TinyCobol: Utilizando GUIs em Tcl/tk

Introdução

O TinyCobol tem uma SCREEN SECTION, que pode ser usada para produzir telas de entrada de dados coloridas e com edição, mas as aplicações modernas requerem um pouco mais. É interessante podermos mostrar *listboxes* e *comboboxes*, imagens, *canvases* onde sejam produzidos desenhos vetoriais, ou outros *widgets*. Infelizmente, o Cobol não tem tais alternativas de uma forma padronizada, daí precisaremos interfacear nosso programa com um *toolkit* que tenha a funcionalidade desejada. Para isso, escolhemos o tcl/tk (na realidade o *toolkit* é só o tk, mas utilizaremos o tcl para adquirirmos um maior e mais natural controle sobre os *widgets* do primeiro) que é disponível como software livre sem restrições, sob uma licença semelhante à das bibliotecas do próprio TinyCobol. Neste breve artigo mostraremos a nossa interface, tctcl, e como obter controle total a partir do programa cobol sobre a operação da nossa GUI (G*raphical User Interface*, ou Interface Gráfica do Usuário em portugues), e como transferir valores entre os dois programas. Para benefício do programador Cobol, tentaremos manter na maioria das vezes o controle no lado do Cobol.

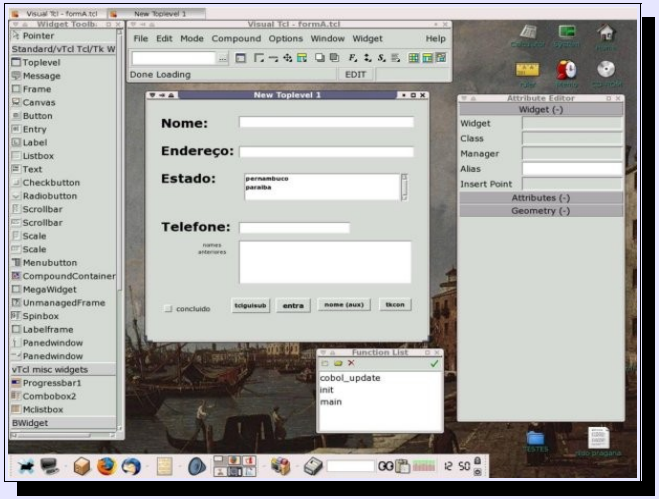
Projetando uma GUI

O primeiro passo para termos o nosso programa com uma interface gráfica é escolhermos como ela será desenhada e implementada. O tk tem comandos bastante simples que permitem a definição dos *widgets* diretamente em um *script* tcl, mas alguns programadores não se sentem muito â vontade em ter que aprender uma linguagem nova, tão somente usada para a definição da interface gráfica. Existem vários *GUI builders* que podem ser empregados para o desenho dessa interface.

Alguns dos *GUI builders que podem ser usados para simplificar o desenho da interface gráfica:*
VisualTcl *– O mais fácil e intuitivo, apesar de ser bastante poderoso. Permite também criar múltiplas janelas com controle independente no mesmo módulo.*
SpecTcl *– Simples e eficiente, também bastante intuitivo. Suporta somente o grid geometry manager*, o que não pode ser considerado essencialmente uma desvantagem
TkproE *– Baseado no antigo XF, tem a virtude de podermos ir projetando e testando a interface simultanemante. Suporta widgets compostos.*
TkBuilder *– Gera o menor código possível, que pode ser editado manualmente, Entretanto, sua organização é hierárquica e não visual como os demais, o que deixa algumas pessoas confusas.*

Independentemente do método usado, um princípio básico que devemos ter em vista sempre é **separar a interface gráfica da lógica do programa**. Se não seguirmos essa regra, corremos o risco de ter um tipo de "espagueti" entre o nosso código (cobol ou tcl, não importa) e a lógica do programa.

Com o **VisualTcl** desenhamos a interface com "clicks" e "*drag-and-drops*" (clicando e arrastando), de maneira bastante intuitiva.



O tctcl

O tcl/tk existe na forma de bibliotecas que devem ser ligadas (na link-edição) à nossa aplicação com GUIs. Igualmente precisaremos do código que faz a interface entre o tcl e o TinyCobol, distribuido na forma de um objeto (acompnhado de todos os fontes, evidentemente, pois trata-se de software livre), **tctcl.o**. Isso se consegue introduzindo na linha de comando da link-edição,

```
gcc -o pgm.o tctcl.o \
   -L/usr/local/lib -ltcl -ltk \
   -lhtcobol -ldb -lncurses -ldl -lm
```

supondo que ambas as bibliotecas do TinyCobol e do tcl/tk `/usr/local/lib`. Se necessário, podemos adicionar outros argumentos `-L<caminho>` que definam a localização de outras bibliotecas.
A API (interface do programador) do tctcl é minúscula, contendo apenas seis chamadas, sendo uma delas para uso por scripts tcl (call_cobol) e as demais pelo programa cobol.

