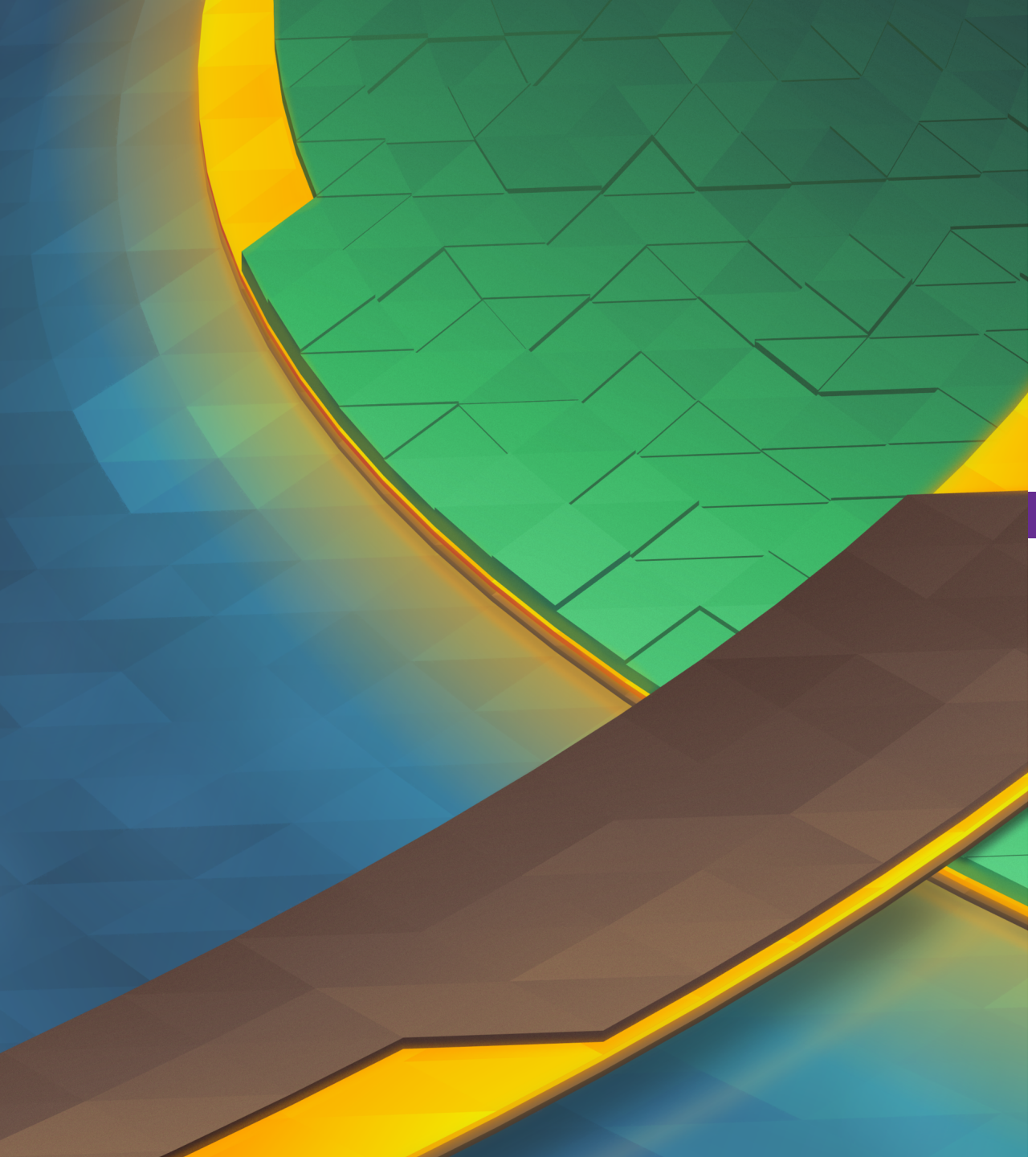
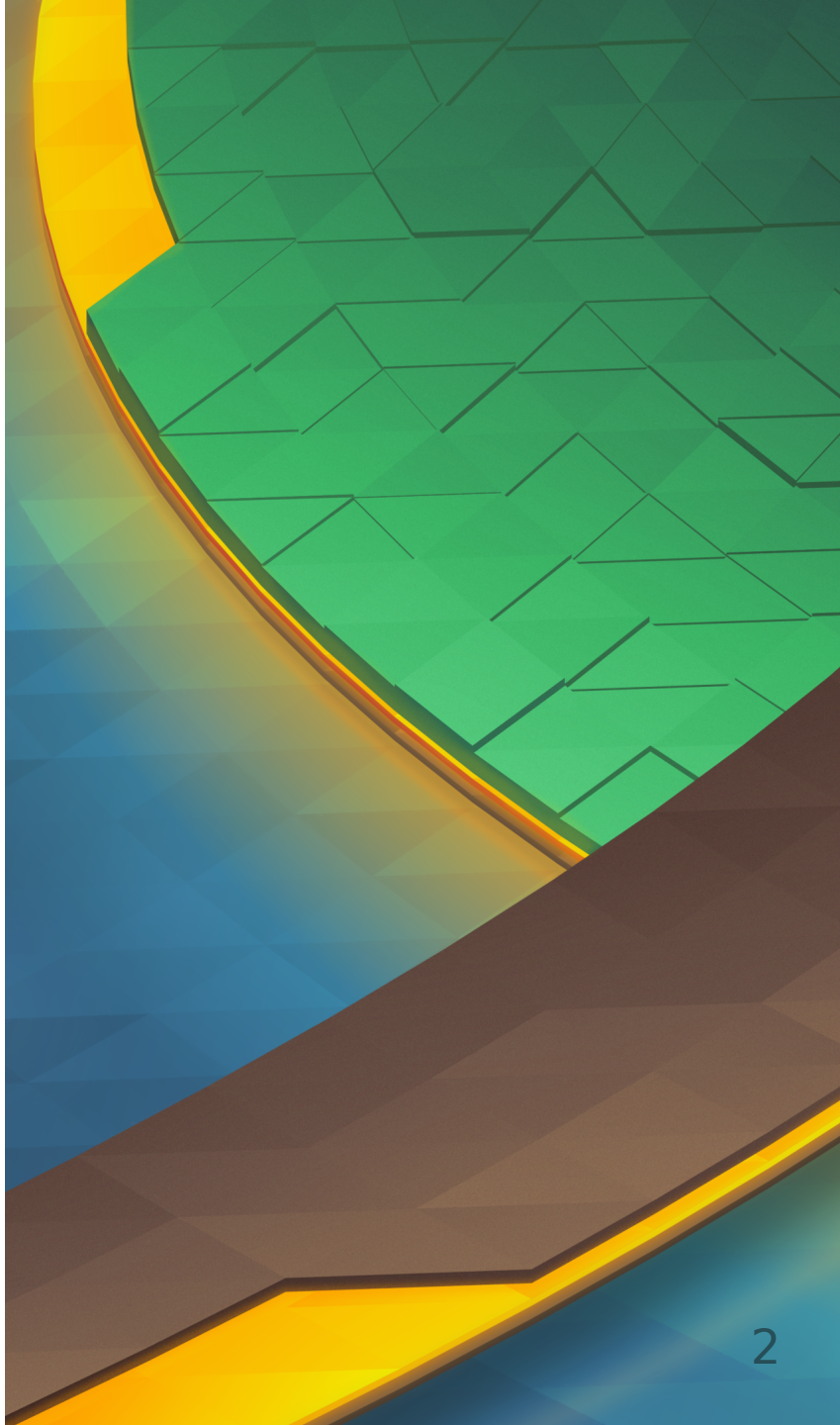


Unix/Linux scripting

Glenn Y. Rolland <teaching@glenux.net>

Préambule



Objectifs de la formation

- Connaître les différentes instructions utilisables dans un script shell
- Maîtriser les entrées/sorties étendues du shell
- Manipuler les chaînes de caractères
- Enrichir vos script à l'aide de getopts
- Effectuer le debugging d'un script shell
- Créer une bibliothèque de fonctions simples
- Debugger et analyser des scripts shell

Qui êtes vous ?

Petit tour de présentation... avec 3 questions

- Quel est votre "bagage" ? (expérience, compétences, etc.)
- Pourquoi participez vous à cette formation aujourd'hui ?
- Comment utiliserez-vous ces nouvelles compétences d'ici 2 ou 5 ans ?

Qui suis-je ?

2021 → aujourd'hui	**Auteur, conférencier et co-fondateur de CRYPTO-CHEMISTS** Formation et conseil sur l'impact des technologies P2P et des blockchains.
2018 → aujourd'hui	**Directeur technique et co-fondateur de BOLDCODE** Audit logiciel (qualité, sécurité, licences logicielles) et développement de logiciel, d'infrastructures, de systèmes et de réseaux. Offshoring éthique au Népal.
2017 → 2022	**Directeur technique et co-fondateur de DATA-TRANSITION** Gestion éthique des données, audit des SI, conformité au RGPD.
2010 → 2017	**Gérant et co-fondateur de NETCAT (GNUSIDE)** Audit et développement d'infrastructures, de systèmes et de réseaux. Optimisation de la fiabilité, de la sécurité et de la performance. Conception et fabrication d'un produit (routeur multiwan avec vpn autoconfiguré).
2006 → 2010	**Ingénieur de recherche chez BEWAN (Pace Group)** Conception de systemes embarqués, spécialité système & réseaux télécoms. Développement d'un SDK. Gestion et automatisation de la qualité logicielle.

