Probeklausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Bei Ankreuzaufgaben können mehrere Antworten richtig sein. Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 120 Minuten.

Aufgabe 1:

Es geht ganz einfach los. Betrachten Sie das folgende Widget und die Eingabe in die **`wish`** unten.


```
wegner@holle:~/tcl> wish
% label .frage -text "Klausur schreiben?"
.frage
% pack .frage -padx 4 -pady 4
% button .yes -text "Ja" -command {puts "Dann los!"
exit}
.yes
% pack .yes -padx 4 -pady 4
% button .no -text "No" -command {puts "Schade - tschüss"
exit}
.no
% pack .no -padx 4 -pady 4

% .no _______________ _______________ "Nein"
%
```

(a) Ergänzen Sie die beiden fehlenden Stellen in der Zeile mit **`.no`** am Anfang.

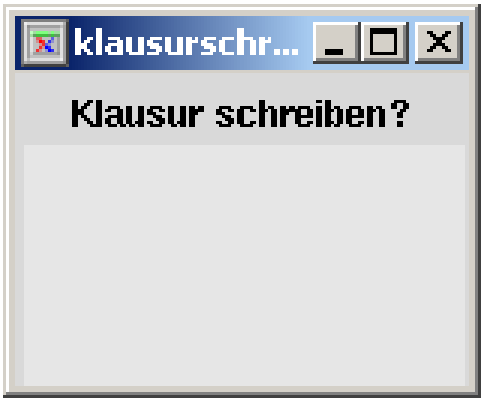
(b) Was passiert, wenn man den **`Ja`**-Knopf drückt?

Aufgabe 2:

Das unten gezeigte Skript (analog zu Frage 1) liefert eine sehr gewöhnungsbedürftige Anordnung der Knöpfe.

```
label .frage -text "Klausur schreiben?"
pack .frage -padx 4 -pady 4
button .yes -text "Ja" -command {
  puts "Dann los!"
  exit}
pack .yes -side left -padx 4 -pady 4
button .no -text "Nein" -command {
  puts "Schade - tschüss"
  exit}
pack .no -padx 4 -pady 4
button .weissnicht -text "Weiß nicht" -command exit
pack .weissnicht -padx 4 -pady 4
```

Zeichen Sie die Knöpfe in die Ausgabe unten ein!



Aufgabe 3:

Wir erzeugen eine Datei **`hallodatei`**, in die wir den Text „Hello world!“ schreiben. Dazu genügt das Skript unten. Das open-Kommando, hier mit der Option **`"w"`** für schreibenden Zugriff, liefert dabei einen Filedeskriptor, den man zum Schreiben, Lesen und Schließen der Datei braucht.

```
set fd [open hallodatei "w"]
puts $fd "Hello world!"
close $fd
exit
```

(a) Wir verfeinern jetzt das Skript um ein Entry-Widget, in dem man den Dateinamen eingeben kann, damit nicht immer in **`hallodatei`** geschrieben wird. Ergänzen Sie die fehlende Stelle!

```
frame .r
pack .r
label .r.prompt -text "Bitte Dateinamen eingeben:"
pack .r.prompt
entry .r.ein -textvariable name -background white -width 40
pack .r.ein -padx 4 -pady 4
button .r.ok -text "OK" -command {

   set fd [_____________________________________________]
   puts $fd "Hello world!"
   close $fd
   exit}
pack .r.ok -padx 4 -pady 4
```

(b) Wir möchten nach dem Öffnen des Fensters den Fokus sofort auf die Eingabe (entry) setzen. Wie lautet die Anweisung dazu?

_____

(c) Das Kommando (Skript), das oben von dem OK-Knopf gestartet wird, soll in eine Prozedur **`writenow {whereto what}`** gesteckt werden, wobei **`whereto`** der zu übergebende Dateiname ist und **`what`** der Text, der in die Datei geschrieben wird. Ergänzen Sie die Prozedur unten, wobei **`exit`** nicht in der Prozedur auftauchen soll.

```
proc writenow {whereto what} {




}
```

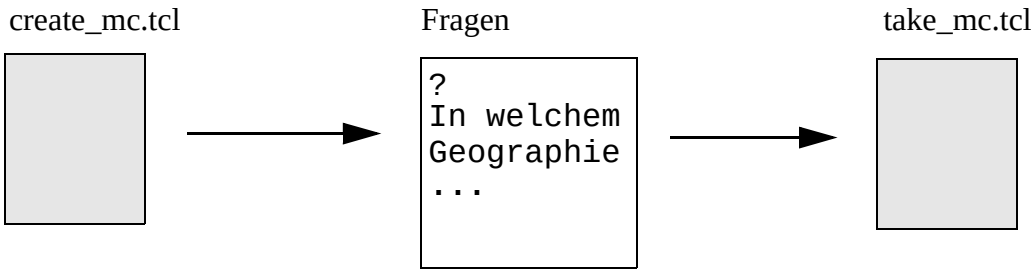
(d) Verwenden Sie **`writenow`** in der Kommando-Option des **`OK`**-Knopfs, wobei wir weiterhin nur **`Hello world!`** schreiben. Vergessen Sie nicht das **`exit`** danach.

```
button .r.ok -text "OK" -command {

    ____________________________________________________________

    ____________________________________________________________

}
```

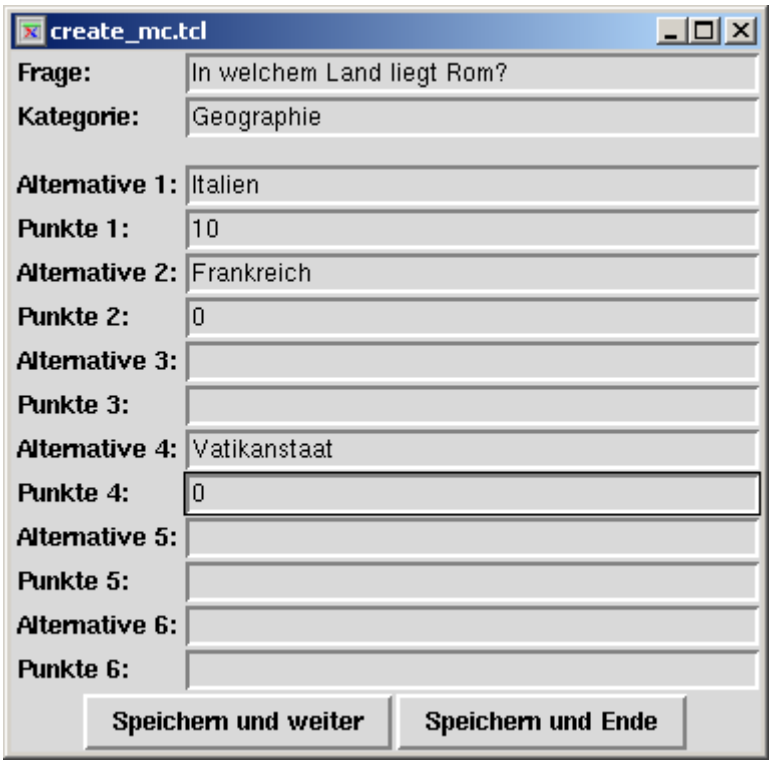
(e) Wir wollen das Schreiben auch schon dann auslösen, wenn die RETURN-Taste im Eingabefeld gedrückt wird. Binden Sie das **`Key-Return`** Ereignis an das Entry-Widget, damit die gleiche Funktion wie beim Drücken des **`OK`**-Knopfs ausgelöst wird.

Aufgabe 4:

Es geht um Multiple-Choice-Tests. Wie unten gezeigt, produzieren (entwerfen) wir mit **`create_mc.tcl`** interaktiv einen Test, der in der Datei **`Fragen`** festgehalten wird. Durch Aufruf eines Skripts **`take_mc.tcl`**, das aus der Datei **`Fragen`** liest, wird für die zu prüfende Person ein interaktiver Test produziert und es werden die Antworten ausgewertet.



Jede Frage gehört zu einer Kategorie und kann bis zu 6 Alternativen als Antwort haben, die alle mit Punkten versehen sind. Nur die nichtleeren Alternativen werden in **`Fragen`** geschrieben. Alle Eingaben beim Entwurf werden zuerst in einem Array zwischengespeichert und dann beim Drücken von **`Speichern und Ende`** geschrieben (siehe Aufgabe 5).



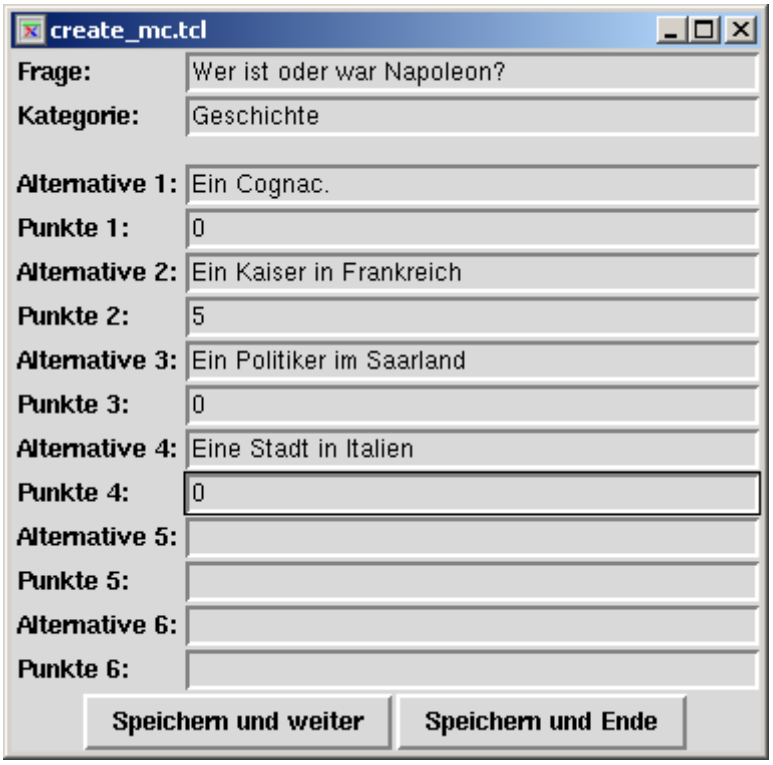
Ergänzen Sie die Lücken in dem Skript unten.

```
proc makeLabelAndEntry { name beschriftung } {
    frame .frame$name
    label .frame$name.label -text _____________________________________
    pack .frame$name.label -side left
    entry .frame$name.entry -textvariable _________________ -width 40
    pack .frame$name.entry -side right
```

```
    return .frame$name
}

foreach {n b} {frage "Frage:" kategorie "Kategorie:"} {
    pack [makeLabelAndEntry $n $b] -fill both
}

frame .frameLuft -height 12
pack .frameLuft


for {set i 1} {$i <= 6} {incr i} {
    pack [makeLabelAndEntry alt($i) "____________________"] -fill both
    pack [makeLabelAndEntry punkte($i) "Punkte $i:"] -fill both
}
```

Aufgabe 5:

Unten sehen Sie eine zweite Eingabemaske und darunter die Datei mit der Ausgabe der beiden produzierten Fragen. Fragen beginnen immer mit einem Fragezeichen, dann kommt der Fragetext, die Kategorie, Alternative 1, deren Punkte, usw.




```
wegner@holle:~/tcl> cat Fragen
?
In welchem Land liegt Rom?
Geographie
Italien
```

```
10
Frankreich
0
Vatikanstaat
0
?
Wer ist oder war Napoleon?
Geschichte
Ein Cognac.
0
Ein Kaiser in Frankreich.
5
Ein Politiker im Saarland.
0
Eine Stadt in Italien.
0
wegner@holle:~/tcl>
```

Ergänzen Sie auch hier die fehlenden Stellen. Geschrieben wird zunächst in den Array **mc**. Beachten Sie, daß auch im Fall von „**Speichern und Ende**“ die letzte Frage noch in diesen Array geschrieben werden muß, dann wird der ganze Array in die Datei übernommen.

```
frame .frameButtons
pack .frameButtons
button .frameButtons.weiter -text "Speichern und weiter" -command {
    incr anzahl
    set mc(F${anzahl}Fragetext) $frage
    set frage ""
    set mc(F${anzahl}Kat) $kategorie
    set kategorie ""
    for {set i 1} {$i <= 6} {incr i} {
        set mc(F${anzahl}Alt${i}Text) _____________________________
        set alt($i) ""
        set mc(F${anzahl}Alt${i}Pkte) _____________________________
        set punkte($i) ""
    }
}
pack .frameButtons.weiter -side left
button .frameButtons.ende -text "Speichern und Ende" -command {

    ____________________________________________invoke
    set fd [open "Fragen" "w"]
    for {set f 1} {$f <= $anzahl} {incr f} {
        puts $fd "?"
        puts $fd $mc(F${f}Fragetext)
        puts $fd $mc(F${f}Kat)
for {set i 1} {$i <= 6} {incr i} {

            if {$mc(_____________________________________) != ""} {
```

```
                puts $fd $mc(F${f}Alt${i}Text)
                puts $fd $mc(F${f}Alt${i}Pkte)
            }
        }
    }
    close $fd
    exit
}
pack _____________________________________________ -side left
set anzahl 0
```

Probeklausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: Vorname:

Musterlösung

Matr.-Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Bei Ankreuzaufgaben können mehrere Antworten richtig sein. Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 120 Minuten.

Aufgabe 1:

Es geht ganz einfach los. Betrachten Sie das folgende Widget und die Eingabe in die **wish** unten.


```
wegner@holle:~/tcl> wish
% label .frage -text "Klausur schreiben?"
.frage
% pack .frage -padx 4 -pady 4
% button .yes -text "Ja" -command {puts "Dann los!"
exit}
.yes
% pack .yes -padx 4 -pady 4
% button .no -text "No" -command {puts "Schade - tschüss"
exit}
.no
% pack .no -padx 4 -pady 4

% .no configure -text "Nein"
%
```

(a) Ergänzen Sie die beiden fehlenden Stellen in der Zeile mit **.no** am Anfang.

(b) Was passiert, wenn man den **Ja**-Knopf drückt?

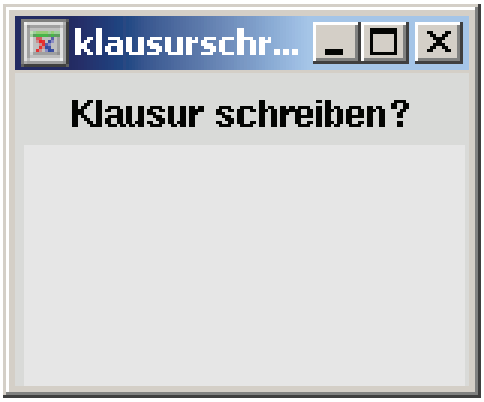
Es erfolgt die Ausgabe „Dann los!“ und das Programm wird beendet.

Aufgabe 2:

Das unten gezeigte Skript (analog zu Frage 1) liefert eine sehr gewöhnungsbedürftige Anordnung der Knöpfe.

```
label .frage -text "Klausur schreiben?"
pack .frage -padx 4 -pady 4
button .yes -text "Ja" -command {
  puts "Dann los!"
  exit}
pack .yes -side left -padx 4 -pady 4
button .no -text "Nein" -command {
  puts "Schade - tschüss"
  exit}
pack .no -padx 4 -pady 4
button .weissnicht -text "Weiß nicht" -command exit
pack .weissnicht -padx 4 -pady 4
```

Zeichen Sie die Knöpfe in die Ausgabe unten ein!



Aufgabe 3:

Wir erzeugen eine Datei **hallodatei**, in die wir den Text „Hello world!“ schreiben. Dazu genügt das Skript unten. Das open-Kommando, hier mit der Option **"w"** für schreibenden Zugriff, liefert dabei einen Filedeskriptor, den man zum Schreiben, Lesen und Schließen der Datei braucht.

```
set fd [open hallodatei "w"]
puts $fd "Hello world!"
close $fd
exit
```

(a) Wir verfeinern jetzt das Skript um ein Entry-Widget, in dem man den Dateinamen eingeben kann, damit nicht immer in **hallodatei** geschrieben wird. Ergänzen Sie die fehlende Stelle!

```
frame .r
pack .r
label .r.prompt -text "Bitte Dateinamen eingeben:"
pack .r.prompt
entry .r.ein -textvariable name -background white -width 40
pack .r.ein -padx 4 -pady 4
button .r.ok -text "OK" -command {

   set fd [____ open $name "w" __]
   puts $fd "Hello world!"
   close $fd
   exit}
pack .r.ok -padx 4 -pady 4
```

(b) Wir möchten nach dem Öffnen des Fensters den Fokus sofort auf die Eingabe (entry) setzen. Wie lautet die Anweisung dazu?

focus .r.ein

(c) Das Kommando (Skript), das oben von dem OK-Knopf gestartet wird, soll in eine Prozedur **`writenow {whereto what}`** gesteckt werden, wobei **`whereto`** der zu übergebende Dateiname ist und **`what`** der Text, der in die Datei geschrieben wird. Ergänzen Sie die Prozedur unten, wobei **`exit`** nicht in der Prozedur auftauchen soll.

```
proc writenow {whereto what} {
  set fd [open $whereto "w"]
  puts $fd $what
  close $fd




}
```

(d) Verwenden Sie **`writenow`** in der Kommando-Option des **`OK`**-Knopfs, wobei wir weiterhin nur **`Hello world!`** schreiben. Vergessen Sie nicht das **`exit`** danach.

```
button .r.ok -text "OK" -command {

  writenow $name "Hello world!"


  exit

}
```

(e) Wir wollen das Schreiben auch schon dann auslösen, wenn die RETURN-Taste im Eingabefeld gedrückt wird. Binden Sie das **`Key-Return`** Ereignis an das Entry-Widget, damit die gleiche Funktion wie beim Drücken des **`OK`**-Knopfs ausgelöst wird.

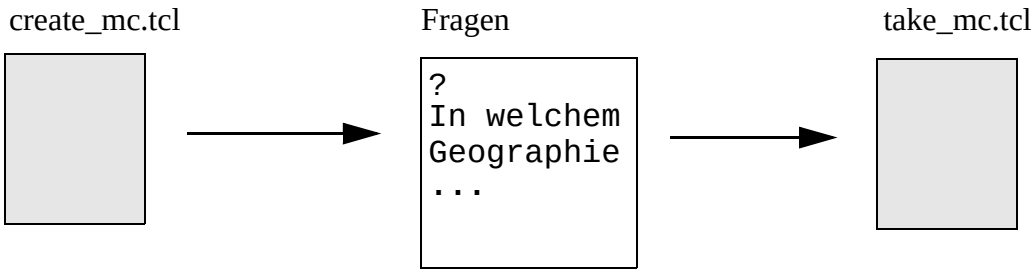
```
bind .r.ein <Key-Return> {
  writenow $name "Hello world!"
  exit}
```

alternativ auch

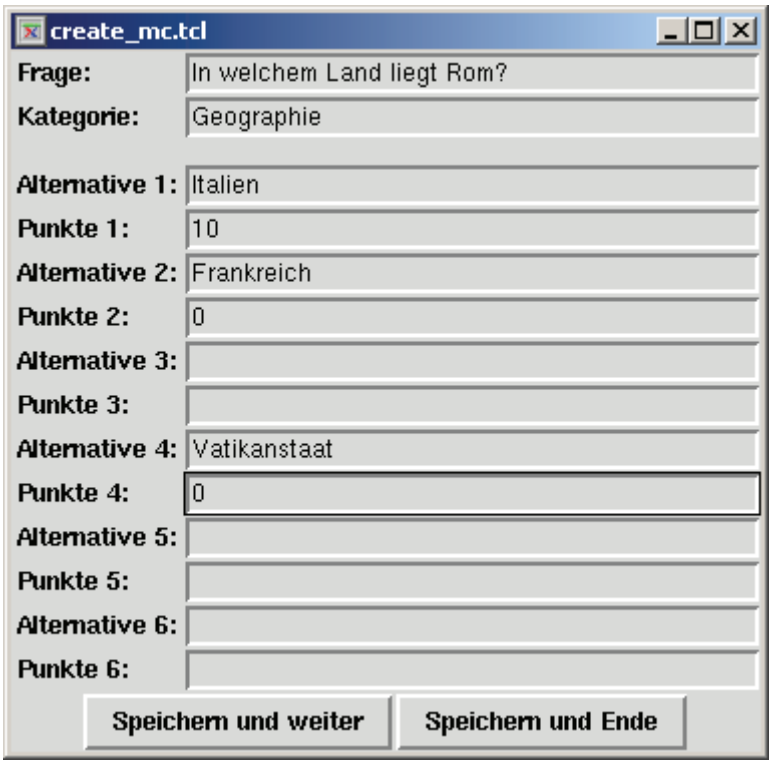
```
bind .r.ein <Key-Return> {.r.ok invoke}
```

Aufgabe 4:

Es geht um Multiple-Choice-Tests. Wie unten gezeigt, produzieren (entwerfen) wir mit **`create_mc.tcl`** interaktiv einen Test, der in der Datei **`Fragen`** festgehalten wird. Durch Aufruf eines Skripts **`take_mc.tcl`**, das aus der Datei **`Fragen`** liest, wird für die zu prüfende Person ein interaktiver Test produziert und es werden die Antworten ausgewertet.



Jede Frage gehört zu einer Kategorie und kann bis zu 6 Alternativen als Antwort haben, die alle mit Punkten versehen sind. Nur die nichtleeren Alternativen werden in **`Fragen`** geschrieben. Alle Eingaben beim Entwurf werden zuerst in einem Array zwischengespeichert und dann beim Drücken von **`Speichern und Ende`** geschrieben (siehe Aufgabe 5).



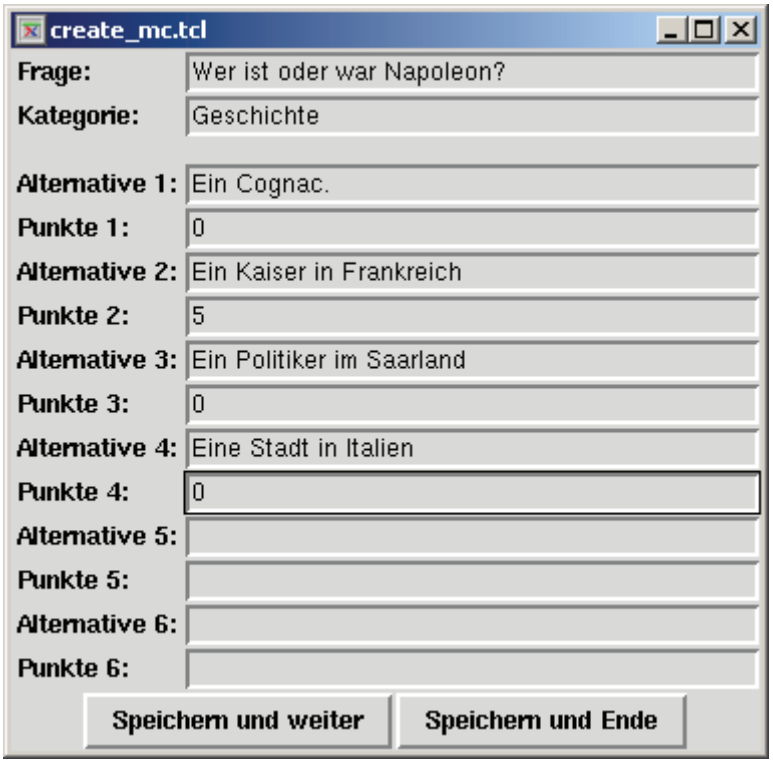
Ergänzen Sie die Lücken in dem Skript unten.

```
proc makeLabelAndEntry { name beschriftung } {
    frame .frame$name
    label .frame$name.label -text $beschriftung
    pack .frame$name.label -side left
    entry .frame$name.entry -textvariable $name -width 40
```

```
    pack .frame$name.entry -side right
    return .frame$name
}

foreach {n b} {frage "Frage:" kategorie "Kategorie:"} {
    pack [makeLabelAndEntry $n $b] -fill both
}

frame .frameLuft -height 12
pack .frameLuft

for {set i 1} {$i <= 6} {incr i} {
    pack [makeLabelAndEntry alt($i) "Alternative $i:"] -fill both
    pack [makeLabelAndEntry punkte($i) "Punkte $i:"] -fill both
}
```

Aufgabe 5:

Unten sehen Sie eine zweite Eingabemaske und darunter die Datei mit der Ausgabe der beiden produzierten Fragen. Fragen beginnen immer mit einem Fragezeichen, dann kommt der Fragetext, die Kategorie, Alternative 1, deren Punkte, usw.




```
wegner@holle:~/tcl> cat Fragen
?
In welchem Land liegt Rom?
Geographie
```

```
Italien
10
Frankreich
0
Vatikanstaat
0
?
Wer ist oder war Napoleon?
Geschichte
Ein Cognac.
0
Ein Kaiser in Frankreich.
5
Ein Politiker im Saarland.
0
Eine Stadt in Italien.
0
wegner@holle:~/tcl>
```

Ergänzen Sie auch hier die fehlenden Stellen. Geschrieben wird zunächst in den Array **mc**. Beachten Sie, daß auch im Fall von „**Speichern und Ende**“ die letzte Frage noch in diesen Array geschrieben werden muß, dann wird der ganze Array in die Datei übernommen.

```
frame .frameButtons
pack .frameButtons
button .frameButtons.weiter -text "Speichern und weiter" -command {
    incr anzahl
    set mc(F${anzahl}Fragetext) $frage
    set frage ""
    set mc(F${anzahl}Kat) $kategorie
    set kategorie ""
    for {set i 1} {$i <= 6} {incr i} {
        set mc(F${anzahl}Alt${i}Text) $alt($i)
        set alt($i) ""
        set mc(F${anzahl}Alt${i}Pkte) $punkte($i)
        set punkte($i) ""
    }
}
pack .frameButtons.weiter -side left
button .frameButtons.ende -text "Speichern und Ende" -command {

    .frameButtons.weiter invoke
    set fd [open "Fragen" "w"]
    for {set f 1} {$f <= $anzahl} {incr f} {
        puts $fd "?"
        puts $fd $mc(F${f}Fragetext)
        puts $fd $mc(F${f}Kat)
for {set i 1} {$i <= 6} {incr i} {
```

```
            if {$mc(F${f}Alt${i}Text) != ""} {
                puts $fd $mc(F${f}Alt${i}Text)
                puts $fd $mc(F${f}Alt${i}Pkte)
            }
        }
    }
    close $fd
    exit
}
pack .frameButtons.ende -side left
set anzahl 0
```

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Bei Ankreuzaufgaben können mehrere Antworten richtig sein. Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 120 Minuten.

