se termine si aucun niveau ne traite ce signal) ; au retour du trap, la fonction interrompue par le signal a un code d’exit égal à 128 + le numéro du signal.  - Signal ignoré : rien ne se passe, exactement comme si le signal n’avait pas été reçu. | En fonction du trap en place au niveau du processus :  - Signal traité : le trap est exécuté dès que possible  dans le contexte courant.  - Action par défaut : le processus est terminé.  - Signal ignoré : rien ne se passe. |

Conclusion : la gestion des traps et des signaux est plus sophistiquée dans AT&T ksh que dans pdksh, dans la mesure où elle prend en compte les fonctions : des traps distincts existent à chaque niveau de la pile d’appel, contrairement à pdksh où les traps sont définis globalement pour le processus et déclenchés indépendamment du contexte. Lorsqu’un signal est reçu et traité par un processus pdksh, une fois le trap exécuté le flot d’exécution se poursuit normalement (en supposant que le trap ne provoque pas d’exit) ; dans AT&T ksh au contraire, le signal provoque un débranchement depuis les fonctions dans lesquelles l’action par défaut est en vigueur jusqu’au niveau qui traite le signal ; ce niveau doit donc obligatoirement prendre en compte la possibilité que les fonctions qu’il appelle soient interrompues par le signal attendu.

|  |  |  |
| --- | --- | --- |
| TRAP-02  pdksh | Recommandé | Les scripts devant s’exécuter sous pdksh ne doivent pas définir de trap sur EXIT à l’intérieur des fonctions. |

La version de ksh d’AT&T permet de définir un trap sur EXIT à l’intérieur d’une fonction non-POSIX, c’est-à-dire une fonction définie par le mot-clef *function* ; un tel trap est exécuté lorsque la fonction se termine (que cela soit normalement ou à la suite d’une erreur d’exécution) [[79]](#footnote-79).

En ce qui concerne les fonctions définies dans la syntaxe Bourne shell / POSIX, si on définit un trap sur EXIT à l’intérieur d’une telle fonction, il s’agit en fait du trap sur EXIT du script principal, déclenché en principe lors de la terminaison du script et non au retour de la fonction concernée.

Dans pdksh, bien que la distinction entre les fonctions non-POSIX et les fonctions définies dans la syntaxe Bourne shell / POSIX existe, seul le trap sur EXIT du script est disponible : si un trap sur EXIT est défini à l’intérieur d’une fonction, en réalité cette définition concerne la terminaison du script et non le retour de la fonction : quand la fonction est exécutée, la nouvelle définition du trap sur EXIT prend effet, annulant et remplaçant la définition antérieure s’il y en avait une ; bien sûr le trap sur EXIT n’est pas exécuté au retour de la fonction, mais seulement lorsque le script se termine [[80]](#footnote-80).

C’est pourquoi il est recommandé de ne pas définir de trap sur EXIT à l’intérieur des fonctions dans les scripts conçus pour s’exécuter avec pdksh :

* D’une part, de tels traps n’auraient pas le même comportement avec pdksh et AT&T ksh
* D’autre part, même si l’intention du programmeur est vraiment de redéfinir le trap sur EXIT du script depuis une fonction particulière, un tel usage risque de surprendre les futurs mainteneurs.

|  |  |  |
| --- | --- | --- |
| TRAP-03  pdksh | Obligatoire | Les scripts devant s’exécuter sous pdksh ne doivent pas dépendre du déclenchement des traps sur EXIT en cas d’erreur d’exécution. |

AT&T ksh et pdksh diffèrent ici de façon significative : si la terminaison du script est provoquée par une erreur d’exécution [[81]](#footnote-81), pdksh n’exécute pas le trap sur EXIT du script ; celui-ci n’est donc exécuté qu’en cas de terminaison normale, suite à un return ou à un exit explicite ou implicite. Au contraire, AT&T ksh exécute le trap sur EXIT quelle que soit la cause de terminaison (cf. cas de test à la section 6.3.3 page 65).

Il faut tenir compte de ce point lors de la conception du script : si la compatibilité avec pdksh est recherchée, sachant que pdksh n’exécute pas le trap sur EXIT en cas de terminaison consécutive à une erreur d’exécution, non seulement il n’est pas recommandé de dépendre des traps sur EXIT pour des tâches critiques, mais il est absolument interdit d’en dépendre pour traiter d’éventuelles erreurs.

|  |  |  |
| --- | --- | --- |
| TRAP-04  AT&T ksh | Obligatoire | Les scripts qui utilisent des traps sur EXIT dans des fonctions ne doivent pas dépendre du déclenchement de ces traps en cas de réception d’un signal. |

En cas de réception d’un signal par le script, en règle générale il ne faut pas compter sur le déclenchement des traps sur EXIT des fonctions, à moins que ce signal ne soit ignoré ou traité par un trap spécifique.

Si l’action par défaut en cas de réception d’un signal, c-à-d. l’arrêt du processus, est en place, dans le meilleur des cas un seul trap sur EXIT sera exécuté : celui de la fonction en cours d’exécution lorsque le signal est reçu, si cette fonction a défini un trap sur EXIT [[82]](#footnote-82) ; dans le cas contraire le processus sera terminé sans qu’aucun trap sur EXIT ne soit déclenché.

Si le signal est traité par un trap dans le script : soient les fonctions f1, f2, … f*n*, munies chacune de leur propre trap sur EXIT ; f1 appelle f2 qui appelle f3, etc. ; on suppose que le signal est traité par un trap dans la fonction f1, l’action par défaut restant en place dans les fonctions f2, f3, … f*n* ; dans ce cas, si le signal est reçu alors que f*n* est en cours d’exécution, les fonctions f2, f3 … f*n* sont toutes terminées par la réception du signal ; l’exécution reprend au niveau de f1 par le déclenchement du trap traitant le signal ; seul le trap sur EXIT de la fonction f2 est déclenché au préalable.

Bien évidemment, si le signal est ignoré, sa réception n’entraîne aucun effet, dès lors que la partie du script en cours d’exécution lorsque le signal est reçu ignore ce signal. Par exemple, soient les fonctions f1, f2, … f*n*, où f1 appelle f2 qui appelle f3, etc. ; on suppose que le signal est traité par un trap dans la fonction f1, mais qu’il est ignoré dans la fonction f2 ; ce qui entraîne qu’il l’est également dans les fonctions f3 … f*n* [[83]](#footnote-83) ; dans ce cas, si le signal est reçu alors qu’une des fonctions f2 … f*n* est en cours d’exécution, le signal est ignoré, malgré le trap positionné au niveau de f1 [[84]](#footnote-84). Dans ce cas, les traps sur EXIT fonctionnent normalement puisque le flot d’exécution n’est pas affecté.

Enfin, en cas de réception du signal SIGKILL, bien évidemment aucun trap n’est déclenché puisque ce signal provoque l’arrêt immédiat du processus, sans qu’il soit possible de l’intercepter.

En résumé, les situations où le déclenchement des traps sur EXIT des fonctions est garanti alors un signal est reçu par le processus sont celles dans lesquelles le signal concerné est soit ignoré, soit traité par la fonction qui reçoit le signal ; au contraire, dès lors que le signal est reçu par une fonction dans laquelle l’action par défaut (c‑à‑d. l’arrêt du processus) reste en vigueur, le déclenchement des traps sur EXIT n’est pas garanti.

|  |  |  |
| --- | --- | --- |
| TRAP-05  AT&T ksh | Obligatoire | Les scripts qui utilisent des traps sur EXIT dans des fonctions ne doivent utiliser aucune fonction définie avec la syntaxe Bourne Shell / POSIX. |

Rappel : les traps sur EXIT des fonctions (par opposition au trap sur EXIT du script) ne sont supportés que par AT&T ksh, mais uniquement dans les fonctions non-POSIX, c’est-à-dire définies par le mot clef *function*. En ce qui concerne les fonctions définies dans la syntaxe Bourne shell / POSIX, a priori si un trap sur EXIT est défini à l’intérieur d’une telle fonction, cette définition concerne le script principal et non la fonction…

En réalité, l’implémentation d’AT&T ksh dans Solaris 10 fonctionne de la façon suivante : le comportement des traps sur EXIT définis à l’intérieur des fonctions ne dépend pas de la syntaxe utilisée pour déclarer chaque fonction, mais uniquement de la syntaxe utilisée par la *dernière* *fonction définie dans le script* ! Si cette fonction est définie dans la syntaxe non-POSIX (mot-clef *function*), tous les traps sur EXIT définis à l’intérieur des fonctions, quelle que soit la syntaxe utilisée pour définir ces fonctions, suivent les règles applicables aux fonctions non-POSIX. Inversement, si la dernière fonction du script est définie en utilisant la syntaxe Bourne shell / POSIX, les traps sur EXIT se comportent exactement comme si toutes les fonctions du script étaient définies dans cette syntaxe !

Cela signifie que si l’on souhaite utiliser des traps sur EXIT déclenchés au retour des fonctions il est très risqué d’utiliser des fonctions définies dans la syntaxe Bourne shell / POSIX en plus des fonctions non-POSIX, sachant que l’ordre dans lequel les fonctions sont définies peut avoir un effet dévastateur : il suffit que la dernière fonction du script soit définie dans la syntaxe Bourne shell / POSIX pour qu’aucun trap sur EXIT ne soit déclenché au retour des fonctions.

Cas de test :

#! /bin/ksh

set -o nounset

function foo {

typeset funnam="foo"

trap "echo \"EXIT from $funnam, back into \$funnam\"" EXIT

echo "$funnam"

bar

}

function bar {

typeset funnam="bar"

trap "echo \"EXIT from $funnam, back into \$funnam\"" EXIT

echo "$funnam"

}

mu() {

echo "I'm not even called, I'm just here to ruin traps on EXIT inside functions."

}

trap "echo \"EXIT from main script\"" EXIT

funnam="MAIN SCRIPT"

foo

echo "All done."

Remarque : dans l’exemple ci-dessus la fonction mu est définie dans la syntaxe Bourne shell / POSIX ; on teste d’abord le script avec cette fonction définie dans la syntaxe non-POSIX, puis dans cette syntaxe.

Résultats :

Avec la fonction mu définie dans la syntaxe non-POSIX :

foo

bar

EXIT from bar, back into foo

EXIT from foo, back into MAIN SCRIPT

All done.

EXIT from main script

Avec la fonction mu définie dans la syntaxe Bourne shell / POSIX :

foo

bar

All done.

EXIT from bar, back into MAIN SCRIPT

Il est clair qu’aucun trap sur EXIT n’a été exécuté au retour des fonctions ; le trap sur EXIT exécuté lors de l’exit du script est celui défini par la fonction bar.

# Bonnes pratiques

## Lancer moins de sous-processus

|  |  |  |
| --- | --- | --- |
| BPR-01 | Conseillé | Lorsqu’il est possible d’utiliser soit des fonctionnalités natives du shell soit des commandes externes, on privilégiera l’emploi des fonctionnalités natives si l’efficacité est recherchée. |

Les commandes externes lancées par le shell sont exécutées dans des processus fils, or la création d’un nouveau processus est une opération coûteuse, non seulement pour le processus parent mais pour le système Unix tout entier. C’est pourquoi il peut y avoir un ordre de grandeur entre le temps d’exécution d’une fonctionnalité native du shell et l’exécution d’une commande externe équivalente : ce n’est pas que la commande externe soit plus lente, c’est simplement que le coût de création d’un nouveau processus est tel que le recours aux mécanismes natifs du shell peut être 100 ou 200 × plus rapide, si la commande à exécuter est très brève. Lorsque la fonctionnalité concernée est fréquemment utilisée, cette différence peut avoir un impact considérable.

Exemple : comparaison entre grep et les fonctionnalités intégrées de *pattern-matching*

On compare la fonction check\_isnum de la page 37, qui utilise exclusivement les fonctionnalités intégrées de pattern-matching, et l’implémentation ci-dessous, qui utilise la commande externe grep :

function check\_isnum {

# arg #1 = value to be tested

typeset arg="$1"

echo "$arg" | $GREP -qE '^[[:space:]]\*[-+]?[0-9]+[[:space:]]\*$'

}

La version ci-dessus est considérablement plus lente, en raison de l’appel à une commande externe (grep).

Test 1 : 200000 appels à la fonction check\_isnum dans la version de la page 37.

% ptime ./bench\_fun\_1.ksh

real 2.213

user 2.104

sys 0.039

Test 2 : 10000 appels seulement à la fonction check\_isnum dans la version présentée ci-dessus.

% ptime ./bench\_fun\_2.ksh

real 31.935

user 4.327

sys 23.177

Si on prend en compte uniquement le temps écoulé, on constate que la version de la page 37 est environ 280 × plus rapide que la version ci-dessus. Si on compare le temps « système », l’écart est encore plus significatif (> 10000 ×).

