



## Copyright AFUU 1993

*Tous droits réservés. Toute reproduction, même partielle, par quelque procédé que ce soit, est interdite sans autorisation préalable. Une copie par xérogaphie, photographie, film, bande magnétique ou autre, constitue une contre-façon passible de peines prévues par la loi sur la protection des droits d'auteurs.*

ISBN : 2-907902-25-3

*AFUU : Association Française des Utilisateurs d'Unix et des systèmes ouverts.*

*11, Rue Carnot*

*94270 Le Kremlin-Bicêtre*

*Té. (+33)(1) 46 70 95 90*

*Fax (+33)(1) 46 58 94 20*

*Télex : 263 887 F*

*E-mail : [secretariat@afuu.fr](mailto:secretariat@afuu.fr)*

---

## Rédaction

- Jean-Michel Cornu Cornu consultant
- Francis Ducroux Renault
- Noë Nader CISI TM
- Marc Pointot Sextant avionique
- François Riche IBM France

- Roberto Rotter Commission européenne DG XII
- Hervé Schauer Hervé Schauer consultants
- Hervé Sortais SLIGOS
- Catherine Tournier-Lasserve SUN Microsystems
- Pierre Viallefont Centre National d'Etudes Spaciales

Groupe de Portabilité de l'AFUU

---

## Avant-propos

L'univers des systèmes ouverts est le théâtre d'un foisonnement de produits couvrant une même fonction. Cette richesse permet à l'utilisateur de le tenir mieux adapté à son besoin, à sa pointure, avec un rapport prestation/prix maîtrisé. Si la variété des composants d'un système d'informations, sur le plan applicatif, le satisfait, la concurrence entre composants du système de base le perturbe.

Le conducteur automobile apprécierait-il un poste de conduite différent, suivant que son véhicule soit mû par un moteur à gasoil ou à essence ? Accepterait-il une signalisation routière dont les conventions changent tous les kilomètres ? Assurément non. Il en est de même du langage de commande du système d'exploitation. La multitude de *shell* apparus sous Unix (Système d'exploitation prototype des systèmes ouverts), met l'utilisateur dans l'embarras. Lequel choisir ? Lequel est le meilleur ? Lequel survivra ? Autant de questions qui invitent notre utilisateur à prier les éditeurs : "Messieurs, de grâce accordez-vous !". Dans cette situation, la standardisation est la bienvenue. La normalisation, par nature consensuelle, a fait son oeuvre, et s'est penchée sur le destin d'un langage de commande unifié.

L'ouvrage "Unix et systèmes ouverts. De la portabilité au portage" (issu des travaux du groupe PORTABILITE de l'AFUU) situe les domaines où la portabilité peut s'avérer problématique et fait le point sur les normes et standards en vigueur. De la même façon, ce présent guide examine en détail les différents *shell*, ainsi que la norme, autant pour aider l'indécis à choisir, que celui qui a tout essayé et qui souhaite mesurer l'envergure d'un passage à la norme.

Ainsi le guide "*shell* et portabilité" se veut pratique, et outil d'aide à la compréhension de la norme. Il s'adresse au décideur qui désire s'y reconnaître, aux développeurs ou administrateurs qui sont confrontés à la portabilité de leurs produits basés sur un ou plusieurs *shell*.

La norme ISO/IEC 9945-2 ayant regroupé le langage de commande et les utilitaires, ce guide est divisé en deux volumes séparés, le premier portant sur le langage de commande *shell*, et le second (à paraître) portant sur les utilitaires.

---

## 1- Introduction

Contenu de ce chapitre

Quand on parle du *shell* des systèmes ouverts, il s'agit du langage de commande.

Depuis le premier interpréteur de commandes apparu dans Unix, le *Bourne-shell* (sh), différentes évolutions virent le jour avec pour justification, un enrichissement fonctionnel. Ainsi le *C-shell* (csh) issu des améliorations apportées par Berkeley fut suivi par de nombreux *shell* dont le *Korn-shell* (ksh) chez AT&T, ou le *Bourne Again Shell* (*bash*) de GNU.

Les fournisseurs ne prenant pas parti, offrent souvent plusieurs d'entre eux, laissant le choix à l'utilisateur, c'est ainsi qu'ils peuvent être alternativement utilisés en diverses circonstances, sachant que leurs différences soulèvent des problèmes de compatibilité.

S'ils sont très proches dans l'usage interactif, les incompréhensions mutuelles apparaissent par contre dans l'interprétation des fichiers de commandes (*shell-script*) écrites pour l'un et exécutées par un autre.

Les travaux entrepris par le groupe IEEE "POSIX.2" : "*Shell and Utilities*" visent à standardiser le langage de commande (*shell*) en préservant un maximum de compatibilité avec l'existant. Toutes les commandes pouvant être invoquées à l'aide du *shell*, comme les utilitaires, sont couvertes par cet effort de standardisation. En effet, à quoi bon uniformiser le langage d'appel des utilitaires, si ceux-ci ne sont pas standards ? Le terme "utilitaires" distinct du terme "commandes", désigne les utilitaires habituellement fournis avec le système d'exploitation, les fichiers de commandes ou même des programmes applicatifs de l'utilisateur. Leur usage implique le chargement d'un nouveau processus, par opposition aux commandes internes au *shell* (*built-in*) exécutées par le processus *shell* lui-même.

## 1.1- Le *shell* interactif

L'usage courant du langage de commande est celui dit "interactif", car l'interpréteur assure le contrôle, les conditions d'exécution et l'exécution des commandes frappées par un opérateur à partir d'un terminal. Lorsque l'exécution d'une commande est terminée, le *shell* est prêt à prendre en compte l'éventuelle commande suivante de l'opérateur.

Le comportement interactif du *shell* est couvert par une extension à la norme, issue du projet P1003.2a encore appelée UPE (*User Portable Environment*), indépendamment d'une interface utilisateur, la présentation des réponses étant limitée au mode ligne sur un terminal. Les réponses données par le *shell* au cours de ce dialogue sont en particulier adaptées par l'observation des particularités nationales. Un "code retour" égal à zéro constitue la marque d'une exécution terminée normalement, et d'une valeur numérique positive, la trace d'une erreur.

Les différents types de *shell* admettent tous, et de la même façon, les redirections avec le même signe (>, >>, <, <<) ainsi que l'enchaînement par tubes (*pipe*), mais certains possèdent des fonctions que d'autres n'ont pas. Le *C-shell* et le *Korn-shell* enregistrent un historique des commandes et supportent les alias; ce n'est pas le cas du *Bourne-shell*. De même au démarrage d'une session utilisateur (*login*), le contexte de départ pour l'interpréteur de commande est activé à partir du contenu d'un ou plusieurs fichiers de configuration se trouvant dans le répertoire privé de l'utilisateur, consacré(s) à cet effet. Ainsi le *C-shell* exploite, s'ils existent dans ledit répertoire privé de l'utilisateur, les fichiers ".cshrc" et

".login" alors que le *Bourne-shell* ne connaît que le fichier ".profile". Le *Korn-shell* par contre, utilise les fichiers ".profile" et ".env" ou ".kshrc".

## 1.2- Programmation en *shell*

Le *shell* n'est pas qu'un langage de commande interactif, c'est aussi un langage de programmation complet. L'interprétation des *shell-script* est liée au type de *shell*. Si ces procédures (fichiers contenant des commandes) ne font références qu'à des commandes externes (avec leurs options) comme les utilitaires, alors les différents *shell* peuvent être compatibles. Mais si les procédures en question contiennent des structures de contrôle (traitement conditionnel, répétitif, fonctions, ...) alors la compatibilité disparaît car les syntaxes sont différentes.

### Exemple :

#### **sh :**

```
if [ commande ]
then
instructions
else
instructions
fi
```

#### **csH :**

```
if ( expression ) then
instructions
else
instructions
endif
```

La programmation de fonctions (déclarations internes) est possible en *Bourne-shell* et en *Korn-shell*, alors qu'elle n'existe pas en *C-shell*. Là aussi les syntaxes diffèrent :

#### **sh :**

```
ma_fonction()
{
instructions
.....
}
```

## **ksh :**

```
function ma_ fonction ()  
{  
  instructions  
  .....  
}
```

Toutefois pour préserver la compatibilité avec le *Bourne-shell*, le *Korn-shell* supporte les deux écritures ci-dessus.

Pour une même fonction interne (*built-in*) existant dans différents *shell*, le résultat peut avoir des présentations différentes (`times` par exemple pour *Bourne-shell* et *Korn-shell*).

Les mécanismes de substitution diffèrent entre les *shell*. Le *Korn-shell* permet des opérations arithmétiques, alors qu'avec les autres il faut avoir recours à la commande externe `expr`.

L'adoption d'un *shell* (comme le *C-shell* par exemple), n'impose pas son utilisation pour l'écriture des *script*. En effet, rappelons qu'il est possible de préciser dès le début du fichier contenant le *script*, avec quel *shell* doivent être interprétées les commandes qui suivent.

La nature du langage utilisé étant présente en première ligne sous la forme d'un pseudo commentaire comme dans les exemples suivants :

```
/cat script.csh  
#!/bin/csh -f  
echo "Ce script est en C-shell"  
%cat script.Psh  
#!/bin/sh  
echo "Ce script est en Shell standard"
```

On remarque que l'expression de ce commentaire est identique pour les différents *shell*, le caractère "dièse" en début de ligne, signalant le commentaire, est immédiatement suivi d'un point d'exclamation. Lorsque la séquence "#!" se trouve en première ligne (au début du fichier), il ne s'agit alors plus d'un commentaire, et ce qui suit est compris comme une commande (décrite avec son chemin d'accès) à exécuter :

```
%cat prog_awk.cmd  
#!/usr/bin/awk -f
```

## **1.3- Disponibilité**

Une license AT&T permet de disposer du *Korn-shell*. La plupart des systèmes Unix basés sur System V

(et depuis la version V.3) possèdent *Korn-shell*. C'est le cas d'AMDAHL, AT&T, CDC, CONVEX, CRAY, DEC, HP, IBM, NCR, OLIVETTI, PRIME, SCO, SUN (Solaris), UNISYS, pour ne citer que les principaux. Cependant, certains *shell*, tels que *bash* et *zsh*, compatibles POSIX.2 sont également disponibles dans le domaine public.

