Les fonctions

1. Définition d'une fonction

Le shell bash propose plusieurs syntaxes pour définir une fonction. Nous utiliserons celle-ci :

function *nom_fct*

{

suite_de_commandes

}

nom_fct spécifie le nom de la fonction. Le corps de celle-ci est *suite_de_commandes*.

Pour appeler une fonction, il suffit de mentionner son nom.

Comme pour les autres commandes composées de bash, une fonction peut être définie directement à partir d'un shell interactif.

```
$ function f0
> {
> echo Bonjour tout le monde !
> }
$
$ f0    => appel de la fonction f0
Bonjour tout le monde !
$
```

Les mots réservés function et } doivent être les premiers mots d'une commande pour qu'ils soient reconnus. Sinon, il suffit de placer un caractère point-virgule avant le mot-clé :

function *nom_fct*

{ *suite_de_commandes* ;}

La définition d'une fonction « à la C » est également possible :

function *nom_fct* {

suite_de_commandes

}

L'exécution d'une fonction s'effectue dans l'environnement courant, autorisant ainsi le partage de variables.

```
$ c=Coucou
$
$ function f1
> {
> echo $c    => utilisation dans la fonction d'une variable externe c
> }
$
$ f1
Coucou
$
```

Les noms de toutes les fonctions définies peuvent être listés à l'aide de la commande : declare -F

```
$ declare -F
declare -f f0
declare -f f1
$
```

Les noms et corps de toutes les fonctions définies sont affichés à l'aide de la commande : declare -f

```
$ declare -f
f0 ()
{
  echo Bonjour tout le monde !
}
f1 ()
{
  echo $c
}
$
```

Pour afficher le nom et corps d'une ou plusieurs fonctions : declare -f *nomfct …*

```
$ declare -f f0
f0 ()
{
  echo Bonjour tout le monde !
}
$
```

Une définition de fonction peut se trouver en tout point d'un programme shell ; il n'est pas obligatoire de définir toutes les fonctions en début de programme. Il est uniquement nécessaire que la définition d'une fonction soit faite avant son appel effectif, c'est-à-dire avant son exécution :

function *f1*

{ … ;}

suite_commandes1

function *f2*

{ … ;}

suite_commandes2

Dans le code ci-dessus, *suite_commandes1* ne peut exécuter la fonction *f2* (contrairement à *suite_commandes2*). Cela est illustré par le programme shell *appelAvantDef* :

```
appelAvantDef
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      # !/bin/bash

      echo Appel Avant definition de fct
      fct        # fct non definie

      function fct
      {
      echo Execution de : fct
      sleep 2
      echo Fin Execution de : fct
      }
      echo Appel Apres definition de fct
      fct        # fct definie
- - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Son exécution se déroule de la manière suivante :

```
$ appelAvantDef
Appel Avant definition de fct
./appelAvantDef: line 4: fct: command not found
Appel Apres definition de fct
Execution de : fct
Fin Execution de : fct
$
```

Lors du premier appel à la fonction *fct*, celle-ci n'est pas définie : une erreur d'exécution se produit. Puis, le shell lit la définition de la fonction *fct* : le deuxième appel s'effectue correctement.

Contrairement au programme précédent, dans le programme shell *pingpong*, les deux fonctions *ping* et *pong* sont définies avant leur appel effectif :

```
pingpong
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function ping
      {
      echo ping
      if (( i > 0 ))
        then
          ((i--))
         pong
      fi
      }
      function pong
      {
      echo pong
      if (( i > 0 ))
        then
          ((i--))
         ping
      fi
      }
      declare -i i=4

      ping    # (1) ping et pong sont definies
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Au point (1), les corps des fonctions *ping* et *pong* ont été lus par l'interpréteur de commandes bash : *ping* et *pong* sont définies.

```
$ pingpong
ping
pong
ping
pong
ping
$
```

2. Suppression d'une fonction

Une fonction est rendue indéfinie par la commande interne : unset -f *nomfct …*

```
$ declare -F
declare -f f0
declare -f f1
$
$ unset -f f1
$
$ declare -F
declare -f f0    => la fonction f1 n'existe plus !
$
```

3. Trace des appels aux fonctions

Le tableau prédéfini FUNCNAME contient le nom des fonctions en cours d'exécution, matérialisant la pile des appels.

