Le shell Bash et plus encore

Gazette Linux n°109 — Décembre 2004

William Park

Article paru dans le n°109 de la Gazette Linux de décembre 2004.
Traduction française par Antonin Mellier <antonin POINT mellier CHEZ laposte POINT net>.
Relecture de la traduction française par Joëlle Cornavin <jcornavi CHEZ club TIRET internet POINT fr>.

Table des matières

1. Création d'un correctif et compilation d'un shell Bash 2
- 1.1. Compilation 3
- 1.2. Création du correctif 3
- 1.3. *Builtins* dynamiquement chargeables 5
- 1.4. Tout-en-un 5

2. `strcat`, `strcpy`, `strlen` et `strcmp` 5

3. ASCII et <ctype.h> 6

4. E/S formatées 7
- 4.1. sscanf 7
- 4.2. Lecture et affichage de lignes DOS 8
- 4.3. Émulation simple de Awk 8
- 4.4. Indentation dans **<<** `here-document` 8

5. Générateurs de séquences `{a..b}` et `{a--b}` 8
- 5.1. Séquence d'entiers `{a..b}` 9
- 5.2. Séquences de caractères `{a--b}` 9

6. Filtrage de contenu `${var|...}` 9

7. Instruction case 10
- 7.1. Le motif (*pattern*) **regex** 10
- 7.2. Continuation 11
- 7.3. Condition de sortie 11

8. Les boucles for/while/until 11
8.1. Boucles **for** à variables multiples 12
8.2. Condition de sortie 12
9. Exception et bloc try 12
10. Récapitulatif 13

1. Création d'un correctif et compilation d'un shell Bash

Dans un précédent article (http://linuxgazette.net/108/park.html), j'ai présenté des fonctions shell qui émulent les fonctions C `strcat(3)`, `strcpy(3)`, `strlen(3)` et `strcmp(3)`. Comme la tâche principale du shell est d'analyser du texte, une solution shell pure était possible pour les opérations sur les chaînes de caractères trouvées dans `<string.h>`. Néanmoins, c'est rare. Il n'est pas toujours possible d'émuler du C en shell, en particulier pour accéder à des bibliothèques système bas niveau et à des applications tierces. Même si c'était possible, vous réinventeriez la roue en ignorant le travail qui a été effectué pour les bibliothèques C. De plus, les scripts shell sont, sans exception, des ordres de grandeur plus lents. Cependant, le shell a l'avantage de permettre un développement et une maintenance aisés, parce qu'il est plus facile à écrire et à lire.

Ce dont on a besoin, ensuite, est la capacité d'écrire un script d'invocation (*shell wrapper*) avec des liaisons vers des routines C. Un mécanisme shell qui permet d'écrire une extension C est appelé *builtin*, par exemple **read**, **echo** et **printf**, etc. Quand certaines fonctionnalités requièrent des changements dans la façon dont le shell interprète une expression, alors des modifications doivent être apportées dans le code d'analyse syntaxique (*parsing*) du shell. Quand vous avez besoin de vitesse, une extension C est ce qu'il y a de mieux.

Mon correctif (*patch*) pour le shell Bash 3.0 est disponible aux adresses suivantes :

- http://freshmeat.net/projects/bashdiff/ : présentation
- http://home.eol.ca/~parkw/index.html#bash : documentation

La dernière archive tar (*tarball*) bashdiff-1.11.tar.gz contient deux fichiers diff :

1. bashdiff-core-1.11.diff porte sur les fonctionnalités qui seront compilées statiquement dans le shell. Il en ajoute de nouvelles en modifiant le code d'analyse syntaxique de Bash. Cette opération est entièrement réversible en ce sens qu'aucune signification existante n'est changée. Donc, ce qui fonctionne dans votre ancien shell, fonctionnera aussi dans le nouveau. Par exemple, il ajoute :
   - une nouvelle expansion d'accolades `{a..b}` : génération d'entiers et de lettres, paramètres positionnels et expansion des tableaux
   - une nouvelle expansion de paramètres `${var|...}` : filtrage du contenu, inclusion des listes (comme Python)
   - une nouvelle substitution de commandes `$(=...)` : ancrage de la virgule flottante dans Awk
   - une extension des instructions **case** : expressions régulières, suites, sections **then/else**
   - une extension pour les boucles `while/until` : sections **then/else**, variables multiples pour les boucles **for**

