Tool Command Language

Matthieu Herrb, Anthony Mallet
basé sur le cours de E.J. Friedman-Hill

6 décembre 2004

Introduction

Motivations : tout programme suffisamment important utilise au moins deux langages.

- un langage compilé, efficace pour l'algorithmique (le cobol, le C, C++, JAVA),
- un langage interprété, utilisé comme colle pour « piloter » et personnaliser les différentes fonctionnalités de l'application. $\longrightarrow$ **TCL**.

TCL = Tool Command Language

- Initialement développé pour UNIX à l'Université de Berkeley par le Dr. John Ousterhout (1988),
- Le langage est maintenant soutenu par la société ActiveState,
- License libre, open-source : TCL peut être librement modifié et intégré dans des applications commerciales,
- Denière version : 8.4.8 (Novembre 2004).

Prononciation : [tisil], on entend aussi « tickle ».

Tcl dans le groupe RIA

- **Calife/Jafar :** Tcl est utilisé comme langage de commandes
- **GDHE :** visualisation 3D (OpenGL) de manips de robotique. Tcl utilisé pour la programmation des modèles 3D des objets.
- **TclServ :** interface de contrôle des modules Genom à partir d'un interpréteur Tcl/Tk
- **Grh2 :** environnement graphique (plutôt 2D) utilisant TclServ pour communiquer avec les modules Genom et visualiser leurs données.

- Langage de commande **interprété** (mais compilé à la volée).
- **Multiplatforme**.
- Conçu pour être **étendu** ou **inclu dans une application** (C, C++, ...).
- Syntaxe tirée à la fois du SHELL, du C et du LISP.
- Définition de **variables** locales, globales ou à portée contrôlée.
- **Redéfinition dynamique** de toutes les commandes.

Extensions

- Programmation orientée objet.
- Compilateurs.
- Exécution autonome — sans interpréteur.

Nombreuses fonctionnalités intégrées :

- procédures,

```
proc fac x {
  if {$x <= 1} {return 1}
  expr $x*[fac [expr $x-1]]
}
```

```
fac 4
→ 24
```

- variables, tableaux associatifs, listes,
- expressions à la mode C,
- conditions, boucles:

```
if "$x < 3" {
    puts "x est trop petit"
}
```

- accès aux fichiers, aux processus, aux sockets (réseau).

Un seul type de données : chaînes de caractères terminées par un \0.

- conversion à la volée en cas de nécessité,
- programme et données interchangeables

```
set cmd "exec emacs"
...
eval $cmd
```

Depuis Tcl 8.3 : un type à part pour des données binaires sur 8 bits (exemple : images).

- `$foo`
  Un identificateur précédé d'un dollar est remplacé par la valeur de la variable correspondante.

- `[clock seconds]`
  Une commande entre crochets est exécutée et remplacée par le résultat de l'exécution.

- `"quelque chose"`
  Les double-quotes rassemblent plusieurs mots en en seul. Les dollars et les crochets sont interprétés normalement à l'intérieur.

- `{quelque chose}`
  Les accolades rassemblent plusieurs mots en un seul. Aucune interprétation des caractères spéciaux.

- `\`
  introduit des caractères spéciaux (`\n`) et supprime l'interprétation du dollar.

Grammaire

- `cmd arg arg...`
  une commande Tcl est composée de mots séparés par des espaces. Le premier mot est le nom de la commande et les autres sont les arguments.

- **Pas d'autre grammaire**.

Groupement et substitution

L'analyseur syntaxique de TCL procède en 3 étapes :

1. **groupement des arguments** sur la base des espaces entre les mots et des groupements à l'aide de quotes et d'accolades.

   Une commande est terminée par un retour à la ligne ou un point-virgule.

2. **substitutions** des valeurs des variables et des résultats de commandes entre crochets

3. **appel des commandes** en utilisant le premier mot de chaque commande comme index dans la table des procédures connues (intégrées ou définies par l'utilisateur en C ou en Tcl).

Expressions Tcl : `expr`

- À la mode C (int et doubles)
- Substitution de commandes et de variables *à l'intérieur* des expressions
- Évaluées par `expr`, `if`, `while`, `for`.

```
set b 5
⟶ 5
expr ($b*4) - 3
⟶ 17
expr $b <= 2
⟶ 0
expr 6 * cos(2*$b)
⟶ -5.03443
expr {$b * [fac 4]}
⟶ 120
```

Étudier à quel moment sont faites les substitutions dans la séquence :

```
set b \$a
set a 4
expr $b * 2
```

Exemple commenté

```
set i 0
while {$i < 10} {
   puts "$i au carré = [expr $i*$i]"
   incr i
}
```

- 2 commandes au 1^{er} niveau : `set` et `while`
- 2 arguments pour `while` : condition et corps
- attention à l'espace entre } et {
- évaluation automatique par `while` de la condition $\longrightarrow$ pas de `expr` dans ce cas.