Le programme shell *traceAppels* affiche le contenu de ce tableau au début de son exécution, c'est-à-dire hors de toute fonction : le contenu du tableau FUNCNAME est vide (a). Puis la fonction *f1* est appelée, FUNCNAME contient les noms *f1* et *main* (b). La fonction *f2* est appelée par *f1* : les valeurs du tableau sont *f2*, *f1* et *main* (c).

Lorsque l'on est à l'intérieur d'une fonction, la syntaxe $FUNCNAME (ou ${FUNCNAME[0]}) renvoie le nom de la fonction courante.

```
      #!/bin/bash

      function f2
      {
      echo " ------ Dans f2 :"
      echo " ------ FUNCNAME : $FUNCNAME"
      echo " ------ tableau FUNCNAME[] : ${FUNCNAME[*]}"
      }

      function f1
      {
      echo " --- Dans f1 :"
      echo " --- FUNCNAME : $FUNCNAME"
      echo " --- tableau FUNCNAME[] : ${FUNCNAME[*]}"
      echo " --- - Appel a f2 "
      echo

      f2
      }

      echo "Debut :"
      echo "FUNCNAME : $FUNCNAME"
      echo "tableau FUNCNAME[] : ${FUNCNAME[*]}"
      echo

      f1

$ traceAppels
Debut :
FUNCNAME :
tableau FUNCNAME[] :                        (a)

 --- Dans f1 :
 --- FUNCNAME : f1
 --- tableau FUNCNAME[] : f1 main           (b)
 --- - Appel a f2

------ Dans f2 :
------ FUNCNAME : f2
------ tableau FUNCNAME[] : f2 f1 main      (c)
$
```

4. Arguments d'une fonction

Les arguments d'une fonction sont référencés dans son corps de la même manière que les arguments d'un programme shell le sont : `$1` référence le premier argument, `$2` le deuxième, etc., `$#` le nombre d'arguments passés lors de l'appel de la fonction.

Le paramètre spécial `$0` n'est pas modifié : il contient le nom du programme shell.

Pour éviter toute confusion avec les paramètres de position qui seraient éventuellement initialisés dans le code appelant la fonction, la valeur de ces derniers est sauvegardée avant l'appel à la fonction puis restituée après exécution de la fonction.

Le programme shell *args* illustre ce mécanisme de sauvegarde/restitution :

```
args
- - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function f
      {
      echo " --- Dans f : \$0 : $0"
      echo " --- Dans f : \$# : $#"
      echo " --- Dans f : \$1 : $1"   => affichage du 1er argument de la
                                          fonction f
      }

      echo "Avant f : \$0 : $0"
      echo "Avant f : \$# : $#"
      echo "Avant f : \$1 : $1"    => affichage du 1er argument du
                                       programme args
      f pierre paul jacques
      echo "Apres f : \$1 : $1"    => affichage du 1er argument du
                                       programme args
- - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ args un deux trois quatre
Avant f : $0 : ./args
Avant f : $# : 4
Avant f : $1 : un
--- Dans f : $0 : ./args
--- Dans f : $# : 3
--- Dans f : $1 : pierre
Apres f : $1 : un
$
```

Utilisée dans le corps d'une fonction, la commande interne `shift` décale la numérotation des paramètres de position internes à la fonction.

```
argsShift
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function f
      {
      echo " --- Dans f : Avant 'shift 2' : \$* : $*"
      shift 2
      echo " --- Dans f : Apres 'shift 2' : \$* : $*"
      }

      echo Appel : f un deux trois quatre
      f un deux trois quatre
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ argsShift
Appel : f un deux trois quatre
--- Dans f : Avant 'shift 2' : $* : un deux trois quatre
--- Dans f : Apres 'shift 2' : $* : trois quatre
$
```

Qu'elle soit utilisée dans une fonction ou à l'extérieur de celle-ci, la commande interne set modifie toujours la valeur des paramètres de position.

```
argsSet
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function f
      {
      echo " -- Dans f : Avant execution de set \$(date) : \$* : $*"
      set $(date)
      cho " -- Dans f : Apres execution de set \$(date) : \$* : $*"
      }

      echo Appel : f alpha beta
      f alpha beta
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ argsSet
Appel : f alpha beta
-- Dans f : Avant execution de set $(date) : $* : alpha beta
-- Dans f : Apres execution de set $(date) : $* : mar mar 7 18:41:39 CET
2006
$
```

5. Variables locales à une fonction

Par défaut, une variable définie à l'intérieur d'une fonction est globale ; cela signifie qu'elle est directement modifiable par les autres fonctions du programme shell.

