Tcl/Tk

Mérési segédlet

Készítette: *Gaál Balázs*, gaal@ttt-atm.ttt.bme.hu

1. Bevezetés

1.1. Áttekintés

A *Tcl* és a *Tk* (ejtsd: "tikl" és "tikéj") két programcsomag, melyek segítségével *UNIX* környezetben *X-Windows* alapú grafikus felhasználói felületek fejleszthetők és futtathatók[1]. Ez a segédlet rödid áttekintést ad a Tcl nyelvről, a Tk alapjairól, és példaként bemutat egy bővítést, amellyel *TCP/IP socket*ek segítségével hálózati kommunikáció valósítható meg. A leírás minimális UNIX, X-Windows ill. TCP/IP előismeretet feltételez.

A Tcl (*Tool Command Language*) egy egyszerű interpreter alapú magasszintű "script" nyelv, amely leginkább a *perl* nyelvhez, vagy a UNIX *C-Shell* scripthez hasonít. Változók, tömbök, vezérlési szerkezetek, eljárások használhatók benne. Nagyon fontos tulajdonsága, hogy a C függvényekként megírt *Tcl interpreter* bármilyen C vagy C++ felhasználói programba beépíthető, és így a Tcl alapfunkciói kibővíthetők. A leírásban a 7.3-as verziót vesszük alapul.

A Tk (*ToolKit*) egy ilyen bővítés, amellyel *Motif* felhasználói felületet készíthetünk a beépített parancsok segítségével. Szinte minden feladat megoldható vele nagyságrendekkel rövidebb idő alatt, mint tisztán C-ben programozva. Ezen felül sokkal áttekinthetőbb és rugalmasabban módosítható a Tk program. A 3.6-os verzió lehetőségeit mutatjuk majd be röviden.

A *Tcl-DP* (*Distributed Programming*) egy a Tcl-höz és a Tk-hez adható bővítés, amely elosztott objektumorientált programozást, távoli eljáráshívást és TCP/IP socket kommunikációt támogat. A 3.2-es verzió néhány parancsáról lesz szó a későbbiekben.

A Tcl/Tk-nek négy fő előnye van:

- Egy felhasználói C vagy C++ program tartalmazhatja a Tcl vagy a Tk interpretert a beépített parancsaival együtt, és egyedül csak az alkalmazáshoz szükséges új parancsokat kell megírni C nyelven. Ehhez rendelkezésre állnak függvénykönyvtárak, amelyek egyszerűen beágyazhatóak, és nagyon sok kényelmes funkciót biztosítanak.
- Nagyon gyors fejlesztési ciklus. Mivel a C nyelvnél magasabb szintű, így rövid tanulás után néhány soros programokkal teljesértékű grafikus felületek készíthetők. A kód mérete és a fejlesztési idő minimum tizedakkora, mintha az *Xt/Motif toolkit* segítségével C-ben készült volna a program. Mivel a nyelv interpretált, újrafordítás és újraindítás nélkül lehet módosítani a kódot, így az új ötletek kipróbálása és a hibák kijavítása nagyon gyorsan végezhető. Ezért természetesen lassabb a végrehajtás mintha lefordított program lenne, de a mai munkaállomások teljesítményénél ez nem észrevehető. Ha a lassulás nem engedhető meg, akkor a kritikus részek kevés munkával átírhatók C-re.
- A Tcl ideális nyelv a programok közötti kommunikáció megvalósítására, mivel minden programba beágyazható az interpreter. A programok Tcl scripteket küldhetnek egymásnak, amelyekben nem csak

[1] Létezik kísérleti jellegű implementáció is MS-Windows alá.

parancsok, hanem változók, vezérlési szerkezetek vannak. A programban csak a parancsokat kell implementálni, a többit a Tcl megoldja.

- A nyelv áttekinthetősége miatt az alapkoncepció gyorsan megtanulható, és a bővítéseknél elég csak az új parancsokat megismerni.

1.2. Jelölések

A példákban a pontosan beírandó parancsok `kiemelt betűtípussal` lesznek jelölve. Pl.:

```
        set a hello
⇒       hello
```

A ⇒ jel jelenti választ, normál visszatérési érték esetén. A ∅ jel jelenti a hibás visszatérési értéket. Pl:

```
        set a hello 23
∅       wrong # args: should be "set varName ?newValue?"
```

A Tcl parancsok szintaxisának leírásánál *dőlt betűvel* jelöli a formális paramétereket, kérdőjelek között vannak az opcionális paraméterek, `...` jelöli az ismétlődést. Pl:

```
set varName ?newValue?
unset varName ?varName varName ...?
```

1.3. Használat

A Tcl és a Tk interpreterrel kétféleképpen dolgozhatunk:

- Interaktív módon, ilyenkor a terminálról beírt parancsok azonnal végrehajtódnak, és az eredmény kiíródik a képernyőre. Így könnyű kísérletezgetni, próbálgatni. Ebben a leírásban felsorolt példák nagy része legegyszerűbben és leglátványosabban így próbálható ki.
- Nem interaktív módon, ilyenkor egy file-ban tárolt Tcl scriptet hajt végre az interpreter.

tclsh

A Tcl interpretert interaktívan a

```
tclsh
```

parancs beírásával indíthatjuk el[2]. A következő sorban kapunk egy % promptot, ami azt jelenti, hogy az interpreter fogadja a parancsokat. A `tclsh` (*Tcl Shell*) tulajdonképpen egy *UNIX shell* (mint az `sh` vagy a `csh`), így alkalmas a beépített parancsokon kívül külső programok futtatására is.

wish

A Tk interpretere a `wish` (*Windowing Shell*). A `wish` ugyanúgy müködik, mint a `tclsh`, de indításkor megjelenít egy ablakot, ami a grafikus felület alapját fogja képezni.

Nem interaktívan pl. a `foo.tcl` file-ba kimentett tcl scriptet a

```
tclsh foo.tcl
```

paranccsal futtathatjuk. A wish esetében a

```
wish -f foo.tcl
```

parancsot kell alkalmazni.

---

[2] UNIX környezetben a programok futtatásához szükséges beállításokat a függelék tartalmazza.

A Tcl script file önmagában is végrehajtható (UNIX környezetben), ha a file-ra van végrehajtási jog (ez a `chmod a+x foo.tcl` paranccsal érhető el) és a file első sora a következő:

```
#!/usr/local/bin/tclsh
```

illetve wish esetén

```
#!/usr/local/bin/wish -f
```

A Tcl-DP bővítést mindkét program tartalmazza.

man

A `tclsh`-ról, a `wish`-ről, és minden Tcl/Tk parancsól rendelkezésre áll teljes on-line leírás, ún. *manual page*, amely a

```
man command
```

paranccsal tekinthető meg. Az online manual fejezetekre van osztva, a szövegben pl. *tclsh(1)* jelöli az 1. fejezet `tclsh` parancsát.

A könnyebb eligazodás kedvéért a leírásban a parancsok a lap bal szélén ki vannak elmelve, illetve mellékletként szerepel néhány referencia kártya.

2. A Tcl nyelv

Ez a fejezet a Tcl nyelv alapjait foglalja össze beleértve a szintaxist és a parancsok rövid leírását példákkal illusztrálva.

2.1. A Tcl nyelv szintaxisa

A Tcl *script* egy vagy több újsorral vagy pontosvesszővel elválasztott *parancs*ból áll.

Minden parancs egy vagy több *szó*ból áll, ahol az első szó a parancs neve, az azt követő szavak a parancs argumentumai. A szavakat szóközök vagy tabulátorjelek választják el egymástól. Egy parancs akárhány szóból állhat. A szavakat illetve a paracsokat elválasztó jelek nem részei egy szónak sem.

Minden parancs két lépésben értékelődik ki. Az első lépésben az interpreter szavakra bontja a parancsot és végrehajtja a helyettesítéseket. Ez a lépés minden parancsra ugyanaz, és az elemző (*parser*) végzi. A második lépésként végrehajtja az első szó átlal megnevezett parancsot, a többi szót átadva, mint argumentumot.

Az elemzés során háromféle helyettesítés lehetséges:

1. Változóhelyettesítés

A változók helyettesítése a `$` jel alkalmazásával érhető el. A `$` jel a szón belül bárhol lehet, és utána egy érvényes változónévnek kell állnia. Hatására a változó értéke helyettesítődik a szóba. Pl.:

```
set inches 12
expr $inches*2.54
⇒   30.48
```

Az első parancs az inches nevű változóba a "`12`" szót tölti. A második parancs elemzésekor a "`$inches`" helyettesítődik "`12`"-vel, így a végrehajtásnál az "`expr 12*2.54`" parancs hajtódik végre, aminek az eredménye "`30.48`".