Le tableau ci-après présente quelques exemples de besoins couramment rencontrés couverts à la fois par des commandes externes et par des fonctionnalités natives de ksh ; celles-ci doivent être employées chaque fois qu’il est possible de le faire sans que cela nuise à la simplicité ou à la clarté des scripts [[85]](#footnote-85).

|  |  |  |
| --- | --- | --- |
| Besoin fonctionnel | Commandes externes | Fonctionnalités natives de ksh |
| Reconnaissance d’un motif | echo | grep | [[ "$*param*" = *pattern* ]] |
| Extraction d’un motif | echo | sed | Expressions :  ${*param*%*pattern*}, ${*param*%%*pattern*}, ${*param*#*pattern*}, ${*param*##*pattern*},  où *pattern* est un motif obéissant à la syntaxe des motifs de recherche des noms de fichiers [[86]](#footnote-86). |
| Décomposition d’un chemin en répertoire + nom de base du fichier | basename  dirname |
| Décomposition d’une chaîne en champs suivant un séparateur | echo | cut  echo | awk |
| echo | read *param1* *param2* … [[87]](#footnote-87) [[88]](#footnote-88) |
| set -A *tab\_champ* ${*chaîne*} (sans guillemets doubles !) avec IFS redéfini temporairement à la valeur du caractère de séparation |
| Calculs arithmétiques | expr | Commande let, ou (de préférence) les expressions (( … )) qui sont équivalentes |
| Conversion de casse  (majuscules ↔ minuscules) | echo | tr | typeset -u *param*="*valeur* " # majuscules  typeset -l *param*="*valeur* " # minuscules |
| Écriture formatées | printf | typeset -L*n* *param*="*valeur*" # left justify  typeset -R*n* *param*="*valeur*" # right justify  typeset -Z*n* *param*="*valeur*" # left pad with zeroes |

Attention : l’exemple ci-dessus ne doit pas inciter à remplacer sans discernement tous les appels de commandes externes : le recours aux mécanismes internes du shell est surtout avantageux dans le cas de commandes dont la durée d’exécution est très brève, c’est-à-dire lorsque le coût de lancement d’un sous-processus (qui est typiquement en millisecondes) est nettement supérieur au temps de traitement, aussi bien par une commande externe que par le shell. Lorsque ce n’est pas le cas, l’emploi des commandes externes s’impose généralement en raison de leurs performances très supérieures à une implémentation en shell. À titre d’exemple, considérons une fonction ksh équivalente à la commande wc -l : tant que les fichiers à traiter sont très brefs, l’appel à cette fonction est plus performant que l’appel à la commande externe wc ‑l, en raison du coût de lancement du processus dans lequel celle-ci s’exécute. Cependant la boucle de lecture en ksh demeure bien plus lente que celle de wc -l, si bien que les courbes se croisent assez vite : le recours à une telle fonction en lieu et place de wc -l n’a donc d’intérêt que pour de très petits fichiers. [[89]](#footnote-89)



## Externaliser les paramètres de configuration

|  |  |  |
| --- | --- | --- |
| BPR-02 | Recommandé | Afin de promouvoir la réutilisabilité, les paramètres de configuration des scripts ksh sont définis dans des fichiers externes. |

À l’évidence, coder les paramètres de configuration directement « en dur » à l’intérieur des scripts ksh est une très mauvaise pratique, qui entraîne 2 conséquences majeures :

1. Les scripts ainsi « configurés » ne sont pas utilisables dans un autre contexte (si ce n’est en clonant les scripts originaux par « copier-coller », une autre pratique à proscrire).
2. Tout changement dans la configuration nécessite de modifier les scripts concernés, ce qui comporte des risques d’erreur (sans parler des questions concernant le *version-tracking* que ce type d’intervention soulève).

A l’inverse, si le paramétrage des scripts est contenu dans des fichiers séparés :

1. Les scripts peuvent être utilisés dans des contextes différents : il suffit de créer de nouveaux jeux de fichiers de configuration.
2. Les modifications de paramètres de configuration ne modifient pas les scripts concernés et n’induisent aucun risque de changement accidentel de ces scripts (la question du *version-tracking* est ainsi limitée aux seuls fichiers de paramétrage).

Bien évidemment, cette approche nécessite un minimum d’effort de la part des développeurs afin de :

* Garantir que les scripts fonctionnent correctement dans l’ensemble des configurations valides.
* Détecter les configurations incorrectes ou non supportées et fournir en réponse des messages d’erreurs facilement exploitables pour y remédier.

Les fichiers de configuration peuvent être mis en œuvre par différents procédés, tels que :

* Les fichiers inclus, définissant les paramètres nécessaires directement en langage ksh
* Les fichiers plats en format CSV (*comma-separated values*)
* Les fichiers de configuration dans un mini-langage dédié, à la manière des fichiers .ini de DOS/Windows, voire dans d’autres syntaxes ou langages (comme XML par exemple).

### Fichiers inclus en syntaxe ksh

Le fichier de configuration est un simple script ksh inclus dans le script principal par la commande .

. /chemin\_vers/fichier\_de\_configuration

Il contient les déclarations des paramètres nécessaires au script principal :

# Chemin du repertoire de traces  
config\_log\_dir=/chemin\_vers/repertoire\_traces

# Nombre maximum d'erreurs tolerees  
config\_max\_errors=100

Remarque : pour faciliter la configuration, il est très important de documenter chacun des paramètres de façon précise, sous forme de commentaires ksh.

Assez fréquemment, le fichier de configuration doit contenir plusieurs jeux de paramètres, correspondant à différentes valeurs d’arguments d’appel du script principal. Exemple : le fichier ci-après contient 2 jeux de paramètres, correspondant aux valeurs *xxx* et *yyy* d’un paramètre d’appel du script.

config\_xxx\_log\_dir=/chemin\_vers/repertoire\_traces\_xxx  
config\_xxx\_max\_errors=100

config\_yyy\_log\_dir=/chemin\_vers/repertoire\_traces\_yyy  
config\_yyy\_max\_errors=50

Dans le script principal, les valeurs utilisées sont déterminées à l’aide de la commande eval :

nom\_config="$arg" # Nom de la configuration à lire

eval "log\_dir=\${config\_${nom\_config}\_log\_dir:-}" # Repertoire des traces

eval "max\_errors=\${config\_${nom\_config}\_max\_errors:-}" # Nombre d'erreurs autorisees

(Notez le caractère \ qui protège le 1er symbole $ afin que celui-ci demeure présent dans la commande soumise à eval.)

Cette méthode d’externalisation des paramètres de configuration est d’une grande facilité de mise en œuvre ; son principal inconvénient réside dans sa relative fragilité : l’utilisateur doit obligatoirement respecter la syntaxe de ksh, sous peine d’erreurs non contrôlables par le script principal. En particulier, une erreur lors d’un appel à la commande eval peut entraîner un abandon immédiat du script – cf. section 3.7.2 page 27 – accompagné d’un simple message d’erreur sur la sortie stderr – c’est pourquoi il est recommandé que cette sortie soit enregistrée ou a minima visible. Dans l’exemple ci-dessus il est indispensable de vérifier a priori que le paramètre *nom\_config* est bien composé exclusivement de caractères autorisés dans un nom de variable ksh.

Pour finir, inutile de préciser que par définition cette méthode permet l’injection de scripts à quiconque a accès en écriture aux fichiers de configuration.

### Fichiers plats au format CSV

Avec ce procédé, le fichier de configuration contient une ligne pour chaque élément de configuration, constituée d’une suite de valeurs séparées par un caractère de séparation, généralement une virgule, un point-virgule ou une tabulation.

# fichier\_configuration.cfg

# Separateur : tabulation

# Nom\_config Nb\_max\_err Chemin\_repertoire\_log  
XXX 100 /chemin\_vers/repertoire\_traces\_xxx  
YYY 50 /chemin\_vers/repertoire\_traces\_yyy

(Note : la définition précédente n’inclut ni les lignes de commentaires ni les lignes vides, mais leur implémentation ne présente aucune difficulté, c’est pourquoi nous les avons représentées ci-dessus.)

La lecture du fichier CSV est faite directement par le script ksh au moyen de commandes telles que grep, cut, awk, ou encore en utilisant les fonctionnalités natives de reconnaissance de motifs de ksh ; cette dernière solution est à privilégier si le script est lancé fréquemment et que les performances comptent, sachant que le lancement de multiples commandes externes effectuant un traitement très bref est nettement moins performant que l’utilisation des mécanismes natifs de ksh – cf. section 4.1 page 51.

L’avantage de ce type de fichiers de configuration est sa simplicité d’implémentation. Son principal inconvénient tient dans la rigidité du format : une erreur aussi facile à commettre que d’utiliser un espace à la place de la tabulation attendue risque de provoquer des résultats imprévus ; c’est pourquoi il est indispensable que le script ksh effectue un contrôle de validité sur les paramètres obtenus. Du reste cette précaution est recommandée quelle que soit la méthode de gestion des fichiers de configuration : c’est une condition *sine-qua-non* pour la robustesse du mécanisme de configuration.

### Fichiers de configuration au format .ini de DOS/Windows

Ce type de fichiers obéit à une syntaxe à la fois plus souple et plus riche que les 2 solutions précédentes. Les fichiers sont composés d’une suite de paramètres regroupés en sections. Les commentaires et les espaces (facultatifs) sont supportés, ainsi que les lignes blanches ou vides. Exemple :

# fichier\_configuration.ini

[Main] # Section principale

# Liste des configurations definies dans ce fichier  
config = xxx  
config = yyy

[Config.xxx] # Section pour la configuration xxx

log\_dir = /chemin\_vers/repertoire\_traces\_xxx  
nb\_max\_erreurs = 100

[Config.yyy] # Section pour la configuration yyy

log\_dir = /chemin\_vers/repertoire\_traces\_yyy  
nb\_max\_erreurs = 50

# -- fin --

Bien que ce format soit plus riche que les précédents, il est relativement aisé d’écrire une fonction ksh qui l’implémente ; une telle fonction prend typiquement 2 arguments : le nom de la section et le nom de l’option à rechercher ; ou 3 arguments si on passe aussi le chemin du fichier de configuration ; en retour, elle affecte un paramètre global contenant la valeur lue. Bien évidemment, elle renvoie une valeur de retour correspondant au succès ou à l’échec de l’interrogation du fichier de configuration.

Cependant les performances d’une implémentation réalisée à 100% en ksh risquent d’être assez médiocres, en raison du nombre très élevé de commandes externes (grep) nécessaires au traitement. Ici encore, si l’on doit lire un grand nombre de paramètres de configuration dans un script ksh lancé fréquemment, il faut impérativement retenir une implémentation plus performante. Une bonne solution consiste à implémenter l’interrogation du fichier de configuration dans un programme C, qui génère sur sa sortie standard du code ksh effectuant une ou plusieurs affectations de paramètres. Si ce programme se termine avec succès, la fonction ksh appelante (décrite au paragraphe précédent) évalue le code ksh généré à l’aide de la commande built-in eval ; le programme C étant écrit de manière à générer systématiquement du code ksh correct, il n’y a pas de risque d’erreur d’évaluation.

Cette technique est sophistiquée (il faut coder un *parser* de fichiers .ini en C à l’aide de Lex et Yacc, ou ANTLR, ou encore en utilisant une descente récursive ad-hoc), si bien qu’elle n’est envisageable que dans le cadre d’un framework de fonctions réutilisables, mais elle est robuste et performante.

Dans un même ordre de sophistication, on peut aussi envisager des fichiers de configuration au format XML, en appliquant le même mécanisme – la commande built-in eval – pour l’interface entre le programme externe qui lit le fichier de configuration et le script ksh appelant. Cependant le format .ini est nettement plus facile à lire pour un être humain qu’un fichier au format XML, c’est pourquoi cette dernière solution ne sera appliquée que pour répondre à des besoins spécifiques, contrairement aux autres méthodes décrites ci-dessus qui répondent à la plupart des besoins courants en matière de fichiers de configuration.

## Variables PATH et LD\_LIBRARY\_PATH

|  |  |  |
| --- | --- | --- |
| BPR-03 | Recommandé | Les modifications *system-wide* des variables PATH et LD\_LIBRARY\_PATH donnent lieu à une vérification d’impact et, le cas échéant, à des tests applicatifs approfondis. |

Les modifications du PATH susceptibles d’entraîner des effets de bords sont l’ajout de répertoires en tête du PATH et la modification de l’ordre des répertoires.

Si vous respectez la règle page 32 relative aux chemins des commandes (i.e. vous n’utilisez que des chemins absolus), vos scripts ne sont pas sensibles à ces modifications. En revanche, si d’autres scripts utilisés dans l’application ne remplissent pas ce critère, alors en cas de modification du PATH (comme décrit ci-dessus) il est possible que certaines commandes appelées par ces scripts soient remplacées par des commandes de même nom mais placées dans d’autres répertoires. Or ces commandes alternatives ne sont pas nécessairement équivalentes à 100% à celles utilisées jusque là, ce qui peut entraîner des d’incidents (en cas d’option non supportée) ou des bugs subtils (en cas de différence de comportement). Les scripts concernés ont-ils été testés auparavant dans cette configuration du PATH ? C’est peu probable [[90]](#footnote-90).

Pour cette raison, il faut considérer que l’ajout de répertoires en tête du PATH, ou les modifications de l’ordre des répertoires, sont des changements potentiellement risqués, susceptibles d’entraîner des régressions imprévisibles des scripts shells. De tels changements peuvent donc nécessiter une vérification d’impact, et le cas échéant des tests applicatifs avec la couverture la plus large possible.