Tableaux Tcl

Les tableaux Tcl sont *associatifs* : les indices sont des chaînes.

```
set x(fred) 44
→ 44
set x(2) [expr $x(fred) + 6]
→ 50
array names x
→ fred 2
```

On peut simuler des tableaux à plusieurs dimensions :

```
set a(1,1) 10
→ 10
set a(1,2) 11
→ 11
array names a
→ 1,1 1,2
```

Opérateurs sur les chaînes de caractères

Certains opérateurs fonctionnent avec des chaînes de caractères :

```
set a Bill
→ Bill
expr {$a < "Anne"}
→ 0
```

- <, >, <=, >=, == et != fonctionnent avec des chaînes
- Attention aux chaînes qui ressemblent à des nombres.
- On peut alors utiliser la fonction `string compare`.

- Zéro ou plusieurs éléments séparés par des espaces :

```
rouge vert bleu
```

- Accolades et backslashs pour grouper :

```
un\ mot deux trois
```

- Commandes sur les listes : `concat lindex llength lsearch foreach linsert lrange lsort lappend list lreplace`
- Les indices commencent à 0. `end` représente le dernier élément.

Exemples :

```
lindex "a b {c d e} f" 2
⟶ c d e
lsort {rouge vert bleu}
⟶ bleu rouge vert
```

Listes et commandes

- une commande *est* une liste
- `eval` permet d'évaluer une liste comme une commande
- pour créer une commande à évaluer, utiliser la commande `list`

```
button .b -text Reset -command {set x $initValue}
```

initValue est lue quand le bouton est utilisé

```
... -command "set x $initValue"
```

Erreur si initValue vaut "New York" (2 mots → set x New York)

```
... -command [list set x $initValue]
```

Toujours correct.

Manipulation de chaînes

- Commandes de manipulation de chaînes de caractères : `regexp format split string regsub scan join`
- sous-commandes de `string` : `compare first last index length match range toupper tolower trim trimleft trimright`
- tous les index commencent à 0, `end` représente le dernier caractère.

Globalisation « globbing »

Expansion des caractères spéciaux dans les noms de fichier à la mode du shell :

- `*` n'importe quelle séquence de caractères
- `?` un caractère quelconque
- [*liste*] un caractère de *liste*
- *c* matche *c* même si *c* est `*`, `?`, `[`, etc.

La commande `glob` expanse un motif en fonction des fichiers locaux.

```
foreach f [glob *.exe] {
    puts "$f est un programme"
}
```

Ne pas confondre avec les expressions régulières

Expressions régulières

Langage de description de motifs puissant. À la mode sed.

.	n'importe quel caractère
^	le début d'une chaîne
$	la fin d'une chaîne
x	supprime la signification de *x*
[*liste*]	un caractère la la *liste*
(*regexp*)	l'expression régulière *regexp*
*	0 ou plus de l'expression précédente
+	1 ou plus de l'expression précédente
?	0 ou 1 occurence de l'expression précédente
|	indique une alternative entre 2 expressions

Expressions régulières : exemples

- `[A-Za-z0-9_]+` : identificateurs Tcl valides
- `T(cl|k)` : Tcl ou Tk
- la commande `regexp` :
  `regexp T(cl|k) "Je parle de Tk" w t`
  $\longrightarrow$ `1`
  `w` vaut “Tk” et `t` vaut “k”.
- la commande `regsub` :
  `regsub -nocase perl "I love Perl" Tcl mantra`
  $\longrightarrow$ `1`
  `mantra` vaut “I love Tcl”

Les commandes `format` et `scan`

- `format` permet de formatter des chaînes à la mode C :
  ```
  format "Je connais %d commandes Tcl" 97
  → Je connais 97 commandes Tcl
  ```

- mêmes fonctionnalités que `printf()` en C.