La formation chez Orsys - en quelques mots

ORSYS formation

Learning for success

- 45 ans d'existance
- 7700 entreprises et administrations clientes
- 85000 personnes formées par an
- 1700 intervenants experts
- 97.5% de clients satisfaits
- 27 centres de formation en France
- 5000+ sessions intra-entreprise
- 7000+ sessions inter-entreprise
- 72,2M€ de chiffres d'affaires

Note: Chiffres de 2019

Déroulement de la formation & règles du jeu

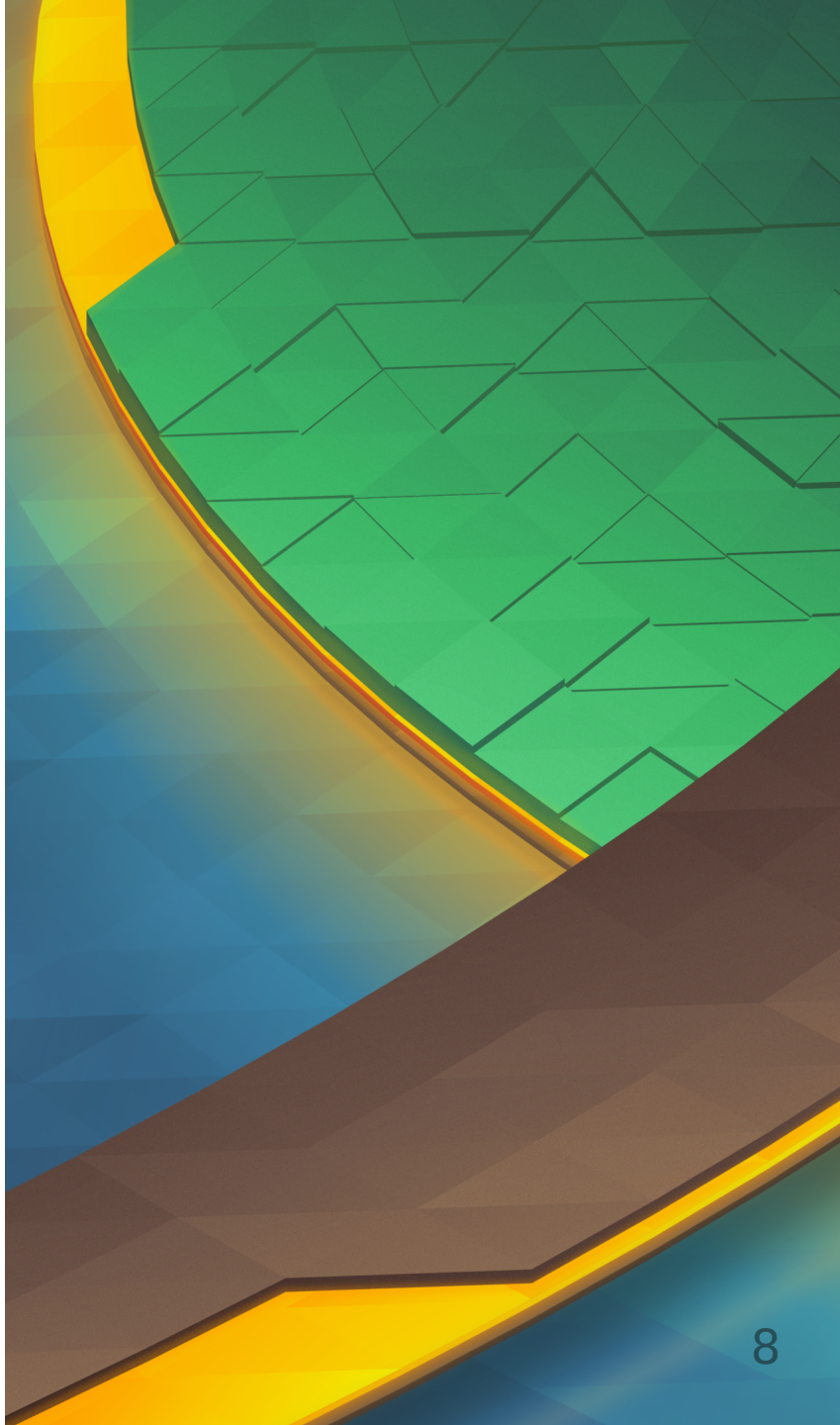
Horaires

- 9h00 - 12h30
- 13h30 - 17h00
- Des pauses le matin et l'après midi

Le cadre

- Liberté de parole dans le respect des autres et des objectifs de la formation
- Bienveillance, nous sommes dans un espace d’apprentissage
- Confidentialité de l’animateur et des participants sur les échanges

Introduction au scripting



Le shell, les scripts et les chaînes de caractères

Shell

Le shell est le programme qui...

- interprète les commandes entrées par l'utilisateur,
- interprète les commandes stockées dans un script shell (fichier),
- exécute les fonctions correspondantes,
- gère aussi les variables d'environnement,
- gère les flux d'entrées / sortie (chaînes de caractères)

Scripts

Les scripts sont des fichiers qui...

- contiennent des instructions (internes au shell)
  - ex: instructions conditionnelles, des boucles et des fonctions
- contiennent des commandes (programmes du système)
  - ex: ls, mkdir, etc.
- exécutent séquentiellement les instructions et les commandes contenues dans le fichier
- peuvent être créés dans n'importe quel éditeur de texte.
  - ex: vim, emacs, nano, etc.

Chaînes de caractères

Une suite de caractères pouvant être manipulée par le shell.

- C'est le format de base des données d'un shell dans les systèmes unix.
- Utilisées pour manipuler du texte, des nombres, etc.

Attention aux chaînes de caractères

Les chaînes de caractères doivent être traitées avec précaution car elles peuvent être très difficiles à débugger.

- Utiliser des guillemets autour des chaînes
  - Pour éviter les problèmes avec les caractères spéciaux et les espaces dans la chaîne.
  - L'oubli peut causer des problèmes avec des commandes telles que `echo` ou `rm`.
- Utiliser `${variable}` est préférable à `$variable`
  - Garantit la référence à la bonne variable
  - L'oubli peut causer des erreurs de substitution

Rappels des fondamentaux

Exécution d'un script

Soit avec l'interpréteur et le script en paramètre

```
sh nom_fichier
```

Soit rendre le fichier exécutable puis taper le nom du fichier

Pour que le système comprenne qu'il faut exécuter le script avec un shell, le fichier doit commencer par `#!/bin/sh` (ou `#!/usr/bin/env xxx`, avec `xxx` le nom de votre shell)

```
chmod u+x nom_fichier
./nom_fichier
```

Modes d'execution spéciaux

Interprèter les commandes sans les exécuter

```
sh -n nom_fichier
```

Mode verbeux. Affiche ce qui est lu en entrée

```
sh -v nom_fichier
```

Trace d'exécution. Affiche les lignes telles qu'elles sont interprétées

```
sh -x nom_fichier
```

Fork & Wait

- **fork** : créer un processus enfant à partir d'un processus parent. Ce processus enfant est une copie exacte de son parent à l'instant où il a été créé.
- **wait** : indiquer au processus parent d'attendre la fin de l'exécution du processus enfant.

Exec & Exit

- **exec** : exécuter un programme en remplaçant le processus parent.
- **exit** : terminer le processus courant.

Propriétés

- **processus** : un programme en cours d'exécution:
  - **pid** : identifiant de processus
  - **ppid** : identifiant du processus parent
  - **pgid** : identifiant du groupe de processus
  - **environement**

Code de retour

Un processus qui s'arrête fournit une valeur (*code de retour*) qui indique comment l'exécution s'est déroulée.

- **0** : le processus s'est déroulé correctement,
- **autre valeur** : il y a eu un problème (ou un cas spécial).

Rappels sur le VFS et les fichiers ouverts (vus par le kernel)

Les descripteurs (fichiers vus par les processus)

- **descripteurs** : numéro des fichiers ouverts attachés aux processus:
  - **stdin** : descripteur d'entrée standard
  - **stdout** : descripteur de sortie standard
  - **stderr** : descripteur d'erreur standard

Traitement des commandes avant l'exécution

- Les commandes sont lues et évaluées par le shell avant d'être exécutées
- Le shell évalue le globbing
- Le shell substitue les variables
- Le shell gère les flux (pipes, descripteurs, etc.)

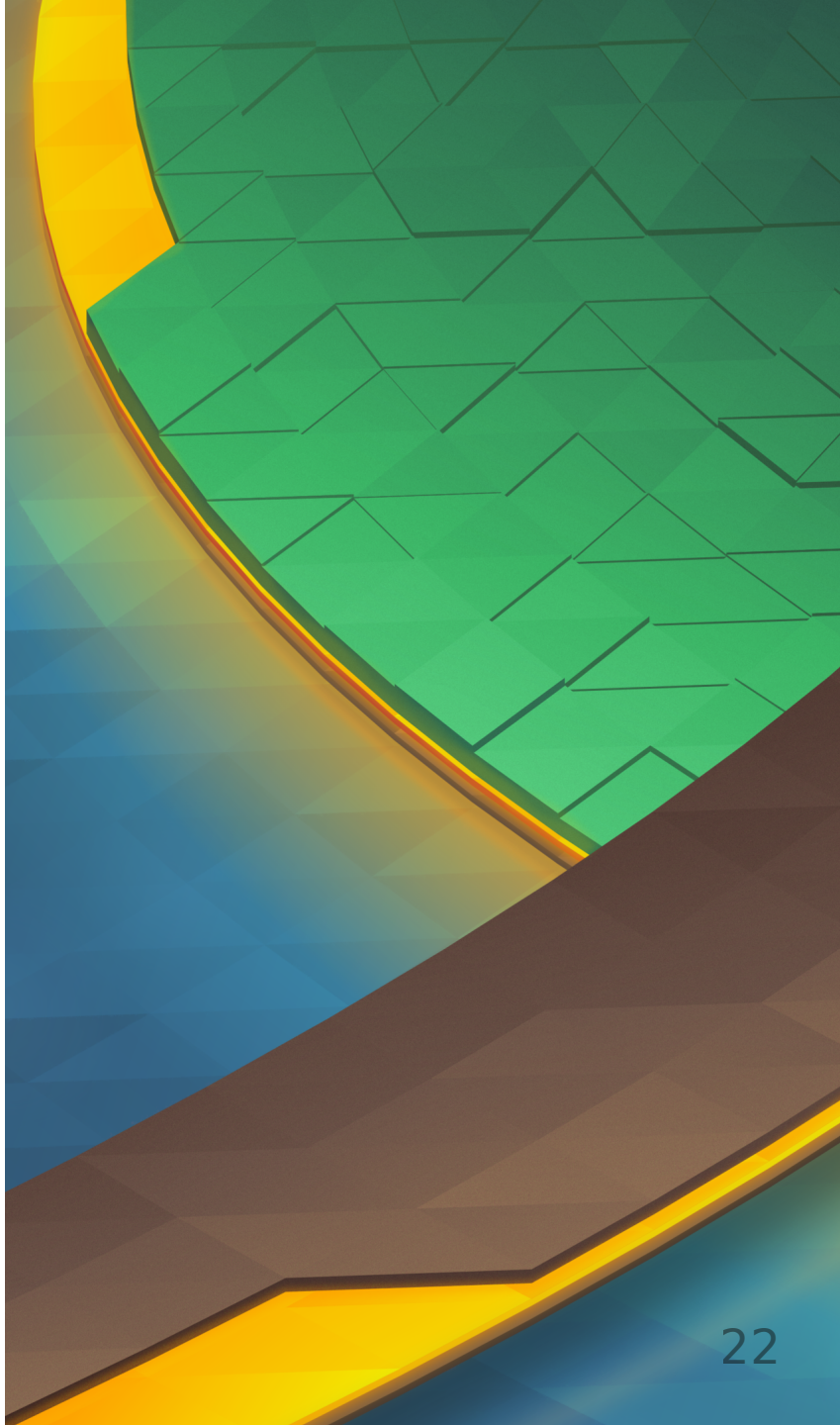
... avant de lancer les commandes !