- des blocs de test avec exception d'entier (comme en Python)
- un nouvel opérateur `<<+ here-document` : indentation relative

2. bashdiff-william-1.11.diff porte sur les *builtins* dynamiquement chargeables (*loadables*) qui sont disponibles à part dans votre session shell. Il ajoute de nouvelles commandes, pour faire l'interface avec les bibliothèques système/applications et pour fournir un *wrapper* plus rapide pour les opérations courantes. Par exemple, il ajoute :
   - une extension aux *builtins* `read/echo` : lignes DOS
   - des *wrappers* `sscanf(3)`, `<string.h>` et `<ctype.h>`, la conversion ASCII/chaînes de caractères
   - un nouveau *builtin* `raise` pour les blocs **try**
   - une découpe/un collage de tableaux, un filtre/map/zip/unzip de tableaux (comme en Python)
   - des opérations sur la fonction `regex(3)` : correspondance, découpage, recherche, remplacement, rappel
   - un générateur de modèles HTML (comme PHP, JSP, ASP)
   - une interface pour les base de données GDBM, SQLite, PostgreSQL, et MySQL
   - une interface d'analyseExpat XML
   - opérations piles/files d'attente sur les tableaux et les paramètres positionnels
   - graphique/tracé `x-y`

Toutes les fonctionnalités sont décrites dans les fichiers d'aide internes du shell auquel on peut accéder avec la commande **help**.

1.1. Compilation

Avant de vous lancer dans un shell corrigé, vous devez savoir comment compiler depuis le source, étant donné que le correctif s'effectue par rapport à l'arborescence source. Voici les étapes nécessaires pour télécharger et compiler le shell standard Bash 3.0 :

On a maintenant un bash binaire exécutable qui est comme votre bash actuel, normalement `/bin/bash` . Vous pouvez l'essayer en saisissant :

1.2. Création du correctif

Pour compiler mon shell corrigé, les étapes sont sensiblement les mêmes que précédemment. Téléchargez une archive tar (*tarball*), appliquez mon correctif dans l'arborescence source (à partir des étapes précédentes) et compilez. `bashdiff.tar.gz` pointera toujours sur le dernier correctif, qui est ici `bashdiff-1.10.tar.gz`.

Maintenant, vous avez :

- bash, qui est le shell principal exactement comme avant ; il sera installé dans `/usr/local/bin/bash` et
- william.so, qui est un objet partagé contenant des modules chargeables à la demande (*loadables*) ; il sera installé dans `/usr/local/lib/libwilliam.so` avec un lien symbolique vers `/usr/local/lib/william.so`. Dans la version 1.10, il y a 33 *builtins* chargeables à la demande, à savoir :
  - Lsql
  - Msql
  - Psql
  - gdbm
  - xml
  - array
  - arraymap
  - arrayzip
  - arrayunzip
  - basp
  - match
  - vplot
  - pp_append
  - pp_collapse
  - pp_flip
  - pp_overwrite
  - pp_pop
  - pp_push
  - pp_rotateleft
  - pp_rotateright
  - pp_set
  - pp_sort
  - pp_swap
  - pp_transpose
  - sscanf
  - strcat
  - strcpy
  - strlen

- strcmp
- tonumber
- tostring
- chnumber
- isnumber

1.3. *Builtins* dynamiquement chargeables

Si votre shell comporte **enable** `-[fd]`, alors vous pouvez charger/décharger des commandes *builtin* dynamiquement, d'où le nom. La procédure est simple. Par exemple :

chargera la commande **vplot** depuis la bibliothèque partagée william.so que vous venez de compiler et d'installer. Utilisez **./william.so** si vous ne l'avez pas encore installée. Une fois chargée, vous pouvez l'utiliser exactement comme les *builtins* standard qui sont liées statiquement dans le shell. Donc :

affichera la version longue et courte du fichier d'aide de la commande, alors que :

et

affichera le tracé `x-y` d'une parabole dans votre terminal. Pour la décharger, saisissez :

1.4. Tout-en-un

Les modules chargeables à la demande (*loadables*) sont pratiques si vous voulez juste charger les *builtins* dont vous avez besoin et si vous ne souhaitez ou ne pouvez pas changer votre shell de connexion. De plus, les modules chargeables sont plus faciles à compiler à partir d'une version précédente (incrémentiellement), ce qui est important puisque de nouveaux *builtins* sont ajoutés ou mis à jour plus souvent que le code d'analyse principal du shell.