## 1.4- Avantage de la norme

Afin de s'affranchir des différences entrevues entre les différents *shell* jusque là disponibles, la standardisation envisagée dans un premier temps par X/Open, a consisté à ne retenir que le *Bourne-shell*, et figer les utilitaires retenus avec leurs principales options (Niveau XPG3).

IEEE"POSIX.2"ou sa contrepartie internationale ISO/IEC9945-2:1993 va plus loin en reconnaissant les améliorations apportées par le *Korn-shell*, compatible en grande partie avec le *Bourne-shell*, mais riche de fonctions nouvelles, dont certaines reprises au *C-shell* sans sa syntaxe particulière. Ainsi le *shell* normalisé (POSIX) garantira la portabilité des commandes. De plus il s'appuie sur des fonctions du système déjà normalisées par IEEE POSIX.1 (ISO/IEC 9945-1:1990), donc assure une cohérence fonctionnelle et comportementale. Le meilleur moyen d'aller aujourd'hui vers la conformité POSIX, est de suivre le guide ...

---

## 2- Définition des listes synoptiques

Contenu de ce chapitre

### 2.1- Utilité de ces listes

Dans les annexes, sont présentées quatre listes synoptiques. La première correspond à un index de portabilité, les autres correspondent à des index de partage pour les trois principaux langages de commande, respectivement le *Bourne-shell*, le *C-shell* et le *Korn-shell*.

La liste de portabilité est conçue pour aider les développeurs de fichiers de commandes (*shell-script*) à écrire leurs nouvelles applications avec un maximum de conseils afin que ces fichiers de commandes soient portables.

Les listes de partage sont faites pour vérifier si des applications existantes ont un bon niveau de portabilité, et pour aider au partage de celles-ci.

Le chapitre 4 regroupe les commentaires des deux types de listes.

### 2.2- Description de la liste de portabilité

#### 2.2.1- Description générale

Nous avons plusieurs listes de portabilité :

- fichiers requis
- commandes internes (*built-in*)
- structures de contrôle
- variables et paramètres
- variables d'environnement
- redirections

Chacune de ces listes est composée de plusieurs points d'entrée ou index de portabilité ("Mots-Clés") détaillé en 2.2.2 . Chaque entrée de la liste comporte cinq éléments :

Les trois premiers ("sh", "csh", "ksh") précisent le degré de portabilité suivant les critères détaillé en 2.2.3.

L'avant dernier ("Expliqué en" ou "détaillé en") contient le renvoi au paragraphe du guide expliquant le problème de portabilité rencontré.

Le dernier ("Voir ISO 9945-2") contient systématiquement le renvoi au paragraphe de la norme ISO 9945-2, permettant ainsi de retrouver tous les détails de la normalisation du *shell*.

On pourra tirer parti de la correspondance entre le chapitre 3 de la norme et le chapitre 2 du guide de portabilité XPG4. Ainsi les différents paragraphes à l'intérieur de ces chapitres se trouvent dans le même ordre, et à défaut de l'un on pourra se reporter à l'autre document pour cette partie.

## 2.2.2- Index de portabilité

Le point d'entrée de cette liste est défini comme étant une entité de portabilité. Par exemple, la commande `export` qui permet de qualifier des variables pour héritage, aura plusieurs points d'entrée suivant ses différents usages comme :

```
export
```

```
export nom_de_variable
```

```
export nom_de_variable=valeur
```

```
export -p
```

Cette liste se veut exhaustive vis-à-vis de la portabilité, c'est-à-dire que l'index a autant d'entrées que de mots-clé du langage *shell*.

## 2.2.3- Les degrés de portabilité

Les symboles suivants classés dans un ordre de gravité croissant, qualifient le mot-clé défini dans la norme dans le synoptique de portabilité, par rapport au *shell* en regard :

**C** : Compatible : fonctionnalité normée de syntaxe et comportement identique ou compatible avec ce *shell*

**CA** : Compatibilité Ascendante : nouvelle fonctionnalité ou extension apportée par la norme sur une commande existant déjà dans ce *shell*

**N/A** : Non applicable : fonctionnalité normée qui n'a pas d'équivalent dans ce *shell*

**NS** : Nouvelle syntaxe : fonctionnalité normée ayant une nouvelle syntaxe pour la fonctionnalité existante de ce *shell* (option différente)

**O** : Obsolète : fonctionnalité normée par une autre syntaxe rendant l'usage équivalent dans ce *shell* (usage avant la norme comportement non défini)

**AS** : Autre syntaxe : fonctionnalité normée qui a une équivalence dans ce *shell*, mais exprimée avec une syntaxe différente (mot-clé différent)

L'usage de "`export name`" ne pose pas de problème de portabilité en *Bourne-shell* et en *Korn-shell*. En effet, la syntaxe et la sémantique sont rigoureusement les mêmes. Ceci est signalé par le symbole C pour compatibilité.

Le *Korn-shell* a introduit l'usage "`export nom=mot`" qui est repris dans la norme. Cet usage ne fonctionne pas en *Bourne-shell*. Ceci est signalé par le symbole CA pour compatibilité ascendante.

Le *Korn-shell* a introduit l'usage "`export`" tout seul pour connaître la liste des variables ainsi que leur valeur, une variable par ligne. La norme reprend cette fonctionnalité par l'usage "`export -p`" qui permet de rejouer la sortie de cette commande.

Donc l'usage "`export`" est signalé par le symbole O obsoléscent c'est-à-dire remplacé par un autre.

La norme précise bien, dans le *rationale*, que le résultat de cet usage est non défini.

Pour le *Bourne-shell*, l'usage "`export`" est signalé par le symbole N/A pour non-applicable même s'il ne génère pas d'erreur ni de sortie.

Pour le *Korn-shell*, l'usage "`export -p`" est signalé par le symbole NS pour nouvelle syntaxe d'une fonctionnalité existante.

Pour le *C-shell*, l'usage "`setenv`" permet d'exporter les variables. Ceci est signalé par le symbole AS pour autre syntaxe.

#### **2.2.4- Usage des listes de portabilité**

Cette liste permet de savoir si une fonctionnalité pose des problèmes de portabilité. Si c'est le cas, un commentaire de quelques lignes indique le risque encouru. Le lecteur peut, s'il souhaite en savoir plus,

se reporter à la norme, grâce au numéro de paragraphe du dernier point de la liste.

## 2.3- Description des liste de portage

### 2.3.1- Description générale

Nous avons trois listes de portage :

- du *C-shell* à la norme (nommé csh)
- du *Korn-shell* à la norme (nommé ksh)
- du *Bourne-shell* à la norme (nommé sh)

Chacune de ces listes se compose de plusieurs points d'entrée ou index de portage ("Mots- Clés"). Contrairement à la liste de portabilité, **ne sont présentes que les entités nécessitant une traduction et/ou un effort de portage**. Chaque entrée de la liste comprend deux éléments :

Le premier ("Equivalent *shell* normalisé") comprend l'équivalent normalisé s'il existe, pour aider à la traduction.

Le deuxième ("Expliqué en") peut renvoyer à un commentaire dans le corps du guide.

### 2.3.2- Usage des listes de portage

Ces listes permettent aux administrateurs système comme aux développeurs de vérifier si les fichiers de commandes qu'ils utilisent, nécessitent un portage. Ils permettent de développer des outils de vérification pour extraire les points délicats d'un *script*.