Dans le programme shell *glob*, la fonction *fUn* est appelée en premier. Celle-ci crée et initialise la variable *var*. Puis la fonction *fDeux* est appelée et modifie la variable (globale) *var*. Enfin, cette variable est à nouveau modifiée puis sa valeur est affichée.

```
glob
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function fUn
      {
      var=Un              # creation de la variable var
      }

      function fDeux
      {
      var=${var}Deux      # premiere modification de var
      }

      fUn
      fDeux
      var=${var}Princ     # deuxieme modification de var

      echo $var
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ glob
UnDeuxPrinc     => trace des modifications successives de la variable
globale var
$
```

Pour définir une variable locale à une fonction, on utilise la commande interne `local`. Sa syntaxe est :

local [*option*] [*nom*[=*valeur*] ...]

Les options utilisables avec `local` sont celles de la commande interne `declare`. Par conséquent, on définira une ou plusieurs variables de type entier avec la syntaxe `local -i` (`local -a` pour un tableau local).

Le programme shell *loc* définit une variable entière *a* locale à la fonction *f1*. Cette variable n'est pas accessible à l'extérieur de cette fonction.

```
loc
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function f1
      {
      local -i a=12               => a est une variable locale à f1
      (( a++ ))
      echo "-- Dans f1 : a  => $a"
      }

      f1
      echo "Dans main : a  => $a" => tentative d'accès à la valeur de a
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ loc
-- Dans f1 : a  => 13
```

```
Dans main : a  =>      => a n'est pas visible dans le corps du programme
$
```

La portée d'une variable locale inclut la fonction qui l'a définie ainsi que les fonctions qu'elle appelle (directement ou indirectement).

Dans le programme shell *appelsCascade*, la variable locale *x* est vue :

- dans la fonction f1 qui définit cette variable
- dans la fonction f2 qui est appelée par la fonction f1
- dans la fonction f3 qui est appelée par la fonction f2.

```
appelsCascade
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function f3
      {
      (( x = -x ))            => modification de x définie dans la fonction
f1
      echo "f3 : x=$x"
      }

      function f2
      {
      echo "f2 : $((x+10))"   => utilisation de x définie dans la fonction
f1
      f3 => appel de f3
      }

      function f1
      {
      local -i x
      x=2                      => initialisation de x
      f2                       => appel de f2
      echo "f1 : x=$x"
      }

      f1                       => appel de f1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ appelsCascade
2 : 12
f3 : x=-2
f1 : x=-2
$
```

6. Exporter une fonction

Pour qu'une fonction puisse être exécutée par un programme shell différent de celui où elle a été définie, il est nécessaire d'exporter cette fonction. On utilise la commande interne export.

Syntaxe : export -f *nomfct ...*

Pour que l'export fonctionne, le sous-shell qui exécute la fonction doit avoir une relation de descendance avec le programme shell qui exporte la fonction.

Le programme shell *progBonj* définit et utilise une fonction *bonj*. Ce script lance l'exécution d'un programme shell *autreProgShell* qui utilise également la fonction *bonj* (mais qui ne la définit pas) ; *autreProgShell* étant exécuté dans un environnement différent de *progBonj*, il ne pourra trouver la définition de la fonction *bonj* : une erreur d'exécution se produit.

```
progBonj
- - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function bonj
      {
      echo bonj : Bonjour $1
      }

      bonj Madame

      autreProgShell
- - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -
autreProgShell
- - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -
#!/bin/bash

echo appel a la fonction externe : bonj
bonj Monsieur
- - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -
$ progBonj
bonj : Bonjour Madame
appel a la fonction externe : bonj
./autreProgShell: line 4: bonj: command not found
$
```

Pour que la fonction *bonj* puisse être exécutée par *autreProgShell*, il suffit que le programme shell qui la contient exporte sa définition.

```
progBonjExport
- - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function bonj
      {
      echo bonj : Bonjour $1
      }

      export -f bonj => la fonction bonj est exportée

      bonj Madame

      autreProgShell
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Après son export, la fonction *bonj* sera connue dans les sous-shells créés lors de l'exécution de *progBonjExport*.