Globing (rappels)

`*`	n'importe quelle chaîne de caractères
`?`	n'importe quel caractère
`[ ... ]`	n'importe quel caractère décrit entre les crochets

Le globbing est évalué par le shell AVANT l'exécution des commandes.

Spécificités du Korn shell



Les différents types de shell

Bourne Shell (sh)

- Un des premiers interpréteurs de commandes pour les systèmes d'exploitation Unix.
- Il a été développé par Stephen Bourne (en ~1970)
- Largement utilisé jusqu'à ce qu'il soit remplacé par des interpréteurs de commandes plus avancés
  - ex: C shell (csh), Bourne-Again shell (bash), etc.

C-shell (csh)

- Un shell Unix qui a été développé à la fin des années 1970.
- Créé comme une alternative au shell Unix original, le Bourne shell (sh)
- Conçu pour fournir une interface de ligne de commande plus interactive et plus conviviale.
- Principalement utilisé dans la branche BSD d'Unix (ex : FreeBSD, OpenBSD, et NetBSD).
- Caractéristiques
  - Edition interactive de la ligne de commande (éditer et modifier ses commandes avant qu'elles ne soient exécutées)

TENEX C-shell (tcsh)

- Version améliorée du csh qui a été développé au milieu des années 80.
- Comprend des fonctionnalités supplémentaires et des améliorations par rapport à csh,
  - Meilleure édition de la ligne de commande, manipulation étendue de l'historique et contrôle des tâches.
  - Complétion de la ligne de commande, correction automatique de l'orthographe et liste automatique des complétions possibles
- Encore utilisé par de nombreux utilisateurs d'Unix aujourd'hui

Bourne-Again Shell (bash)

- Interpréteur de commandes compatible avec sh,
- Développé comme amélioration du Bourne shell par Brian Fox pour le projet GNU (en 1989),
- Devient le shell par défaut pour la plupart des distributions Linux,
- Est rapidement porté vers d'autres systèmes d'exploitation tels que macOS.
- Fonctionnalités en plus (par rapport à sh)
  - Historique des commandes, complétion automatique de commande,
  - Structures de contrôle de flux plus avancées,
  - Tableaux, typage de variables, variables locales,
  - Des fonctions avec une sémantique plus riches,

Korn shell (ksh)

- Interpréteur de commandes Unix/Linux
- Conçu pour être plus riche que le Bourne Shell (sh) et le Bourne Again Shell (bash).
- Systèmes de shell parmis les plus populaires notamment sur AIX (IBM)
- Fonctionnalités supplémentaires
  - Historique des commandes, complétion automatique de commande,
  - Structures de contrôle de flux plus avancées,
  - Tableaux, typage de variables, variables locales, variables complexes
  - Des fonctions avec une sémantique plus riches,
  - Gestion avancée des processus, techniques de débogage avancées.

Z Shell (zsh)

- Interpréteur de commandes Unix/Linux.
- conçu par Paul Falstad en 1990.
- Shell de choix pour de nombreux utilisateurs
  - en raison de ses fonctionnalités avancées et de sa personnalisation flexible.
  - en raison de sa licence permissive (MIT)
- Fonctionnalités supplémentaires par rapport à Bash & Ksh
  - Meilleure complétion des commandes
  - Personnalisation plus poussée de l'interface utilisateur,
  - Options d'édition de commandes, la substitution de commandes,
  - L'expansion de noms de fichiers,
  - Configuration plus sûre, les fonctionnalités dangereuses sont désactivées par défaut,
  - Mécanisme de rappel de commandes puissant, qui permet aux utilisateurs de modifier et de réutiliser facilement les commandes précédemment exécutées.

Autres shells notables

- Ash (Almquist shell) - used by Busybox.
- Dash (Debian Almquist shell) - used by Debian
- PowerShell - used by Microsoft Windows
- Fish (Friendly Interactive SHell)
- etc.

POSIX

La norme POSIX (Portable Operating System Interface) définit l'interface de programmation pour les applications shell sous les systèmes d'exploitation UNIX et UNIX-like.

- POSIX spécifie les commandes, les variables d'environnement, les fichiers et les flux de données, ainsi que les règles de syntaxe pour les commandes et les scripts shell.
- POSIX fournit une base commune pour les commandes et les scripts shell.
- POSIX permet la portabilité des scripts entre différents systèmes d'exploitation (sans modifications)
- POSIX définit des fonctionnalités telles que les expressions régulières et le fonctionnement des flux(pipes, etc.)
- POSIX a été adoptée comme norme ISO/IEC 9945 dans les années 1990.

🟦 Wikipedia: POSIX

Bashismes, fonctionnalités avancés et compatibilité

Presque tous les shells modernes prennent en charge des extensions de syntaxe qui ne sont pas compatibles ni avec le Bourne shell d'origine, ni avec POSIX, ni entre eux.

Il faut donc faire extrèmement attention à ce que vous trouvez comme "solutions" en ligne (ou dans les livres)

> L'utilisation de la norme POSIX pour le shell est recommandée pour assurer la portabilité et la compatibilité de vos scripts entre différents systèmes d'exploitation.

Note: Pour repérer facilement et ainsi éviter les bashismes et incompatibilités, vous pouvez utiliser l'outil `shellcheck`.

📘 https://www.shellcheck.net/

Configuration de ksh

Initialisation de l'environnement

Le Korn Shell permet l'initialisation de l'environnement et permet d'utiliser des scripts afin de spécifier les variables d'environnement, les fonctions et les commandes qui seront disponibles. Il est également capable d'utiliser des fichiers de configuration pour les personnaliser.

Pour initialiser l'environnement, le Korn Shell utilise les fichiers de configuration suivants :

- `~/.login` pour le login
- `~/.profile` pour le login
- `~/.kshrc` pour les paramètres personnalisés de l'utilisateur
- `/etc/ksh.kshrc` pour les paramètres système

Personnalisation de la ligne de commande

Le Korn Shell permet également de personnaliser la ligne de commande en modifiant les paramètres de l'environnement.

`PS1`	Prompt système primaire (début de ligne de shell)
`PS2`	Prompt système secondaire (continuation de ligne)

Par exemple

```
user@machine ~/src $ echo "$PS1"
$USER@$(hostname) ~${PWD##/home/user} $
```

```
user@machine ~/src $ echo "$PS2"
>
```

Mode Vi et Emacs

Il prend en charge les modes Vi et Emacs pour l'édition des commandes et est capable de sauvegarder et restaurer les commandes en cours d'exécution.

Pour le mode vi

```
set -o vi
```

Pour le mode emacs

```
set -o emacs
```

Raccourcis pour le mode Emacs

CTRL-A	Déplace le curseur au début de la ligne
CTRL-E	Déplace le curseur en fin de la ligne
CTRL-F	Déplace le curseur à droite d'un caractère
ALT-F	Déplace le curseur à droite d'un mot
CTRL-B	Déplace le curseur à gauche d'un caractère
ALT-B	Déplace le curseur à gauche d'un mot
CTRL-D	Efface le caractère sous le curseur2
ALT-D	Supprime le mot à droite du curseur
CTRL-H	Supprime le caractère à gauche du curseur

ALT-H	Supprime le mot à gauche du curseur
CTRL-K	Supprime tous les caractères à droite du curseur
CTRL-U	Supprime tous les caractères de la ligne en cours
CTRL-T	Transpose les deux caractères immédiatement à gauche du curseur
ALT-C	Convertit le mot suivant en majuscules
CTRL-L	Redessine la ligne en cours (n'efface pas l'écran comme sur bash)

Alias

Le Korn Shell prend également en charge les alias, qui sont des abréviations pour les commandes et les fonctions. Les alias peuvent être utilisés pour simplifier les commandes et les fonctions complexes.

Lister les alias

```
alias
```

Initialiser un nouvel alias

```
alias nom=valeur
```

Supprimer un alias

```
unalias nom
```

Alias (suite)

Les alias suivants sont définies par défaut par le shell

```
autoload='typeset -fu'
false='let 0'
functions='typeset -f'
hash='alias -t'
history='fc -l'
integer='typeset -i'
nohup='nohup '
r='fc -e -'
true=':'
type='whence -v'
```

Quelques variables d'environnement utiles

HOME	le home directory (répertoire de login)
PATH	chemin de recherche pou l'exécution des commandes
CDPATH	chemin de recherche pour la commande cd
MAIL	chemin indiquant le répertoire du courrier
IFS	internal field separator
SHELL	indique le shell de login

Les variables (rappels)

Introduction aux variables

- Les variables sont utilisées pour stocker des informations dans le shell.
- Elles permettent d'adapter le comportement du script à différentes conditions.
- En Ksh et Bash, les variables sont non typées : elles peuvent contenir des chaînes de caractères, des nombres ou des références à des fichiers.

Déclaration et utilisation

`variable=valeur`	affectation (Attention , ne pas mettre d'espace autour de =)
`$variable`	valeur de la variable
`${variable}`	valeur de la variable

Ex: si notre shell connait seulement `a="var"` alors `${a}b` renvoie `varb` alors que `$ab` est invalide.

Les variables d'environnement, qui sont disponibles dans tout le shell, sont définies en utilisant le mot-clé `export`, comme dans `export PATH=/usr/local/bin:$PATH`.

Paramètres dans la ligne de commande

`$0`	nom de la commande
`$n`	nieme paramètre
`$#`	nombre de paramètres
`$*` ou `$@`	liste de tous les paramètres

La signification de `$*` et de `$@` est identique, mais:

- `$*` est équivalent à `"$1 $2 "`
- `$@` est équivalent à `"$1" "$2"`

Pour décaler les paramètres, on peut utiliser la commande *shift*

Quelques variables spéciales

`$$`	le numéro de processus de la dernière commande exécutée
`$?`	Status (code de retour) de la dernière commande

Les caractères spéciaux

`\`	banalise le caractère suivant (ex: `\"`)
`" ... "`	banalise les caractères sauf `\` , `$` et `
`' ... '`	banalise tous les caractères (les variables ne sont donc pas évaluées)
`...`	substitution de commande

La gestion des fonctions dans ksh

Déclaration d'une fonction

Syntaxe

```
function nom_de_la_fonction {
  declaration_des_variables

  instruction ou commande
  # ...
  instruction ou commande

  return ...
}
```

La liste est composée de deux partie : la déclaration de variables et les commandes

Note: On peut faire des fonctions recursives.