Egyes esetekben az egyételműség miatt a `${`*`varName`*`}` forma is használható.

2. Parancshelyettesítés

A szögletes zárójelek közé zárt szó mint paracs értékelődik ki, és a visszatérési értéke kerül behelyettesítésre. A zárójelen belüli szónak egy érvényes Tcl script-nek kell lennie. Pl:

```
      set inches 12
      set mm [expr $inches*2.54]
⇒     30.48
```

A második `set` parancs a végrehajtás során csak a "`30.48`" szót "látja".

3. Backslash-helyettesítés:

Ez a helyettesítés használtató a speciális karakterek pl. "`$`", "`[`", "`;`", szóközök és újsorjel szóban való elhelyezésére. Az összes *ANSI C*-ben rögzített *escape szekvencia* használható (`\n` - újsor, `\t` - tab, `\\` - \ stb.) Pl:

```
      set msg Clock\ type\ 6342\nPrice:\ \$19.99
⇒     Clock type 6342
⇒     Price: $19.99
```

A backslash-újsor szekvencia használható hosszú sorok tördelésére is. Pl:

```
      set thisisaverylongvariablename \
          value
```

A tördelt sornak szóközzel vagy tabulátorjellel kell kezdődnie, és a tördelés szóelválasztásnak minősül.

Lehetőség van más módon is kérni az elemzőt karakterek speciális értelmezésének megszüntetésére (*quoting*). Két módon érhető el:

1. Idézőjelekkel:

Az idézőjelek közé tett karaktersorozat egy szónak minősül, bármi is van benne (szóköz, tab, újsor stb.). A változó-, parancs- és backslash-helyettesítések viszont ugyanúgy végrehajtódnak, mint egyéb esetben. Az idézőjel nem része a szónak. Pl. a

```
      set msg "Clock type 6342\nPrice: \$19.99"
```

és a

```
      set msg "Clock type 6342
      Price: \$19.99"
```

ugyanazt az eredményt adják, mint a fenti esetben.

2. Kapcsos zárójelekkel:

A kapcsos zárójelek között az elemző semmilyen helyettesítést nem hajt végre. (Kivétel ezalól a backslash-újsor szekvencia.) Pl.:

```
      set msg {Clock type 6342\nPrice: \$19.99}
⇒     Clock type 6342\nPrice: \$19.99
```

A kapcsos zárójelek legfontosabb felhasználása a késleltetett kiértékelés. Például az 5! értékének kiszámítása:

```
set result 1
set i 5
while {$i  > 0} {
     set result [expr $result*$i]
     set i [expr $i-1]
}
```

A kapcsos zárójelek miatt a `while` hurokban mindig a változók aktuális értéke helyettesítődik be.

Megjegyzéseket a sor elejére tett # jel után lehet írni - pontosabban olyan helyre, ahová parancs első karaktere kerülhet. Máshol elhelyezett # jel nem jelent megjegyést, az elemző közönséges karakternek veszi.

A helyettesítésekkel kapcsolatban két fontos szabályt kell megjegyezni:

1. Az elemzés egy menetben hajtódik végre balról jobbra. Minden egyes karakter pontosan egyszer kerül beolvasásra.
2. Legfeljebb egyszeri helyettesítés történik az egyes karaktereken, a behelyettesített érték nem kerül felhasználásra a további behelyettesítések során.

A következő példa illusztrálja ezt:

```
set x 1
set a x
set b $$a
```

A `b` változó értéke "`$x`" lesz, mivel a helyettesítés csak egyszer hajtódik végre.

A szabályok egyik következménye, hogy a

```
set city "Los Angeles"
set bigCity $city
```

script-ben a második parancs helyesen hajtódik végre, mivel a `city` változó értéke egy szóként helyettesítődik be.

Ez néha nem a kívánt viselkedés. Például a következő parancs hibás lehet:

```
exec rm [glob *.o]
∅   rm: a.o b.o c.o nonexistent
```

A `glob` parancs visszadja a mintára illeszkedő fileneveket, az `exec` parancs megpróbálja végrehajtani az `rm` UNIX parancsot a visszakapott "`a.o b.o c.o`" filenéven, ami nem létezik. A helyes működéshez szükséges lenne a `glob` által visszadott érték felbontására. Ez a második elemzés kikényszeríthető az `eval` paranccsal:

```
eval exec rm [glob *.o]
```

2.2. Változók

Az egyszerű változók egy névvel és egy értékkel rendelkeznek. A Tcl-ben a változónevek és értékek tetszőleges karaktersorozatok lehetnek. A változóknak nincs típusa, minden érték string-ként tárolódik. A változóhozzárendelés dinamikus, bármikor készíthetők és törölhetők.

set A `set varName ?value?` paranccsal készíthetők, módosíthatók, olvashatók változók. Ha a *`value`* meg van adva, akkor a *`varName`* változó a *`value`* étéket veszi fel. A visszatérési érték minden esetben a változó új értéke.

Az egyszerű változókon kívül asszociatív tömbök is használhatóak. Egy tömb az elemek gyűjteménye, amelyek maguk is változók saját névvel és értékkel. Pl.:

```
        set uid(root) 1000
⇒       1000
        set uid(guest) 200
⇒       200
        set uid(root)
⇒       1000
```

A tömböket nem kell előre delkarálni, elemszáma tetszőlegesen változhat. A változóhelyettesítés a tömbökre is működik:

```
        set login guest
        set a [expr $uid($login)+1]
⇒       201
```

unset Az `unset` paranccsal törölhetők változók, tömbelemek, vagy egész tömbök:

```
        unset login
        unset uid(guest)
        unset uid
```

incr Az `incr varName ?increment?` paranccsal egész értékű változók értéke növelhető *`increment`* értékkel, vagy ha ez nincs megadva eggyel.

append Az `append varName value ?value ...?` parancs a változó értéke után illeszti a *`value`* értékeket.

2.3. Kifejezések

expr Az `expr arg ?arg ..?` parancs kiértékeli az argumentumai által megadott kifejezést és a visszatérési értéke az eredmény.

A numerikus értékek szintaxisa megegyezik az ANSI C-ben definiáltakkal (decimális, oktális, hexadecimális, lebegőpontos). Minden ANSI C operátor és nagyon sok numerikus függvény használható az ott megszokott tulajdonságokkal, azzal a többlettel, hogy a relációs operátorok stringekre is alkalmazhatóak.

A különböző típusú argumentumok automatikusan konvertálódnak, de explicit konverzió is lehetséges a `double`, az `int` és a `round` függvényekkel. Bővebb információk az *expr(n)* on-line leírásban találhatók.

2.4. Listák

A Tcl-ben listának hívják elemek egy rendezett együttesét. A lista elemei szavak egymástól szóközzel vagy tabulátorjellel elválasztva:

```
        Apple Orange Strawberry Lemon
```

lindex Az `lindex` *`list index`* visszaadja a lista adott elemét. Az elemek indexelése 0-tól kezdődik.

```
        lindex {Apple Orange Strawberry Lemon} 1
⇒       Orange
```

A parancsokban előforduló listák általában kapcsos zárójelek között szerepelnek, mivel egy szót képeznek, de a zárójelek nem részei a listának:

```
        set fruits {Apple Orange Strawberry Lemon}
⇒       Apple Orange Strawberry Lemon
```

A lista elemei lehetnek listák is:

```
        lindex {a b {c d e} f} 2
⇒       c d e
```

Két parancs áll rendelkezésre, amivel listák készíthetők: a `concat` és a `list`.

concat A `concat` *`list ?list ...?`* az argumentumaiban megadott listákat egyetlen listává egyesíti:

```
        concat {a b c} {d e} f {g h i}
⇒       a b c d e f g h i
```

list A `list` *`value ?value ...?`* az argumentumait egyenként listaelemeknek veszi:

```
        list {a b c} {d e} f {g h i}
⇒       {a b c} {d e} f {g h i}
```

A `list` parancs mindig helyes formátumú listát ad vissza az argumentumaitól függetlenül, ha szükséges backslash-ek vagy kapcsos zárójelek hozzáadásával. A `concat`-nál ez nem garantált.