Néanmoins, vous pouvez être amené à compiler et inclure le tout dans un seul exécutable, comme sous Windows par exemple. Pour compiler un fichier binaire « tout-en-un », vous devez saisir quelques lignes de code supplémentaires. Il faut générer le fichier binaire par défaut du bash, car nous avons besoin de tous les fichiers `.h` et `.o`.

Ici, bash+william est identique au bash, sauf que tous les *builtins* sont liés statiquement dedans. Je recommande le fichier binaire simple bash+william aux débutants, car il évite d'avoir à se rappeler de ce qu'il faut charger et décharger. Tout est là sous la main.

2. `strcat`, `strcpy`, `strlen` et `strcmp`

Dans un article précédent, vous avez vu `strcat(3)`, `strcpy(3)`, `strlen(3)` et `strcmp(3)` en tant que fonctions shell. Dorénavant, une version shell de ces fonctions C est également disponible sous forme de *builtins* :

Vous allez découvrir que l'usage des commandes est le même qu'avec les fonctions shell, sauf pour l'option `-i` de `strcmp` dans la comparaison insensible à la casse, c'est-à-dire :

- strcat var string [a:b]
- strcpy var string [a:b]
- strlen string...
- strcmp [-i] string1 string2 [a:b]

Par exemple :

Si vous avez à la fois une fonction shell et un *builtin* shell qui ont le même nom, alors la fonction shell aura la priorité. Pour découvrir qui est qui, saisissez :

et pour supprimer des fonctions shell, saisissez :

Pour comparer leur vitesse, saisissez :

Vous constaterez que la fonction shell n'est qu'environ 5 fois plus lente environ, ce qui est tout à fait satisfaisant, puisque nous parlons de script shell par rapport au C. Cependant, si vous utilisez des options sur les sous-chaînes :

la différence est de 25 fois.

3. ASCII et <ctype.h>

Bien que les opérations sur les chaînes de caractères soient faciles en shell, il est généralement difficile d'examiner et de manipuler les caractères individueles de la chaîne. De plus, il est très difficile d'afficher la plage entière des caractères (`0-127`) et des caractères étendus sur 8 bits (`128-255`) de la table ASCII, car il faut utiliser les caractères en octal, hexadécimal ou échappés avec une barre oblique inverse s'ils ne sont pas affichables. Mettre en majuscule un mot, par exemple, est extrêmement verbeux dans un shell normal :

ce qui ne fonctionne que dans les paramètres régionaux (*locales*) en anglais, à cause des registres explicites [`a-z`] et [`A-Z`]. En revanche, en C, il suffit d'appeler `isupper(3)`, `islower(3)`, `toupper(3)` et `tolower(3)`, qui fonctionnent avec tous les paramètres régionaux que la bibliothèque C prend en charge.

Ce dont nous avons besoin, ce sont des scripts d'invocation *shell wrappers*) pour les fonctions C standard suivantes : `toupper(3)`, `tolower(3)`, `toascii(3)`, `toctrl()`, `isalnum(3)`, `isalpha(3)`, `isascii(3)`,`isblank(3)`,`iscntrl(3)`, `isdigit(3)`, `isgraph(3)`, `islower(3)`, `isprint(3)`, `ispunct(3)`, `isspace(3)`, `isupper(3)`, `isxdigit(3)` et `isword()`. La plupart d'entre elles étant définies dans `<ctype.h>`, les opérations sur les caractères peuvent être effectuées simplement et efficacement.

- tonumber ***chaine***...
- tostring [-v var] ***nombre***...

