CENTRE NATIONAL D'ÉTUDES SPATIALES

REFERENTIEL NORMATIF du CNES

RNC

Référence: **RNC-CNES-Q-HB-80-516**
Version 5
02 Juin 2008

MANUEL

ASSURANCE PRODUIT

REGLES POUR L'UTILISATION DES SHELLS SOUS UNIX

ACCORD du Bureau de Normalisation	**BN n° 34 du 25/06/07 – BN n°44 du 08/09/08**
APPROBATION Président du CDN Alain CUQUEL	

PAGE D'ANALYSE DOCUMENTAIRE

TITRE : **REGLES POUR L'UTILISATION DES SHELLS SOUS UNIX**
MOTS CLES : Règles, Programmation, Codage, Shell Unix
NORME EQUIVALENTE : Néant
OBSERVATIONS : Néant
RESUME : Ce document présente les règles pour l'utilisation du Shell UNIX. Il s'applique au développement et à la maintenance des systèmes informatiques sols du CNES.
SITUATION DU DOCUMENT : Ce document fait partie de la collection des Manuels approuvés du Référentiel Normatif du CNES (RNC). Il est affilié au document « RNC- ECSS-Q-ST-80 Software Product Assurance ».

NOMBRE DE PAGES : 44	**LANGUE** : Française

Progiciels utilisés / version : Word 2002
SERVICE GESTIONNAIRE : Inspection Générale Direction de la Fonction qualité (IGQ)
AUTEUR(S) : **DATE : 02/06/2008** **Repris par J-C. DAMERY**

PAGES DES MODIFICATIONS

VERSION	DATE	PAGES MODIFIEES	OBSERVATIONS
PR.0	02/10/95	Toutes	Création document avec intégration des remarques suite à large relecture interne
PR.1	12/01/96	Toutes	Mise en forme Création du niveau recommandation
1.0	18/06/96	i.1 ; i.2	Approbation du Comité Technique Référentiel
2.0	11/01/99	Toutes	Ajout de la référence à l'annexe technique associée à ce document réalisé par Logiqual (G. Anési) et normalisation
3	02/03/00		Nouvelle codification des documents
4	10/12/2006	Toutes	Introduction de règles « BASH » avec le support de P. Ficheux (Open Wide). Mise en commun de règles et mise en forme, avec le support de T. Leydier (Virtualité Réelle) Modification du titre. Cf. FEB 48/2006 acceptée au BN n° 22 du 06/03/06. Document accepté au BN n° 34 du 25/06/07 pour introduction dans le RNC.
5	02/06/2008	Toutes	Changement de nomenclature suite à la phase de benchmarking ECSS (ancienne référence « RNC-CNES-Q-80-516 »).

TABLE DES MATIERES

1. INTRODUCTION

Le document « Règles pour l'utilisation des SHELLS sous UNIX » est rattaché à la norme RNC-ECSS-Q-ST-80 « Software Product Assurance». Il décrit les règles applicables pour un logiciel utilisant le langage SHELL UNIX.

Il est complété par une annexe technique (DR5) présentant les moyens de vérifications, quand cela est possible, de certaines règles.

2. OBJET

Le but de ce document est d'établir les règles pour l'utilisation des Shells sous UNIX. Ces règles ont été établies à partir de « l'état de l'art » et du « retour d'expérience » accumulé sur les projets. Ce document est indispensable pour toute utilisation des Shells sous UNIX dans un projet du CNES.

Ce document n'est pas un manuel de référence des Shells sous UNIX, et ne traite pas des aspects interactifs des différents interpréteurs de commandes. Il nécessite les connaissances de base d'un Shell UNIX. Une section spécifique est réservée au choix du Shell à utiliser.

3. DOMAINE D'APPLICATION

Ce document est applicable à tous systèmes informatiques dont le logiciel développé en SHELL fait partie de la livraison du système opérationnel et doit être maintenu. Il est complété par le document « Règles communes pour l'utilisation des langages de programmation » (DA1).

Pour utiliser les règles SHELL UNIX sur un projet, il faudra suivre la procédure suivante :

- ? Sélectionner dans les règles communes et les règles SHELL UNIX, les règles applicables au projet en fonction des critères de tailorisation ; cette sélection s'effectuera avec l'outil de tailorisation.
- ? Adapter certaines règles au projet.

Le document est ainsi destiné à plusieurs types de lecteurs :

- ? le chef de projet qui doit spécifier correctement l'application SHELL UNIX à développer,
- ? le chef de projet et/ou l'ingénieur qualité qui doit sélectionner les règles et éventuellement adapter et compléter les règles en fonction du contexte de réalisation,
- ? les personnes chargées de la réalisation du projet : elles doivent appliquer les règles retenues.

4. DOCUMENTS

4.1. DOCUMENTS DE REFERENCE

DR	Identification	Titre
(DR1)	RNC-ECSS-Q-ST-80	Software Product Assurance
(DR2)	IEEE 1003.2D	Standard for Information Technology Portable Operating System Interface (POSIX) Part 2 : Shell and Utilities
(DR3)		C-Shell programming considered harmful - 11 août 1993 - Tom Christiansen.
(DR4)	RNC-CNES-Q-HB-80-523	Règles de sécurité pour les développements sous UNIX
(DR5)	RNC-CNES-Q-HB-80-516-A	Guide de vérification des Règles du Manuel Shell UNIX

4.2. DOCUMENTS APPLICABLES

DA	Identification	Titre
(DA1)	RNC-CNES-Q-HB-80-501	Règles communes pour l'utilisation des langages de programmation.

4.3. AUTRES DOCUMENTS

- The Korn shell - Prentice Hall - Bolsky et Korn
- UNIX shell programming - McMillan - Kochan et Wood
- The UNIX Programming Environment - Prentice Hall - Kernighan et Pike
- Shell command language (POSIX)
  http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html
- Advanced BASH-scripting guide sur http://www.tldp.org/LDP/abs/html

5. TERMINOLOGIE

5.1. GLOSSAIRE

Terme	Définition
Argument d'un programme	Un argument d'un programme est une information permettant de définir ce sur quoi (au sens large) le programme doit travailler. Par rapport à une option, un argument a un caractère obligatoire.
Interpréteur de commandes, shell	Un interpréteur de commandes est un logiciel capable de réaliser des actions suite à des instructions qu'on lui soumet. Les instructions sont composées de commandes internes et externes, écrites selon la syntaxe de l'interpréteur. L'ensemble syntaxe et commandes internes constitue un langage de programmation à part entière. Ainsi les shells disponibles sur les systèmes d'exploitation de type UNIX sont des interpréteurs de commande. La soumission d'instructions ou de commandes Shell peut se faire: soit en mode interactif i.e. en ligne de commande dans un terminal ; dans ce cas le shell constitue l'interface des utilisateurs avec le système soit via un script shell ; à la différence de programmes classiques, les scripts shell n'ont pas besoin d'être compilés en binaire pour être exécutés.
Option d'un programme	Une option d'un programme est une information transmise au programme, lors de son invocation, permettant d'influer sur le comportement du programme. Par exemple, l'option --help du programme gcc permet d'afficher toutes les options reconnues. Elle définit une partie du comportement de gcc. Par opposition à un argument, une option est à caractère optionnel.
Programme SUID/SGID	Un processus s'exécute normalement sous le couvert de l'utilisateur qui l'a lancé. Il hérite donc des droits d'accès liés à cet utilisateur et à son groupe. La fonctionnalité Setuid ou suid (Set User Id) permet au processus d'hériter des droits d'accès du propriétaire du fichier, quel que soit l'utilisateur qui a lancé le programme. La fonctionnalité Setgid ou sgid (Set Group Id) permet au processus d'hériter des droits d'accès du groupe du fichier.
Script, script shell	Fichier regroupant des commandes et structures de contrôle, qui peut être lu et exécuté par un interpréteur de commandes.

5.2. ABREVIATIONS

5.2.1. Codification des règles

Voir DA1

5.2.2. Autres abréviations ou acronymes

Voir DA1

6. ADEQUATION EVENTUELLE AUX NORMES ET STANDARDS

Sans Objet

7. REGLES

7.1. CONCEPTION / ORGANISATION DU CODE

Org.FonctionPrivées	Un script ne doit pas exporter les fonctions qu'il définit.
M=2;F=1;P=48;V=1	
Quelconque	

Description

Sans Objet

Justification

Une fonction exportée sera héritée par tous les sous-processus qui seront exécutés par la suite. Il peut se révéler difficile de comprendre un script qui hérite d'une telle fonction, définie à un autre niveau fonctionnel.
De plus, la maintenance devient plus difficile, une altération d'une fonction (par exemple pour en modifier le comportement) pouvant provoquer des incidents dans les scripts qui en héritent.

Exemple

Sans objet.