llength Az `llength` *`list`* parancs visszaadja a lista elemeinek számát:

```
        llength {{a b c} {d e} f {g h i}}
⇒       4
        llength a
⇒       1
        llength {}
⇒       0
```

linsert Az `linsert` *`list index value ?value ...?`* parancs visszaadja a listát, amelybe a megadott indexű elem elé beszúrja a megadott értékeket (ha az index nagyobb vagy egyenlő mint az elemek száma, akkor a lista végére illeszti):

```
        linsert {{a b c} {d e} f {g h i}} 1 A B C
⇒       {a b c} A B C {d e} f {g h i}
```

lreplace Az `lreplace` *`list first last ?value value ..?`* parancs visszaadja a listát, amiből a *`fist`* és a *`last`* indexű elemeket kitörölte, és - ha vannak - helyettesítette a *`value`* paraméterekkel:

```
        lreplace {{a b c} {d e} f {g h i}} 2 2
⇒       {a b c} {d e} {g h i}
        lreplace {{a b c} {d e} f {g h i}} 0 1 X {A B} Y
⇒       X {A B} Y f {g h i}
```

lrange Az `lrange` *`list first last`* parancs visszaadja a lista egy részét a *`first`* indexű elemétől kezdve a *`last`* indexű eleméig (ha a *`last`* index `end`, akkor a végéig).

```
        lrange {{a b c} {d e} f {g h i}} 1 2
⇒       {d e} f
        lrange {A B C D E F} 2 end
⇒       C D E F
```

lappend Az `lappend` *`varName value ?value ...?`* parancs hozzáilleszti a megadott listaértékeket a *`varName`* nevű változóhoz:

```
     set l {A B C D E F}
⇒    A B C D E F
     lappend l X {Y Z}
⇒    A B C D E F X {Y Z}
     set l
⇒    A B C D E F X {Y Z}
```

Az `lappend`, ugyanúgy mint az `append` nem feltétlenül szükséges parancs, mivel más parancsokból felépíthető, de hosszú listákra nagyon hatékony.

split A `split` *`string ?splitChars?`* felbontja a megadott stringet, és visszadja, mint listát. Az elválasztó karakter(ek) a *`splitChars`* argumentumban adható(k) meg:

```
     set f /usr/local/lib/libtcl.a
     split $f /
⇒    {} usr local lib libtcl.a
```

join A `join` *`list ?joinString?`* ennek a fordítottját csinálja:

```
     join {{} usr local lib libtcl.a} /
⇒    /usr/local/lib/libtcl.a
```

2.5. Vezérlési szerkezetek

Kétfajta feltételes parancs van: az `if` és a `switch`.

if Az `if` *`test1`* `?then?` *`body1`* `?elseif` *`test2`* `?then?` *`body2`* `elseif ..? ?else?` *`?bodyn?`* parancs kiértékeli a *`test1`* kifejezést, ha nemnulla értékű, akkor végrehajtja a *`body1`* scriptet, és visszadja az értékét. Különben kiértékeli a *`test2`* kifejezést, ha nemnulla értékű, akkor végrehajtja a *`body2`* scriptet, és visszadja az értékét, és így tovább. Ha egy teszt sem volt sikeres, akkor végrehajtja a *`bodyn`* scriptet, és visszadja az értékét. Pl.:

```
if {$x < 0} {
     set x 0
}
```

Az `if`-nél és a többi vezérési szerkezetnél a paraméterekként szereplő kifejezéseket általában kapcsos zárójelek közé érdemes tenni, hogy a kiértékelés a megfelelő időben töténjék. Minden nyitó kapcsos zárójelnek ugyanabban a sorban kell lennie, mint az előző szónak. A következő script tehát hibás:

```
if {$x < 0}
{
     set x 0
}
```

switch A `switch` *`?options? string pattern body ?pattern body ...?`* parancs illeszti a *`string`*-et minden egyes *`pattern`* mintára amíg egyezést nem talál, ekkor végrehajtja a hozzá tartozó *`body`* scriptet és visszatér. Ha az utolsó *`pattern`* `default`, akkor ez mindenre illeszkedik. Az *`options`* lehet `-exact`, `-glob`, `-regexp` aszerint, hogy pontos, glob stílusú, vagy reguláris kifejezés szerinti illesztést akarunk (az alapaértelmezés `glob`). (lásd később). A

```
switch $x a {incr t1} b {incr t2} c {incr t3}
```

forma írható így is:

```
switch $x {
     a {incr t1}
     b {incr t2}
     c {incr t3}
```

```
}
```

Ha a *body* parancs "-", akkor a következő minta parancsát hajtja végre, így lehet több mintához egyazon parancsot rendelni. Pl:

```
switch $x {
     a -
     b -
     c {incr t1}
     d {incr t2}
     default {incr t3}
}
```

Ha az x változó értéke "a", "b" vagy "c", akkor a t1 változó értékét növeli, ha "d", akkor a t2-ét, más értékek esetén a t3-ét.

Három ciklusutasítás van: a while, a for és a foreach.

while A while *test body* parancs kiértékeli a *test* kifejezést és ha az nem-nulla, vérehajtja a *body* scriptet, majd újra kiérékeli a kifejezést. Ezt ismétli egészen addig, amíg a kiértékelés nullát ad eredményül, ezután üres stringgel tér vissza. A következő script az a lista elemeit fordított sorrendben másolja a b listába:

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

for A for *init test reinit body* parancs végrehajtja az *init* scriptet, utána kiértékeli a *test* kifejezést, ha ez nem-nulla, akkor végrehajtja a *body* scriptet, majd a *reinit* scriptet, majd újra kiértékeli a kifejezést. Ezt ismétli egészen addig, amíg a kiértékelés nullát ad eredményül, ezután üres stringgel tér vissza. Az előbbi példa for-ral megvalósítva:

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

foreach A foreach *varName list body* parancs a megadott lista sorrendben minden elemére beállítja a *varName* nevű változót, és végrehajtja a *body* scriptet. Ezzel könnyűvé válik listák feldolgozása. A fenti példa megalósítása foreach segítségével:

```
set b ""
foreach i $a {
     set b [linsert $b 0 $i]
}
```

Használhatók a C-ben megszokott break és continue parancsok is.

break A break megszakítja a legbelső ciklus végrehajtását.

continue A continue pedig azonnal a legbelső ciklus következő iterációjára ugrik.

eval Az eval *arg ?arg ..?* egy általánosan használható parancs scriptek készítésére és végrehatására. Az egyik alkalmazása a változókban tárolt parancsok végrehajtása:

```
set cmd "set x 0"
eval $cmd
```

A másik fontos alkalmazás második elemzés elvégzése, amit korábban már láttunk.

source A `source` *`fileName`* parancs beolvassa és végrehajtja a megadott script file-t.

2.6. Eljárások

A Tcl-ben sokrétűen paraméterezhető eljárások definiálhatók visszatérési értékkel.

proc A `proc` *`name arglist body`* parancs létrehoz egy *`name`* nevű, *`arglist`*-ben felsorolt paraméterlistájú eljárást *`body`* törzzsel. Pl.:

```
     proc add {a b} {expr $a+$b}
     add 12 4
⇒    16
     add 3
∅    no value given for parameter "b" to "plus"
```

return Az eljárás visszatérési értéke az utolsó parancs visszatérési értéke. A `return` parancs használatával azonnal vissza lehet térni a megadott értékkel.

global Az eljárás törzsének kiértékelésekor a létrejövő változók automatikusan lokálisak. Globális változókat a `global` *`name1 ?name2 ...?`* paranccsal érhetünk el. A felsorolt nevű - nem szükségképpen létező - változók ilyenkor globális hatáskörűek.

Lehetőség van paraméter alapértelmezések beállítására. Ha híváskor nincs megadva argumentum, akkor az eljárás az alapértelmezést használja. Ha alapértelmezés van egy paraméterre beállítva, akkor az összes ezután következő paraméternek is ilyennek kell lennie. Pl.:

```
     proc inc {value {increment 1}} {
          expr $value+$increment
     }
     inc 23 4
⇒    27
     inc 62
⇒    63
```

Ha az argumentumlista utolsó eleme `args`, akkor híváskor változó számú paraméter adható át. Az `args` lista fogja tartalmazni a paramétereket (amely üres lista is lehet). Pl.:

```
     proc sum args {
          set s 0
          foreach I $args {
                incr s $I
          }
          return $s
     }
     sum 1 2 3 4 5
⇒    15
     sum
⇒    0
```

Az eljáráson belülről a `global` parancson kívül máshogy is hozzá lehet férni a hívó eljárás változóihoz. Az `upvar` és az `uplevel` paranccsal tetszőleges hívási szint változói elérhetők. További információ az on-line leírásban található.