  J'ai décidé de suivre la même méthode que la commande **od** et de convertir les chaînes de caractères en suites de nombres ASCII (`0-255`). **tonumber** affiche le nombre ASCII de chaque caractère d'une

chaine, un peu comme **od -A n -t dC**. Ces nombres sont maintenant séparés par des espaces, ce qui facilite les choses pour travailler avec en shell. À l'inverse, **tostring** convertit chaque ***nombre*** en caractère ASCII. Si l'option `-v` est spécifiée, la sortie sera enregistrée dans la variable ***var***. Par exemple :

Une des caractéristiques remarquables de **tostring** est qu'elle peut interpréter l'octet nul (`\0`), si bien que l'on peut écrire des scripts shell pour décrire des données binaires, comme :

- chnumber { toupper | tolower | toascii | toctrl } [***nombre***...]

  Versions shell de `toupper(3)`, `tolower(3)` et autres, incluses dans `<ctype.h>`. Celles-ci lisent les nombres et les affichent après leur conversion en fonction d'options qui ont le même nom que celles des fonctions C correspondantes, par exemple :

- isnumber { alnum | alpha | ascii | blank | cntrl | digit | graph | lower | print | punct | space | upper | xdigit | word } [number...]

  Versions shell de `isupper(3)`, `islower(3)`, et autres incluses dans <ctype.h>. Elles lisent les nombres et renvoient succès (*success*) ou échec (*failure*) selon les options qui ont été choisies. Ces fonctions sont les mêmes que les fonctions C correspondantes avec le préfixe `is` en moins..

  Ainsi, l'exemple précédent de mise en majuscules d'un mot devient :

  ce qui est beaucoup plus rapide et compréhensible.

Maintenant que Bash couvre plutôt bien les fonctions de `<string.h>` et `<ctype.h>`, vous pouvez effectuer des opérations sur des caractères et des chaînes de caractères dans un script shell pratiquement de la même manière que dans du code C. Le texte ainsi que les données binaires sont gérées avec facilité et cohérence. Ces quelques modifications à elles seules représentent déjà une grande amélioration par rapport au shell standard.

4. E/S formatées

4.1. sscanf

Une des premières choses que l'on apprend dans tout langage est la lecture et l'affichage. En C, on utilise les fonctions `printf(3)`, `scanf(3)` et autres définies dans `<stdio.h>`. Pour afficher dans le shell, on fait appel aux *builtins* **echo** et **printf**. Curieusement, pourtant, il manque une version shell de `scanf(3)`. Par exemple, pour analyser les 4 nombres de ***11.22.33.44***, vous pouvez saisir :

Cependant, si votre champ n'est pas aussi bien délimité que ci-dessus, les choses se compliquent.

J'ai ajouté la version shell de la fonction C `sscanf(3)` :

- sscanf input format var1 [... var9]

Du fait que le shell n'a que des données de type chaînes de caractères, il ne prend en charge que des formats de chaînes, c'est-à-dire `%s`, `%c`, `%[...]`, `%[^...]` et jusqu'à 9 variables. Donc, vous pouvez analyser des chaînes formatées tout comme vous le feriez en C, par exemple :

4.2. Lecture et affichage de lignes DOS

De temps en temps, il est nécessaire d'afficher et de lire des lignes DOS qui se terminent par `\r\n` (`CR`/`NL`). Bien que vous puissiez afficher `\r` explicitement, l'insertion automatique de `\r` juste devant `\n` est difficile à réaliser en shell. Pour la lecture, vous devez explicitement supprimer le `\r` de fin.

J'ai créé un correctif des *builtins* standard **echo** et **read** pour qu'ils puissent lire et écrire des lignes DOS :

- **echo** [-....] `-D` [***arg*** ...]
- **read** [-...] `-D` [***nom*** ...]

Par exemple :

4.3. Émulation simple de Awk

Souvent, il est nécessaire d'analyser des lignes et de travailler avec des variables de type Awk comme `NF, NR, $1, $2, ..., $NF`. Cependant, quand vous utilisez Awk, il est difficile de ramener ces variables dans le shell ; vous devez les écrir en syntaxe shell dans un fichier temporaire pour ensuite le reprendre. Cela rend fastidieux de faire des allers-retours entre le shell et Awk.

