Introduction aux systèmes informatiques

Codage et Systèmes d'exploitation

Pôle ASR – Module M1101 – Semestre 1

Bruno BEAUFILS

⟨bruno.beaufils@univ-lille.fr⟩

https://beaufils.u-lille.fr

Université de Lille, IUT « A », Département informatique

Année 2020/2021

1. Généralités
   Systèmes d'exploitation
   Unix

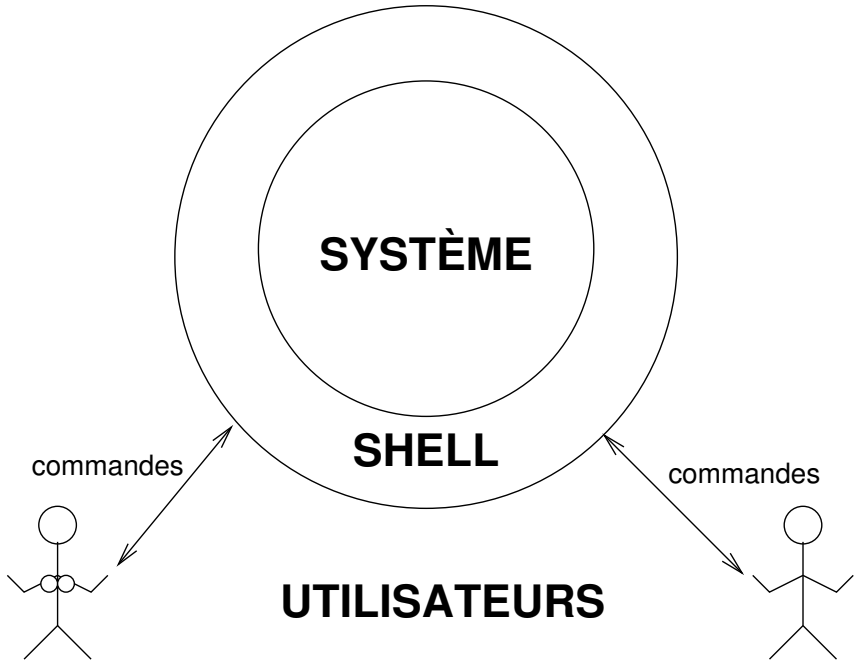
2. Système de fichiers
   Utilisation
   Sous le capot
   Mode d'accès et droits

3. Processus
   Principes
   Entrées/Sorties
   Sous le capot

4. Langages de commandes - Shell
   Scripts
   Interprétation
   Programmation

Langages de commandes

Un langage de commande (*shell*) est un programme capable d'interpréter des commandes qui seront exécutées par le système d'exploitation.



- Interface entre le système d'exploitation et l'utilisateur.
- Permet d'écrire des programmes comportant plusieurs commandes.

Familles

Il existe un grand nombre de shells différents séparés, essentiellement par la syntaxe, en 2 grandes familles :

1. ceux dérivant du **Bourne-shell** (`/bin/sh`)
   Historiquement le premier shell (écrit par Steve Bourne).
   Plutôt orienté programmation qu'interaction.
   - **Bourne Again SHell** (`/bin/bash`)
     une implémentation du Bourne shell faite par le projet GNU.
   - **Korn SHell** (`/bin/ksh`)
     écrit par David Korn.
   - **Z Shell** (`/bin/zsh`)
     une implémentation très modulaire et orientée interaction
   - **(Debian) Almquist shell** (`/bin/ash` ou `/bin/dash`)
     une version respectueuse de la norme POSIX
2. ceux dérivant du **C-shell** (`/bin/csh`)
   Syntaxe très proche de celle du langage C.
   Plutôt orienté interaction que programmation.
   - le **Tenex C-SHell** (`/bin/tcsh`) implémentation libre du C-shell (beaucoup de fonctionnalités dédiées interaction).