2.7. Hibák kezelése

Ha egy parancs végrehajtása közben hiba lép fel, akkor a program futása megszakad, és hibaüzenet íródik ki. Az `errorInfo` globális változóban található további információ a hiba okáról.

catch Szükség lehet a hibák programon belüli lekezelésére. Erre használható a `catch` *`command ?varName?`* parancs, ami végrehajtja a megadott parancsot, és ennek eredményét a *`varName`* nevű változóban helyezi el (hiba esetén a hibaüzenetet). A `catch` parancs visszatérési értéke a végrehajtott parancs visszatérési hibakódja lesz (`0`, ha sikeres).

error Eljárásból hibával visszatérni az `error` *`message ?info? ?code?`* paranccsal lehet, ahol *`message`* a hibaüzenet, *`info`* az `errorInfo` változó értéke, a *`code`* a visszatérési hibakód (általában `1`).

2.8. Stringkezelés

string A Tcl-ben sok stringkezelő parancs van. Ezekből sokat a `string` parancs valósít meg. A parancsnak számos alparancsa van. Például a `string index` és a `string range` parancsok ugyanazt a feladatot oldják meg stringeken, mint az `lindex` és az `lrange` listákon. A `string first` *`string1 string2`* ill. a `string last` *`string1 string2`* parancs megkeresi a *`string1`*-et a *`string2`*-ben balról, ill. jobbról. A `string compare` funkciójában megfelel az `strcmp()` C függvénynek.

format Formázott stringek készíthetők a `format` *`formatString ?value`* `value ...?` paranccsal, amelynek használata megegyezik az ANSI C `sprintf()` függvényével.

scan A `scan` *`string format varName ?varName varName`* `...?` paranccsal lehet a megadott *`string`*-et egy adott formátumra illeszteni, és a változókat eszerint feltölteni. Használata hasonló, mint az `sscanf()` ANSI C függvényé.

string Mintaillesztésre használható a `string match` *`pattern string`* parancs, amellyel *glob* stílusú mintát illeszthetünk. A glob minta tartalmazhat "`*`" és "`?`" karaktereket, amelyek értelmezése a szokásos. Szögletes zárójelek között megadott karakterek bármelyike illeszkedik a karakterre: pl. a "`[ch]`" minta a "`c`"-re vagy "`h`"-ra, az "`[a-z]`" minta minden kisbetűre illeszkedik. Az említett karakterek különleges értelmezése a "`\`" jellel feloldható.

A `regexp` és a `regsub` parancsokkal UNIX reguláris kifejezésekkel illeszthetünk. Részletes útmutató az on-line leírásban található.

2.9. File- és proceszkezelés

A C-ben használatos filekezelő műveletek léteznek a Tcl-ben is.

open File az `open` *`name ?access?`* paranccsal nyitható. Az *`access`* értéke lehet `r`, `r+`, `w`, `w+`, `a` vagy `a+`, ez a file megnyitási módját adja meg. Az `open` egy *file azonosítóval* tér vissza, amivel a további műveletek során lehet hivatkozni a file-ra.

gets Megnyitott file-t soronként a `gets` *`fileId ?varName?`* paranccsal olvashatjuk. Ha a *`varName`* paraméter meg van adva, akkor ebbe a változóba helyezi a beolvasott stringet (újsorjel nélkül), és visszaadja a beolvasott karakterek számát (file vége esetén `-1`-et). Ha nincs megadva *`varName`*, akkor magát a stringet adja vissza.

puts Soronként írni a `puts ?-nonewline?` *`?fileId? string`* paranccsal lehet. Ha nincs file azonosító, akkor az `stdout`-ra ír, a `-nonewline` opció megadásával az újsor karakter automatikus kiírása elnyomható.

seek File-on belüli pozícionálás a `seek` *`fileId offset ?origin?`* paranccsal végezhető. Az *`origin`*-hoz képest (ami lehet `start`, `current` vagy `end` - az alapértelmezés `start`) a következő olvasás az *`offset`* által megadott byte-nál fog kezdődni.

tell, eof Az aktuális pozíciót a `tell` *`fileId`* parancs szolgáltatja. File vége esetén az `eof` *`fileId`* parancs 1-el tér vissza.

glob, file A `glob` és a `file` parancs segítégével lehet az aktuális könyvtár file-jairól információt kapni.

cd, pwd A `cd` és a `pwd` parancs használata megegyezik a UNIX megfelelőikével.

flush, close A kimeneti buffer a `flush` *`fileId`* paranccsal üríthető ki. A megnyitott file-okat a `close` *`fileId`* paranccsal kell lezárni (ez egy `flush` műveletet is eredményez).

exec A Tcl programból indíthatók UNIX proceszek és lehetőség van *pipeline*-ok használatára is. Az `exec` paranccsal indítható procesz ahol standard I/O átirányítások (`<`, `<<`, `>`, `|`) és háttérbeli futtatás (`&`) is lehetséges. Pipeline megnyitására az `open` parancs használható, a filenévnek ilyenkor a `|` jellel kell kezdődnie.

pid A `pid` paranccsal az aktuális procesz vagy megnyitott pipeline *UNIX procesz azonosítója* kérdezhető le. A UNIX környezeti változók az `env` beépített tömbben találhatók.

exit Az `exit` parancs használtató a proceszből való kilépésre a kilépési státusz megadásával.

2.10. Egyéb Tcl parancsok

A Tcl interpreter lehetőséget ad saját belső állapotának lekérdezésére és módosítására.

array, info Az `array` parancs az asszociatív tömbök méretét, elemeinek azonosítóját stb. adja vissza. Az `info` paranccsal a létező globális és lokális változók nevét, eljárások nevét, paraméterezését, törzsét, parancsok nevét, az interpreter verziószámát stb. lehet lekérdezni. Az `info` és az `array` parancsok több alparancsot tartalmaznak, lásd on-line leírás.

trace A Tcl változók használata követhető programból is a `trace` parancs segítségével. Minden változóhoz rendelhető egy eljárás, ami az adott esemény bekövetkezésekor meghívódik. Az esemény lehet olvasás, módosítás vagy megszüntetés. Lásd *trace(n).*

rename A parancsok tetszőlegesen átnevezhetők vagy törölhetők a `rename` parancs segítségével.

unknown Speciális lehetőség az `unknown` parancs használata. Ez a parancs akkor hajtódik végre, ha az interpreter nem talál egy parancsot. Új `unknown` eljárás definiálásával módosíthatjuk az eredeti parancsot. A következő példa lehetővé teszi parancsok rövidített használatát, amíg az egyértelműség engedi:

```
proc unknown {name args} {
     set cmds [info commands $name*]
     if {[llength $cmds] != 1} {
          error "unkown command \"$name\""
     }
     uplevel $cmds $args
}
```

3. A Tk toolkit

A Tk toolkit segítségével *X-Windows* (*X11*) alapú grafikus felhasználói felületeket készíthetünk Tcl nyelven. Ugyanúgy, mint a Tcl, ez is C könyvtári függvénycsomagként felhasználható C vagy C++ programokban. A bemutatott példák a `wish` shell elindítása után próbálhatók ki.

A grafikus felület alapelemeit *widget*eknek hívják (window object). A widgetek *osztály*okba sorolhatók, mint például nyomógombok, szövegek, keretek, görgetősávok stb. A beépített widget osztályok megfelelnek az *OSF Motif* ajánlásnak. Az egyes widget *objektum*oknak van egyedi neve és vannak rá jellemző tulajdonságai.

Minden widget egy X *ablak*, amely tartalmazhat további ablakokat (widgeteket). Így egy fa szerkezetű hierarchia képezhető, és ez a widgetek nevében is megjelenik. Az elnevezés hasonlóan történik a UNIX filerendszeréhez, csak itt a "`/`" karakter helyett a "`.`" karaktert használják. A gyökér a "`.`" nevű ablak, a *main window*, aminek az összes ablak a leszármazottja. Pl. a "`.a`" nevű widget szülője a "`.`", a "`.a.b`" szülője a "`.a`".