---

## 3- Listes synoptiques

Contenu de ce chapitre

Les degrés de portabilité

### 3.1- Portabilité

#### 3.1.1- Fichiers requis

1. Mot-clé : /

sh : C

csh : C

ksh : C

détaillé en 4.3.1

Voir ISO 9945-2 : 2.7

2. Mot-clé : /dev/null

sh : C

csh : C

ksh : C

détaillé en 4.3.1

Voir ISO 9945-2 : 2.7

3. Mot-clé : /dev/tty

sh : C

csh : C

ksh : C

détaillé en 4.3.1

Voir ISO 9945-2 : 2.7

4. Mot-clé : /tmp

sh : C

csh : C

ksh : C

détaillé en 4.3.1

Voir ISO 9945-2 : 2.7

### **3.1.2- Fonctions internes (*Built-in*)**

1. Mot-clé : ?

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.13.1

2. Mot-clé : \*

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.13.1

3. Mot-clé : [

sh : C

csch : C

ksh : C

détaillé en 4.2.2

Voir ISO 9945-2 : 3.13.1

4. Mot-clé : break

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.1

5. Mot-clé : break *n*

sh : C

csch : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.13.1

6. Mot-clé : :

sh : C

csch : N/A

ksh : C

détaillé en 4.3.6.1

Voir ISO 9945-2 : 3.14.2

7. Mot-clé : continue

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.3

8. Mot-clé : *continue n*

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.3

9. Mot-clé : *. fichier*

sh : CA

csh : AS

ksh : C

détaillé en 4.3.5.1

Voir ISO 9945-2 : 3.14.4

10. Mot-clé : *eval*

sh : C

csh : C

ksh : C

détaillé en 4.3.8.2

Voir ISO 9945-2 : 3.14.5

11. Mot-clé : *exec commande*

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.6

12. Mot-clé : *exec close*

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.6

13. Mot-clé : *ouverture/duplication*

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.6

14. Mot-clé : *exit*

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.7

15. Mot-clé : `exit n`

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.7

16. Mot-clé : `export`

sh : AS

csh : AS

ksh : O

Expliqué en : -

Voir ISO 9945-2 : 3.14.8

17. Mot-clé : `export variable`

sh : C

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.8

18. Mot-clé : `export variable=valeur`

sh : CA

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.8

19. Mot-clé : `export -p`

sh : CA

csh : AS

ksh : NS

détaillé en 4.3.4.2.1

Voir ISO 9945-2 : 3.14.8

20. Mot-clé : `readonly [variable]`

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.9

21. Mot-clé : `readonly -p`

sh : NS

csh : N/A

ksh : NS

détaillé en 4.3.4.2.1

Voir ISO 9945-2 : 3.14.9

22. Mot-clé : *return*

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.9

23. Mot-clé : *shift [n]*

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.12

24. Mot-clé : *trap action signaux*

sh : C

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.13

25. Mot-clé : `trap`

sh : CA

csh : N/A

ksh : C

détaillé en 4.3.4.2.2

Voir ISO 9945-2 : 3.14.13

26. Mot-clé : `exec pour close on exec`

sh : N/A

csh : N/A

ksh : O

détaillé en 4.2.3

Voir ISO 9945-2 : 3.14.6

27. Mot-clé : `unset variable`

sh : O

csh : C

ksh : O

détaillé en 4.3.4.2.4

Voir ISO 9945-2 : 3.14.14

28. Mot-clé : `unset -v variable`

sh : CA

csh : C

ksh : CA

Expliqué en : -

Voir ISO 9945-2 : 3.14.14

29. Mot-clé : `unset -f fonction`

sh : NS

csh : N/A

ksh : NS

Expliqué en : -

Voir ISO 9945-2 : 3.14.14

30. Mot-clé : `set --`

sh : NS

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.11

31. Mot-clé : `set -[aeuvx]`

sh : C

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.11

32. Mot-clé : `set -C`

sh : N/A

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.14.11

33. Mot-clé : `"\a"`

sh : C

csh : C

ksh : O

Expliqué en : 4.3.4.1.6

Voir ISO 9945-2 : 3.2.3

34. Mot-clé : `true`

sh : C

csh : C

ksh : C

détaillé en 4.3.6.1

Voir ISO 9945-2 : 2.3

35. Mot-clé : `false`

sh : C

csh : C

ksh : C

détaillés en 4.3.6.3/4.3.6.2

Voir ISO 9945-2 : 2.3

36. Mot-clé : alias

sh : N/A

csh : AS

ksh : C

détaillé en 4.3.4.1.4

Voir ISO 9945-2 : 2.3

37. Mot-clé : unalias

sh : N/A

csh : AS

ksh : C

détaillé en 4.3.4.1.4

Voir ISO 9945-2 : 2.3

38. Mot-clé : jobs

sh : N/A

csh : C

ksh : C

détaillé en 4.3.7.2

Voir ISO 9945-2 : 2.3

39. Mot-clé : bg

sh : N/A

csh : C

ksh : C

détaillé en 4.3.7.2

Voir ISO 9945-2 : 2.3

40. Mot-clé : fg

sh : N/A

csh : C

ksh : C

détaillé en 4.3.7.2

Voir ISO 9945-2 : 2.3

41. Mot-clé : fc

sh : N/A

csh : N/A

ksh : C

détaillé en 4.3.4.1.7

Voir ISO 9945-2 : 2.3

42. Mot-clé : newgrp

sh : C

csh : N/A

ksh : C

détaillé en 4.3.7.1

Voir ISO 9945-2 : 2.3

43. Mot-clé : `getopts`

sh : N/A

csh : N/A

ksh : C

détaillé en 4.3.4.1.3

Voir ISO 9945-2 : 2.3

44. Mot-clé : `umask`

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.3

45. Mot-clé : `wait`

sh : C

csh : C

ksh : C Expliqué en : -

Voir ISO 9945-2 : 2.3

46. Mot-clé : `read`

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.3

47. Mot-clé : `kill -s nomsignal-pgid`

sh : N/A

csh : NS

ksh : N/S

détaillé en 4.3.4.1.1

Voir ISO 9945-2 : 4.32.4

48. Mot-clé : `kill -- -pgid`

sh : N/A

csh : NS

ksh : N/S

détaillé en 4.3.4.1.1

Voir ISO 9945-2 : 4.32.4

49. Mot-clé : `kill -n pid` sh : C

csh : AS

ksh : O

détaillé en 4.3.4.1.2

Voir ISO 9945-2 : 4.32

50. Mot-clé : `kill -s nomsignal pid`

sh : NS

csh : AS

ksh : AS

détaillé en 4.3.4.1.2

Voir ISO 9945-2 : 4.32

51. Mot-clé : `kill -l`

sh : NS

csh : AS

ksh : O

détaillé en 4.3.7.3

Voir ISO 9945-2 : 4.32.6.1

52. Mot-clé : `command`

sh : N/A

csh : N/A

ksh : N/A

détaillé en 5.2

Voir ISO 9945-2 : 2.3

53. Mot-clé : `cd`

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.3

54. Mot-clé : *cd repertoire*

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.3

### **3.1.3- Structures de contrôle**

1. Mot-clé : *if then else elif fi*

sh : C

csch : AS

ksh : C

détaillé en 4.3.2.1

Voir ISO 9945-2 : 3.10

2. Mot-clé : *for do done*

sh : C

csch : AS

ksh : C

détaillé en 4.3.2.1

Voir ISO 9945-2 : 3.10

3. Mot-clé : case esac

sh : C

csh : AS

ksh : C

détaillé en 4.3.2.1

Voir ISO 9945-2 : 3-10

4. Mot-clé : while do done

sh : C

csh : AS

ksh : C

détaillé en 4.3.2.1

Voir ISO 9945-2 : 3.10

5. Mot-clé : until do done

sh : C

csh : AS ksh : C

détaillé en 4.3.2.1

Voir ISO 9945-2 : 3.10

6. Mot-clé : in

sh : C

csh : AS

ksh : C

détaillé en 4.3.2.1

Voir ISO 9945-2 : 3-10

7. Mot-clé : &&

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3-10

8. Mot-clé : ||

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3-10

9. Mot-clé : !

sh : C

csch : C

ksh : C

détaillé en 4.3.2.1/4.3.6.4

Voir ISO 9945-2 : 3-10

10. Mot-clé : ;

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.10

11. Mot-clé : ; ;

sh : C

csch : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3-10

12. Mot-clé : &

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3-10

13. Mot-clé : *commande* | *commande*

sh : C

csh : C

ksh : C

détaillé en 4.3.6.4

Voir ISO 9945-2 : 3-10

14. Mot-clé : *pattern*)

sh : C

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.10

15. Mot-clé : *pattern* | *pattern*

sh : C

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3-10

16. Mot-clé : *nom\_fonction()*{...}

sh : C

csh : N/A

ksh : C

détaillé en 4.3.2.2

Voir ISO 9945-2 : 3-10

17. Mot-clé : ( *liste\_objets* )

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3-10

18. Mot-clé : {}

sh : C

csh : C

ksh : C

détaillé en 4.3.2.1

Voir ISO 9945-2 : 3-10

19. Mot-clé : *commande*

sh : C

csh : AS

ksh : C

détaillé en 4.2.4

Voir ISO 9945-2 : 3-10

20. Mot-clé : (*commande*)

sh : AS

csh : AS

ksh : C

Expliqué en :