Org.Config	Les initialisations nécessaires à l'ensemble des programmes shell d'un projet doivent être regroupées dans un fichier spécifique.
M=1;F=0;P=76;V=1	
Quelconque	

Description

Sans Objet

Justification

Le fonctionnement des scripts est homogénéisé par l'utilisation d'initialisations communes. Tous les scripts restent alors en phase en ce qui concerne les initialisations globales au projet.
Ce fichier de configuration peut contenir la définition de variables intéressantes, qui n'ont plus à être insérées dans l'environnement de tous les processus, puisqu'elles sont chargées automatiquement par les scripts.

Exemple

Sans objet.

Org.Héritage	Les scripts shell doivent prendre soin de ne pas hériter des alias ou des fonctions définis par l'utilisateur.
M=0;F=2;P=61;V=1	
Quelconque	

Description

Sans Objet

Justification

Le script shell doit être sûr d'exécuter ce qu'il souhaite. Il y a un risque de cheval de Troie, ou de rupture du fonctionnement normal du script, si le script exécute un alias de l'utilisateur et non la commande voulue.

Exemple

```
#!/bin/bash
# unal.sh
#
# Ce programme s'assure qu'il n'hérite pas des alias ou
# fonctions définis par l'utilisateur, en les effaçant
# explicitement
#
function F_effacer { # Pas d'arguments
   # alias produit une liste nom=valeur. On récupère les
   # noms uniquement (avec cut)
   for alias in `alias | cut -d= -f1`
   do
      echo Trouve alias $alias
          unalias $alias
   done
   # typeset -f affiche une liste function nom. On
   # récupère les noms de fonctions uniquement.
   for fonction in `typeset -f | cut -d\  -f2`
   do
      echo Trouve fonction $fonction
      unset -f $fonction
   done;
}

echo --- Avant F_effacer
alias
typeset -f

F_effacer

echo --- Apres F_effacer
alias
typeset –f
```

Ce programme définit une fonction particulière, `F_effacer`, dont le rôle est de retirer (avec `unalias` et `unset -f`) tous les alias et toutes les fonctions dont il pourrait avoir hérité. On notera, ci-dessous, que le script retire la définition de la fonction `F_effacer` elle-même.

```
$ unal.sh
--- Avant F_effacer
autoload=typeset -fu
functions=typeset -f
hash=alias -t -
history=fc -l
integer=typeset -i
```

```
nohup=nohup
r=fc -e -
stop=kill -STOP
suspend=kill -STOP $$
type=whence -v
function F_effacer
Trouve alias autoload
Trouve alias functions
Trouve alias hash
Trouve alias history
Trouve alias integer
Trouve alias nohup
Trouve alias r
Trouve alias stop
Trouve alias suspend
Trouve alias type
Trouve fonction F_effacer
--- Apres F_effacer
```

La norme POSIX définit la commande `unalias -a` afin de réaliser la fonction d'effacement de tous les alias hérités par un shell. Cette fonction est également disponible sous bash.

Org.LimitScripts	Le projet doit fixer une taille maximale des scripts inclus dans des Makefiles.
M=2;F=0;P=62;V=2	
Quelconque	

Description

Sans Objet

Justification

Un Makefile n'est pas un script. Il n'est pas adapté à l'exécution de commandes complexes, pour des raisons de lisibilité et de maintenance. Si la régénération d'une cible particulière demande autre chose qu'une exécution séquentielle de commandes simples, il est préférable de créer un script spécifique qui sera appelé par le Makefile.

Exemple

Voici un exemple de ce que l'on peut écrire (extrait du Système X Window) :

```
install:: $(OBJS)
    @if [ -d $(DESTDIR)$(FONTDIR)/misc ]; then set +x; \
    else (set -x; $(MKDIRHIER) \
          $(DESTDIR)$(FONTDIR)/misc); fi
    @case '${MFLAGS}' in *[i]*) set +e;; esac; \
    for i in $(OBJS); do \
    (set -x; $(INSTALL) -c $(INSTDATFLAGS) $$i \
             $(DESTDIR)$(FONTDIR)/misc); \
    done

install:: fonts.dir
    @if [ -d $(DESTDIR)$(FONTDIR)/misc ]; then set +x; \
```

```
        else (set -x; $(MKDIRHIER) \
              $(DESTDIR)$(FONTDIR)/misc); fi
        $(INSTALL) -c $(INSTDATFLAGS) fonts.dir \
        $(DESTDIR)$(FONTDIR)/misc
```

Cet exemple reste assez lisible, donc presque tolérable. Voici mieux :

```
Makefiles::
 @case '${MFLAGS}' in *[ik]*) set +e;; esac; \
 for i in $(SUBDIRS) ;\
 do \
 echo "making Makefiles in $(CURRENT_DIR)/$$i..."; \
 case "$$i" in \
 ./?*/?*/?*/?*) newtop=../../../../ sub=subsubsubsub;; \
 ./?*/?*/?*) newtop=../../../ sub=subsubsub;; \
 ./?*/?*) newtop=../../ sub=subsub;; \
 ./?*) newtop=../ sub=sub;; \
 */?*/?*/?*) newtop=../../../../ sub=subsubsubsub;; \
 */?*/?*) newtop=../../../ sub=subsubsub;; \
 */?*) newtop=../../ sub=subsub;; \
 *) newtop=../ sub=sub;; \
 esac; \
 case "$(TOP)" in \
 /?*) newtop= upprefix= ;; \
 *) upprefix=../ ;; \
 esac; \
 $(MAKE) $${sub}dirMakefiles UPPREFIX=$$upprefix \
         NEWTOP=$$newtop \
         MAKEFILE_SUBDIR=$$i \
         NEW_CURRENT_DIR=$(CURRENT_DIR)/$$i;\
 Done
```

Org.LimitAwk	Le projet doit fixer une taille maximale, en nombre de règles et en nombre de lignes par règle, pour toute utilisation de `awk`.
M=0;F=1;P=77;V=2	
Quelconque	

Description

Sans Objet

Justification

`awk` est un mini-langage de traitement de fichiers assez pratique. Mais il présente des performances déplorables, et relativement peu de personnes sont réellement habituées à programmer correctement avec. Les messages d'erreur qu'il peut retourner sont souvent limités à '`error near line XXX, bailing out`', ce qui n'est pas spécifiquement informatif (d'autant plus que le numéro de ligne est souvent faux).

Exemple

Les scripts `awk` doivent être limités à cinq règles au maximum, chacune ne dépassant pas cinq instructions.

Org.LimitSed	Le projet doit fixer les limites d'utilisation de `sed`.
M=2;F=1;P=49;V=1	
Quelconque	

Description

Sans Objet

Justification

Comme `awk`, `sed` permet de réaliser facilement certaines opérations. Mais il présente le même type de performances que `awk` (mauvaises) et une syntaxe qui peut se révéler sibylline. La lisibilité d'un script contenant de nombreux appels à `sed` est plus que douteuse.

Exemple

`sed` ne doit être utilisé que pour des substitutions simples sur des chaînes de caractères ou des fichiers.

7.2. PRESENTATION DU CODE

Pr.Interpréteur	La première ligne d'un script doit préciser l'interpréteur de commandes à utiliser. Elle est de la forme : `#!/<chemin_interpréteur_à_utiliser>`..
M=2;F=0;P=64;V=2	
Quelconque	

Description

Sans Objet

Justification

Cette ligne indique l'interpréteur de commandes shell pour lequel le script a été écrit. Cela permet à l'interpréteur d'exécuter le script au sein d'un shell fils adéquate. Ce comportement évite à l'utilisateur de lancer le script avec un interpréteur de commandes pour lequel il n'a pas été écrit et donc de générer des erreurs d'exécution. Si toutefois pour des problèmes de performance, l'utilisateur désire exécuter le script avec le shell courant, il lui est possible d'appeler le script avec la syntaxe suivante :
. <script_shell>
Le « . » désigne le shell courant.

Exemple

Sans Objet

7.3. IDENTIFICATEURS

Id.MotsClés	L'utilisation des mots-clés de l'interpréteur est interdite.
M=2;F=0;P=65;V=1	
Quelconque	

Description

Sans Objet

Justification

Les mots réservés (for, case, if, etc.) ne sont reconnus par le shell, en tant que mots-clés, que lorsqu'ils se trouvent à un emplacement précis d'une commande (souvent le premier mot).
Malgré cela (for est un "mot normal" s'il n'est pas en première position), la lisibilité du programme se dégrade fortement si l'on utilise des mots réservés pour une raison autre que leur rôle fonctionnel lié au shell.