Le cas de la variable LD\_LIBRARY\_PATH est analogue… en pire ! En effet, s’il est facile de se protéger des effets de bord de modifications du PATH en utilisant des chemins complets pour désigner les commandes (ce qui revient à ne pas utiliser la variable PATH), il n’y a pas vraiment d’équivalent pour la recherche des bibliothèques d’objets partagées : si la variable LD\_LIBRARY\_PATH est définie, le *runtime linker* examine systématiquement les répertoires listés dans cette variable en premier ; les répertoires éventuellement mentionnés dans le *runpath* des binaires, ainsi que les répertoires prédéfinis du système, sont toujours examinés dans un 2ème temps. Il n’y a pas de solution complètement satisfaisante à ce problème [[91]](#footnote-91) [[92]](#footnote-92).

|  |  |  |
| --- | --- | --- |
| BPR-04 | Conseillé | Les variables PATH et LD\_LIBRARY\_PATH sont aussi réduites que possible. |

Dans le cas de la variable PATH, c’est avant tout une question de simplicité : si jamais vous êtes confronté à un problème lié à l’ordre des répertoires dans le PATH, ce problème sera plus facile à résoudre si le PATH en question ne contient pas trop de répertoires. Par ailleurs la longueur du PATH peut influer légèrement sur les performances (cf. encadré page suivante) mais uniquement dans le cas d’appels de commandes effectués sans préciser le chemin (pratique à proscrire, cf. règle page 32).

En ce qui concerne LD\_LIBRARY\_PATH, c’est aussi une question de simplicité – cette variable ne devrait être définie qu’au moment d’appeler des commandes pour lesquelles il n’existe pas d’autre moyen de spécifier le chemin des bibliothèques partagées – mais pas seulement : d’une part, limiter l’utilisation de cette variable réduit les risques de conflits, d’autre part, l’effet du nombre de répertoires dans LD\_LIBRARY\_PATH sur les performances est plus sensible que dans le cas du PATH (cf. encadrés page suivante).

|  |  |  |
| --- | --- | --- |
| BPR-05 | Conseillé | La variable FPATH n’est pas définie. |

La variable FPATH définit un chemin de recherche pour les fonctions. Les répertoires listés dans FPATH sont interrogés après les répertoires du PATH lors de la résolution des noms de commandes : si une commande nommée *nom\_cmde* n’est pas trouvée dans le PATH, et qu’un des répertoires de FPATH contient un fichier dont le nom est *nom\_cmde*, ce fichier est chargé par le shell courant puis le shell tente d’appeler la fonction *nom\_cmde*, qui en principe est définie dans ce fichier. Il est possible d’accélérer cette recherche en pré-déclarant la fonction *nom\_cmde* à l’aide de la commande :

typeset -fu *nom\_cmde* # déclare *nom\_cmde* comme une fonction non définie

Dans ce cas, la recherche s’effectue uniquement dans les répertoires de FPATH. Les fonctions définies en utilisant ce mécanisme sont parfois appelées fonctions *auto-loaded* car elles ne sont chargées que lorsqu’elles sont appelées pour la 1ère fois.

Nous déconseillons d’utiliser FPATH, pour les raisons ci-après :

1. FPATH, étant analogue à PATH, présente les mêmes inconvénients : le fonctionnement des scripts peut être directement modifié par un simple changement de FPATH, qui est un paramètre externe (à moins de forcer sa définition à l’intérieur des scripts). Cela peut poser le problème de la qualification des scripts après modification de FPATH.
2. En ce qui concerne les fonctions du framework ksh applicatif (cf. règle page 41), il nous paraît plus simple de les regrouper par thèmes dans quelques fichiers, inclus explicitement par les scripts clients.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Le nombre de répertoires dans la variable PATH influe-t-il sur les performances ?  Réponse : oui… mais de manière marginale : Il faut que la variable PATH contienne beaucoup de répertoires pour que son influence soit notable. En effet, l’interrogation des répertoires listés dans le PATH est très rapide, et cela d’autant plus que les répertoires concernés sont fréquemment consultés [[93]](#footnote-93). Le tableau ci-dessous résume une expérience consistant à lancer répétitivement un programme binaire placé dans le dernier répertoire du PATH. Le programme invoqué est une copie de /bin/true, donc son temps d’exécution est négligeable : la durée mesurée est donc imputable uniquement au lancement du programme binaire, c’est-à-dire à la recherche de la commande dans le PATH, plus l’*overhead* inhérent à toute création d’un processus par le shell (fork + exec + démarrage du processus).   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Nombre de répertoires dans la variable PATH | | 1 | 50 | 100 | 200 | | Durée  d’exécution (s) | real | 2.702 | 2.967 (+10%) | 3.118 (+15%) | 3.411 (+26%) | | user | 0.304 | 0.420 (+38%) | 0.502 (+65%) | 0.662 (+117%) | | sys | 2.154 | 2.310 (+7%) | 2.375 (+10%) | 2.521 (+17%) |   Bilan : entre un PATH réduit à un seul répertoire, et un PATH contenant 50 répertoires (c’est beaucoup, sans que cela soit exceptionnel), la différence sur le temps de création des processus est d’environ 10% ; avec un PATH contenant 200 répertoires (ce qui est vraiment trop), cet écart atteint 25%. Cependant cette différence doit être relativisée pour 3 raisons :   1. Le cas testé est celui d’un processus ultra-court ; au contraire, si la durée d’exécution du processus concerné est longue, l’influence de sa durée de lancement sur la performance globale devient négligeable. 2. Le cas testé est un scénario défavorable où la commande à lancer est placée à la fin du PATH ; au contraire, si elle est trouvée au début du PATH, la longueur du PATH n’a pas d’importance. 3. Enfin, le mécanisme des *tracked alias* épargne à ksh la résolution du PATH pour certaines commandes fréquemment utilisées. |

|  |
| --- |
| Le nombre de répertoires dans LD\_LIBRARY\_PATH affecte-t-il les performances ?  Réponse : la discussion précédente concernant l’effet de la variable PATH sur les performances vaut également pour la variable LD\_LIBRARY\_PATH : le *runtime linker* recherche les bibliothèques dynamiques dans les répertoires listés dans LD\_LIBRARY\_PATH, qu’il interroge successivement. Cette recherche étant effectuée pour chaque bibliothèque à charger, le nombre de répertoires interrogés est finalement fonction du nombre de répertoires dans la variable LD\_LIBRARY\_PATH multiplié par le nombre de bibliothèques à charger. L’influence de LD\_LIBRARY\_PATH est d’autant plus grande que le *runtime linker* recherche les bibliothèques partagées systématiquement dans LD\_LIBRARY\_PATH avant de les rechercher dans le *runpath* des binaires ou dans les répertoires prédéfinis du système [[94]](#footnote-94). |

# Récapitulatif

Règles générales

| Catégorie | Importance de la règle | Résumé de la règle | Référence (n° de page) |
| --- | --- | --- | --- |
| Emploi du langage | Obligatoire | Les scripts ksh sont utilisés pour réaliser des fonctions *élémentaires* de manipulation de fichiers et de lancement de traitements, non pour implémenter des fonctions métier ou des enchaînements complexes. | page 4 |
| Nommage des scripts | Conseillé | Les noms des scripts sont en lettres minuscules uniquement ; le tiret de soulignement « \_ » peut être utilisé comme séparateur de mots. | page 4 |
| Nommage des scripts | Conseillé | Les scripts ksh portent l’extension .ksh | page 4 |
| Nommage des scripts | Conseillé | Les noms de scripts peuvent être préfixés par le trigramme du projet, éventuellement suivi de l’identification (trigramme) du module ou du lot. | page 4 |
| Permissions | Obligatoire | Le bit d’exécution nécessaire au user applicatif est activé sur les scripts ksh. | page 5 |
| Permissions | Conseillé | Le bit d’exécution n’est pas activé sur les fichiers *inclus* dans les scripts ksh via la commande . *fichier* | page 5 |
| Permissions | Obligatoire | L’utilisation des bits setuid et setgid est interdite. | page 5 |
| Organisation générale des scripts ksh | Obligatoire | La première ligne des scripts ksh exécutables est obligatoirement :  #! /bin/ksh | page 5 |
| Organisation générale des scripts ksh | Recommandé | Description de l’organisation générale des scripts ksh (cf. [énoncé](#ORG_02_Structure_gnle)) | page 6 |
| Règles de style | Recommandé | La longueur des lignes est limitée à 80 caractères (cf. [énoncé](#STY_01_Longueur_lignes)). | page 8 |
| Règles de style | Obligatoire | Largeur d’indentation : tabulation logique de 4 caractères ; indentation uniquement avec des blancs, ou avec des blancs et des tabulations sous condition que la largeur des tabulations physiques reste à 8 caractères. | page 9 |
| Règles de style | Obligatoire | Les paramètres positionnels $1, $2, etc., ne doivent être utilisés que pour copier leurs valeurs dans des paramètres nommés ; toute autre référence à ces paramètres est interdite. | page 10 |
| Règles de style | Obligatoire | Utilisation de la casse (cf. [énoncé](#STY_04_Casse_min_maj)) :   - paramètres locaux et fonctions : casse minuscule  - constantes et paramètres exportés : casse majuscule | page 10 |
| Règles de style | Recommandé | Les paramètres dont le nom contient un tiret de soulignement (\_) doivent être évalués en les inscrivant dans une paire d’accolades. | page 11 |
| Règles de style | Recommandé | La syntaxe recommandée pour les expressions arithmétiques est la forme entre (( et )). | page 11 |
| Règles de style | Recommandé | Ne pas utiliser les backquotes ! La syntaxe recommandée pour substituer le résultat d’une commande est la forme $(*commande*) | page 12 |

Règles de programmation

| Catégorie | Importance de la règle | Énoncé (résumé) de la règle | Référence (n° de page) |
| --- | --- | --- | --- |
| Généralités | Obligatoire | La duplication de code par « copier-coller » est interdite. | page 12 |
| Codes d’exit | Obligatoire | Règle de normalisation des codes d’exit (cf. [énoncé](#PRG_02_Codes_exit)). | page 13 |
| Codes d’exit | Obligatoire | Règle de normalisation des valeurs de retour des fonctions (cf. [énoncé](#PRG_03_Function_ret_codes)). | page 13 |
| Codes d’exit | Obligatoire | Les codes d’exit des scripts et les valeurs de retour des fonctions sont documentés dans leurs en-têtes respectifs. | page 13 |
| Codes d’exit | Conseillé | Les codes d’exit des scripts, ainsi que les valeurs de retour des fonctions, sont standardisés et définis dans des scripts de définitions communes. | page 14 |
| Utilisation des variables | Recommandé | Si possible, utiliser l’option nounset afin de traiter l’évaluation d’un paramètre non défini comme une erreur d’exécution. | page 14 |
| Utilisation des variables | Obligatoire | Emploi des guillemets doubles afin que les paramètres susceptibles de contenir des blancs ne soient pas évalués comme une suite d’arguments distincts ; idem afin que les paramètres susceptibles d’être nuls ou non définis soient traités comme des arguments nuls (cf. [énoncé](#VAR_02_Double_quotes)). | page 15 |
| Utilisation des variables | Recommandé | Idiomes recommandés pour tester si un paramètre est défini et non nul (cf. [énoncé](#VAR_03_Test_non_nul)). | page 16 |
| Utilisation des variables | Recommandé | Idiomes recommandés pour tester si un paramètre est nul ou non défini (cf. [énoncé](#VAR_04_Test_nul_ou_undef)). | page 17 |
| Utilisation des variables | Recommandé | Idiomes recommandés pour tester si 2 paramètres de type chaîne de caractères sont égaux (cf. [énoncé](#VAR_05_Test_égalité_str)). | page 17 |
| Utilisation des variables | Recommandé | Forme recommandée pour tester si une chaîne est conforme à un motif dans le langage des motifs de recherche des noms de fichiers (cf. [énoncé](#VAR_06_Test_pattern_match)). | page 17 |
| Utilisation des variables | Recommandé | Forme recommandée pour comparer 2 paramètres de type entier, ou pouvant être convertis en entiers (cf. [énoncé](#VAR_07_Test_egalité_integer)). | page 18 |
| Utilisation des variables | Conseillé | Les paramètres utilisés en tant qu’indices, ou dans des expressions arithmétiques, sont déclarés par typeset -i *param* ou integer *param*. | page 18 |
| Contrôle des arguments | Obligatoire | Les scripts vérifient que leurs arguments sont valides et conformes à la syntaxe d’appel préalablement à tout traitement (cf. [énoncé](#PRG_06_Ctrl_arguments)). | page 19 |
| Gestion des options | Conseillé | Privilégier l’ajout de nouvelles options dans la syntaxe d’appel pour enrichir les scripts existants de fonctionnalités supplémentaires (cf. [énoncé](#OPT_01_Favor_options)). | page 19 |
| Gestion des options | Recommandé | La commande built-in getopts est recommandée pour interpréter les options sur la ligne de commande. | page 20 |
| Gestion des options | Conseillé | Les scripts dont la syntaxe d’appel est non triviale acceptent l’option -h (cf. [énoncé](#OPT_03_Option_h_help)). | page 23 |
| Détection des erreurs | Obligatoire | Le code d’exit de chaque commande appelée est systématiquement contrôlé. Idem pour tous les programmes, fonctions et autres scripts appelés dans les scripts ksh (cf. [énoncé](#PRG_07_Check_exit_codes)). | page 24 |
| Détection des erreurs | Obligatoire | Les redirections d’I/O pouvant échouer, les codes d’exit des commandes redirigées sont systématiquement testés. | page 24 |
| Détection des erreurs | Recommandé | Les erreurs susceptibles de survenir dans les commandes lancées dans des pipelines doivent être prises en considération. | page 25 |
| Chemins d’accès | Recommandé | Les fichiers lus / écrits par les scripts shells sont désignés par des chemins complets ; l’utilisation des chemins relatifs est fortement déconseillée. | page 31 |
| Chemins d’accès | Obligatoire | Les commandes externes ou les scripts invoqués dans les scripts ksh sont toujours appelés en utilisant leur chemin complet. | page 32 |
| Chemins d’accès | Obligatoire | Le lancement de commandes ou de scripts via des chemins relatifs est interdit. | page 34 |
| Fichiers de traces | Recommandé | Règles d’emploi des fichiers de sortie stdout et stderr (cf. [énoncé](#TRC_01_Traces_stdout_stderr)). | page 34 |
| Fichiers de traces | Recommandé | Règles concernant la redirection des fichiers de sortie stdout et stderr (cf. [énoncé](#TRC_02_Redirect_stdout_stderr)). | page 34 |
| Fichiers de traces | Recommandé | Les traces d’exécution des scripts sont aussi verbeuses que nécessaire. En particulier, le début et la fin des étapes de traitement sont tracées, ainsi que toutes les erreurs rencontrées. | page 35 |
| Fichiers de traces | Recommandé | Toute erreur ou circonstance inattendue, telle que l’échec d’une commande, l’absence d’un fichier en entrée, etc., doit être détectée et tracée. | page 35 |
| Fichiers de traces | Recommandé | Les messages de début et de fin des étapes de traitement sont horodatés. | page 36 |
| Fichiers de traces | Recommandé | Règles relatives au découpage des fichiers de traces et à la création de nouveaux fichiers de traces lors des itérations successives d’un même job (cf. [énoncé](#TRC_06_Decoupage_traces)). | page 36 |
| Utilisation des fonctions | Recommandé | Découpez les traitements en utilisant des fonctions. | page 36 |
| Utilisation des fonctions | Obligatoire | Les fonctions sont déclarées en utilisant le mot-clef *function*. Les fonctions déclarées dans la syntaxe Bourne Shell / POSIX sont interdites. | page 38 |
| Utilisation des fonctions | Recommandé | Utilisez la commande typeset pour créer des variables locales aux fonctions. | page 39 |
| Utilisation des fonctions | Obligatoire | Emploi des paramètres positionnels ($1, $2, etc.) dans les fonctions : idem règle valable pour le script principal (cf. [énoncé](#FUN_04_Positional_params)). | page 40 |
| Utilisation des fonctions | Recommandé | Favorisez la réutilisation des fonctions en créant un *framework* de fonctions communes réutilisables. | page 41 |
| Utilisation des fonctions | Recommandé | Les fonctions communes réutilisables sont documentées méticuleusement. | page 42 |
| Utilisation des fonctions | Conseillé | Utilisez la commande built-in getopts pour gérer les options dans la syntaxe d’appel des fonctions. | page 43 |
| Utilisation des tableaux | Obligatoire | Les scripts qui utilisent des tableaux doivent tenir compte des limites sur la taille des tableaux en ksh. | page 43 |
| Utilisation des tableaux | Conseillé | Éviter les tableaux « creux ». | page 44 |
| Utilisation des traps | Recommandé | Les scripts qui utilisent des traps et doivent fonctionner aussi bien avec pdksh qu’avec AT&T ksh doivent tenir compte des différences significatives entre les deux implémentations. | page 46 |
| Utilisation des traps pdksh | Recommandé | Les scripts devant s’exécuter sous pdksh ne doivent pas définir de trap sur EXIT à l’intérieur des fonctions. | page 47 |
| Utilisation des traps pdksh | Obligatoire | Les scripts devant s’exécuter sous pdksh ne doivent pas dépendre du déclenchement des traps sur EXIT en cas d’erreur d’exécution. | page 48 |
| Utilisation des traps AT&T ksh | Obligatoire | Les scripts qui utilisent des traps sur EXIT dans des fonctions ne doivent pas dépendre du déclenchement de ces traps en cas de réception d’un signal. | page 48 |
| Utilisation des traps AT&T ksh | Obligatoire | Les scripts qui utilisent des traps sur EXIT dans des fonctions ne doivent utiliser aucune fonction définie avec la syntaxe Bourne Shell / POSIX. | page 49 |