➠ nous allons étudier le **Bourne Shell** via `bash`

Modes de fonctionnement

Tous les shells ont 3 modes de fonctionnement :

1. login interactif ..... (connexion à la machine)
2. shell interactif ..... (appel de `bash`)
3. shell non-interactif ..... (script)

Initialisation

Certaines commandes sont exécutées au démarrage de chacun des modes :

1. dans le processus d'un login interactif
   1. les commandes du fichier `/etc/profile` sont lues et exécutées
   2. les commandes du fichier `${HOME}/.bash_profile` sont lues et exécutées
   3. les commandes du fichier `${HOME}/.profile` sont lues et exécutées
2. dans le processus d'un shell interactif les commandes du fichier `${HOME}/.bashrc` sont lues et exécutées
3. dans le processus d'un shell non interactif si la variable `BASH_ENV` a une valeur le shell considère que son contenu (après transformation) est le nom d'un fichier dont les commandes doivent être lues et exécutées

Fichier de commandes

- **ensemble de commandes**
  `source` ⟨*chemin*⟩
  `.` ⟨*chemin*⟩
  L'ensemble des commandes de ⟨*chemin*⟩ sont lues et exécutées comme si elles provenaient du clavier
- **script** (programme shell)
  - est une commande constituée d'appel à d'autres commandes shells
  - est écrit dans un simple fichier texte :
    1. la **première ligne** du fichier définie le *langage* à utiliser :
       #! ⟨*emplacement_du_shell*⟩
    2. le fichier doit être exécutable.
    3. le shell doit pouvoir trouver le fichier.
    4. le caractère « **#** » permet d'insérer des commentaires dans le fichier.

Remarque

Un script est une commande comme une autre sur laquelle on peut faire redirections, tubes, etc.

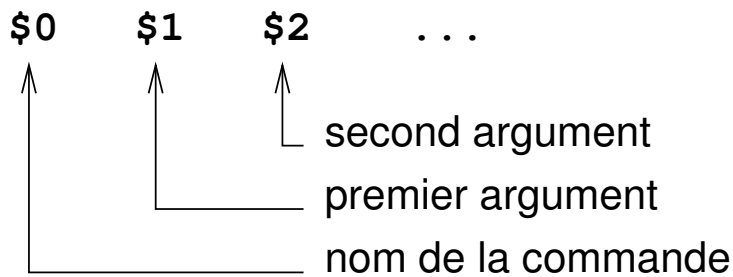
Paramètres de script (1)

Comme toute commande un script peut être appelé avec des paramètres :

- pour modifier son comportement
- pour spécifier les données qu'il doit manipuler

Chacun des paramètres passés sur la ligne de commandes :

- est repéré par sa position *paramètres positionnels*
- est utilisable dans le script via des *variables*



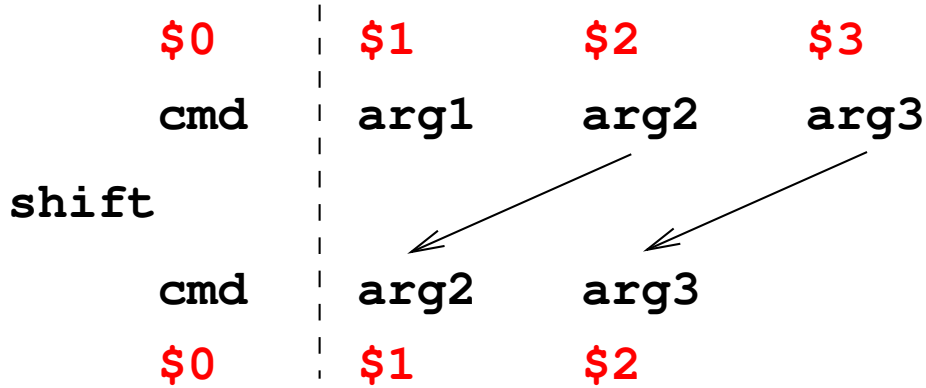
Paramètres de script (2)

Lors de l'exécution le shell remplace **automatiquement** certains *mots* :

`$0`	→	le nom du script tel qu'il a été appelé
`$1`	→	le premier mot apparaissant après le nom du script
`$2`	→	le second mot apparaissant après le nom du script
⋮		
`$n`	→	le n^{e} mot apparaissant après le nom du script
`$#`	→	le nombre de paramètres passés au script
`$*`	→	**une chaîne** contenant tous les paramètres passés au script (à partir de `$1`) séparés les uns des autres par **un** espace
`"$@"`	→	autant de chaîne que de paramètres passés au script

shift

La commande **shift** permet de décaler les paramètres positionnels vers la gauche.