Exemple

```
#!/bin/bash
# *** MAUVAIS EXEMPLE ***
# Utilisation abusive des mots réservés comme s'il
# s'agissait de mots normaux.
#
for for in in do done if fi case esac
do
   echo $for
done
```

Id.Options	Les options des scripts doivent être si possible compréhensibles et différenciées.
M=2;F=1;P=50;V=1	
Quelconque	

Description

Sans Objet

Justification

Il faut éviter des options pouvant être facilement confondues (faute de frappe, inversion de lettres, etc.). Si les options sont auto-descriptives, il devient plus facile de comprendre la signification d'une ligne de commande particulière, sans avoir à se référer à la documentation du programme appelé.

Exemple

Sans objet.

7.4. DONNEES

Don.Constante	Une constante doit être définie comme telle par le mot-clé `readonly`.
M=3;F=1;P=25;V=1	
Quelconque	

Description

Sans Objet

Justification

Le shell interceptera toute tentative de modification de la constante concernée, et déclenchera une erreur.
Avec ksh, on peut aussi utiliser `typeset -r`.

Exemple

```
#!/bin/bash
# const.sh
#
# Exemple de l'intérêt de la définition d'une vraie
# constante : la tentative de modification de pi échoue.
#
readonly pi=3.14159
typeset -r x=2

pi=3
x=3
$ const.sh
const.sh: line 4: pi: readonly variable
const.sh: line 5: x: readonly variable
```

Don.VarEntier	Une variable entière doit être définie par `typeset -i`.
M=1;F=1;P=72;V=1	
Quelconque	

Description

Sans Objet

Justification

Les variables numériques sont souvent utilisées pour réaliser des calculs (indices de boucles, compteurs, etc.). Le fait de les définir comme variables entières permet d'éviter qu'elles ne soient mémorisées, par le shell, sous la forme d'une chaîne de caractères. On évite ainsi les conversions implicites chaîne/entier qui, sinon, seraient automatiquement exécutées, et on facilite l'utilisation de constructions de type `let`.
Cette fonctionnalité n'existe pas dans la norme POSIX.

Exemple

```
#!/bin/bash
#
# *** MAUVAIS EXEMPLE ***
#
cpt=0
while [ $cpt -lt 1000 ]
do
   # Instructions quelconques
   let cpt=cpt+1
done
```

Le calcul de l'incrément est exécuté par la fonction `let`, mais la variable est stockée, par le shell, sous la forme d'une chaîne de caractères. Il y a alors perte de temps due aux conversions constantes entre des chaînes et des entiers.

```
#!/bin/bash
#
# *** BON EXEMPLE ***
#
typeset -i cpt=0
while [ $cpt -lt 1000 ]
do
   # Instructions quelconques
   let cpt=cpt + 1
done
```

La variable `cpt` est stockée, par le shell, sous la forme d'un entier. Les conversions ne sont donc plus utiles. Le script y gagne en vitesse d'exécution.

7.5. TRAITEMENTS

Tr.ShellInterne	Lorsque le cas se présente, on doit préférer une fonction interne au shell plutôt qu'une commande externe.
M=0;F=2;P=66;V=1	
Quelconque	

Description

Sans Objet

Justification

Le lancement d'une commande externe est plus coûteux, en termes de performances, que l'appel d'une fonction interne.
Cette règle est particulièrement importante lorsque l'appel (de commande ou de fonction) est lié à une boucle.

Exemple

Pour lire le répertoire courant on écrira pwd (fonction interne) plutôt que /usr/bin/pwd (commande externe). A titre indicatif, voici la liste des fonctions internes (builtins) qui existent aussi sous forme de commandes UNIX (/usr/bin/*) :

command	built into
Alias	bash, ksh
Cd	bash, ksh, sh
Echo	bash, ksh, sh
Fc	bash, ksh
Getopts	bash, ksh, sh
Hash	bash, ksh, sh
Kill	bash, ksh, sh
Login	ksh, sh
Newgrp	ksh, sh
Pwd	bash, ksh, sh
Read	bash, ksh, sh
Test	bash, ksh, sh
Ulimit	bash, ksh, sh
Umask	bash, ksh, sh
Unalias	bash, ksh

Tr.RedefArguments	L'utilisation de la commande `set` pour redéfinir les arguments du programme est interdite.
M=2;F=0;P=68;V=1	
Quelconque	

Description

Sans Objet

Justification

La lisibilité du programme est fortement diminuée par ce style de programmation. Le lecteur peut ainsi perdre de vue quelles sont les valeurs des arguments d'appel du script, voire même ce que fait le script.

Exemple

```
#!/bin/bash
# set.sh
#
# *** MAUVAIS EXEMPLE ***
#
# Ce programme utilise extensivement set afin de
# modifier ses propres arguments, dans une boucle
# de traitement des arguments.

while [ $# -ne 0 ]
do
   case $1 in
      -[dD][iI]*)
         if [ -d $2 ]
         then
            repertoire=$2
            shift; shift
            set - bidon $* $repertoire/*
         fi
         ;;
      -*)
         echo "Option $1 inconnue, ignoree"
         ;;
      *)
         if [ -f $1 ]
         then
            fichiers="$fichiers $1"
         elif [ -d $1 ]
         then
            repertoire=$1
            shift
            set - bidon $* -DIR $repertoire
         fi
         ;;
   esac
   shift
done

echo FICHIERS TROUVES :
echo $fichiers
```

Le programme ci-dessus dresse la liste (éventuellement récursive) de tous les fichiers contenus dans une liste de répertoires qui lui est transmise en argument. La recherche se fait par une modification (permanente) de ses arguments.
Par contre l'exemple suivant est intéressant pour traiter les chaînes de caractères

```
#!/bin/bash
#
# Extraction d'une liste de paramètres
# En cas de séparateur quelconque, on utilise la commande tr.
```

```
P1=$1
P2=$2

echo Paramètre passé vaut \"$P1\"
set $P1
echo Les valeurs sont $*
echo

echo Paramètre passé vaut \"$P2\"
set `echo $P2 | tr ':' ' '`
echo Les valeurs sont $*

$ ./set2.sh "a b c d" "e:f:g"
Paramètre passé vaut "a b c d"
Les valeurs sont a b c d

Paramètre passé vaut "e:f:g"
Les valeurs sont e f g
```

Tr.Let	Les fonctions internes `let` (bash, ksh) ou `$(())` (bash, sh et POSIX shell) doivent être préférées à la commande `expr` pour les calculs arithmétiques.
M=0;F=1;P=78;V=1	
Quelconque	

Description

Sans Objet

Justification

Ces fonctions permettent de réaliser directement divers calculs arithmétiques sur des variables. Elles évitent de passer par un programme externe du genre de `expr`.
La commande `expr` doit être réservée aux fonctionnalités qui ne sont pas couvertes par `let` ou`((...))`. Cela concerne essentiellement les manipulations de chaînes de caractères.

Exemple

```
#!/bin/bash
# test1.sh
#
# *** BON EXEMPLE ***
#
# Ce programme réalise une boucle de 1000 itérations.
# L'incrémentation du compteur est faite par let.
cpt=0
while [[ $cpt -lt 1000 ]]
do
   let cpt=cpt+1
done
$ time test1.sh
real 0m0.168s
user 0m0.020s
sys  0m0.000s
```

Ce premier programme respecte la règle courante (ainsi que Tr.ShellInterne, pour le test de poursuite de l'itération).

```
$ cat t1
#!/bin/bash
# test2.sh
#
# *** MAUVAIS EXEMPLE ***
#
# Ce programme réalise une boucle de 1000 itérations.
# L'incrémentation du compteur est faite par expr.
#
cpt=0
while [[ $cpt -lt 1000 ]]
do
   cpt=`expr $cpt+1`
done
$ time test2.sh
real 0m2.762s
user 0m1.904s
sys  0m0.744s
$
```

La comparaison des temps d'exécution des deux programmes ne laisse aucun doute sur les performances que l'on peut obtenir ainsi. Comme pour la règle Tr.ShellInterne, tous les calculs sont réalisés par le shell dans `test1.sh` alors qu'ils passent par un programme externe pour `test2.sh`.
La déperdition de temps observées pour `test2.sh` provient non seulement du délai de lancement de chaque instance du programme `expr`, mais aussi de la lenteur de celui-ci, qui doit convertir les arguments qu'il reçoit (des chaînes de caractères) en entiers.

Tr.ControlArguments	Une fonction doit contrôler le nombre des arguments qu'elle reçoit.
M=2;F=2;P=42;V=1	
Quelconque	

Description

Sans Objet

Justification

Les scripts sont interprétés, et non compilés. Une erreur de transmission (nombre d'arguments insuffisants) ne sera relevée que lors d'une tentative d'accès à l'argument manquant. Si la fonction a déjà exécuté différentes actions, il peut être difficile de revenir en arrière.
Certains interpréteurs peuvent choisir de mettre fin immédiatement au script et renvoyer une erreur. Cela peut toutefois dépendre de la configuration locale définie par l'utilisateur (dans son fichier d'initialisation `.bashrc`, `.kshrc`, `.profile`, etc.).