J'ai créé un correctif du *builtin* standard **read** pour offrir une émulation simple de Awk, en créant les variables `NF` et `NR`, et en affectant les champs à `$1`, `$2`, ..., `$NF`.

- **read** [-...] `-A` [***nom*** ...]

Par exemple :

En outre, comme dans Awk, chaque appel pour lire `-A` incrémentera `NR`.

4.4. Indentation dans `<< here-document`

<< est l'opérateur de redirection d'entrée, où l'entrée standard est tirée du texte même du source du script. **<<** conservera les espaces blancs de début et **<<-** supprimera toutes les tabulations de début. Le problème avec **<<-** est que l'indentation relative est perdue.

J'ai ajouté un nouvel opérateur **<<+** qui conserve l'indentation par tabulation du `here-document` par rapport à la première ligne. Il est accessible directement via le shell (c'est-à-dire `./bash` ou `/usr/local/bin/bash`), parce que le correctif est intégré dans le code d'analyse syntaxique principal. Donc :

affichera

5. Générateurs de séquences `{a..b}` et `{a--b}`

5.1. Séquence d'entiers `{a..b}`

Bash 3..0 (et Zsh) comportent l'expression `{a..b}` qui génère une séquence d'entiers en tant qu'élément de l'expansion d'accolades, mais on ne peut pas utiliser la substitution de variables car l'expression `{a..b}` doit contenir explicitement des entiers.

Mon correctif étend l'expansion d'accolades de façon à inclure la substitution de variables, de paramètres et de tableaux, de même qu'un générateur de séquences de lettres uniques. Par exemple ;:

donneront toutes le même résultat, c'est-à-dire 1 2 ... 10. D'autres détails sont disponibles dans le fichier d'aide :

Une application utile pourrait résider dans le téléchargement d'un groupe d'images sur un site Internet. Il y a tant de sites familiaux sur le *Web* qu'il est difficile d'en recommander un. Quand vous trouverez un site rempli de contenu éducatif, vous pourrez saisir :

pour pouvoir continuer votre étude en privé (en fonction de ce que permet la loi sur les droits d'auteur de votre pays) plus tard quand vous aurez plus de temps.

5.2. Séquences de caractères `{a--b}`

En plus des entiers, vous pouvez aussi générer une séquence de lettres uniques à l'aide de la variation `{a--b}`, où `a` et `b` sont des lettres explicites comme reconnue par la fonction `isletter(3)` dans `<ctype.h>`. Par exemple :

en sautant tout caractère autre qu'une lettre (s'il y en a) entre les extrémités.

6. Filtrage de contenu `${var|...}`

On appelle cela inclusion de listes en Python et dans les langages fonctionnels. Pour l'essentiel, c'est une façon de générer une liste à partir d'une autre liste. Pour chaque élément de la liste, vous pouvez changer le contenu ou choisir de ne pas l'inclure du tout.

- `${var|command}`

  Par défaut, on utilise la sortie de la substitution de commande `command var` dans l'expansion de paramètres au lieu de la chaîne originelle. Si la sortie standard (stdout) est vide, alors elle est supprimée de l'expansion. Ici, `var` peut être quelque chose qui peut apparaître dans d'autres expansions de paramètres, c'est-à-dire `${var:...}`, `${var#...}`, `${var%...}` et `${var/...}`. `command` est tout ce que vous pouvez saisir sur la ligne de commande, c'est-à-dire un alias, une fonction shell, une commande *builtin*, une commande externe et des scripts shell. Donc :

  Ce comportement est similaire à ce qui est disponible dans les langages fonctionnels, sauf qu'il soit implémenté dans le cadre du shell. Malheureusement, la subsitution de commandes ne conserve pas les espaces, parce qu'elle capture stdout.