```
$ progBonjExport
bonj : Bonjour Madame
appel a la fonction externe : bonj
bonj : Bonjour Monsieur    => affiché lors de l'exécution de autreProgShell
$
```

La visibilité d'une fonction exportée est similaire à celle d'une variable locale, c'est-à-dire une visibilité *arborescente* dont la racine est le point d'export de la fonction.

Le programme shell *progBonjExport2Niv* définit et exporte la fonction *bonj*. Celle-ci est utilisée dans le programme shell *autreProg1* exécuté par *progBonjExport2Niv* et est utilisée par le programme shell *autreProg2* exécuté par*autreProg1*.

```
progBonjExport2Niv
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function bonj
      {
      echo bonj : Bonjour $1
      }

      export -f bonj

      bonj Madame

      autreProg1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
autreProg1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      echo "$0 : appel a la fonction externe : bonj"
      bonj Monsieur

      autreProg2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
autreProg2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#!/bin/bash

echo "$0 : appel a la fonction externe : bonj"
bonj Mademoiselle
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

$ progBonjExport2Niv
bonj : Bonjour Madame
./autreProg1 : appel a la fonction externe : bonj
bonj : Bonjour Monsieur
./autreProg2 : appel a la fonction externe : bonj
bonj : Bonjour Mademoiselle
$
```

7. Commande interne return

Syntaxe : return [*n*]

La commande interne return permet de sortir d'une fonction avec comme code de retour la valeur *n* (0 à 255). Celle-ci est mémorisée dans le paramètre spécial ?.

Si *n* n'est pas précisé, le code de retour fourni est celui de la dernière commande exécutée.

Dans le programme shell *return0*, la fonction *f* retourne le code de retour 1 au corps du programme shell.

```
retour0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function f
      {
      echo coucou
      return 1
      echo a demain # jamais execute
      }

      f
      echo code de retour de f : $?
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ return0
coucou
code de retour de f : 1
$
```

Remarque :il ne faut confondre return et exit. Cette dernière arrête l'exécution du programme shell qui la contient.

8. Substitution de fonction

La commande interne return ne peut retourner qu'un code de retour. Pour récupérer la valeur modifiée par une fonction, on peut :

- enregistrer la nouvelle valeur dans une variable globale, ou
- faire écrire la valeur modifiée sur la sortie standard, ce qui permet à la fonction ou programme appelant de capter cette valeur grâce à une substitution de fonction : $(fct [arg …]).

```
recupres
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function ajouteCoucou
      {
      echo $1 coucou
      }

      echo la chaine est : $( ajouteCoucou bonjour )
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

La fonction *ajouteCoucou* prend un argument et lui concatène la chaîne *coucou*. La chaîne résultante est écrite sur la sortie standard afin d'être récupérée par l'appelant.

```
$ recupres
La chaine est: bonjour coucou
$
```

9. Fonctions récursives

Comme pour les programmes shell, bash permet l'écriture de fonctions récursives.

Le programme *fctfactr* implante le calcul d'une factorielle sous la forme d'une fonction shell *f* récursive. Outre la récursivité, ce programme illustre

- la définition d'une fonction « au milieu » d'un programme shell
- la substitution de fonction.

```
fctfactr
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      shopt -s extglob

      if (( $# != 1 )) || [[ $1 != +([0-9]) ]]
        then
          echo "syntaxe : fctfactr n" >&2
          exit 1
      fi

      function f
      {
      declare -i n
      if (( $1 == 0 ))
        then echo 1
        else
          (( n=$1-1 ))
          n=$( f $n ) => appel récursif
          echo $(( $1 * $n ))
      fi
      }

      f $1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ fctfactr
syntaxe : fctfactr n
$ fctfactr euzte12uz
syntaxe : fctfactr n
$ fctfactr 1
1
$ fctfactr 4
24
$
```

10. Appels de fonctions dispersées dans plusieurs fichiers

Lorsque les fonctions d'un programme shell sont placées dans différents fichiers, on exécute ces derniers dans l'environnement du fichier shell « principal ». Cela revient à exécuter plusieurs fichiers shell dans un même environnement.

Dans l'exemple ci-dessous, pour que la fonction *f* définie dans le fichier *def_f* puisse être accessible depuis le fichier shell *appel*, on exécute *def_f* dans l'environnement de *appel* (en utilisant la commande interne source ou .). Seule la permission lecture est nécessaire pour *def_f*.

```
appel
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      source def_f    # ou plus court : . def_f
                      # Permissions de def_f : r--r--r—

      x=2
      f               # appel de la fonction f contenue dans def_f
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def_f
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      #!/bin/bash

      function f()
      {
      echo $((x+2))
      }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ appel
4
$
```