Hello world em tctcl

Começaremos mostrando o clássico *Hello world* escrito em cobol (com o TinyCobol) usando a interface gráfica proporcionada pelo tcl/tk. Nosso *script* sozinho poderia ser usado com a mesma finalidade, mas o programa inteiro nos servirá para mostrarmos os pontos básicos do uso da API. Começaremos pelo *script*:

```
label .lab -text "Hello, world!"
button .b -text ok -command {set ok 1}
grid .lab
grid .b
```

Se você é conhecedor de um mínimo de tcl, notará que este programa já é o nosso "*hello world*", exceto pelo comando atribuido ao botão (que normalmente seria **exit**), escrevendo em uma variável. É que esta variável, **ok**, será uma forma de indicar ao programa cobol que a operação foi concluida, de forma que ele continue sua execução. Vejamos agora como podemos escrever o programa cobol para executar esse *script*.

A API (Application Program Interface) do tctcl

initTcl	Deve ser usada antes de qualquer outra chamada.
tcleval	Executa um script de forma simplificada, "semi-automática".
stcleval	Executa comando(s) tcl contidos em um buffer. Permite maior "controle fino" sobre os scripts a partir do programa cobol.
newGui	Permite usar outra GUI na forma simplificada.
endTcl	Finaliza o interpretador tcl. Útil quando queremos recomeçar com outros *scripts* carregados.
call_cobol	Permite ao *script* tcl/tk chamar uma sub-rotina do cobol *(callback)*.

Usando o tcltcl em um programa Cobol

Antes de qualquer outra chamada, o programa cobol deverá chamar **initTcl**, que irá inicializar o tctcl e carregar um interpretador tcl para processar nossos scripts. Nosso exemplo terá apenas mais uma função, o **stcleval**, que interpreta comandos arbitrários no interpretador tcl criado anteriormente, e retorna um resultado textual numa variável cobol. A *string* numa variável, com os comandos a serem executados pelo tcl, deverá ser terminada por um caracter nulo, como na linguagem C. Conseguimos isso através de um comando STRING onde o último argumento é uma variável definida como

```
77 EOSTR pic X value LOW-VALUES.
```

Essencialmente, queremos executar dois comandos no interpretador tcl, o primeiro para carregar o nosso script, hello.tcl, e o segundo para manter o script sendo executado até que a variável **ok**, seja modificada, o que acontecera quando o usuário pressionar o botão (vide fonte do script na página anterior).

- O primeiro comando será **`source hello.tcl`**, como esperado. Isso irá carregar no interpretador tcl a definição da nossa GUI, como descrito na página anterior.
- O segundo será mais sutil., pois ele deverá manter a GUI tcl/tk ativa enquanto o usuário ainda não tiver terminado (pressionando o botão no nosso exemplo) a sua interação com a GUI. Acontece que o tk tem exatamente um comando para isso. Como queremos aguardar a modificação da variável **ok**, sua forma será a seguinte: **`tkwait variable ok`**.

Resumindo, agora precisamos apenas juntar essas peças e escrever definitivamente nosso programa cobol. No programa, a variável TSCR contém o script (comandos) a serem interpretados, e TRES contém o resultado dessa interpretação.

```
            IDENTIFICATION DIVISION.
            PROGRAM-ID.    hello.
            AUTHOR. Rildo Pragana.
            ENVIRONMENT DIVISION.
            DATA DIVISION.
            WORKING-STORAGE SECTION.
            77      TSCR        PIC X(512).
            77      TRES        PIC X(80).
            77      EOSTR   pic X value LOW-VALUES.

            PROCEDURE DIVISION.
            CALL "initTcl"
* source (load) our script into the tcl interpreter
            string "source hello.tcl" EOSTR into TSCR
            call "stcleval" using TSCR TRES.
* make it stay with control till
* the variable "ok" is modified
            string "tkwait variable ok" EOSTR into TSCR
            call "stcleval" using TSCR TRES.
            STOP RUN.
```

Como dissemos anteriormente, qualquer comando pode ser enviado ao tcl, inclusive múltiplos comandos numa só chamada de **stcleval**, desde que separados por ";" (ponto-e-vírgula), como o tcl requer. Não há limite no tamanho do comando enviado, mas ele precisa sempre ser terminado por um nulo (variável EOSTR no programa). A resposta porém é padronizada no tamanho máximo de 80 caracteres, devido a limitações no cobol. Muitos comandos em tcl não produzem nenhuma resposta, mas mesmo assim, precisamos fornecer um *buffer* de 80 caracteres para que a resposta seja armazenada.