A Tk automatikusan elkészíti a "`.`" ablakot, ami a képernyő *root* ablakának a gyermeke lesz, és a *window manager* (például a *Motif Window Magager*, az `mwm`) tesz hozzá keretet. Az ezután elkészített ablakok, ennek a gyermekei lesznek, és ezen az ablakon belül helyezkednek el (*internal window*). Szükség lehet különálló ablakokra is, amelyek a főablaktól függetlenül mozgathatók, átméretezhetők, ikonizálhatók stb. Ezek a *toplevel* ablakok, de valójában ezek is a "`.`" widget gyermekei (amely maga is toplevel).

A Tk programok *eseményvezéreltek*. Ez azt jelenti, hogy a program két részből áll:

- *Inicializáló* parancsokból, ami a futás elején létrehozza az ablakokat, és kezdőállapotba hozza az alkalmazást.
- *Eseménykezelő* parancsokból, amelyek az egyes widget-ekhez vannak rendelve, és bizonyos esemény bekövetkezésekor hajtódnak végre. Például egy nyomógomb widget-hez kell rendelni azt a parancsot, amit a gomb lenyomásakor végre kell hajtani.

Miután az inicializáló rész lefutott, a Tk automatikusan egy *eseményhurok*ba kerül, ahol feldolgozza az *X-windows* által küldött eseményeket, és végrehajtja a hozzájuk rendelt eseménykezelő scripteket.

A Tk négy fő parancskészletet nyújt:

1. Widgetek készítése

Widgetek létrehozására az osztály nevével egyező parancsok szolgálnak. Például a következő parancs lérehoz egy nyomógombot, amely piros színű "`Hello, world!`" szöveget tartalmaz:

```
button .b -text "Hello, world!" -foreground red \
          -command "exit"
```

Minden widget létrehozása ehhez hasonló. Az első paraméter a widget neve, a többi pedig *konfigurációs opció*, amivel a widget tulajdonságait állíthatjuk be. Az opciók két szóból állnak, az első az opció neve, a második az értéke. Minden widget osztályra definiált, hogy milyen opciók használhatók. A `-command` opció adja meg, hogy a nyomógomb lenyomásakor mit kell tenni (a példában kilépés a programból).

2. Widgetek elhelyezése a képernyőn

Az íly módon létrehozott ablak még nem jelenik meg automatikusan a szülőjében. Az elhelyezést, illetve a widget méretét az ún. *geometry manager* határozza meg. A *placer* egy egyszerű implementáció, amelynél meg kell mondani, hogy "helyezd a .x nevű ablakot (10,100) helyre, és legyen a mérete 2cm x 1cm". A másik, a *packer* nevű az általánosan használt. Képes automatikusan meghatározni a kellő méreteket és alkalmazkodik a változtatásokhoz (pl. a felhasználó növeli a top-level ablak méretét). Az előzőleg kiadott `button` parancs a

```
pack .b
```

parancs végrehajtása után jelenik meg ténylegesen a `wish` elindítása után létrejövő ablakban.

3. Kommunikáció létező widgetekkel

Miután a nyomógomb lérejött, létrejön a nevével egyező nevű widget parancs (*widget command*) is, amellyel kommunikálni tudunk az objektummal. Ezzel állíthatók és kérdezhetők le az opciók, műveletek hajthatók végre:

```
.b configure -foreground blue
.b flash
.b invoke
```

Az első parancs beállít egy opciót, az előtérszínt. A második hatására a gomb felvillan, a harmadik hatására pedig úgy viselkedik, minha lenyomták volna. A `configure` alparancs mindig használható opciók beállítására, a többi alparancsot az adott osztály definiálja. Az on-line leírásban megtalálható az összes osztály ismeretője (pl. *button(n)*). Ebben szerepel az opciók felsorolása és a widget parancsok alparancsainak leírása. Az általánosan használt opciók az *options(n)*-ben találhatók.

4. Kommunikáció widgetek között

Lehetőség van widgetek egymás közötti információcseréjére is, például egy görgetősáv értesíti a hozzá kapcsolt szövegablakot, ha a felhasználó elmozdította. Kommunikácó történhet az eseménykezelővel és a window manager-rel, és más Tk applikációknak is küldhető parancs.

3.1. Widget osztályok

Ebben a fejezetben rövid áttekintés található a Tk widget osztályairól. Szemléltető programok a

```
/usr/local/lib/tk/demos
```

könyvtárban találhatók. A `widget` nevű script bemutatja az összes widgetet és azok képességeit.

frame A `frame` használható ablakok keretezésére többféle különböző árnyékolt 3D megjelenéssel (*relief*). Másik fő feladata az ablakok csoportosítása, amivel lehetővé válik a struktúrált elhelyezés a packer segítségével.

toplevel A `toplevel` widget különálló ablakként jelenik meg, a képernyőn bárhová elhelyezhető, általában dialógusdobozok és különálló panelek elkészítésére használhatjuk.

label A `label` képes megjeleníteni szöveget vagy bittérképet.

button A `button` hasonló a label-hez, de mint aktív elem megváltozik a színe, ha a mutató fölé kerül, és az egér 1-es gombjának lenyomására "benyomódik" és a gomb felengedése után végrehajtja a megadott parancsokat. A `button`, és az összes többi aktív widget letiltható (*disable*), ilyenkor nem reagál az egérműveletekre.

checkbutton A `checkbutton` mindazt tudja mint a `button`, de kétállapotú, amit egy négyzet színe jelöl. Bináris választások megadására használható. A gombhoz hozzárendelhető egy Tcl változó egy "*on*" és egy "*off*" értékkel, amely mutatja a gomb aktuális állapotát, illetve ennek a változónak a módosításával, a gomb állapota is változtatható.

radiobutton A `radiobutton` az előző osztályhoz hasonlít, de itt több összerendelt gomb közül csak az egyik lehet bekapcsolt állapotú, amit egy sarkára állított négyzet színe jelöl. Egymást kölcsönösen kizáró választások megadására használható. Az összetartozó gombokhoz ugyanaz a változó van hozzárendelve, és az egyes gombokhoz rendelt "on" érték közül mindig a kiválasztott gombhoz tartozót tartalmazza. A változó állításával a kijelölés is megváltoztatható.

menu A `menu` és a `menubutton` widgetek használhatók legördülő (*pulldown*) és felbukkanó (*popup*) menük készítésére. A menü elemei lehetnek:

- `command (button)` : parancsot hajt végre
- `checkbutton`: kétállapotú kijelölés
- `radiobutton`: választás több lehetőség közül
- `cascade`: almenü megjelenítése
- `separator`: elválasztó vonal

A menüpontok kiválasztása történhet egérrel, a legördülő menünél az "`Alt`" billentyű segítségével (*keyboard traversal*) és mindkét menünél az elemhez rendelt billentyűkombináció lenyomásával (*keyboard shortcut*).

listbox A `listbox` widget megjelenít egy listát, amiből egy vagy több elem kiválasztható. A listához elemek adhatók, belőle elemek törölhetők illetve lekérdezhetők.

entry Az `entry` egysoros szöveges adatbevitelre használható. A beírt szöveg lekérdezhető, módosítható.

scrollbar A `scrollbar` (görgetősáv) helyezhető el például a `listbox` vagy az `entry` mellé, hogy az ablakból ki nem látszó részek is könnyen elérhetők legyenek. (Az említett és a többi görgethető widgetnél azonban nem feltétlenül szükséges a görgetősáv használata, mivel az 2-es egérgomb lenyomásával a tartalmuk görgethető.)

text A `text` widgettel többsoros (akár több ezer soros) szöveges információ jeleníthető meg vagy szerkeszthető. Műveletek végzésére az egérrel kijelölhetők részletek (*mark*). Egyes szövegrészek más-más színnel és betűtípussal jeleníthetők meg (*tag*), illetve a szövegbe widgetek is ágyazhatók.

canvas A `canvas`, mint egy rajzolófelület használható, amelyen elemek helyezhetők el. Egy elem (*item*) lehet egyenes, négyszög, sokszög, ellipszis, ív, görbe, szöveg, bittérkép, ikon, és bármilyen widget.

scale A `scale` egy számérték kiválasztására alkalmas megadott tartományból egy "potenciométer" mozgatásával. Az érték a widgethez rendelt változóban jelenik meg.

message A `message` widget használható többsoros egyszerű üzenet megjelenítésére.

3.2. Konfigurációs opciók

Minden widget objektumhoz 15-30 attribútum vagy más néven konfigurációs opció tartozik az opciók osztályokba vannak sorolva. Ebben a fejezetben a fontosabb opció osztályok áttekintése található.