- `scan` fonctionne comme `scanf()` :
  ```
  set x "SSN#:  1630799140049"
  → SSN#:  1630799140049
  scan $x "SSN#:  %d" ssn
  → 1
  puts "Numero de securite sociale:  $ssn"
  → Numero de securite sociale:  1630799140049
  ```

Structures de contrôle

- Ressemblent au C
- En réalité, ce sont des commandes Tcl comme les autres qui prennent des scripts en argument
- **Exemple :** retourne dans `b` la liste stockée dans `a`

```
set b ""
set i [expr [llength $a] -1]
while {$i >= 0} {
   lappend b [lindex $a $i]
   incr i -1
}
```

- Commandes : `if for switch break foreach while continue eval source`

Structures de contrôle : exemples

- `if` *expr* `?then?` *script*
- `if` *expr* `?then?` *script* `?else?` *script*
- `for` *script expr script script*

```
for {set i 0} {$i < 10} {incr i} {
   ...
}
```

- `switch`

```
switch -regexp $name {
   ^pete* {incr pete_count}
   ^(Bob|Robert)* {incr bob_count}
   default {incr other_count}
}
```

Structures de contrôle : pièges

Les accolades sont optionnelles, mais danger :

```
set x 3
   if $x>2 { ...
```

OK car la condition n'est évaluée qu'une fois

```
while $x>2 {...
```

Pas OK car la condition est évaluée plusiseurs fois.

`proc` définit une procédure :

```
proc sub1 x {expr $x-1}
⟶ sub1
sub1 3
⟶ 2
```

Les arguments peuvent avoir des valeurs par défaut :

```
proc decr {x {y 1}} {expr $x-$y}
⟶ decr
decr 3
⟶ 2
decr 3 2
⟶ 1
```

Procédures et portée des variables

- Portée des variables : variables globales et locales
  - l'interpréteur connaît les noms des variables et leur portée
  - chaque procédure introduit un nouvel espace de noms
- `global` permet d'accéder à une variable globale.
- `::variable` est équivalent à `global variable`.

```
set x 10
proc deltax {d} {
   set x [expr \$x-\$d]
}
```

```
deltax 1
→ can't read x:  no such variable
```

```
proc deltax {d} {
   global x
   set x [expr \$x-\$d]
}
```

```
deltax 1
→ 9
```

Portée des variables : upvar

- `global` est une commande ordinaire. On peut l'utiliser pour des effets tordus :

```
proc affecte {nom valeur} {
   global $nom
   set $nom $valeur
}
```

- `upvar` généralise ce principe :

```
proc incr {nom} {
   upvar 1 $nom var
   set var [expr $var+1]
}
```

- nommage des niveaux :
  - #0 global, #1 premier niveau d'appel, #2 deuxième, etc.
  - 0 courant, 1 niveau de l'appelant, 2 appelant de l'appelant, etc.

Procédures : listes d'arguments de longueur variable

```
proc sum args {
   set s 0
   foreach i $args {
      incr s $i
   }
   return $s
}
```

```
sum 1 2 3 4 5
→ 15
sum
→ 0
```

Espaces de nommage Tcl

- Pour des logiciels importants, il est nécessaire de structurer le code
  → espaces de nommage (*namespaces*).
- Notation : `Module::variable`
- Système hiérarchique
- Permet d'encapsuler des données.
- Commande `namespace` :
  - `namespace eval` *namespace arg...*
    Évalue *arg* dans l'espace désigné. Crée le namespace s'il n'existe pas.
  - `namespace export` *nom*
    Rend utilisable de l'extérieur d'un espace les variables ou les procédures désignées par *nom*.

Espaces de nommage : exemple

```
namespace eval Counter {
   namespace export bump
   variable num 0

   proc bump {} {
      variable num
      incr num
   }
}
```

- crée l'espace `::Counter`
- une commande visible `::Counter::bump`
- la variable `$Counter::num` contient la valeur courante.

- Une erreur avorte la commande en cours et provoque l'affichage d'un message

```
set n 0
→ 0
foreach i {1 2 3 4 5} {
set n [expr {$n + i*i}]
}
→ syntax error in expression "$n + i*i"
```

- La variable globale `errorInfo` contient la trace de la pile d'appels :

```
set errorInfo
→ syntax error in expression "$n + i*i"
while executing
"expr {$n + i*i}"
invoked from within
"set n [expr {$n + i*i}]..."
("foreach" body line 2)
...
```

Traitement des erreurs

- Interception des erreurs avec `catch`

```
catch {set x [expr $y + 3]} msg
→ 1
set msg
→ can't read "y":  no such variable
```

  - Retourne 1 si `y` n'est pas défini 0 sinon,
  - `msg` contient le message d'erreur,
  - l'exécution n'est pas interrompue.

- `error` permet de générer des erreurs:

```
error "bad argument"
return -code error "bad argument"
```

- la commande `unknown` est appelée lorsque le premier élément d'une liste à évaluer n'est pas reconnu comme une commande.