- ${var|?command}

  Lorsque `command` suit immédiatement `?`, alors la chaîne originelle est incluse dans l'expansion de paramètres uniquement si `command var` renvoie `success` (0). Si tel n'est pas le cas, alors elle est supprimée de l'expansion. Le contenu reste inchangé mais pouvez décider de l'inclure ou non. Par conséquent, `${var|?true}` sera équivalent à `${var}`, puisque `true` renvoie toujours `success` (0). Par exemple :

- ${var|/regex}
- ${var|=glob}
- ${var|~regex}
- ${var|!glob}

  Si l'on veut faire un filtrage spécial, vous pouvez spécifier un motif [*pattern*] `glob(7)` ou `regex(7)` à faire correspondre par rapport à certains élements de la variable : `${var|=glob}` et `${var|/regex}` incluront la chaîne uniquement s'il y a une correspondance ; à l'inverse, `${var|!glob}` et `${var|~regex}` incluront la chaîne uniquement s'il n'y a pas de correspondance. Les exemples ci-dessus peuvent s'écrire sous la forme suivante :

- ${var|:a:b}

  Vous pouvez extraire une plage `[a:b]` de style Python à l'aide de `${var|:a:b}`, qui est identique à la syntaxe shell standard `${var:a:n}`. Si `var` est une chaîne, alors ce sera une sous-chaîne ; si `var` est une liste, alors ce sera une sous-liste. Par exemple :

  affichera, respectivement, les trois premiers caractères ou éléments de la liste, les trois derniers et tous les éléments, sauf le premier et le dernier.

- ${var|*n}

  Si vous devez dupliquer une chaîne ou une liste, alors `${var|*n}` copiera la chaîne ou la liste `n` fois. Par exemple :

7. Instruction case

7.1. Le motif (*pattern*) regex

La syntaxe standard de l'instruction **case** est la suivante :

J'ai étendu la syntaxe à :

de sorte que la liste motif sera interprétée en tant que **regex** si l'expression se termine par une double parenthèse `))`. Pour le reste, le fonctionnement reste le même.. Bien que Bash 3.0 ait `[[ string =~`

`regex ]]`, une instruction **case** est encore une meilleure syntaxe pour deux motifs ou plus, ou si vous devez tester à la fois **glob** et **regex** dans le même contexte.

Alors que **glob** reconnaît la chaîne entière pour renvoyer `success`, **regex** peut reconnaître une sous-chaîne. S'il y a une correspondance, alors la variable tableau `SUBMATCH` contiendra la sous-chaîne correspondante dans `SUBMATCH[0]` et tout groupe entre parenthèses dans le motif **regex** dans `SUBMATCH[1]`, `SUBMATCH[2]`, etc. Par exemple :

aboutira à une correspondance, et

- `SUBMATCH[0]=a<bc123`, l'expression régulière entière `([a-z]+)([0-9]+)`
- `SUBMATCH[1]=abc`, le premier groupe `([a-z]+)`
- `SUBMATCH[2]=123`, le second groupe `([0-9]+)`

7.2. Continuation

En Zsh et en Ksh, vous pouvez continuer avec la liste de commandes suivante si vous utilisez **;&** au lieu de **;;**. Donc :

`command1` sera exécutée si `pattern1` est vérifié. Ensuite, l'exécution continuera avec `command2`, puis la liste de commande suivante, tant qu'elle ne rencontrera pas de double point-virgule. Or, avec le Bash c'est également possible.

De plus, si vous terminez la liste de commande par **;;&**,

commande1 s'exécutera si ***motif***1 correspond. L'exécution continuera ensuite en testant ***motif***2 au lieu de sortir de l'instruction **case**. Par conséquent, il testera toute la liste de motifs de la liste, qu'il y ait ou non une correspondance avérée. Zsh et Ksh n'offrent pas cette fonctionnalité.

7.3. Condition de sortie

Souvent, il est nécessaire de connaître la condition de sortie d'une instruction **case**. Vous pouvez utiliser `*)` comme motif par défaut, mais il n'est pas évident de déterminer s'il y a eu une correspondance puisque vous sortez de l'instruction **case**. Avec mon correctif, vous pouvez ajouter une section `then` et `else` optionnelle à la fin de l'instruction **case** juste après `esac` et de traiter l'instruction **case** comme si c'était une vraie instruction `if` . Voici à quoi ressemblera la nouvelle syntaxe :

où **esac then** et **esac else** ne peuvent pas être séparés par `;` ou par un retour à la ligne. Le **then-COMMANDS** sera exécuté si il y a une correspondance, sinon ce sera **else-COMMANDS** s'il n'y en a pas.