Exemple

Toutes les fonctions devraient commencer par un test du style :

```
ma_fonction () {   # Trois arguments
   if [ $# -ne 3 ]
   then
      print 'Erreur grave dans ma_fonction'
      exit $code_erreur_nbargs
   fi
   ...
}
```

Tr.CommandInteractive	L'utilisation de commandes interactives est interdite dans un script.
M=1;F=1;P=73;V=1	
Quelconque	

Description

Sans Objet

Justification

Les commandes interactives (historique, alias, etc.) sont réservées à l'utilisation du shell par un humain, afin de lui faciliter son travail. Elles n'ont pas lieu d'être dans un script. Elles en obscurcissent la structure, voire même en faussent la compréhension.

Exemple

```
#!/bin/bash
# mon-script
#
fichier=t0
cp $fichier $fichier.old
```

Ce premier script réalise une simple copie d'un fichier dans un autre.

```
#!/bin/bash
# ma-config
alias cp=rm

#!/bin/bash
# mon-script
. ma-config
# ... Plus loin dans le fichier
fichier=truc
cp $fichier $fichier.old
```

L'exemple est artificiel, mais est loin d'être totalement impossible. Une personne ne lisant que le fichier `mon-script` fera une interprétation erronée du comportement du script, sauf si elle connaît parfaitement le contenu du fichier `ma-config`...

Tr.TestCodeErreur	Un programme appelant une autre commande ou un autre programme doit contrôler le code d'erreur renvoyé par celui-ci, quand cela se justifie au niveau applicatif.
M=2;F=3;P=22;V=1	

Quelconque	

Description

Sans Objet

Justification

Un script ne peut pas supposer, a priori, que les commandes qu'il lance se terminent toujours correctement. Il est très dangereux de ne pas réaliser ce type de tests, car cela peut introduire des dysfonctionnements importants dans un programme.

Exemple

```
#!/bin/bash
# test1.sh
#
# *** BON EXEMPLE ***
#
# Ce programme fait une sauvegarde d'un fichier avant de
# l'effacer. Il contrôle le code de fin de la copie
# avant l'effacement.
#
fichier=truc
cp $fichier $fichier.old
if [ $? -eq 0 ]
then rm $fichier
fi
$ ls -l
total 484
-rw------- 1 pyb  113826 Jul 24 11:44 truc
-r-------- 1 pyb   10553 Jul 24 11:44 truc.old
$ test1.sh
cp: cannot create truc.old: Permission denied
$ ls -l
total 484
-rw------- 1 pyb  113826 Jul 24 11:44 truc
-r-------- 1 pyb   10553 Jul 24 11:44 truc.old
```

Ce premier programme respecte la règle Tr.TestCodeErreur. Il contrôle le code de terminaison de la commande de copie et n'efface le fichier initial que si tout s'est bien passé. Dans l'exemple, la copie ne peut pas se faire, le fichier `truc.old` existant déjà et ne donnant pas de droits de modification. L'opération destructive n'est donc pas exécutée.

```
#!/bin/bash
# test2.sh
#
# *** MAUVAIS EXEMPLE ***
#
# Ce programme fait une sauvegarde d'un fichier avant de
# l'effacer. Il ne contrôle pas le code de fin de la
# copie avant l'effacement.

fichier=truc
cp $fichier $fichier.old
```

```
rm $fichier
$ ls -l
total 484
-rw------- 1 pyb  113826 Jul 24 11:44 truc
-r-------- 1 pyb   10553 Jul 24 11:44 truc.old
$ test2.sh
cp: cannot create truc.old: Permission denied
$ ls -l
total 244
-r-------- 1 pyb  staff     113826 Jul 24 11:44 truc.old
$
```

Ce second programme ne contrôlant pas la bonne fin de la copie, il ne relève pas l'erreur et exécute l'opération destructive.

Tr.OptionAnalyse	Un script doit afficher un message particulier lorsqu'il ne reconnaît pas une option. Ce message doit inclure le synopsis d'utilisation du script.
M=1;F=3;P=26;V=1	
Quelconque	

Description

Sans Objet

Justification

Les scripts sont généralement invoqués par des utilisateurs. Il est important de leur offrir un message explicite afin qu'ils puissent bien comprendre le fonctionnement du script, ses options et arguments, etc.

Exemple

Sans objet.

Tr.OptionUsage	Le projet peut définir l'option, commune à tous les scripts shell, qui provoque l'affichage d'une aide en ligne (rôle, arguments et options de chaque script.
M=1;F=2;P=52;V=1	
Quelconque	

Description

Sans Objet

Justification

Il n'est pas toujours facile de se souvenir de toutes les options possibles pour un script, surtout s'il existe de nombreux scripts pour un projet.
Une option commune, provoquant l'affichage des options et paramètres d'un script, permet à l'utilisateur de déterminer rapidement comment appeler le programme (voire même de vérifier qu'il s'agit bien du programme réalisant la fonction désirée).

Exemple

Tous les scripts devront proposer l'option `-usage`. Celle-ci affichera la liste des arguments, les options du script et un résumé d'une ou deux lignes du comportement du script.

Tr.GetOpts	Le traitement des arguments d'un programme doit être réalisé à l'aide d'un `while/case/esac` et d'un appel à `getopts`.
M=2;F=0;P=69;V=1	
Quelconque	

Description

Sans Objet

Justification

La maintenance des scripts est facilitée par des structures fonctionnelles communes.

Exemple

```
#!/bin/bash
#
# Exemple d'utilisation de getopts
#
# Options reconnues :
#    -a
#    -b nom
#    -c
#
while getopts "ab:c" option
do
   case $option in
      a) echo Option a;;
      b) echo Option b, argument ${OPTARG};;
      c) echo Option c;;
   esac
done

# On efface les options reconnues, pour ne conserver que
# les arguments supplementaires
let OPTIND=OPTIND-1
shift $OPTIND
echo $*
```

Tr.Global$0	Un script ne doit pas faire directement référence à la variable globale `$0`.
M=0;F=2;P=70;V=2	
Quelconque	

Description

Sans Objet

Justification

Selon les systèmes et interpréteurs, cette variable peut contenir soit le nom du programme, soit le chemin d'accès complet au programme. L'utilisation de cette variable, par exemple pour afficher un message d'erreur ou pour créer un fichier temporaire, peut ne pas produire les résultats escomptés.

Au besoin, il est préférable d'utiliser `basename $0`.

Exemple

Dans l'exemple ci-dessous, si la variable `$0` contient le chemin complet d'accès au programme `mon_script`, que nous supposerons dans le répertoire `/usr/projet/scripts`, la création du fichier temporaire échouera.

```
#!/bin/bash
# mon_script.sh
#
# *** MAUVAIS EXEMPLE ***
#
# Erreur possible à l'exécution : tentative de création
# du fichier /tmp/usr/projet/scripts/mon_script
touch /tmp/$0
```

Il faut alors utiliser une variable intermédiaire (si l'on souhaite exploiter souvent `$0`), ou appeler la commande `basename` :

```
#!/bin/bash
# mon_script.sh
#
# *** BON EXEMPLE ***
#
nom_prog=`basename $0`
touch /tmp/$nom_prog
```

Tr.IFS	La variable d'environnement `IFS` ne doit pas être modifiée.
M=1;F=3;P=27;V=2	
Quelconque	

Description

Sans Objet

Justification

La valeur de cette variable est utilisée, par le shell, pour réaliser diverses opérations qui lui sont vitales, notamment la séparation des 'mots' sur une ligne de commande.
Altérer cette variable, ne serait-ce que pour un temps très court, revient à l'altérer pour toutes les commandes qui seront appelées, directement ou non, durant cet intervalle de temps. Il est très facile de déclencher des erreurs subtiles, dans diverses configurations, du fait de cet héritage de `IFS` par tous les sous-processus.
Qui plus est, le lecteur du script peut avoir quelques difficultés à comprendre le fonctionnement de celui-ci, pour peu que la modification de `IFS` soit cachée (par exemple héritée du script appelant).

Exemple

Sans objet.

Tr.ExprRegulières	Les expressions régulières non triviales doivent être commentées.
M=3;F=1;P=29;V=1	
Quelconque	

Description

Sans Objet

Justification

La fonctionnalité exécutée par une expression régulière doit être rapidement compréhensible. Le lecteur ne doit pas devoir étudier soigneusement l'expression, voire l'exécuter à la main, pour en comprendre le rôle.