Programa mostrando a mensagem *Hello, wolrd!*

Escrevendo e lendo variáveis

O mesmo comando **stcleval**, serve igaulmente para transferir conteúdo de variáveis entre o programa cobol e *scripts.* Para modificar o valor de uma variável no *script*, usamos o comando **set** do tcl. Surpreendentemente, o mesmo comando **set** tambémservepara obter o valor de uma variável do tcl. É porque este comando sempre retorna o valor (atribuido ou não) da variável que é dada como argumento, modificando-a apenas quando mais um argumento é fornecido. Assim, **`set msg "novo valor"`** irá modificar a variável msg, mas **`set msg`** irá apenas retornar o seu valor atual.

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID.    count.
      AUTHOR. Rildo Pragana.
      ENVIRONMENT DIVISION.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      77      TSCR        PIC X(512).
      77      TRES        PIC X(80).
      77      EOSTR   pic X value LOW-VALUES.
      77      CNT pic 99 value zeros.

      PROCEDURE DIVISION.
      CALL "initTcl"
* source (load) our script into the tcl interpreter
      string "source count.tcl" EOSTR into TSCR
      call "stcleval" using TSCR TRES.
    10-loop.
      add 1 to CNT
* modify variable "msg" value
      string 'set msg "valor = ' CNT '"' EOSTR into TSCR
      call "stcleval" using TSCR TRES.
* make it stay with control till the variable "ok" is modified
      string "tkwait variable ok ; set ok" EOSTR into TSCR
      call "stcleval" using TSCR TRES.
      if TRES = 1
          go to 10-loop.
      STOP RUN.
```

A variável CNT é usada para compor a mensagem que será mostrada em um *widget*, através do conteúdo da variável tcl **msg**. Assim, no início de cada **10-loop**, enviamos para o tcl um comando na forma **`set msg "valor = 01"`**. Observe que precisamos das aspas para agrupar as palavras "valor", o símbolo "=" e o próprio valor numérico. Neste exemplo, teremos nossa GUI com dosi botões, ambos executando código que modifica o valor da variável **ok.** Para distinguir qual botão foi pressionado pelo usuário, usamos dois valores diferentes e verificamos qual valor foi retornado. Daí a linha contendo `tkwait variable ok ; set ok`. Na realidade são dois comandos separados por um ";" (ponto-e-vírgula). O segundo fará o interpretador retornar o valor da variável **ok** na variável TRES do cobol.

```
label .lab -textvariable msg
button .b -text ok -command {set ok 1}
button .exit -text fim -command {set ok 2}
grid .lab -
grid .b .exit
```

Poderíamos, se não quiséssemos usar a variável **msg**, modificar diretamnte a opção **-text** do label, substituindo o comando

```
set msg "valor = ..."
```

por um comando de configuração direto:

```
.lab configure -text "valor = ..."
```

Neste exemplo, a cada vez que o botão"ok" é pressionado, o cobol receb o controle de volta, incrementa o contador e redefine a mensagem. O botão "fim" retorna com ok=2 e faz o programa cobol executar "stop run".

Listas e listboxes

Muitas aplicações precisam de *widgets* na forma de listas com múltiplas escolhas. Vejamos uma forma de como introduzir uma série de valores numa variável tipo "lista" do tcl. Esse mesmo procedimento se aplica a outros *widgets* como *comboboxes, trees,* etc.
Podemos modificar uma lista como uma variável qualquer, se soubermos de antemão todos os seus elementos, como no seguinte exemplo, uma lista de cores:

```
set lista {vermelho laranja amarelo azul
     violeta {verde azulado} cinza preto}
```

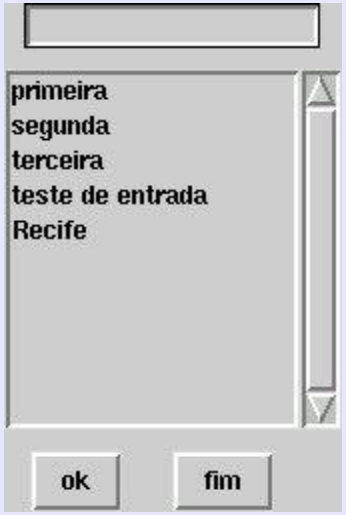
Observe que o elemento "verde azulado" figura entre chaves. É que pretendemos que ele seja um único elemento na lista e não duas palavras separadas.
Para adicionar elementos a uma lista já existente (ou criá-la se ela ainda não existir), podemos usar o comando **lappend**, ou o comando **linsert**. O primeiro adiciona o novo elemento (ou novos se mais de um) no final da lista, enquanto que o segundo permite a adição se processar em qualquer posição.
Primeiramente vamos ao nosso *script*. Nele definimos como novidade uma *listbox,* e adicionamos igualmente uma *scrollbar* que nos permita visualizar todos os elementos da lista, quando este crescer mais que a altura (número de linhas) do *listbox.* Para sincronizar os dois *widgets* (a listbox .lbox com a scrollbar .sv), a "receita" é simples, como vemos no script, consistindo no comando **set** para posicionar a scrollbar, e o comando **yview** para fazer o mesmo com a listbox (consulte um livro sobre tcl para entender detalhadamente esse assunto).
Nosso *script* tem também um *binding*, **bind .e <Return> {set ok 1}**, para deixar mais conveniente o preenchimento por parte do usuário. No lugar de ter que pressionar o botão **ok** para cada "entrada", o usuário simplesmente finaliza sua entrada com a tecla **<Return>** (ou tecla <Enter>, se usarmos a nomenclatura "Microsoftiana").