Valeur de retour

- On positionne le status de la fonction par `return` (récupéré par `$?`)
- Indique seulement la bonne exécution (ou non)

Résultat

On renvoie une valeur par l'intermédiaire du STDOUT de la fonction (commande `echo` ou `printf`), qui est ensuite récupéré par :

```
resultat=$(appel_de_la_fonction)
```

ou

```
resultat=`appel_de_la_fonction`
```

La gestion des signaux

Réception de signaux

La commande `trap` permet de spécifier le comportement du shell à la réception d'un signal (indiqué par son nom ou son numéro).

Syntaxe

```
trap handler nsig1 nsig2
```

- A la réception d'un signal la commande handler est exécutée
- Puis le shell reprend sont exécution au point de réception du signal.
- Dans le cas de reception simultané de plusieurs signaux le korn shell exécute les handler dans l'odre défini par les numéros de signaux.

Exemple

```
# Ignorer les signaux intr (2) et quit (3)
trap "" 2 3
```

Retour au comportement par défaut

Syntaxe

```
trap - sig
```

Note: la gestion des signaux est hierarchique dans Unix (= signaux passés de pere en fils). Si un signal était ignoré par le shell a son lancement il n'est plus possible de positioner un handler sur se signal!

Fin de programme

On peut détecter la fin d'un script avec:

```
trap handler EXIT
```

la commande handler est réalisé a la fin du script quelle que soit la raison de terminaison (excepté SIGKILL).

Note: Si le trap est défini dans une fonction le handler est exécuté a la fin de la fonction.

📘 Putorius: Using Trap to Exit Bash Scripts Cleanly

Signaux spéciaux

En plus des signaux Unix (POSIX), le korn shell introduit des signaux spéciaux:

- DEBUG le handler est exécuté après chaque commande
- ERR le handler est exécuter après chaque commande de code de retour non nul.

`trap` sans argument affiche la liste des handler associés a chaque signal

Le signal 1 (HUP) de fin de session

Le shell propriétaire d'une session envoie quand il meurt un signal 1 à tous les processus de la session afin que vous soyez délogé proprement. Mais si vous voulez lancer un processus qui doit tourner pendant votre absence il faut utiliser la commande nohup .

```
nohup commande
```

Ainsi le processus créé ignore le signal SIGHUP de fin de session.

Il est également possible pour un script de se protéger avec

```
trap "" 1.
```

Debug d'un script

Concept

- Le debugging est une étape essentielle pour assurer la qualité d'un script.
- Il permet d'identifier et de corriger les erreurs ou bugs dans le code.
- En Korn Shell et Bourne Again Shell, il existe plusieurs options et outils pour faciliter le debugging.

Erreurs courantes

- Erreurs de syntaxe: fautes de frappe, erreurs de ponctuation.
- Erreurs logiques: erreurs dans la logique du script, entraînant un comportement imprévu.
- Erreurs de portabilité: problèmes liés à l'exécution du script sur différents systèmes ou shells.
- Les erreurs liées à l'espace de noms des variables: utiliser une variable non déclarée ou mal initialisée.

Options avec `set -o optionname`

`set -o optionname` est une commande utilisée dans les shells Unix, comme Bash et Korn Shell (ksh), pour contrôler divers paramètres et comportements du shell.

L'option que vous spécifiez après `-o` détermine ce que vous modifiez.

Pour désactiver une option, vous pouvez utiliser `set +o optionname`.

📘 https://docstore.mik.ua/orelly/unix3/korn/ch09_01.htm

Aide basique au débugging (2)

NoUnset

```
set -o nounset
```

- Equivalent à `set -u`
- Cela génère une erreur si vous essayez d'utiliser une variable qui n'a pas été définie.

NoExec

```
set -o noexec
```

- Cette option est similaire à `set -n`.
- N'exécute pas les commandes
- Vérifie seulement les erreurs de syntaxe

Aide basique au débugging (3)

ErrExit

```
set -o errexit
```

- Equivalent à `set -e`
- Avec cette option, le shell arrêtera l'exécution du script si une commande se termine avec un statut non nul (c'est-à-dire une erreur).

Pipefail

```
set -o pipefail
```

- Avec cette option un pipeline renvoie un statut de sortie non nul si l'une quelconque des commandes du pipeline échoue.

Aide basique au débugging (4)

Xtrace

```
set -o xtrace
```

- Cette option est identique à l'utilisation de la commande `set -x`.
- Elle imprime chaque commande avant son exécution, ce qui est utile pour le débogage.

Verbose

```
set -o verbose
```

- Cette option est similaire à `set -v`.
- Comme `xtrace`, elle imprime également les commandes avant leur exécution,
- Cependant elle n'imprime pas les valeurs des variables d'environnement et elle n'évalue pas les substitutions de commandes.

Trap DEBUG

- En KSH et BASH, *trap* est une fonction qui permet de capturer des signaux et d'exécuter un bloc de code lorsqu'ils sont reçus.
- DEBUG est un signal particulier qui est émis avant chaque commande exécutée.
- L'utilisation de *trap DEBUG* permet de suivre l'exécution du script et d'injecter du code de débogage à la volée.
- Pour utiliser *trap DEBUG*, ajoutez une ligne comme `trap echo $BASH_COMMAND DEBUG` dans votre script. Cela affichera chaque commande avant qu'elle ne soit exécutée.

Introduction au Korn Shell Debugger

- Le Korn Shell Debugger (kshdb) est un outil de débugging dédié aux scripts écrits en Korn Shell.
- Il permet un contrôle plus précis de l'exécution des scripts par rapport aux outils de débogage de base.
- Il offre des fonctionnalités similaires à gdb (GNU Debugger), mais spécifiques à l'environnement ksh.

ℹ️ Il existe des outils équivalents pour Bash et pour Zsh.

📘 O'Reilly - Korn Shell Chapter 9: Debugging

📘 Github: Kshdb

Installation et configuration du Korn Shell Debugger

- Pour l'installation, utiliser la commande `git clone https://github.com/rocky/kshdb.git` pour cloner le dépôt GitHub de kshdb.
- Pour configurer kshdb, définir un alias pour l'outil en ajoutant `alias kshdb='path_to_kshdb'` dans le fichier de configuration de votre shell (`.kshrc` pour Korn Shell).

L'utilisation de breakpoints

- Les breakpoints permettent de stopper l'exécution d'un script à des points spécifiques.
- Pour ajouter un *breakpoint*, utiliser la commande `kshdb script.ksh` puis `break linenumber` pour définir un *breakpoint* à une ligne précise.
- Pour lister les *breakpoints* actuels, utiliser la commande `info breakpoints`.
- Pour supprimer un *breakpoint*, utiliser la commande `delete linenumber`.

L'inspection de variables et l'exécution pas à pas (step by step)

- Pour inspecter la valeur d'une variable, utiliser la commande `print variable_name`.
- Pour exécuter un script pas à pas, utiliser la commande `step` pour exécuter la ligne suivante ou `next` pour exécuter la ligne suivante sans entrer dans les fonctions.
- Pour continuer l'exécution jusqu'au prochain breakpoint, utiliser la commande `continue`.

Gestion des erreurs avec le Korn Shell Debugger

- En cas d'erreur dans le script, le débogueur s'arrête et montre la ligne qui a causé l'erreur.
- Pour obtenir plus d'informations sur l'erreur, utiliser la commande `backtrace`.
- Pour quitter le débogueur en cas d'erreur, utiliser la commande `quit`.
- Pour reprendre l'exécution après correction de l'erreur, utiliser la commande `continue`.

Programmation orientée objet en shell

Types définis par l'utilisateur.

Définition

- Peuvent être défini soit par une bibliothèque partagée,
  - soit au moyen de la nouvelle commande de déclaration typeset -T,
  - soit au moyen de la commande de déclaration enum.
- Fournissent un moyen de déclarer et d'instancier des objets
  - peuvent contenir des données (éléments),
  - peuvent contenir des méthodes (fonctions contraintes)

Exemple

```
typeset -T Point_t=( ... )
```

Note: Par convention, les noms de types commencent par une majuscule et se terminent par _t.

Création de types et d'instances

Une instance d'un type est créée en invoquant le nom du type suivi d'un ou plusieurs noms d'instance.

```
Point_t point
```

Lorsqu'un type est défini, une commande intégrée spéciale de ce nom est ajoutée à la liste des commandes intégrées que ksh93 connaît. Les définitions de type sont en lecture seule et ne peuvent pas être annulées.

📘 Using Types To Create Object Orientated Korn Shell 93 Scripts

Exemple

```
#!/usr/bin/env ksh93
typeset -T Point_t=(
     integer -h 'x coordinate' x=0
     integer -h 'y coordinate' y=0
     typeset -h 'point color'  color="red"

     function getcolor {
          print -r ${_.color}
     }

     function setcolor {
          _.color=$1
     }

     setxy() {
          _.x=$1; _.y=$2
     }

     getxy() {
          print -r "(${_.x},${_.y})"
     }
)
```

Exemple (suite)

```
Point_t point

echo "Initial coordinates are (${point.x},${point.y}). Color is ${point.color}"

point.setxy 5 6
point.setcolor blue

echo "New coordinates are ${point.getxy}. Color is ${point.getcolor}"
```

Résultat

```
$ ./example1
Initial coordinates are (0,0). Color is red
New coordinates are (5,6). Color is blue
```

Définition

Un *getter* est une fonction qui est appelée quand un certaine variable est lue.

```
function demo.get {
  .sh.value="3"
  echo "Getter called"
}
```

Exemple

```
$ echo $demo
Getter called
3
```

Définition

Un *setter* est une fonction qui est appelée quand un certaine variable est modifiée.

```
function demo.set {
  echo "Setter called"
  echo "${.sh.value}"
}
```