Voir ISO 9945-2 : 3-10

21. Mot-clé : {*commande*}

sh : C

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3-10

22. Mot-clé : *!commande*

sh : C

csch : AS

ksh : C

détaillé en 4.3.6.4

Voir ISO 9945-2 : 3.10

23. Mot-clé : *&-*

sh : C

csch : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.5

24. Mot-clé : *[[ ]]*

sh : N/A

csch : AS

ksh : O

détaillé en 4.2.5

Voir ISO 9945-2 : 3.4

### **3.1.4- Variables et paramètres**

1. Mot-clé : \$\*

sh : C

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

2. Mot-clé : \$@

sh : C

csh : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

3. Mot-clé : \$#

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

4. Mot-clé : \$#

sh : C

csH : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

5. Mot-clé : \$-

sh : C

csH : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

6. Mot-clé : \$\$

sh : C

csH : AS

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

7. Mot-clé : \$!

sh : C

csH : D

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

8. Mot-clé : `$o`

sh : C

ssh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

9. Mot-clé : `$n`

sh : C ssh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.5.2

10. Mot-clé : `$variable`

sh : C

ssh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

11. Mot-clé : `}${variable}`

sh : C

ssh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

12. Mot-clé :  $\${variable:-mot}$

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

13. Mot-clé :  $\${variable:+mot}$

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

14. Mot-clé :  $\${variable:?mot}$

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

15. Mot-clé :  $\${variable:=mot}$

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

16. Mot-clé :  $\${var\#masque}$

sh : N/A

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

17. Mot-clé :  $\${var##masque}$

sh : N/A

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

18. Mot-clé :  $\${var\%masque}$

sh : N/A

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

19. Mot-clé : `${var%%masque}`

sh : N/A

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

20. Mot-clé : `${#var}`

sh : N/A

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.6.2

21. Mot-clé : `$((expression))`

sh : N/A

csh : N/A

ksh : C

détaillé en 4.3.8.4

Voir ISO 9945-2 : 3.6.4

### **3.1.5- Variables d'environnement**

1. Mot-clé : `$HOME`

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.6 et 3.5.3

2. Mot-clé : `$IFS`

sh : C

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.6 et 3.5.3

3. Mot-clé : `$LANG`

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.6 et 3.5.3

4. Mot-clé : `$LC_*`

sh : C

csch : C

ksh : C

détaillé en 4.3.5.2

Voir ISO 9945-2 : 2.6 et 3.5.3

5. Mot-clé : \$PATH

sh : C

csch : C

ksh : C

détaillé en 4.3.5.1

Voir ISO 9945-2 : 2.6 et 3.5.3

6. Mot-clé : \$PS1

sh : C

csch : AS

ksh : C

détaillé en 4.3.5.4

Voir ISO 9945-2 : 2.6 et 3.5.3

7. Mot-clé : \$PS2

sh : C

csch : AS

ksh : C

détaillé en 4.3.5.4

Voir ISO 9945-2 : 2.6 et 3.5.3

8. Mot-clé : `$PS4`

sh : N/A

csh : N/A

ksh : C

détaillé en 4.3.5.4

Voir ISO 9945-2 : 3.5.3

9. Mot-clé : `$COLUMNS`

sh : C

csh : N/A

ksh : C

détaillé en 4.3.5.4

Voir ISO 9945-2 : 2.6

10. Mot-clé : `$LINES`

sh : C

csh : N/A

ksh : C

détaillé en 4.3.5.4

Voir ISO 9945-2 : 2.6

11. Mot-clé : `$LINENO`

sh : N/A

csch : N/A

ksh : C

détaillé en 4.3.5.4

Voir ISO 9945-2 : 3.5.3

12. Mot-clé : \$ENV

sh : N/A

csch : N/A

ksh : C

détaillé en 4.3.5.4

Voir ISO 9945-2 : 3.5.3

13. Mot-clé : \$PPID

sh : N/A

csch : N/A

ksh : C

détaillé en 4.3.5.4

Voir ISO 9945-2 : 3.5.3

14. Mot-clé : \$SHELL

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.6

15. Mot-clé : \$LOGNAME

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.6

16. Mot-clé : \$TERM

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.6

17. Mot-clé : \$TMPDIR

sh : C

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.6

18. Mot-clé : \$TZ

sh : C

ssh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 2.6

### **3.1.6- Redirections**

1. Mot-clé : *commande [n]< fichier*

sh : C

ssh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.1

2. Mot-clé : *commande [n]> fichier*

sh : C

ssh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.2

3. Mot-clé : *commande [n]>| fichier*

sh : N/A

ssh : AS

ksh : C

détaillé en 4.3.3.2

Voir ISO 9945-2 : 3.7.2

4. Mot-clé : *commande [n]>> fichier*

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.3

5. Mot-clé : *commande [n]<< fichier*

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.4

6. Mot-clé : *commande [n]<<- fichier*

sh : C

csch : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.4

7. Mot-clé : *commande [n]<& fichier*

sh : AS

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.5

8. Mot-clé : *commande [n]& fichier*

sh : AS

csh : C

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.6

9. Mot-clé : *commande [n]<> fichier*

sh : N/A

csh : N/A

ksh : C

Expliqué en : -

Voir ISO 9945-2 : 3.7.7

## **3.2- Portage**

### **3.2.1- Portage : csh vers la norme**

1. Mot-clé : *break [break[...]]*

Equivalent *shell* normalisé : *break n*

Expliqué en : -

2. Mot-clé : `label` :

Equivalent *shell* normalisé : N/A

Expliqué en : -

3. Mot-clé : `goto`

Equivalent *shell* normalisé : N/A

Expliqué en : -

4. Mot-clé : `source fichier`

Equivalent *shell* normalisé : `. fichier`

Expliqué en : -

5. Mot-clé : `time`

Equivalent *shell* normalisé : N/A

détaillé en 4.3.9.2

6. Mot-clé : `time commande`

Equivalent *shell* normalisé : `time commande`

détaillé en 4.3.9.2

7. Mot-clé : `exit (expression)`

Equivalent *shell* normalisé : `exit expression`

Expliqué en : -

8. Mot-clé : `setenv`

Equivalent *shell* normalisé : `export`

Expliqué en : -

9. Mot-clé : `onintr`

Equivalent *shell* normalisé : `trap`

détaillé en 4.3.4.2.3

10. Mot-clé : `set noclobber`

Equivalent *shell* normalisé : `set -C`

Expliqué en : -

11. Mot-clé : `nohup arg`

Equivalent *shell* normalisé : `nohup arg`

détaillé en 4.3.9.3

12. Mot-clé : `nohup`

Equivalent *shell* normalisé : N/A

détaillé en 4.3.9.3

13. Mot-clé : `$argv[ * ]`

Equivalent *shell* normalisé : `$*`

Expliqué en : -

14. Mot-clé :  `$#argv`

Equivalent *shell* normalisé : \$#

Expliqué en : -

15. Mot-clé : \$status

Equivalent *shell* normalisé : \$#

Expliqué en : -

16. Mot-clé : nice

Equivalent *shell* normalisé : N/A

détaillé en 4.1.2

17. Mot-clé : chdir

Equivalent *shell* normalisé : cd

détaillé en 4.1.3

18. Mot-clé : alias *nom commande*

Equivalent *shell* normalisé : alias *nom=commande*

Expliqué en : -

19. Mot-clé : foreach

Equivalent *shell* normalisé : for

Expliqué en : -

20. Mot-clé : \$home

Equivalent *shell* normalisé : \$HOME

Expliqué en : -

21. Mot-clé : `$ifs`

Equivalent *shell* normalisé : `$IFS`

Expliqué en : -

22. Mot-clé : `$USER` Equivalent *shell* normalisé : `$LOGNAME`

Expliqué en : -

23. Mot-clé : `if (expression)`

Equivalent *shell* normalisé : `if [ expression ]`

Expliqué en : -

24. Mot-clé : `repeat`

Equivalent *shell* normalisé : N/A

Expliqué en : -

25. Mot-clé : `$shell`

Equivalent *shell* normalisé : `$SHELL`

Expliqué en : -

26. Mot-clé : `$path`

Equivalent *shell* normalisé : `$PATH`

détaillé en 4.3.5.1

27. Mot-clé : `$pwd`

Equivalent *shell* normalisé : `$(pwd)`

détaillé en 4.1.3

28. Mot-clé : `$prompt`

Equivalent *shell* normalisé : `$PS1`

Expliqué en : -

29. Mot-clé : `$mail`

Equivalent *shell* normalisé : `$MAIL`

Expliqué en : -

30. Mot-clé :  `$?variable`

Equivalent *shell* normalisé : ``({var:?})|| echo $?``

Expliqué en : 4.3.6.5

31. Mot-clé : `switch endsw`

Equivalent *shell* normalisé : `case esac`

Expliqué en : -

32. Mot-clé : `set variable=valeur`

Equivalent *shell* normalisé : `variable=valeur`

Expliqué en : -

33. Mot-clé : `@variable=valeur`

Equivalent *shell* normalisé : `variable=valeur`

détaillé en 4.3.8.1

34. Mot-clé : `set variable [/]=valeur`

Equivalent *shell* normalisé : N/A

Expliqué en : -

35. Mot-clé : `>! fichier`

Equivalent *shell* normalisé : `>| fichier`

Expliqué en : -

36. Mot-clé : `>& fichier`

Equivalent *shell* normalisé : `1> fichier 2>&1`

Expliqué en : -

37. Mot-clé : `>&! fichier`

Equivalent *shell* normalisé : `1> fichier 2>&1`

Expliqué en : -

38. Mot-clé : `>>& fichier`

Equivalent *shell* normalisé : `1>> fichier 2>&1`

Expliqué en : -

39. Mot-clé : `>>! fichier`

Equivalent *shell* normalisé : `>> fichier`

détaillé en 4.3.3.1

40. Mot-clé : `>>&! fichier`

Equivalent *shell* normalisé : >> *fichier 2>&l*  
détaillé en 4.3.3.1

41. Mot-clé : glob

Equivalent *shell* normalisé : set -f

Expliqué en : -

42. Mot-clé : kill -n *pid*

Equivalent *shell* normalisé : kill -s *signal pid*

Expliqué en : -

43. Mot-clé : kill -TERM *pid*

Equivalent *shell* normalisé : kill -s TERM *pid*

Expliqué en : -

44. Mot-clé : echo

Equivalent *shell* normalisé : echo **OU** printf

détaillé en 4.3.9.1

### **3.2.2- ksh vers la norme**

1. Mot-clé : l&

Equivalent *shell* normalisé : N/A

détaillé en 4.1.1

2. Mot-clé : *commande* | read *foo*

Equivalent *shell* normalisé : N/A

détaillé en 4.2.1

3. Mot-clé : select

Equivalent *shell* normalisé : N/A

détaillé en 4.2.5

4. Mot-clé : `times`

Equivalent *shell* normalisé : N/A

détaillé en 4.3.9.2

5. Mot-clé : `export`

Equivalent *shell* normalisé : `export -p`

détaillé en 4.3.4.2.1

6. Mot-clé : `command commande`

Equivalent *shell* normalisé : `command commande`

détaillé en 5.2

7. Mot-clé : `$PWD`

Equivalent *shell* normalisé : `$ (pwd)`

détaillé en 4.1.3

8. Mot-clé : `$OLDPWD`

Equivalent *shell* normalisé : N/A

détaillé en 4.3.5.5

9. Mot-clé : `let variable=valeur`

Equivalent *shell* normalisé : `variable=valeur`

Expliqué en : -

10. Mot-clé : `$OPTARG`

Equivalent *shell* normalisé : `getopts...`

détaillé en 4.3.4.1.3

11. Mot-clé : `$OPTIND`

Equivalent *shell* normalisé : getopts...  
détaillé en 4.3.4.1.3

12. Mot-clé : \$ERRNO  
Equivalent *shell* normalisé : \$?  
Expliqué en : -

13. Mot-clé : \$REPLY  
Equivalent *shell* normalisé : N/A  
détaillé en 4.3.4.1.5