Exemple

```
# Extraction de l'extension d'un fichier
ext=`expr $fichier : '^.*\.\(.*\)$'`

# Liste de tous les fichiers .c dont le nom se termine
# par une suite de chiffres et une lettre minuscule.
liste=`find . -type f -name \*\[0-9\]+\[a-z\].c -print`

# Pour filtrer toute ligne contenant deux suites de
# chiffres séparées par un plus, la chaîne 'records,'
# puis soit 'in', soit 'out'.
dd_stats='^[0-9]+\+[0-9]+ records (in|out)$'
```

Tr.TestPipeLine	Une commande dont on veut tester le code de retour ne doit pas être insérée dans un pipe-line.
M=0;F=3;P=38;V=1	
Quelconque	

Description

Sans Objet

Justification

Il devient extrêmement difficile de récupérer l'erreur déclenchée, et de conserver l'éventuel diagnostic affiché par le programme en erreur.
Dans un tel cas, il faut utiliser des fichiers temporaires et tester directement le code de fin de la commande. Les fichiers temporaires devront être effacés pratiquement immédiatement après leur utilisation, pour respecter E/S-2.

Exemple

Voici un exemple des circonvolutions nécessaires pour récupérer le code d'erreur déclenché par le programme `dd`, tout en éliminant le message d'information qu'il affiche en fin d'exécution.

```
dd_stats='^[0-9]+\+[0-9]+ records (in|out)$'
res=`((dd if=/dev/rst0 ibs=63k 2>&1 1>&3 3>&- 4>&-;
       echo $? > &4) |
       egrep -v "$dd_stats" 1>&2 3>&- 4>&-) 4>&1`
```

Tr.FichierConfig	Chaque programme shell doit charger explicitement le ou les fichiers de configuration dont il a besoin.
M=2;F=2;P=44;V=1	
Quelconque	

Description

Sans Objet

Justification

Les scripts de premier niveau (lancés par l'utilisateur) ne peuvent pas supposer que celui-ci aura toujours chargé le ou les bons fichiers de configuration.

Exemple

Sans objet.

Tr.Continue	L'instruction `continue` ne doit être utilisée qu'à titre exceptionnel.
M=2;F=1;P=53;V=1	
Quelconque	

Description

Sans Objet

Justification

Les instructions `continue` posent le même type de problèmes que `break`.

Exemple

Sans objet.

Tr.TestsLogiques	Les abréviations de tests && et `||` sont interdites, sauf pour l'affichage de messages et la définition des valeurs par défaut des variables.
M=2;F=1;P=54;V=2	
Quelconque	

Description

Sans Objet

Justification

Ces structures diminuent la lisibilité d'un programme. Elles cachent l'existence de plusieurs branches d'exécution, et peuvent compliquer le plan de tests.

Exemple

Plutôt que

```
[ ! -s $fichier.c ] && cc -c $fichier.c && echo ok
```

Préférer une structure directement compréhensible, même si elle est plus longue :

```
if [ ! -s $fichier ]
then
   if cc -c $fichier.c
   then echo ok
   fi
fi
```

Par contre, on peut autoriser le type d'utilisation ci-dessous, qui allège le script :

```
# Définition de la valeur par défaut d'une variable
[ -z "$variable" ] || variable='valeur par défaut'
# Ou bien :
# variable=${variable:-'valeur par défaut'}

# Terminaison d'un programme en cas d'erreur
commande || ( echo 'Erreur d\'exécution'; exit 1 )
```

Tr.TestCrochets	La notation `[[...]]` doit être préférée à `test`.
M=1;F=1;P=74;V=2	
Quelconque	

Description

Sans Objet

Justification

La notation `[[ test_à_réaliser ]]` est plus lisible que son équivalent `test test_à_réaliser`. Il est à noter que, pour certains shells (POSIX notamment), la commande `test` est une fonction interne. Il n'y a donc pas de pénalité de performance à son utilisation.
La notation est `[ test à réaliser ]` avec le Bourne et le POSIX shells (utilisation de crochets simples, et non doubles).

Exemple

```
#!/bin/bash
# exemple1.sh
#
# *** BON EXEMPLE ***
#
# On utilise [[ ]] pour délimiter un test.
for fichier
do
   if [[ -w $fichier ]]
   then
      ... # Faire quelque chose
   fi
done
```

Cet exemple respecte la règle. Le test est clairement délimité, contrairement au programme ci-dessous :

```
#!/bin/bash
# exemple2.sh
#
# *** MAUVAIS EXEMPLE ***
#
# On utilise test.
for fichier
do
   if test -w $fichier
   then
      ... # Faire quelque chose
   fi
done
```

Tr.TestChaine	Les tests sur des chaînes de caractères doivent prendre en considération le cas spécial d'une chaîne vide.
M=3;F=3;P=20;V=1	
Quelconque	

Description

Sans Objet

Justification

Il s'agit d'un cas de test souvent oublié. Par malheur, il peut provoquer l'arrêt brutal d'un script qui, jusqu'ici, fonctionnait très bien.

Exemple

Le test ci-dessous n'est pas protégé contre cette erreur :

```
# *** MAUVAIS EXEMPLE ***
read info
if [[ $info -eq 'bonjour' ]]
     then echo 'bonsoir'
fi
```

Si l'utilisateur ne tape rien au clavier, la variable `info` sera vide. Dans ce cas, la commande de test sera appelée avec les arguments "`-eq 'bonjour'`", ce qui déclenchera une erreur de syntaxe et l'arrêt du script.
Il est préférable d'écrire :

```
# *** BON EXEMPLE ***
read info
if [[ "$info" -eq 'bonjour' ]]
     then echo 'bonsoir'
fi
```

Si la variable `info` est vide, l'expansion de la commande créera un argument vide `""` qui sera transmis au test.

Tr.UtilPOSIX	Les commandes appelées par un script doivent être de préférence prises dans la liste des utilitaires POSIX.
M=3;F=0;P=40;V=1	
Quelconque	

Description

La liste de ces utilitaires est la suivante :
awk, basename, bc, cat, cd, chgrp, chmod, chown, cksum, cmp, comm, command, cp, cut, date, dd, diff, dirname, echo, ed, env, expr, false, find, fold, getconf, getopts, grep, head, id, join, kill, ln, locale, localedef, logger, logname, lp, ls, mailx, mkdir, mkfifo, mv, nohup, od, paste, pathchk, pax, pr, printf, pwd, read, rm, rmdir, sed, sh, sleep, sort, stty, tail, tee, test, touch, tr, true, tty, umask, uname, uniq, wait, wc, xargs

Justification

Le comportement de ces utilitaires tend à se normaliser, du fait même de POSIX. Un tel choix permet donc de limiter les problèmes de portabilité, tant pour la présence des commandes appelées que pour leur comportement.

Exemple

Sans objet.

7.6. GESTION DES ERREURS

Err.ERRNOLINENO	Les messages d'aide à la mise au point doivent afficher le contenu de la variable `LINENO`.
M=1;F=3;P=30;V=1	
Quelconque	

Description

Sans Objet

Justification

La mise au point, et la correction des erreurs, en sont grandement facilitées. Lorsqu'un programme affiche une erreur à l'utilisateur, celui-ci dispose de plus d'informations à transmettre au support logistique. Le diagnostic de l'erreur est souvent plus rapide.

Exemple

Sans objet.

7.7. DYNAMIQUE

Dyn.DefinirComp	Le projet doit définir le comportement des scripts sur réception des signaux `hangup(HUP), interrupt(INT), quit(QUIT)` et `term(TERM)`.
M=2;F=2;P=45;V=0	
Quelconque	

Description

Sans Objet

Justification

Il est très facile d'expédier un signal à un script, de façon volontaire (par une commande de type `kill`) ou involontaire (en fermant accidentellement une fenêtre, en se déconnectant, par une erreur de frappe au clavier, etc.).
Dans de nombreux cas, cela provoque l'arrêt immédiat du script, et de ceux qu'il a lancés. Ce comportement n'est pas toujours acceptable.
Le comportement défini devrait notamment effacer tous les fichiers temporaires qui ne l'auraient pas déjà été.

Exemple

Tous les scripts devront inclure la commande suivante :

```
trap 'echo Signal ignoré' HUP INT QUIT TERM
```

Cette commande revient à ignorer les signaux hangup, interrupt, quit et term, tout en affichant un message pour l'utilisateur.

Dyn.Trap	Les signaux interceptés par `trap` doivent être définis par leur nom plutôt que par leur code numérique (bash, ksh et POSIX shells).
M=2;F=2;P=46;V=0	
Quelconque	

Description

Sans Objet

Justification

Il est plus facile de comprendre à quoi correspondent les signaux lorsqu'ils sont indiqués par leur nom.