Aufgabe 1:

Es geht ganz einfach los. Betrachten Sie das folgende Eingabefenster und den Programmtext (aus der Testphase) dazu unten. Ergänzen Sie die fehlenden Stellen!


```
frame .r
pack .r
label .r.prompt -text "Bitte Passwort eingeben:"
pack .r.prompt
entry .r.ein1 -textvariable pw1 -background white -width 40 -show "*"
pack .r.ein1 -padx 4 -pady 4
label .r.prompt2 -text "Bitte Passwort wiederholen:"
pack .r.prompt2
entry .r.ein2 -textvariable pw2 -background white -width 40 -show "*"
pack .r.ein2 -padx 4 -pady 4
button _____________________________________________-command {
   if {_____________________________________} {
     puts "Eingabe ungueltig weil ungleich!" } else {
     puts "Anmeldung erfolgt!" }
   exit}
pack .r.send -padx 4 -pady 4
focus _____________________________
```

Aufgabe 2:

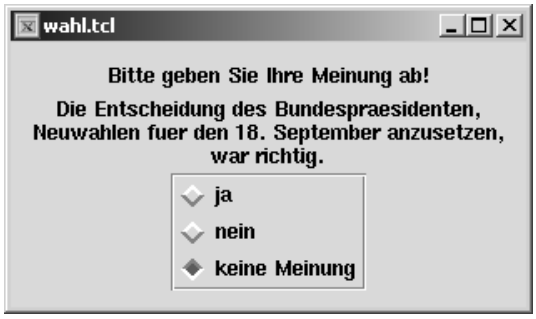
Das unten gezeigte Skript (analog zu Frage 1) liefert eine alternative Anordnung der Labels, Entries und Knöpfe.

```
label .r.prompt1 -text "Passwort eingeben:"
entry .r.ein1 -textvariable pw1 -background white -width 20 -show "*"
grid .r.prompt1 .r.ein1 -padx 4 -pady 4 -sticky e
label .r.prompt2 -text "Bitte das Passwort wiederholen:"
entry .r.ein2 -textvariable pw2 -background white -width 20 -show "*"
grid .r.prompt2 .r.ein2 -padx 4 -pady 4 -sticky e
button .r.ok -text "OK" -command {
   exit}
button .r.cancel -text "CANCEL" -command {
   exit}
grid .r.ok .r.cancel -padx 4 -pady 4
```

Zeichnen Sie die entstehende Ausgabe in das Fenster unten ein! Die Ausrichtung der Entries, Labels und Knöpfe muß klar erkennbar sein!

Aufgabe 3:

Betrachten Sie das folgende Fenster und das erzeugende Tcl-Skript unten.




```
proc erzeugeAuswahlFrame { win } {
  frame $win
  label $win.ansage -text "Bitte geben Sie Ihre Meinung ab!"
  pack $win.ansage
  label $win.frage -text "Die Entscheidung des Bundespraesidenten,
Neuwahlen fuer den 18. September anzusetzen,
war richtig."
  pack $win.frage
  frame $win.rahmen -borderwidth 2 -relief groove
  pack $win.rahmen
  set alts(1) ja
  set alts(2) nein
  set alts(3) "keine Meinung"
  for {set i 1} {$i <= 3} {incr i} {
    radiobutton $win.rahmen.rb$i \
      -text "$alts($i)" \
      -variable Antwort -value $i \
      -command {puts "Antwort lautet $Antwort"
        toplevel .danke
        label .danke.text -text "Danke fuer Ihr Votum."
        pack .danke.text -padx 10 -pady 10
        update idletasks
        grab .danke
        after 5000 exit
      }
    pack $win.rahmen.rb$i -anchor w
  }
  return $win
}

pack [erzeugeAuswahlFrame .frame] -padx 10 -pady 10
```

Beantworten Sie die Fragen auf der folgenden Seite.

Frage 3a:

Welcher Text (genaue Angabe) wird in der **`wish`** ausgegeben, wenn - wie oben gezeigt - die dritte Alternative („keine Meinung“) ausgewählt wurde?

_____

Frage 3b:

Kann man ohne Neustart des Programms zweimal abstimmen? Begründung!

Frage 3c:

Ist eine der drei Alternativen vorab ausgewählt, wenn das Fenster aufgemacht wird? Begründung!

Frage 3d:

Wo genau erscheint der Text **`"Danke fuer Ihr Votum."`**?

Frage 3e:

Statt die Texte der Alternativen aus einem assoziativen Array zu holen, hätte man einer Variablen **`alts`** eine Liste {**`ja nein "keine Meinung"`**} zuweisen können. Statt

```
-text $alts($i)
```

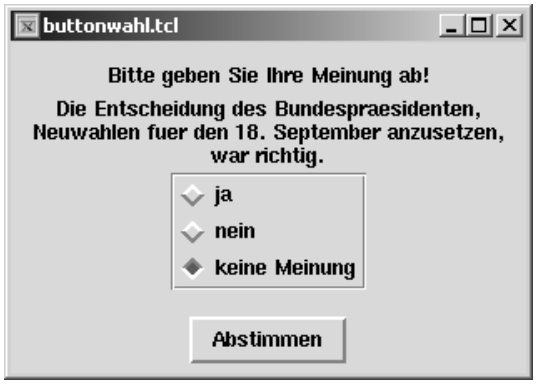
müsste man dann schreiben

```
-text [lindex ____________________________________________ ]
```

Hinweis: Beachten Sie, daß der Index in Listen von 0 an läuft, der Index im Programm aber weiterhin von 1 an laufen soll.

Aufgabe 4:

Das Widget von Aufgabe 3 wurde verändert. Speziell wurde ein **Abstimmen**-Knopf eingeführt.



Frage 4a: Ergänzen Sie die Lücken im Skript unten!

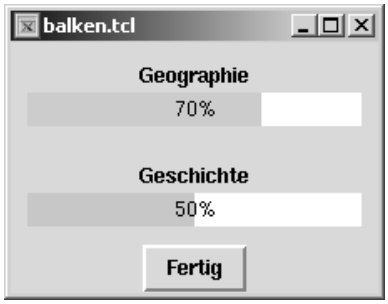
```
proc erzeugeAuswahlFrame { win } {
    frame $win
    label $win.ansage -text "Bitte geben Sie Ihre Meinung ab!"
    pack $win.ansage
    label $win.frage -text "Die Entscheidung des Bundespraesidenten,
Neuwahlen fuer den 18. September anzusetzen,
war richtig."
    pack $win.frage
    frame $win.rahmen -borderwidth 2 -relief groove
    pack $win.rahmen
    set alts(1) ja
    set alts(2) nein
    set alts(3) "keine Meinung"
    for {set i 1} {$i <= 3} {incr i} {
        radiobutton $win.rahmen.rb$i \
            -text "$alts($i)" \

            ____________________________________________________________

            ____________________________________________________________
        pack $win.rahmen.rb$i -anchor w
    }
    return $win
}

frame .frame
pack .frame
pack [erzeugeAuswahlFrame .frame.wahl ] -padx 10 -pady 10
button .weiter -text "Abstimmen" -command {
    puts "Antwort lautet $Antwort"
    destroy ____________________________________________________
    label .frame.danke -text "Danke fuer Ihr Votum."
    pack .frame.danke -padx 10 -pady 10
    .weiter configure -command exit -text "Ende"
    }
pack .weiter -padx 5 -pady 5
```

Frage 4b:

Wie sieht das Fenster nach Drücken der **`Abstimmen`**-Knopfes aus? Malen Sie eine Skizze!

Aufgabe 5:

Resultate von Tests, etwa hier ein Multiple-Choice Test zu Geographie und Geschichte, stellt man gern mit Balken dar. Dazu haben wir eine Prozedur **`balken_create`** entworfen. Füllen Sie die Lücken in der Prozedur und beim Aufruf für die hier gezeigte Ausgabe. Die Farbe für den oberen Balken ist **`lightblue`**, der darunter ist **`lightgreen`**.




```
proc balken_create {win val kat {color ""}} {
  frame $win
  label $win.what -text $kat
  pack $win.what
  set len 200
  canvas $win.display -borderwidt 0 -background white \
    -highlightthickness 0 -width $len -height 20
  pack $win.display -expand yes
  if {$color == ""} {
    set color "black"
  }
  $win.display create rectangle 0 0 [_____________________] 20 \
    -outline "" -fill $color
  $win.display create text [expr 0.5*$len] 10 \
    -anchor c -text "$val%"
  return $win
}

balken_create _____________________________________________
pack .geo -expand yes -fill both -padx 10 -pady 10
balken_create _____________________________________________
pack .geschi -expand yes -fill both -padx 10 -pady 10
button .fertig -text "Fertig" -command exit
pack .fertig
```

Aufgabe 6:

Wollte man für die Auswertung von Tests oder der Wahlbefragung oben einen Server einrichten, an den der Client die gewählte Antwort zu einer Frage schickt, dann

( ) geht dies nur, wenn Client und Server auf dem selben Rechner laufen.

( ) würde man eine socket-Verbindung aufmachen, was es in Tcl gibt.

( ) würde man eine socket-Verbindung aufmachen, was es aber nicht in Tcl, nur in C gibt.

( ) müßte man den Umweg über eine Web-Implementierung mit Browser als Client und Tclets gehen.

Aufgabe 7:

Um das Verhalten mehrerer Objekte auf dem **`canvas`** gleichzeitig umzuschalten, empfiehlt es sich, diesen eine Marke zu geben, unter der diese ansprechbar sind. Eine Marke vergibt man

( ) mit der **`-tags`**-Option bei jedem Objekt auf dem **`canvas`**.

( ) mit dem **`bindtag`**-Kommando.

( ) durch Vergabe eindeutiger IDs für jedes Objekt auf dem **`canvas`**.

( ) einem **`label`**-Widget.

Aufgabe 8:

Vergleichen Sie das **`after`**-Kommando, z.B. **`after 50 {mache_dies}`**, mit einer rekursiven Prozedur, z.B. **`proc mache_dies {...wait 50; mache_dies; ...}`**!

( ) Gibt es keine oder nur geringe Unterschiede.

( ) Das **`after`**-Kommando bewirkt, daß zur angegebenen Zeit ein Event ausgelöst wird, der **`mache_dies`** aufruft.

( ) Das Programm *X*, das ein **`after`**-Kommando enthält, wartet bei der Ausführung von **`after`** auf den Ablauf der Zeit, startet dann das enthaltene Kommando, hier **`mache_dies`**, wartet auf dessen Beendigung und macht dann mit X weiter.

( ) Innerhalb von **`mache_dies`** darf nicht wieder ein **`after`**-Kommando stehen.

( ) Rekursive Prozeduren sind in Tcl **nicht** möglich.

( ) Das **`after`**-Kommando läßt sich mit dem **`before`**-Kommando kombinieren, das wiederum auf Events, z.B. ein Mausereignis, reagiert (**`before <ENTER> wipe_feet`**).

ENDE DER KLAUSUR

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Musterlösung

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Bei Ankreuzaufgaben können mehrere Antworten richtig sein. Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 120 Minuten.

Aufgabe 1:

Es geht ganz einfach los. Betrachten Sie das folgende Eingabefenster und den Programmtext (aus der Testphase) dazu unten. Ergänzen Sie die fehlenden Stellen!

6 Punkte


```
frame .r
pack .r
label .r.prompt -text "Bitte Passwort eingeben:"
pack .r.prompt
entry .r.ein1 -textvariable pw1 -background white -width 40 -show "*"
pack .r.ein1 -padx 4 -pady 4
label .r.prompt2 -text "Bitte Passwort wiederholen:"
pack .r.prompt2
entry .r.ein2 -textvariable pw2 -background white -width 40 -show "*"
pack .r.ein2 -padx 4 -pady 4
button .r.send -text "Abschicken" -command {
   if {$pw1 != $pw2} {
     puts "Eingabe ungueltig weil ungleich!" } else {
     puts "Anmeldung erfolgt!" }
   exit}
pack .r.send -padx 4 -pady 4
focus .r.ein1
```

Aufgabe 2:

Das unten gezeigte Skript (analog zu Frage 1) liefert eine alternative Anordnung der Labels, Entries und Knöpfe.

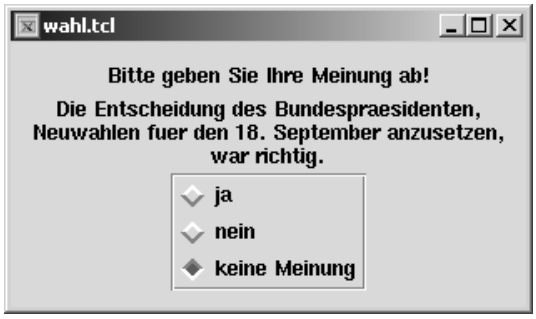
```
label .r.prompt1 -text "Passwort eingeben:"
entry .r.ein1 -textvariable pw1 -background white -width 20 -show "*"
grid .r.prompt1 .r.ein1 -padx 4 -pady 4 -sticky e
label .r.prompt2 -text "Bitte das Passwort wiederholen:"
entry .r.ein2 -textvariable pw2 -background white -width 20 -show "*"
grid .r.prompt2 .r.ein2 -padx 4 -pady 4 -sticky e
button .r.ok -text "OK" -command {
   exit}
button .r.cancel -text "CANCEL" -command {
   exit}
grid .r.ok .r.cancel -padx 4 -pady 4
```

Zeichnen Sie die entstehende Ausgabe in das Fenster unten ein! Die Ausrichtung der Entries, Labels und Knöpfe muß klar erkennbar sein!

4 Punkte

Aufgabe 3:

Betrachten Sie das folgende Fenster und das erzeugende Tcl-Skript unten.




```
proc erzeugeAuswahlFrame { win } {
  frame $win
  label $win.ansage -text "Bitte geben Sie Ihre Meinung ab!"
  pack $win.ansage
  label $win.frage -text "Die Entscheidung des Bundespraesidenten,
Neuwahlen fuer den 18. September anzusetzen,
war richtig."
  pack $win.frage
  frame $win.rahmen -borderwidth 2 -relief groove
  pack $win.rahmen
  set alts(1) ja
  set alts(2) nein
  set alts(3) "keine Meinung"
  for {set i 1} {$i <= 3} {incr i} {
    radiobutton $win.rahmen.rb$i \
      -text "$alts($i)" \
      -variable Antwort -value $i \
      -command {puts "Antwort lautet $Antwort"
        toplevel .danke
        label .danke.text -text "Danke fuer Ihr Votum."
        pack .danke.text -padx 10 -pady 10
        update idletasks
        grab .danke
        after 5000 exit
      }
    pack $win.rahmen.rb$i -anchor w
  }
  return $win
}

pack [erzeugeAuswahlFrame .frame] -padx 10 -pady 10
```

Beantworten Sie die Fragen auf der folgenden Seite.

je 2 Punkte für a - e

Frage 3a:

Welcher Text (genaue Angabe) wird in der **`wish`** ausgegeben, wenn - wie oben gezeigt - die dritte Alternative („keine Meinung“) ausgewählt wurde?

___ Antwort lautet 3 ____

Frage 3b:

Kann man ohne Neustart des Programms zweimal abstimmen? Begründung!

Nein, grab auf Toplevel-Fenster verhindert Eingabe.

Frage 3c:

Ist eine der drei Alternativen vorab ausgewählt, wenn das Fenster aufgemacht wird? Begründung!

Nein, $Antwort ist leer.

Frage 3d:

Wo genau erscheint der Text **`"Danke fuer Ihr Votum."`**?

Im neu angelegten Toplevel-Fenster.

Frage 3e:

Statt die Texte der Alternativen aus einem assoziativen Array zu holen, hätte man einer Variablen **`alts`** eine Liste {**`ja nein "keine Meinung"`**} zuweisen können. Statt

```
-text $alts($i)
```

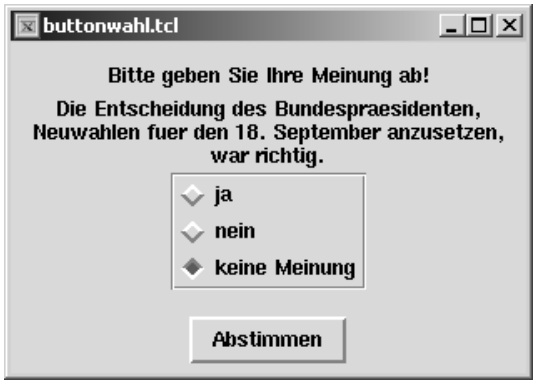
müsste man dann schreiben

```
-text [lindex $alts [expr $i -1] ]
```

Hinweis: Beachten Sie, daß der Index in Listen von 0 an läuft, der Index im Programm aber weiterhin von 1 an laufen soll.

Aufgabe 4:

Das Widget von Aufgabe 3 wurde verändert. Speziell wurde ein **Abstimmen**-Knopf eingeführt.



Frage 4a: Ergänzen Sie die Lücken im Skript unten! **4 Punkte**

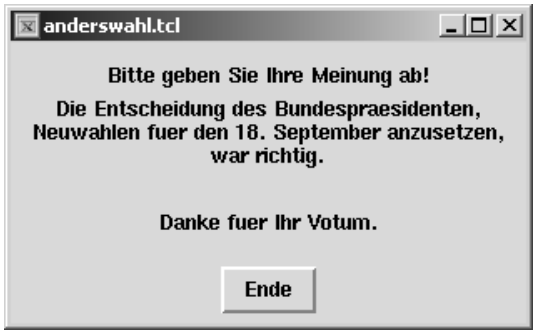
```
proc erzeugeAuswahlFrame { win } {
    frame $win
    label $win.ansage -text "Bitte geben Sie Ihre Meinung ab!"
    pack $win.ansage
    label $win.frage -text "Die Entscheidung des Bundespraesidenten,
Neuwahlen fuer den 18. September anzusetzen,
war richtig."
    pack $win.frage
    frame $win.rahmen -borderwidth 2 -relief groove
    pack $win.rahmen
    set alts(1) ja
    set alts(2) nein
    set alts(3) "keine Meinung"
    for {set i 1} {$i <= 3} {incr i} {
        radiobutton $win.rahmen.rb$i \
            -text "$alts($i)" \
            -variable Antwort \
            -value $i
        pack $win.rahmen.rb$i -anchor w
    }
    return $win
}

frame .frame
pack .frame
pack [erzeugeAuswahlFrame .frame.wahl ] -padx 10 -pady 10
button .weiter -text "Abstimmen" -command {
    puts "Antwort lautet $Antwort"
    destroy .frame.wahl
    label .frame.danke -text "Danke fuer Ihr Votum."
    pack .frame.danke -padx 10 -pady 10
    .weiter configure -command exit -text "Ende"
    }
pack .weiter -padx 5 -pady 5
```

Frage 4b:

Wie sieht das Fenster nach Drücken der **`Abstimmen`**-Knopfes aus? Malen Sie eine Skizze!

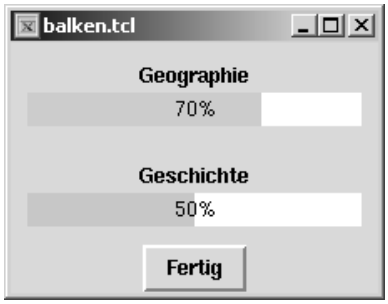
3 Punkte

Hinweis: Falls in 4a für **`destroy`** ein anderes Widget gewählt wurde, z.B. **`.frame.wahl.rahmen`**, muß das nächste Fenster entsprechend anders aussehen.



Aufgabe 5: **6 Punkte**

Resultate von Tests, etwa hier ein Multiple-Choice Test zu Geographie und Geschichte, stellt man gern mit Balken dar. Dazu haben wir eine Prozedur **`balken_create`** entworfen. Füllen Sie die Lücken in der Prozedur und beim Aufruf für die hier gezeigte Ausgabe. Die Farbe für den oberen Balken ist **`lightblue`**, der darunter ist **`lightgreen`**.




```
proc balken_create {win val kat {color ""}} {
  frame $win
  label $win.what -text $kat
  pack $win.what
  set len 200
  canvas $win.display -borderwidt 0 -background white \
    -highlightthickness 0 -width $len -height 20
  pack $win.display -expand yes
  if {$color == ""} {
    set color "black"
  }
  $win.display create rectangle 0 0 [ expr 2*$val ] 20 \
    -outline "" -fill $color
  $win.display create text [expr 0.5*$len] 10 \
    -anchor c -text "$val%"
  return $win
}
balken_create .geo 70 Geographie lightblue
pack .geo -expand yes -fill both -padx 10 -pady 10
balken_create .geschi 50 Geschichte lightgreen
pack .geschi -expand yes -fill both -padx 10 -pady 10
button .fertig -text "Fertig" -command exit
pack .fertig
```

2 Punkte

Aufgabe 6:

Wollte man für die Auswertung von Tests oder der Wahlbefragung oben einen Server einrichten, an den der Client die gewählte Antwort zu einer Frage schickt, dann

( ) geht dies nur, wenn Client und Server auf dem selben Rechner laufen.

(X) würde man eine socket-Verbindung aufmachen, was es in Tcl gibt.

( ) würde man eine socket-Verbindung aufmachen, was es aber nicht in Tcl, nur in C gibt.

( ) müßte man den Umweg über eine Web-Implementierung mit Browser als Client und Tclets gehen.

2 Punkte

Aufgabe 7:

Um das Verhalten mehrerer Objekte auf dem **`canvas`** gleichzeitig umzuschalten, empfiehlt es sich, diesen eine Marke zu geben, unter der diese ansprechbar sind. Eine Marke vergibt man

(X) mit der **`-tags`**-Option bei jedem Objekt auf dem **`canvas`**.

( ) mit dem **`bindtag`**-Kommando.

( ) durch Vergabe eindeutiger IDs für jedes Objekt auf dem **`canvas`**.

( ) einem **`label`**-Widget.

2 Punkte

Aufgabe 8:

Vergleichen Sie das **`after`**-Kommando, z.B. **`after 50 {mache_dies}`**, mit einer rekursiven Prozedur, z.B. **`proc mache_dies {...wait 50; mache_dies; ...}`**!

( ) Gibt es keine oder nur geringe Unterschiede.

(X) Das **`after`**-Kommando bewirkt, daß zur angegebenen Zeit ein Event ausgelöst wird, der **`mache_dies`** aufruft.

( ) Das Programm *X*, das ein **`after`**-Kommando enthält, wartet bei der Ausführung von **`after`** auf den Ablauf der Zeit, startet dann das enthaltene Kommando, hier **`mache_dies`**, wartet auf dessen Beendigung und macht dann mit X weiter.

( ) Innerhalb von **`mache_dies`** darf nicht wieder ein **`after`**-Kommando stehen.

( ) Rekursive Prozeduren sind in Tcl **nicht** möglich.

( ) Das **`after`**-Kommando läßt sich mit dem **`before`**-Kommando kombinieren, das wiederum auf Events, z.B. ein Mausereignis, reagiert (**`before <ENTER> wipe_feet`**).

ENDE DER KLAUSUR

Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1	5	
2	3	
3	4	
4	3 + 4	
5	4	
6	3	
7	3	
8	5	
9	2 + 2 + 2	
Summe	40	

Aufgabe 1:

Nach dem Spiel ist vor der Klausur. Hier dreht sich alles um Fußball. Damit man sich mit ausländischen Gästen unterhalten kann, gibt es hier ein kleines Programm für eine Übersetzungshilfe. Ergänzen Sie die fehlenden Stellen im Programmtext unten!


```
pack [frame .r]
# Belegen des Arrays mit Namen-Wert-Paaren
array set uebersetz {Bier beer Eintrittskarte ticket \
  Eckball "corner kick" Strafstoß "penalty kick"}

label .r.prompt -text "_____________________________________________"
pack .r.prompt
entry .r.ein1 -textvariable frage -background white -width 50
pack .r.ein1 -padx 4 -pady 4
label .r.antwort -text "Übersetzung:"
pack .r.antwort -padx 4 -pady 4
button .r.send -text "Abschicken" -command {

if {[expr [lsearch [array _____________________] $frage] < 0 ]} {
     .r.antwort configure -text "Übersetzung: Unbekannt"} else {

    .r.antwort configure -text "Übersetzung: _____________________"
   }
  }
pack .r.send -padx 4 -pady 4

focus .r.ein1
```

Aufgabe 2:

In Aufgabe 1 wollen wir die Übersetzung auch dadurch starten, daß man nach der Eingabe des Begriffs die Return-Taste drückt. Binden Sie dazu das **`Key-Return`** Ereignis an das Entry-Widget, das wiederum als auszuführendes Skript das Drücken des Abschicken-Knopfes auslöst. Button-Widgets haben dazu die **`invoke`**-Methode.

Aufgabe 3:

Beim Spiel Niederlande gegen Portugal wurden vier rote Karten verteilt, so daß gegen Ende neun gegen neun antraten. Nach FIFA-Regeln sollte eine Mannschaft aus mindestens sieben Spielern bestehen.

Sei in einer Listen-Variablen, z.B. **`TeamA`**, die Liste der Spieler auf dem Platz gespeichert. Geben Sie ein einzelnes Tcl-Kommando an, das prüft, ob **`TeamA`** noch mindestens 7 Einträge hat und wenn nicht, die Ausgabe „Spiel-Abbruch“ erzeugt.

Aufgabe 4:

Betrachten Sie das folgende Programm.

```
set A {Deutschland Polen Ecuador "Costa Rica"}

foreach team1 $A {
  foreach team2 [lrange $A [expr [lsearch $A $team1] + 1] end] {
    puts "$team1 : $team2"
  }
}
```

(a) Wie lautet die Ausgabe des Programms?

(b) Damit der Algorithmus oben für alle Gruppen funktioniert, bauen Sie ihn in eine Prozedur

```
gruppenspiele {win gruppe gruppenid} {...}
```

ein. Der Parameter **win** ist der Pfadname für ein bereits existierendes Fensterwidget, **gruppe** die Liste der Länder und **gruppenid** z.B. der Buchstabe A, B oder H.