14. Mot-clé : \${#tableau[\*]}  
Equivalent *shell* normalisé : N/A  
Expliqué en : -

15. Mot-clé : \${#tableau[@]}  
Equivalent *shell* normalisé : N/A  
Expliqué en : -

16. Mot-clé : \$ENV  
Equivalent *shell* normalisé : \$ENV  
détaillé en 4.3.5.4

17. Mot-clé : \$FCEDIT  
Equivalent *shell* normalisé : N/A  
détaillé en 4.3.5.3

18. Mot-clé : \$HISTFILE  
Equivalent *shell* normalisé : N/A  
détaillé en 4.3.5.3

19. Mot-clé : \$LINENO

Equivalent *shell* normalisé : \$LINENO

détaillé en 4.3.5.4

20. Mot-clé : \$PPID

Equivalent *shell* normalisé : \$PPID

détaillé en 4.3.5.4

21. Mot-clé : \$PS3

Equivalent *shell* normalisé : N/A

détaillé en 4.3.5.4

22. Mot-clé : \$PS4

Equivalent *shell* normalisé : \$PS4

détaillé en 4.3.5.6

23. Mot-clé : (( ))

Equivalent *shell* normalisé : N/A

détaillé en 4.3.8.3

24. Mot-clé : kill -n *pid*

Equivalent *shell* normalisé : kill -s *nomsignal pid*

Expliqué en : -

25. Mot-clé : kill -TERM *pid*

Equivalent *shell* normalisé : kill -s TERM *pid*

Expliqué en : -

26. Mot-clé : echo

Equivalent *shell* normalisé : echo **OU** printf

détaillé en 4.3.9.1

27. Mot-clé : `cd -`

Equivalent *shell* normalisé : N/A

Expliqué en : -

28. Mot-clé : `cd repertoire repertoire`

Equivalent *shell* normalisé : N/A

Expliqué en : -

29. Mot-clé : `typeset`

Equivalent *shell* normalisé : N/A

Expliqué en : -

30. Mot-clé : `whence`

Equivalent *shell* normalisé : `find - - -`

Expliqué en : -

### **3.2.3- Portage : sh vers la norme**

1. Mot-clé : `commande|read foo`

Equivalent *shell* normalisé : N/A

détaillé en 4.2.1

2. Mot-clé : `times`

Equivalent *shell* normalisé : N/A

détaillé en 4.3.9.2

3. Mot-clé : `nom=valeur ; export nom`

Equivalent *shell* normalisé : `export nom=valeur`

Expliqué en : -

4. Mot-clé : `trap signaux`

Equivalent *shell* normalisé : `trap - signaux`

détaillé en 4.3.4.2.2

5. Mot-clé : `unset fonction`

Equivalent *shell* normalisé : `unset -f fonction`

détaillé en 4.3.4.2.4

6. Mot-clé : `shift $#`

Equivalent *shell* normalisé : `set --`

Expliqué en : -

7. Mot-clé : `kill -n pid`

Equivalent *shell* normalisé : `kill -s nomsignal pid`

Expliqué en : -

8. Mot-clé : `kill -TERM pid`

Equivalent *shell* normalisé : `kill -s TERM pid`

Expliqué en : -

9. Mot-clé : `echo`

Equivalent *shell* normalisé : `echo` OU `printf`

détaillé en 4.3.9.1

10. Mot-clé : `getopt`

Equivalent *shell* normalisé : `getopts`

détaillé en 4.3.4.1.3

---

## 4- Commentaires

Contenu de ce chapitre

## 4.1- Ce qui n'existe plus

### 4.1.1- Co-processus en *Korn-shell*

La combinaison "l&" (non quotée) permet de faire communiquer par "pipe" des processus lancés en arrière-plan.

Cette communication est mise en oeuvre avec les commandes internes read, print associées à l'option "-p".

Cette fonctionnalité n'est donc pas reportée en *shell* normalisé.

Voir par. 3.2.2 ligne [1].

### 4.1.2- Commande interne nice du *C-shell*

nice est désormais un utilitaire (5.20.). Sa syntaxe a évolué.

La commande a été spécifiée pour retourner le code d'erreur 127, afin de distinguer une erreur à nice d'une erreur du *shell* (accès à la commande).

**Exemple :** nice -n increment commande arg

Voir par. 3.2.1 ligne [16].

### 4.1.3- Accès au répertoire courant par l'environnement

Le paramètre d'environnement \$PWD n'existe plus puisque l'on peut accéder à sa valeur par la commande pwd évaluée avec la syntaxe suivante : \$(pwd), qui correspond à l'écriture 'pwd' en *Bourne-shell*.

Voir par. 3.2.1 ligne [27]

Voir par. 3.2.2 ligne [7]

## 4.2- Risques liés à l'implémentation

### 4.2.1- Sous-*shell* implicites

La redirection des entrées sorties au sein de commandes internes du *shell*, provoque un 'fork' pour l'exécution des sous-commandes.

En conséquence, en raison des différences de comportement à l'utilisation des commandes internes, cette fonctionnalité est permise mais non supportée.

**Exemple :**

```
echo chien chat souris | read x y z
```

```
echo $x $y $z
```

Si le read est dans un sous-*shell*, les variables x, y et z ne sont pas affectées par la commande contenant le pipe.

Voir par. 3.2.2 ligne [2]

Voir par. 3.2.3 ligne [1].

## 4.2.2- Comparaison du caractère "/" dans l'expansion des noms de fichiers

Attention, le caractère "/" ne peut correspondre aux masques habituels "\*", "?", "[", en raison de son interprétation qui est prioritaire dans la boucle d'interprétation du *shell*.

En conséquence la chaîne a[b/c]d correspond bien à a[b/c]d et non pas a/d par exemple. Toutefois, certains systèmes en ont une implémentation différente.

Le résultat peut être non-portable.

Voir par. 3.1.2. ligne [3].

## 4.2.3- Commande interne *exec*

La syntaxe *exec* seule (commande sans argument), introduite par le *Korn-shell* pour fermer tous les descripteurs de fichiers, sauf 0, 1 et 2, a un comportement non spécifié dans le *shell* normalisé.

Voir par. 3.1.2 ligne [26]

## 4.2.4- Grammaire

La grammaire actuelle est identique à celle du *Bourne-shell* et du *Korn-shell*, et ce, pour des raisons de compatibilité historique. Toutefois, cette grammaire du *shell* (définie au 3.10 de la norme) ne représente pas une syntaxe formelle. En effet, la grammaire pourrait évoluer, et par conséquent, la compatibilité avec la grammaire des *shell* pré-cités ne serait plus assurée.

Voir par. 3.1.3 ligne [19].

## 4.2.5- Compatibilité non garantie avec le *Korn-shell*

Le mot-clé *select* peut être reconnu comme un mot réservé mais d'effet non garanti (cf 3.4 de la norme). La syntaxe [[]] utilisée pour l'évaluation d'expressions, en particulier dans les tests (*if*), peut être encore reconnue dans certaines implémentations, mais la norme ne garantit pas le résultat de leur emploi car celui-ci n'est pas spécifié.

En d'autres termes, toute implémentation empruntant, au titre d'extension, des commandes de *shell* antérieures à la norme, ne peut garantir un comportement de celles-ci compatible avec la norme et/ou avec le *shell* duquel cette fonctionnalité s'inspire.

Voir par. 3.1.3 ligne [24].

# 4.3- Ce qui a changé

## 4.3.1- Fichiers requis

Les fichiers requis définis ici sont les fichiers obligatoires pour que le *shell* et les commandes fonctionnent.

La norme impose aussi les répertoires `/dev` et `/etc`, mais ne spécifie pas d'arborescence complète normalisée.

Celle-ci reste à la discrétion du fournisseur et des utilisateurs, même si l'usage a consacré des arborescences considérées comme désormais standards.

Voir par. 3.1.1 ligne [1 à 4].

## 4.3.2- Structures de contrôle

### 4.3.2.1- Mots réservés

La normalisation du *shell* définit des mots réservés qui ont une signification pour le *shell*. Ces mots sont `!`, `case`, `for`, `do`, `done`, `elif`, `else`, `esac`, `fi`, `for`, `fi`, `in`, `then`, `until`, `while`, `{`, `}`. Ceci peut poser un problème de portabilité si le programmeur déclare un alias redéfinissant un de ces mots réservés. En effet son programme fonctionnera en *C-shell* et en *Korn-shell*, mais pas en *shell* normalisé.

**Exemple :**

**csh :**

```
alias case pwd
```

```
case
```

**norme :**

```
alias case=pwd
```

```
case
```

Le *C-shell* le tolérera très bien, mais l'écriture équivalente en *shell* normalisé sera rejetée.

Voir par. 3.1.3 ligne [1 à 6]

Voir par. 3.1.3 ligne [9]

Voir par. 3.1.3 ligne [18].

### 4.3.2.2- Fonctions en *Korn-shell*

Le *Korn-shell* admet la même syntaxe des définitions de fonctions que celles du *shell* normalisé, préservant la compatibilité avec le *Bourne-shell* (`sh`).

Il utilise en plus le mot-clé "*function*" précédant le nom de la fonction définie pour "*typer*" explicitement celle-ci. Il convient donc pour tout portage, de supprimer purement et simplement ce mot-clé "*function*".

Voir par. 3.1 ligne 3 [16].

## 4.3.3- Redirections

### 4.3.3.1- Ecriture forcée en fin de fichier

La norme ne dit pas si l'écriture forcée en fin de fichier pour outrepasser la

protection en écriture du noclobber peut s'écrire >>|

Il convient donc, dans le doute, de précéder ce type de commande par set +C.

Voir par. 3.2.1 ligne [39]

Voir par. 3.2.1 ligne [40].

### **4.3.3.2- Ecriture forcée en début de fichier**

La norme a repris la syntaxe >| pour écrire sur un fichier protégé en écriture par le noclobber positionné par la commande set -C

Voir par. 3.1.6 ligne [3].

## **4.3.4- Options et fonctions internes**

### **4.3.4.1- Options et fonctions internes régulières**

#### **4.3.4.1.1- Numéro de PID**

Un numéro négatif de PID est considéré comme un PGID (n'existe pas en *Korn-shell*). Un numéro de PGID seul doit être précédé de --.

Un PGID est un identificateur d'ordonnancement commun à tous les processus d'une même session.

Voir par. 3.1.2 ligne [47]

Voir par. 3.1.2 ligne [48].

#### **4.3.4.1.2- Syntaxe du kill**

La forme de kill avec numéro de signal et numéro de processus est encore acceptée par le *shell* normalisé mais considérée comme obsolète.

Suivant l'implantation système, les numéros de signaux peuvent varier dans la tranche au-dessus de 15 (cf 9945-1 par. 3.3.1.1).

Voir par. 3.1.2 ligne [49]

Voir par. 3.1.2 ligne [50].

#### **4.3.4.1.3- Commande getopts**

La manipulation des arguments d'une commande en nombre et position est possible avec l'utilitaire getopts.

getopts peut être considéré comme une commande interne de type régulier.

De plus, getopts remplace l'ancien utilitaire getopt en proposant une syntaxe plus simple à programmer. En particulier, il n'est plus nécessaire avec getopts de procéder aux décalages de paramètres à analyser (commande shift).

Voir par. 3.1.2 ligne [43]

Voir par. 3.2.2 ligne [11]

Voir par. 3.2.2 ligne [12]

Voir par. 3.2.3 ligne [10].

#### **4.3.4.1.4- Commandes `alias` et `unalias`**

Les commandes *built-in* `alias` et `unalias` ne sont pas recommandées dans un *shell-script*. De plus, contrairement aux variables d'environnement, les `alias` ne sont pas transmis aux processus enfants, il faut donc les reinitialiser à chaque nouveau *shell* créé. C'est pour cette raison que l'on a l'habitude de les déclarer dans le fichier `.cshrc` en *C-shell*. Voir 3.1.2 [36]

Voir par. 3.1.2 ligne [37]

Voir par. 3.1.2 ligne [41].