Par défaut **shift** décale un argument à gauche.

Avec un nombre entier en paramètre **shift** décale plusieurs arguments.

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   a. Transformations successives des mots de la ligne :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
   b. Exécution de la ligne transformée :
      - Découpage de la ligne en commandes
      - Préparation des processus (redirections, etc.)
      - Recherche de commande(s)
      - Exécution ou envoi d'un message sur la sortie d'erreur
4. Retour en 1

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   1. Transformations successives des mots de la ligne :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
   2. Exécution de la ligne transformée :
      - Découpage de la ligne en commandes
      - Préparation des processus (redirections, etc.)
      - Recherche de commande(s)
      - Exécution ou envoi d'un message sur la sortie d'erreur
4. Retour en 1

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   1. **Transformations successives des mots de la ligne** :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
   2. **Exécution de la ligne transformée** :
      - Découpage de la ligne en commandes
      - Préparation des processus (redirections, etc.)
      - Recherche de commande(s)
      - Exécution ou envoi d'un message sur la sortie d'erreur
4. Retour en 1

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   1. **Transformations successives des mots de la ligne** :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
   2. **Exécution de la ligne transformée** :
      - Découpage de la ligne en commandes
      - Préparation des processus (redirections, etc.)
      - Recherche de commande(s)
      - Exécution ou envoi d'un message sur la sortie d'erreur
4. Retour en 1

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   1. **Transformations successives des mots de la ligne** :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
   2. **Exécution de la ligne transformée** :
      - Découpage de la ligne en commandes
      - Préparation des processus (redirections, etc.)
      - Recherche de commande(s)
      - Exécution ou envoi d'un message sur la sortie d'erreur
4. Retour en 1

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   1. **Transformations successives des mots de la ligne** :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
   2. **Exécution de la ligne transformée** :
      - Découpage de la ligne en commandes
      - Préparation des processus (redirections, etc.)
      - Recherche de commande(s)
      - Exécution ou envoi d'un message sur la sortie d'erreur
4. Retour en 1

Transformations successives de la ligne

La *ligne* lue subies plusieurs développements (*expansion*) avant exécution :

1. Développement des variables
2. Substitution de commandes
3. Découpage des mots
4. Développement des chemins de fichier

Transformations successives de la ligne

La *ligne* lue subies plusieurs développements (*expansion*) avant exécution :

1. Développement des variables
2. Substitution de commandes
3. Découpage des mots
4. Développement des chemins de fichier

Transformations successives de la ligne

La *ligne* lue subies plusieurs développements (*expansion*) avant exécution :

1. Développement des variables
2. Substitution de commandes
3. Découpage des mots
4. Développement des chemins de fichier

Transformations successives de la ligne

La *ligne* lue subies plusieurs développements (*expansion*) avant exécution :

1. Développement des variables
2. Substitution de commandes
3. Découpage des mots
4. Développement des chemins de fichier

Transformations

Les transformations sont effectuées grâce à

- des **caractères spéciaux** (*meta-characters*)

  espace, tabulation,

  `| & ; ( ), < >`

- des **mots réservés**, notamment :
  - ceux commençant par les caractères : `$ ~ ‘`
  - ceux contenant les caractères : `* ? [ ] ^ -`
  - les mots réduits aux caractères : `{ }`

Tous ces caractères ont donc **un sens particulier** pour le shell

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

- **Protection d'un caractère** : faire précéder le caractère à protéger d'une barre de fraction inversée (*backslash*) \
  ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.

- **Protection complète** : entourer la zone à protéger par des guillemets simples (*quote*) '
  ➠ aucune transformation n'est faite à l'intérieur de la zone protégée.
- **Protection simple** : entourer la zone à protéger par des guillemets doubles (*double-quote*) "
  ➠ hormis \, $ et ‘ aucun caractère spécial n'est plus interprété.

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

- **Protection d'un caractère** : faire précéder le caractère à protéger d'une barre de fraction inversée (*backslash*) \
  ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.
- **Protection complète** : entourer la zone à protéger par des guillemets simples (*quote*) '
  ➠ aucune transformation n'est faite à l'intérieur de la zone protégée.
- **Protection simple** : entourer la zone à protéger par des guillemets doubles (*double-quote*) "
  ➠ hormis \, $ et ‘ aucun caractère spécial n'est plus interprété.