Bonnes pratiques

| Catégorie | Importance de la règle | Énoncé (résumé) de la règle | Référence (n° de page) |
| --- | --- | --- | --- |
| Lancer moins de sous-processus | Conseillé | Préférer l’emploi des fonctionnalités natives du shell à l’utilisation des commandes externes si l’efficacité est recherchée (cf. [énoncé](#BRP_01_Minimiser_cmd_externes)). | page 51 |
| Externaliser les paramètres de config. | Recommandé | Afin de promouvoir la réutilisabilité, les paramètres de configuration des scripts ksh sont définis dans des fichiers externes. | page 53 |
| Variables PATH et LD\_LIBRARY\_PATH | Recommandé | Les modifications *system-wide* des variables PATH et LD\_LIBRARY\_PATH donnent lieu à une vérification d’impact et à des tests approfondis. | page 56 |
| Variables PATH et LD\_LIBRARY\_PATH | Conseillé | Les variables PATH et LD\_LIBRARY\_PATH sont aussi réduites que possible. | page 57 |
| Variables PATH et LD\_LIBRARY\_PATH | Conseillé | La variable FPATH n’est pas définie. | page 57 |

# Différences entre ksh et pdksh

Cette section traite des différences les plus flagrantes entre ksh et pdksh.

De multiples différences existent entre l’implémentation de référence du Korn Shell (celle d’AT&T) et pdksh, de même que des différences existent entre les implémentations de ksh par AT&T, Sun, HP, IBM [[95]](#footnote-95)… Cependant ces dernières portent généralement sur des points subtils, ce qui n’est pas le cas de celles qui sont mentionnées ici.

Une longue liste de différences entre l’implémentation d’AT&T et pdksh est également disponible sur internet : cf. [].

## Commandes built-in à droite d’un pipe

AT&T ksh : si la commande à droite d’un pipe est une built-in, elle est exécutée dans le processus courant.

pdksh : la commande à droite d’un pipe est exécutée dans un sous-processus, même si c’est une built-in.

Exemple :

echo "hi" | read line ; echo "[$line]"

- AT&T ksh : affiche [hi]

- pdksh : []

Il faut tenir compte de cette différence si la portabilité avec pdksh est recherchée :

AT&T ksh uniquement :

cat *fich* | while read line ; do

# traitement de la ligne lue : $line

...

done

AT&T ksh et pdksh :

while read line ; do

# traitement de la ligne lue : $line

...

done <*fich*

## Traitement des erreurs d’exécution

### Erreur d’exécution lors des appels de commandes built-in spéciales [[96]](#footnote-96)

AT&T ksh : les erreurs d’exécution lors des appels de commandes built-in spéciales provoquent un arrêt immédiat du script (avec un code d’exit non nul).

pdksh : sauf cas particuliers, les erreurs d’exécution lors des appels de commandes built-in spéciales provoquent seulement l’abandon de l’unité d’exécution courante (avec un code de retour non nul). Cas particulier : commande exec (cf. section 3.7.3 page 28).

### Échecs de redirection et appels de commandes built-in spéciales

AT&T ksh : en cas d’échec de redirection (c'est-à-dire si le fichier de redirection ne peut pas être *ouvert*) lors d’un appel de commande built-in spéciale, le script se termine avec un code d’exit non nul.

pdksh : en cas d’échec de redirection lors d’un appel de commande built-in spéciale, seule l’unité d’exécution courante (script principal ou fonction) est abandonnée, avec un code de retour non nul.

### Paramètre non défini et redirection des commandes built-in

Si le paramètre *param* n’est pas défini, la redirection vers le fichier désigné par $*param* échoue : on ne peut pas créer un fichier nommé "" (chaîne vide).

*commande* >"${*param*}" # Erreur si *param* n’est pas défini

Par ailleurs, si l’option nounset est activée, le fait de substituer un paramètre non défini est également une erreur en soi, si bien que la ligne ci-dessus contient finalement 2 erreurs.

AT&T ksh : si *commande* est une commande built-in, c’est l’échec de redirection qui est retenu : cette erreur provoque l’exit du script si *commande* est une built-in spéciale ; sinon, seule la ligne concernée échoue : l’unité d’exécution courante continue malgré la substitution du paramètre non défini.

pdksh : avec l’option nounset activée, le traitement est toujours celui de la tentative de substitution d’un paramètre non défini : cette erreur provoque systématiquement l’abandon de l’unité d’exécution courante.

### Traitement des erreurs par la commande built-in eval

AT&T ksh : une erreur d’exécution lors d’un appel à la commande built-in eval provoque l’exit du script (exemples : erreur de syntaxe dans la commande à évaluer ; paramètre non défini alors que l’option nounset est activée).

pdksh : les erreurs d’exécution lors des appels à la commande built-in eval provoquent seulement l’abandon de l’unité d’exécution courante.

### Échecs de redirection et appels de fonction [[97]](#footnote-97)

AT&T ksh : un échec de redirection lors d’un appel de fonction entraîne l’abandon de l’unité d’exécution courante, et non un simple échec de l’appel concerné.

pdksh : les appels de fonctions sont traités comme des appels de commandes ordinaires : un échec de redirection entraîne un simple échec de l’appel concerné.

## Gestion des traps

### Traps sur EXIT et fonctions

AT&T ksh : si un trap sur EXIT est défini à l’intérieur d’une fonction non-POSIX, c’est-à-dire définie à l’aide du mot-clef *function*, il est exécuté au retour de la fonction ; de plus les traps sur EXIT définis à l’intérieur des fonctions non-POSIX sont indépendants du trap sur EXIT défini au niveau du script principal le cas échéant. Au contraire, si un trap sur EXIT est défini à l’intérieur d’une fonction déclarée dans la syntaxe Bourne / POSIX, ce trap affecte en fait le script principal et non la fonction : il n’est exécuté que lorsque le script principal se termine [[98]](#footnote-98).

pdksh : les fonctions non-POSIX et les fonctions définies dans la syntaxe Bourne shell / POSIX ont le même comportement vis-à-vis des traps sur EXIT ; si un trap sur EXIT est défini à l’intérieur d’une fonction :

1. Il annule toute définition antérieure du trap sur EXIT du script principal
2. Il est exécuté uniquement quand le script principal se termine.

### Traps sur EXIT et paramètre $?

AT&T ksh : le trap sur EXIT est exécuté dans l’environnement « tel qu’il était immédiatement après la dernière commande exécutée *avant* que le trap sur EXIT ne soit déclenché » [[99]](#footnote-99) [[100]](#footnote-100) ; c’est une façon compliquée de dire que le trap sur EXIT ne voit pas la valeur du paramètre $? qui sera retournée à l’appelant : il voit ce paramètre tel qu’il était après la « dernière commande exécutée », c’est-à-dire celle qui *précède* la commande return ou exit, ou encore l’erreur provoquant la sortie de la fonction ou du script ; le code du trap sur EXIT ne peut donc pas se fonder sur la valeur de $? pour déterminer s’il est déclenché par une sortie normale (sans erreur) ou non.

pdksh : le trap sur EXIT voit le paramètre $? tel qu’il sera retourné à l’appelant.

Script de test (test\_trap\_exit.ksh) :

#! /bin/ksh

function do\_return { return $1 ; }

trap 'echo "In trap on EXIT, exit code: $?"' EXIT

do\_return 42

exit 13

Résultats :

AT&T ksh :

% ./test\_trap\_exit.ksh ; echo "Script exit code: $?"

In trap on EXIT, exit code: 42

Script exit code: 13

pdksh :

% ./test\_trap\_exit.ksh ; echo "Script exit code: $?"

In trap on EXIT, exit code: 13

Script exit code: 13

### Traps sur EXIT et erreurs d’exécution

AT&T ksh : les traps sur EXIT sont déclenchés si la terminaison fait suite à une erreur d’exécution.

pdksh (5.2.14) : les traps sur EXIT ne sont pas déclenchés lorsque la terminaison est consécutive à une erreur d’exécution.

Script de test (test\_trap\_exit\_2.ksh)

#! /bin/ksh

trap 'echo "In trap on EXIT, exit code: $?"' EXIT

eval "echo \"Hello World\"" >/no\_such\_file\_or\_directory

# redirection error: script will exit because eval is a special builtin

echo "Unreachable statement (CAN'T HAPPEN)"

exit 13

Résultats :

AT&T ksh :

% ./test\_trap\_exit\_2.ksh ; echo "Script exit code: $?"

./test\_trap\_exit\_2.ksh[5]: /no\_such\_file\_or\_directory: cannot create

In trap on EXIT, exit code: 0

Script exit code: 1

pdksh :

% ./test\_trap\_exit\_2.ksh ; echo "Script exit code: $?"

./test\_trap\_exit\_2.ksh[5]: cannot create /no\_such\_file\_or\_directory: Permission denied

Script exit code: 1

### Traps sur ERR et paramètre LINENO

AT&T ksh : quand le trap sur ERR est exécuté le paramètre LINENO est renseigné comme attendu : il contient le numéro de ligne [[101]](#footnote-101) de la commande ayant retourné un code d’exit ≠ 0.

pdksh (5.2.14) : quand le trap sur ERR est exécuté $LINENO vaut systématiquement 0 !

### Traps sur réception de signaux

Cf. tableau de synthèse à la règle page 46.

## Autres différences

Attention : cette section recense des différences entre pdksh et AT&T ksh (sous SunOS) déjà connues ou constatées dans le cadre de la rédaction de ce document. Cette liste est certainement incomplète. La leçon essentielle à retenir est que les implémentations de ksh peuvent différer sur un certain nombre de points, nécessitant une grande vigilance de la part du programmeur, ainsi que des tests unitaires poussés sur chacune des implémentations cibles.