#### **4.3.4.1.5- Variable `$REPLY` en *Korn-shell***

`REPLY` contient la valeur entrée lors de l'exécution de la commande `select` ou d'une commande `read` sans argument (en *Korn-shell*).

##### **Exemple :**

**ksh :**

```
print "nom de fichier"
read
print $REPLY
```

**norme :**

```
echo "nom de fichier"
read f
echo $f
```

Voir par. 3.2.2 ligne [14].

#### **4.3.4.1.6- Séquences d'échappement**

Certains caractères qui sont significatifs perdent leur caractéristique. `echo "\a"` doit afficher `\a` et non produire un *beep* sonore.

#### **4.3.4.1.7- Edition des commandes**

La commande `fc` est consacrée à l'usage interactif de l'édition des commandes précédentes, enregistrées dans l'historique; il n'est pas possible de baser la programmation sur le résultat de `fc`.

Voir par. 3.1.2 ligne [41].

### **4.3.4.2- Options et fonctions internes spéciales**

#### **4.3.4.2.1- Option de rejeu dans la norme**

Pour les commandes `trap`, `export` et `readonly`, le *shell* normalisé ajoute le rejeu.

Pour export et readonly, ceci est réalisé avec l'option "-p".

Le *Korn-shell* permet de connaître la liste des déroutements (des signaux) mais cette liste doit être modifiée pour être rejouée.

Voir par. 3.1.2 ligne [19]

Voir par. 3.1.2 ligne [21]

Voir par. 3.2.2 ligne [5]

#### **Exemple : fonctionnement normalisé**

```
/trap
%trap "echo toto" SIGQUIT
%trap > derout
%cat derout
trap "echo toto" SIGQUIT
%kill -s SIGQUIT $$
toto
%trap - SIGQUIT
%trap
%chmod u+x derout
%derout
%trap
trap "echo toto" SIGQUIT
%
```

#### **4.3.4.2.2- Rejeu de la commande `trap` en *Bourne-shell***

Comme pour les commandes export et readonly, la possibilité de rejeu a été introduite. Le rejeu ne se fait pas avec l'option -p, mais en reprenant la syntaxe du *Korn-shell*. Ceci pose un problème de portabilité, car une même syntaxe fournit deux formats de sortie différents. Le format de la sortie rejouable est le même que celui d'export et readonly.

Voir par. 3.2.3 ligne [4].

#### **4.3.4.2.3- Traitements d'exceptions**

La commande trap a des fonctionnalités plus étendues que onintr qui ne permet d'agir que sur le signal SIGINT. La liste des signaux sans leur suffixe est donnée par la commande kill -l.

Le paramétrage des deux commandes est différent :

#### **Exemple 1 :**

```
trap action sig
```

permet d'exécuter *action* lorsque le *shell* reçoit le signal *sig*.

```
onintr label
```

le *shell* exécute un goto *label* lors de la réception de l'interruption.

#### **Exemple 2 :**

```
onintr -
```

permet d'ignorer toutes les interruptions, est équivalent à :

```
trap '' sig1 sig2..
```

#### **Exemple 3 :**

```
onintr
```

permet de retrouver l'action par défaut associée à l'interruption.

```
trap sig
```

```
trap - sig
```

permet de retrouver l'action par défaut associée au signal *sig*.

Si plusieurs exceptions (signaux) sont apparues et en attente pour un *shell* qui comporte des actions liées à la capture de ces exceptions, et ce, à travers la commande *trap*, l'ordre d'exécution de ces actions n'est pas spécifié.

La capture des signaux SIGKILL et SIGSTOP que certaines implémentations anciennes pouvaient autoriser, n'est désormais, plus possible.

#### **4.3.4.2.4- Commande unset en Korn-shell**

La possibilité de supprimer une fonction avec la commande *unset fonction* sans l'option *-f* est interdite.

Une fonction ne peut être supprimée que par : *unset -f fonction*.

Voir par. 3.2.3 ligne [5].

## **4.3.5- Variables d'environnement**

### **4.3.5.1- Chemin d'accès aux programmes**

Certains *shell* anciens pouvaient lancer, dans leur contexte de processus, l'exécution d'un fichier du répertoire courant même si ce dernier n'était pas inclus dans la variable *PATH*. La normalisation du *shell* impose que tous les fichiers, exécutés par un *shell*, aient un chemin indiqué dans la variable *PATH*. Cette modification permet de se protéger des chevaux de Troie (mesure de sécurité).

Remarque :

L'expansion du tilde (~) est conservée pour compatibilité avec le *C-shell* et le *Korn-shell*, à condition d'être évoqué comme premier chemin d'accès dans la variable *PATH*.

Voir par. 3.1.5 ligne [5]

Voir par. 3.2.1 ligne [26].

### 4.3.5.2- Internationalisation

Les possibilités d'internationalisation du *shell* normalisé comprennent deux aspects :

- les jeux de caractères,
- les "locales".

Le *shell* normalisé impose un jeu de caractères contenant l'ensemble des caractères ASCII. Il est à noter que seuls les caractères sont définis et non leur code (à l'exception du caractère qui doit être codé à zéro).

L'utilisation de plusieurs jeux de caractères au sein d'une même machine ou d'une même application est encore aujourd'hui complexe et pose de nombreux problèmes de portabilité. Une "locale" est un sous-ensemble de l'environnement de l'utilisateur, qui permet d'adapter les programmes aux conventions nationales.

Six variables d'environnement sont utilisées :

LC\_CTYPE : différents caractères utilisables par classe (alphabétique, numérique, ponctuation) et façon de passer des minuscules aux majuscules, et vice versa

LC\_COLLATE : classement des caractères (jusqu'à présent les tris se faisaient sur le code, ce qui en ASCII se traduit par un classement mettant "Z" avant "a").

LC\_TIME : format des dates et heures

LC\_MONETARY : format des montants monétaires

LC\_NUMERIC : format des nombres (hors monnaie)

LC\_MESSAGES : format des messages d'information et de diagnostic, ainsi que des réponses interactives

L'utilitaire `localedef` permet de définir une "locale" particulière.

Le mécanisme des "locales" est encore embryonnaire. L'annexe H de la norme ISO9945-2 définit les améliorations qui seront apportées lors des prochaines versions de la norme.

Ce sera en particulier le cas pour l'utilisation des caractères "multi-octets" déjà disponibles en langage C.

Afin de garantir une portabilité lors de l'utilisation des "locales", il est indispensable de connaître les utilitaires affectés par chacune des variables d'environnement (regroupées sous le nom `LC_*`).

Cela est détaillé dans le guide de migration XPG3 - XPG4 de X/Open.

**Exemple** : pour maîtriser le classement des noms de fichiers dont l'expansion est automatique dans la commande qui l'invoque, il vaut mieux écrire :

```
Save_COLLATE=$LC_COLLATE
```

```
LC_COLLATE=POSIX
```

```
cat x??.data mon_fichier
```

```
LC_COLLATE=$Save_COLLATE
```

Voir par. 3.1.5 ligne [4].

#### 4.3.5.3- Edition des commandes

Les variables d'environnement FCEDIT, HISTFILE et LINENO ne sont pas spécifiées dans la norme.

Elles sont conservées pour assurer la compatibilité avec le *Korn-shell* en particulier; la plupart d'entre elles concernent le comportement interactif du *shell*.

Voir par. 3.2.3 ligne [17 à 19]

#### 4.3.5.4- Particularisation du dialogue

PS1 et PS2 sont hérités du *Bourne-shell*. PS3 est la variable d'environnement contenant la chaîne de caractères du message de sollicitation (*prompt*) réservé à *select* en *Korn-shell*. Il suffit de garder l'initialisation de PS3, de remplacer chaque *select* par un *case*, et de faire précéder chacun d'eux, d'une commande *echo \$PS3*.

PS4 est réservé aux traces (voir plus loin 4.3.5.6)

COLUMNS et LINES dédiées à la caractérisation du terminal (ou de la fenêtre), LINENO, ENV et PPID, peuvent affecter les utilitaires et sont, au même titre que PS4, considérés comme des options des systèmes assurant une portabilité pour l'utilisateur (*systems supporting the User Portability Utilities Option*).

Voir par. 3.1.5 ligne [6 à 13]

Voir par. 3.2.2 ligne [16]

Voir par. 3.2.2 ligne [19 à 21]

#### 4.3.5.5- Répertoire précédemment accédé

OLDPWD contient l'avant-dernier chemin d'accès utilisé (répertoire courant avant la dernière commande *cd*).

Cette variable n'est pas conservée en *shell* normalisé. Il faut donc enregistrer dans cette variable, le nom du répertoire courant avant chaque invocation de la commande *cd*.

Voir par. 3.2.2 ligne [8].

#### 4.3.5.6- Mise au point

PS4 est un message de sollicitation ("*prompt*") utile au débogage dans le *Korn-shell*. La norme l'adopte pour le comportement interactif du *shell* afin d'assurer une trace (suite à l'emploi de la commande *set-x*) sur la sortie standard de toutes les commandes exécutées. Sa valeur par défaut est "+".

Voir par. 3.1.5 ligne [8]

Voir par. 3.2.2 ligne [22]

## 4.3.6- Logique

### 4.3.6.1- Vrai

Dans la norme, la commande `true` est simulée pour être *built-in*, comme un alias exemple en *shell* normalisé :

```
true=':'
```

La ligne *shell* :

```
while true; do sleep 1; done
```

devient

```
while : ; do sleep 1; done
```

`true` peut être considéré comme une commande interne de type régulier

Voir par. 3.1.2 ligne [34]

### 4.3.6.2- Faux

La commande `false` est similaire à l'alias suivant :

```
false='let 0'
```

Lors du portage vers le *shell* normalisé, il faut supprimer l'alias : en effet, celui-ci ne sera plus utilisé.

`false` peut être considéré comme une commande interne de type régulier.

Voir par. 3.1.2 ligne [35]

### 4.3.6.3- Succès et insuccès

Il est important de tenir compte, lors du portage de *shell-script*, de la valeur retournée (*exit status*) par la commande `false`.