Exemple

```
$ demo=3
Setter called
3
```

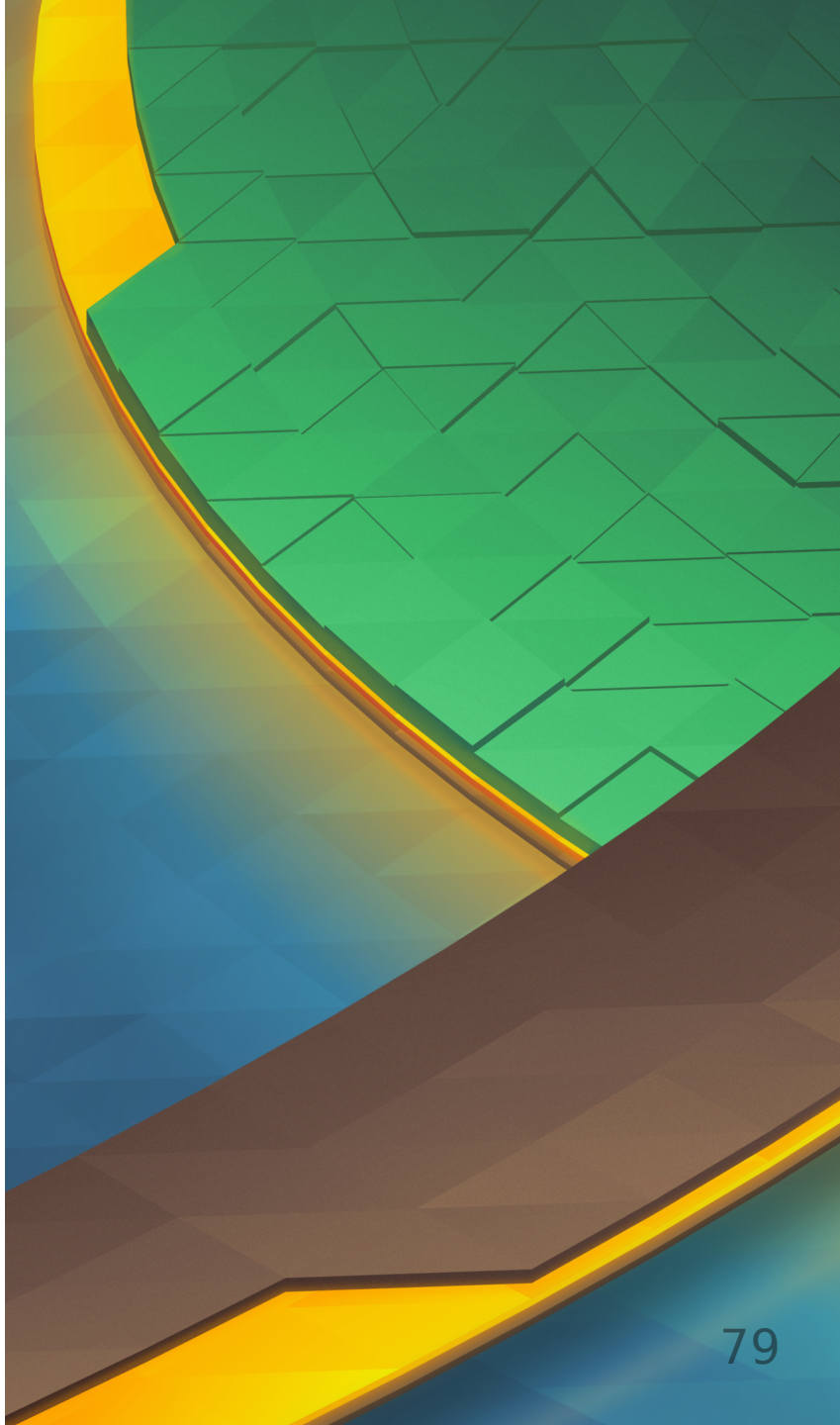
Documentation auto-générée

```
$ Point_t --man
NAME
  Point_t - set the type of variables to Point_t

SYNOPSIS
  Point_t [ options ] [name[=value]...]

DESCRIPTION
  Point_t sets the type on each of the variables specified by name to Point_t. If =value is
  specified, the variable name is set to value before the variable is converted to Point_t.

  If no names are specified then the names and values of all variables of this type are
  written to standard output.

  Point_t is built-in to the shell as a declaration command so that field splitting and
  pathname expansion are not performed on the arguments. Tilde expansion occurs on
  value.

OPTIONS
  -r              Enables readonly. Once enabled, the value cannot be changed or unset.
  -a[type]        Indexed array. Each name will converted to an index array of type Point_t. If
                  a variable already exists, the current value will become index 0. If [type] is
...
```

Travaux pratiques

- Adapter son environnement pour ksh
- Editer le `~/.profile`
- Mode d'édition de commande
- Utilisation de la trap DEBUG et des setter/getter pour analyser un script

Variables, expressions arithmétiques et logiques



Variables et operateurs

Expressions arithmétiques

- En Ksh et Bash, les expressions arithmétiques sont généralement écrites entre doubles parenthèses, comme dans `((a = b * 2))`.
- Les opérations arithmétiques standard, telles que l'addition (+), la soustraction (-), la multiplication (*) et la division (/), sont supportées.
- Le shell permet également des opérations plus avancées, telles que l'augmentation (++) et la réduction (--), ainsi que des opérations bit à bit.

Expressions logiques

- Les expressions logiques sont utilisées dans les structures de contrôle, comme les instructions if, while et until.
- Les opérateurs logiques standards sont `-eq` (égal), `-ne` (non égal), `-gt` (plus grand que), `-ge` (plus grand ou égal), `-lt` (moins que) et `-le` (moins ou égal).
- Les expressions peuvent être combinées avec les opérateurs logiques `&&` (et) et `||` (ou).

Types et typage des variables

Variables scalaires

- Variables scalaires : type le plus simple de variable.
- Déclaration dans ksh et bash : `varname=value` (pas d'espace autour du signe égal).
- Affichage de la valeur d'une variable : `echo $varname`.
- Les variables scalaires ne peuvent contenir qu'une seule valeur à la fois.

Variables d'environnement

- Les variables d'environnement sont accessibles à tous les processus de l'utilisateur.
- Pour définir une variable d'environnement dans ksh et bash : `export VARNAME=value`.
- Accéder à la valeur d'une variable d'environnement : `echo $VARNAME`.
- Les variables d'environnement courantes incluent `PATH`, `HOME`, `USER`, etc.

Variables normales, locales et globales

- Les variables (normales) ne sont accessibles que dans le shell ou le script où elles ont été définies.
- Les variables locales, sont accessibles seulement au sein de la fonction qui les a défini.
- Les variables globales, ou variables d'environnement, sont accessibles à tous les processus de l'utilisateur.
- Pour définir une variable (normale) : varname=value
- Pour définir une variable locale
  - dans POSIX: `varname=value`
  - dans bash : `local varname=value`
- Pour définir une variable globale dans ksh et bash : `export varname=value`

L'instruction typeset

Syntaxe

```
typeset [-a] [-i] [-r] [-x] var1 [var2 ?]
typeset
```

Les shells évolués (Korn Shell, Bourne Again Shell et autres descendants) permettent de restreindre les propriétés des variables et correspondent à une certaine forme de typage « simpliste ».

ℹ️ l'instruction `declare` accessible uniquement en Bourne Again Shell (et autres descendants) est un synonyme de l'instruction `typeset`.

L'instruction typeset (2)

Exemple d'usage

- `typeset -a var` : la variable sera traitée comme un tableau.
- `typeset -i var` : la variable sera traitée comme un entier et peut être incluse dans des opérations arithmétiques.
- `typeset -r var` : la variable sera mise en « lecture seule » (équivalent de « readonly »).
- `typeset -x var` :la variable sera exportée automatiquement dans les processus fils (équivalent de « export »).

Déclaration typée des variables

```
typeset var              | déclaration d'une chaîne de caractères
integer var              | déclaration d'un entier
typeset -i var           | déclaration d'un entier
typeset -r var = valeur | définition d'une constante
readonly var = valeur    | définition d'une constante
```

On peut utiliser des tableaux qui ne sont déclarés que lors de leur assignation: `tab[100]=toto`

Tableaux indexés et tableaux associatifs

Tableaux indexés - déclaration et assignation

Les tableaux peuvent contenir plusieurs valeurs indexées.

Déclaration "sur place"

```
students[1]=bob
students[0]=alice
students[2]=charlie
```

Déclaration avec set

```
set -A students alice bob charlie
```

Tableaux indexés - déclaration et assignation (2)

Déclaration composée (recommandée pour la compatibilité)

```
students=(alice bob charlie)
```

⚠️ les tableaux indexés sont limités à 4096 éléments

Tableaux indexés - utilisation

Obtenir une valeur

```
$ echo "${students[0]}"
alice
```

Obtenir toutes les valeurs

```
$ echo "${students[*]}"
alice bob charlie

$ echo "${students[@]}"
alice bob charlie
```

Obtenir toutes les clés

```
$ echo "${!students[0]}"
0 1 2
```

Tableaux associatifs - déclaration et assignation

Déclaration

```
typeset -A wavelength
```

Assignation de valeur

```
wavelength["red"]=650
wavelength["orange"]=590
wavelength["yellow"]=570
wavelength["green"]=510
wavelength["blue"]=475
wavelength["indigo"]=445
wavelength["violet"]=400
```

Déclaration et assignation (en même temps)

```
typeset -A wavelength
wavelength=([red]=650
            [orange]=590
            [green]=510
            [blue]=475
            [indigo]=445
            [violet]=400)
```

Tableaux associatifs - utilisation

Récupération d'une valeur

```
echo "${wavelength["red"]}"
650
```

Récupération de toutes les valeurs

```
echo "${wavelength[@]}"
475 400 590 510 650 570 44
```

Récupération de toutes les clés

```
echo "${!wavelength[@]}"
red orange yellow green blue indigo violet
```

Variable composée, agrégée (compound, aggregate)

Concept

- Les variables composées et agrégées sont des fonctionnalités avancées de ksh.
- Ces types de variables permettent de manipuler des groupes de données comme une seule unité.
- Ils peuvent améliorer l'efficacité du code et sa lisibilité.
- Ksh permet une manipulation plus riche de ces variables que bash.

Variables Composées

- En ksh, une variable composée est une variable qui contient un ensemble d'autres variables,
- Principe similaire aux struct (en C), aux records (en Pascal) ou aux attributs d'un objet dans les langages orientés objet.
- Chaque variable à l'intérieur de la variable composée est appelée un "membre" et peut être de n'importe quel type (scalaire, tableau, une autre variable composée, etc.). Par exemple :

Variables agrégées

- Dans certains contextes, "variable agrégées" désigne une variable composée dont les membres sont des tableaux ou d'autres variables composées, créant ainsi une structure de données arborescente et plus complexe.