### Paramètre ERRNO [[102]](#footnote-102)

AT&T ksh : ERRNO contient la valeur de errno positionnée par le dernier appel système en échec.

pdksh : le paramètre ERRNO est documenté, mais il n’est pas implémenté.

### Surcharge des built-ins par des fonctions

AT&T ksh : définir une fonction avec le même nom qu’une commande built-in n’est pas autorisé et provoque une erreur [[103]](#footnote-103) [[104]](#footnote-104).

pdksh : définir une fonction avec le même nom qu’une commande built-in est autorisé ; sauf dans le cas d’une commande built-in spéciale, la fonction est ensuite invoquée à la place de la commande built-in.

### Taille des entiers dans les expressions arithmétiques

AT&T ksh (SunOS 5.10) : les expressions arithmétiques sont en entiers signés sur 64 bits.

pdksh (i386) : les expressions arithmétiques sont en entiers signés sur 32 bits.

((x = 1 << 31))

[ $x -ge 0 ] && echo positif || echo negatif # AT&T ksh (SunOS 5.10) : positif  
 # pdksh (i386) : negatif

### Taille maximum des tableaux

AT&T ksh : les tableaux peuvent contenir jusqu’à 4096 éléments (sous Solaris [[105]](#footnote-105)).

pdksh : la taille limite des tableaux est de 1024 éléments.

### Paramètre OPTIND dans les fonctions

AT&T ksh : le paramètre OPTIND est spécial [[106]](#footnote-106) : il est sauvegardé avant chaque appel de fonction et restauré au retour des appels de fonction ; à l’intérieur des fonctions il n’est pas nécessaire de le déclarer avec typeset. Cela permet d’utiliser la commande getopts aussi bien dans le script principal que dans les fonctions, sans risque d’interférence.

pdksh : le paramètre OPTIND est spécial (comme dans AT&T ksh) mais uniquement vis-à-vis des fonctions non-POSIX, c’est-à-dire définies avec le mot-clef *function*. Les fonctions définies dans la syntaxe Bourne shell / POSIX partagent le paramètre OPTIND de l’appelant, c’est pourquoi il est dangereux d’utiliser getopts dans ce type de fonction [[107]](#footnote-107).

### Paramètre non défini et ${#param}, ${param%word}

AT&T ksh : l’expression ${*#param*} ainsi que les expressions ${*param*#*word*}, ${*param*##*word*}, ${*param*%*word*} et ${*param*%%*word*} ne provoquent pas d’erreur si le paramètre *param* n’est pas défini, même si l’option nounset est activée.

pdksh : si le paramètre *param* n’est pas défini alors que l’option nounset est activée, ces expressions provoquent systématiquement l’erreur « *param*: *parameter not set* » et entraînent l’abandon de l’unité d’exécution courante.

### Paramètre non défini en contexte d’évaluation arithmétique

AT&T ksh : si le paramètre *param* n’est pas défini, toute occurrence de *param* (sans le symbole $) dans un contexte d’évaluation arithmétique provoque une erreur « *bad number* » et entraîne l’abandon de l’unité d’exécution courante – ce comportement est indépendant de l’option nounset.

pdksh : si le paramètre *param* n’est pas défini, toute occurrence de *param* (sans le symbole $) dans un contexte d’évaluation arithmétique est évaluée à zéro, indépendamment de l’option nounset.

((x = 1 + param)) # AT&T ksh : param non défini => erreur x = 1 + param: bad number + exit fonction

# pdksh : param non défini => param est évalué à zéro (pas d’erreur)

### Erreurs dans les expressions arithmétiques

Les différences entre AT&T ksh et pdksh dans ce domaine sont détaillées à la section 3.7.4 page 29.

### pdksh uniquement : bug de l’instruction set -A dans une fonction

Cas de test :

#! /bin/ksh

set -o nounset

# Declare and initialize the tab array

set -A tab 1 2 3 4

# Prints the contents of the tab array

function showtab {

print -n "$1: ${tab[0]:-?}, ${tab[1]:-?}, ${tab[2]:-?}, ${tab[3]:-?}"

print " (${#tab[\*]} elements)"

}

# Creates a local array with name tab, inserts elements into it,

# then calls showtab to print its contents.

function testcase {

typeset tab # create a new instance of tab (dynamically scoped)

set -A tab # reset the local instance of the tab array

tab[0]=5 # pdksh bug: assignment is performed in the global instance!

tab[1]=6 # pdksh bug: assignment is performed in the global instance!

typeset tab[2]=7 # Now pdksh remembers about the local instance of tab!

tab[3]=8 # Now pdksh remembers about the local instance of tab!

showtab "2nd"

}

# Main program

showtab "1st"

testcase

showtab "3rd"

Résultats :

AT&T ksh (Solaris 10) :

1st: 1, 2, 3, 4 (4 elements)

2nd: 5, 6, 7, 8 (4 elements)

3rd: 1, 2, 3, 4 (4 elements)

pdksh 5.2.14 (Cygwin 1.7) :

1st: 1, 2, 3, 4 (4 elements)

2nd: ?, ?, 7, 8 (2 elements)

3rd: 5, 6, 3, 4 (4 elements)

Solutions de contournement :

1. Éviter l’instruction set -A *tableau* nue dans les fonctions. Cette instruction n’est pas indispensable dans la mesure où il n’est pas nécessaire de pré-déclarer les tableaux [[108]](#footnote-108).
2. Dans les fonctions, utiliser typeset pour affecter des valeurs aux éléments des tableaux :

typeset *tableau*[*indice*]=*valeur*

De cette manière, l’utilisation d’un tableau local (créé si nécessaire) est garantie.

### Commande (built-in) print –u

AT&T ksh : print -u 2 "*message*" écrit « *message* » sur le descripteur 2 (stderr).

pdksh : print -u 2 "*message*" écrit « 2 *message* » sur le descripteur 1 (stdout). Il faut écrire : print -u2 (sans espace entre -u et 2) pour obtenir le résultat souhaité.

### pdksh uniquement : bug de la substitution ${1#$2}

Script de test (bug\_substitution.ksh) [[109]](#footnote-109) :

#! /bin/ksh

set -o nounset

echo "\$1: \"$1\""

echo "\$2: \"$2\""

echo "\${1#\$2}: \"${1#$2}\""

foo="$1"

bar="$2"

echo "\${foo#\$bar}: \"${foo#$bar}\""

Résultats :

AT&T ksh :

% ./bug\_substitution.ksh 1122 11

$1: "1122"

$2: "11"

${1#$2}: "22"

${foo#$bar}: "22"

pdksh (5.2.14) :

% ./bug\_substitution.ksh 1122 11

$1: "1122"

$2: "11"

${1#$2}: ""

${foo#$bar}: "22"

### Paramètre LINENO

AT&T ksh : à l’extérieur des fonctions le paramètre LINENO indique le numéro de la ligne courante relativement au début du script ; à l’intérieur des fonctions le paramètre LINENO indique le numéro de la ligne courante relativement au début de la fonction [[110]](#footnote-110).

Exception : dans le cas d’un fichier inclus (par la commande .) la valeur de LINENO est constante, y compris à l’intérieur des fonctions : elle est égale au numéro de la ligne qui effectue l’inclusion dans le script principal.

pdksh : le paramètre LINENO indique le numéro de la ligne courante relativement au début du script ; il n’y a pas de distinction entre les lignes à l’extérieur ou à l’intérieur des fonctions.

Cas des scripts inclus : LINENO est relatif au fichier courant, autrement dit la 1ère ligne de chaque fichier inclus est toujours numérotée 1.

### Paramètre $0

AT&T ksh : le paramètre $0 est invariant et égal au chemin du script ksh, tel qu’invoqué lors de la création du processus.

pdksh : le paramètre $0 est invariant et égal au chemin du script ksh, tel qu’invoqué lors de la création du processus, sauf à l’intérieur des fonctions non-POSIX, où $0 est égal à l’identifiant de la fonction.

### Liste des différences

Michael Rendell, le mainteneur de pdskh, a établi une liste détaillée de différences entre AT&T ksh et pdksh ; cf. [].

# Références

## Généalogie de AT&T ksh et de pdksh

Le Korn shell a été développé par David Korn au sein de Bell Labs à partir de 1983 et intégré dans les principaux systèmes Unix propriétaires tels que Sun Solaris, IBM AIX, HP-UX, dérivés sous licence d’Unix System V d’AT&T [[111]](#footnote-111). Les implémentations de ksh sur ces systèmes sont issues de l’implémentation d’AT&T dans Unix System V R4 (1988), plus ou moins débogguée et améliorée sur chaque système [[112]](#footnote-112).

Une version plus riche de ksh, appelée ksh93, a été développée ultérieurement par David Korn, toutefois la diffusion de cette version a été relativement lente jusqu’à la fin des années 2000 ; à ce jour cette version demeure encore peu utilisée, mais son adoption devrait progresser rapidement ces prochaines années.

Pdksh, ou Public Domain Korn Shell, est une ré-implémentation complète de ksh (1988), indépendante de l’implémentation d’AT&T, visant à en produire un « clone ». Le site officiel de pdksh résume ainsi son niveau de compatibilité : « *at the moment, it* [pdksh] *has most of the ksh88 features, not much of the ksh93 features, and a number of its own features.* »

## ksh

|  |  |  |
| --- | --- | --- |
| Référence | Document | Remarques |
| Deffaix 2008 | Programmation shell sous Unix/Linux sh (Bourne), ksh, bash, 2ème édition Christine Deffaix Rémy. Éditions ENI, mai 2008 | Guide d’initiation et de programmation en ksh et bash, complet et précis. |
| Korn1995 | The New KornShell Command and Programming Language Morris I. Bolsky et David G. Korn, Prentice Hall, 1995. | Présentation et manuel de référence de ksh (ksh88 et ksh93), par son inventeur. |
| ksh(1) | manpages section 1: User Commands  PartNo: 816–5165–16  September 2010 | Page de manuel de ksh dans la documentation de Solaris 10 (Reference Manual Collection) |
| pdksh(1) | man ksh | Page de manuel de pdksh (sous Linux / Cygwin) |
| pdksh\_Notes | Notes d’implémentation de pdksh  http://web.cs.mun.ca/~michael/pdksh/NOTES | Contient une liste détaillée des différences entre AT&T ksh et pdksh |

## ksh93

**Attention : ksh93 est la version de Korn Shell installée par défaut dans Solaris 11 [[113]](#footnote-113) et dans Red Hat Enterprise Linux 5 et 6 [[114]](#footnote-114).**

|  |  |  |
| --- | --- | --- |
| Référence | Document | Remarques |
| Korn1995 | The New KornShell Command and Programming Language Morris I. Bolsky et David G. Korn, Prentice Hall, 1995. | Présentation et manuel de référence de ksh (ksh88 et ksh93), par son inventeur [[115]](#footnote-115). |
| – | Enhanced Korn shell (ksh93)  http://publib.boulder.ibm.com/infocenter/aix/v6r1/index.jsp?topic=/com.ibm.aix.baseadmn/doc/baseadmndita/korn\_shell\_enhanced.htm | Synthèse des fonctionnalités supplémentaires de ksh93 par rapport à ksh88 – documentation d’AIX 6.1. |
| – | http://mail.opensolaris.org/pipermail/ksh93-integration-discuss/2007-September/002932.html | Principales incompatibilités entre ksh88 et ksh93 – attention : les différences liées aux bugs des implémentations ne sont pas mentionnées [[116]](#footnote-116). |
| ksh93(1) | manpages section 1: User Commands  PartNo: 821–1461–10  November 2011 | Page de manuel de ksh93 dans la documentation de Solaris 11. |
| – | http://www2.research.att.com/~gsf/download/gen/ast-ksh.html#ast-ksh%20package | Sources de ksh93 sur le site d’AT&T Research. Cette page contient le changelog détaillé de ksh93. |

# Historique des modifications

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Description | Auteur(s) |
| 1.0-d1 | Mars 2012 | Draft initial | R. Vassallo |
| 1.0-d2 | Mars 2012 | Corrections mineures | R. Vassallo |
| 1.0-d3 | Avril 2012 | Ajout de la règle  ; additions à la section  ; révision de la section  ; corrections diverses | R. Vassallo |
| 1.0-d4 | Mai 2012 | Précision concernant les indices de tableaux ; ajouts des tests en [[ … ]] (règles et ) ; compléments concernant le test de pattern-matching (règle ) | R. Vassallo |
| 1.0-d5 | Février 2013 | Corrections mineures | R. Vassallo |
| 1.1-d1 | Mai 2015 | Modification mineures ; ajout règles et | R. Vassallo |