Widgetek attribútumai négyféleképpen állíthatók:

1. A widget létrehozásakor a *`class window ?optionName value optionName value ...?`* formában.
2. Opció adatbázis használatával. Ha létrehozáskor nincs megadva egy bizonyos opció, akkor a Tk ugyanúgy mint más X-windows programok az `.Xdefaults` nevű file-ban tárolt adatok alapján állítja be az attribútumokat. Az opció adatbázis az `option` paranccsal is hozzáférhető.
3. Ha az opció adatbázisban nincs információ az adott opcióról, akkor a widget beépített alapértelmezését fogja a Tk használni.
4. Létező objektum a `configure` paranccsal konfigurálható át. A *`window`* `configure` *`optionName value`* parancs beállítja a *`window`* widget megadott opcióját. A *`window`* `configure` *`optionName`* visszaadja a *`window`* widget megadott opcióját egy listában, ami az opció nevét, osztályát, alapértelmezését és jelenlegi értékét tartalmazza. A *`window`* `configure` visszadja az összes opciót.

A *színek* legtöbbször a `-foreground` (előtérszín) és `-background` (háttérszín) opcióknál használatosak. Megadhatók névvel vagy az *RGB* színösszetevőkkel *#RGB* formában, ahol az *R*, a *G* és a *B* egy, két, vagy háromjegyű hexadecimális számok. (pl. `#bb00aa` vagy `#3f8`)

A *képernyőtávolságok* megadhatók egész számként képpontban mérve, illetve felbontástól függetlenül a szám után írt betűtől függően centiméterben (pl. `3c`), milliméterben (`30m`), inch-ben (`.5i`), 1/72 inch-ben (`24p`).

Bittérkép lehet beépített vagy külső file-ban tárolt. Nyolc beépített bittérkép van (`error`, `info`, `hourglass`, `question`, `warning`, `questhead`, `gray25`, és `gray50`). Ha a bittérkép neve `@` jellel kezdődik, akkor az *X bitmap file*-ként töltődik be.

A *betűtípusok* és *egérmutatók* megadása megfelel az X-windows konvencióinak. Az *időintervallumok* milliszekundumban adhatók meg.

A *horgony* (*anchor*) az ablakok adott koorditátájú pontra helyezésekor megadja, hogy az ablak mely pontja a bázispont. Ez az égtájak angol nevének rövidítésével adható meg:

```
nw     n     ne

w      c      e

sw     s     se
```

Az opciók között szerepelhetnek *parancs*ok is, ami például a `-command` opcióban megadja a kiválasztáskor végrehajtandó parancsot. Speciális parancsopciókkal kapcsolható össze egy listbox widget a hozzá tartozó scrollbar widgettel. A következő példában az .l lista a .v görgetősávval van összekapcsolva úgy, hogy egymás állapotát automatikusan befolyásolni tudják:

```
listbox .l -yscrollcommand {.v set}
scrollbar .v -orient vertical -command {.l yview}
pack .l -side left
pack .v -side right
```

Az elmozgatott vagy megváltozatott lista a "`.v set`" parancsot hajtja végre a megfelelő adatokkal kiegészítve, ami értesíti a `.v` görgetősávot az változásról. A görgetősáv eltolásának hatására a "`.l yview`" parancs hajtódik végre adatokkal kiegészítve, aminek hatására a `.l` lista elmozdul.

3.3. A packer

A packer geometry manager alkalmas ablakok elhelyezésére a szülő ablakban. Nagyon kevés opció megadásával is működik, és rugalmasan alkalmazkodik a szülő vagy egy gyermek méretének megváltozásakor.

pack

A `pack` *`window ?window? ?option value option value ...?`* parancs a felsorolt gyermek ablakokat a szülőben a megadott sorrendben helyezi el három lépésben:

1. A rendelkezésre álló négyszöletes hely `-side` opcióban megadott oldaláról (`left`, `right`, `top`, vagy `bottom`) "levág" egy *frame*-et. A frame mérete megegyzik a gyerek méretével ami az `-ipadx` és az `-ipady` opciókkal növelhető.
2. Ha a frame-ben még van szabad hely a gyerek mellett akkor a gyerek méretét a `-fill` opció által megadott irányban kiterjeszti (`none`, `x`, `y`, vagy `both`). Ha a gyermek nagyobb, mint a frame, akkor a gyermek méretét csökkenti.
3. Elhelyezi a gyereket a frame-ben az `-anchor` opcióban megadott helyre. A frame széle és a gyerek széle közötti távolság a `-padx` és a `-pady` opciókkal adható meg.

A megmaradt négyszögletes szabad helyből foglal helyet a következő ablaknak. A maradék helyet egyenlően töltik ki azok az ablakok, amelyenél az `-expand` opció `1`-re van állítva. Az `-in` opció megadásával a "természetes" szülőtől eltérő szülő jelölhető ki.

A `pack` parancsot általában hierachikusan használják, ahol a hierarchia fában a gyökér egy `toplevel` widget, a többi szülő pedig `frame` widget. Pl.:

```
pack .l -side left -padx 3m -pady 3m
pack .r -side right -padx 3m -pady 3m
pack .a1 .a2 .a3 .a4 .a5 -in .l -side top -anchor w
pack .b1 .b2 .b3 -in .r -side top -anchor w
```

Az `.l` nevű `frame`-ben az `.a1 .a2 .a3` `button` widgetek ill. a `.r` nevűben a `.b1 .b2 .b3` `button` widgetek vannak elhelyezve. A `pack`-ról részletes leírás a *pack(n)*-ben található.

3.4. Egyéb Tk parancsok

destroy A `destroy` *`window ?window ...?`* parancs megszünteti a megadott ablak(ok)at és azok gyermekeit (a "`destroy .`" kilép a programból).

bind A `bind` *`windowSpec sequence script`* paranccsal a *`sequence`*-ben meghatározott eseményhez rendelhetünk egy *`script`* parancssorozatot, amely végrehajtódik ha az esemény bekövetkezik *`windowSpec`*-ben megadott ablak(ok)ban. Az események jelölése megegyezik az X window által definiáltakkal.

tkerror Hiba bekövetkezése esetén a szabadon átdefiniálható `tkerror` eljárás hívódik meg, paramétreként a hibaüzenetet kapja meg.

focus A lenyomott billentyűkhöz tartozó eseményeket mindig az aktív ablak kapja. Ezt átalában a felhasználó az egérrel választja ki. Egyes esetekben szükség lehet ezt programból kijelölni. Erre szolgál a `focus` *`window`* parancs, amelyik a megadott ablakot teszi aktívvá. Az önmagában kiadott `focus` visszadja az éppen aktív ablak nevét.

grab Néha szükség lehet arra, hogy a felhasználó csak egy bizonyos ablakkal tudjon kapcsolatot tartani, más ablakokkal ne. Például egy fatális hiba esetén megjelenő dialógusdobozra kötelező válaszolni, és addig semmi mást nem tehet a felhasználó. Ezek a *modális* ablakok, amelyek lehetnek lokálisak, applikáción belüliek, vagy globálisak, egész képernyőre kiterjedőek. A `grab ?-global?` *`window`* paranccsal tehetünk egy ablakot modálissá. A `grab current` visszadja a modális ablak nevét, a `grab release` *`window`* pedig megszünteti a modalitást (a `destroy` paranccsal megszüntetett ablak elveszti a modalitását).

tkwait A programban várakozhatunk bizonyos események bekövetkezésére (például a modalitás feloldása előtt egy gomb lenyomására). A `tkwait variable` *`varName`* addig vár, amíg a megadott vátozó értéke meg nem változik. A `tkwait visibility` *`window`* parancs az ablak megjelenéséig, a `tkwait window` *`window`* parancs az ablak megszüntetéséig vár. Az `after` *`ms`* parancs a megadott számú milliszekundum hosszúságú ideig késleltet.

wm A window manager-rel való kapcsolattartásra használható a `wm` parancs. Ezzel lehet toplevel ablakokat mozgatni, átméretezni, ikonizáni, stb. Meghatározható a címsor, a megengedett mérettartomány és több más `toplevel` ablakra vonatkozó paraméter.

send A `send` *`appName arg ?arg arg ...?`* paranccsal egy másik futó Tk programnak küldhetünk az *`arg`* argumentumokban megadott parancsot, amit az végrehajt és a visszatérési érték a parancs visszatérési értéke lesz. A kommunikáló programoknak azonos ***X szerver***t kell használniuk.

winfo A futó applikációk nevét a `winfo interps` parancs adja vissza, a `send` parancs címzettjének ilyennek kell lennie. Ezzel "távvezérelhetünk" más Tk interpretereket.