En effet elle est différente suivant le *shell* utilisé :

```
1 "en shell normalisé"
```

```
1Korn-shell
```

```
255 (-1) Bourne-shell
```

Voir par. 3.1 ligne [35]

### 4.3.6.4- Cas des enchaînements avec "tubes"

L'usage des tubes entre commandes (*pipelines*) permet un enchaînement compatible des commandes quel que soit le *shell*. Néanmoins, il faut être conscient d'une éventuelle différence de comportement, si la parenté des sous-*shell* est différente dans la séquence `a | b | c` selon l'ordre de lancement des processus.

Utilisation de la négation avec "!" : La sémantique en *shell* normalisé du "pipeline" est [ ! ] commande1 [ | commande2 ... ].

Celle-ci est identique pour le *Bourne-shell* et le *Korn-shell*, à l'exception de la négation marquée par '!'.

En effet, cette facilité d'écriture n'est pas disponible dans les *shell* pré-cités. Cette option permet d'inverser le code retour du pipeline, à savoir si la négation est invoquée avec '!' et que le code retour est 0, alors on obtient son complément, c'est-à-dire 1.

Voir par. 3.1.3 ligne [13]

Voir par. 3.1.3 ligne [22].

#### **4.3.6.5- Paramètre non initialisé**

Pour savoir si un paramètre est nul ou non initialisé, la seule méthode est de recourir comme en *Bourne-shell* à l'écriture \$ {paramètre:?}. Cette écriture produit un message sur "stderr" (sortie d'erreur) suivie d'une sortie du *shell* avec un code retour différent de 0.

Ceci impose, pour traduire l'écriture du *C-shell* qui permet de savoir si une variable est initialisée par \$?variable, d'avoir recours aux deux possibilités suivantes :

##### **Exemple 1 :**

**cs**h :

```
if ($?var) then
echo parametre set
else
echo parametre unset
endif
```

**norme :**

```
x=`(${var:?}) || echo $?`
if [ ! $x ]
then
echo parametre set
else
echo parametre unset
fi
```

##### **Exemple 2 : passer par une assignation bidon**

**cs**h :

```
if ($?var) then
```

```

echo parametre set

else

echo parametre unset

endif

norme :

if [ ! ${var:=x} ]

then

echo parametre set

else

echo parametre unset

fi

```

## 4.3.7- Sécurité et interactivité

### 4.3.7.1- Changement de groupe : `newgrp`

La commande `newgrp` permet de changer le GID ( "*Group IDentifier*" : numéro de groupe d'utilisateurs) principal de l'utilisateur avec un des GID colatéraux de celui-ci. Le GID principal est celui initialisé à la connexion à partir de la base de données des utilisateurs, généralement `/etc/passwd`.

Les GIDs colatéraux sont les GIDs des groupes auxquels l'utilisateur appartient, et décrits dans la base de données des groupes, généralement `/etc/group`.

Une utilisation du changement de GID principal, permet à l'utilisateur de changer le groupe propriétaire par défaut des fichiers nouvellement créés (dans un répertoire qui ne bénéficie pas d'un SGID sur lui-même).

Une autre utilisation est de pouvoir faire de la comptabilité analytique en sachant sous quel GID et donc, par exemple, sous quel projet l'utilisateur travaille.

En *Bourne-shell*, *Korn-shell*, et dans la norme, `newgrp` change le GID du *shell* courant.

En *C-shell*, `newgrp` ne fonctionne pas, et ne sait changer le GID principal qu'en ré-exécutant un *C-shell*. Cette duplication interdit l'utilisation de `newgrp` dans les *shell-script* en *C-shell*.

En interactif, cela pose d'autres problèmes, comme avec les systèmes de fenêtrage, et donc l'utilisation d'un *shell* normalisé est toujours préférable.

Voir par. 3.1.2 ligne [42].

### 4.3.7.2- Gestion des travaux

Les commandes *built-in jobs*, `fg`, `bg` peuvent éventuellement être invoquées dans des *shell-script* mais cela n'a pas beaucoup de sens. De plus, elles doivent être considérées comme réservées à un usage interactif.

### 4.3.7.3- Liste des signaux

La forme `kill -l` doit énumérer les signaux sans autres caractères séparateurs que `<space>` et `<newline>`, les signaux étant identifiés par leur nom sans le préfixe SIG (cf `signal.h`).

Dans tous les autres cas, le format de sortie n'est pas garanti.

Voir par. 3.1.2 ligne [51].

## 4.3.8- Arithmétique / Evaluation

### 4.3.8.1- Expression arithmétique en *C-shell*.

Pour toute assignation de valeur arithmétique calculée, il faut utiliser l'utilitaire `expr` comme pour le *Bourne-shell*. **Exemple :**

**csch :**

```
@x=$c * 5
```

**norme :**

```
x=`expr $c \* 5`
```

Voir par. 3.2.1 ligne [33].

### 4.3.8.2- Evaluation des expressions

Le mécanisme d'appel à l'évaluateur interne du *shell*, par construction dynamique d'une commande à l'aide d' `eval`, est le même dans les trois *shell*. Cependant, la construction d'une commande qui ne serait pas marquée C dans les tableaux synoptiques rend le *script* non portable vis-à-vis du *shell* normalisé

Voir par. 3.1.2 ligne [10].

### 4.3.8.3- Evaluation en *Korn-shell*

`(( ))` remplace l'utilisation de `let` pour le *Korn-shell* (arithmétique avec des entiers longs).

```
%let x=2+3+4*2
```

```
%echo $x
```

```
13
```

```
%(( y = 2*3+7-17))
```

```
%echo $y
```

```
- 4
```

Voir par. 3.1.2 ligne [23].

### 4.3.8.4- Evaluation arithmétique

La structure `$((expression))` n'est pas supportée en *Bourne-shell* et *C-shell*. Il faut respectivement utiliser `expr` et le caractère `@`.

**norme :**

```
x = $(( $x-1 ))
```

**sh :**

```
x = `expr $x-1`
```

**csch :**

```
@x = $x-1
```

## 4.3.9- Utilitaire

### 4.3.9.1- Commande echo

Historiquement, la commande `echo` autorise la suppression du caractère 'new-line' (ou `\n`) de deux manières différentes selon l'origine du système Unix (BSD ou System V) :

```
BSD echo -n "string"
```

```
System V echo "string\c"
```

System V autorise au contraire de BSD, l'utilisation de séquences d'échappement `\b`, `\r`, `\t`, `\Onum`, ... etc. La commande *shell* normalisé `echo`, n'a pas tranché face à ce problème de portabilité, et retient la forme la plus simple c'est-à-dire `echo "string"`.

L'implémentation de format d'affichage plus complet nécessite le recours à l'usage de la commande *shell* normalisé `printf` .

```
exemple : printf "%b\n" "$@"
```

équivalent à la commande `echo` de System V, le format `"%b"` permettant le traitement des séquences d'échappement de celle-ci.

Notons que XPG4 garantit à la commande `echo` une compatibilité avec les syntaxes SystemV et BSD.

Voir par. 3.2.1 ligne [44]

Voir par. 3.2.2 ligne [26]

Voir par. 3.2.3 ligne [9]

### 4.3.9.2- Commandes time et times

Il faut passer de la forme :

```
0.1 real 0.0 user 0.0 sys donné par time
```

à la forme

```
14m9s 4m48s donné par times
```

Remarque : le format de sortie est différent ainsi que le contenu

Voir par. 3.2.1 ligne [5]

Voir par. 3.2.1 ligne [6]

Voir par. 3.2.2 ligne [4]

Voir par. 3.2.3 ligne [2].

### 4.3.9.3- Commande `nohup`

La commande `nohup` devient un utilitaire.

Voir par. 3.2.1 ligne [11]

Voir par. 3.2.1 ligne [12].

---

## 5- Implémentation

Contenu de ce chapitre

### 5.1- Rappels sur la boucle d'interprétation du *shell*

Ces différentes phases sont données à titre indicatif afin de clarifier le fonctionnement générique des *shell*.

**Phase 1 :** Lecture des lignes de commandes. Ces lignes sont décomposées en mots sur reconnaissance des séparateurs 'space', <tab> et 'newline'. Le *shell* procède en une analyse syntaxique afin de reconnaître le format des commandes. En particulier, le *shell* reconnaît les commandes séparées par un point virgule ou qui sont décrites sur plusieurs lignes.

**Phase 2 :** Le *shell* procède à une vérification syntaxique du format de la commande. A ce niveau sont détectées les redirections d'entrée-sortie ou l'utilisation de "tubes". Le *shell* crée alors les fichiers correspondants aux redirections si nécessaire.

**Exemple :**

```
%var = "ls | wc -l"
```

```
;%$var
```

```
provoque les erreurs
```

```
| not found
```

```
wc not found
```

```
-l not found
```

En effet, l'interprétation du "l" se faisant avant la substitution de variables, l'apparition du "l" arrive trop tard dans l'exemple. ("l", wc et -l, sont pris comme arguments de la commande ls).

Ce problème est résolu par la commande interne `eval` qui force un nouveau passage dans la boucle.

```
%eval $var.
```

**Phase 3 :** Le *shell* applique sur la ligne de commande les mécanismes de substitution. A l'issue de cette phase le nom de la commande et tous ses arguments sont connus.

**Phase 4 :** La commande à exécuter est évaluée dans l'ordre suivant :

alias/fonction/commande interne/commande dans le PATH.

- Pour supprimer l'évaluation des alias, on préfixe la commande par "`\`". - Pour supprimer l'évaluation des alias et des fonctions, on lance la commande comme argument de la commande interne `command`.

- Pour supprimer alias/fonction/commande interne, il faut nommer la commande par son chemin complet.

Remarques :

Les *shell* actuels peuvent avoir des comportements très différents vis-à-vis de l'ordre d'évaluation de la phase 4.

Si la première ligne d'un *shell-script* débute par "`!`", le résultat n'est pas spécifié par la norme.

## 5.2- Utilitaires internes

Tout utilitaire standard du *shell* peut être implémenté en interne dans l'interpréteur. On améliore ainsi ses performances. C'est le cas des utilitaires les plus courants.