1. Cet objectif conduit à utiliser des fonctionnalités spécifiques de ksh, incompatibles avec d’autres interpréteurs shell, tels que le Bourne Shell ou bash. Cette incompatibilité ne semble pas un inconvénient majeur au regard des bénéfices apportés, du moins tant que l’on accepte que les scripts concernés soient conçus spécialement pour ksh. L’objectif contraire, qui serait de rechercher la portabilité vers le plus grand nombre d’interpréteurs shell, serait non seulement difficile à atteindre dans la pratique, mais elle conduirait à n’utiliser que le plus petit dénominateur commun entre ces interpréteurs, ce qui serait très limitant. Étant donné que ksh est disponible sur tous les grands Unix commerciaux (IBM AIX, HP-UX, Solaris), nous pensons qu’il est préférable d’utiliser les fonctionnalités disponibles, plutôt que de rechercher à tous prix une portabilité universelle hypothétique. Autrement dit, la présente norme s’applique aux scripts ksh conçus sans exigence de portabilité vers d’autres interpréteurs shell (Bourne, bash, etc.) ; cela concerne en fait la majorité des scripts ksh utilisés dans les applications. [↑](#footnote-ref-1)
2. Le lecteur intéressé est invité à se référer aux documents listés à la section page 71. Par ailleurs, les machines virtuelles pré-packagées par Oracle pour VirtualBox (http://www.oracle.com/technetwork/community/developer-vm/index.html) permettent d’installer rapidement un environnement de tests, aussi bien en OEL 5 qu’en Solaris 11. [↑](#footnote-ref-2)
3. La permission de lecture sur le script est bien évidemment nécessaire, sinon le lancement échoue avec le message d’erreur « cannot open ». [↑](#footnote-ref-3)
4. L’important est que le user qui doit exécuter le script dispose à la fois de la permission de lecture et de la permission d’exécution. En règle générale, le bit d’exécution n’est pas activé pour « other », cependant la sécurité de l’application ne devrait pas en dépendre, i.e. les permissions sur les ressources applicatives doivent être telles que rien de grave ne se produit si un script est exécuté accidentellement par un utilisateur non formellement autorisé. [↑](#footnote-ref-4)
5. L’idiome « #! pathname [arg1] » est connu sous le nom de *shebang syntax*. Il est possible de passer un argument (arg1) à l’interpréteur, mais pas davantage ; sous Solaris, les arguments suivants sont ignorés – cf. man exec(2). [↑](#footnote-ref-5)
6. Sous Solaris 10, si la ligne #! /bin/ksh est omise, le script sera exécuté dans un Korn shell si le processus parent est lui-même un Korn Shell, et dans un Bourne shell si le processus parent est un Bourne shell – bref, le système fait de son mieux pour trouver un interpréteur, cependant il vaut mieux ne pas compter sur ce type de mécanisme. En particulier, les ordonnanceurs (Autosys, Control‑M, etc.) lancent habituellement les commandes dans des Bourne shells. [↑](#footnote-ref-6)
7. Même si votre script ksh n’est pas conçu pour être lancé « à la main » vous avez quand même des utilisateurs : vos collègues qui, demain ou dans 10 ans, appelleront votre script dans leurs propres programmes ou dans des jobs ordonnancés, et qui ont besoin de savoir exactement ce que fait votre script et comment l’utiliser. Ce besoin croît rapidement avec la complexité du script. [↑](#footnote-ref-7)
8. Cf. règle page 14. [↑](#footnote-ref-8)
9. Bien sûr, la variable APP\_HOME ne peut pas être définie dans le fichier xyz\_profile.ksh : c’est l’une des rares variables de l’application qui doit être définie dans le fichier .profile du user applicatif, ou dans un fichier d’environnement inclus dans le .profile. Si cette variable n’est pas définie, l’inclusion . ${APP\_HOME}/shells/xyz\_profile.ksh échouera (a priori), car la commande . provoque un échec du script si le fichier à lire n’est pas trouvé ; cependant le message d’erreur obtenu « /shells/xyz\_profile.ksh: not found » est peu explicite, c’est pourquoi on pourra vérifier au préalable que la variable APP\_HOME est bien définie (sortir en erreur sinon). Ou plus simplement, on utilisera l’option set -u de ksh (cf. règle page 14) qui effectue cette vérification automatiquement : si la variable APP\_HOME n’est pas définie, cette option provoquera l’échec du script avec le message d’erreur « APP\_HOME: parameter not set » qui est sans ambiguïté. [↑](#footnote-ref-9)
10. Autrement dit, les éditeurs de texte *doivent* faire une distinction entre la largeur du retrait utilisée pour l’indentation, qui doit être de 4 caractères, et l’espacement des colonnes de tabulation ; ce dernier doit être maintenu à 8 caractères car c’est l’espacement communément utilisé, par défaut, par les terminaux et les éditeurs de texte. [↑](#footnote-ref-10)
11. Par souci de simplicité, les contrôles de validité des arguments (qui sont obligatoires – cf. règle page 19) ne sont pas détaillés dans cet exemple. [↑](#footnote-ref-11)
12. Une règle simple concernant l’emploi des majuscules et des minuscules permet aux noms de variables d’être porteurs d’information, on aurait tort de s’en priver ! Quant aux noms en casse mixte, à l’usage ils « marchent » moins bien en shell qu’en Java pour 2 raisons : d’une part ils sont plus difficiles à lire que les noms en casse uniforme, d’autre part ils induisent des risques d’erreur (de frappe, ou de confusion) d’autant plus grands que le programmeur shell n’est assisté ni par un compilateur ni par un AGL tel qu’Éclipse capable de détecter automatiquement ces erreurs. [↑](#footnote-ref-12)
13. En contexte d’évaluation arithmétique, le symbole $ est facultatif devant les paramètres déjà définis. Cela permet d’écrire des expressions telles que : let "y = a\*x + b" ou ((y = a\*x + b)) au lieu de : let "y = $a\*$x + $b" ou ((y = $a\*$x + $b)) ; ou encore, dans le cas des indices de tableaux : ${*nom\_tableau*[a\*x + b]} plutôt que ${*nom\_tableau*[$(($a\*$x + $b))]}. [↑](#footnote-ref-13)
14. Le manuel de ksh précise que les expressions peuvent faire appel aux opérateurs du langage C (sauf ++ et --), avec les mêmes règles de syntaxe, de priorité et d’associativité. Attention : l’évaluation est faite en entiers signés longs, sur un nombre de bits qui dépend de l’implémentation : 64 bits pour AT&T ksh sous SunOS 5.10 (y compris sur i386), 32 bits pour pdksh (sur i386). [↑](#footnote-ref-14)
15. La syntaxe `*commande*` présente des difficultés d’emploi, liées notamment au caractère backslash (\) utilisé comme échappement devant $ et ` à l’intérieur des backquotes, ainsi que des limitations qui n’affectent pas la forme $(*commande*), qui s’avère beaucoup plus naturelle et simple d’utilisation. Cf. . [↑](#footnote-ref-15)
16. Les codes d’exit sous Unix sont de types *int*, mais seuls les 8 bits de poids faible sont transmis au processus parent ; un script shell qui se termine par un exit -1 renvoie donc 255 à l’appelant ; exit -2 renvoie 254, etc. [↑](#footnote-ref-16)
17. Attention, certains de ces cas ne sont pas anodins : cf. description détaillée à la section 3.7.1, page 26. [↑](#footnote-ref-17)
18. *default\_value* peut être nulle (exemple : file\_in=${1:-}). Par ailleurs les substitutions de paramètres s’appliquent, ce qui autorise des substitutions complexes, telles que ${2:-${user\_default:-${system\_default:-${script\_default}}}}. [↑](#footnote-ref-18)
19. La syntaxe if [[ *…* ]]; then … avec les crochets doublés présente 2 avantages sur la syntaxe classique if [ … ]; then … (ou encore sur la forme traditionnelle if test … ; then … qui est équivalente à if [ … ]; then …) :

    Les opérateurs de l’expression conditionnelle sont déterminés avant de traiter leurs arguments.

    Les arguments ne subissent pas de découpage (*field splitting*) ni d’expansion des noms de fichiers (*pathname expansion*).