Par exemple :

affichera `no`, alors que

affichera `matched` et `yes`.

8. Les boucles for/while/until

8.1. Boucles for à variables multiples

Dans le shell standard, on ne peut utiliser qu'une seule variable dans une boucle **for**. J'ai ajouté une syntaxe à plusieurs variables, de sorte que

affichera

comme vous l'espérez. Ici, les variables doivent être séparées par des virgules. S'il y a trop peu d'éléments à affecter dans la dernière itération, les variables restantes seront affectées à la chaîne vide (`null`).

8.2. Condition de sortie

Comme dans l'instruction **case**, vous devez souvent savoir si vous êtes sorti de la boucle normalement ou en employant l'instruction **break**. Avec mon correctif, vous pouvez ajouter des sections **then** et **else** optionnelles à la fin des boucles **for**, **while** et **until** juste après **done**. Voici à quoi ressemblera la nouvelle syntaxe :

où **done then** et **done else** ne peuvent pas être séparés par `;` ou par un retour à la ligne. Ici, **then-COMMANDS** sera exécuté si la boucle s'est terminée normalement, sinon **else-COMMANDS** sera exécuté si **break** a été utilisé. Par « normalement », je veux dire que la boucle **for** a épuisé toute la liste d'éléments, que le test sur le **while** a échoué ou que le test **until** a réussi.

Par exemple :

affichera `1` uniquement pour la première itération, puis il sortira de la boucle. Mais :

affichera `1 2 3` et la condition de sortie s'effectuera normalement. Il en sera de même pour les boucles **while** et **until**.

La capacité de tester la condition de sortie améliore la lisibilité des scripts shell, car il n'est plus nécessaire d'utiliser une variable comme drapeau (*flag*). Python possède un mécanisme similaire pour tester la condition de sortie d'une boucle, mais il utilise la valeur de retour du test. On a donc une sortie de la boucle **while** quand le test échoue et Python utilise **else** comme condition normale de sortie, ce qui crée un peu de confusion.

9. Exception et bloc try

En pratique, tous les langages modernes ont la capacité de lever une exception pour sortir d'un code très imbriqué, traiter les erreurs ou faire des sauts multipoint. J'ai ajouté un nouveau bloc **try** au Bash qui saisira les exceptions levées par le nouveau *builtin* **raise**.

- raise [n]
- try

où **done in** ne peut pas être séparé par **;** ou par un retour à la ligne. En outre, la liste de motif, dans l'instruction de type « instruction **case** », doit être explicitement un nombre entier.

Ceci combine les éléments des boucles, le *builtin* de sortie et l'instruction **cas**. À l'intérieur d'un bloc **try**, le *builtin* **raise** peut être utilisé pour lever une exception sur un entier. L'exécution sortie ensuite du bloc **try**, exactement comme pour quitter les boucles **for/until/while**. On peut utiliser une instruction de type **case** pour prendre capturer l'exception. Si l'exception est capturée, alors elle est réinitialisée et l'exécution continue en suivant le bloc **try**. Si l'exception n'est pas capturée, alors l'exécution repart jusqu'à ce qu'elle soit capturée ou jusqu'à ce qu'il n'y ait plus de blocs **try**.

Par exemple :

affichera `a aa` et

affichera `a` et l'exception sera `2`.

10. Récapitulatif

Dans le prochain article, j'aborderai les *builtins* dynamiquement chargeables liés aux tableaux, le découpage des expressions régulières, l'interfaçage avec des bibliothèques externes comme une base de données SQL et un analyseur syntaxique XML et je traiterai aussi d'autres applications intéressantes comme les modèles (*templates*) HTML et d'un système de blocage de *spam* POP3.

J'ai appris Unix avec le shell Bourne originel. Et, après une expédition dans la jungle des langages, je suis revenu au shell. Dernièrement, j'ai contribué à de nouvelles fonctionnalités de Bash, rendant la monnaie de leur pièce aux autres langages de scriptage. Slackware est ma distribution favorite depuis le début, parce que je peux saisir au clavier. Dans ma boite à outils, j'ai Vim, Bash, Mutt, Tin, Tex/Latex, Python, Awk et Sed. Même ma ligne de commande est en mode Vi.