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

- **Protection d'un caractère** : faire précéder le caractère à protéger d'une barre de fraction inversée (*backslash*) \
  ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.

- **Protection complète** : entourer la zone à protéger par des guillemets simples (*quote*) '
  ➠ aucune transformation n'est faite à l'intérieur de la zone protégée.

- **Protection simple** : entourer la zone à protéger par des guillemets doubles (*double-quote*) "
  ➠ hormis \, $ et ‘ aucun caractère spécial n'est plus interprété.

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

- **Protection d'un caractère** : faire précéder le caractère à protéger d'une barre de fraction inversée (*backslash*) \
  ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.

- **Protection complète** : entourer la zone à protéger par des guillemets simples (*quote*) '
  ➠ aucune transformation n'est faite à l'intérieur de la zone protégée.
- **Protection simple** : entourer la zone à protéger par des guillemets doubles (*double-quote*) "
  ➠ hormis \, $ et ‘ aucun caractère spécial n'est plus interprété.

Variables

En shell, comme dans tous langages de programmation il existe une notion de variables avec quelques spécificités :

- Les noms de variables sont des identificateurs ne comprenant que des lettres, des chiffres ou le caractère de soulignement _.

- Il n'existe pas de types de variables
  ➠ toutes les valeurs sont considérées comme des suites de caractères

- Il n'y a pas de réévaluation des variables
  ➠ une variable ne peut être modifiée que par une affectation

Affectation des variables

- **Variables classiques** :
  elles ne sont définies que dans le contexte d'exécution du processus dans lequel elles sont déclarées.

  ⟨*nom*⟩=⟨*valeur*⟩

- **Variables d'environnement** :
  elles sont définies dans le contexte d'exécution du processus dans lequel elles sont déclarées et dans tous les contextes d'exécution des processus que celui-ci peut créer (processus fils)

  ⟨*nom*⟩=⟨*valeur*⟩
  `export` ⟨*nom*⟩

Développement des variables

`${`⟨*nom*⟩`}`

- le shell substitue la variable par la dernière valeur qui lui a été affecté
- les accolades `{` et `}` sont optionnelles, elles sont cependant souvent utilisées pour délimiter le nom de la variable
- si la variable n'existe pas le shell substitue par une chaîne vide

Quelques variables particulières

variables	description
`$HOME`	Le chemin absolu du répertoire principal de l'utilisateur.
`$PATH`	Liste des répertoires dans lesquels le shell recherche les commandes à exécuter. Les répertoires sont séparés par des deux-points `:`.
`$?`	Le code de retour de la dernière commande exécutée.
`$$`	Le PID du processus exécutant le shell en cours.
`$PPID`	Le PID du processus père du processus `$$`.
`$!`	Le PID du dernier processus exécuté en tâche de fond.
`$PWD`	Le répertoire de travail en cours.
`$PS1`	Le message d'invite (*prompt*) principal du shell.
`$PS2`	L'invite secondaire du shell.

Substitution des commandes

Il est possible de remplacer un bout de la ligne de commande par le résultat de l'exécution d'une commande :

1. la zone représentant la commande à exécuter doit
   - être entourée de guillemet inverse (*back-quote*) `‘`
   - ou débuter par `$(` et se terminer par `)`
2. un processus fils (sous-shell) exécutant la commande située dans la zone entourée est créé
3. la sortie standard de ce processus est capturée et remplace la zone sur la ligne de commande.

```
echo 2 + 3 = $(expr 2 + 3)
                ⇓
echo 2 + 3 = 5
```

Résultat de l'exécution : `2 + 3 = 5`

Développement des chemins de fichiers

Un **joker** est un caractère utilisé *à la place* d'un (de plusieurs) autre(s).

Un **modèle** (*glob pattern*) est un mot qui contient un ou plusieurs jokers.