Exemple

```
trap 'rm $copie' EXIT HUP INT QUIT TERM
```

Dyn.MaxShell	Le projet doit définir le nombre maximal de processus shell en tâche de fond qui peuvent exister simultanément.
M=1;F=3;P=31;V=1	
Quelconque	

Description

Sans Objet

Justification

Les performances globales de la machine peuvent être affectées par ce type de fonctionnement. Un processus shell, sauf s'il est très simple, n'est pas adapté à un fonctionnement en tâche de fond.
Les messages produits par le programme en arrière plan peuvent, suivant l'interpréteur interactif de l'utilisateur, venir perturber son écran, provoquer la suspension du programme, ou être purement et simplement perdus.

Exemple

Un script ne peut lancer plus d'une commande en tâche de fond.

Dyn.ProcFils	Un programme ne doit pas prendre fin avant ceux qu'il a lancés en tâches de fond.
M=3;F=3;P=21;V=0	
Quelconque	

Description

Sans Objet

Justification

Il y a risque de création de processus zombies (`<defunct>`). Des problèmes de pollution de la table des processus peuvent se manifester si un programme prend fin avant les fils qu'il a lancés. Ces problèmes peuvent aller jusqu'à imposer le redémarrage fréquent du système (il est parfois automatique, la machine s'arrêtant toute seule).
Un tel comportement (fin du père avant celle du fils) n'est autorisé que s'il est volontaire. C'est notamment le cas lorsque le processus fils est un démon (qui ne doit alors pas être écrit en shell, selon la règle SH.Restrictions).

Exemple

Voici deux programmes, `pere` et `fils`. Le programme `pere` appelle de façon répétitive `fils`, toujours en tâche de fond. Celui-ci réalise des actions diverses (simulées, pour l'exemple, par un `sleep` variable) puis prend fin.
La communication entre les processus fils et le processus père se fait par le biais d'un signal `SIGUSR1`.
Le processus père comptabilise le nombre de processus fils vivants qui restent, et ne prend fin que lorsque tous ses descendants sont terminés. `pere` a installé une fonction spécifique de traitement des signaux `USR1`.

```
#!/bin/bash
# pere.sh
#
# Ce programme lance un processus fils de façon
# répétitive, et comptabilise le nombre de fils actifs
# qui restent. Il ne prend fin que lorsque tous ses fils
# se sont terminés.
#
typeset -i cpt=0 nbfils=0

corrige() { # Pas d'argument
   # Redéfinition du traitement du signal, par sécurité
   trap corrige USR1
```

```
   # On diminue le nombre de fils restants
   let nbfils=nbfils-1
}

# Traitement du signal USR1
trap corrige USR1 || echo Probleme trap $?

while [ $cpt -lt 5 ]
do
   echo Lancement avec $cpt
   let nbfils=nbfils+1
   fils.sh $cpt &
   let cpt=cpt+1
done

# Attente que tous les fils soient terminés.
while [ $nbfils -ne 0 ]
do
   wait
   echo $nbfils processus restants
done
exit 0
```

Le programme `fils` envoie à son père un signal `SIGUSR1` juste avant de prendre fin.

```
#!/bin/bash
# fils.sh
#
# Ce programme envoie un signal particulier à son père
# afin de lui signaler sa propre fin.
#
typeset -i cpt
let cpt=$1*5
echo $$ -\> $cpt
sleep $cpt
echo $$ -\> termine

# On signale notre décès imminent
kill -USR1 $PPID
exit 0
```

7.8. INTERFACES

Int.LongPipeLine	Une taille maximale (en nombre de processus) doit être fixée pour définir la longueur d'un pipe-line.
M=2;F=1;P=56;V=1	
Quelconque	

Description

Sans Objet

Justification

Une ligne de commande enchaînant un trop grand nombre de programmes est difficilement compréhensible.
Sa mise au point peut s'avérer délicate, les erreurs déclenchées par l'un des composants du pipe-line n'étant pas conservées dans un quelconque fichier temporaire.

Exemple

Sans règle, il est possible d'écrire ceci :

```
tar tvf /dev/rst0
       awk -e '{print $3, $8 } | egrep '\.c$' |
   sort -nr | head -1
```

Cette commande ne contient que cinq composantes. A supposer que le format produit par `tar` ne soit pas celui attendu, les résultats obtenus n'auront rien à voir avec ceux espérés.
La mise au point nécessitera alors de découper le pipe-line en commandes indépendantes et d'utiliser des fichiers temporaires.

Int.Redirection	Toute redirection non standard des sorties d'une commande doit être commentée de façon très soigneuse.
M=2;F=1;P=57;V=1	
Quelconque	

Description

Sans Objet

Justification

Il est parfois très difficile de comprendre ce que l'on cherche à réaliser dans une telle situation. La présence d'un commentaire est alors indispensable pour le développeur et pour le futur mainteneur.

Exemple

```
dd_stats='^[0-9]+\+[0-9]+ records (in|out)$'
res=`((dd if=/dev/rst0 ibs=63k 2>&1 1>&3 3>&- 4>&-;
       echo $? > &4) |
       egrep -v "$dd_stats" 1>&2 3>&- 4>&-) 4>&1`
```

Sans commentaires, bien sûr.

7.9. QUALITE

QA.Path	On doit initialiser la variable `PATH` dans un script shell.
M=3;F=1;P=33;V=2	
Quelconque	

Description

Sans Objet

Justification

Un script n'a aucune garantie que le chemin de recherche de l'utilisateur est correct et contient les bons répertoires dont il a besoin. Il y a là un risque de sécurité (présence d'un cheval de Troie) et de fiabilité (échec possible du programme).
Cette initialisation de la variable `PATH` n'est nécessaire que pour les scripts de premier niveau, invoqués directement par l'utilisateur. Pour les scripts appelés par d'autres scripts, l'héritage du chemin de recherche correct qui a été défini par le script père est suffisant.

Exemple

```
#!/bin/bash
#
# Initialisation explicite du chemin
PATH=/usr/bin:/bin:/usr/local/bin
# Début réel du script...
```

QA.DefCommande	Les commandes et programmes appelés par le script doivent être définis par des variables ou des fonctions internes.
M=2;F=1;P=58;V=1	
Quelconque	

Description

Sans Objet

Justification

Les programmes (noms, chemins d'accès et options) sont très variables entre les systèmes. Il est fréquent qu'un programme ne se situe pas au même endroit de l'arborescence des fichiers, selon les machines et les systèmes. Parfois, une même option correspond à deux comportements différents. Plus le script est paramétré, plus son portage vers une nouvelle architecture/nouvelle version du système d'exploitation en sera facilité.
Les résultats des différentes commandes, même lorsqu'elles portent le même nom et utilisent les mêmes options (avec la même signification) peuvent différer.

Exemple

Voici un script fonctionnant sur SunOS, Solaris, HP-UX et Linux (bash u ksh). Il réalise la fonction très simple d'affichage de sa propre taille. Les options de `awk` sont différentes entre SunOS/Solaris et HP-UX. Le format d'affichage de `ls -l` n'est pas le même entre Solaris/HP-UX et SunOS. Le programme `cpp`

se trouve au même endroit sous SunOS et Solaris, mais pas sous HP-UX. Linux et HP-UX sont similaires sur ce point ;

```
#!/bin/bash
case `uname` in
   SunOS)
      cpp='/usr/ccs/lib/cpp'
      version=`uname -r | cut -d. -f1`
      if [ $version -eq 5 ]
      then
         echo Solaris 2
         lire_taille() {
            ls -l $1 | awk -e '{print $5}';
         }
      else
         echo SunOS
         lire_taille() {
            ls -l $1 | awk -e '{print $4}';
         }
      fi
      ;;
   HP-UX|Linux)
      cpp='/lib/cpp'
      lire_taille() {
         ls -l $1 | awk '{print $5}';
      }
      ;;
   *) echo Systeme `uname` non reconnu.
      exit 1;
      ;;
esac

lire_taille $0
```

QA.MiseAuPoint	Les options de mise au point `-n`, `-v` et `-x` doivent être utilisées durant le développement et la maintenance du programme.
M=3;F=1;P=34;V=2	
Quelconque	

Description

Sans Objet

Justification

Ces options, positionnées par la commande set, permettent d'obtenir diverses traces d'exécution du script, des diverses commandes appelées, des substitutions réalisées, etc.
Ce sont pratiquement les seuls outils de mise au point d'un script qui existent. Malgré leur faible niveau fonctionnel (ce ne sont que des outils de trace, et non de mise au point réelle comme le sont les débogueurs), il est dommage (et parfois dangereux) de les ignorer.

Exemple

Sans objet.

7.10. AUTRES REGLES

Il existe plusieurs interpréteurs possibles dans l'environnement UNIX. Les plus connus, et aussi les plus courants sur toutes les machines, sont :

Le Bourne shell, (commande `sh`), qui est l'ancêtre de tous les autres interpréteurs de commandes. Il est disponible sur toutes les machines.

Le Bourne Again SHell (commande `bash`) est la version du Bourne Shell développée par la FSF (Free Software Foundation) pour le projet GNU.