```
entry .e -textvariable entrada
listbox .lbox -listvariable \
     lista -yscrollcommand {.sv set}
scrollbar .sv -command {.lbox yview}
button .b -text ok -command {set ok 1}
button .exit -text fim -command {set ok 2}
grid .e - -
grid .lbox - .sv -sticky nsew -pady 10
grid .b .exit
bind .e <Return> {set ok 1}
```

Nota: Eliminando excesso de espaços nas variáveis cobol

Como sabemos, as variáveis ou campos do cobol têm tamanho previamente declarados, mas em alguns casos queremos usar somente os caracteres significativos (não espaços) no *script* tcl. O próprio tcl nos fornece uma solução, através dos comandos **`string trim`** `(elimina espácos antes e depois da string)`, **string trimleft** (elimina espaços à esquerda) e **string trimright**(elimina espaços à direita, ou seja, no final da string). No exemplo que veremos a seguir, utilizaremos estes comandos para evitar que a resposta obtida do tcl para o cobol seja interpretada como uma linha esparsa (com poucos caracteres no início, com tamanho fixo de 80 caracteres!

O usuário introduz linhas na caixa de entrada (entry) na parte superior e elas são adicionadas ao fim da lista, que é reposicionada automaticamente.



Listas e listboxes: o lado cobol

Vejamos agora como fica o lado cobol do nosso programa.

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID.    listas.
        AUTHOR. Rildo Pragana.
        ENVIRONMENT DIVISION.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        77      TSCR        PIC X(512).
        77      TRES        PIC X(80).
        77      EOSTR   pic X value LOW-VALUES.

        PROCEDURE DIVISION.
        CALL "initTcl"
* source (load) our script into the tcl interpreter
        string "source listas.tcl" EOSTR into TSCR
        call "stcleval" using TSCR TRES.
* inicia com alguns elementos na lista
        10-loop.
* clear our entry widget variable
        string "set entrada {}" EOSTR into TSCR
        call "stcleval" using TSCR TRES
* wait until the variable "ok" is modified
        string "tkwait variable ok ; set ok"
             EOSTR into TSCR
        call "stcleval" using TSCR TRES.
* append the input (entry) to the listbox's list variable
        if TRES = 1
            string "set entrada" EOSTR into TSCR
            call "stcleval" using TSCR TRES
            string "lappend lista [string trim {"
            TRES "}] ; .lbox yview end" EOSTR into TSCR
            call "stcleval" using TSCR TRES
            go to 10-loop.
        STOP RUN.
```

Como vemos, é semelhante ao exmplo anterior, exceto por executar vários outros comandos a cada vez que o *script* retorna com a variável **ok=1** (ou seja, não é para finalizar ainda...). Primeiramente, precisamos obter o conteúdo da variável **entrada** do tcl, que está sincronizada com o conteúdo da caixa de entrada, *entry* **.e**. Depois queremos adicionar esse conteúdo à nossa lista (chamada **lista**). O problema é que recuperamos essa variável na nossa TRES do lado cobol, que é definida com **pic X(80)**, ou seja, teremos o conteúdo real formatado em um campo de oitenta posições, mesmo que a variável contivesse no tcl somente 2 ou 3 caracteres! Isso evidentemente é uma deficiência da linguagem cobol (que foi criada há cerca de 45 anos!). Para contornar esse problema, executamos um outro comando no tcl para "filtrar" essa variável, antes de fazer a atribuição final desejada. No tcl, podemos executar um comando "embutido" dentro de outro, se colocado entre colchetes **[...]**. Assim, podemos executar **lappend lista [string trim {...}]**, para obter o resultado. Como o "miolo" desta expressão está numa variável cobol, aproveitamos o comando STRING para concatenar cada uma destas partes, substituindo os pontinhos **"..."** pela variável TRES (que é o resultado do comando anterior, onde obtivemos a entrada do usuário).
Outro comando que aí aparece é o **.lbox yview end**, destinado a fazer o listbox mostrar o final da sua lista (que poderia fica encoberta caso ultrapassasse a altura do *widget*).