Verändern Sie ferner den Ausgabeteil der Prozedur (wo bisher nur **puts** steht) so, daß in den Rahmen **$win.f$gruppenid** die Überschriftszeile „**Spiele der Gruppe** ...“ als **Label** und darunter alle Gruppenspiele als **Button** eingefügt werden. Die Buttons unterscheiden sich über die Pfadkomponente **.b$i** und haben die Spielepaarungen **"$team1 : $team2"** als Beschriftung. Die Aktionen (**-command**) brauchen Sie nicht vorzusehen. Ein möglicher Aufruf von **gruppenspiele** steht ganz unten. Teile der Prozedur, insbesondere die Schleifen von oben, haben wir schon hierher mit Lücken zur Anpassung des Texts kopiert.

```
proc gruppenspiele {win gruppe gruppenid} {
  set w $win.f$gruppenid
  frame $w

  label ____________________________________________________________

  pack ____________________________________________________________
  set i 0
  foreach team1 _________________________ {
    foreach team2 [lrange ______ [expr [lsearch _______ $team1]+1] end] {
      incr i

      ____________________________________________________________

      ____________________________________________________________
    }
  }
  return $w
}
```

Verwendung z.B. wie folgt:

```
set gruppeA {Deutschland Polen Ecuador "Costa Rica"}
frame .s
pack .s
pack [gruppenspiele .s $gruppeA "A"]
```

Aufgabe 5:

Das unten gezeigte Skript soll eine Art Spielstandsanzeige - ähnlich wie in der übernächsten Aufgabe 7 - liefern. Zeichnen Sie in das Bild unten das anfängliche Aussehen ein. Die Anordnung der Eingabefelder und Knöpfe muß klar erkennbar sein.

```
proc tic {min sec} {
  ...
}

label .team1 -text "Heimmannschaft:"
entry .ein1 -textvariable heim -background white -width 20
entry .ein2 -textvariable toreheim -background white -width 2
grid .team1 .ein1 .ein2 -padx 4 -pady 4 -sticky e
label .team2 -text "Gast:"
entry .ein3 -textvariable gast -background white -width 20
entry .ein4 -textvariable toregast -background white -width 2
grid .team2 .ein3 .ein4 -padx 4 -pady 4 -sticky e
label .zeit -text "Spielzeit 00:00"
grid .zeit
grid configure .zeit -columnspan 3
frame .b
grid .b
grid configure .b -columnspan 3
button .b.start -text "ANPFIFF" -command {
   tic 0 0 }
button .b.stop -text "ABPFIFF" -command {
   exit}
pack .b.start .b.stop -side left -padx 4 -pady 4
focus .ein1
```

Aufgabe 6:

In der Spielanzeige oben soll die Spielzeit loslaufen, wenn wir den ANPFIFF-Knopf drücken. Dazu steht dort **`-command "tic 0 0"`**. Ergänzen Sie die folgende Prozedur **`tic`**, die eine selbstgebaute Uhr ist (nicht zu empfehlen, besser würde man mit dem Tcl-Kommando **`clock`** arbeiten).

```
proc tic {min sec} {

  incr sec

  if { [expr ____________________] } { incr min; set sec 0 }

  .zeit _______________ -text [format "Spielzeit %02d:%02d" $min $sec]

  after 1000 "____________________"

}
```

Aufgabe 7:

Beim Drücken des ABPFIFF-Knopfes sollen die Werte der Eingabefelder ausgegeben werden. Bei einer Anzeige wie hier gezeigt,

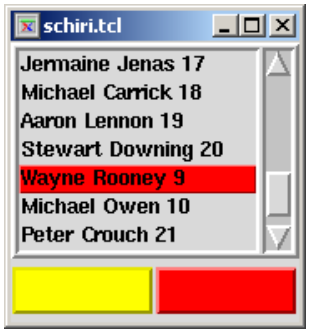
soll mit **`puts`** die folgende Ausgabe geliefert werden.

```
Das Spiel Deutschland - Italien endete 0 : 2.
```

Ergänzen Sie die **`-command`** Option des ABPFIFF-Knopfs!

Aufgabe 8:

Krasse Fehlleistungen der Schiedsrichter, wie z.B. die rote Karte erst nach der dritten gelben Karte, ließen sich gut vermeiden, wenn jeder „Schiri“ ein Notebook mit folgendem Tcl/Tk-Programm mit sich führen würde (extra Akkus für die Verlängerung nicht vergessen!).



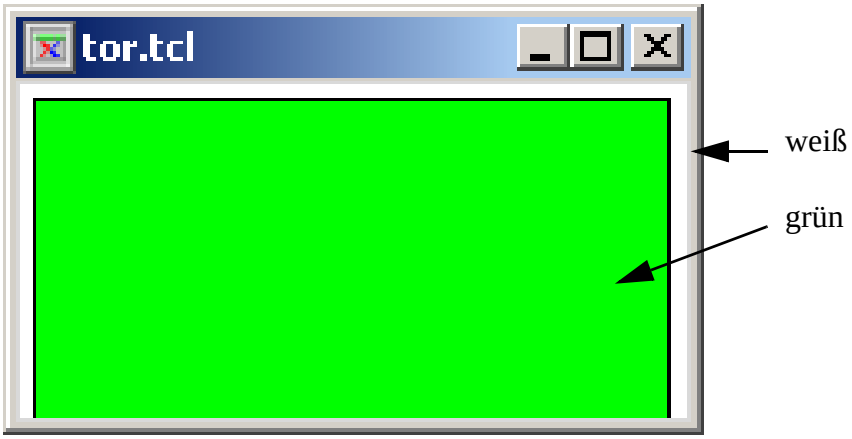

```
proc setcolor {inlistbox whatcolor} {
  $inlistbox itemconfigure [$inlistbox curselection] \
   -background "$whatcolor" -selectbackground "$whatcolor"
}
proc getcolor {inlistbox} {
  return [$inlistbox itemcget [$inlistbox curselection] -background]
}

frame .r -relief raised -borderwidth 2
pack .r
listbox .r.teamA -yscroll ".r.scroll set" -setgrid 1 \
  -height 7
scrollbar .r.scroll -command ".r.teamA yview"
pack .r.scroll -side right -fill y
pack .r.teamA -side left -expand 1 -fill both

frame .k
pack .k  -pady 4

button .k.gelb -background "yellow" -activebackground "yellow" \
  -width 10 -padx 4 -pady 4 -command {setcolor .r.teamA "yellow"}
pack .k.gelb -side left

button .k.rot -background "red" -activebackground "red" -width 10 \
  -padx 4 -pady 4 -command {setcolor .r.teamA "red"}
pack .k.rot -side right

.r.teamA insert 0 \
"Paul Robinson 1"  "David James 13"  "Scott Carson 22"  \
... "Stewart Downing 20"  "Wayne Rooney 9"  "Michael Owen 10"  \
"Peter Crouch 21"  "Theo Walcott 23"
```

Ändern Sie das Kommando für den **gelben** Knopf, indem Sie zuerst mit **`getcolor`** prüfen, welche Farbe der selektierte Spieler hat. Ist das Resultat gelb, setzen Sie ihn auf rot. Hat er schon rot, wird die Farbe nicht neu gesetzt, sonst setzen Sie die Farbe auf gelb.

```
button .k.gelb -background "yellow" -activebackground "yellow" \
    -width 10 -padx 4 -pady 4 -command {









    }
pack .k.gelb -side left
```

Aufgabe 9:

(a) Das Runde muß in das Eckige! Malen Sie in das Bild die fehlenden Graphikteile aus dem Programm unten. Farben deuten Sie durch Pfeile an.



(b) Die mit **`oval`** erzeugten Leinwandobjekte sollen **einzeln** die Farbe wechseln, wenn man mit der Maus darüberstreicht. Ergänzen Sie dazu die Lücken im Programm.

```
canvas .t -width 200 -height 100 -bg white
pack .t
.t create rectangle 5 5 195 200 -fill green
.t create oval 10 10 40 40 -fill black -outline white -width 5
    -tags _____________________________
.t create oval 160 70 190 100 -fill black -outline white -width 5
    -tags _____________________________
.t bind obenunten <Enter> {
   .t itemconfigure _____________________ -fill red}
.t bind obenunten <Leave> {
   .t itemconfigure _____________________ -fill black}
.t bind obenunten <1> {
   .t create text 100 50 -text "TOR!" -anchor sw -tags tortext
   after 500 ".t delete tortext" }
```

(c) Was bewirkt das letzte **`bind`** für das Ereignis **`<1>`**? Eine kurze Beschreibung genügt.

Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: **Musterlösung** Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1	5	
2	3	
3	4	
4	3 + 4	
5	4	
6	3	
7	3	
8	5	
9	2 + 2 + 2	
Summe	40	

Aufgabe 1:

Nach dem Spiel ist vor der Klausur. Hier dreht sich alles um Fußball. Damit man sich mit ausländischen Gästen unterhalten kann, gibt es hier ein kleines Programm für eine Übersetzungshilfe. Ergänzen Sie die fehlenden Stellen im Programmtext unten!


```
pack [frame .r]
# Belegen des Arrays mit Namen-Wert-Paaren
array set uebersetz {Bier beer Eintrittskarte ticket \
  Eckball "corner kick" Strafstoß "penalty kick"}
label .r.prompt -text "Bitte Begriff eingeben:"
pack .r.prompt
entry .r.ein1 -textvariable frage -background white -width 50
pack .r.ein1 -padx 4 -pady 4
label .r.antwort -text "Übersetzung:"
pack .r.antwort -padx 4 -pady 4
button .r.send -text "Abschicken" -command {
if {[expr [lsearch [array names uebersetz] $frage] < 0 ]} {
     .r.antwort configure -text "Übersetzung: Unbekannt"} else {
    .r.antwort configure -text "Übersetzung: $uebersetz($frage)"
   }
  }
pack .r.send -padx 4 -pady 4
focus .r.ein1
```

Aufgabe 2:

In Aufgabe 1 wollen wir die Übersetzung auch dadurch starten, daß man nach der Eingabe des Begriffs die Return-Taste drückt. Binden Sie dazu das **Key-Return** Ereignis an das Entry-Widget, das wiederum als auszuführendes Skript das Drücken des Abschicken-Knopfes auslöst. Button-Widgets haben dazu die **invoke**-Methode.

```
bind .r.ein1 <Key-Return> {.r.send invoke}
```

Aufgabe 3:

Beim Spiel Niederlande gegen Portugal wurden vier rote Karten verteilt, so daß gegen Ende neun gegen neun antraten. Nach FIFA-Regeln sollte eine Mannschaft aus mindestens sieben Spielern bestehen.

Sei in einer Listen-Variablen, z.B. **TeamA**, die Liste der Spieler auf dem Platz gespeichert. Geben Sie ein einzelnes Tcl-Kommando an, das prüft, ob **TeamA** noch mindestens 7 Einträge hat und wenn nicht, die Ausgabe „Spiel-Abbruch“ erzeugt.

```
if {[llength $TeamA] < 7} {
    puts "Spiel-Abbruch"
}
```

Aufgabe 4:

Betrachten Sie das folgende Programm.

```
set A {Deutschland Polen Ecuador "Costa Rica"}

foreach team1 $A {
  foreach team2 [lrange $A [expr [lsearch $A $team1] + 1] end] {
    puts "$team1 : $team2"
  }
}
```

(a) Wie lautet die Ausgabe des Programms?

```
Deutschland : Polen
Deutschland : Ecuador
Deutschland : Costa Rica
Polen : Ecuador
Polen : Costa Rica
Ecuador : Costa Rica
```

(b) Damit der Algorithmus oben für alle Gruppen funktioniert, bauen Sie ihn in eine Prozedur

```
gruppenspiele {win gruppe gruppenid} {...}
```

ein. Der Parameter **win** ist der Pfadname für ein bereits existierendes Fensterwidget, **gruppe** die Liste der Länder und **gruppenid** z.B. der Buchstabe A, B oder H.

Verändern Sie ferner den Ausgabeteil der Prozedur (wo bisher nur **puts** steht) so, daß in den Rahmen **$win.f$gruppenid** die Überschriftszeile „**Spiele der Gruppe** ...“ als **Label** und darunter alle Gruppenspiele als **Button** eingefügt werden. Die Buttons unterscheiden sich über die Pfadkomponente **.b$i** und haben die Spielepaarungen **"$team1 : $team2"** als Beschriftung. Aktionen (**-command**) brauchen Sie nicht vorzusehen. Ein möglicher Aufruf von **gruppenspiele** steht ganz unten. Teile der Prozedur, insbesondere die Schleifen von oben, haben wir schon hierher mit Lücken zum Anpassen des Texts kopiert.

```
proc gruppenspiele {win gruppe gruppenid} {
  set w $win.f$gruppenid
  frame $w

  label $w.ueberschrift -text "Spiele der Gruppe $gruppenid"

  pack $w.ueberschrift
  set i 0
  foreach team1 $gruppe {
    foreach team2 [lrange $gruppe [expr [lsearch $gruppe $team1] + 1] end] {
      incr i

      button $w.b$i -text "$team1 : $team2"

      pack $w.b$i
    }
  }
  return $w
}
```

Verwendung z.B. wie folgt:

```
set gruppeA {Deutschland Polen Ecuador "Costa Rica"}
frame .s
pack .s
pack [gruppenspiele .s $gruppeA "A"]
```

Aufgabe 5:

Das unten gezeigte Skript soll eine Art Spielstandsanzeige - ähnlich wie in der übernächsten Aufgabe 7 - liefern. Zeichnen Sie in das Bild unten das anfängliche Aussehen ein. Die Anordnung der Eingabefelder und Knöpfe muß klar erkennbar sein.

```
proc tic {min sec} {
  ...
}

label .team1 -text "Heimmannschaft:"
entry .ein1 -textvariable heim -background white -width 20
entry .ein2 -textvariable toreheim -background white -width 2
grid .team1 .ein1 .ein2 -padx 4 -pady 4 -sticky e
label .team2 -text "Gast:"
entry .ein3 -textvariable gast -background white -width 20
entry .ein4 -textvariable toregast -background white -width 2
grid .team2 .ein3 .ein4 -padx 4 -pady 4 -sticky e
label .zeit -text "Spielzeit 00:00"
grid .zeit
grid configure .zeit -columnspan 3
frame .b
grid .b
grid configure .b -columnspan 3
button .b.start -text "ANPFIFF" -command {
   tic 0 0 }
button .b.stop -text "ABPFIFF" -command {
   exit}
pack .b.start .b.stop -side left -padx 4 -pady 4
focus .ein1
```

Aufgabe 6:

In der Spielanzeige oben soll die Spielzeit loslaufen, wenn wir den ANPFIFF-Knopf drücken. Dazu steht dort **`-command "tic 0 0"`**. Ergänzen Sie die folgende Prozedur **`tic`**, die eine selbstgebaute Uhr ist (nicht zu empfehlen, besser würde man mit dem Tcl-Kommando **`clock`** arbeiten).

```
proc tic {min sec} {

  incr sec

  if { [expr $sec >= 60] } { incr min; set sec 0 }

  .zeit configure -text [format "Spielzeit %02d:%02d" $min $sec]

  after 1000 "tic $min $sec"

}
```

Aufgabe 7:

Beim Drücken des ABPFIFF-Knopfes sollen die Werte der Eingabefelder ausgegeben werden. Bei einer Anzeige wie hier gezeigt,

soll mit **`puts`** die folgende Ausgabe geliefert werden.

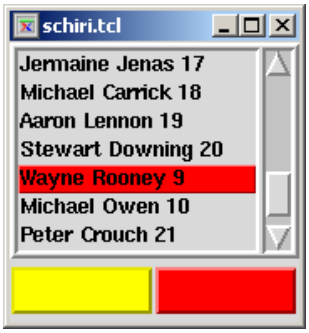
```
Das Spiel Deutschland - Italien endete 0 : 2.
```

Ergänzen Sie die **`-command`** Option des ABPFIFF-Knopfs!

```
-command {
   puts "Das Spiel $heim - $gast endete $toreheim : $toregast"
}
```

Aufgabe 8:

Krasse Fehlleistungen der Schiedsrichter, wie z.B. die rote Karte erst nach der dritten gelben Karte, ließen sich gut vermeiden, wenn jeder „Schiri“ ein Notebook mit folgendem Tcl/Tk-Programm mit sich führen würde (extra Akkus für die Verlängerung nicht vergessen!).



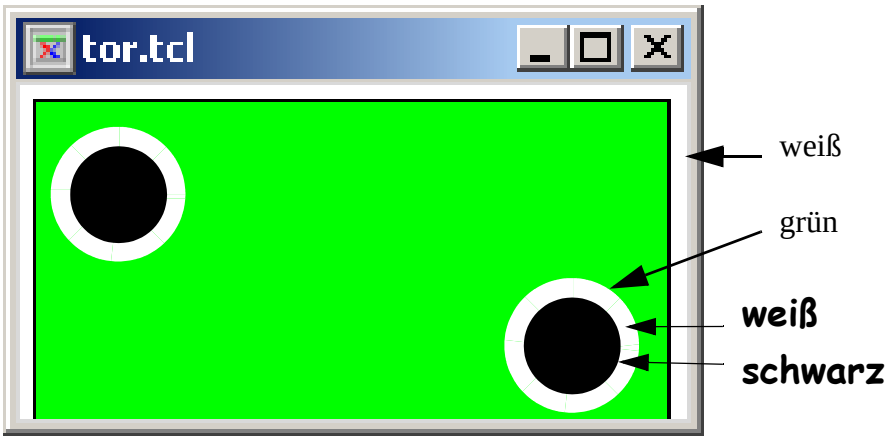

```
proc setcolor {inlistbox whatcolor} {
  $inlistbox itemconfigure [$inlistbox curselection] \
   -background "$whatcolor" -selectbackground "$whatcolor"
}
proc getcolor {inlistbox} {
  return [$inlistbox itemcget [$inlistbox curselection] -background]
}

frame .r -relief raised -borderwidth 2
pack .r
listbox .r.teamA -yscroll ".r.scroll set" -setgrid 1 \
  -height 7
scrollbar .r.scroll -command ".r.teamA yview"
pack .r.scroll -side right -fill y
pack .r.teamA -side left -expand 1 -fill both

frame .k
pack .k  -pady 4

button .k.gelb -background "yellow" -activebackground "yellow" \
  -width 10 -padx 4 -pady 4 -command {setcolor .r.teamA "yellow"}
pack .k.gelb -side left

button .k.rot -background "red" -activebackground "red" -width 10 \
  -padx 4 -pady 4 -command {setcolor .r.teamA "red"}
pack .k.rot -side right

.r.teamA insert 0 \
"Paul Robinson 1"  "David James 13"  "Scott Carson 22"  \
... "Stewart Downing 20"  "Wayne Rooney 9"  "Michael Owen 10"  \
"Peter Crouch 21"  "Theo Walcott 23"
```

Ändern Sie das Kommando für den **gelben** Knopf, indem Sie zuerst mit **`getcolor`** prüfen, welche Farbe der selektierte Spieler hat. Ist das Resultat gelb, setzen Sie ihn auf rot. Hat er schon rot, wird die Farbe nicht neu gesetzt, sonst setzen Sie die Farbe auf gelb.

```
button .k.gelb -background "yellow" -activebackground "yellow" \
    -width 10 -padx 4 -pady 4 -command {
    if {[getcolor .r.teamA] == "yellow"]} {
      setcolor .r.teamA "red"
    } else {
        if {[getcolor .r.teamA] != "red"]} {
            setcolor .r.teamA "yellow"
        }
     }
    }
pack .k.gelb -side left
```

Aufgabe 9:

(a) Das Runde muß in das Eckige! Malen Sie in das Bild die fehlenden Graphikteile aus dem Programm unten. Farben deuten Sie durch Pfeile an.



(b) Die mit **`oval`** erzeugten Leinwandobjekte sollen **einzeln** die Farbe wechseln, wenn man mit der Maus darüberstreicht. Ergänzen Sie dazu die Lücken im Programm.

```
canvas .t -width 200 -height 100 -bg white
pack .t
.t create rectangle 5 5 195 200 -fill green
.t create oval 10 10 40 40 -fill black -outline white -width 5
    -tags obenunten
.t create oval 160 70 190 100 -fill black -outline white -width 5
    -tags obenunten
.t bind obenunten <Enter> {
   .t itemconfigure current -fill red}
.t bind obenunten <Leave> {
   .t itemconfigure current -fill black}
.t bind obenunten <1> {
   .t create text 100 50 -text "TOR!" -anchor sw -tags tortext
   after 500 ".t delete tortext" }
```

(c) Was bewirkt das letzte **`bind`** für das Ereignis **`<1>`**? Eine kurze Beschreibung genügt.

Bei Klick mit der linken Maustaste auf einen der Kreise erscheint in Tormitte der Text „TOR!". Der Text wird nach 500 ms wieder gelöscht.

Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Bei Ankreuzaufgaben können mehrere Antworten richtig sein. Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1a	4	
1b	6	
1c	2	
2a	3	
2b	2	
3	4	
4a	2	
4b	2	
4c	4	
4d	6	
4e	7	
Summe	42	

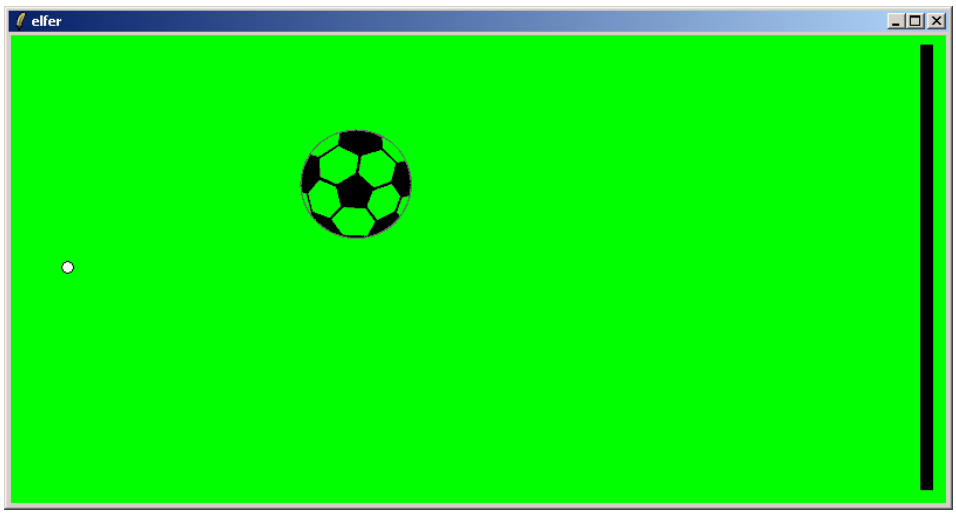
Aufgabe 1:

Wir sind immer noch beim Fußball. Wir wollen das Elfmeterschießen üben. In dem Programm unten zieht man den Ball mit gedrückter linker Maustaste in Richtung Tor (der schwarze Balken rechts). Wenn man die Maustaste losläßt, fliegt der Ball selbstständig weiter.

Das Programm verwendet das Widget-Kommando **`coords`**, das ähnlich wie das Kommando **`itemconfigure`** für Items auf der Leinwand die Koordinaten ändert.

pathName **coords** *tagOrId* ?*x0 y0* ...?

(a) Ergänzen Sie die fehlenden Stellen im Programmtext unten!




```
proc fliege {worauf was x y d} {
  ____________________________ coords ____________________________________
  after 20 "fliege $worauf $was [expr $x + 10] \
        [expr int($y + 10 * $d)] $d"

}

canvas .c -width 800 -height 400 -bg green
pack .c
image create photo ball -file "C:/ball.gif"
.c create oval 45 195 55 205 -fill white -tag elferpunkt
.c create image 50 200 -image ball -tag fussball
.c create rectangle 780 10 790 390 -fill black -tag tor
.c bind fussball <B1-Motion> {
  .c coords fussball %x %y}
.c bind fussball <ButtonRelease-1> {
  set m [expr double(%y - 200) / (%x - 50)]
  fliege .c fussball %x %y $m}
```

(b) Im Programm oben kann man den Ball mit der Maus bis ins Tor ziehen. Das sollte nicht sein!

Bauen Sie eine Abfrage ein, die den Schuß mit einer Meldung „Ungültig“ (als text item auf dem canvas) abbricht, wenn die X-Koordinate der Maus zum Zeitpunkt des Loslassens der Taste schon über einem Viertel der Leinwandbreite liegt.

```
.c bind fussball <ButtonRelease-1> {

    set m [expr double(%y - 200) / (%x - 50)]
    fliege .c fussball %x %y $m

}
```

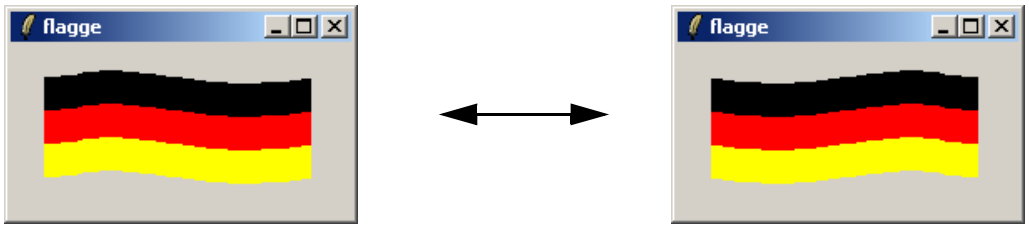
(c) Vor das Tor stellen wir einen „Torwart“, dargestellt durch einen roten Balken, 10 Pixel breit, 50 Pixel hoch, 10 Pixel Abstand zum Tor. Legen Sie den Torwart an und geben Sie ihm den Tag **`olli`**.[1]

1. Das Animieren des Torwarts (rauf und runterbewegen) verschieben wir auf die Klausur im Sommer.

Aufgabe 2:

Kein Länderspiel ohne Schwenken der Fahne. Die folgende Fahne haben wir uns aus Polygonen selbst gestrickt.

(a)Ergänzen Sie die fehlenden Stellen!



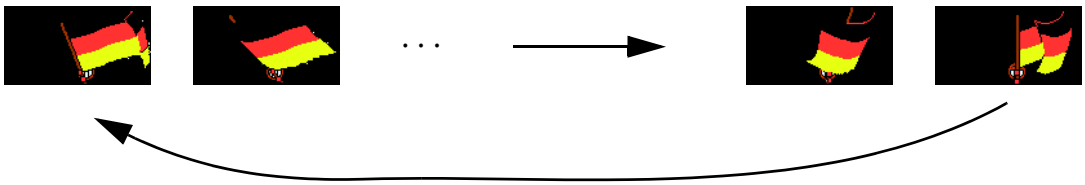

```
proc hin {} {
  .c coords schwarz 20 20 20 20 60 15 100 20 140 25 180 20 \
    180 20 180 40 180 40 140 45 100 40 60 35 20 40 20 40
  .c coords rot 20 40 20 40 60 35 100 40 140 45 180 40 \
    180 40 180 60 180 60 140 65 100 60 60 55 20 60 20 60
  .c coords gold 20 60 20 60 60 55 100 60 140 65 180 60 \
    180 60 180 80 180 80 140 85 100 80 60 75 20 80 20 80
  after [expr int(rand() * 600 + 100)] ____________
}

proc her {} {
  .c coords schwarz 20 20 20 20 60 25 100 20 140 15 180 20 \
    180 20 180 40 180 40 140 35 100 40 60 45 20 40 20 40
  .c coords rot 20 40 20 40 60 45 100 40 140 35 180 40 \
    180 40 180 60 180 60 140 55 100 60 60 65 20 60 20 60
  .c coords gold 20 60 20 60 60 65 100 60 140 55 180 60 \
    180 60 180 80 180 80 140 75 100 80 60 85 20 80 20 80
  after [expr int(rand() * 600 + 100)] _____________
}

canvas .c -width 200 -height 100
pack .c
.c create polygon 20 20 180 20 180 40 20 40 \
  -tag schwarz -fill black -smooth 1
.c create polygon 20 40 180 40 180 60 20 60 \
  -tag rot -fill red -smooth 1
.c create polygon 20 60 180 60 180 80 20 80 \
  -tag gold -fill yellow -smooth 1
```

_____

(b) In welchem Intervall (Zeitabstand) wird zwischen den beiden Bildern zufällig gewechselt?