Déclaration (ligne par ligne)

```
person.firstname=John
person.initial=Q.
person.lastname=Public
```

Déclaration (d'un coup)

```
person=(firstname=John initial=Q. lastname=Public)
```

Utilisation

```
print $person
(lastname=Public initial=Q. firstname=John)

print $person.firstname  # attention !
John.firstname

print ${person.firstname}  # attention !
John.firstname
```

Références

- https://docstore.mik.ua/orelly/unix3/korn/ch04_03.htm
- https://www.ibm.com/docs/en/aix/7.2?topic=shell-enhanced-korn-ksh93

Variables binaires

Usage

- Les variables binaires dans Ksh sont des variables spéciales qui stockent des données sous forme binaire.
- Ces variables sont extrêmement utiles lors de la manipulation de données bas niveau (disque, réseau, etc.)
- L'option `typeset -b` est utilisée pour déclarer une variable comme binaire en Ksh.

Déclaration

```
typeset -b var
```

Assignation

```
var=$(printf "%b" '\x01\x02\x03')
```

Lecture de valeurs binaires

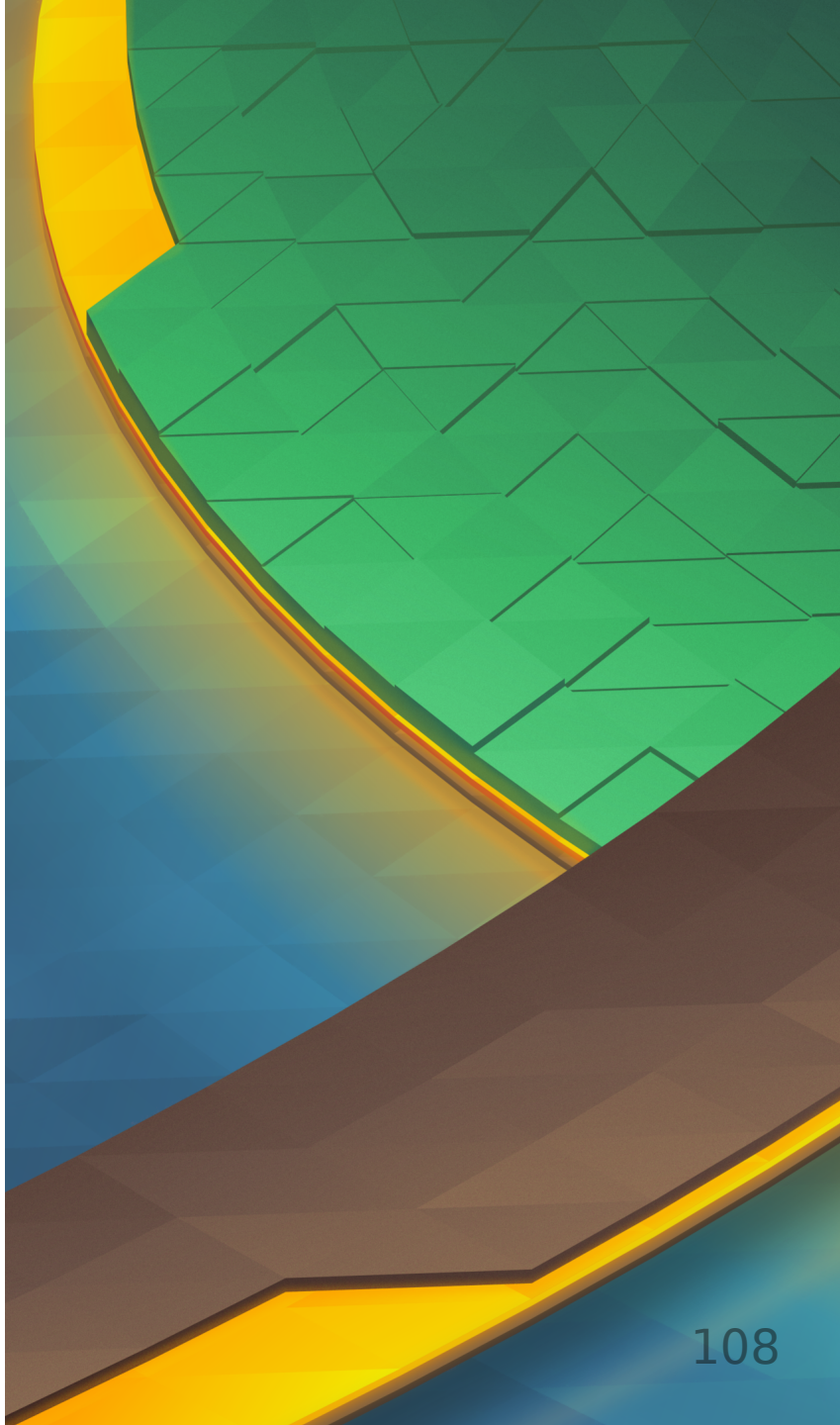
On peut lire la valeur d'une variable binaire en utilisant `$var`. Par exemple, `echo $var` affiche la valeur de var.

⚠️ L'utilisation de variables binaires est une caractéristique unique de Ksh. Bash ne dispose pas de cette fonctionnalité, ce qui peut rendre la manipulation de données binaires plus complexe.

Travaux pratiques

- Utilisation d'un tableau associatif (Key/Value) et des aggregate variables pour la gestion d'une base de données en ksh.

Entrées/sorties étendues



Rappels sur les I/O

Processus séquentiels

```
proc1
proc2
proc3

proc1 ; proc2 ; proc3
```

Redirection des entrées-sorties

```
  <  |  l'entrée standard est lu à partir d'un fichier
-----|---
  >  |  La sortie standard est redirigée dans un fichier (RAZ du fichier)
 >>  |  La sortie standard est redirigée dans un fichier (concaténation du
fichier)
2>   |  les erreurs sont redirigées dans un fichier
2>&1 |  les erreurs sont redirigées dans le même fichier que la sortie standard
```

Here Document

Création de fichier dans un script

```
cmde<<[-] Délimiteur
ligne1
ligne2
ligne3
...
Délimiteur
```

Si `-` est ajouté en préfixe du délimiteur, alors les tabulations de début de ligne sont supprimées du document.

```
cat > fichier <<-MARK
abc
def
MARK
```

Les pipes

```
proc1 **|** proc2

équivaut à :

proc1 > fich
proc2 < fich
```

Les co-processus

Processus en arrière-plan

Lance la commande en tant que processus d'arrière-plan. Le contrôle revient immédiatement au shell.

```
command &
```

La variable spéciale $! contient l'ID du processus du dernier travail en arrière-plan qui a été lancé. Vous pouvez l'enregistrer et examiner le processus plus tard (ps -p $bgpid) ou lui envoyer un signal (kill -HUP $bgpid).

```
bgpid=$!
```

Exemple:

```
proc1 & proc2 & proc3 &
```

Les co-process (spécifiques à ksh)

Les coprocessus permettent de lancer un processus distinct qui s'exécute de manière asynchrone, mais dont les stdin/stdout sont connectés au script parent via des tuyaux.

Démarrer un coprocessus avec un pipe à 2 voies vers celui-ci

```
command |&
```

Lecture du pipe vers le coprocess, au lieu de l'entrée standard

```
read -p var
```

Écriture dans le pipe connecté au coprocesseur, au lieu de la sortie standard

```
print -p args
```

Les co-process (spécifiques à ksh) (suite)

Les coprocessus multiples peuvent être gérés en déplaçant les descripteurs de fichiers spéciaux connectés aux pipes vers l'entrée et la sortie standard, et ou vers des descripteurs de fichiers explicitement spécifiés.

The input from the coprocess is moved to standard input

```
exec <&p
```

The output from the coprocess is moved to standard output

```
exec >&p
```

Note: en bash `command1 |& command2` équivaut à `command1 2>&1 | command2`

Références

- StackExchange: How do you use the command coproc in various shells?
- Bash Hackers Wiki: The coproc keyword
- Dartmouth College: Coprocesses and Background jobs

Gestion des flux (avec exec)

Le principe de exec

- Le builtin `exec` est une commande interne au shell qui modifie l'environnement d'exécution d'un processus.
- `exec` peut remplacer le shell courant par un nouveau programme.
- `exec` peut aussi être utilisé pour manipuler les descripteurs de fichiers du processus courant.

Rediriger les descripteurs de fichiers

- exec permet de rediriger les entrées/sorties de manière permanente pour le reste du script.
- Par exemple, `exec > outfile` redirigera tout le stdout du shell vers outfile.
- Pour effectuer une opération similaire à dup2 (C) avec exec, vous pouvez utiliser la syntaxe `exec 3<&4`
  - Ici le descripteur de fichier 3 est dupliqué à partir du descripteur de fichier 4. Ainsi, quand le processus lit à partir du descripteur de fichier 3, il lira en fait à partir du descripteur de fichier 4.
  - Notez que cela ne ferme pas automatiquement le descripteur de fichier 4.

Ouvrir et fermer les descripteurs de fichiers

- La commande exec peut également ouvrir de nouveaux fichiers (et donc créer de nouveaux descripteurs).
- Cela est particulièrement utile pour gérer plusieurs fichiers ou flux de données simultanément.
- La commande `exec 3< myfile` ouvrira le fichier myfile sur le descripteur de fichier 3, qui peut être lu plus tard dans le script.
- Pour fermer un descripteur de fichier avec exec, vous pouvez utiliser la syntaxe `exec 3<&-`

La substitution de processus

Utiliser exec pour remplacer le processus courant

- La commande exec permet également de lancer un programme définitivement, en remplacement du processus actuel
- Le processus avec le même PID continue, mais le shell n'est plus en exécution.
- Cela est utile pour lancer un programme de manière finale dans un script shell, car il n'y a pas de retour au shell.
- Ex: dans des scripts d'initialisation où le script doit terminer par l'exécution d'un daemon.
- Ex: en utilisant la commande `exec /bin/ls`, le shell courant est remplacé par le programme `/bin/ls`.

Substitution de commande par un flux

La substitution de processus envoie la sortie d'un ou plusieurs processus dans l'entrée standard stdin d'un autre processus.

Syntaxe

```
>(liste_de_commandes)
<(liste_de_commandes)
```

Exemple

```
comm <(ls -l) <(ls -al)
diff <(ls $premier_repertoire) <(ls $deuxieme_repertoire)
cat <(ls -l)  # idem à: ls -l | cat
sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
```

Subtitution de commande par un flux (suite)

La substitution de processus utilise les fichiers `/dev/fd/<n>` pour envoyer le résultat des processus entre parenthèses vers un autre processus.