Lorsque le shell trouve un modèle sur la ligne de commande :

1. il cherche la liste des fichiers dont le **chemin correspond** au modèle
2. il remplace le modèle par les **chemins des fichiers** correspondant en les séparant par **un** espace.
3. si aucun fichier ne correspond le modèle est laissé tel quel

Jokers (*Méta-caractères*)

- `*` remplace n'importe quelle **suite de caractères** (y compris vide)
- `?` remplace n'importe quel **caractère**
- `[` ⟨*liste*⟩ `]` remplace n'importe quel caractère de ⟨*liste*⟩
  - on spécifie la liste des caractères que l'on veut représenter
  - `^` placé en début de liste signifie que l'on veut remplacer n'importe quel caractère **non présent** dans la liste
  - `-` utilisé dans la liste définit un intervalle plutôt qu'un ensemble de valeurs

Exemples de modèles de chemins

`f*`	Tous les fichiers dont le nom commence par `f`
`f?`	Tous les fichiers dont le nom fait 2 caractères et commence par `f`
`*.java`	Tous les fichiers dont le nom se termine par `.java`
`tit[oi]`	Les fichiers `titi` et `tito`
`[a-z]*[0-9]`	Tous les fichiers dont le nom commence par une minuscule et se termine par un chiffre
`[^a-z]*`	Tous les fichiers dont le nom **ne commence pas** par une minuscule
`???*`	Tous les fichiers dont le nom est composé d'au moins 3 caractères.
`../*`	Tous les fichiers du répertoire parent.
`/tmp/*.log`	Tous les fichiers se terminant par `.log` du répertoire `/tmp`.
`/tmp/.[^.]*`	*presque* tous les fichiers cachés du répertoire `/tmp`.

Découpage de la ligne en commandes

- Les mots sont séparés par des **blancs** non protégés
  Un blanc est un caractère espace ou une tabulation

- Une commande est un mot quelconque :
  - situé en **première position** de la ligne
  - ou situé **juste après un séparateur** de commandes
  - éventuellement **suivies** par des paramètres (d'autres mots)
  - ne contenant pas le caractère **=**

- Les commandes peuvent être séparées par :
  - des **points-virgules** **;**
    Attendre la fin d'une commande avant de passer à la suivante
  - des **esperluètes** **&**
    Ne pas attendre la fin d'une commande pour passer à la suivante
  - des **tubes** **|**
    Démarrer les commandes en parallèle en les connectant

Opérateurs

Autres séparateurs possibles : opérateurs logiques séquentiels paresseux

- **ET** `cmd1 && cmd2`
  `cmd2` est exécutée si et seulement si `cmd1` a réussi
- **OU** `cmd1 || cmd2`
  `cmd2` est exécutée si et seulement si `cmd1` a échoué
- **Le shell essaie de faire réussir la ligne**
  - évaluation de gauche à droite
  - dès qu'on sait que la séquence ne peut pas réussir (ou qu'elle est déjà réussie) on arrête son évaluation

Définition de réussite (échec) dans le manuel

Recherche et exécution de la commande

1. Si la commande est interne elle est exécutée directement

2. Sinon
   1. Recherche répertoire et fichier
      - si la commande contient au moins un caractère `/`
        extraction du répertoire et du nom de fichier
      - sinon pour tous les répertoires définis dans la variable `$PATH`
        recherche d'un fichier correspondant à la commande
   2. Exécution ou erreur
      - si un fichier a été trouvé **et qu'il est exécutable**
        exécution du code qu'il contient dans un nouveau processus
      - sinon envoi d'un message d'erreur

Commandes internes

Les commandes internes (*builtins commands*) sont traités directement :

- pas de nouveaux processus pour les exécuter
- leur code est intégré au shell
- peuvent modifier le contexte d'exécution du shell courant

commandes	**description**
`cd`	change le répertoire courant
`echo`	envoie ses arguments sur la sortie standard
`pwd`	envoie le nom du répertoire courant sur la sortie standard
`.` ou `source`	lit et exécute les commandes d'un fichier
`exec`	remplace le code par une autre commande
`exit`	termine le processus courant
`read`	affecte une variable en lisant l'entrée standard
`!`	inverse la réussite de la commande suivante