Complemento: outras manipulações com listas

O tcl suporta inúmeros outros comandos com listas. Por exemplo, poderíamos obter o ítem selecionado (o primeiro, se múltiplos estiverem habilitados) pelo usuário no listbox, com o comando:

```
string ".lbox get [lindex [.lbox curselection] 0]"
            EOSTR into TSCR
call "stcleval" using TSCR TRES.
```

O comando **lindex** certifica que somente o primeiro ítem da lista retornada por **.lbox curselection** será considerado.
Podemos eliminar um determinado ítem através do comando **lreplace**, repetindo duas vezes o seu número. Por exemplo, se quisermos eliminar o terceiro ítem (numerados 0,1,2...) usaremos o comando tcl **set lista [lreplace $lista 3 3]**. Em geral, o comando **lreplace** troca elementos por outros fornecidos, mas se presta também para eliminar ítens, caso nenhum *replacement* (substitutos) for fornecido.
Listas são muito importantes no tcl (assim como nas linguagens lisp, scheme, prolog, e nas linguagens funcionais como haskell, ml, etc). Com listas podemos implementar estruturas de dados complexas como árvores, grafos, etc, mas o seu tratamento completo foge ao nosso escopo aqui.

Controlando a visibilidade de janelas

Numa aplicação real, muitas janelas serão mostradas e deverão desaparecer (ficar invisíveis, ou "não-mapeadas") quando não estiverem sendo utilizadas. Estas janelas também são conhecidas como "diálogos". Para evitar retardos na redefinição destas janelas temporárias, o ideal é que deixemos previamente definidas todas as janelas que a nossa aplicação requer e ir controlando quando uma determinada janela deverá aparecer ou tornar-se invisível. Para isso, podemos usar dois comandos do tcl/tk, **wm withdraw .janela** e **wm deiconify .janela**. O primeiro faz a janela desaparecer, o segundo torna esta janela visível novamente. Mostraremos um programa um pouco mais complexo onde duas janelas são empregadas, sendo a segunda (um **toplevel**, na linguagem do tk) um diálogo invocado indiretamente quando um determinado botão é pressionado (botão rotulado "?"). Vamos ao *script*:

```
#---------main-----------------
entry .e -textvariable entrada
button .qry -text ? -command {set ok 3}
button .b -text ok -command {set ok 1}
button .exit -text fim -command {set ok 2}
grid .e .qry -pady 10
grid .b .exit
bind .e <Return> {set ok 1}
#---------dialog--------------
toplevel .dlg
set lista {vermelho laranja amarelo {amarelo ouro} azul
    anil violeta {verde azulado} marron cinza branco}
listbox .dlg.lbox -listvariable lista \
      -yscrollcommand {.dlg.sv set} -height 6
scrollbar .dlg.sv -command {.dlg.lbox yview}
button .dlg.cancel -text cancela -command {set ok 2}
grid .dlg.lbox - .dlg.sv -sticky nsew -pady 10
grid x .dlg.cancel x
bind .dlg.lbox <1> {set ok 1}
```

Temos no *script* dois toplevels, o primeiro é o default **"."**, criado pelo próprio tk, o segundo **.dlg**, sera nosso diálogo. Criamos um *binding* para retornar ao programa cobol quando um ítem da listbox for selecionado, de forma que o simples clique num dos ítens dará como resultado o preenchimento da caixa de entrada do programa principal com o ítem selecionado. Para simplifcar, todos os ítens da listbox são previamente definidos, mas não seria difícil criá-los dinamicamente, como vimos no exemplo anterior.

O botão **.qry** retorna um valor diferente dos outros dois, de forma que o programa cobol podera testar esse valor e decidir mostrar o diálogo.
No diálogo, se o botão **cancel** for pressionado, nada sera copiado para a caixa de entrada, que permanecerá como estava. Evidentemente, o usuário podera preencher também essa entrada manualmente, caso o que ele deseje não se encontre na nossa *listbox*.

Veja na figura um *screen shot* com as duas janelas em ação.

Aplicação com múltiplas janelas, com a visibilidade de cada uma contralada livremente pelo programa cobol.

Multiplas janelas controladas pelo cobol

Entendido o *script*, torna-se fácil escrever o programa cobol (abaixo).
No parágrafo **30-show-input**, simplesmente mostramos no terminal o que foi introduzido pelo usuario, quando o botão **ok** foi pressionado, seja por uma edição manual ou através do diálogo, pois ambos alteram a variável **entrada**.
No parágrafo **40-dialog**, o diálogo é tornado visível, como já discutimos, e ao retornar o seu preenchimento (via variável **ok**) copiamos ou não o ítem selecionado para a variável **entrada**.