Le C-shell (commande `csh`), issu de la mouvance BSD. Il existe aujourd'hui sur presque toutes les machines, mais présente un langage de commande (structures de contrôle, gestion des variables, etc.) différent et incompatible avec le Bourne shell.

Le Korn shell, (commande `ksh`), qui reprend les mêmes structures de contrôle que le Bourne shell, en lui ajoutant des fonctionnalités interactives (gestion de l'historique des commandes tapées, etc.).

Le POSIX shell (commande `sh`), résultat de la normalisation POSIX 1003.2. Cet interpréteur est très proche de l'union du Bourne et du Korn shell. Pour plus de détails, voir le DR2.

Il faut remarquer que pour des raisons de compatibilité la commande `sh` peut parfois être un lien vers un autre shell (exemple : Sous Linux, la commande `sh` est un lien vers `bash`). Le nom du POSIX shell étant identique à celui du Bourne shell, nous avons utilisé la dénomination « POSIX shell » dans la suite du document alors que pour les autres shell nous avons utilisé les noms des commandes (`bash`, `ksh`, `sh`).

Il existe de nombreux autres interpréteurs, tant dans le domaine public que dans le domaine commercial (`tcsh`, `zsh`, etc.).

SH.BashPréconisé	L'interpréteur préconisé est le bash si celui-ci est disponible en standard.
M=3;F=1;P=35;V=2	
Quelconque	

Description

Sans Objet

Justification

Le shell `bash`, qui est la version FSF du Bourne shell est à présent « disponible » sur toutes les versions courantes d'UNIX (Solaris, HP/UX, Linux). Le fait d'utiliser `bash` permet d'assurer la *compatibilité* des scripts d'une version d'UNIX à une autre (cf: l'utilisation généralisée de GCC évoquée dans le document RNC-CNES-Q-80-522).
Si nous prenons le cas de Linux, `bash` est l'interpréteur de commande standard.
Si nous prenons l'exemple de Solaris, la version 10 inclut bash en standard (voir http://www.sun.com/software/solaris/soe_features.jsp « Integrated Open Sources Applications »). Pour des versions plus anciennes de Solaris (ou d'autres UNIX), il est aisé d'obtenir des paquetages binaires ou au pire de livrer les sources avec le projet. Le cas échéant on pourra utiliser `ksh` ou `sh`.
Le Bourne SHell est strictement compatible avec le bash. Ce dernier apporte un certains nombres d'extensions intéressantes (exemple: gestion avancées des variables, fonctions telles que `printf`, l'historique, etc.) qui pourraient devenir « standards » pour les développements CNES au fur et à mesure

que bash s'imposera (ce qui est en bonne voie aujourd'hui). A l'heure actuelle on se limitera la plupart du temps à la compatibilité avec le Bourne shell et le `ksh` voire le POSIX shell (SUSv3).
Concernant le « POSIX shell », il existe des implémentations livrées en standard dans les UNIX majeurs comme Solaris ou HP/UX. La conformité POSIX permet d'assurer l'objectif de portabilité des scripts développés même si ce n'est pas strictement le même interpréteur de commande. Coté `bash` on peut le faire fonctionner en mode POSIX en utilisant l'option `--posix`.
En résumé :

- Pour les systèmes sur lesquels `bash` est installé en standard (exemple: Linux) on l'utilise.
- Si `bash`, n'est pas disponible, on utilise le POSIX Shell si il existe.
- A défaut on utilise `ksh` ou `sh`.

Exemple

Sans objet.

SH.csh	Le shell csh n'est pas une alternative envisageable pour le développement de scripts.
M=0;F=0;P=79;V=0	
Quelconque	

Description

Sans Objet

Justification

Le shell csh présente un certain nombre de dysfonctionnements ou de comportements non-intuitifs. Qui plus est, sa syntaxe de commande n'est pas compatible avec celle des autres interpréteurs.
Se reporter au DR3 pour de plus amples détails.

Exemple

Sans objet.

SH.Limitations	Les scripts shell ne sont pas adaptés pour remplir des fonctionnalités de type temps réel, calcul intensif ou démon système, ni pour manipuler des données fragiles.
M=0;F=3;P=41;V=0	
Quelconque	

Description

Sans Objet

Justification

Les performances des scripts shell, qui sont interprétés, les rendent incompatibles avec les contraintes des fonctionnalités citées (la liste n'est pas exhaustive). D'une façon générale, un script shell ne constitue pas l'outil adapté à des traitements qui sont, par leur nature même, de longue durée.
De plus, la détection des erreurs de fonctionnement, et donc des incidents de traitement (au sens large du terme) est excessivement difficile dans un script shell. Dans le pire des cas, un programme C produit une image mémoire (core) qu'il est possible d'exploiter avec un outil de mise au point. Ce n'est pas le cas d'un

script shell. Détecter la cause de l'erreur peut alors nécessiter de relancer (souvent plusieurs fois) le script, en espérant que l'erreur se reproduira.
Par données fragiles, nous désignons les données dont une mise à jour partielle ou incorrecte peut provoquer des désastres fonctionnels ultérieurs.
Même en déroutant les signaux, il est très difficile, en shell, de garantir un retour en arrière correct (du style ROLLBACK sous Oracle). Si une telle nécessité fonctionnelle venait à se présenter, le programme devra être écrit avec un autre langage.

Exemple

Sans objet.

SH.Restrictions	Les scripts shell doivent être réservés à certaines fonctionnalités restreintes.
M=1;F=2;P=60;V=0	
Quelconque	

Description

Sans Objet

Justification

Un script shell ne doit en aucun cas se trouver au coeur d'une application, mais à sa périphérie. Il s'agit d'un outil annexe exploité pour faciliter différentes tâches simples et fortement liées à des manipulations de fichiers, des interactions brèves avec l'utilisateur, etc.
De façon typique, les scripts devraient être réservés aux procédures d'installation ou de mise à jour d'un système ou d'un projet.

Exemple

Sans objet.

SH.AutreIdentité	Il est interdit d'attribuer la propriété `SetUID` ou `SetGID` à un script shell.
M=2;F=3;P=23;V=2	
Quelconque	

Description

Sans Objet

Justification

Ces fonctionnalités permettent à un processus de s'exécuter sous une autre identité (ou sous un autre groupe) que ceux de l'utilisateur qui a appelé le programme. Dans le cas où le script shell appartient à root et qu'il possède la propriété Setuid ou Setguid, lors de son exécution le processus résultant endosse les privilèges de root. Ce comportement constitue une brèche potentielle exploitable par les crackers d'autant plus si le script est lisible. Une analyse soignée du code peut permettre de relever des possibilités de détournement du programme, afin de lui faire exécuter des commandes initialement non prévues.
Les règles du DR4 interdisent de tels scripts sur un serveur agréé. Si le projet décide tout de même d'outrepasser ces règles, il risque de se trouver bloqué, à terme lors de son agrémentation.

Il faut noter que l'utilisation de Setuid/Setgid sur les scripts shell n'est *pas possible* sous Linux.

Exemple

```
#
# mon-shell
#
# Ce script est SUID
#
repertoire=/usr/oracle/bdd
touch $repertoire/la_base_de_donnees_vitale
```

Ce script met à jour la date de dernière modification du fichier `la_base_de_donnees`_vitale situé dans le répertoire `/usr/oracle/bdd`. Une telle opération ne peut être exécutée que par le propriétaire du fichier. La base appartenant à l'utilisateur `oracle`, le script `mon-shell` est **`SUID`** et appartient à l'utilisateur `oracle`.
Il appelle la commande `touch`, qui est recherchée dans le chemin de recherche (`PATH`) de l'utilisateur. Un individu malicieux peut alors faire ce qui suit :

```
PATH=mon_repertoire:$PATH
export PATH
cd mon_repertoire
echo 'rm $*' > touch
chmod a+x touch
mon-shell
```

Le script mon-shell trouvera la commande touch située dans le répertoire mon_repertoire, exécutera celle-ci, et détruira la_base_de_donnees_vitale en toute tranquillité.

SH.SpecifiquesBash	On ne doit pas utiliser les fonctions spécifiques `bash`.
M=3;F=1;P=37;V=2	
Quelconque	

Description

Sans Objet

Justification

Le shell `bash` ajoute de nombreuses extensions facilitant grandement la programmation des scripts shell. D'après la règle SH.Restrictions, il est recommandé de limiter l'utilisation des scripts à des fonctions auxiliaires simples, ce qui fait qu'il n'est normalement pas nécessaire d'envisager des extensions complexes. Cependant, dans certains cas ou une extension facilite grandement le travail, on pourra l'envisager.