Aufgabe 3:

Im Internet findet man viele GIF-Bilderfolgen für schwenkende Fahnen, z.B. die folgende.



Ergänzen Sie die Lücken im Programm unten!

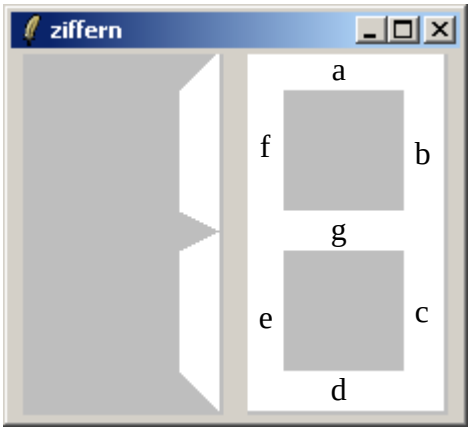
```
proc changeflag {j} {
   .c itemconfigure fahne -image _____________________________
   if {$j==7} {set j 0} else {incr j}
   after 50 "changeflag $j"
}

canvas .c -width 200 -height 100
pack .c
for {set i 0} {$i<8} {incr i} {
   image create photo f$i -file "fl-de07_frame_00$i.gif"
}
.c create image 100 50 -image f0 -tag ____________________________
changeflag 0
```

Aufgabe 4:

Bei internationalen Begegnungen im Fußball gibt es einen 4. Offiziellen, der den Schiedsrichter unterstützt. Bei Auswechslungen hält er eine elektronische Tafel hoch, auf der in rot die Rückennummer des auszuwechselnden Spielers und in grün die Nummer des einzuwechselnden steht. Dafür braucht man 4 Ziffern. Mit der Tafel wird auch z.B. die Nachspielzeit angezeigt.

Im Internet bieten Sportausstatter die Tafeln für schlappe 1300,- EUR an. Das ist unsere Chance als Tcl/Tk-Programmierer.

Unten zeigen wir eine selbstgebaute Tafel mit zwei Ziffern. Jede Ziffer ist eine Leinwand (canvas widget), die Ziffern sind aus Segmenten (Polygonen) in der Art einer LCD-Anzeige zusammengesetzt. Die Segmente sind von a - g durchnumeriert.



Hier ist die Prozedur zum Erzeugen einer Leinwand mit allen 7 Segmenten. Über unterschiedliche Werte von **`id`** können wir mehrere Leinwände (damit mehrere Ziffern nebeneinander) erzeugen.

```
proc createdigit {win id} {
  canvas $win.$id -width 100 -height 180 -bg gray

  $win.$id create polygon 0 0 100 0 \
    80 20 20 20 -fill white -tags a

  $win.$id create polygon 100 0 100 90 \
    80 80 80 20 -fill white -tags b

  ...

  $win.$id create polygon 20 100 \
    0 90 20 80 80 80 100 90 80 100\
    -fill white -tags g

  return $win.$id
}
```

(a) Es folgt die Prozedur, um alle Segmente einzuschalten (**`-fill white`**). Fügen Sie eine zweiten Parameter **`farbe`** hinzu und ändern Sie den Prozedurrumpf, um beliebig gefärbte Segmente zu erzeugen, z.B. rote und grüne.

```
proc turnallon {c _____________________} {
  foreach item {a b c d e f g} {
    $c itemconfigure $item -fill white
  }
}
```

(b) Mit der Prozedur **`turnoff`** kann man eine Liste von Segmenten (**`items`**) abschalten. Ergänzen Sie die fehlende Stelle.

```
proc turnoff {c items} {
  foreach ____________________________________ {
    $c itemconfigure $item -fill gray
  }
}
```

(c) Ziffern werden angezeigt, indem Segmente ausgeschaltet werden. Ergänzen Sie wieder die Färbung der Segmente und fügen Sie in **`switch`** eine Alternative ein, bei der **`" "`** (blank) für Parameter **`digit`** alle Segmente ausschaltet.

```
proc showdigit {where digit _____________________} {
  turnallon $where ____________________
  switch $digit {
    0 {turnoff $where {g}}
    1 {turnoff $where {a d e f g}}
    2 {turnoff $where {f c}}
    3 {turnoff $where {e f}}
    4 {turnoff $where {a e d}}
    5 {turnoff $where {b e}}
    6 {turnoff $where {b}}
    7 {turnoff $where {d e f g}}
    8 {turnoff $where {}}
    9 {turnoff $where {e}}
  ____________________________________________________________________
  }
}
```

(d) Mit den folgenden Aufrufen wird eine Tafel mit nur zwei weißen Ziffern (hier 1 und 8) aufgerufen. Dieses Modell „Dr. Markus Merck“ mit schwächerer Leistung bieten wir deutlich günstiger an.

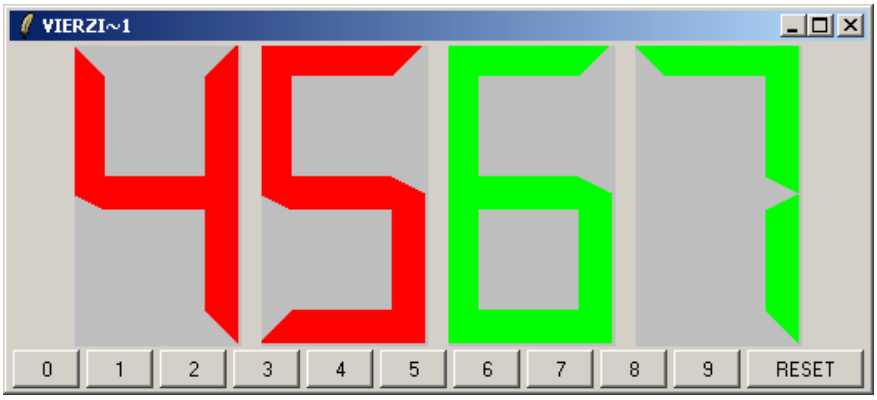
Erweitern Sie dieses Modell zum Luxusmodell „Pierluigi Collina“ mit vier Ziffern. Mit den Modifikationen von Teil (a) sind die ersten zwei Ziffern rot, die restlichen zwei grün. Wechseln Sie den Spieler mit der „18“ aus und die „21“ ein!

```
frame .tafel
pack .tafel
set ziffern(1) [createdigit .tafel 1]
pack $ziffern(1) -side left -padx 4
set ziffern(2) [createdigit .tafel 2]
pack $ziffern(2) -side left -padx 4

_____________________________________________________

_____________________________________________________

_____________________________________________________

_____________________________________________________

showdigit $ziffern(1) 1 ____________________________

showdigit $ziffern(2) 8 ____________________________

_____________________________________________________

_____________________________________________________
```

(e) Jetzt haben wir unten noch eine horizontale Reihe von Knöpfen mit der Beschriftung 0 - 9 auf der Tafel eingebaut. Man gibt mit 4-mal Drücken alle vier Ziffern ein. Gesteuert wird dies mit der globalen Variablen **`nextdigit`**, die festhält, welche Ziffer (1 - 4) als nächstes modifiziert wird. Dies ist allerdings kein so guter Programmierstil.

Ergänzen Sie die zwei fehlenden Einträge in **`setdigit`**. Fügen Sie rechts noch einen Reset-Knopf hinzu. Drückt man ihn, gehen alle Ziffern auf blank und **`nextdigit`** wird auf 1 gesetzt.

```
proc setdigit {what} {
  global ziffern
  global nextdigit
  if {$nextdigit ==1 || $nextdigit ==2} {
    showdigit $ziffern($nextdigit) $what _________________} else {
    showdigit $ziffern($nextdigit) $what _________________}
  incr nextdigit
  if {$nextdigit>4} {set nextdigit 1}
}
frame .tafel
  ... ;# hier nichts einsetzen

frame .k
pack .k
for {set i 0} {$i <=9} {incr i} {
  button .k.b$i -text $i -width 5 -command "setdigit $i"
  pack .k.b$i -side left -padx 2
}
button .k.reset _________________________________ -width 10 -command {

_____________________________________________________________________

_____________________________________________________________________

_____________________________________________________________________

_____________________________________________________________________
}
pack .k.reset -side left -padx 2
set nextdigit 1
```



Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

MUSTERLÖSUNG

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Bei Ankreuzaufgaben können mehrere Antworten richtig sein. Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1a	4	
1b	6	
1c	2	
2a	3	
2b	2	
3	4	
4a	2	
4b	2	
4c	4	
4d	6	
4e	7	
Summe	42	

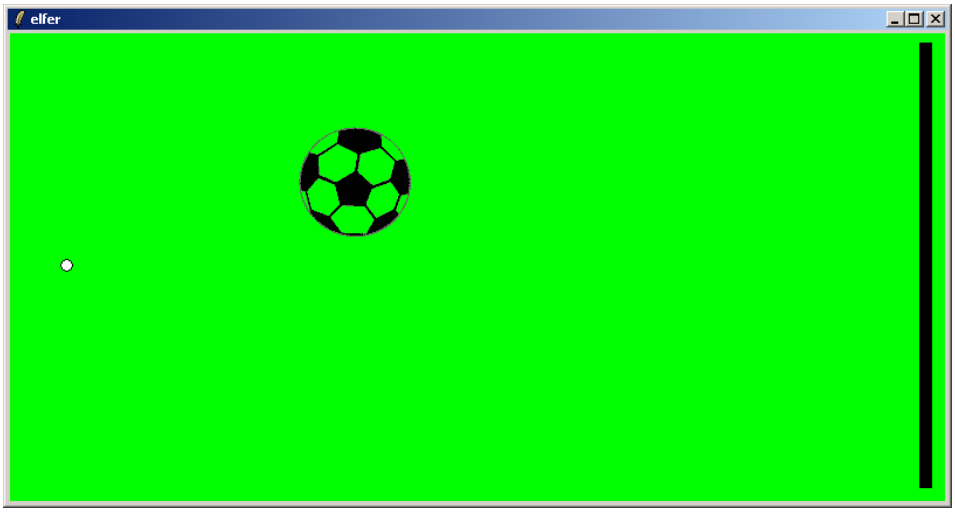
Aufgabe 1:

Wir sind immer noch beim Fußball. Wir wollen das Elfmeterschießen üben. In dem Programm unten zieht man den Ball mit gedrückter linker Maustaste in Richtung Tor (der schwarze Balken rechts). Wenn man die Maustaste losläßt, fliegt der Ball selbstständig weiter.

Das Programm verwendet das Widget-Kommando **`coords`**, das ähnlich wie das Kommando **`itemconfigure`** für Items auf der Leinwand die Koordinaten ändert.

pathName **coords** *tagOrId* ?*x0 y0* ...?

(a) Ergänzen Sie die fehlenden Stellen im Programmtext unten!




```
proc fliege {worauf was x y d} {
  $worauf coords $was $x $y
after 20 "fliege $worauf $was [expr $x + 10] \
        [expr int($y + 10 * $d)] $d"
}

canvas .c -width 800 -height 400 -bg green
pack .c
image create photo ball -file "C:/ball.gif"
.c create oval 45 195 55 205 -fill white -tag elferpunkt
.c create image 50 200 -image ball -tag fussball
.c create rectangle 780 10 790 390 -fill black -tag tor
.c bind fussball <B1-Motion> {
  .c coords fussball %x %y}
.c bind fussball <ButtonRelease-1> {
  set m [expr double(%y - 200) / (%x - 50)]
  fliege .c fussball %x %y $m}
```

(b) Im Programm oben kann man den Ball mit der Maus bis ins Tor ziehen. Das sollte nicht sein!

Bauen Sie eine Abfrage ein, die den Schuß mit einer Meldung „Ungültig“ (als text item auf dem canvas) abbricht, wenn die X-Koordinate der Maus zum Zeitpunkt des Loslassens der Taste schon über einem Viertel der Leinwandbreite liegt.

```
.c bind fussball <ButtonRelease-1> {
  if {%x > [expr [.c cget -width] / 4]} {
    .c create text 400 200 -text "Ungültig!"
  } else {
    set m [expr double(%y - 200) / (%x - 50)]
    fliege .c fussball %x %y $m
  }
}
```

(c) Vor das Tor stellen wir einen „Torwart“, dargestellt durch einen roten Balken, z.B. 10 Pixel breit, 50 Pixel hoch, 10 Pixel Abstand zum Tor. Legen Sie den Torwart an und geben Sie ihm den Tag **`olli`**.[1]

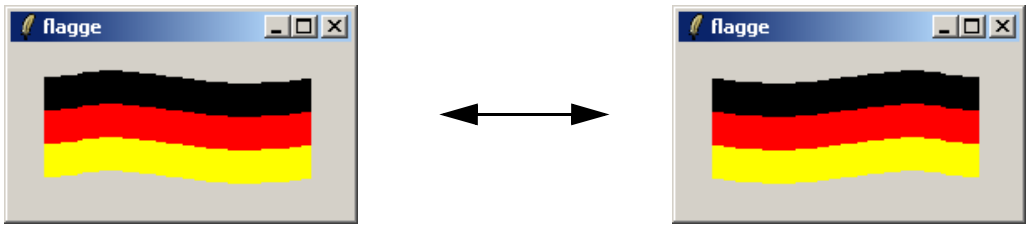
```
.c create rectangle 760 200 770 250 -fill red -tag olli
```

1. Das Animieren des Torwarts (rauf und runterbewegen) verschieben wir auf die Klausur im Sommer.

Aufgabe 2:

Kein Länderspiel ohne Schwenken der Fahne. Die folgende Fahne haben wir uns aus Polygonen selbst gestrickt.

(a)Ergänzen Sie die fehlenden Stellen!




```
proc hin {} {
  .c coords schwarz 20 20 20 20 60 15 100 20 140 25 180 20 \
    180 20 180 40 180 40 140 45 100 40 60 35 20 40 20 40
  .c coords rot 20 40 20 40 60 35 100 40 140 45 180 40 \
    180 40 180 60 180 60 140 65 100 60 60 55 20 60 20 60
  .c coords gold 20 60 20 60 60 55 100 60 140 65 180 60 \
    180 60 180 80 180 80 140 85 100 80 60 75 20 80 20 80
  after [expr int(rand() * 600 + 100)] her
}

proc her {} {
  .c coords schwarz 20 20 20 20 60 25 100 20 140 15 180 20 \
    180 20 180 40 180 40 140 35 100 40 60 45 20 40 20 40
  .c coords rot 20 40 20 40 60 45 100 40 140 35 180 40 \
    180 40 180 60 180 60 140 55 100 60 60 65 20 60 20 60
  .c coords gold 20 60 20 60 60 65 100 60 140 55 180 60 \
    180 60 180 80 180 80 140 75 100 80 60 85 20 80 20 80
  after [expr int(rand() * 600 + 100)] hin
}

canvas .c -width 200 -height 100
pack .c
.c create polygon 20 20 180 20 180 40 20 40 \
  -tag schwarz -fill black -smooth 1
.c create polygon 20 40 180 40 180 60 20 60 \
  -tag rot -fill red -smooth 1
.c create polygon 20 60 180 60 180 80 20 80 \
  -tag gold -fill yellow -smooth 1
```

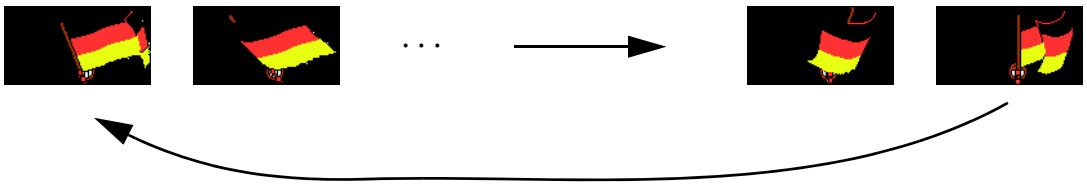
hin; #oder her

(b) In welchem Intervall (Zeitabstand) wird zwischen den beiden Bildern zufällig gewechselt?

Minimum 100 ms, Maximum 700 ms (genauer 699 ms) Hinweis: Im Skript von 2006 wird fälschlicherweise noch behauptet, rand() liefere Zahlen von 0.0 bis 1.0, tatsächlich liefert rand() Zufallszahlen im offenen Intervall (0, 1). berichtigt 7/2010

Aufgabe 3:

Im Internet findet man viele GIF-Bilderfolgen für schwenkende Fahnen, z.B. die folgende.



Ergänzen Sie die Lücken im Programm unten!

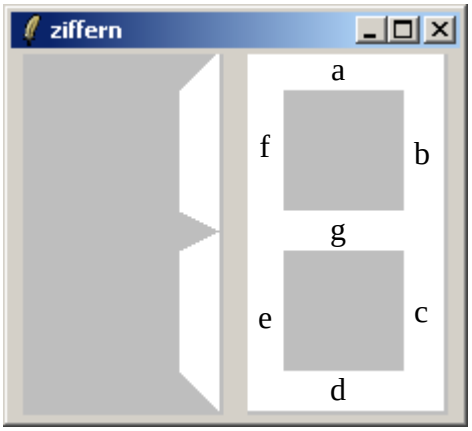
```
proc changeflag {j} {
   .c itemconfigure fahne -image f$j
   if {$j==7} {set j 0} else {incr j}
   after 50 "changeflag $j"
}

canvas .c -width 200 -height 100
pack .c
for {set i 0} {$i<8} {incr i} {
   image create photo f$i -file "fl-de07_frame_00$i.gif"
}
.c create image 100 50 -image f0 -tag fahne
changeflag 0
```

Aufgabe 4:

Bei internationalen Begegnungen im Fußball gibt es einen 4. Offiziellen, der den Schiedsrichter unterstützt. Bei Auswechslungen hält er eine elektronische Tafel hoch, auf der in rot die Rückennummer des auszuwechselnden Spielers und in grün die Nummer des einzuwechselnden steht. Dafür braucht man 4 Ziffern. Mit der Tafel wird auch z.B. die Nachspielzeit angezeigt.

Im Internet bieten Sportausstatter die Tafeln für schlappe 1300,- EUR an. Das ist unsere Chance als Tcl/Tk-Programmierer.

Unten zeigen wir eine selbstgebaute Tafel mit zwei Ziffern. Jede Ziffer ist eine Leinwand (canvas widget), die Ziffern sind aus Segmenten (Polygonen) in der Art einer LCD-Anzeige zusammengesetzt. Die Segmente sind von a - g durchnumeriert.



Hier ist die Prozedur zum Erzeugen einer Leinwand mit allen 7 Segmenten. Über unterschiedliche Werte von **`id`** können wir mehrere Leinwände (damit mehrere Ziffern nebeneinander) erzeugen.

```
proc createdigit {win id} {
  canvas $win.$id -width 100 -height 180 -bg gray

  $win.$id create polygon 0 0 100 0 \
    80 20 20 20 -fill white -tags a

  $win.$id create polygon 100 0 100 90 \
    80 80 80 20 -fill white -tags b

  ...

  $win.$id create polygon 20 100 \
    0 90 20 80 80 80 100 90 80 100\
    -fill white -tags g

  return $win.$id
}
```

(a) Es folgt die Prozedur, um alle Segmente einzuschalten (**`-fill white`**). Fügen Sie eine zweiten Parameter **`farbe`** hinzu und ändern Sie den Prozedurrumpf, um beliebig gefärbte Segmente zu erzeugen, z.B. rote und grüne.

```
proc turnallon {c farbe} {
  foreach item {a b c d e f g} {
    $c itemconfigure $item -fill white $farbe
  }
}
```

(b) Mit der Prozedur **`turnoff`** kann man eine Liste von Segmenten (**`items`**) abschalten. Ergänzen Sie die fehlende Stelle.

```
proc turnoff {c items} {
  foreach item $items {
    $c itemconfigure $item -fill gray
  }
}
```

(c) Ziffern werden angezeigt, indem Segmente ausgeschaltet werden. Ergänzen Sie wieder die Färbung der Segmente und fügen Sie in **`switch`** eine Alternative ein, bei der **`" "`** (blank) für Parameter **`digit`** alle Segmente ausschaltet.

```
proc showdigit {where digit farbe} {
  turnallon $where $farbe
  switch $digit {
    0 {turnoff $where {g}}
    1 {turnoff $where {a d e f g}}
    2 {turnoff $where {f c}}
    3 {turnoff $where {e f}}
    4 {turnoff $where {a e d}}
    5 {turnoff $where {b e}}
    6 {turnoff $where {b}}
    7 {turnoff $where {d e f g}}
    8 {turnoff $where {}}
    9 {turnoff $where {e}}
    " " {turnoff $where {a b c d e f g}}
  }
}
```

(d) Mit den folgenden Aufrufen wird eine Tafel mit nur zwei weißen Ziffern (hier 1 und 8) aufgerufen. Dieses Modell „Dr. Markus Merck“ mit schwächerer Leistung bieten wir deutlich günstiger an.

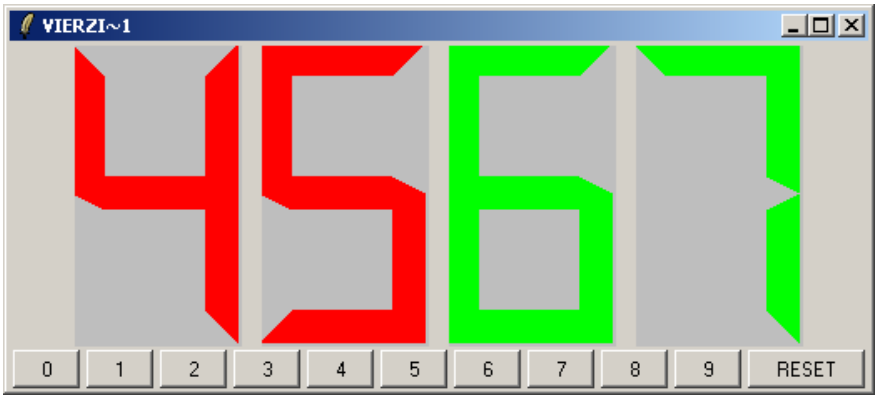
Erweitern Sie dieses Modell zum Luxusmodell „Pierluigi Collina“ mit vier Ziffern. Mit den Modifikationen von Teil (a) sind die ersten zwei Ziffern rot, die restlichen zwei grün. Wechseln Sie den Spieler mit der „18“ aus und die „21“ ein!

```
frame .tafel
pack .tafel
set ziffern(1) [createdigit .tafel 1]
pack $ziffern(1) -side left -padx 4
set ziffern(2) [createdigit .tafel 2]
pack $ziffern(2) -side left -padx 4
set ziffern(3) [createdigit .tafel 3]
pack $ziffern(3) -side left -padx 4
set ziffern(4) [createdigit .tafel 4]
pack $ziffern(4) -side left -padx 4

showdigit $ziffern(1) 1 "red"
showdigit $ziffern(2) 8 "red"
showdigit $ziffern(3) 2 "green"
showdigit $ziffern(4) 1 "green"
```

(e) Jetzt haben wir unten noch eine horizontale Reihe von Knöpfen mit der Beschriftung 0 - 9 auf der Tafel eingebaut. Man gibt mit 4-mal Drücken alle vier Ziffern ein. Gesteuert wird dies mit der globalen Variablen **nextdigit**, die festhält, welche Ziffer (1 - 4) als nächstes modifiziert wird. Dies ist allerdings kein so guter Programmierstil.

Ergänzen Sie die zwei fehlenden Einträge in **setdigit**. Fügen Sie rechts noch einen Reset-Knopf hinzu. Drückt man ihn, gehen alle Ziffern auf blank und **nextdigit** wird auf 1 gesetzt.

```
proc setdigit {what} {
  global ziffern
  global nextdigit
  if {$nextdigit ==1 || $nextdigit ==2} {
    showdigit $ziffern($nextdigit) $what "red"} else {
    showdigit $ziffern($nextdigit) $what "green"}
  incr nextdigit
  if {$nextdigit>4} {set nextdigit 1}
}
frame .tafel
... ;# hier nichts einsetzen

frame .k
pack .k
for {set i 0} {$i <=9} {incr i} {
  button .k.b$i -text $i -width 5 -command "setdigit $i"
  pack .k.b$i -side left -padx 2
}
button .k.reset -text "RESET" -width 10 -command {

  set nextdigit 1 ;# könnte man auch weglassen
  setdigit " "; setdigit " "; setdigit " "; setdigit " ";
  set nextdigit 1


}
pack .k.reset -side left -padx 2
set nextdigit 1
```



Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: Vorname:

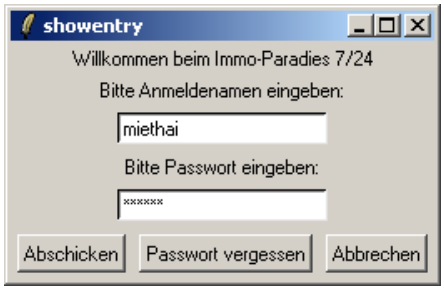
Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1	6	
2	8	
3	8 + 4	
4	7	
5	3 + 6	
6	6	
Summe	48	

Aufgabe 1:

In dieser Klausur geht es um das Anmieten, Vermieten, Kaufen und Verkaufen von Wohnungen und Häusern. Im folgenden Portal müssen Sie sich als Vermieter oder Verkäufer zuerst anmelden. Das Programm enthält einige Fehler, gegebenenfalls mehrfach. Markieren Sie diese!




```
frame .r
pack .r
label .r.welcome -text "Willkommen beim Immo-Paradies 7/24"
pack .r.welcome
label .r.prompt1 -text "Bitte Anmeldenamen eingeben:"
pack .r.prompt1
entry .r.ein1 -textvariable login -background white -width 20
pack .r.ein1 -padx 4 -pady 4
label .r.prompt2 -text "Bitte Passwort eingeben:"
pack .r.prompt2
entry .r.ein2 -textvariable pw -background white -width 20 -show "*"
pack .r.ein2 -padx 4 -pady 4
frame .r.buttons
pack .r.buttons

button .r.buttons.send -text "Abschicken" -command {
   exit}
pack .r.buttons.send -side east -padx 4 -pady 4
button .r.buttons.forgot -text Passwort vergessen -invoke {
   exit}
pack .r.buttons.forgot -side east -padx 4 -pady 4
button .r.buttons.cancel -text "Abbrechen" -invoke {
   exit}
pack .r.buttons.cancel -side left -padx 4 -pady 4

focus .r.eingabe1
```

Aufgabe 2:

Das unten gezeigte Skript erzeugt eine Eingabemaske für die Suche nach geeigneten Wohnungen, gesteuert durch Angaben für Anzahl der Zimmer, Größe der Wohnung, maximale Miete.