    Ceci élimine de nombreuses erreurs possibles avec la syntaxe classique. Ainsi, if [[ -n ${*param*} ]]; then … est toujours interprété comme un test de non-nullité et demeure syntaxiquement correct même si ${*param*} s’évalue à une chaîne vide ; on préfère cependant la variante : if [[ -n ${*param*:+X} ]]; then … pour 2 raisons : (a) pour éviter une erreur avec l’option nounset si *param* n’est pas défini ; (b) de cette manière, la longueur de *param* n’influe pas sur les performances : sinon, la durée du test est proportionnelle à la longueur de *param* (problème corrigé en ksh93) ! [↑](#footnote-ref-19)
20. Ce problème était présent sous Solaris 8 notamment, lorsque la valeur de *param* pouvait être confondue avec un opérateur de la commande test, tel que "!", "=", "!=", etc. Il semble corrigé sous Solaris 10 et dans la version testée de pdksh, mais il est possible que d’autres implémentations aient conservé ce comportement (il subsiste en Bourne shell). Les formes recommandées ci-dessus ne substituent jamais la valeur réelle de *param*, ce qui évite tout effet de bord lié à des valeurs particulières ; elles doivent donc être utilisées lorsque *param* peut prendre n’importe quelle valeur. A contrario, si les valeurs de *param* sont connues et non susceptibles de provoquer ce type d’erreur, les variantes simplifiées ci-dessus peuvent être utilisées. [↑](#footnote-ref-20)
21. L’emploi de la syntaxe if [[ … ]]; then … rend inutiles les guillemets doubles ; cf. note 19 page 16. [↑](#footnote-ref-21)
22. Cette forme convient, y compris lorsque l’option nounset est activée, dans le cas le plus général où *param1* et *param2* peuvent être non définis, si l’on admet l’égalité d’un paramètre défini et nul et d’un paramètre non défini. [↑](#footnote-ref-22)
23. Ici les guillemets doubles restent obligatoires au niveau de l’argument "${*param2*:-}" afin d’effectuer un test d’égalité et non un test de conformité à un motif (cf. règle ). [↑](#footnote-ref-23)
24. Cette limitation est corrigée en ksh93. [↑](#footnote-ref-24)
25. Avec les mêmes règles de syntaxe que la commande let, en particulier l’omission possible du symbole $ devant les noms des paramètres déjà définis (cf. note 13 page 11). [↑](#footnote-ref-25)
26. Cf. règle relative aux codes d’exit, page 13, et règle relative aux traces applicatives, page 34. [↑](#footnote-ref-26)
27. Attention : ne pas confondre getopts (avec un « s ») qui est une commande built-in et getopt (sans « s ») qui n’est pas une commande built-in ; getopt (sans « s ») est obsolète et ne doit plus être utilisée. [↑](#footnote-ref-27)
28. Où *c* est un caractère, alphanumérique en général. Il est possible d’utiliser *certains* caractères non alphanumériques, mais ce n’est pas une pratique courante : le point d’interrogation (?) et les deux points (:) sont interdits par la spécification de getopts, le point d’exclamation (!) semble fonctionner, mais non le dièse (#)… [↑](#footnote-ref-28)
29. Dans le cas contraire, getopts émets automatiquement certains messages d’erreurs, ce qui est rarement souhaité sachant que le format de ces messages dépend entièrement de l’implémentation et de la *locale* courante. En outre, dans cette configuration le script ksh reçoit moins d’information en cas d’erreur : si une option inconnue est détectée, le paramètre OPTARG n’est pas valorisé. [↑](#footnote-ref-29)
30. Cette affectation doit servir uniquement dans le cadre du contrôle de la syntaxe d’appel : dans le cœur du script il est préférable d’utiliser un paramètre distinct avec un nom plus explicite en rapport avec la fonction du paramètre (ici, le chemin du fichier de traces), plutôt que *opt\_l* qui est propre au traitement des options. Non seulement cela rendra le script plus clair, mais cela réduira le couplage entre le cœur du script et les étapes de contrôle de la syntaxe d’appel. [↑](#footnote-ref-30)
31. La raison d’être de ce paramètre est qu’il permet de poursuivre l’analyse des options au-delà de la première erreur rencontrée, afin de retourner le maximum d’information, sous forme de messages d’erreurs, à l’utilisateur. Ce comportement (indiquer à l’utilisateur toutes les erreurs, et non pas uniquement la première) est généralement souhaité, en particulier pour les commandes complexes comportant de multiples options. [↑](#footnote-ref-31)
32. Cf. règle page 14. [↑](#footnote-ref-32)
33. La recherche s’arrête également si la chaine « -- » (deux signes moins (-) consécutifs) est rencontrée sur la ligne de commande. [↑](#footnote-ref-33)
34. L’interface de getopts ne permet pas de déterminer si les options non reconnues sont précédées d’un signe plus (+) ou d’un signe moins (-). [↑](#footnote-ref-34)
35. Alternativement, il est facile d’interdire les options introduites par le signe plus, en traitant le cas +\*) comme un cas d’erreur dans l’instruction *case*. [↑](#footnote-ref-35)
36. Dans ce cas $OPTARG vaut =*logfile*, il faut donc supprimer le signe égal (=) dans le script si on souhaite supporter cette syntaxe. [↑](#footnote-ref-36)
37. C’est pourquoi la seule situation dans laquelle l’omission de l’argument d’une option est détectée et signalée comme une erreur (cas traité à la ligne 29 dans notre exemple) est celle dans laquelle la liste des paramètres positionnels est épuisée. [↑](#footnote-ref-37)
38. Sans parler des nouvelles difficultés que cette fonctionnalité ajoute. Par exemple, l’abréviation du double signe moins en un seul n’est pas supportée (il est vrai que cela ouvrirait un vaste champ d’ambiguïtés supplémentaires), de sorte que -log logfile – notez l’erreur de l’utilisateur : il manque un signe moins – est interprété comme l’option -l suivie de l’argument « og », suivi de l’argument non optionnel « logfile » ; ce n’est sans doute pas ce que l’utilisateur attendait. [↑](#footnote-ref-38)
39. L’affectation de $? à un paramètre est obligatoire car la valeur de $? est mise à jour après chaque commande. Ainsi, tester la valeur de $? entraîne un appel implicite à la commande built-in test de ksh, ce qui suffit à affecter une nouvelle valeur à $?. Démonstration : la séquence « false ; if [ $? -ne 0 ]; then echo $? ; fi » affiche toujours 0, alors que la séquence « false ; rc=$? ;   
    if [ $rc -ne 0 ]; then echo $rc ; fi » affiche toujours 1. [↑](#footnote-ref-39)
40. Les cas où tenter une action de reprise automatique après un échec technique est justifié sont relativement peu fréquents. Du reste ce type de traitement des erreurs est nettement plus complexe à implémenter qu’un simple arrêt du script, et il ne dispense pas de logguer convenablement (avec détail et précision) les échecs techniques, même s’ils sont correctement repris. [↑](#footnote-ref-40)
41. Si la commande *k* écrit dans un pipe alors que la commande (*k*+1) à l’autre extrémité du pipe s’est terminée prématurément, la commande *k* reçoit le signal 13 (SIGPIPE : broken pipe), ce qui termine la commande *k* à moins qu’elle n’ait positionné une action spécifique pour traiter ce signal. Cette situation n’est qu’un cas possible parmi d’autres. [↑](#footnote-ref-41)
42. Cf. section 3.7.2 ci-après, page 27. [↑](#footnote-ref-42)
43. Rappel : une commande built-in (ou interne) est une commande intégrée à l’interpréteur ksh : elle s’exécute directement dans le processus ksh, contrairement aux commandes externes qui sont toujours exécutées dans des processus fils – c’est pourquoi l’emploi des commandes built-in est recommandé pour des raisons de performances (cf. section 4.1 page 51). Les commandes built-in sont toujours appelées sans préciser de chemin : elles ont priorité sur les commandes présentes dans les répertoires du PATH. [↑](#footnote-ref-43)
44. D’après la documentation de AT&T ksh, wait est une commande built-in spéciale ; cependant d’après nos tests, son comportement en cas d’erreur (cas testé : échec de redirection) est celui d’une commande ordinaire ; pdksh liste cette commande parmi les commandes built-in régulières. [↑](#footnote-ref-44)
45. Cf. section 3.7.2 page 27. [↑](#footnote-ref-45)
46. Remarque : même si un système de fichiers est plein à 100%, ce qui signifie qu’il n’y a plus aucun *bloc* disponible pour les *données*, il peut demeurer possible (au moins dans un premier temps) de créer de nouveaux fichiers (vides), car la création d’un fichier n’implique que la réservation d’un i-node dans la table des i-nodes et l’écriture d’une entrée (*directory entry*) dans un fichier de répertoire. Et ce, jusqu’à ce que le fichier de répertoire soit lui-même saturé et ne puisse plus accepter de nouvelles entrées. [↑](#footnote-ref-46)
47. Il serait tentant d’examiner la variable ERRNO, mais : (1) ce n’est absolument pas portable, notamment cette variable n’est pas implémentée dans pdksh et elle est supprimée en ksh93 ; (2) ce mécanisme n’est pas garanti à 100% : par exemple, l’appel à la fonction système write qui effectue l’écriture dans le fichier de redirection peut être différé si celui-ci est *bufferisé* ; (3) la valeur retournée n’est pas nécessairement pertinente, car elle dépend de la séquence des appels systèmes effectués par l’implémentation. Le manuel de ksh souligne que ERRNO est destinée à des fins de débogage uniquement ; tout autre utilisation est à proscrire. [↑](#footnote-ref-47)
48. Seules les variables spécifiques à l’application doivent être définies dans ce fichier : les variables propres aux composants du socle technique (ORACLE\_HOME, JAVA\_HOME, LD\_LIBRARY\_PATH, etc.) sont à définir dans d’autres fichiers. En ce qui concerne les bases de données, seuls les noms des sources de données devraient figurer dans le fichier d’environnement applicatif. [↑](#footnote-ref-48)
49. Inclure le fichier d’environnement applicatif depuis le .profile du user applicatif est un choix structurant qui dispense, par la suite, de devoir inclure explicitement ce fichier dans tous les scripts de l’application. [↑](#footnote-ref-49)
50. Cette exigence est rencontrée sur certains sites d’exploitation, où le .profile est lu non seulement par des scripts ksh mais également par des shells de type Bourne ; c’est le cas notamment des scripts générés par l’ordonnanceur Control-M. [↑](#footnote-ref-50)
51. Quelques exemples issus d’une longue liste : /usr/bin/ps et /usr/ucb/ps, /usr/bin/mv et /usr/xpg4/bin/mv, /usr/bin/rm et /usr/xpg4/bin/rm, /usr/bin/grep et /usr/xpg4/bin/grep, enfin /usr/ccs/bin/make et /usr/local/bin/make (GNU make, s’il est installé). [↑](#footnote-ref-51)
52. En outre, rien n’empêche d’ajouter un répertoire quelconque en tête du PATH, ou encore de créer des programmes ou des scripts exécutables avec le même nom que des commandes existantes. [↑](#footnote-ref-52)
53. Supposons que le PATH soit fixé dans un premier temps à /usr/bin:/usr/ucb. À un stade ultérieur du développement, vous souhaitez utiliser /usr/xpg4/bin/grep à la place de /usr/bin/grep car la version XPG4 de grep est plus riche fonctionnellement ; rien de plus facile que d’ajouter /usr/xpg4/bin en tête du PATH… mais ce faisant vous remplacez 70 commandes de /usr/bin par leurs versions compatibles XPG4 : pouvez-vous garantir que ce changement n’a aucun effet de bord sur votre script ? [↑](#footnote-ref-53)
54. Cette variable est normalement définie dans un fichier d’environnement applicatif commun (tel que $HOME/profile.xyz) qui est soit directement inclus dans tous les scripts ksh, soit inclus dans le fichier $HOME/.profile du user applicatif ; cf. exemple page 31. [↑](#footnote-ref-54)
55. La qualité des traces peut être améliorée si certaines situations d’erreurs sont détectées *a priori* : par exemple, avant de copier un fichier, il peut être utile de vérifier que le fichier source est bien présent ; cette vérification n’est pas toujours indispensable, dans la mesure où le cas échéant la commande de copie échouera avec un message d’erreur explicite (qu’il est possible de capturer), mais elle permet un traitement adapté de l’erreur car la nature exacte du problème est parfaitement caractérisée. Au contraire, si on se contente d’une simple vérification a posteriori du code d’exit ($?), la cause exacte de l’échec ne sera pas connue. Toutefois détecter certaines erreurs *a priori* ne dispense pas de contrôler *a posteriori* : par exemple, ce n’est pas parce que la vérification préalable de présence d’un fichier réussit que le fichier est ouvert avec succès par la suite ; le contrôle systématique du code d’exit ($?) de toutes les commandes appelées demeure donc strictement indispensable. [↑](#footnote-ref-55)
56. En pratique il n’est pas toujours possible de se prémunir contre toutes les erreurs envisageables ; ce n’est d’ailleurs pas forcément souhaitable (cas des variables non définies : si l’option set -o nounset est utilisée, vérifier en début de script que les variables d’environnement applicatives sont bien définies est inutilement coûteux). En cas d’erreur d’exécution (cf. section 3.7 page 26), dans certains cas le script sera terminé immédiatement avec un message d’erreur sur la sortie des erreurs (stderr) ; bien évidemment, ce message ne pourra pas figurer dans le fichier de traces applicatives si celui-ci est généré par le script lui-même, c’est pourquoi il est très important que la sortie stderr soit conservée dans la mesure du possible, afin de faciliter le diagnostic en pareil cas. [↑](#footnote-ref-56)
57. Le paramètre SECONDS de ksh, qui indique le nombre de secondes écoulées depuis le lancement du processus, permet d’éviter des appels inutiles à la commande date en déterminant si l’on est toujours dans la même seconde que lors du message précédent. [↑](#footnote-ref-57)
58. En plus de l’horodatage, il peut être utile d’inclure le PID du processus (paramètre $$) dans le nom des fichiers de traces, afin de distinguer les fichiers de traces créés par un même script qui serait lancé 2 fois ou davantage dans la même seconde. [↑](#footnote-ref-58)
59. Un nombre très élevé de fichiers dans un même répertoire peut non seulement causer des difficultés pratiques, mais également entraîner des problèmes de performances avec certains types de systèmes de fichiers. Anecdote vécue sur un système Solaris 8 qui hébergeait de multiples applications : régulièrement le répertoire /tmp dépassait un seuil critique de plusieurs dizaines de milliers de fichiers, au-delà duquel ses performances s’écroulaient pour la plupart des opérations : création et suppression de fichiers, parcours du répertoire. Les performances du host se trouvaient alors fortement perturbées, étant donné que le répertoire /tmp est très fréquemment utilisé. Par exemple, les *here-documents* – cf. man  – utilisés dans les scripts ksh sont implémentés par des fichiers temporaires créés obligatoirement sous /tmp : d’un seul coup, des dizaines de scripts ksh se trouvaient fortement ralentis voire presque gelés par ce phénomène. [↑](#footnote-ref-59)
60. La résolution des noms dans ksh fonctionne dans l’ordre suivant : (a) commandes built-in spéciales ; (b) fonctions définies par l’utilisateur ; (c) commandes built-in ordinaires ; (d) recherche des commandes dans le PATH. [↑](#footnote-ref-60)
61. À la différence des commandes externes, qui sont exécutées dans des sous-processus, les fonctions ksh sont exécutées dans le processus courant, c’est pourquoi elles partagent avec celui-ci :

    Les définitions d’alias, de fonctions, ainsi que les options du shell (telles que : set -o nounset)

    Les paramètres, à l’exception (i) des paramètres positionnels, (ii) des paramètres déclarés comme locaux à l’aide de la commande typeset, et enfin (iii) de certains paramètres spéciaux (OPTARG, OPTIND, etc.)

    Le PID (paramètre $$)

    Le répertoire courant (paramètre $PWD)

    Les fichiers ouverts. [↑](#footnote-ref-61)
62. Cette fonction vérifie que son argument est conforme à un motif défini dans la syntaxe des motifs de noms de fichiers (cf. règle page 17). Une alternative consisterait à utiliser la commande externe grep, cependant l’implémentation ci-dessus est nettement plus rapide (plus de 200 ×) que si grep était utilisée, en raison du coût inhérent à la création des processus sous Unix ; cf. section 4.1 page 51. [↑](#footnote-ref-62)
63. Ce type de trap n’est pas supporté dans pdksh ; cf. sections 3.12 page 46, et 6.3 page 64. [↑](#footnote-ref-63)
64. Le fait que la sémantique des fonctions définies dans la syntaxe Bourne shell / POSIX change en ksh93 est également une raison majeure pour s’abstenir de l’utiliser. En effet dans ksh93, contrairement à ksh88 : (i) ces fonctions partagent leurs variables avec l’unité d’exécution appelante, même si la commande typeset est utilisée ; et (ii) une erreur d’exécution à l’intérieur d’une telle fonction provoque un abandon immédiat du script, et non plus un simple échec de la fonction concernée. [↑](#footnote-ref-64)
65. Si on reprend l’exemple précédent, en ksh93 le résultat change de la façon suivante : les définitions des paramètres var1 et var2 à l’intérieur de la fonction foo ne sont plus visibles qu’à l’intérieur de cette fonction ; dans la fonction bar seules les définitions du script principal sont visibles, et cela quelque soit le chemin d’appels de fonctions conduisant à la fonction bar ; les lignes 3 et 4 dans le résultat du script deviennent donc identiques aux lignes 5 et 6. [↑](#footnote-ref-65)
66. Pour l’homogénéité de l’interface, Il n’est pas inutile de prévoir des options correspondant à des comportements par défaut. Entre autres bénéfices, la documentation de ces options (inutiles en apparence) fournit un prétexte pour décrire ces comportements par défaut, ce qui complète et clarifie la documentation. [↑](#footnote-ref-66)
67. Les pré-requis sont les conditions qui doivent impérativement être remplies avant d’appeler la fonction. Par définition, la satisfaction des pré-requis relève de la responsabilité de l’appelant, cependant il n’est pas inutile d’indiquer dans la documentation si la fonction vérifie ou non si ces conditions sont remplies, ainsi que les conséquences éventuelles si elles ne le sont pas. Par ailleurs il est recommandé, dans la mesure du possible, que les fonctions communes servant d’interface (par opposition aux fonctions communes présentes dans le framework mais qui ne sont pas supposées être appelées directement) soient codées de manière défensive et fassent donc le moins possible confiance à l’appelant ; ce principe doit conduire à inclure un maximum de vérifications de pré-requis dans les fonctions d’interface, dans la limite où cela n’est pas préjudiciable aux performances. [↑](#footnote-ref-67)
68. Il est utile d’établir une convention de nommage particulière pour les paramètres globaux, telle que *PREFIXE*\_*nomparam* ou \_\_*PREFIXE*\_*nomparam*, où *PREFIXE* est un préfixe propre au framework et *nomparam* le nom du paramètre concerné. Une telle convention déroge à la règle fixant l’utilisation de la casse (cf. règle page 10), mais elle a l’intérêt de rendre très visible l’utilisation des paramètres globaux. [↑](#footnote-ref-68)
69. Comme tous les paramètres spéciaux dans ksh, si le paramètre OPTIND est supprimé au moyen de la commande unset, il perd son caractère spécial et il est ensuite traité comme un paramètre ordinaire. [↑](#footnote-ref-69)
70. Sauf cas particulier des fonctions déclarées en syntaxe Bourne shell / POSIX dans pdksh – cf. section page 66. [↑](#footnote-ref-70)
71. D’après les documentations respectives de HP et IBM, cette limite est de 1024 éléments sous HP-UX 11.3, et de 512 éléments seulement sous IBM AIX 6.1 ! [↑](#footnote-ref-71)
72. Le dépassement de l’indice maximum autorisé provoque l’erreur « *subscript out of range* » qui entraîne l’arrêt du script (dans une fonction, l’abandon de la fonction) avec le code d’exit 1. [↑](#footnote-ref-72)
73. Si vous devez vraiment utiliser un tableau creux, n’oubliez pas d’inclure une explication complète en commentaires. [↑](#footnote-ref-73)
74. Rappel : les indices de tableaux sont des expressions arithmétiques, c’est pourquoi le symbole $ n’est pas nécessaire devant idx. [↑](#footnote-ref-74)
75. A ne pas confondre avec "${*tableau*[\*]}" qui est équivalente à la *chaîne* "${*tableau*[*n*1]} ${*tableau*[*n*2]} … ${*tableau*[*nN*]}". Les expressions "${*tableau*[\*]}" et "${*tableau*[@]}" fonctionnent donc de manière analogue aux expressions "$\*" et "$@" – entre guillemets doubles – pour manipuler les paramètres positionnels. [↑](#footnote-ref-75)
76. Les traps sur signaux TERM, USR1, USR2, etc. sont surtout utilisés pour contrôler des processus asynchrones, c’est pourquoi ils sont très rarement employés dans les scripts applicatifs, mais peuvent être rencontrés dans des scripts d’infrastructure (A/R, etc.). [↑](#footnote-ref-76)
77. Seuls les traps sur EXIT sont concernés par cette distinction entre les fonctions définies dans la syntaxe Bourne shell / POSIX et les fonctions non-POSIX (définies à l’aide du mot-clef function). [↑](#footnote-ref-77)
78. Si un processus s’exécute en avant-plan, ksh attend la fin de ce processus pour traiter le signal – à moins qu’aucun trap ne soit en place pour traiter le signal, auquel cas le processus ksh se termine sans attendre le processus fils. [↑](#footnote-ref-78)
79. Précisions : bien que ce trap soit défini à l’intérieur de la fonction, il s’exécute dans le contexte de l’appelant : les variables locales de la fonction qui a défini le trap ne sont plus visibles. Par ailleurs, la valeur de $? qui sera restituée à l’appelant n’est pas visible par le code du trap : l’environnement dans lequel le trap s’exécute est « *tel que immédiatement après la dernière commande exécutée juste avant l’événement déclencheur du trap* », c-à-d. juste avant le return ou l’erreur d’exécution qui termine la fonction. [↑](#footnote-ref-79)
80. Encore faut-il qu’il s’agisse d’un cas de terminaison normale : si la terminaison du script fait suite à une erreur d’exécution, pdksh n’exécute pas le trap sur EXIT – cf. règle TRAP-03. [↑](#footnote-ref-80)
81. Exemples d’erreurs d’exécution provoquant l’exit du script (AT&T ksh) ou l’abandon de l’unité d’exécution courante (pdksh) :