Exemple

On peut utiliser la fonction `printf` de `bash` pour des besoins particuliers :

```
$ X=2
$ printf "La valeur est %02d\n", $X
```

```
La valeur est 02
```

8. AUTRES ASPECTS SPECIFIQUES AU LANGAGE

Sans Objet

9. SYNTHESE

9.1. TABLE RECAPITULATIVE DES REGLES

Les règles sont récapitulées ici, classées par ordre alphabétique.

Id. Règle	Intitulé	Page
Don.Constante	Une constante doit être définie comme telle par le mot-clé `readonly`.	15
Don.VarEntier	Une variable entière doit être définie par `typeset -i`.	17
Dyn.DefinirComp	Le projet doit définir le comportement des scripts sur réception des signaux `hangup(HUP)`, `interrupt(INT)`, `quit(QUIT)` et `term(TERM)`.	32
Dyn.MaxShell	Le projet doit définir le nombre maximal de processus shell en tâche de fond qui peuvent exister simultanément.	32
Dyn.ProcFils	Un programme ne doit pas prendre fin avant ceux qu'il a lancés en tâches de fond.	33
Dyn.Trap	Les signaux interceptés par `trap` doivent être définis par leur nom plutôt que par leur code numérique (bash, ksh et POSIX shells).	32
Err.ERRNOLINENO	Les messages d'aide à la mise au point doivent afficher le contenu de la variable `LINENO`.	31
Id.MotsClés	L'utilisation des mots-clés de l'interpréteur est interdite.	14
Id.Options	Les options des scripts doivent être si possible compréhensibles et différenciées.	15
Int.LongPipeLine	Une taille maximale (en nombre de processus) doit être fixée pour définir la longueur d'un pipe-line.	35
Int.Redirection	Toute redirection non standard des sorties d'une commande doit être commentée de façon très soigneuse.	35
Org.Config	Les initialisations nécessaires à l'ensemble des programmes shell d'un projet doivent être regroupées dans un fichier spécifique.	10
Org.FonctionPrivées	Un script ne doit pas exporter les fonctions qu'il définit.	10
Org.Héritage	Les scripts shell doivent prendre soin de ne pas hériter des alias ou des fonctions définis par l'utilisateur.	10
Org.LimitAwk	Le projet doit fixer une taille maximale, en nombre de règles et en nombre de lignes par règle, pour toute utilisation de `awk`.	13
Org.LimitScripts	Le projet doit fixer une taille maximale des scripts inclus dans des Makefiles.	12
Org.LimitSed	Le projet doit fixer les limites d'utilisation de `sed`.	14
Pr.Interpréteur	La première ligne d'un script doit préciser l'interpréteur de commandes à utiliser. Elle est de la forme : `#!/<chemin_interpréteur_à_utiliser>.`	14
QA.DefCommande	Les commandes et programmes appelés par le script doivent être définis par des variables ou des fonctions internes.	36
QA.MiseAuPoint	Les options de mise au point `-n`, `-v` et `-x` doivent être utilisées durant le développement et la maintenance du programme.	37
QA.Path	On doit initialiser la variable `PATH` dans un script shell.	36
SH.AutreIdentité	Il est interdit d'attribuer la propriété `SetUID` ou `SetGID` à un script shell.	40
SH.BashPréconisé	L'interpréteur préconisé est le bash si celui-ci est disponible en standard.	38
SH.csh	Le shell csh n'est pas une alternative envisageable pour le développement de scripts.	39
SH.Limitations	Les scripts shell ne sont pas adaptés pour remplir des fonctionnalités de type temps réel, calcul intensif ou démon système, ni pour manipuler des données fragiles.	39
SH.Restrictions	Les scripts shell doivent être réservés à certaines fonctionnalités restreintes.	40

9.2. TRAÇABILITE VIS A VIS DU DOCUMENT « COMMUN »

Cette table a été créée lors de la mise en commun des règles entre langages. Elle donne pour chaque règle :

- ? L'ancienne identification de la règle (Version **3**)
- ? La nature de la modification : Aucune, Renommée (on a changé l'identification de la règle), Communalisé (la règle a été déplacée vers les document commun), Supprimée.
- ? La nouvelle identification dans le cas où la règle n'a pas été supprimée, une raison de la suppression sinon.

Identification Version **3**	Nature	Nouvelle identification
ARGS-1	Renommée	Tr.OptionAnalyse
ARGS-2	Renommée	Tr.OptionUsage
ARGS-3	Commonalisée	Tr.ProgDefensive
ARGS-4	Renommée	Id.Options
ARGS-5	Renommée	Tr.GetOpts
CHOIX-1	Renommée	SH.KornPréconisé
CHOIX-2	Renommée	SH.CShell
CHOIX-3	Renommée	SH.Limitations
CHOIX-4	Renommée	SH.Restrictions
CHOIX-5	Renommée	SH.AutreIdentité
CMD-1	Renommée	Dyn.MaxShell
CMD-2	Renommée	Dyn.ProcFils
COMT-1	Commonalisée	Pr.CartStd
COMT-2	Commonalisée	Pr.CartStd
COMT-3	Renommée	Pr.Interpréteur
COMT-4	Commonalisée	Tr.ModifVarGlobal
CONF-1	Renommée	Org.Config
CONF-2	Renommée	Tr.FichierConfig
CONF-3	Renommée	Org.Héritage
CTRL-1	Commonalisée	Tr.OrdreChoix
CTRL-2	Commonalisée	Tr.ModifCompteur
CTRL-3	Commonalisée	Tr.BoucleSortie
CTRL-4	Renommée	Tr.Continue
CTRL-5	Renommée	Tr.TestsLogiques
CTRL-6	Renommée	Tr.TestCrochets
CTRL-7	Renommée	Tr.TestChaine
CTRL-8	Commonalisée	Tr.ProgDefensive
DEBUG-1	Renommée	QA.MiseAuPoint
DEBUG-2	Renommée	Err.ERRNOLINENO
E/S-1	Commonalisée	Int.Temporaire
E/S-2	Commonalisée	Int.Temporaire
E/S-3	Renommée	Int.LongPipeLine
E/S-4	Renommée	Int.Redirection
E/S-5	Commonalisée	Err.Canal
E/S-6	Renommée	Tr.TestPipeLine
ENV-1	Commonalisée	Don.Localite
ENV-2	Renommée	QA.Path
ENV-3	Commonalisée	Int.CheminAbsolu
ENV-4	Commonalisée	Int.Environement

Identification Version **3**	Nature	Nouvelle identification
ENV-5	Commonalisée	Don.Initialisation
ENV-6	Renommée	Tr.IFS
ERR-1	Commonalisée	Err.Mecanisme
ERR-2	Commonalisée	Err.Mecanisme
ERR-3	Commonalisée	Err.Mecanisme
ERR-4	Renommée	Tr.TestCodeErreur
EXPR-1	Renommée	Tr.ExprRegulières
FCT-1	Renommée	Tr.ShellInterne
FCT-2	Renommée	Tr.RedefArguments
FCT-3	Renommée	Tr.Let
FCT-4	Renommée	Tr.ControlArguments
FCT-5	Renommée	Tr.CommandInteractive
FCT-6	Renommée	Id.MotsClés
FCT-7	Renommée	Org.FonctionPrivées
FICHIER-1	Commonalisée	Org.ModuleNom
FICHIER-2	Supprimée	Métriques
FICHIER-3	Commonalisée	Id.Procedure;Id.Fonction
FICHIER-4	Supprimée	Métriques
FICHIER-5	Commonalisée	Org.DonneesOper
FICHIER-6	Commonalisée	Pr.Identation
PORT-1	Commonalisée	Qa.MatérielIndep
PORT-2	Renommée	QA.DefCommande
SIG-1	Renommée	Dyn.DefinirComp
SIG-2	Renommée	Dyn.Trap
UTIL-1	Renommée	Tr.UtilPOSIX
UTIL-2	Renommée	Org.LimitScripts
UTIL-3	Renommée	Org.LimitAwk
UTIL-4	Renommée	Org.LimitSed
VAR-1	Renommée	Tr.Global$0
VAR-2	Commonalisée	Id.VarSignif
VAR-3	Renommée	Don.Constante
VAR-4	Renommée	Don.VarEntier
VAR-5	Commonalisée	Don.Localite

REFERENTIEL NORMATIF REALISE PAR :

Centre National d'Etudes Spatiales

Inspection Générale Direction de la Fonction Qualité

18 Avenue Edouard Belin

31401 TOULOUSE CEDEX 9

Tél. : 05 61 27 31 31 - Fax : 05 61 28 28 49

CENTRE NATIONAL D'ETUDES SPATIALES

Siège social : 2 pl. Maurice Quentin 75039 Paris cedex 01 / Tel. (33) 01 44 76 75 00 / Fax : 01 44 46 76 76
RCS Paris B 775 665 912 / Siret : 775 665 912 00082 / Code APE 731Z