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/dev/fd/63
```

Lire et écrire du binaire en shell

Mode d'emploi

- typeset -b byte : Crée une variable de type binaire nommée "byte".
- `exec 3<image.jpg` : Ouvre le fichier image.jpg en lecture seule avec le descripteur de fichier 3
- `eof=$(3<#((EOF)))` : Récupère la position de la fin du fichier (EOF) et l'affecte à la variable "eof".
- `3<#((0))` : Se déplace au début du fichier.
- `:> image.cpy` : Crée un nouveau fichier image.cpy, ou vide le fichier s'il existe déjà.
- `while (( $(3<#((CUR))) < $eof ))` lit le fichier byte par byte jusqu'à ce que la fin du fichier soit atteinte.

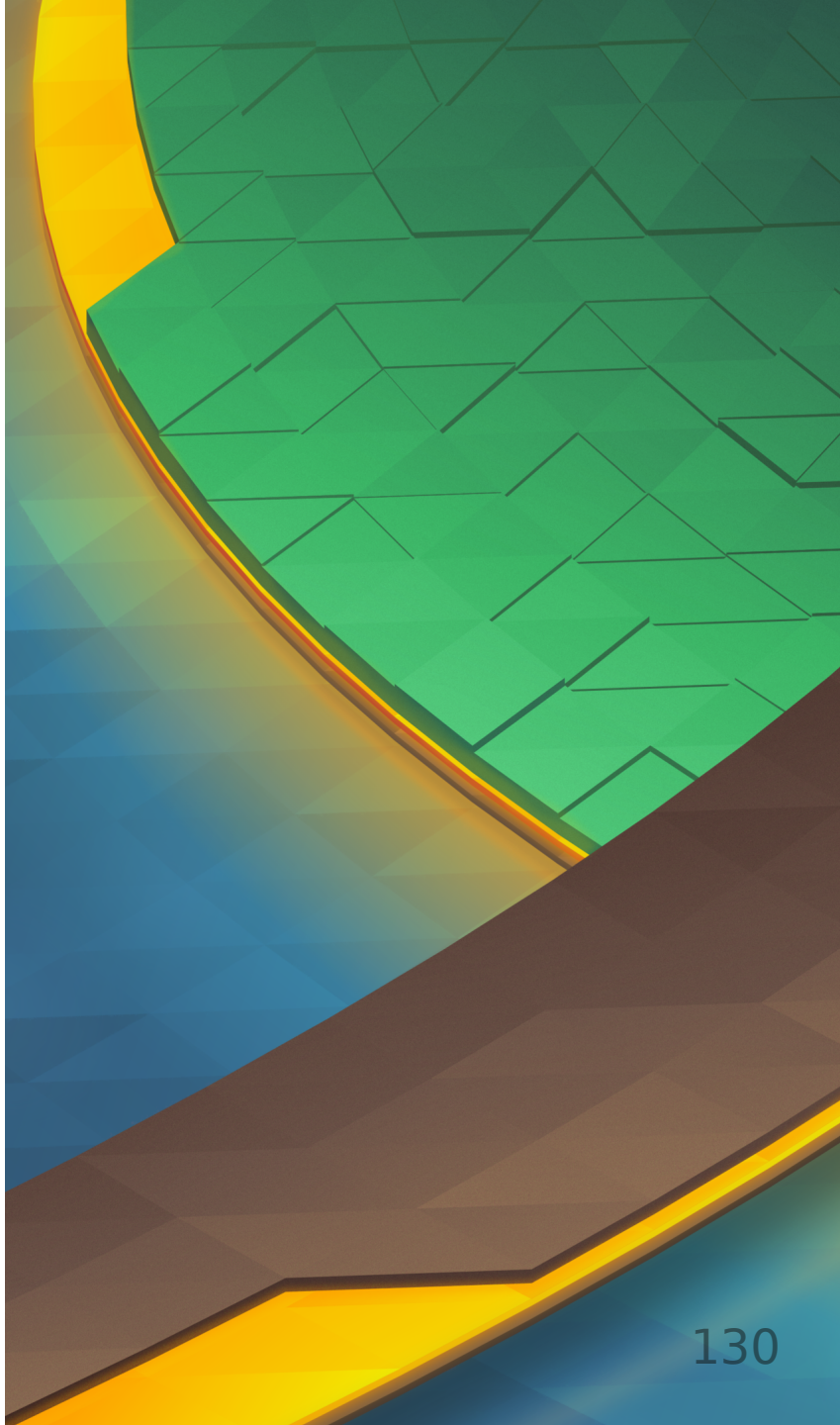
Reférences

- https://blog.fpmurphy.com/2017/08/manipulating-binary-data-using-the-korn-shell.html

Travaux pratiques

- Ecriture d'un daemon HTTP en shell (exec), pour créer un serveur de nombres aléatoires.
  - Avec ou sans coprocess : quelle différence ?
- Ecriture d'un programme d'analyse de binaires, inspiré de https://github.com/wader/fq

Les chaînes de caractères



Substitution et expansion de paramètres

Syntaxe

```
commande1 argument1 $(commande2) ... argumentN
```

Le commande2 est exécutée en priorité par le shell, puis remplacée par son résultat avant exécution de la commande1.

Exemple

```
echo "Vous etes actuellement connecte sur la machine $(uname -n) et vous etes $(logname)"
```

Gestion des paramètres d'une commande

Getopts

Cette commande permet de récuperer facilement les options passées en paramètre du script.

Syntaxe générale

```
getopts Chaîne_options Nom [Argument ...]
```

Nom prend successivement comme valeur celles passées en paramètre du script.
Les seules valeurs valides sont données par chaine_option (voir ci dessous pour plus de détails)

```
while getops vy argument
do
        case $argument in
                v) ...;;
                y) ... ;;
        esac
done
```

Traitement des erreurs

Pour ajouter un traitement d'erreur automatique, il suffit de mettre **:** devant la liste des options valides
Lorsque l'utilisateur utilise une option non valide, **getopts** renvoie **?**

```
while getops :vy argument
do
        case $argument in
                v) ...;;
                y) ... ;;
                \?) ... ; exit ;;
        esac
done
```

Option avec arguments (et gestion des erreurs)

Pour permettre de saisir un paramètre apres une option, il suffit de rajouter **:** après l'option concernée
La variable **OPTARG** contient le paramètre Si l'utilisateur omet le paramètre, **getops** renvoie **:** et **OPTARG** l'option concernée.

```
while getops :y: argument
do
        case $argument in
                y) ... ; arg=$OPTARG;;
                :) echo " l'option -$OPTARG a besoin d'un argument " ; exit ;;
                \?) ... ; exit ;;
        esac
done
```

Extensions ksh (printf, read, select)

Introduction aux extensions ksh

- KSH fournit des extensions pour plusieurs commandes Unix standard comme printf, read et select.
- Ces extensions offrent plus de flexibilité et de fonctionnalités par rapport à leurs équivalents en Bourne Shell ou Bash.
- Les extensions sont conçues pour améliorer l'interactivité des scripts et la manipulation des données.

printf

- printf dans ksh offre une compatibilité avec la commande C éponyme.
- Il permet une mise en forme fine des sorties, avec des spécificateurs de format pour différents types de données.
- Syntaxe : printf format [arguments]
- Exemple : printf "%s\n" $variable permet d'imprimer le contenu de la variable suivi d'une nouvelle ligne.

read

- read en ksh est utilisé pour lire une ligne d'entrée.
- Il a la possibilité de lire dans plusieurs variables à la fois, permettant une manipulation facile des données structurées.
- Syntaxe : read var1 var2 ... varN
- Exemple : read name age lira une ligne d'entrée et assignera les valeurs à $name et $age.

La gestion des menus avec select

La commande **select** écrit sur la sortie d'erreur standard la liste des choix, chacun étant précédé d'un numéro. Si **in liste_choix** n'est pas spécifié, les paramètres positionnels sont utilisés.

```
select Identificateur [in liste_choix]
do
        commande
        ...
done
```

Le contenu de la variable **PS3** s'affiche et l'entrée standard est lu. Si le numéro d'un des mots listés est saisie, le paramètre **Identificateur** prend la valeur de ce mot.

La gestion des menus avec select (suite)

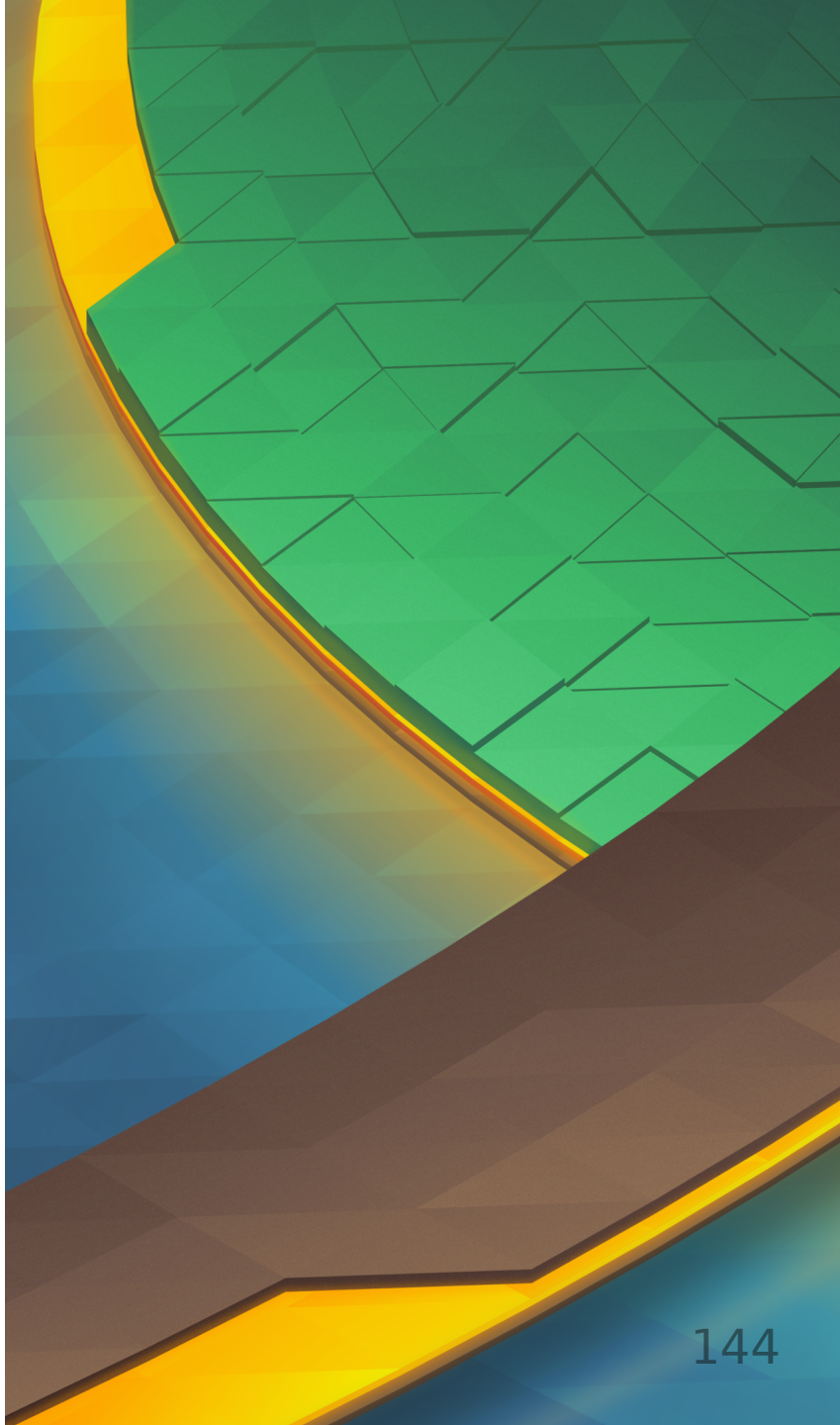
Si la ligne est vide, la liste s'affiche de nouveau. Sinon, la valeur du paramètre Identificateur est "". Le contenu de la ligne lue à partir de l'entrée standard est sauvegardé dans le Paramètre **REPLY**. La liste est exécutée pour chaque sélection jusqu'à un caractère d'interruption ou de fin de fichier.