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID.    visibilidade.
        AUTHOR. Rildo Pragana.
        ENVIRONMENT DIVISION.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        77      TSCR        PIC X(512).
        77      TRES        PIC X(80).
        77      EOSTR   pic X value LOW-VALUES.

        PROCEDURE DIVISION.
        CALL "initTcl"
* source (load) our script into the tcl interpreter
        string "source visibilidade.tcl" EOSTR into TSCR
        call "stcleval" using TSCR TRES.
        10-loop.
* wait till the variable "ok" is modified
        string "tkwait variable ok ; set ok" EOSTR into TSCR
        call "stcleval" using TSCR TRES.
* check the result
        if TRES = 1
            perform 30-show-input.
        if TRES = 2
            go 99-end.
        if TRES = 3
            perform 40-dialog.
        go 10-loop.
```

Finalmente, no parágrafo **50-fill-entrada** é onde fazemos a cópia.

Seria aconselhável usar uma variável **ok** separada para cada diálogo. Para não poluir o *namespace* do tcl, é sábio valer-se do uso de *arrays associativo*, disponibilizados pelo tcl, mas não aprofundaremos essa discussão aqui.

```
* show the input (entrada) got from user
        30-show-input.
        string "set entrada"
            EOSTR into TSCR
        call "stcleval" using TSCR TRES.
        display "Entrada: " TRES.

* show our dialog
       40-dialog.
       string "wm deiconify .dlg" EOSTR into TSCR
       call "stcleval" using TSCR TRES.
       string "tkwait variable ok ; set ok" EOSTR into TSCR
       call "stcleval" using TSCR TRES.
       if TRES = 1
           perform 50-fill-entrada.
* hide the dialog again
        string "wm withdraw .dlg" EOSTR into TSCR
        call "stcleval" using TSCR TRES.

* get the selected item from the listbox
* and copy to the entry
        50-fill-entrada.
        string
       ".dlg.lbox get [lindex [.dlg.lbox curselection] 0]"
            EOSTR into TSCR
        call "stcleval" using TSCR TRES.
        string "set entrada [string trimright {" TRES "}]"
            EOSTR into TSCR
        call "stcleval" using TSCR TRES.

        99-end.
           stop run.
```


Usando *callbacks* para o programa cobol

Da mesma forma que o programa cobol pode executar comandos do tcl, através do **CALL stcleval**, um programa tcl pode executar uma subrotina escrita em cobol pelo comando **call_cobol**. Há algumas diferenças contudo, nos argumentos que são enviados.

```
call_cobol rotina_cobol string
```

A função tcl **call_cobol** espera dois argumentos: o primeiro é o nome da rotina cobol a ser chamada, que deverá estar em uma biblioteca dinâmica, e não em qualquer local, pois o *loader* precisa encontrá-la pelo nome (dynamic binding); o segundo é uma string de até 80 caracteres, que deverá ser definido no lado do cobol como **pic X(80)**. Com o uso judicioso de verbos como INSPECT e UNSTRING, é possível quebrar esse único argumento em vários, como mostraremos em um exemplo a seguir.
Outra peculiaridade é que não há forma do cobol retornar um resultado, mas isso não se torna um problema porque, do programa cobol podemos chamar **stcleval**, como usamos até aqui, para fornecer os resultados desejados ao *script* tcl.
Vamos iniciar com um exemplo em cobol puro. Analise o programa abaixo. Ele produz a seguinte saida:

```
STRING ORIGINAL:
"Este programademonstrao usode múltiplas
stringsconcatenadasnuma só.        "
Num. campos: 006
01- Este programa
02- demonstra
03- o uso
04- de múltiplas strings
05- concatenadas
06- numa só.
```

O "segredo" está no UNSTRING, que decompõe o campo original delimitado por um caracter "especial" invisível **X"01"** (nulos, X"00" ou LOW-VALUES daria problemas com as rotinas do tctcl) em vários outros campos, possivelmente menores. Usando a claúsula TALLYING podemos saber quantas *sub-strings* foram contadas e assim efetivamente receber vários parâmetros em um só argumento, ou seja, no original INPBUF.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    unstring1.
       AUTHOR. Rildo Pragana.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77      EOSTR   pic X value LOW-VALUES.
       77      STRSEP  pic X value X"01".
       77      WKEY    pic x.
       77      INPBUF  pic X(80).
       77      WFTALLY pic 9(3).
       77      WFN     pic 9(2).
       01  WBUF.
        05 WF     occurs 8 times     pic X(20).

       PROCEDURE DIVISION.
       string "Este programa" STRSEP "demonstra" STRSEP
           "o uso" STRSEP "de múltiplas strings" STRSEP
           "concatenadas" STRSEP "numa só." into INPBUF
       display "STRING ORIGINAL:"
       display '"' INPBUF '"'
       move zeros to WFTALLY
       unstring INPBUF
           delimited by STRSEP
           into WF(1) WF(2) WF(3) WF(4)
                WF(5) WF(6) WF(7) WF(8)
           tallying in WFTALLY.
```

```
       display "Num. campos: " WFTALLY
       move 1 to WFN.
       10-next.
       display WFN "- " WF(WFN)
       if WFN < WFTALLY
           add 1 to WFN
           go to 10-next.
       accept wkey.
       99-end.
          stop run.
```