Accès aux fichiers

- Commandes d'accès aux fichiers : `open gets seek flush glob close read tell cd fconfigure fblocked fileevent puts source eof pwd filename`

- Les commandes utilisent des mots clefs pour désigner les fichiers ouverts.

```
set f [open "myfile.txt" "w"]
→ file4
puts $f "Ecriture dans le fichier"
→
close $f
→
```

- `gets` et `puts` travaillent ligne par ligne.

```
set x [gets $f]
```

  lit une ligne de `$f` dans `x`.

- modes configurables par `fconfigure`

Entrées/sorties et processus

- exec crée un nouveau processus, peut utiliser & comme le shell :

```
exec emacs &
eval exec "ls [glob *.c]"
```

- open permet de créer des tubes :

```
set f [open "|grep foo bar.tcl" "r"]
while {[eof $f] != 0} {
   puts [gets $f]
}
```

Informations sur le contexte d'exécution

- arguments sur la ligne de commande : `argc` contient leur nombre, `argv` est la liste des arguments, `argv0` est le nom de l'interpréteur
- version de Tcl/Tk: `tcl_version`, `tk_version`
- informations sur la machine : le tableau `tk_platform` avec les indices `osVersion`, `machine`, `platform`, `os`.
- la commande `info` :
  - sur les variables : `info vars`, `info globals`, `info locals`, `info exists`
  - sur les procédures : `info procs`, `info args`, `info body`, `info commands`

Chargement de code dans l'interpréteur

- `source` lit un fichier source Tcl et l'interprète
- `eval` évalue ses arguments
- `load` charge un module Tcl (un ensemble de fonctions C implémentant de nouvelles commandes

La philosophie

- Pourquoi écrire des scripts Tcl plutôt que du C :
  - développement plus rapide
  - plus souple
- Pourquoi écrire du C ?
  - Accès à des données de bas niveau (matériel)
  - Efficacité de l'exécution
  - Besoin de plus de structure
- Implémenter de nouvelles commandes Tcl à 2 niveaux:
  - bas niveau pour fournir un accès aux fonctions de base
  - haut niveau pour masquer des détails peu importants
- SWIG (`http://www.swig.org`).

À l'aide de modules

- Fonction d'initialisation du module `MonModule` :

```
int MonModule_Init(Tcl_Inter *interp);
```

- Installation de la fonction dans l'interpréteur
  Dans la fonction d'initialisation du module :

```
Tcl_CreateObjCommand(interp, "maCommande", myFunction, NULL, NULL);
```

- Prototype d'une fonction C implémentant une commande TCL:

```
int myFunction(ClientData clientData, Tcl_Interp *interp,
               int objc, Tcl_Obj *const objv[]);
```

clientData données utilisateur définies plus loin

interp interpréteur Tcl courant

objc nombre de paramètres

objv tableau d'objets Tcl $\longrightarrow$ liste des paramètres

Accès aux données Tcl depuis le C

- objets Tcl (`Tcl_Obj`): représentation des données Tcl soit sous forme de chaîne, soit sous forme binaire.
- création d'un objet Tcl : `Tcl_NewStringObj()`, `Tcl_NewDoubleObj()`, `Tcl_NewListObj()`, etc.
- conversion object Tcl en données C : `Tcl_GetIntFromObj()`, `Tcl_GetDoubleFromObj()`, etc.
- évalutation d'une expression Tcl : `Tcl_EvalObj()`
- résultat de la commande Tcl : `Tcl_SetObjResult()`

- compiler le fichier C pour produire un objet partagé :

```
cc -c -Kpic -g MonModule.c
ld -G -z text MonModule.o -o MonModule.so
```

- charger le module dans l'interpréteur Tcl `tclsh` ou `wish` :
  `load MonModule.so`
  $\longrightarrow$

- Tcl appelle automatiquement la fonction C `MonModule_Init()` pour initialiser le module.

- `MaCommande` devient appelable à partir de l'interpréteur.

Tcl dans le groupe RIA

- Version courante : 8.4.7 dans `/usr/local/tcl-8.4`
- Interpréteurs maison : `rtclsh` et `rwish` dans `/usr/local/robots/bin/${TARGET}`
- Interpréteurs maison : `eltclsh` et `elwish` dans `/usr/local/robots/${TARGET}/bin` (disponibles sur `http://softs.laas.fr/openrobots/`)
- Documentation :
  - `http://www.tcl.tk`
  - `http://wiki.tcl.tk`