    Échec de redirection lors d’un appel à une commande built-in spéciale

    Erreur de syntaxe dans l’argument de la commande eval

    Commande non trouvée appelée par la commande built-in exec (exit immédiat du script, même avec pdksh).

    Cf. section 3.7 page 26. [↑](#footnote-ref-81)
82. Il s’agit de la fonction au sommet de la pile d’appels : aucun dépilement n’a lieu, contrairement à ce qui se produirait si le signal était traité à un niveau quelconque ; dans ce cas particulier le trap sur EXIT est exécuté dans le contexte de la fonction concernée et non dans le contexte de l’appelant. [↑](#footnote-ref-82)
83. Les signaux ignorés par l’appelant le sont aussi dans les fonctions appelées (ainsi que, plus largement, dans les commandes appelées et dans les sous-processus) ; de plus si une fonction tente de définir un trap sur un signal déjà ignoré au moment de l’appel, la tentative n’a aucun effet. [↑](#footnote-ref-83)
84. Pour que le trap défini au niveau de f1 soit déclenché, il faut que f1 soit en cours d’exécution lorsque le signal est reçu, ou qu’une fonction appelée par f1 soit en cours d’exécution, dans laquelle l’action par défaut demeure définie. [↑](#footnote-ref-84)
85. D’une manière générale, la simplicité est plus importante que les performances ; les critères s’inversent seulement dans quelques cas, pour des scripts ou des fonctions très fréquemment appelés. Dans tous les cas, la clarté des scripts est primordiale. [↑](#footnote-ref-85)
86. Le langage des expressions régulières utilisé par grep est bien sûr plus puissant que le langage des motifs de noms de fichiers reconnu par ksh ; cependant celui-ci s’avère suffisant dans un grand nombre de situations courantes. [↑](#footnote-ref-86)
87. Cet idiome entraîne bien la création d’un sous-processus, mais il permet de lire tous les champs de la ligne en une seule fois. [↑](#footnote-ref-87)
88. Attention, cet idiome n’est pas compatible avec pdksh ; cf. section 6.1 page 62. [↑](#footnote-ref-88)
89. Dans le même ordre d’idées, une fonction implémentant l’algorithme « quick sort » pour trier un tableau sera un peu plus rapide qu’une implémentation utilisant la commande externe sort, mais uniquement si les tableaux à trier sont de petite taille, typiquement de moins de 100 éléments. [↑](#footnote-ref-89)
90. Malheureusement, il y a aussi de grandes chances pour que le PATH utilisé lorsque les scripts en question ont été validés initialement ne soit documenté nulle part. [↑](#footnote-ref-90)
91. Le *Linker and Libraries Guide* de Solaris 10 précise : « *although useful as a temporary mechanism of influencing the runtime linker's search path, the use of LD\_LIBRARY\_PATH is strongly discouraged in production software. Any dynamic executables that can reference this environment variable will have their search paths augmented. This augmentation can result in an overall degradation in performance.* » Malheureusement, beaucoup d’éditeurs ne prennent pas le soin d’inscrire un *runpath* dans leurs programmes binaires (car cela nécessiterait une étape d’édition des liens au moment de l’installation) et de ce fait imposent d’utiliser LD\_LIBRARY\_PATH pour désigner les répertoires dans lesquels leurs bibliothèques d’objets sont installées. [↑](#footnote-ref-91)
92. En définitive, le seul moyen de se prémunir des modifications intempestives de LD\_LIBRARY\_PATH consiste à définir cette variable soi-même ! Bien évidemment cela n’est envisageable qu’à condition de gérer cette variable de façon centralisée au niveau du *framework* ksh applicatif commun, afin que cette gestion ne soit pas ré-implémentée localement dans chaque script, ce qui la rendrait impossible à maintenir. Une telle solution nécessite une parfaite communication avec les ingénieurs système, afin qu’en cas de montée de version (ou toute autre modification sur l’installation des produits concernés) les changements nécessaires sur les fichiers d’environnement applicatifs soient déployés de façon synchrone. [↑](#footnote-ref-92)
93. L’OS conserve en cache RAM les chemins les plus utilisés (voire des répertoires complets) afin de minimiser les lectures physiques de répertoires. Ce mécanisme, appelé *directory name lookup cache* (DNLC), est généralement très efficace : la recherche d’un fichier dans un répertoire individuel du PATH prend typiquement entre 1/1000ème et 1/100ème de la durée complète de lancement d’un nouveau processus ; bien sûr cette durée est à multiplier par le nombre de répertoires interrogés. Ce mécanisme n’est bien sûr pas spécifique au PATH : il s’applique en fait à toutes les recherches de fichiers. [↑](#footnote-ref-93)
94. Cf. note 91, page 56. [↑](#footnote-ref-94)
95. Exemple : sous Solaris, vérifier l’existence d’un fichier peut être accompli aussi bien avec : test -e *file* qu’avec : test -a *file*, les arguments -a et -e de la commande test étant synonymes ; sous HP-UX 11.3, seule la syntaxe test -a *file* est disponible. [↑](#footnote-ref-95)
96. Cf. section 3.7.2 page 27. [↑](#footnote-ref-96)
97. Rappel : les appels de commandes composées de la forme { liste } sont traités comme les appels de fonction. [↑](#footnote-ref-97)
98. Rappel : bug dans ksh sous Solaris 10 : si le script contient à la fois des fonctions définies avec le mot-clef function et des fonctions définies dans la syntaxe Bourne shell / POSIX, le comportement des traps sur EXIT définis à l’intérieur des fonctions n’est pas garanti – cf. règle TRAP-05 page 49. [↑](#footnote-ref-98)
99. Cf. man  : *“the environment in which the shell executes a trap on EXIT is identical to the environment immediately after the last command executed before the trap on EXIT was taken.”* [↑](#footnote-ref-99)
100. Rappel : dans le cas d’un trap sur EXIT défini à l’intérieur d’une fonction non-POSIX, le trap est exécuté au retour de la fonction, dans le contexte de l’appelant : les variables locales propres à la fonction qui a défini le trap ne sont plus visibles. [↑](#footnote-ref-100)
101. Attention : dans AT&T ksh les lignes à l’intérieur des fonctions sont numérotées relativement au début de la fonction, et non relativement au début du fichier. [↑](#footnote-ref-101)
102. Ce paramètre est mentionné par souci d’exhaustivité, cependant son utilisation, intrinsèquement non portable, est plus que déconseillée – le manuel de ksh précise : « *this value is system dependent and is intended for debugging purposes* ». En outre ce paramètre perd sa signification spéciale en ksh93. [↑](#footnote-ref-102)
103. Comportement constaté sous Solaris 10 ; ce point n’étant pas spécialement documenté, le comportement des autres implémentations de AT&T ksh peut être différent. [↑](#footnote-ref-103)
104. Par exemple, tenter de définir la fonction echo échoue avec le message d’erreur « ksh: restricted: echo is a shell builtin » ; un autre essai utilisant le mécanisme des fonctions *autoload* échoue également (aucun message n’est émis, mais typeset -fu echo retourne 1). [↑](#footnote-ref-104)
105. D’après leurs documentations respectives, la limite est de 1024 éléments sous HP-UX 11.3 et de 512 éléments sous IBM AIX 6.1. [↑](#footnote-ref-105)
106. Le paramètre OPTIND perd cette particularité s’il est supprimé au moyen de la commande unset puis re-déclaré. [↑](#footnote-ref-106)
107. D’après nos tests, utiliser la commande typeset à l’intérieur d’une fonction afin de créer une instance locale du paramètre OPTIND ne suffit pas à éviter les interférences. [↑](#footnote-ref-107)
108. Le bug ne se produit pas si une liste de valeurs est utilisée pour initialiser le tableau ; il se produit seulement en l’absence d’une liste de valeurs, c-à-d. si on pré-déclare un tableau vide. [↑](#footnote-ref-108)
109. Ce script n’est pas conforme à la règle page 10 concernant l’utilisation des paramètres positionnels ($1, $2, etc.) dans les scripts ; le respect de cette règle est donc un moyen très simple d’éviter ce bug. [↑](#footnote-ref-109)
110. Les lignes du script sont numérotées à partir de 1 ; les lignes des fonctions sont numérotées à partir de 0 (= numéro de la ligne qui porte le mot-clef function, ou l’identifiant de la fonction dans le cas d’une fonction définie dans la syntaxe Bourne shell / POSIX). [↑](#footnote-ref-110)
111. Bell Telephone Laboratories était alors une filiale d’AT&T. En 1996, lors de la scission entre AT&T et Lucent Technologies, Bell Labs a été intégrée à Lucent – désormais Alcatel-Lucent – cependant certains chercheurs, parmi lesquels David Korn, ont poursuivi leur carrière au sein d’AT&T Laboratories, l’entité de R&D d’AT&T. Pour plus d’information, cf. :

     http://www.corp.att.com/history/milestones.html

     http://www.bell-labs.com/history/unix/blcontributions.html

     http://kornshell.com/doc/faq.html.

     http://www.gnu.org/software/autoconf/manual/autoconf.html#Shellology

     http://www.faqs.org/faqs/unix-faq/shell/shell-differences [↑](#footnote-ref-111)
112. Comme illustré en différentes occasions dans ce document, ces améliorations propriétaires sont elles-mêmes une source d’incompatibilités entre les implémentations. Par exemple, la taille maximum des tableaux en ksh est égale à 1024 sous HP-UX (11.3), à 512 sous IBM AIX (6.1) et à 4096 sous Solaris 10. Autre exemple : test -e et test -a sont synonymes sous Solaris 10, mais seul test -a est reconnu sous HP-UX. Finalement ces différences font que la portabilité des scripts ksh, même entre implémentations issues de l’implémentation originale d’AT&T, n’est pas de 100%. [↑](#footnote-ref-112)
113. La version ksh88 reste disponible dans un package optionnel ; si ce package est installé, la commande /usr/sunos/bin/ksh correspond à ksh88. [↑](#footnote-ref-113)
114. Idem pour Oracle Enterprise Linux 5 et 6 ; un package pdksh optionnel est également disponible pour ces distributions. [↑](#footnote-ref-114)
115. Attention, cet ouvrage date de 1995 ; il ne tient donc pas compte des multiples améliorations apportées dans les releases ultérieures de ksh93. [↑](#footnote-ref-115)
116. Cette liste concerne les implémentations de ksh et ksh93 développées par AT&T Research, non les implémentations disponibles dans les Unix commerciaux d’Oracle, HP, IBM, etc., qui peuvent comporter des différences par rapport aux implémentations d’AT&T. [↑](#footnote-ref-116)