Zeichnen Sie in das Bild auf Seite 4 das anfängliche Aussehen ein. Die relative Anordnung der Eingabefelder und Knöpfe muß klar erkennbar sein.

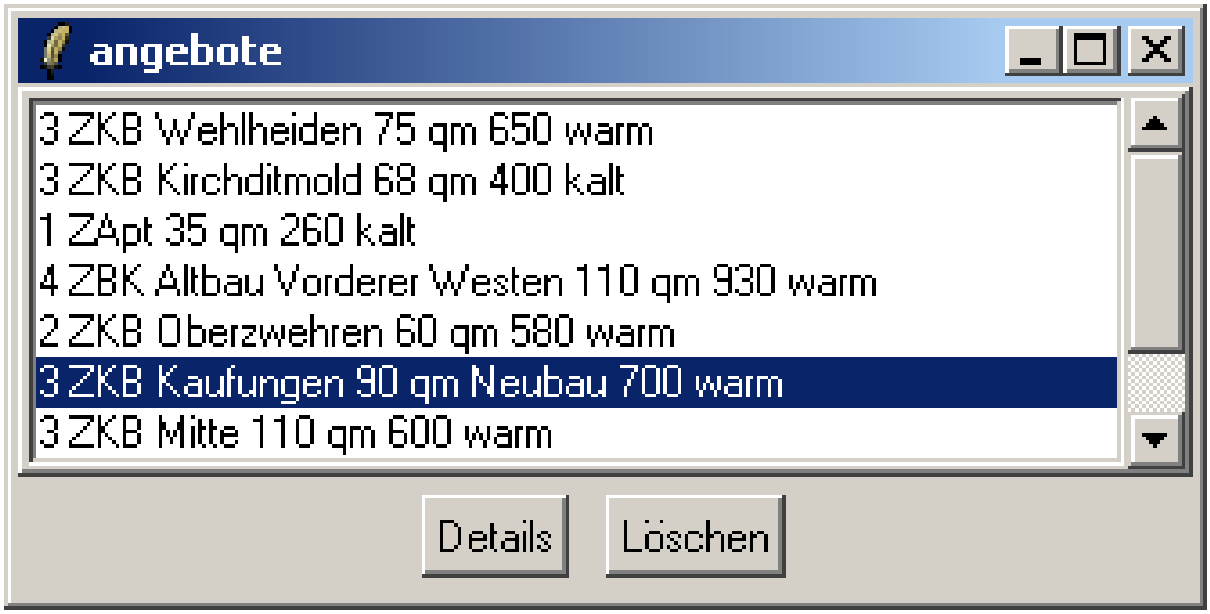
```
proc suche {zmin zmax qmin qmax kmax} {exit}

label .banner -text "Willkommen zur Wohnungssuche!"
grid .banner -columnspan 5 -pady 10

label .kosten -text "Maximale Miete"
entry .kostenmax -textvariable kmax -background white -width 20
grid x x x .kosten .kostenmax -pady 10
grid configure .kosten -sticky w
grid configure .kostenmax -sticky e

label .zimmer -text "Anzahl Zimmer"
label .zminlabel -text "min"
entry .zimmermin -textvariable zmin -background white -width 20
label .zmaxlabel -text "max"
entry .zimmermax -textvariable zmax -background white -width 20
grid .zimmer .zminlabel .zimmermin .zmaxlabel .zimmermax -pady 10
grid configure .zimmer -sticky w
grid configure .zminlabel .zmaxlabel -sticky e

label .qmeter -text "Anzahl Quadratmeter"
label .qminlabel -text "min"
entry .qmetermin -textvariable qmin -background white -width 20
label .qmaxlabel -text "max"
entry .qmetermax -textvariable qmax -background white -width 20
grid .qmeter .qminlabel .qmetermin .qmaxlabel .qmetermax  -pady 10
grid configure .qmeter -sticky w
grid configure .qminlabel .qmaxlabel -sticky e

button .start -text "Suche starten" -command {
   suche $zmin $zmax $qmin $qmax $kmax }
grid .start -columnspan 5 -pady 10
focus .kostenmax
```


Willkommen zur Wohnungssuche!
ssuche

Aufgabe 3:

Betrachten Sie die Liste (listbox) der Wohnungsangebote, die sich im übrigen den Änderungen der Fenstergröße anpaßt. Der Löschen-Knopf entfernt den selektierten Eintrag aus der Liste.

(a) Füllen Sie die Lücken!



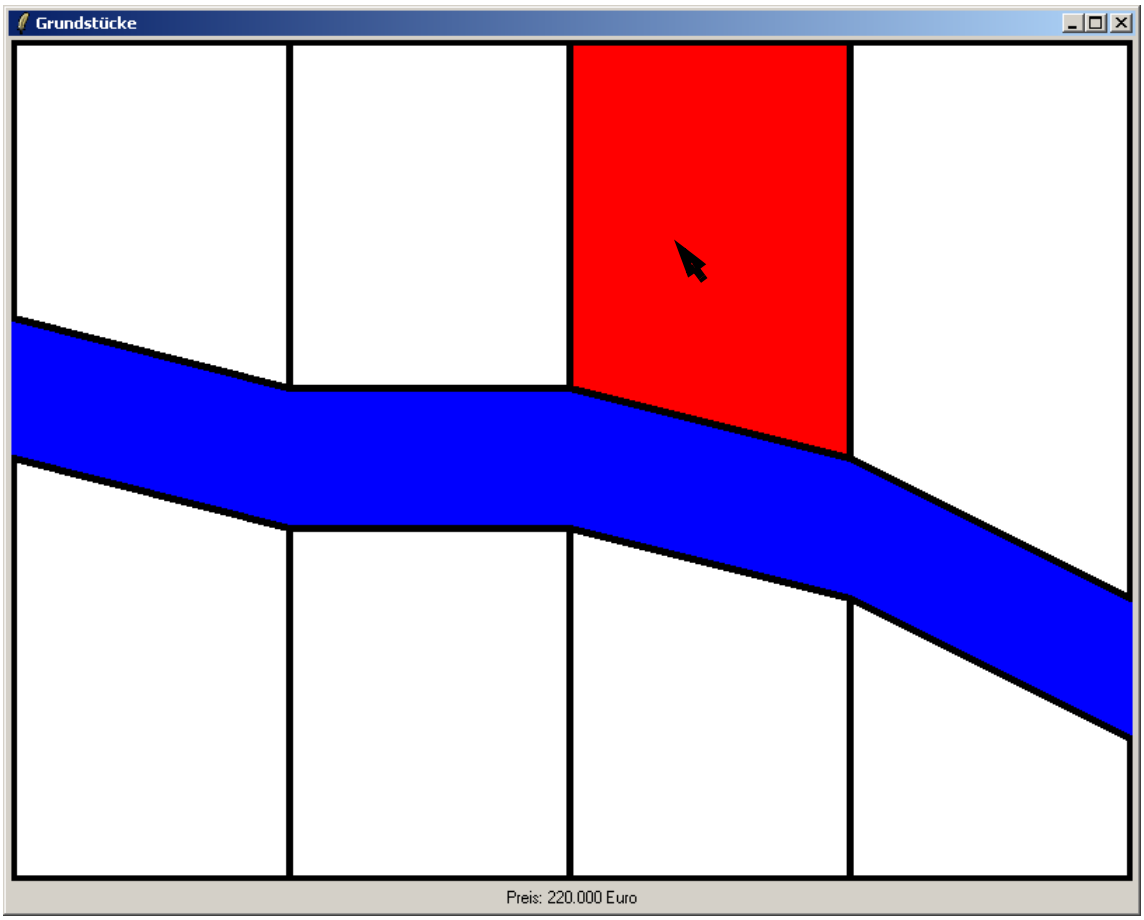

```
frame .f -relief raised -borderwidth 2
pack .f -expand yes -fill both
listbox .f.list -yscroll ".f.scroll set" -setgrid 1 -height 7 -width 50
scrollbar .f.scroll -command ".f.list yview"
pack .f.scroll -side right -fill _____________________________
pack .f.list -side left -expand ___________________ -fill both

frame .buttons
pack .buttons
button .buttons.details -text "Details" -command {
    einzelheiten _____________ }
pack .buttons.details -side left -padx 5 -pady 5
button .buttons.loeschen -text "Löschen" -command {
    loeschen ____________________________ }
pack .buttons.loeschen -side right -padx 5 -pady 5

.f.list insert 0 \
  "3 ZKB Wehlheiden 75 qm 650 warm"  \
  "3 ZKB Kirchditmold 68 qm 400 kalt"  "1 ZApt 35 qm 260 kalt"  \
  "4 ZBK Altbau Vorderer Westen 110 qm 930 warm"  \
```

```
  "2 ZKB Oberzwehren 60 qm 580 warm"  \
  "3 ZKB Kaufungen 90 qm Neubau 700 warm"  \
  "3 ZKB Mitte 110 qm 600 warm"  \
  "3 ZBK Niederzwehren 75 qm 380 warm"  \
  "2 ZKB Wesertor 50 qm 240 warm"


proc einzelheiten { listbox } {
  if { [$listbox curselection] != "" } {
    toplevel .t
    wm title .t "Details"
    message _____________________________ -width 400 -justify center \
      -text "Zu dem Angebot\n\
        [$listbox get [$listbox curselection]]\n\
        liegen keine Details vor."
    pack .t.mess
    button _____________________________ -text "OK" -command { destroy .t }
    pack .t.but -padx 5 -pady 5
  }
}

proc loeschen { listbox } {
  if { [$listbox curselection] != "" } {
    $listbox _____________________________ [ _____________________________ ]
  }
}
```

(b) Beschreiben Sie kurz aber präzise das sichtbare Verhalten des Programms, wenn in der oben abgebildeten Situation der Details-Knopf gedrückt wird.

Aufgabe 4:

Wir haben acht Flußgrundstücke im Angebot. Fährt man mit der Maus in das zugehörige Polygon, färbt es sich rot und unten wird der Preis angezeigt. Ergänzen Sie die Lücken!




```
canvas .c -width 800 -height 600 -background blue
pack .c
label .lab
pack .lab
set grundstuecke {
  { {   3   3 200   3 200 250   3 200 } 170 }
  { { 200   3 400   3 400 250 200 250 } 200 }
  { { 400   3 600   3 600 300 400 250 } 220 }
  { { 600   3 800   3 800 400 600 300 } 270 }
  { {   3 300 200 350 200 600   3 600 } 210 }
  { { 200 350 400 350 400 600 200 600 } 200 }
  { { 400 350 600 400 600 600 400 600 } 150 }
  { { 600 400 800 500 800 600 600 600 }  80 }
}
```

```
foreach grundstueck $grundstuecke {
  set koordinaten [lindex $grundstueck 0]
  set preis [lindex $grundstueck 1]
  set tag [.c create _____________________ $koordinaten -fill white
              -outline black -width 5]

  .c bind ___________________ <Enter> \
    ".lab configure -text \"Preis: ${preis}.000 Euro\"
     .c _____________________________ $tag -fill _________________"

  .c bind ___________________ <Leave> \
    ".lab configure -text \"\"
     .c _____________________________ $tag -fill _________________"

}
```

Aufgabe 5:

Um den möglichen Käufern der Grundstücke klarzumachen, dass das gewünschte Grundstück vielleicht schon weg ist, setzen wir ein Verkauft-Schild zufällig auf ein Grundstück. Hier ist die Prozedur **`verkauft`**, die am Ende des obigen Programms einmal aufgerufen wird.

```
proc verkauft { } {

  .c delete schild

  set zufall [expr int(rand() * 8 + 1)]
  if { $zufall <= 4 } {
    set y 70
  } else {
    set y 470
  }
  set x [expr ($zufall % 4) * 200 + 50]

  .c create rectangle [expr $x+45] [expr $y] [expr $x+55] \
      [expr $y+120] -tag schild -fill brown
  .c create rectangle [expr $x] [expr $y+10] [expr $x+100] \
      [expr $y+50] -tag schild -fill orange
  .c create text [expr $x+50] [expr $y+30] -text "VERKAUFT!" \
      -tag schild

  after 1000 verkauft

}
```

(a) Zeichnen Sie in das Fenster aus Aufgabe 4 **ein** Schild ein. Es kommt nicht auf Schönheit an, aber Position und Größenverhältnisse sollten erkennbar sein.

(b) Statt das eigentliche Schild immer in der Farbe **`orange`** zu malen, wählen wir über eine Variable **`zufall2`** eine der sechs Farben **`green`**, **`yellow`**, **`pink`**, **`orange`**, **`gray`**, **`cyan`** aus. Ändern Sie das Programm oben, indem Sie einen Liste **`farben`** entsprechend belegen, den zweiten Zufallswert (**`zufall2`**) erzeugen und die **`fill`**-Option richtig setzen.

Aufgabe 6:

Ergänzen Sie im folgenden Programm die Erzeugung und das Anzeigen des gezeigten Knopfs zum Eintreten. Dieser soll die Prozedur **`weiter`** aufrufen, wenn der Haken beim Checkbutton gesetzt ist, sonst erzeugt er nur einen Klingelton (Aufruf des Kommandos **`bell)`**.


```
set img [image create photo -file haus.gif]
label .bild -image $img
pack .bild

label .willkommen -text "Willkommen im Immo-Paradies!" -font\
    "Arial 20 bold"
pack .willkommen

checkbutton .agb -variable agbOK -text \
    "Ich akzeptiere die Allgemeinen Geschäftsbedingungen."
pack .agb

proc weiter { } {exit}
```

Ende der Klausur

Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

MUSTERLÖSUNG

Nachname: Vorname:

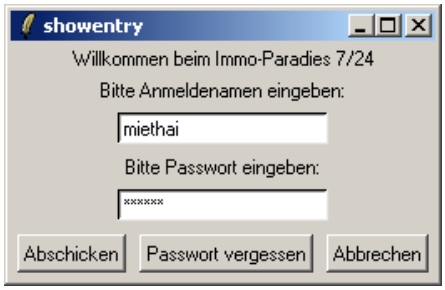
Mat.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1	6	
2	8	
3	8 + 4	
4	7	
5	3 + 6	
6	6	
Summe	48	

Aufgabe 1:

In dieser Klausur geht es um das Anmieten, Vermieten, Kaufen und Verkaufen von Wohnungen und Häusern. Im folgenden Portal müssen Sie sich als Vermieter oder Verkäufer zuerst anmelden. Das Programm enthält einige Fehler, gegebenenfalls mehrfach. Markieren Sie diese!



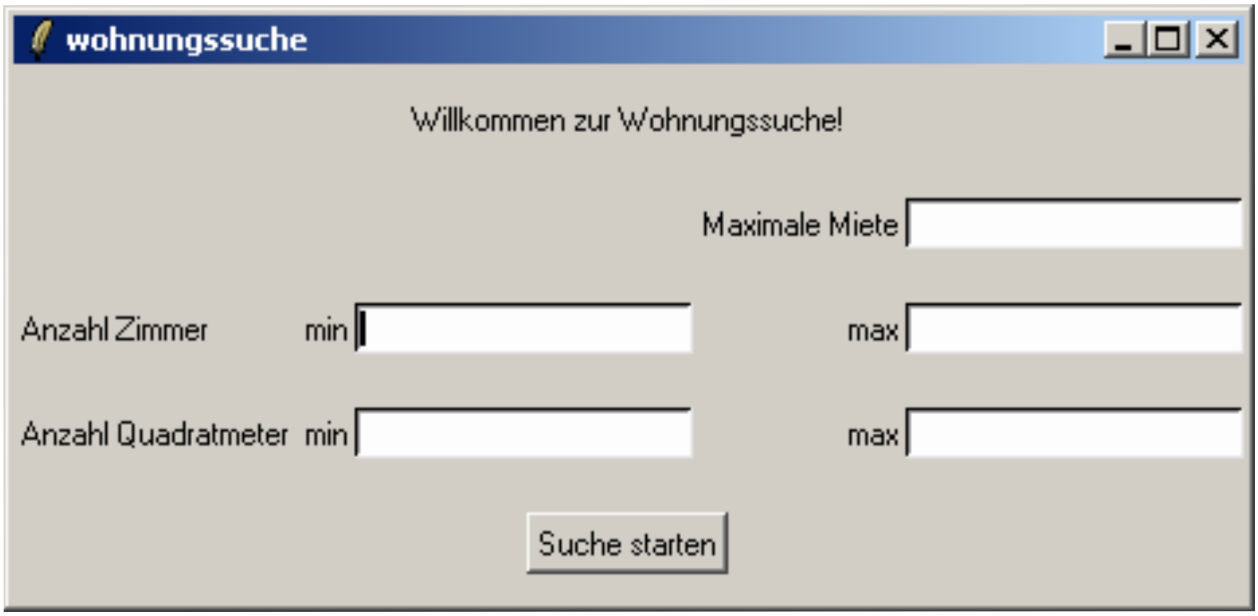

```
frame .r
pack .r
label .r.welcome -text "Willkommen beim Immo-Paradies 7/24"
pack .r.welcome
label .r.prompt1 -text "Bitte Anmeldenamen eingeben:"
pack .r.prompt1
entry .r.ein1 -textvariable login -background white -width 20
pack .r.ein1 -padx 4 -pady 4
label .r.prompt2 -text "Bitte Passwort eingeben:"
pack .r.prompt2
entry .r.ein2 -textvariable pw -background white -width 20 -show "*"
pack .r.ein2 -padx 4 -pady 4
frame .r.buttons
pack .r.buttons

button .r.buttons.send -text "Abschicken" -command {
   exit}
pack .r.buttons.send -side east -padx 4 -pady 4
button .r.buttons.forgot -text Passwort vergessen -invoke {
   exit}
pack .r.buttons.forgot -side east -padx 4 -pady 4
button .r.buttons.cancel -text "Abbrechen" -invoke {
   exit}
pack .r.buttons.cancel -side left -padx 4 -pady 4

focus .r.eingabe1
```

Aufgabe 2:

Das unten gezeigte Skript erzeugt eine Eingabemaske für die Suche nach geeigneten Wohnungen, gesteuert durch Angaben für Anzahl der Zimmer, Größe der Wohnung, maximale Miete.

Zeichnen Sie in das Bild auf Seite 4 das anfängliche Aussehen ein. Die relative Anordnung der Eingabefelder und Knöpfe muß klar erkennbar sein.

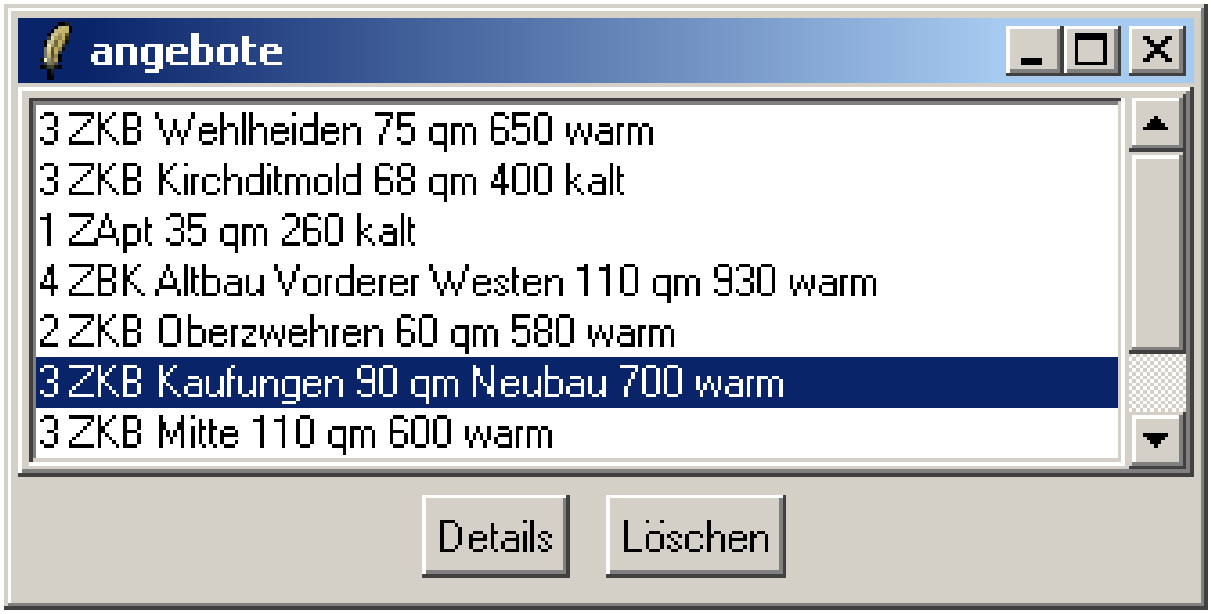
```
proc suche {zmin zmax qmin qmax kmax} {exit}

label .banner -text "Willkommen zur Wohnungssuche!"
grid .banner -columnspan 5 -pady 10

label .kosten -text "Maximale Miete"
entry .kostenmax -textvariable kmax -background white -width 20
grid x x x .kosten .kostenmax -pady 10
grid configure .kosten -sticky w
grid configure .kostenmax -sticky e

label .zimmer -text "Anzahl Zimmer"
label .zminlabel -text "min"
entry .zimmermin -textvariable zmin -background white -width 20
label .zmaxlabel -text "max"
entry .zimmermax -textvariable zmax -background white -width 20
grid .zimmer .zminlabel .zimmermin .zmaxlabel .zimmermax -pady 10
grid configure .zimmer -sticky w
grid configure .zminlabel .zmaxlabel -sticky e

label .qmeter -text "Anzahl Quadratmeter"
label .qminlabel -text "min"
entry .qmetermin -textvariable qmin -background white -width 20
label .qmaxlabel -text "max"
entry .qmetermax -textvariable qmax -background white -width 20
grid .qmeter .qminlabel .qmetermin .qmaxlabel .qmetermax  -pady 10
grid configure .qmeter -sticky w
grid configure .qminlabel .qmaxlabel -sticky e

button .start -text "Suche starten" -command {
   suche $zmin $zmax $qmin $qmax $kmax }
grid .start -columnspan 5 -pady 10
focus .kostenmax
```


wohnungssuche
Willkommen zur Wohnungssuche!
Maximale Miete
Anzahl Zimmer
min
max
Anzahl Quadratmeter
min
max
Suche starten

Aufgabe 3:

Betrachten Sie die Liste (listbox) der Wohnungsangebote, die sich im übrigen den Änderungen der Fenstergröße anpaßt. Der Löschen-Knopf entfernt den selektierten Eintrag aus der Liste.

(a) Füllen Sie die Lücken!




```
frame .f -relief raised -borderwidth 2
pack .f -expand yes -fill both
listbox .f.list -yscroll ".f.scroll set" -setgrid 1 -height 7 -width 50
scrollbar .f.scroll -command ".f.list yview"
pack .f.scroll -side right -fill y ;# oder auch -fill both
pack .f.list -side left -expand yes -fill both ;# auch -expand true

frame .buttons
pack .buttons
button .buttons.details -text "Details" -command {
    einzelheiten .f.list }
pack .buttons.details -side left -padx 5 -pady 5
button .buttons.loeschen -text "Löschen" -command {
    loeschen .f.list }
pack .buttons.loeschen -side right -padx 5 -pady 5

.f.list insert 0 \
  "3 ZKB Wehlheiden 75 qm 650 warm"  \
  "3 ZKB Kirchditmold 68 qm 400 kalt"  "1 ZApt 35 qm 260 kalt"  \
  "4 ZBK Altbau Vorderer Westen 110 qm 930 warm"  \
```

```
  "2 ZKB Oberzwehren 60 qm 580 warm"  \
  "3 ZKB Kaufungen 90 qm Neubau 700 warm"  \
  "3 ZKB Mitte 110 qm 600 warm"  \
  "3 ZBK Niederzwehren 75 qm 380 warm"  \
  "2 ZKB Wesertor 50 qm 240 warm"


proc einzelheiten { listbox } {
  if { [$listbox curselection] != "" } {
    toplevel .t
    wm title .t "Details"
    message .t.mess -width 400 -justify center \
      -text "Zu dem Angebot\n\
        [$listbox get [$listbox curselection]]\n\
        liegen keine Details vor."
    pack .t.mess
    button .t.but -text "OK" -command { destroy .t }
    pack .t.but -padx 5 -pady 5
  }
}

proc loeschen { listbox } {
  if { [$listbox curselection] != "" } {
    $listbox delete [ $listbox curselection ]
  }
}
```

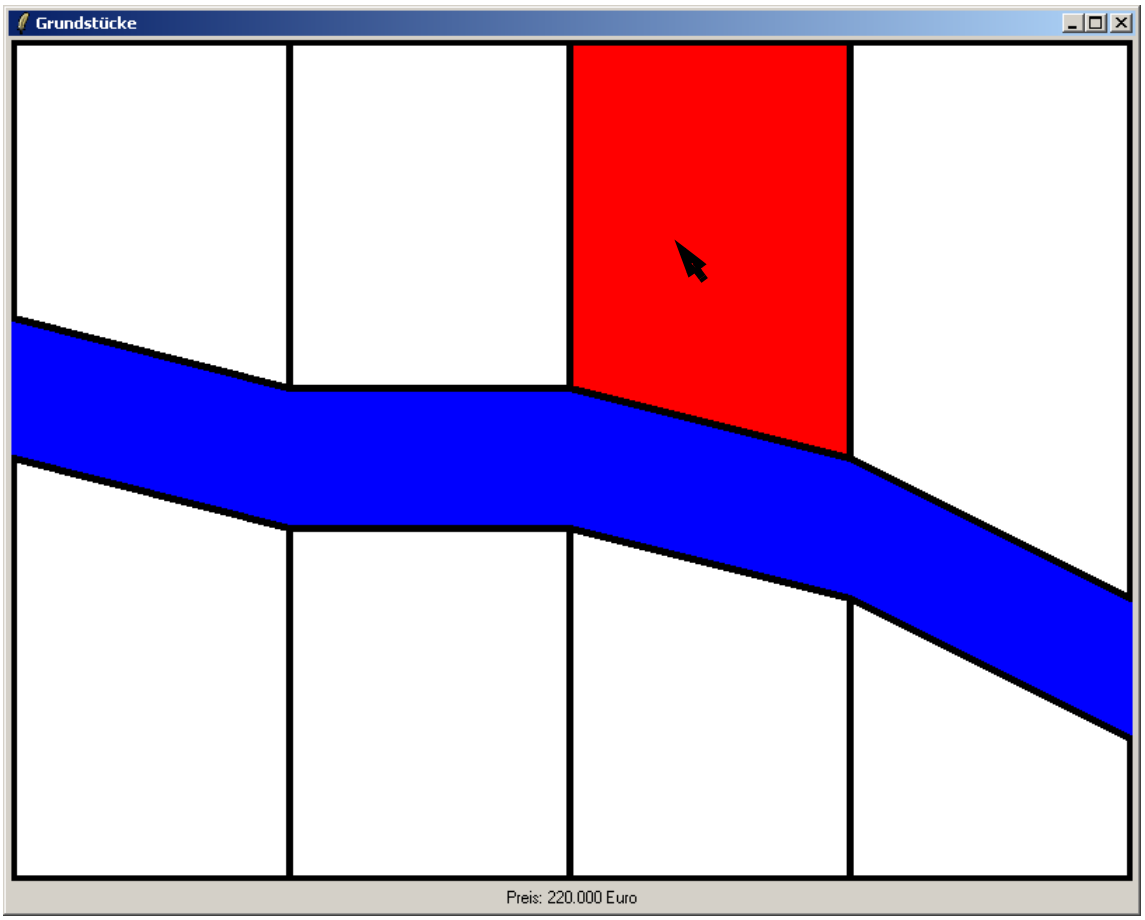
(b) Beschreiben Sie kurz aber präzise das sichtbare Verhalten des Programms, wenn in der oben abgebildeten Situation der Details-Knopf gedrückt wird.