3.5. Példák

A következő program az első `entry`-be beírt Celsius fok értéket a `<Return>` lenyomása után átszámolja Fahrenheit fokba és beírja a második `entry`-be. A második `entry`-be íráskor pedig visszafelé történik ugyanez.

```
#!/usr/local/bin/wish -f
entry .c -width 6 -relief sunken -textvariable c
label .label1 -text "Celsius is"
entry .f -width 6 -relief sunken -textvariable f
label .label2 -text "Fahrenheit"
pack .c .label1 .f .label2 -side left \
     -padx 1m -pady 2m
wm title . "Thermometer conversion"
bind .c <Return> {set f [expr 9*$c/5+32]}
bind .f <Return> {set c [expr ($f-32)*5/9]}
```

A `redo` program a beírt UNIX parancsot végrehatja, és nyomógombként megjeleníti, aminek megnyomásával az újra végrehajtható. Maximálisan öt nyomógomb tartalmazza az utoljára beírt parancsokat.

```
#!/usr/local/bin/wish -f
wm title . redo
set id 0
entry .entry -width 30 -relief sunken -textvariable cmd
pack .entry -padx 1m -pady 1m
bind .entry <Return> {
     incr id
     if {$id > 5} {
          destroy .b[expr $id-5]
     }
     button .b$id -text $cmd \
          -command "exec <@stdin >@stdout $cmd"
     pack .b$id -fill x
     .b$id invoke
     .entry delete 0 end
}
```

A dialog eljárás megjelenít egy modális dialógusdobozt és visszaadja a lenyomott nyomógomb sorszámát. Az eljárás a `dialog` *`w title text bitmap default button ...`* formában hívható. A megjelenő toplevel ablak neve *`w`*, címe *`title`* lesz. A *`text`* lesz a megjelenítendő üzenet, a *`bitmap`* - ha nem üres - akkor az üzenet jobb oldalán jelenik meg. A *`default`* az alapértelmezés szerinti nyomógomb száma (`-1`, ha nincs ilyen), majd ezután kell felsorolni a nyomógombok címkéit.

```
proc dialog {w title text bitmap default args} {
global button

# 1. Toplevel ablak létrehozása, alsó és felső részre
#    osztása

toplevel $w -class Dialog
wm title $w $title
wm iconname $w Dialog
frame $w.top -relief raised -bd 1
pack $w.top -side top -fill both
frame $w.bot -relief raised -bd 1
pack $w.bot -side bottom -fill both

# 2. A felső rész kitöltése az üzenettel

message $w.top.msg -width 3i -text $text\
         -font -Adobe-Times-Medium-R-Normal-*-180-*
pack $w.top.msg -side right -expand 1 -fill both\
         -padx 3m -pady 3m
if {$bitmap != ""} {
     label $w.top.bitmap -bitmap $bitmap
     pack $w.top.bitmap -side left -padx 3m -pady 3m
}

# 3. Az alsó rész kitöltése gombokkal

set i 0
foreach but $args {
     button $w.bot.button$i -text $but -command\
                "set button $i"
     if {$i == $default} {
          frame $w.bot.default -relief sunken -bd 1
          raise $w.bot.button$i
          pack $w.bot.default -side left -expand 1\
                     -padx 3m -pady 2m
          pack $w.bot.button$i -in $w.bot.default\
                     -side left -padx 2m -pady 2m\
                     -ipadx 2m -ipady 1m
     } else {
          pack $w.bot.button$i -side left -expand 1\
                -padx 3m -pady 3m -ipadx 2m -ipady 1m
     }
     incr i
}

# 4. A <Return> lenyomására az alapértelmezés aktiválása
#    a fókusz és modalitás beállítása

if {$default >= 0} {
     bind $w <Return> "$w.bot.button$default flash; \
          set button $default"
}
set oldFocus [focus]
grab set $w
focus $w


# 5. Várakozás gomb lenyomására, fókusz visszaállítása
#    a gomb sorszámának visszaadása

tkwait variable button
destroy $w
focus $oldFocus
return $button
}
```

Például a következő formaban hívhatjuk meg az eljárást:

```
dialog .d {File Modified} {File "tcl.h" has been \
     modified sincethe last time it was saved. Do you \
     want to save it before exiting the \
     application?} warning 0 {Save File} \
     {Discard Changes} {Return To Editor}

dialog .d {Not Responding} {The file server isn`t \
     responding right now; I`ll keep trying.} {} -1 OK
```

Ez az eljárás `tk_dialog` néven beépítetten is rendelkezésre áll.

4. A Tcl-DP

A Tcl-DP kiterjesztés lehetővé tesz Tcl programokban elosztott objektum orientált programozást, távoli eljáráshívást (*RPC - Remote Procedure Call*) és programok közötti kommunikációt TCP/IP socketeken keresztül. Ebben a fejezetben röviden a két utóbbiról lesz szó.

4.1. Távoli eljáráshívás

A Tk `send` parancsához hasonló funkciót valósítanak meg az RPC parancsok, azzal a különbséggel, hogy amíg a `send` az X szerveren keresztül küldte az információt, az RPC közvetlenül *TCP porton* át kommunikál.

A távoli eljáráshívás a következő lépésekben zajlik:

1. A *szerver* alkalmazás egy mindeki által ismert TCP porton keresztül fogadja a kapcsolatfelvételi kéréseket.
2. A *kliens* alkalmazás erre a portra küld egy kapcsolatfelvételi kérést, amit a szerver megvizsgál, és ha megfelelőnek találja, felveszi a kapcsolatot a klienssel.
3. Ezután bármelyik alkalmazás küldhet a másiknak Tcl parancsot, amit az vérgehajt és az eredményt visszaküldi. A vérgehajtás lehet szinkron - ilyenkor a parancs vérehajtása alatt a hívó várakozik, vagy aszinkron - ilyenkor a hívó alkalmazás fut tovább, és az eredmény megérkezésekor egy kijelölt parancs vérgehajtódik. Biztonsági okokból mindkét oldalon definiálható egy ellenőrző eljárás ami kiszűri a nem kívánt parancsokat (mivel pl. "`exit`" is küldhető, ami a távoli alkalmazás befejeződését eredményezi).
4. A kapcsolat biztonságosan lezárható - és le is kell zárni - mindkét oldalról (az elküldött parancsok nem vesznek el).

dp_MakeRPCServer A `dp_MakeRPCServer ?port? ?loginProc? ?cmdCheckProc? ?retFile?` paranccsal a szerver alkalmazás a megadott számú vagy nevű szabad portot fogalhatja le. (Ha nincs megadva port, vagy az `0`, akkor választ egyet). A *`loginProc`*-ban megadott eljárás minden kapcsolatfelvétel kérésnél végrehajtódik, paraméterként a kérő *internet címét* kapja. Ha ez az eljárás `error` paranccsal fejeződik be, akkor a kapcsolatot visszautasítja. Ha nincs megadva, akkor a beépített `dp_CheckHost` eljárás hívódik meg (lásd *dp_Host(n)*). A *`cmdCheckProc`*-ban megadott eljárás

használható a bejövő parancsok szűrésére. Ha nincs megadva, vagy az értéke `none`, akkor nincs ellenőrzés. A parancs a port számát adja vissza, illetve, ha a *`retFile`* nemnulla, akkor a *figyelő socket* azonosítóját is (lásd később).

dp_MakeRPCClient A `dp_MakeRPCClient` *`host port ?cmdCheckProc?`* paranccsal lehet a megadott hálózati című hoszton a megadott portra csatlakozni, mint kilens. A *`cmdCheckProc`* ugyanúgy állítható, mint a szervernél. Visszaadja a *socket azonosítót*, amire később hivatkozni tudunk.

dp_RPC A `dp_RPC` *`peer`* `?-events` *`events`*`? ?-timeout` *`ms`*`? ?-timeoutReturn` *`callback`*`?` *`command ?arg arg ...?`* parancs szinkron módon elküldi a *`peer`* socket azonosítójú alkalmazásnak a megadott parancsot az argumentumokkal együtt. Visszatérési értéke a távoli eljárás visszatérési értéke. Maximum az *`ms`* paraméterben megadott ideig vár, és ekkor a *`callback`* parancs végrehajtódik. (ha nincs megadva, vagy nulla, akkor nincs időkorlát). Várakozás alatt az *`events`* paraméterben megadott eseményeket dolgozza fel (bővebben lásd *dp_RPC(n)* és *Tk_DoOneEvent(3)*).

dp_RDO A `dp_RDO` *`peer`* `?-callback` *`resultCallback`*`? ?-onerror` *`errorCallback`*`?` *`command ?arg arg ...?`* parancs aszinkron módon küldi el a parancsot. Sikeres végrehajtás esetén a *`resultCallback`*, hiba esetén az *`errorCallback`* parancs értékelődik ki paraméterként az eredményt ill. a hibaüzenetet kapják.

dp_CloseRPC A kapcsolatot mindkét oldalon a `dp_CloseRPC` *`peer`* paranccsal kell lezárni.