```
PS3=" votre choix "
select menu_list in "choix 1"  "choix 2"  "fin"
do
  case $menu_list in
    "choix 1").... ;;
    "choix 2")... ;;
    "fin") exit 0 ;;
    "") echo "$REPLY est une option invalide "
  esac
done
```

Travaux pratiques

- Renforcer un shell script à l'aide de getopts, de la substitution de paramètres pour gérer les extensions et les répertoires
- Utiliser select, read et printf dans un shell interactif

Augmenter les capacités du korn shell



Les fonctions

Spécificité des fonctions

- Les fonctions Korn Shell (ksh) sont des ensembles de commandes regroupées sous un nom unique.
- Les fonctions peuvent être utilisées pour automatiser des tâches répétitives, améliorant ainsi l'efficacité et la maintenabilité du code.
- KSH, tout comme Bash, prend en charge deux types de fonctions : définies par l'utilisateur et internes (builtins).

Définition des fonctions (rappel)

- Une fonction est définie avec le mot-clé function, suivi de son nom et d'un bloc de code entre accolades {}.
- Syntaxe: function function_name { command_block }
- Les fonctions sont invoquées en utilisant leur nom, comme une commande.

Fonctions *varname* : Explication et Utilisation

- Les fonctions varname sont spécifiques à KSH et ne se retrouvent pas dans Bash.
- Elles permettent d'intercepter les lectures et les écritures de variables, fonctionnant comme des "getters" et des "setters" en programmation orientée objet.
- Pour déclarer une fonction varname, la syntaxe est : `function .sh.name.variable { case $1 in -*) … esac; }`.
- Pour une variable VAR, les fonctions de lecture `.sh.name.VAR.get` et d'écriture `.sh.name.VAR.set` peuvent être définies.
- Les fonctions varname sont particulièrement utiles pour la validation des entrées et la transformation des données.

Exemples pratiques et cas d'utilisation

- Exemple d'une fonction simple : `function hello { echo "Hello, World!"; }`
- Exemple d'une fonction varname :

```
function .sh.name.VAR.get {
    .sh.value=${VAR:-default_value}
}
function .sh.name.VAR.set {
    VAR=$1
}
```

Les bibliothèques de fonctions

Les bibliothèques

On créé un répertoire dans lequel on stocke les fichiers contenant les fonctions (le nom du fichier doit être le même que le nom de la fonction)

Pour utiliser cette bibliothèque, positionner la variable **FPATH** , puis importer les fonctions (par **autoload** ou **typeset -fu**)

```
FPATH=$HOME/lib/rep1:$HOME/lib/rep2
tyteset -fu ma_fonction
autoload ma_fonction
```

On peut définir des fonctions dans le fichiers définie par ENV mais elles ne seront visibles que du shell interactif, pour les rendre toujours visible il faut les exporter par **typeset -fx** fonc

📘 display or modify KornShell functions

Les builtins

Introduction aux builtins et leur rôle

- Les builtins sont des commandes intégrées directement dans le shell Korn (KSH).
- Elles ont généralement une exécution plus rapide que les commandes externes car elles ne nécessitent pas de `fork()` ou `exec()`.
- Les builtins permettent de manipuler directement les structures internes du shell, comme les variables d'environnement ou les paramètres de shell.

> ℹ️ Bash ne supporte pas l'ajout de builtins personnalisés comme KSH. Il est possible de créer des fonctions en Bash qui agissent de manière similaire aux builtins, mais elles n'auront pas le même niveau de performance ou d'accès aux structures internes du shell que les builtins en KSH.

Écrire un builtin pour Ksh

- Choisir un nom pour le builtin et définir sa fonctionnalité.
- Créer une fonction C qui implémente la logique du builtin.
- Utiliser la macro DEF_BUILTIN pour déclarer le builtin.
- Ajouter la déclaration du builtin à l'array builtins.
- Compiler et tester le shell avec le nouveau builtin.

Intégrer un builtin

- Intégrer un builtin dans KSH nécessite de recompiler le shell.
- Une fois compilé, le shell inclut le nouveau builtin et peut être utilisé comme n'importe quelle autre commande.
- Pour utiliser le builtin dans un script, il suffit d'appeler son nom comme n'importe quelle autre commande.

Les bibliothèques de builtin

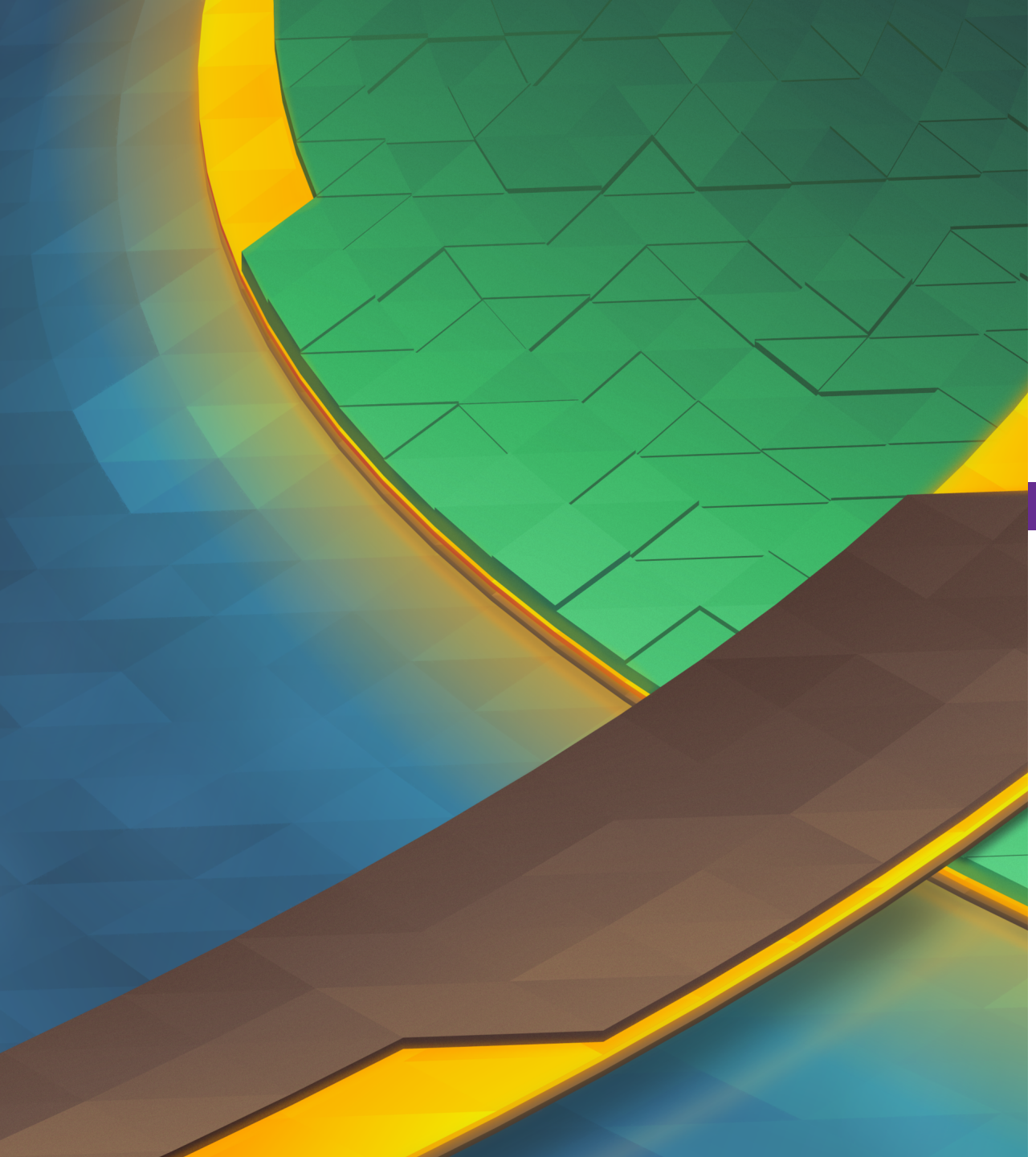
- Les bibliothèques de builtins permettent d'ajouter des collections de builtins au shell - sans recompiler celui-ci.
- Pour créer une bibliothèque de builtins, il faut regrouper plusieurs déclarations DEF_BUILTIN dans un seul fichier source.
- Ensuite, compilez ce fichier pour obtenir une bibliothèque partagée (.so) qui peut être chargée dans le shell avec la commande builtin -f.

Exemples pratiques et cas d'utilisation

- Un exemple de builtin couramment utilisé dans KSH est cd, qui change le répertoire de travail du shell.
- Une utilisation courante des builtins personnalisés pourrait être la création d'une commande pour manipuler une base de données spécifique à une application.
- Par exemple, un builtin dbinsert pourrait insérer une ligne dans la base de données, tandis qu'un builtin dbquery pourrait récupérer des données.

Travaux pratiques

- Créer une bibliothèque de fonctions simples, basename, dirname...
- Ajouter à ksh un builtin permettant de lire une estampille timer précise à la nanoseconde



Merci pour votre attention !