Es erscheint ein neues Toplevel-Fenster mit dem zentrierten Text: Zu dem Angebot 3ZKB Kaufungen ... liegen keine Details vor.

Druck auf den OK-Knopf schließt das Fenster wieder.

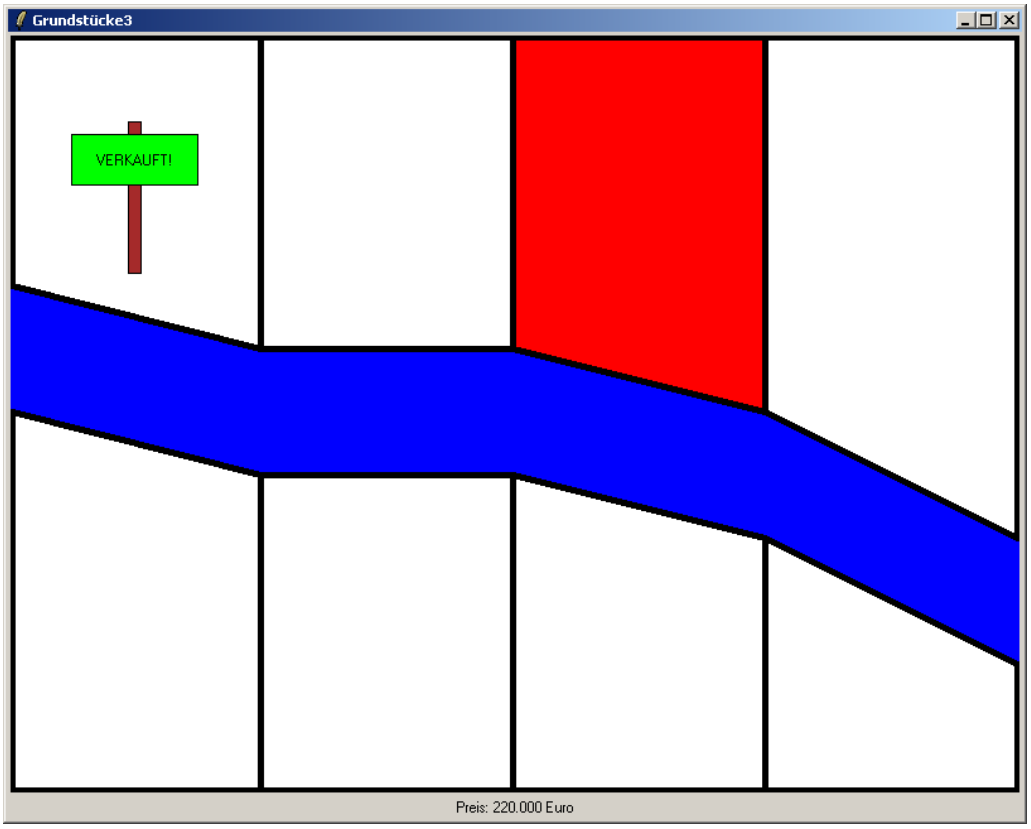
Aufgabe 4:

Wir haben acht Flußgrundstücke im Angebot. Fährt man mit der Maus in das zugehörige Polygon, färbt es sich rot und unten wird der Preis angezeigt. Ergänzen Sie die Lücken!




```
canvas .c -width 800 -height 600 -background blue
pack .c
label .lab
pack .lab
set grundstuecke {
  { {   3   3 200   3 200 250   3 200 } 170 }
  { { 200   3 400   3 400 250 200 250 } 200 }
  { { 400   3 600   3 600 300 400 250 } 220 }
  { { 600   3 800   3 800 400 600 300 } 270 }
  { {   3 300 200 350 200 600   3 600 } 210 }
  { { 200 350 400 350 400 600 200 600 } 200 }
  { { 400 350 600 400 600 600 400 600 } 150 }
  { { 600 400 800 500 800 600 600 600 }  80 }
}
```

```
foreach grundstueck $grundstuecke {
  set koordinaten [lindex $grundstueck 0]
  set preis [lindex $grundstueck 1]
  set tag [.c create polygon $koordinaten -fill white
              -outline black -width 5]

  .c bind $tag <Enter> \
    ".lab configure -text \"Preis: ${preis}.000 Euro\"
     .c itemconfigure $tag -fill red"

  .c bind $tag <Leave> \
    ".lab configure -text \"\"
     .c itemconfigure $tag -fill white"
}
```



Aufgabe 5:

Um den möglichen Käufern der Grundstücke klarzumachen, dass das gewünschte Grundstück vielleicht schon weg ist, setzen wir ein Verkauft-Schild zufällig auf ein Grundstück. Hier ist die Prozedur **verkauft**, die am Ende des obigen Programms einmal aufgerufen wird.

```
proc verkauft { } {

  .c delete schild
                              (1)
  set zufall [expr int(rand() * 8 + 1)]
  if { $zufall <= 4 } {
    set y 70
  } else {
    set y 470
  }
  set x [expr ($zufall % 4) * 200 + 50]
                              (2)
  .c create rectangle [expr $x+45] [expr $y] [expr $x+55] \
      [expr $y+120] -tag schild -fill brown
  .c create rectangle [expr $x] [expr $y+10] [expr $x+100] \
      [expr $y+50] -tag schild -fill orange                    (3)
  .c create text [expr $x+50] [expr $y+30] -text "VERKAUFT!" \
      -tag schild

  after 1000 verkauft

}
```

(a) Zeichnen Sie in das Fenster aus Aufgabe 4 **ein** Schild ein. Es kommt nicht auf Schönheit an, aber Position und Größenverhältnisse sollten erkennbar sein. **Siehe Seite zuvor!**

(b) Statt das eigentliche Schild immer in der Farbe **orange** zu malen, wählen wir über eine Variable **zufall2** eine der sechs Farben **green**, **yellow**, **pink**, **orange**, **gray**, **cyan** aus. Ändern Sie das Programm oben, indem Sie einen Liste **farben** entsprechend belegen, den zweiten Zufallswert (**zufall2**) erzeugen und die **fill**-Option richtig setzen.

(1) set farben {green yellow pink orange gray cyan}

(2) set zufall2 [expr int(rand() * 6)]

statt -fill orange setze bei (3) -fill [lindex $farben $zufall2]

Aufgabe 6:

Ergänzen Sie im folgenden Programm die Erzeugung und das Anzeigen des gezeigten Knopfs zum Eintreten. Dieser soll die Prozedur **`weiter`** aufrufen, wenn der Haken beim Checkbutton gesetzt ist, sonst erzeugt er nur einen Klingelton (Aufruf des Kommandos **`bell)`**.


```
set img [image create photo -file haus.gif]
label .bild -image $img
pack .bild

label .willkommen -text "Willkommen im Immo-Paradies!" -font\
    "Arial 20 bold"
pack .willkommen

checkbutton .agb -variable agbOK -text \
    "Ich akzeptiere die Allgemeinen Geschäftsbedingungen."
pack .agb

proc weiter { } {exit}
```

```
button .eintreten -text "Eintreten" -command {
    if $agbOK weiter else bell}
pack .eintreten
```

Ende der Klausur

Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1	6	
2	6 + 3	
3	6	
4	6 + 2	
5	6	
Summe	35	

Aufgabe 1:

In dieser Klausur geht es um Wahlen. Zuerst bauen wir uns einen vereinfachten Wahlcomputer. Wie man sieht, kann man für fünf Parteien abstimmen. Klickt man einen der Partei-Buttons an, erscheint das rechte obere Fenster. Klickt man darin „Ja“ an, erscheint das rechte untere Fenster. Schreiben Sie auf die nächste Seite die passende Prozedur `waehle`, die dieses Fenster erzeugt. Beim Klicken des OK-Buttons beendet sich das Programm.

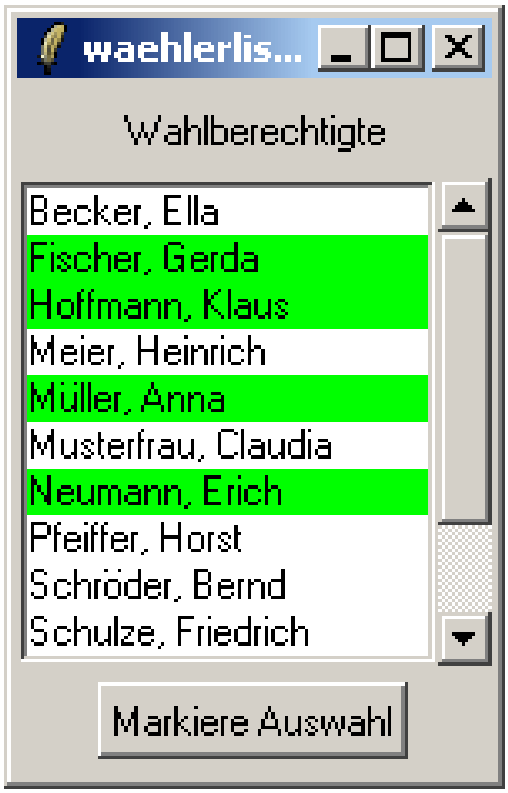

```
set parteien { CDU SPD FDP Grünen Linke }

label .prompt -text "Bitte treffen Sie Ihre Wahl:"
pack .prompt -padx 5 -pady 5

set i 1
foreach partei $parteien {
  button .b$i -text $partei -command "bestaetige $partei"
  pack .b$i -padx 5 -pady 5
  incr i
}

proc bestaetige { partei } {
  toplevel .t
  label .t.label \
    -text "Bitte bestätigen Sie: Möchten Sie für die $partei stimmen?"
  pack .t.label -padx 5 -pady 5
  frame .t.buttons
  pack .t.buttons -padx 5 -pady 5
  button .t.buttons.ja -text "Ja" -command "destroy .t; waehle $partei"
  button .t.buttons.nein -text "Nein" -command "destroy .t"
  pack .t.buttons.ja .t.buttons.nein -side left
}
```

Hier die Prozedur einfügen:

Aufgabe 2:

Wenn man wählen geht, prüft die Wahlkommission, ob man im Wählerverzeichnis eingetragen ist. Hierzu haben wir ein Programm, das eine Liste der vorhandenen Wähler enthält. Darin kann man Namen markieren, die farblich hinterlegt werden, wie gezeigt. Man markiert, indem man mit der Maus den Namen auswählt und dann den „Markiere Auswahl“-Button anklickt.

(a) Füllen Sie die Lücken im Programm!



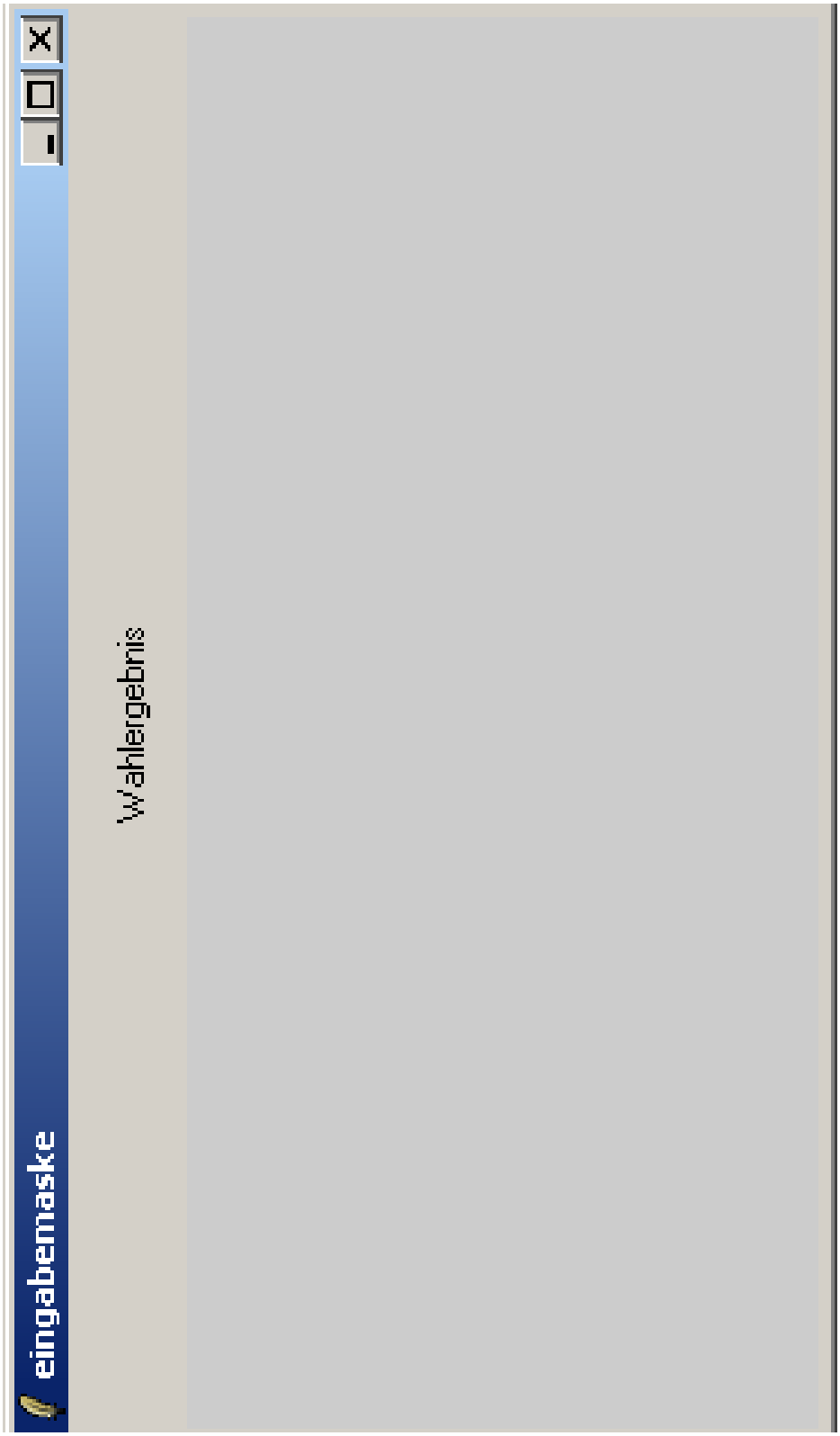

```
label .lb -text "_____________________________________________"
pack .lb -padx 5 -pady 5
frame .fr
pack .fr
listbox .fr.liste -yscroll ".fr.scroll set"
scrollbar .fr.scroll -command ".fr.liste yview"
pack ____________________ -side left -expand yes -fill both
pack ____________________ -side right -fill y

.fr.liste insert 0 \
  "Becker, Ella" "Fischer, Gerda" "Hoffmann, Klaus" \
  "Meier, Heinrich" "Müller, Anna" "Musterfrau, Claudia" \
  "Neumann, Erich" "Pfeiffer, Horst" "Schröder, Bernd" \
  "Schulze, Friedrich" "Wagner, Britta" "Weber, Anton" \
  "Zimmermann, Karin"
```

```
button .bt -text "Markiere Auswahl" -_____________________ {
  if { [.fr.liste curselection] != "" } { _________________ _________________ }
}
pack .bt -padx 5 -pady 5

proc istMarkiert { listbox } {
  if {[$listbox itemcget [$listbox curselection] -background] == "green"} {
    return true
  } else {
    return false
  }
}

proc markiere { listbox } {
  if { [istMarkiert $listbox] } {
    toplevel .tl
    label .tl.lb -text \
      "[$listbox get [$listbox curselection]] ist bereits markiert!"
    pack .tl.lb -padx 5 -pady 5
    button .tl.bt -text "OK" -command "destroy .tl"
    pack .tl.bt -padx 5 -pady 5
  } else {
    $listbox itemconfigure [$listbox curselection] \
      -background green
  }
}
```

(b) Was passiert, wenn man einen bereits markierten Listeneintrag nochmals mit der Maus auswählt und dann den Markier-Button anklickt? Beschreiben Sie die Anzeige und das Verhalten genau!

Aufgabe 3:

Jeder Wahlbezirk meldet die Ergebnisse an den Wahlleiter. Dazu gibt es eine Eingabemaske, die von dem Tcl/Tk-Programm unten erzeugt wird. Zeichnen Sie die Ausgabe möglichst genau ein!



```
label .titel -text "Wahlergebnis"
pack .titel -padx 10 -pady 10

frame .bezirk
pack .bezirk -fill both -padx 10 -pady 10
label .bezirk.label -text "Wahlbezirk"
entry .bezirk.entry -textvariable bezirk -background white
pack .bezirk.entry .bezirk.label -side right

frame .stimmen
pack .stimmen -fill both -padx 10 -pady 10
label .stimmen.abgegebenLabel -text "Abgegebene Stimmen"
entry .stimmen.abgegebenEntry -textvariable StimmenTotal \
    -background white
label .stimmen.ungueltigLabel -text "Ungültige Stimmen"
entry .stimmen.ungueltigEntry -textvariable StimmenUngueltig \
    -background white
pack .stimmen.abgegebenLabel .stimmen.abgegebenEntry \
    .stimmen.ungueltigLabel .stimmen.ungueltigEntry -side left

frame .parteien
pack .parteien -padx 10 -pady 10
label .parteien.name -text "Name"
label .parteien.stimmen -text "Stimmen"
grid x .parteien.name .parteien.stimmen

for { set i 1 } { $i <= 5 } { incr i } {
  label .parteien.partei$i -text "Partei $i"
  entry .parteien.name$i -textvariable name$i -background white
  entry .parteien.stimmen$i -textvariable stimmen$i \
    -background white
  grid .parteien.partei$i .parteien.name$i .parteien.stimmen$i
}
```

Aufgabe 4:

Das folgende Programm berechnet die Sitzverteilung nach Hare-Niemeyer. Bei diesem Verfahren wird zuerst eine Idealzahl berechnet, die sich aus den *Stimmen für die Partei* dividiert durch *Stimmen insgesamt* mal *Sitze zu vergeben* ergibt. In der Regel hat die Idealzahl Nachkommastellen, die zunächst abgeschnitten werden. Daraus ergibt sich die Grundverteilung. Verbleiben nichtvergebene Sitze, werden diese nacheinander den Parteien mit den größten Nachkommastellen der Idealzahl zugeteilt.

Das Programm enthält Lücken. Dort muss entweder `stimmen($partei)`, `$stimmen($partei)`, `sitze($partei)` oder `$sitze($partei)` stehen.

(a) Füllen Sie die Lücken!

```
# Sitzverteilung nach Hare-Niemeyer

# Gesamtzahl der Sitze einlesen
puts -nonewline "Anzahl der zu vergebenen Sitze? "; flush stdout
gets stdin sitzeGesamt

# Parteinamen und deren Stimmenzahl einlesen
set stimmenGesamt 0
while { true } {
  puts -nonewline "Parteiname? "; flush stdout
  gets stdin partei
  if { $partei == "" } break
  puts -nonewline "Stimmenzahl? "; flush stdout
  gets stdin anzahl
  set ______________________ $anzahl
  set stimmenGesamt [expr $stimmenGesamt + $anzahl]
}

# Grundverteilung der Sitze
set sitzeVergeben 0
foreach partei [array names stimmen] {
  set sitzeIdeal($partei) \
    [expr (______________________ / $stimmenGesamt.0) * $sitzeGesamt]
  set ______________________ [expr int($sitzeIdeal($partei))]
  set sitzeVergeben [expr $sitzeVergeben + ______________________]
  set bruchteil [expr $sitzeIdeal($partei) - ______________________]
  lappend bruchteile [list $partei $bruchteil]
}
```

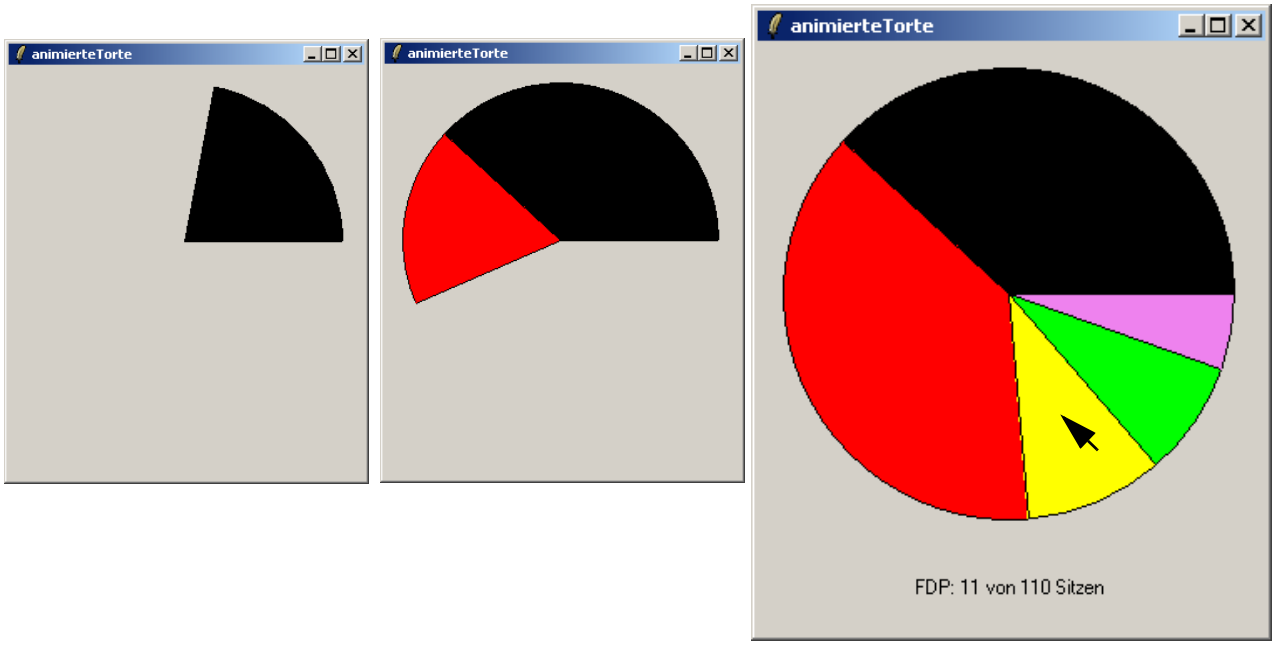
```
# Restsitzverteilung
set bruchteile [lsort -real -decreasing -index 1 $bruchteile]
foreach bruchteil $bruchteile {
  if { $sitzeVergeben == $sitzeGesamt } break
  set partei [lindex $bruchteil 0]
  set ____________________ [expr ____________________________ + 1]
  incr sitzeVergeben
}

# Ausgabe der Sitzverteilung
puts "Sitzverteilung:"
foreach partei [lsort [array names sitze]] {
  puts "$partei: ____________________ Sitze"
}
```

(b) In welcher Reihenfolge erscheint die Ausgabe der Sitzverteilung?

Aufgabe 5:

Die Sitzverteilung nach Wahlen wird im Fernsehen immer mit animierten „Tortendiagrammen“ angezeigt. Die drei Bilder zeigen das Diagramm in der Entstehung, beim letzten Diagramm haben wir mit der Maus ein Ereignis ausgelöst.

Ergänzen Sie die Lücken!




```
set parteien { { CDU 42 black } { SPD 42 red } { FDP 11 yellow }
    { Grüne 9 green } { Linke 6 violet } }


set gesamtsitze 0
foreach partei $parteien {
  set sitze [lindex $partei 1]
  set gesamtsitze [expr ____________________________________]
}


canvas .c -width 300 -height 350
pack .c
.c create text 150 325 -anchor center -tags ____________________


set start 0
```

```
foreach partei $parteien {
  set name [lindex $partei 0]
  set sitze [_____________________________________]
  set farbe [_____________________________________]
  set anteil [expr $sitze.0/$gesamtsitze]
  set winkel [expr $anteil*360]
  .c create arc 15 15 285 285 -start $start -extent 0 \
      -fill $farbe -tags $name
  for {set i 0} {$i < $winkel} {incr i} {
    .c itemconfigure ___________________________________________
    update
    after 30
  }
  .c itemconfigure $name -extent $winkel
  set start [expr $start + $winkel]

  .c bind $name <Enter> \
      ".c itemconfigure ausgabe \
      -text \"_____________________________________________________\""
  .c bind $name <Leave> \
      ".c itemconfigure ausgabe -text {}"

}
```

Ende der Klausur

Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

MUSTERLÖSUNG

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1	6	
2	6 + 3	
3	6	
4	6 + 2	
5	6	
Summe	35	

Aufgabe 1:

In dieser Klausur geht es um Wahlen. Zuerst bauen wir uns einen vereinfachten Wahlcomputer. Wie man sieht, kann man für fünf Parteien abstimmen. Klickt man einen der Partei-Buttons an, erscheint das rechte obere Fenster. Klickt man darin „Ja“ an, erscheint das rechte untere Fenster. Schreiben Sie auf die nächste Seite die passende Prozedur `waehle`, die dieses Fenster erzeugt. Beim Klicken des OK-Buttons beendet sich das Programm.


```
set parteien { CDU SPD FDP Grünen Linke }

label .prompt -text "Bitte treffen Sie Ihre Wahl:"
pack .prompt -padx 5 -pady 5

set i 1
foreach partei $parteien {
  button .b$i -text $partei -command "bestaetige $partei"
  pack .b$i -padx 5 -pady 5
  incr i
}

proc bestaetige { partei } {
  toplevel .t
  label .t.label \
    -text "Bitte bestätigen Sie: Möchten Sie für die $partei stimmen?"
  pack .t.label -padx 5 -pady 5
  frame .t.buttons
  pack .t.buttons -padx 5 -pady 5
  button .t.buttons.ja -text "Ja" -command "destroy .t; waehle $partei"
  button .t.buttons.nein -text "Nein" -command "destroy .t"
  pack .t.buttons.ja .t.buttons.nein -side left
}
```

Hier die Prozedur einfügen:

```
proc waehle { partei } {
    toplevel .t
    label .t.label -text "Sie haben für die $partei gestimmt."
    pack .t.label -padx 5 -pady 5
    button .t.ok -text "OK" -command exit
    pack .t.ok -padx 5 -pady 5
}
```

Aufgabe 2:

Wenn man wählen geht, prüft die Wahlkommission, ob man im Wählerverzeichnis eingetragen ist. Hierzu haben wir ein Programm, das eine Liste der vorhandenen Wähler enthält. Darin kann man Namen markieren, die farblich hinterlegt werden, wie gezeigt. Man markiert, indem man mit der Maus den Namen auswählt und dann den „Markiere Auswahl“-Button anklickt.