A következő példában a szerver egyedi azonosítót szolgáltat a `GetId` hívással:

```
dp_MakeRPCServer 4545
set myId 0
proc GetId {} {global myId; incr myId}
```

A kilens például a következő lehet:

```
set server [dp_MakeRPCClient localhost 4545]
dp_RPC $server GetId
dp_CloseRPC $server
```

4.2. TCP kommunikáció

A TCP socketeken keresztül zajló kommunikáció hasonló fázisokból áll mint az RPC esetében. A szerver procesz egy ismert porton fogadja a kapcsolatfelvételi kéréseket, ezt hívják figyelő socketnek. A kérés elfogadásával jön létre a kapcsolat ami egy kommunikációs socket megnyitását jelenti. A socketeknek egyedi azonosítójuk van és a file-okhoz hasonlóan viselkednek, ugyanazokkal a parancsokkal lehet írni-olvasni őket, mint a file-okat. Mivel a filekezelő parancsok nem kezelik megfelelően a túloldalon lezárt socketeket, és nem biztosítják az üzenethatárok megtartását, vannak speciális socket kommunikációs parancsok.

A kapcsolat ideje alatt aszinkron kétirányú forgalom zajlik. A socket pillanatnyi állapota lehet olvasható (érkezett adat) és nem olvasható (nincs rendelkezésre álló adat) ill. írható (küldhető adat) vagy nem írható (a puffer tele van). A kommunikáció során ezeket figyelembe kell venni. Az adatcsere végeztével a socketeket le kell zárni a `close` paranccsal.

dp_connect A `dp_connect -server ?port? ?-linger? ?-reuseAddr?` parancs létrehoz egy figyelő (szerver) socketet a megadott porton (ha hiányzik, vagy nulla, akkor választ egy szabadot). Ha a `-linger` opció meg van adva, akkor lezáráskor nem veszhetnek el adatok. A `-resuseAddr` lehetővé teszi a port későbbi újrafelhasználását. Visszatérési érték a figyelő socket azonosító és a port.

A kliens a `dp_connect host port` paranccsal tud kapcsolódni a szerverhez. A figyelő socket olvashatóvá válik, ha kapcsolatfelvételi kérés jön be. Visszatérési érték a socket azonosító és a port.

dp_accept A kérést a `dp_accept sockId` paranccsal fogadhatja el a szerver, ahol a *sockId* a figyelő socket azonosítója, a visszatérési érték a létrejött socket azonosítója és a kérő internet címe. Ha a figyelő port nem olvasható, akkor addig vár, amíg kérés nem jön.

dp_send A `dp_send sockId message ?-nonewline?` parancs hasonló a `puts` parancshoz, de automatikusan lezárja a socketet, ha a távoli fél már lezárta. Visszatérési értéke az elküldött byte-ok száma. Ha a szabad puffer mérete kisebb, mint az üzenet, hossza, akkor addig vár amíg az összes adatot el nem tudja küldeni.

dp_receive A `dp_receive sockId ?numBytes? ?-peek?` parancs visszadja a socketen rendelkezésre álló adatokat, ha nem olvalható, akkor vár amíg az lesz. Ha a `-peek` opció meg van adva, akkor nem veszi el tényegesen az adatokat a pufferből. Automatikusan lezárja a socketet, ha a távoli fél már lezárta.

dp_packetSend A `dp_packetSend sockId message` parancs a `dp_send`-hez hasonlóan elküld egy üzenetet, de garantálja, hogy a vevő ezt egy egységben kapja meg.

dp_packetReceive A `dp_packetReceive sockId ?-peek?` a `dp_packetSend` paranccsal elküldött csomagot adja vissza. Egyebekben a `dp_receive`-hez hasonlít.

dp_isready A `dp_isready sockId` visszadja a socket állapotát egy listában. A lista első eleme olvashatóságot, a második az írhatóságot mutatja logikai értékként.

dp_filehandler A `dp_filehandler sockId ?mode command?` paranccsal megszabhatjuk, hogy a megadott socketen bizonyos események bekövetkezésekor eljárás hívódjék meg. A esemény lehet "`r`" - a socket olvashatóvá válik, "`w`" - a socket írhatóvá válik, "`e`" - a socketen kizárási esemény tötrénik, vagy ezek kombinációja. A *command* parancs az esemény és a socket azonosítójával, mint paraméterrel hívódik meg.

A következő példában a szerver procesz visszhangozza az érkező üzeneteket:

```
proc echo {socket mode} {
     set message [dp_receive $socket]
     dp_send $socket $message -nonewline
}
set lsid [lindex [dp_connect -server 4646] 0]
set sid [lindex [dp_accept $sid] 0]
dp_filehandler $lsid r echo
```

A kliens procesz a következőképpen nézhet ki:

```
proc reply {socket mode} {
     set message [dp_receive $socket]
     puts "Received: $message"
}
set sid [lindex [dp_connect localhost 4646] 0]
```

```
dp_filehandler $sid r reply
dp_send $sid "Hello!"
...
```

5. További információk

Részletes leírás a Tcl/Tk-ről a programcsomag készítője által írt könyvben található:

John K. Ousterhout: *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, 1994

A Tcl, a Tk és az összes többi bővítés forráskódja *anonymous ftp*-vel lehozható, és szinte minden UNIX rendszer alá lefordítható. A hivatalos európai ftp szerver címe: `ftp.ibp.fr /pub/tcl`. Helyben is megtalálható a `boss.ttt.bme.hu` szerveren az `/usr/local/archives/tcl` könyvtárban. Ugyanitt az említett könyv kézirata is megtalálható *PostScript* formában.

Komoly érdeklődők számára javasolható még a `comp.lang.tcl` *USENET newsgroup* böngészése.

6. Függelék

A `tclsh` és a `wish` futtatásához, ill. az on-line manual eléréséhez a következő környezeti változókat kell beállítani:

```
PATH            /usr/local/bin:$PATH
MANPATH         /usr/local/man
TCL_LIBRARY     /usr/local/lib/tcl
TK_LIBRARY      /usr/local/lib/tk
```

A környezeti változókat a `csh` és a `tcsh` shellekben a

`setenv` *`variable value`*

paranccsal, az `sh`, `bash`, `ksh`, `zsh` shellekben a

`variable``=`*`value`*`; export` *`variable`*

paranccsal állíthatjuk be.

A Tcl interpreter C programokba ágyazásához rendelkezésre áll az `/usr/local/lib` könyvtárban a `libtcl.a` függvénykönyvtár, amely a teljes interpretert tartalmazza, és az `/usr/local/includes` könyvtárban a `tcl.h` header file, amelyben a szükséges definíciók találhatók. További magyarázat helyett lássuk a következő C programot, amely végrehajt egy parancssorban átadott Tcl programot. (Érdekességképpen megjegyzendő, hogy a `tclsh` és a `wish` program maga sem sokkal hosszabb ennél.)

```
#include <stdio.h>
#include <tcl.h>

main(int argc, char *argv[]) {
     Tcl_Interp *interp;
     int code;

     if (argc != 2) {
          fprintf(stderr, "Wrong # arguments: ");
          fprintf(stderr, "should be \"%s fileName\"\n",
                         argv[0]);
          exit(1);
```

```
        }

        interp = Tcl_CreateInterp();
        code = Tcl_EvalFile(interp, argv[1]);
        if (*interp->result != 0) {
             printf("%s\n", interp->result);
        }
        if (code != TCL_OK) {
             exit(1);
        }
        exit(0);
}
```