Toutefois, ces utilitaires, comme par exemple les commandes `cd`, `false`, `kill`, `true`, `wait`, `command`, `getopts`, `read`, `umask`, seront implémentées de telle sorte qu'elles puissent être accessibles via la famille d'appels système `exec` référencées dans ISO 9945-1 (sous réserve que le système d'exploitation fournisse de tels services aux programmes d'applications) et puissent être directement invoquées par les utilitaires standards tels que `env`, `find`, `nohup`, `xargs` ...

**Exemple :** `find ... exec true ;-a...` où "`true`" est substitué en tant que binaire pour un débogage temporaire d'une commande `find`.

Aujourd'hui, tout utilitaire peut être invoqué par une fonction de la famille `exec`. Il n'y a aucune obligation à ce qu'il soit implémenté en interne, mais dans la plupart des cas cette implémentation est justifiée par le fait qu'on évite ainsi une recherche dans le PATH. Il y a plusieurs raisons pour lesquelles `umask` et `command` sont incluses dans les commandes internes ("*built-in commands*.) :

- Accroissement sensible des performances fonctionnelles (pas de génération de processus).

- Garantie d'intégrité de la commande (on évite ainsi les effets de bord imprévisibles) .

Le *shell* normalisé fait clairement la distinction entre les utilitaires internes dits réguliers ("*Regular*") et spéciaux ("*Special*").

Les distinctions sont les suivantes :

- Une erreur dans le déroulement d'une commande interne spéciale force la sortie du *shell*, ce qui n'est pas le cas pour une commande interne régulière.

- La commande "command" permet de transformer un "Spécial" en "Regular". En effet, cette commande permet de traiter l'argument comme une simple commande .

**Exemple :**

```
% command exec > unwritable file.
```

On évite ici, la sortie (*exit*) au niveau du *shell*, si le fichier est protégé en écriture.

Liste des commandes régulières :

- alias
- false
- getopts
- newgrp
- umask
- bg
- fc
- jobs
- read
- unalias
- col
- fg
- kill
- true
- wait
- command

Liste des commandes spéciales :

- break - continue
- .
- eval
- exec
- exit
- export

- readonly
- return
- set
- shift
- trap
- unset
- ;

## 5.3- Valeurs de configuration

Toute commande admettant des arguments et/ou des données disponibles sur l'entrée standard peut voir le nombre et la taille de celles-ci limités par des paramètres de configuration. Ces paramètres héritent des valeurs déjà définies dans ISO 9945-1. Il y a un jeu de valeurs pour chaque utilitaire.

Les paramètres de configuration indépendants des utilitaires sont principalement :

{PATH\_MAX} défini dans POSIX 1 (8) 2.8 à rapprocher de MAX\_CANON

{LINE\_MAX}= 2048 ( = {POSIX\_2\_LINE\_MAX} )

{ARG\_MAX} défini dans POSIX 1 (8) mais considéré comme difficile à exploiter compte tenu du fait que l'utilisateur peut difficilement connaître combien d'espaces il a déjà consommé dans l'environnement utilisateur, et donc combien il reste de place pour empiler tous les arguments des commandes à traiter.

Il existe des constantes symboliques relatives à la portabilité des environnements de développement; elles sont :

{POSIX2\_C\_BIND} pour accès au *shell* depuis le langage C (via system(), ... etc)

{POSIX2\_C\_DEV} pour accès aux outils de développement en C (c89, lex, yacc, ... etc)

{POSIX2\_FORT\_DEV} pour accès aux outils de développement en Fortran (fort77, asa)

{POSIX2\_FORT\_RUN}

{POSIX2\_LOCALEDEF} pour création de localisations (cf 4.35: localedef)

{POSIX2\_SW\_DEV} pour accès aux outils de développement logiciel (make, ar, strip)

L'usage de l'utilitaire getconf permet d'adapter tout traitement aux limites imposées par l'implémentation des commandes utilisées.

**Exemple :**

```
if [ "$(getconf POSIX2_C_DEV)" -eq 1 ]; then echo "langage C disponible"
fi
```

## 5.4- Cas particulier des liens symboliques.

Le lien symbolique est un procédé pour pointer vers un fichier ou un répertoire. Il

offre en outre la possibilité de s'affranchir du système de fichier contrairement au lien classique.

On comprend bien que le lien symbolique résoud de nombreux problèmes. Par exemple, il est possible de simplifier les chemins d'accès dans l'arbre des fichiers.

**Exemple :** `ln -s /usr/spool/lpd/qdir /etc/qdir`

Au niveau portabilité, on constate hélas, un comportement dépendant du type de *shell* que l'on utilise. Pour cela, examinons le comportement face aux commandes `ls -l`, `cd` et `pwd`.

`ls -l :`

Pour tous les *shell*, `ls -l` indique en premier caractère un "l" pour indiquer qu'il s'agit d'un lien symbolique. L'emplacement réservé habituellement à la taille du fichier indique le nombre de caractères de l'objet pointé.

**Exemple :** `ln -s /usr/spool/lpd/qdir /etc/qdir`

`% cd /etc`

`% ls -l`

`lrwxrwxrwx owner group 19 date qdir->/usr/spool/lpd/qdir`

Certaines implémentations du *Bourne-shell* distinguent les deux cas où le lien pointe sur un objet existant ou non. Si l'objet n'existe pas, un "l" apparaît (comme ci-dessus) suivi des attributs du lien. Si l'objet existe, le type de l'objet pointé apparaît, ainsi que ses attributs habituels (droits, taille,...). Notons que ce comportement est assez rare et que, dans la plupart des *shell*, les informations du lien symbolique apparaissent en permanence.

`drwxr-xr-x owner group 1024 date qdir ->/usr/spool/lpd/qdir`

`pwd :`

En *Bourne-shell* et *C-shell*, la commande `pwd` indique l'endroit physique pointé, c'est-à-dire le chemin par lequel on serait passé sans utiliser le lien symbolique.

**Exemple :**

`%cd /etc/qdir`

`%pwd`

`/usr/spool/lpd/qdir`

En *Korn-shell*, `pwd` indique le chemin indiqué par le lien symbolique.

**Exemple :**

`%cd /etc/qdir`

`%pwd`

`/etc/qdir`

On comprend aisément le type d'erreurs que cela peut engendrer dans les *shell-script* utilisant la commande `pwd` pour obtenir les informations. Par exemple, construction de noms de fichiers, comparaison de paramètres par rapport au répertoire courant,

etc...

cd :

Le comportement de cd se déduit de celui de pwd.

*Bourne-shell et C-shell :*

```
%cd /etc/qdir
```

```
%cd ... -> usr/spool/lpd
```

*Korn-shell*

```
/cd /etc/qdir
```

```
%cd .. ->/etc
```

Le positionnement de la norme ISO 9945-2 est pour l'instant de ne pas implémenter les liens symboliques.

---

## 6- Le Futur

Contenu de ce chapitre

### 6.1- Evolution de la norme

La norme ISO/IEC 9945-2 comprend dans son annexe H un ensemble d'évolutions prévues, mais il n'y a aucune garantie de voir celles-ci aboutir. Néanmoins, citons les principales :

- Interfaces associées à P1003.1 comme les liens symboliques.
- Le classement (tri) sur caractères pourra faire référence (via LC\_COLLATE) à une autre "locale" grâce au mécanisme de remplacement avec *replace-after*.
- Les jeux de caractères étendus (16 et 32 bits) et les variantes nationales sur jeux de caractères ISO 646:1983 seront pris en compte à tous les niveaux; et probablement par le *shell* et les utilitaires, *awk*, *bc*, *lex*, *make* et *yacc* , ainsi que par les expressions régulières.

Les tests de conformité sont définis dans le cadre du projet CTS-5 (*Conformance Testing Service*) sous le contrôle d'X/Open. Une "suite de tests" appelée XCUTS (*X/Open Commands and Utilities Test Suite*) est en cours de développement, et doit observer les exigences de IEEE 1003.3 (Méthodes de tests pour mesurer la conformité à POSIX).

### 6.2- Evolution du marché

Comme nous l'avons vu au cours des chapitres précédents, la norme au niveau du langage de commande est très conservatrice, privilégiant la compatibilité avec l'ancêtre historique qu'est le *Bourne-shell* (*sh*), et adoptant quelques nouveautés héritées du *Korn-shell* (*ksh*). Ce dernier peut être considéré comme un surensemble

par rapport à la norme. Les *shell* conformes à la norme seront des réadaptations du *shell* existant le plus proche de celle-ci qu'est le *Korn-shell*.

Le savoir-faire pour développer de nouveaux *script* portables sera celui des utilisateurs actuels du *Korn-shell*, moyennant les restrictions évoquées au 3.2.2. Pour les *script* existants, seuls ceux écrits en *Bourne-shell* seront faciles à maintenir, quand ils ne s'exécutent pas tels quels, sous réserve de ne pas faire appel à des utilitaires notablement modifiés par la norme.

Le monde de l'offre va avoir le beau rôle de mettre en avant son implémentation, éventuellement réadaptée, du *Korn-shell* en tirant parti du fait qu'il va au-delà de cette norme. Comme tous les fournisseurs ont déjà un *Korn-shell* disponible, cela sera très rapide et, ainsi, le *Korn-shell* risque de devenir un standard de fait surensemble de la norme.

---

## 7- Tableau de filiation

Contenu de ce chapitre

### 7.1- Terminologie (utile) des étapes de normalisation

Les termes suivants sont classés dans l'ordre chronologique des étapes de normalisation.

NP New Proposal

WD Working Draft

CD Committee Document

DIS Draft International Standard

IS International Standard

### 7.2- Tableau

N° IEEE actuel	N° IEEE ancien	Description	Statut	Norme Int.	Prévision
P1003.2 P1003.2a	-	Utilitaires et langage de commande	IS	ISO/IEC 9945-2:1993	-
P1003.2b	-	Révision de la norme 9945-2	WD	-	-
P1..3.2c	P1003.6.2	Extension de la norme afin de prendre en compte le traitement non interactif (batch)	WD	-	CD début 1994
P1003.2d	P1003.15		WD	-	CD début 1994

NB. L'IEEE a été mandaté par l'ISO afin de définir l'ensemble des documents de travail pour la normalisation des interfaces du système d'exploitation portable (POSIX).