(a) Füllen Sie die Lücken im Programm!


```
label .lb -text "____Wahlberechtigte_____"
pack .lb -padx 5 -pady 5
frame .fr
pack .fr
listbox .fr.liste -yscroll ".fr.scroll set"
scrollbar .fr.scroll -command ".fr.liste yview"
pack ___.fr.liste___ -side left -expand yes -fill both
pack ___.fr.scroll___ -side right -fill y

.fr.liste insert 0 \
  "Becker, Ella" "Fischer, Gerda" "Hoffmann, Klaus" \
  "Meier, Heinrich" "Müller, Anna" "Musterfrau, Claudia" \
  "Neumann, Erich" "Pfeiffer, Horst" "Schröder, Bernd" \
  "Schulze, Friedrich" "Wagner, Britta" "Weber, Anton" \
  "Zimmermann, Karin"
```

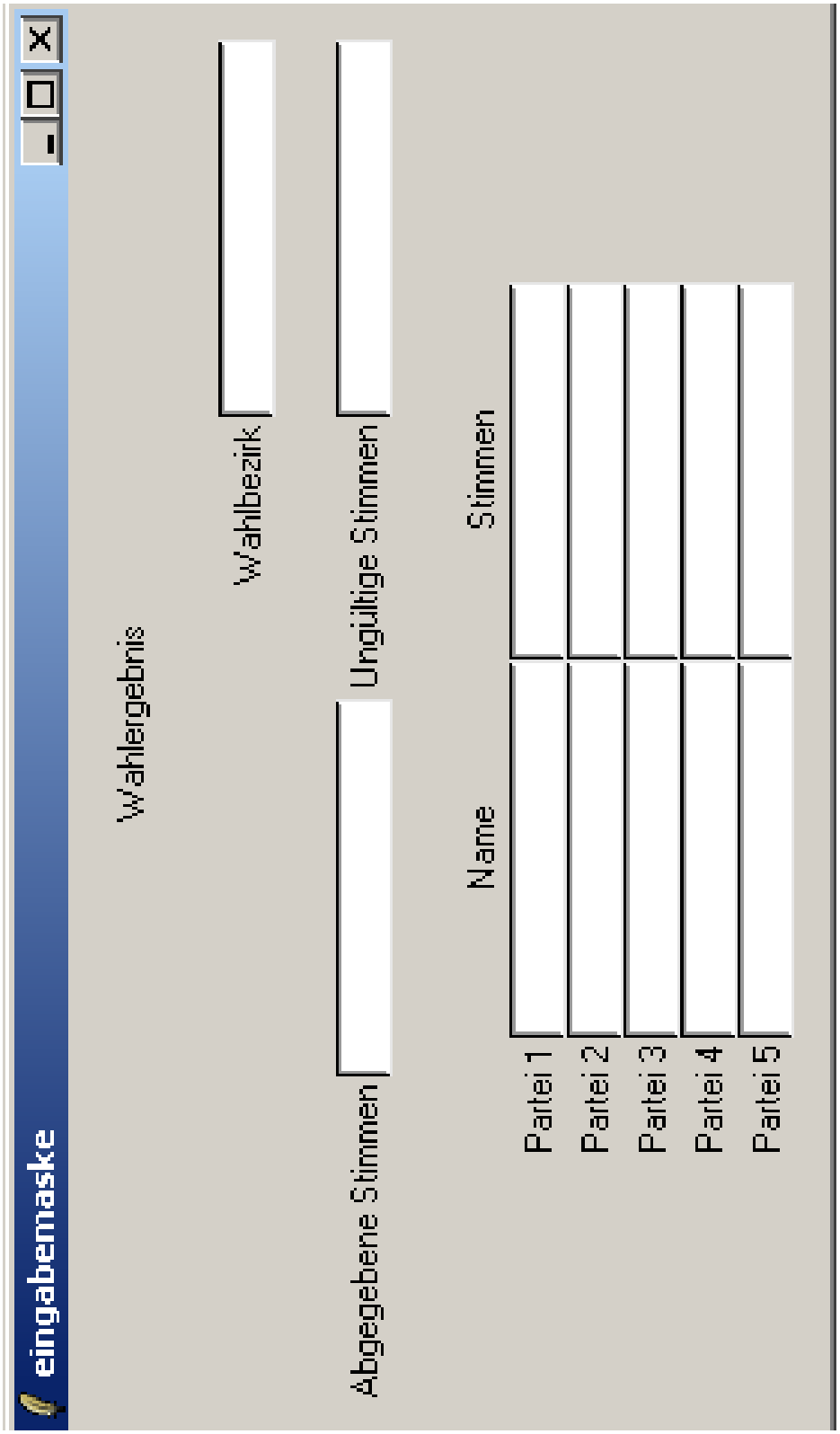
```
button .bt -text "Markiere Auswahl" -___command____ {
  if { [.fr.liste curselection] != "" } { __markiere__ _.fr.liste__ }
}
pack .bt -padx 5 -pady 5

proc istMarkiert { listbox } {
  if {[$listbox itemcget [$listbox curselection] -background] == "green"} {
    return true
  } else {
    return false
  }
}

proc markiere { listbox } {
  if { [istMarkiert $listbox] } {
    toplevel .tl
    label .tl.lb -text \
      "[$listbox get [$listbox curselection]] ist bereits markiert!"
    pack .tl.lb -padx 5 -pady 5
    button .tl.bt -text "OK" -command "destroy .tl"
    pack .tl.bt -padx 5 -pady 5
  } else {
    $listbox itemconfigure [$listbox curselection] \
      -background green
  }
}
```

(b) Was passiert, wenn man einen bereits markierten Listeneintrag nochmals mit der Maus auswählt und dann den Markier-Button anklickt? Beschreiben Sie die Anzeige und das Verhalten genau!

Es erscheint ein Toplevel-Fenster mit dem Label „*Name* ist bereits markiert!" Darunter ein Knopf mit Beschriftung OK. Bei Klicken auf diesen OK-Knopf wird das Fenster wieder geschlossen.

Aufgabe 3:

Jeder Wahlbezirk meldet die Ergebnisse an den Wahlleiter. Dazu gibt es eine Eingabemaske, die von dem Tcl/Tk-Programm unten erzeugt wird. Zeichnen Sie die Ausgabe möglichst genau ein!



```
label .titel -text "Wahlergebnis"
pack .titel -padx 10 -pady 10

frame .bezirk
pack .bezirk -fill both -padx 10 -pady 10
label .bezirk.label -text "Wahlbezirk"
entry .bezirk.entry -textvariable bezirk -background white
pack .bezirk.entry .bezirk.label -side right

frame .stimmen
pack .stimmen -fill both -padx 10 -pady 10
label .stimmen.abgegebenLabel -text "Abgegebene Stimmen"
entry .stimmen.abgegebenEntry -textvariable StimmenTotal \
    -background white
label .stimmen.ungueltigLabel -text "Ungültige Stimmen"
entry .stimmen.ungueltigEntry -textvariable StimmenUngueltig \
    -background white
pack .stimmen.abgegebenLabel .stimmen.abgegebenEntry \
    .stimmen.ungueltigLabel .stimmen.ungueltigEntry -side left

frame .parteien
pack .parteien -padx 10 -pady 10
label .parteien.name -text "Name"
label .parteien.stimmen -text "Stimmen"
grid x .parteien.name .parteien.stimmen

for { set i 1 } { $i <= 5 } { incr i } {
  label .parteien.partei$i -text "Partei $i"
  entry .parteien.name$i -textvariable name$i -background white
  entry .parteien.stimmen$i -textvariable stimmen$i \
    -background white
  grid .parteien.partei$i .parteien.name$i .parteien.stimmen$i
}
```

Aufgabe 4:

Das folgende Programm berechnet die Sitzverteilung nach Hare-Niemeyer. Bei diesem Verfahren wird zuerst eine Idealzahl berechnet, die sich aus den *Stimmen für die Partei* dividiert durch *Stimmen insgesamt* mal *Sitze zu vergeben* ergibt. In der Regel hat die Idealzahl Nachkommastellen, die zunächst abgeschnitten werden. Daraus ergibt sich die Grundverteilung. Verbleiben nichtvergebene Sitze, werden diese nacheinander den Parteien mit den größten Nachkommastellen der Idealzahl zugeteilt.

Das Programm enthält Lücken. Dort muss entweder `stimmen($partei)`, `$stimmen($partei)`, `sitze($partei)` oder `$sitze($partei)` stehen.

(a) Füllen Sie die Lücken!

```
# Sitzverteilung nach Hare-Niemeyer

# Gesamtzahl der Sitze einlesen
puts -nonewline "Anzahl der zu vergebenen Sitze? "; flush stdout
gets stdin sitzeGesamt

# Parteinamen und deren Stimmenzahl einlesen
set stimmenGesamt 0
while { true } {
  puts -nonewline "Parteiname? "; flush stdout
  gets stdin partei
  if { $partei == "" } break
  puts -nonewline "Stimmenzahl? "; flush stdout
  gets stdin anzahl
  set _stimmen($partei)__ $anzahl
  set stimmenGesamt [expr $stimmenGesamt + $anzahl]
}

# Grundverteilung der Sitze
set sitzeVergeben 0
foreach partei [array names stimmen] {
  set sitzeIdeal($partei) \
    [expr (_$stimmen($partei)__ / $stimmenGesamt.0) * $sitzeGesamt]
  set __sitze($partei)__ [expr int($sitzeIdeal($partei))]
  set sitzeVergeben [expr $sitzeVergeben + __$sitze($partei)__]
  set bruchteil [expr $sitzeIdeal($partei) - __$sitze($partei)__]
  lappend bruchteile [list $partei $bruchteil]
}
```

```
# Restsitzverteilung
set bruchteile [lsort -real -decreasing -index 1 $bruchteile]
foreach bruchteil $bruchteile {
  if { $sitzeVergeben == $sitzeGesamt } break
  set partei [lindex $bruchteil 0]
  set __sitze($partei)___ [expr __$sitze($partei)__ + 1]
  incr sitzeVergeben
}

# Ausgabe der Sitzverteilung
puts "Sitzverteilung:"
foreach partei [lsort [array names sitze]] {
  puts "$partei: ___$sitze($partei)__ Sitze"
}
```

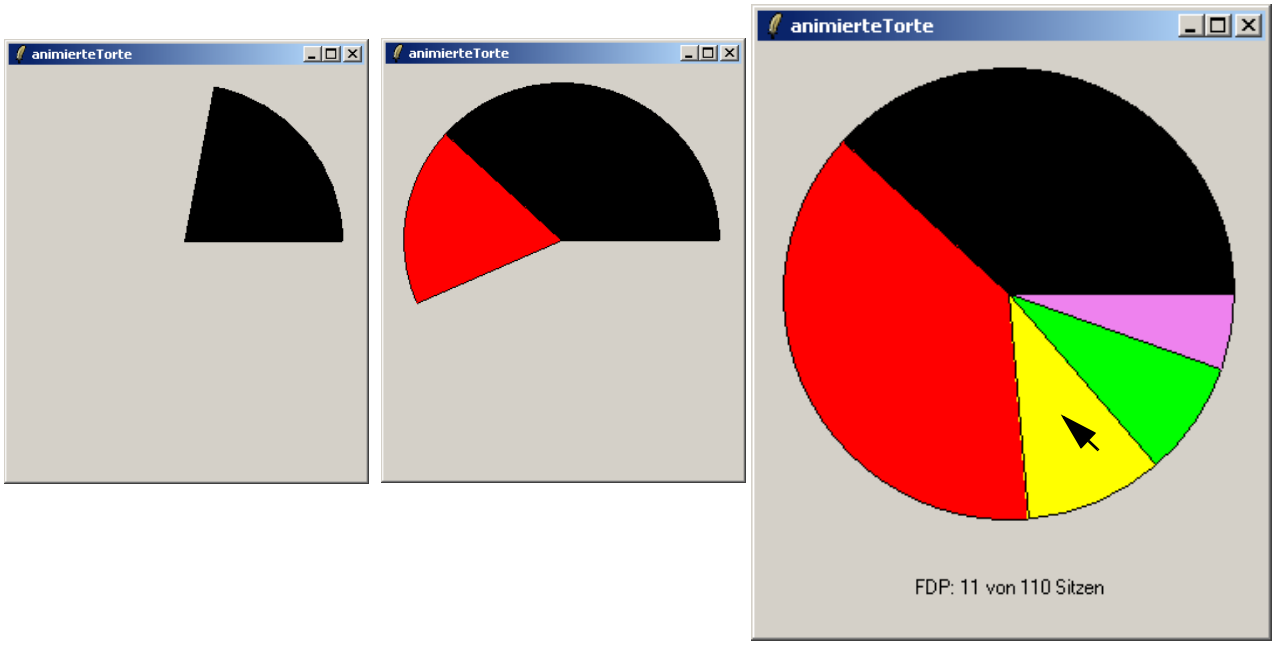
(b) In welcher Reihenfolge erscheint die Ausgabe der Sitzverteilung?

Die Ausgabe erfolgt alphabetisch sortiert nach den Parteinamen.

Aufgabe 5:

Die Sitzverteilung nach Wahlen wird im Fernsehen immer mit animierten „Tortendiagrammen“ angezeigt. Die drei Bilder zeigen das Diagramm in der Entstehung, beim letzten Diagramm haben wir mit der Maus ein Ereignis ausgelöst.

Ergänzen Sie die Lücken!




```
set parteien { { CDU 42 black } { SPD 42 red } { FDP 11 yellow }
    { Grüne 9 green } { Linke 6 violet } }


set gesamtsitze 0
foreach partei $parteien {
  set sitze [lindex $partei 1]
  set gesamtsitze [expr __$gesamtsitze + $sitze__]
}


canvas .c -width 300 -height 350
pack .c
.c create text 150 325 -anchor center -tags __ausgabe__


set start 0
```

```
foreach partei $parteien {
  set name [lindex $partei 0]
  set sitze [__lindex $partei 1__]
  set farbe [__lindex $partei 2__]
  set anteil [expr $sitze.0/$gesamtsitze]
  set winkel [expr $anteil*360]
  .c create arc 15 15 285 285 -start $start -extent 0 \
      -fill $farbe -tags $name
  for {set i 0} {$i < $winkel} {incr i} {
    .c itemconfigure __$name -extent $i___
    update
    after 30
  }
  .c itemconfigure $name -extent $winkel
  set start [expr $start + $winkel]

  .c bind $name <Enter> \
      ".c itemconfigure ausgabe \
      -text \"__$name: $sitze von $gesamtsitze Sitzen__\""
  .c bind $name <Leave> \
      ".c itemconfigure ausgabe -text {}"

}
```

Ende der Klausur

Klausur zur Vorlesung
„Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1	2+2+2+5+3	
2	8+4+4+2+2	
3	6+2+2+2	
4	6+2+2+2	
Summe	58	

Zu Aufgabe 1:

Hier ist eine Liste der gängigen Listen-Kommandos.

```
lindex list i
# Gibt Listenelement mit Index i zurück
# ACHTUNG: Das erste Element hat Index 0

llength list
# Gibt Anzahl der Listenelemente zurück

lrange list i j
# Gibt die Listenelemente zwischen
# Index i und j (inklusive) als Liste zurück

lappend listVar arg ?...?
# Fügt neue Elemente an die Liste mit Variablennamen
# listVar an

linsert list i arg
# Fügt vor Position i das neue Element arg ein. Gibt
# die modifizierte Liste zurück

lsearch ?options? list pattern
# Sucht in der Liste nach dem Muster und liefert den Index
# des ersten passenden Elements, sonst -1. Die Optionen bestimmen
# die Suche, ohne Angabe wird -glob (wie string match) genommen.

lreplace list first last ?element element ...?
# Ersetzt Elemente der Liste durch neue Elemente und liefert
# die neue Liste zurück. first und last bestimmen den ersten und
# letzten Index der zu ersetzenden Elemente. Die optional angegebenen
# Elemente treten an deren Stelle. Fehlt diese Angabe, werden die
# Elemente zwischen first und last einfach entfernt.

lsort ?options? list
# Sortiert die Elemente der Liste und liefert eine neue sortierte
# Liste zurück. Die Standardvorgabe ist ASCII-Sortierung, eine
# numerische Sortierung erreicht man mit der Option -integer.
```

Aufgabe 1:

Hinweis: Für die folgenden Teilaufgaben (a) - (e) verwenden Sie die Listenkommandos auf der gegenüberliegenden Seite.

Nehmen wir an, der Wert der Variablen **zahlen** sei eine Liste der von uns ausgewählten Lottozahlen.

(a) Die Liste muss genau 6 Zahlen enthalten. Prüfen Sie die Länge der Liste!

```
if {_____________________________________________ } {
  puts "Falsche Anzahl: $zahlen" }
```

(b) In der Liste sollte jedes Element nur einmal auftreten. Wir regeln das dadurch, dass wir ein neues Element **zahl** nur dann in **zahlen** einfügen, wenn es dort nicht schon vorkommt.

```
if { ____________________________________ == -1 } {
  lappend zahlen $zahl }
```

(c) Die Liste der Lottozahlen sollte sortiert sein.

```
set zahlen ____________________________________________
```

(d) Prüfen Sie, ob **zahl** in **zahlen** bereits vorkommt und wenn ja, löschen Sie das Element in **zahlen**. Sonst fügen Sie **zahl** an die Liste **zahlen** an.

```
set temp ____________________________________ ;# Index von zahl merken
IF { ____________________________________________ } {
  # loeschen; set zahlen ____________________________________} else {
  # anfuegen; ____________________________________________________
}
```

(e) Eine einzelne Lottozahl könnten wir uns auch per Zufallszahlgenerator erzeugen. Schreiben Sie hierzu eine kleine, parameterlose Prozedur **erzeuge**, die eine Zufallszahl zwischen 1 und 49 erzeugt. Hinweis: **rand()** erzeugt eine Zufallszahl im Intervall (0, 1), **int()** schneidet Nachkommastellen ab.

Aufgabe 2:

Die unten gezeigte Eingabemaske erlaubt die Eingabe der Lottozahlen mit einem numerischen Tastaturblock. Das Programm wurde aus dem Taschenrechnerbeispiel der Vorlesung abgeleitet.

(a) Ergänzen Sie die fehlenden Programmstellen.

```
#!/usr/bin/wish
wm title . "Lotto Eingabe"
set zahlen ""; # die Lottozahlen als Liste
set zahl 0   ; # die gerade gewählte Zahl
set i 1      ; # wir wollen 6 Zahlen

proc zahlhinzu {z} {
  global zahlen i
  if {$z < 1 || $z > 49} {

    toplevel .f -bg red
    wm title .f "Falsche Eingabe"
    message .f.m -width 400 -justify center \
      -padx 20 -bg red -fg white\
      -text "Falsche Zahl $z: nur 1 bis 49 möglich!"
    pack .f.m
    button .f.b -text weiter -command {destroy .f}
    pack .f.b -padx 5 -pady 5

  } else {
    lappend ____________________________________
    set zahlen [_____________________________________________];#sortiert
    incr i
  }
}

proc danke {zz} {
    toplevel .d -bg blue
    wm title .d "Fertig"
    message .d.m -width 400 -justify center \
    -padx 20 -bg blue -fg white\
        -text "Danke!\n Die gewählten Zahlen lauten:\n $zz"
    pack .d.m
    button .d.b -text "Zahlen abschicken" -command {destroy .d; exit}
    pack .d.b -padx 5 -pady 5
}
```

```
#
# This procedure handles the entry keys:
#
proc keypress {char} {
    global zahl zahlen i

    switch -- $char {
    "CE" { set zahl "" }
    "C" {
      set zahl ""
      set zahlen ""
      set i 1
      .next configure -text "$i. Zahl"
    }
    0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 {
      if {[string compare $zahl "0"] == 0} { set zahl "" }
      append zahl $char
    }
    "ENTER" {
      set result [_____________________________]; #neue Zahl dazu
      set zahl ""
      if {$i > 6} { danke $zahlen }
      .next configure -text "$i. Zahl"
      .liste configure -text "Ihre Zahlen: _____________________"
    }
  };#switch
}

label .next -text "_____________________________"
entry .readout -textvariable zahl -bg white -fg black -justify right
label .liste -text "Ihre Zahlen: $zahlen" -anchor w
for {set j 0} {$j <= 9} {incr j} {
  button .key$j  -text "$j" -width 3 -command "keypress $j"
}
button .clearentry  -text "CE" -width 3 -command {keypress "CE"}
button .clearall -text "C" -width 3 -command {keypress "C"}
button .enter -text "E\nN\nT\nE\nR" -width 3 -command {keypress "ENTER"}

grid .next .readout -sticky nsew
grid .liste -sticky nsew
grid .key1 .key2 .key3  .enter -sticky nsew
grid .key4 .key5 .key6  -sticky nsew
grid .key7 .key8 .key9  -sticky nsew
grid .clearentry .key0 .clearall -sticky nsew

grid configure .readout -columnspan 3
grid configure .liste -columnspan 4
grid configure .enter _____________________________________________

for {set j 0} {$j <= 9} {incr j} {
  bind all <_________________________> "_____________________________"
};# Tastatureingabe der Ziffern
bind all <KeyPress-Return>     {keypress "ENTER"}
bind all <KeyPress-Escape>     {keypress "CE"}
```

```
set max [lindex [grid size .] 0]
for {set j 0} {$j < $max} {incr j} {
    grid columnconfigure . $j -weight 1
}

set max [lindex [grid size .] 1]
for {set j 1} {$j < $max} {incr j} {
    grid rowconfigure . $j -weight 1
}
```

(b) Was passiert, wenn alle 6 Zahlen eingegeben wurden? Beschreiben Sie den Ablauf und die Anzeige.

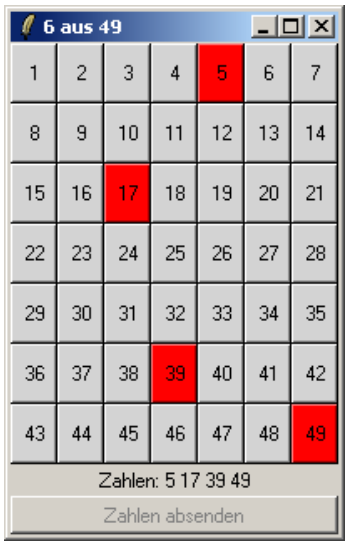
(c) Was passiert, wenn man eine ungültige Zahl *x* kleiner 1 oder größer 49 eingibt? Beschreiben Sie den Ablauf und die Anzeige.

(d) Wie reagiert das Programm auf den Versuch, eine Lottozahl mehrfach einzugeben?

(e) Erklären Sie den Unterschied der Wirkungsweise des C- und des CE-Knopfes in diesem Programm.

Aufgabe 3:

Bei dieser Eingabemaske erfolgt die Eingabe über eine 7 x 7 Tastenmatrix. Das Programm wurde aus dem Tic-Tac-Toe-Programm in den Übungen abgeleitet.



(a) Füllen Sie die Lücken!

```
#!/usr/bin/wish
set numberlist ""
wm title . "6 aus 49"

frame .board
pack .board -side top -expand yes -fill both

button .send -text "_____________________________" -state disabled \
    -command {exit}
pack .send -side bottom -fill x

label .status
pack .status -side bottom -fill x

#create buttons
for {set i 0} {$i<=6} {incr i} {
  grid rowconfigure .board $i -weight 1
  for {set j 1} {$j<=7} {incr j} {
    grid columnconfigure .board $j -weight 1
    button .board.field$i,$j -height 2 -width 3 \
      -text [_____________________________________] \
      -bg "lightgrey" -activebackground "grey92"
    grid .board.field$i,$j -row _______-column ________ -sticky nsew
    bind .board.field$i,$j <ButtonRelease-1> {
      # %W is widget pathname
      %W flash
      if { [lsearch $numberlist [%W cget -text]] == -1 &&
           [llength $numberlist] < 6 } {
        %W configure -bg "red" -activebackground "IndianRed1"
```

```
        set numberlist [lsort -integer [lappend numberlist \
          [%W cget -text]]]
      } else { ;# selektierte Zahl loeschen
          %W configure -bg "lightgrey" \
            -activebackground "grey92"
           set pos [lsearch $numberlist [%W cget -text]]
          set numberlist [lreplace ____________________________]
      }
      .status configure -text "Zahlen: $numberlist"
      if { [llength $numberlist] == 6 } {
        .send configure -state normal
        for {set i 0} {$i<=6} {incr i} {
          for {set j 1} {$j<=7} {incr j} {
            if { [lsearch $numberlist \
                 [.board.field$i,$j cget -text]] == -1 } {
              .board.field$i,$j configure -state "disabled"
            };# if
          };# for j
        };# for i
      } else {
        .send configure -state disabled
        for {set i 0} {$i<=6} {incr i} {
          for {set j 1} {$j<=7} {incr j} {
              .board.field$i,$j configure -state "normal"
          };# for j
        };# for i
      };# if
    };# bind
  };# for j
};#for i
```

(b) Was genau passiert, wenn man einen der 49 Knöpfe drückt und dann losläßt? Beschreiben Sie auch die Anzeige.

(c) Was genau passiert, wenn man einen bereits gedrückten Knopf nochmals drückt?

(d) Was genau passiert, wenn die 6. Zahl eingegeben wurde?

Aufgabe 4:

Jetzt werden die Zahlen direkt auf einem 7 x 7 Feld eingegeben, indem man mit der linken Maustaste eine Zahl anklickt. Klickt man nochmals auf ein bereits angekreuztes Feld, wird die Zahl aus der Liste und das Kreuzchen aus der Anzeige wieder entfernt. Das Programm verwendet Techniken aus dem Kalenderprogramm im Skript, speziell werden 49 leere Rahmen über die Zahlen gelegt, die per Mausklick reagieren.