Editando uma *entry* em cobol

O *entry widget* do tk tem algumas opções de configuração que nos possibilita controlar o conteúdo que será aceito, tecla a tecla. Essa operação é denominada **validação** (que não deve ser confundida com a validação usual no linguajar das aplicações comerciais). A primeira opção é **-validate**, com valor *default* **none** (nenhuma), mas que pode assumir diversos valores, entre os quais **key**, que nos interessa. Isso faz com que cada tecla inserida ou removida receba nossa atenção. O comando de validação é definido pela opção **-validatecommand** ou **-vcmd**, que funciona como um **bind**, com várias substituições (tabela abaixo).
Para evitar a reentrada na função de validação, a opção **-validate** é desabilitada quando ocorre modificação da *entry* durante a execução do comando associado a **-vcmd** e precisa ser reinstalada na saida, mas somente no final. Daí usamos o comando

after idle {%W configure -validade %v},

que irá reinstalar na janela (%W=.e) a opção (%v=key).
Para deixar o controle inteiramente com o *callback*, fazemos com que o **-vcmd** retorne sempre zero, o que significa que a tecla processada não foi aceita. Isso faz com que a edição seja apenas comandada pelo lado cobol (cback) e não por funções do próprio *widget*.

Substituições no -validatecommand

%d	tipo de ação, insert (1) ou delete (0)
%i	índice no conteúdo da *entry*
%P	o valor que ficaria a *entry* se o caracter for introduzido na posiçso corrente
%s	conteúdo da *entry* sem considerar a tecla atual
%S	caracteres que serão inseridos/removidos
%v	tipo de validação atual (none, key, focus,...)
%V	tipo de validação que gerou o evento
%W	nome do *widget*

A **proc do_cback** é a reponsável por chamar o *callback*, passando para ele todos os argumentos concatenados como um só, adicionados de separadores **X"01"**. Isso é conseguido pelo comando **join $args "\x1"**.
Devido à forma como o **cback** é definido (em cobol), o conteúdo da *entry* deverá sempre estar válido, daí no *script* já colocamos um valor inicial, através do comando **set e {00/00/00}.**

```
label .lb -text Data:
entry .e -textvariable e -width 8 -validate key -vcmd {
        do_cback %P %d %i %s %S %W
        after idle {%W config -validate %v}
        return 0
    }
button .b -text ok -command {set ok 1}
button .exit -text fim -command {set ok 2}
grid .lb .e - - -pady 5
grid  .b .exit x -pady 5
bind .e <Return> {set ok 1}

proc do_cback {args} {
    global e
    set arg [join $args "\x1"]
    call_cobol cback $arg
}
set e {00/00/00}
focus .e
```

Uma entrada de dados inteiramente processada pelo coo, usando o mecanismo de *callback* a cada tecla pressionada.

Entrada de dados editada

Nosso programa consiste em três módulos, o programa principal, **cback_main.cob**, o *script* **cback.tcl**, e no que processa cada tecla, **cback.cob**, este último compilado como uma biblioteca dinâmica, para possibilitar sua

```
.SUFFIXES: .cob .cbl .o
CC=gcc -g

.o:
    gcc -g -o $@ $< tctcl.o -L/usr/local/lib \
      -ltcl -ltk -lhtcobol -ldb -lncurses -ldl -lm

.cob.o:
    htcobol -c -v -g -e $* -D $<

cback_main: cback_main.o cback.so tctcl.o
    gcc -g -o $@ $< tctcl.o -L. -lcback \
      -L/usr/local/lib -ltcl -ltk \
      -lhtcobol -ldb -lncurses -ldl -lm

cback.so: cback.cob
    htcobol -c -v -g -D $<
    gcc -shared -Wl,-soname,libcback.so \
      -o libcback.so cback.o
```

chamada pela **proc call_cobol**. A **Makefile** acima mostra como os compilamos.
O programa principal é bastante simples, semelhante a muitos que já mostramos anteriormente. Ao detectar o botão **ok** pressionado (TRES=1), ele solicita ao interpretador tcl o valor da variável **e**, que esta associada à *entry*, obtendo assim o valor do campo editado.
O módulo **cback.cob**, responsável pelo aceite e edição de cada tecla é o mais elaborado. No início ele fará a separação do argumento recebido (listagem na próxima página) nas variáveis **EPREV, ECURR, ..., EWIN**, correspondendo aos valores passados pelo tcl, e, examinando o conteúdo do caracter adicionado, **ECHAR**, deverá verificar se ele é um dígito numérico válido. Para isso, usará funções **ORD** do cobol,comparando com os caracteres '0' e '9'. A variável **EINDEX** indica a posição do caracter em questão, mas começando do índice zero. Quando um caracter é removido, a condição **88-EINS-FALSE** se aplica, e nesse caso o usuário deseja remover um caracter. Para "redesenhar" o campo na *entry,* enviamos ao tcl a seguinte sequência: **.e delete 0 end ; .e insert end <novo_conteúdo> ; .e icursor <nova_posição_do_cursor>,** onde **.e** é recebdio em **EWIN**.