Documentées dans la page du manuel de `bash` et par la commande `help`

Commandes externes

- code stocké dans un **fichier régulier exécutable**
- rangée dans un répertoire de la hiérarchie du système
  la convention est d'utiliser des répertoire nommés `bin`
- recherchée dans une liste de répertoires
  répertoires séparés par des `:` dans la variable `$PATH`
- exécutée dans un nouveau processus par le shell
- elles **ne peuvent pas** modifier le contexte d'exécution du shell

Quelques exemples :

`/bin/ls`, `/bin/cp`, `/bin/mv`, `/bin/mkdir`, `/usr/bin/vi`,
`/usr/local/bin/regarder_ecran`

Structure pour

```
for var in ⟨liste⟩
do
    ⟨cmds⟩
done
```

- ⟨*cmds*⟩ exécutés autant de fois qu'il y a d'élément dans ⟨*liste*⟩
- Pour chaque tour **`$var`** a comme valeur un des éléments de ⟨*liste*⟩.
- Éléments utilisés de la gauche vers la droite de la liste
- ⟨*liste*⟩ est définie après les développements du shell

Code de retour

Sous UNIX toutes les commandes ont un code de retour :

- invisible sur la sortie standard
- visible pour le shell via une variable (`$?`)
- convention :
  - si la commande **réussit** le code de retour **vaut 0**
  - si elle **échoue** le code de retour **est différent de 0**
- le code de retour vu comme le **nombre d'erreurs**

➠ possibilité de faire des actions conditionnées au résultat d'autres actions

Vrai ≡ **réussite** ≡ code de retour = 0
Faux ≡ **échec** ≡ code de retour ≠ 0

Structure si

```
if ⟨cmd-si⟩
then
    ⟨cmds-if⟩
elif ⟨cmd-sinon-si⟩
then
    ⟨cmds-elif⟩
...
else
    ⟨cmds-else⟩
fi
```

1. si ⟨*cmd-si*⟩ réussit alors ⟨*cmds-if*⟩ exécutés
2. sinon
   1. si ⟨*cmd-sinon-si*⟩ réussit alors ⟨*cmds-elif*⟩ exécutés
   2. ...
   3. sinon ⟨*cmds-else*⟩ est exécutés

Structure tant que

while ⟨*cmd-tq*⟩
do
    ⟨*cmds-while*⟩
done

1. ⟨*cmd-tq*⟩ est exécutée
2. si elle a réussit
    1. les ⟨*cmds-while*⟩ sont exécutées
    2. retour en 1

Structure case

```
case ⟨nom⟩ in
    ⟨choix1-1⟩ | ⟨choix1-2⟩ )
        ⟨cmds-choix1-1⟩
        ;;
    ⟨choix2-1⟩ | ⟨choix2-2⟩ )
        ⟨cmds-choix1⟩
        ;;
esac
```

La valeur de la variable ⟨*nom*⟩ est comparée en séquence avec chacun des choix fournit. Si elle correspond à un des choix alors les commandes spécifiées sont exécutées et les choix suivants sont oubliés.

Les choix peuvent faire l'objet d'expansion des noms génériques, de sorte qu'il est souvent fait usage d'un choix placé en dernier correspondant au choix par défaut grâce au caractère « * ».

Regroupement de commandes

Intéressant pour traiter plusieurs commandes comme un tout :

- la sortie standard
- le code de retour

2 regroupements possibles :

- Les commandes sont exécutés dans le shell courant :

```
{ commande1 ; commande2 ; ... ; commandeN ; }
```

- Les commandes sont exécutés dans un sous-shell (nouveau processus) :

```
( commande1 ; commande2 ; ... ; commandeN )
```

En pratique

- Les parenthèses sont des caractères spéciaux
- Les accolades sont des mots réservés

Fonctions : un regroupement nommé.

```
⟨nom⟩ ()
{
  ⟨commandes⟩
}
```

- s'appelle exactement comme une commande
- exécutée comme une commande **mais** dans l'environnement courant
- les arguments positionnels sont fixés à l'intérieur de la fonction avec les arguments d'appels de la fonction
  - uniquement durant l'exécution de la fonction
  - à l'exception de `$0` qui demeure inchangé.
- les fonctions doivent être définies avant d'être utilisées
- statut de retour de la fonction est :
  - le retour de la dernière commande exécutée dans la fonction
  - ou l'argument de la commande **`return`** s'il est présent.