Certamente este módulo **cback.cob** não é um pérola de projeto. Você caro leitor é convidado a realizar essa tarefa melhor! Meu objetivo era tão somente fornecer um exemplo de como podemos fazer toda a edição para a entrada de dados no próprio cobol.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    cback_main.
       AUTHOR. Rildo Pragana.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77      TSCR        PIC X(512).
       77      TRES        PIC X(80).
       77      EOSTR   pic X value LOW-VALUES.

       PROCEDURE DIVISION.
       CALL "initTcl"
       string "source cback.tcl" EOSTR into TSCR
       call "stcleval" using TSCR TRES.

       10-loop.
       string "tkwait variable ok ; set ok" EOSTR into TSCR
       call "stcleval" using TSCR TRES
       if TRES = 1
           string "set e" EOSTR into TSCR
           call "stcleval" using TSCR TRES
           display "INPUT: '" TRES "'"
           go to 10-loop.

       99-end.
          stop run.
```

O *callback* em cobol

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    cback.
       AUTHOR. Rildo Pragana.
       ENVIRONMENT DIVISION.
       DATA DIVISION.

       WORKING-STORAGE SECTION.
       77      TSCR        PIC X(512).
       77      TRES        PIC X(80).
       77      EOSTR   pic X value LOW-VALUES.
       77      STRSEP   pic X value X"01".
       77      WKEY    pic x.
       77      WTALLY pic 9(3).
       01  EBUF.
        05 EPREV           pic X(20).
        05 ECURR           pic X(20).
        05 ECHAR           pic X(1).
        05 EINS-FLAG       pic X(1).
           88 EINS-TRUE    value 1.
           88 EINS-FALSE   value 1.
        05 EINDEX          pic 9(2).
        05 EWIN            pic X(20).
       01  DATAFMT     pic 99/99/99.
       01  DATAFMT1    redefines DATAFMT pic X(08).
       01  DATAAUX1    redefines DATAFMT.
        05 DATAAUX1-DIG    occurs 8 pic 9.
       77  INDEX1          pic 9(2).
       77  NDIGIT          pic X(1) value "0".
       77  NMIN        pic X(1) value "0".
       77  NMAX        pic X(1) value "9".

       LINKAGE SECTION.
       01 INPBUF pic X(80).

       PROCEDURE DIVISION using INPBUF.
       move zeros to WTALLY
```

- do_cback %P %d %i %s %S %W
- (veja "man n entry" section VALIDATION)

```
       unstring INPBUF
           delimited by STRSEP
           into ECURR EINS-FLAG EINDEX EPREV ECHAR EWIN
           tallying in WTALLY.
       add 1 to EINDEX GIVING INDEX1
       move EPREV to datafmt1
       if EINS-TRUE
           go 20-INSERT.

       10-REMOVE.
       move dataaux1-dig ( index1 ) to ndigit
       if  not ( function ord ( nmin ) <=
                 function ord ( ndigit ) and
                 function ord ( nmax ) >=
                 function ord ( ndigit ) )
                   subtract 1 from index1.
       move "0" to DATAAUX1-DIG ( index1 ).
       subtract 1 from index1
       perform 30-refresh
       go 99-exit
       .
       20-INSERT.
       move echar to ndigit
       if  function ord ( nmin ) <=
           function ord ( ndigit ) and
           function ord ( nmax ) >=
           function ord ( ndigit )
               perform 25-place-digit
       else
           display 'ERROR: "' ndigit '" is NOT a digit.'.
       .
      25-place-digit.
      move dataaux1-dig ( index1 ) to ndigit
       if  not ( function ord ( nmin ) <=
                 function ord ( ndigit ) and
                 function ord ( nmax ) >=
                 function ord ( ndigit ) )
            add 1 to index1.

       move ECHAR to DATAAUX1-DIG ( INDEX1 ).
       perform 30-refresh
       .
       30-refresh.
       string  EWIN "delete 0 end;"
               EWIN "insert end " DATAFMT ";"
               EWIN "icursor " INDEX1 EOSTR into TSCR
       call "stcleval" using TSCR TRES
       .
       99-exit.
       exit program.
```