(a) Ergänzen Sie die Lücken im Programm!

```
#!/usr/bin/wish
wm title . "Spiel 6 aus 49"
set zahlen ""

proc setze {win num} {
  # berechnen Spalte und Zeile für das Kreuzchen
  global zahlen
  set row [expr ($num - 1)/ 7]
  set col [expr ($num - 1) % 7]
  if {[lsearch $zahlen $num] < 0 && [llength $zahlen] < 6} {
    lappend zahlen $num
    set zahlen [lsort -integer $zahlen]
    $win create line [expr $col * 20 + 7] [expr $row * 20 + 7] \
      [expr $col * 20 + 23] [expr $row * 20 + 23] \
      -fill black -width 2 -tags [list $num-line1]
    $win create line [expr $col * 20 + 23] [expr $row * 20 + 7] \
      [expr $col * 20 + 7] [expr $row * 20 + 23] \
      -fill black -width 2 -tags [list $num-line2]
    #unsichtbares Rechteck muss ganz oben sein
    $win raise $num-sensor $num-line2; update
  } else {
      set zahlen [lreplace $zahlen [lsearch $zahlen $num] \
        [lsearch $zahlen $num]]
      $win delete $num-line1 $num-line2
  }; # else Zahl schon gesetzt oder schon 6 Zahlen
  .t.anzeige configure -text "Ihre Zahlen: $zahlen"

  if {[_____________________] == 6} { .t.send configure -state normal
  } else {.t.send configure -state disabled}
};# setze
```

```
frame .t
pack .t
canvas _____________________ -width 150 -height 150
pack ______________________ -expand yes -fill both


set x0 5; set y0 5; set x1 25; set y1 25
for {set i 1} {$i <=7} {incr i} {
  for {set j 1} {$j <=7} {incr j} {
    .t.spiel create ____________________ $x0 $y0 $x1 $y1 -outline red
    set zahl [expr ($i-1)*7 + $j]
    .t.spiel create text [expr $x0 + 10] [expr $y0 + 10] \
      -anchor c -text $zahl -fill red
    # leeres Rechteck für Sensor
    .t.spiel create rectangle $x0 $y0 $x1 $y1 -outline "" -fill "" \
      -tags [list $zahl-sensor]
    .t.spiel bind $zahl-sensor <ButtonPress-1> \
      [list setze _________________  ____________________________ ]
    incr x0 20; incr x1 20
  }
  set x0 5; set x1 25
  incr y0 20; incr y1 20
}

label .t.anzeige -text "Ihre Zahlen: $zahlen"
pack .t.anzeige -padx 5 -pady 2
button .t.send -text "Zahlen absenden" -state disabled \
    -command {exit}
pack .t.send -side bottom -fill x
```

(a) Was genau passiert, wenn alle sechs Zahlen angekreuzt wurden?

(b) Wie wird das Kreuzchen über einer angeklickten Zahl produziert?

(c) Wie heißt der Tag, der an das Feld mit der Zahl 7 gebunden ist?

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2008

Nachname: MUSTERLÖSUNG Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte maximal	Punkte erreicht
1	2+2+2+5+3	
2	8+4+4+2+2	
3	6+2+2+2	
4	6+2+2+2	
Summe	58	

Zu Aufgabe 1:

Hier ist eine Liste der gängigen Listen-Kommandos.

lindex *list i*
Gibt Listenelement mit Index *i* zurück
ACHTUNG: Das erste Element hat Index 0

llength *list*
Gibt Anzahl der Listenelemente zurück

lrange *list i j*
Gibt die Listenelemente zwischen
Index *i* und *j* (inklusive) als Liste zurück

lappend *listVar arg ?...?*
Fügt neue Elemente an die Liste mit Variablennamen
listVar an

linsert *list i arg*
Fügt vor Position *i* das neue Element arg ein. Gibt
die modifizierte Liste zurück

lsearch *?options? list pattern*
Sucht in der Liste nach dem Muster und liefert den Index
des ersten passenden Elements, sonst -1. Die Optionen bestimmen
die Suche, ohne Angabe wird -glob (wie string match) genommen.

lreplace *list first last ?element element ...?*
Ersetzt Elemente der Liste durch neue Elemente und liefert
die neue Liste zurück. *first* und *last* bestimmen den ersten und
letzten Index der zu ersetzenden Elemente. Die optional angegebenen
Elemente treten an deren Stelle. Fehlt diese Angabe, werden die
Elemente zwischen *first* und *last* einfach entfernt.

lsort *?options? list*
Sortiert die Elemente der Liste und liefert eine neue sortierte
Liste zurück. Die Standardvorgabe ist ASCII-Sortierung, eine
numerische Sortierung erreicht man mit der Option -integer.

Aufgabe 1:

Hinweis: Für die folgenden Teilaufgaben (a) - (e) verwenden Sie die Listenkommandos auf der gegenüberliegenden Seite.

Nehmen wir an, der Wert der Variablen **zahlen** sei eine Liste der von uns ausgewählten Lottozahlen.

2 Pkte (a) Die Liste muss genau 6 Zahlen enthalten. Prüfen Sie die Länge der Liste!

```
if { ____[llength $zahlen] != 6____ } {
  puts "Falsche Anzahl: $zahlen" }
```

2 Pkte (b) In der Liste sollte jedes Element nur einmal auftreten. Wir regeln das dadurch, dass wir ein neues Element **zahl** nur dann in **zahlen** einfügen, wenn es dort nicht schon vorkommt.

```
if { ___[lsearch $zahlen $zahl]___ == -1 } {
  lappend zahlen $zahl }
```

2 Pkte (c) Die Liste der Lottozahlen sollte sortiert sein.

```
set zahlen _____[lsort -integer $zahlen]___
```

5 Pkte (d) Prüfen Sie, ob **zahl** in **zahlen** bereits vorkommt und wenn ja, löschen Sie das Element in **zahlen**. Sonst fügen Sie **zahl** an die Liste **zahlen** an.

```
set temp ___[lsearch $zahlen $zahl]__ ;# Index von zahl merken
IF { __$temp != -1_ } {                auch $temp >= 0
  # loeschen; set zahlen _[lreplace $zahlen $temp $temp]__ } else {
  # anfuegen; __lappend zahlen $zahl__
}
```

3 Pkte (e) Eine einzelne Lottozahl könnten wir uns auch per Zufallszahlgenerator erzeugen. Schreiben Sie hierzu eine kleine, parameterlose Prozedur **erzeuge**, die eine Zufallszahl zwischen 1 und 49 erzeugt. Hinweis: **rand()** erzeugt eine Zufallszahl im Intervall (0, 1), **int()** schneidet Nachkommastellen ab.

```
proc erzeuge { } {
  return [expr int(49 * rand() + 1)]
}
# oder ähnliche Konstruktionen
```

Aufgabe 2:

Die unten gezeigte Eingabemaske erlaubt die Eingabe der Lottozahlen mit einem numerischen Tastaturblock. Das Programm wurde aus dem Taschenrechnerbeispiel der Vorlesung abgeleitet.

8 Pkte (a) Ergänzen Sie die fehlenden Programmstellen.

```
#!/usr/bin/wish
wm title . "Lotto Eingabe"
set zahlen ""; # die Lottozahlen als Liste
set zahl 0   ; # die gerade gewählte Zahl
set i 1      ; # wir wollen 6 Zahlen

proc zahlhinzu {z} {
  global zahlen i
  if {$z < 1 || $z > 49} {

    toplevel .f -bg red
    wm title .f "Falsche Eingabe"
    message .f.m -width 400 -justify center \
      -padx 20 -bg red -fg white\
      -text "Falsche Zahl $z: nur 1 bis 49 möglich!"
    pack .f.m
    button .f.b -text weiter -command {destroy .f}
    pack .f.b -padx 5 -pady 5

  } else {
    lappend __zahlen $z__
    set zahlen [___lsort -integer $zahlen__];#sortiert
    incr i
  }
}

proc danke {zz} {
    toplevel .d -bg blue
    wm title .d "Fertig"
    message .d.m -width 400 -justify center \
    -padx 20 -bg blue -fg white\
        -text "Danke!\n Die gewählten Zahlen lauten:\n $zz"
    pack .d.m
    button .d.b -text "Zahlen abschicken" -command {destroy .d; exit}
    pack .d.b -padx 5 -pady 5
}
```

```
#
# This procedure handles the entry keys:
#
proc keypress {char} {
    global zahl zahlen i

    switch -- $char {
    "CE" { set zahl "" }
    "C" {
      set zahl ""
      set zahlen ""
      set i 1
      .next configure -text "$i. Zahl"
    }
    0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 {
      if {[string compare $zahl "0"] == 0} { set zahl "" }
      append zahl $char
    }
    "ENTER" {
      set result [___zahlhinzu $zahl___]; #neue Zahl dazu
      set zahl ""
      if {$i > 6} { danke $zahlen }
      .next configure -text "$i. Zahl"
      .liste configure -text "Ihre Zahlen: ___$zahlen__"
    }
  };#switch
}

label .next -text "___$i. Zahl___"          oder auch "1. Zahl"
entry .readout -textvariable zahl -bg white -fg black -justify right
label .liste -text "Ihre Zahlen: $zahlen" -anchor w
for {set j 0} {$j <= 9} {incr j} {
  button .key$j  -text "$j" -width 3 -command "keypress $j"
}
button .clearentry  -text "CE" -width 3 -command {keypress "CE"}
button .clearall -text "C" -width 3 -command {keypress "C"}
button .enter -text "E\nN\nT\nE\nR" -width 3 -command {keypress "ENTER"}

grid .next .readout -sticky nsew
grid .liste -sticky nsew
grid .key1 .key2 .key3  .enter -sticky nsew
grid .key4 .key5 .key6  -sticky nsew
grid .key7 .key8 .key9  -sticky nsew
grid .clearentry .key0 .clearall -sticky nsew

grid configure .readout -columnspan 3
grid configure .liste -columnspan 4
grid configure .enter -rowspan 4            Enter-Knopf über 4 Zeilen

for {set j 0} {$j <= 9} {incr j} {
  bind all <___KeyPress-$j___> "___keypress $j__"
};# Tastatureingabe der Ziffern
bind all <KeyPress-Return>     {keypress "ENTER"}
bind all <KeyPress-Escape>     {keypress "CE"}
```

```
set max [lindex [grid size .] 0]
for {set j 0} {$j < $max} {incr j} {
    grid columnconfigure . $j -weight 1
}

set max [lindex [grid size .] 1]
for {set j 1} {$j < $max} {incr j} {
    grid rowconfigure . $j -weight 1
}
```

4 Pkte (b) Was passiert, wenn alle 6 Zahlen eingegeben wurden? Beschreiben Sie den Ablauf und die Anzeige.

Es wird die danke-Prozedur aufgerufen, die ein Fenster mit blauem Hintergrund, einem Text und den Zahlen erzeugt. Durch Drücken des „Zahlen abschicken"-Knopfes wird es wieder entfernt und das Programm wird beendet.

4 Pkte (c) Was passiert, wenn man eine ungültige Zahl *x* kleiner 1 oder größer 49 eingibt? Beschreiben Sie den Ablauf und die Anzeige.

Es wird ein Fenster mit rotem Hintergrund, dem Warntext „Falsche Eingabe <die zahl x>: nur 1 bis 49 möglich!" und einem Weiter-Knopf erzeugt. Durch Drücken des Knopfes wird das Fenster wieder abgeräumt. Die Zahl x wird natürlich nicht aufgenommen.

2 Pkte (d) Wie reagiert das Programm auf den Versuch, eine Lottozahl mehrfach einzugeben?

Der Fall ist nicht berücksichtigt, die Zahl wird mehrfach aufgenommen.

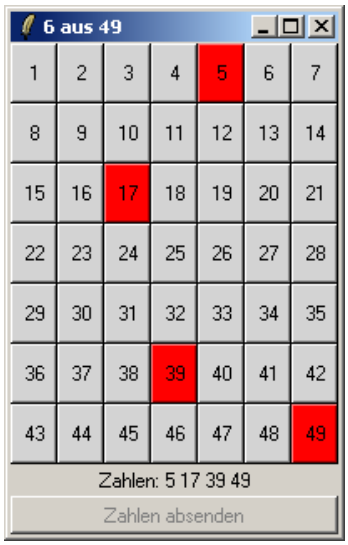
2 Pkte (e) Erklären Sie den Unterschied der Wirkungsweise des C- und des CE-Knopfes in diesem Programm.

C (Clear) löscht die eingegebene Zahl und die bereits gespeicherten Zahlen,

CE (Clear Entry) nur die letzte eingegebene Zahl.

Aufgabe 3:

Bei dieser Eingabemaske erfolgt die Eingabe über eine 7 x 7 Tastenmatrix. Das Programm wurde aus dem Tic-Tac-Toe-Programm in den Übungen abgeleitet.



6 Pkte (a) Füllen Sie die Lücken!

```
#!/usr/bin/wish
set numberlist ""
wm title . "6 aus 49"

frame .board
pack .board -side top -expand yes -fill both

button .send -text "__Zahlen absenden__" -state disabled \
    -command {exit}
pack .send -side bottom -fill x

label .status
pack .status -side bottom -fill x

#create buttons
for {set i 0} {$i<=6} {incr i} {
  grid rowconfigure .board $i -weight 1
  for {set j 1} {$j<=7} {incr j} {
    grid columnconfigure .board $j -weight 1
    button .board.field$i,$j -height 2 -width 3 \
      -text [__expr $j+7*$i__] \
      -bg "lightgrey" -activebackground "grey92"
    grid .board.field$i,$j -row __$i__ -column __$j__ -sticky nsew
    bind .board.field$i,$j <ButtonRelease-1> {
      # %W is widget pathname
      %W flash
      if { [lsearch $numberlist [%W cget -text]] == -1 &&
           [llength $numberlist] < 6 } {
        %W configure -bg "red" -activebackground "IndianRed1"
```

```
        set numberlist [lsort -integer [lappend numberlist \
          [%W cget -text]]]
      } else { ;# selektierte Zahl loeschen
          %W configure -bg "lightgrey" \
            -activebackground "grey92"
           set pos [lsearch $numberlist [%W cget -text]]
          set numberlist [lreplace __$numberlist $pos $pos__]
      }
      .status configure -text "Zahlen: $numberlist"
      if { [llength $numberlist] == 6 } {
        .send configure -state normal
        for {set i 0} {$i<=6} {incr i} {
          for {set j 1} {$j<=7} {incr j} {
            if { [lsearch $numberlist \
                 [.board.field$i,$j cget -text]] == -1 } {
              .board.field$i,$j configure -state "disabled"
            };# if
          };# for j
        };# for i
      } else {
        .send configure -state disabled
        for {set i 0} {$i<=6} {incr i} {
          for {set j 1} {$j<=7} {incr j} {
              .board.field$i,$j configure -state "normal"
          };# for j
        };# for i
      };# if
    };# bind
  };# for j
};#for i
```

2 Pkte (b) Was genau passiert, wenn man einen der 49 Knöpfe drückt und dann losläßt? Beschreiben Sie auch die Anzeige.

Der Knopf blinkt und wird rot gefärbt und die Zahl der Liste hinzugefügt.

(c) Was genau passiert, wenn man einen bereits gedrückten Knopf nochmals drückt?

2 Pkte

Der Knopf wird wieder normal gefärbt (grau) und die Zahl wieder aus der Liste gelöscht.

(d) Was genau passiert, wenn die 6. Zahl eingegeben wurde?

2 Pkte Der Send-Button wird aktiviert und alle nicht-gedrückten Zahl-Knöpfe werden deaktiviert. Bereits gedrückte Zahlen können aber noch gelöscht werden.

Aufgabe 4:

Jetzt werden die Zahlen direkt auf einem 7 x 7 Feld eingegeben, indem man mit der linken Maustaste eine Zahl anklickt. Klickt man nochmals auf ein bereits angekreuztes Feld, wird die Zahl aus der Liste und das Kreuzchen aus der Anzeige wieder entfernt. Das Programm verwendet Techniken aus dem Kalenderprogramm im Skript, speziell werden 49 leere Rahmen über die Zahlen gelegt, die per Mausklick reagieren.

6 Pkte (a) Ergänzen Sie die Lücken im Programm!

```
#!/usr/bin/wish
wm title . "Spiel 6 aus 49"
set zahlen ""

proc setze {win num} {
  # berechnen Spalte und Zeile für das Kreuzchen
  global zahlen
  set row [expr ($num - 1)/ 7]
  set col [expr ($num - 1) % 7]
  if {[lsearch $zahlen $num] < 0 && [llength $zahlen] < 6} {
    lappend zahlen $num
    set zahlen [lsort -integer $zahlen]
    $win create line [expr $col * 20 + 7] [expr $row * 20 + 7] \
      [expr $col * 20 + 23] [expr $row * 20 + 23] \
      -fill black -width 2 -tags [list $num-line1]
    $win create line [expr $col * 20 + 23] [expr $row * 20 + 7] \
      [expr $col * 20 + 7] [expr $row * 20 + 23] \
      -fill black -width 2 -tags [list $num-line2]
    #unsichtbares Rechteck muss ganz oben sein
    $win raise $num-sensor $num-line2; update
  } else {
      set zahlen [lreplace $zahlen [lsearch $zahlen $num] \
        [lsearch $zahlen $num]]
      $win delete $num-line1 $num-line2
  }; # else Zahl schon gesetzt oder schon 6 Zahlen
  .t.anzeige configure -text "Ihre Zahlen: $zahlen"

  if {[__llength $zahlen__] == 6} { .t.send configure -state normal
  } else {.t.send configure -state disabled}
};# setze
```

```
frame .t
pack .t
canvas ___.t.spiel___ -width 150 -height 150
pack ___.t.spiel___ -expand yes -fill both


set x0 5; set y0 5; set x1 25; set y1 25
for {set i 1} {$i <=7} {incr i} {
  for {set j 1} {$j <=7} {incr j} {
    .t.spiel create ___rectangle___ $x0 $y0 $x1 $y1 -outline red
    set zahl [expr ($i-1)*7 + $j]
    .t.spiel create text [expr $x0 + 10] [expr $y0 + 10] \
      -anchor c -text $zahl -fill red
    # leeres Rechteck für Sensor
    .t.spiel create rectangle $x0 $y0 $x1 $y1 -outline "" -fill "" \
      -tags [list $zahl-sensor]
    .t.spiel bind $zahl-sensor <ButtonPress-1> \
      [list setze ___.t.spiel___  ___$zahl___]
    incr x0 20; incr x1 20
  }
  set x0 5; set x1 25
  incr y0 20; incr y1 20
}

label .t.anzeige -text "Ihre Zahlen: $zahlen"
pack .t.anzeige -padx 5 -pady 2
button .t.send -text "Zahlen absenden" -state disabled \
    -command {exit}
pack .t.send -side bottom -fill x
```

2 Pkte (a) Was genau passiert, wenn alle sechs Zahlen angekreuzt wurden?

Man kann keinen weiteren Zahlen auswählen, bereits angekreuzte aber noch löschen. Der Send-Knopf wird aktiviert.

2 Pkte (b) Wie wird das Kreuzchen über einer angeklickten Zahl produziert?

Durch zwei sich kreuzende Linien (Canvas-Items)

2 Pkte (c) Wie heißt der Tag, der an das Feld mit der Zahl 7 gebunden ist?

7-sensor

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2009

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 120 Minuten.

Aufgabe	Punkte max.	Punkte	Aufgabe	Punkte max.	Punkte
1a	2		2b - g	2+2+2+2+2+2	
1b	4		2h - k	2+2+2+2	
1c	3		3a	4	
2a	5		3b - d	4+3+5	
Zw.-Summe	14		Summe	14 + 36 = 50	

Zu Aufgabe 1:

Hier ist eine Aufstellung der gängigen Listen-Kommandos.

lindex *list i*
Gibt Listenelement mit Index *i* zurück
ACHTUNG: Das erste Element hat Index 0

llength *list*
Gibt Anzahl der Listenelemente zurück

lrange *list i j*
Gibt die Listenelemente zwischen
Index *i* und *j* (inklusive) als Liste zurück

lappend *listVar arg ?...?*
Fügt neue Elemente an die Liste mit Variablennamen
listVar an

linsert *list i arg*
Fügt vor Position *i* das neue Element *arg* ein. Gibt
die modifizierte Liste zurück

lsearch *?options? list pattern*
Sucht in der Liste nach dem Muster und liefert den Index
des ersten passenden Elements, sonst -1. Die Optionen bestimmen
die Suche, ohne Angabe wird -glob (wie string match) genommen.

lreplace *list first last ?element element ...?*
Ersetzt Elemente der Liste durch neue Elemente und liefert
die neue Liste zurück. *first* und *last* bestimmen den ersten und
letzten Index der zu ersetzenden Elemente. Die optional angegebenen
Elemente treten an deren Stelle. Fehlt diese Angabe, werden die
Elemente zwischen *first* und *last* einfach entfernt.

lsort *?options? list*
Sortiert die Elemente der Liste und liefert eine neue sortierte
Liste zurück. Die Standardvorgabe ist ASCII-Sortierung, eine
numerische Sortierung erreicht man mit der Option -integer.

Aufgabe 1:

Hinweis: Für die folgenden Teilaufgaben (a) - (c) verwenden Sie die Listenkommandos auf der gegenüberliegenden Seite.

(a) Nehmen wir an, eine Liste enthalte die X- und Y-Koordinaten einer Linie, d. h. die ersten beiden Werte bilden den ersten Punkt, das nächste Paar den zweiten Punkt usw. Vergleichen Sie auch die Liste in der Abbildung zu Aufgabe 3. Natürlich muss so eine Liste eine gerade Anzahl an Werten enthalten. Vervollständigen Sie die Prozedur **geradeAnzahl**, die prüft, ob das Argument eine Liste mit einer geraden Anzahl von Werten ist.

```
proc geradeAnzahl {folge} {
  if {[expr ____________________________ % 2] != 0} {
    return 0 } else {
    return 1 }
} ;# proc
```

(b) Vervollständigen Sie die folgende Prozedur **Trenne**, die eine Liste, die als Argument übergeben wird, in zwei Listen **XPunkte** und **YPunkte** mit der offensichtlichen Bedeutung trennt. Wir verwenden – wenig elegant – **XPunkte** und **YPunkte** als globale Variablen.

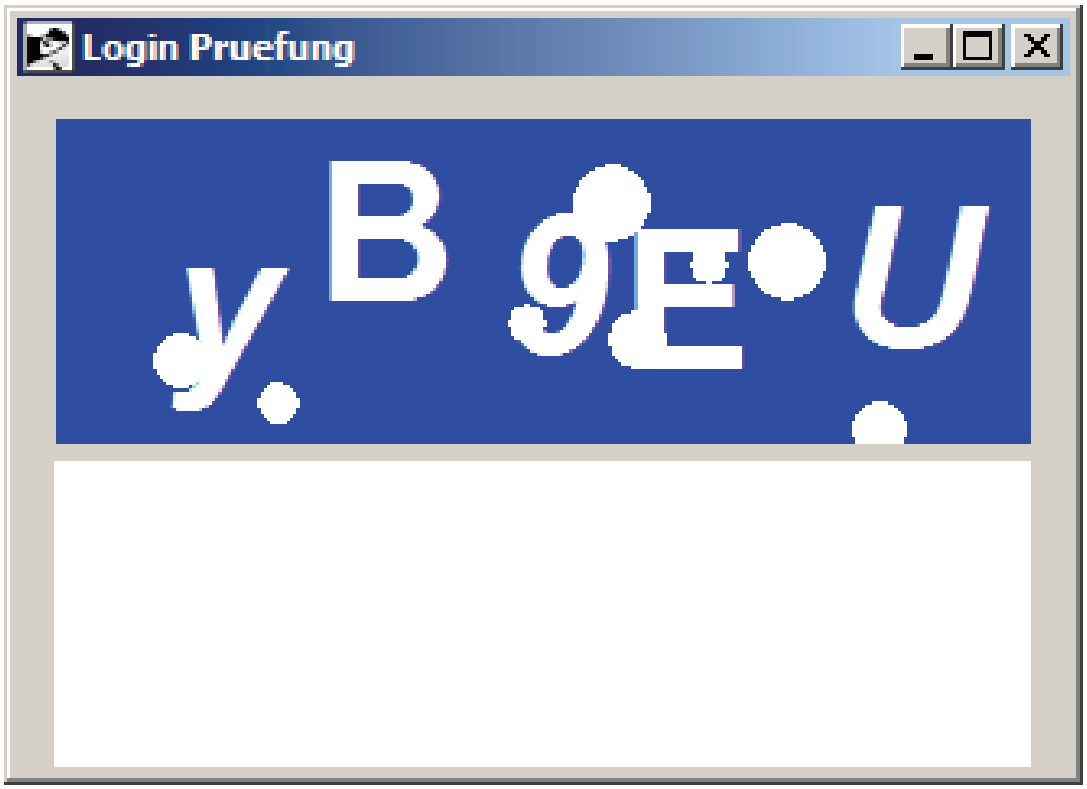
```
proc Trenne {folge} {
  global XPunkte YPunkte
  for {set i 0} {$i < ________________________ } {incr i} {
    if {____________ % 2 == 0} {
      lappend XPunkte ____________________________
    } else {
      lappend YPunkte ____________________________
    }
  };# for
} ;# proc
```

(c) Die **XPunkte**-Liste sollte aufsteigende Werte haben. Prüfen Sie das mit einer Prozedur **sorted**, die genau dann 1 (wahr) liefert, wenn das Argument eine Liste mit aufsteigend angeordneten Werten ist, sonst 0 (false). Vervollständigen Sie die Prozedur!

```
proc sorted {folge} {
  for {set i 0} {$i < [expr [____________________] - 1]} {incr i} {
    if {[_________________ $i] > [____________________ [expr $i + 1]]} {
      return 0 ;#false}
  };# end for
  return 1 ;#true
} ;# proc sorted
```

Aufgabe 2:

Viele Web-Portale, bei denen man sich anmelden muss, schützen sich durch ein sog. CAPTCHA vor bösartigen Angriffen durch „Bots“. Ein CAPTCHA ist eine zufällig ausgewählte Zeichenfolge mit meist 5 Zeichen, die so verfremdet angezeigt wird, dass sie in der Regel nur ein Mensch lesen kann. Der Anwender muss diese Zeichenfolge dann eingeben, worauf das Anmeldeprogramm die Übereinstimmung der Eingabe mit dem vorgegebenen Codewort prüft.



(a) Hier ist das Programm, mit dem wir Fenster der obigen Art erzeugen (der richtige Code wäre yB9EU). Der untere Teil ist ausgeblendet. Zeichnen Sie die fehlenden Teile ein!

```
#!/usr/bin/wish
wm title . "Login Pruefung"
frame .f
pack .f
set cwidth 300
set cheight 100
canvas .f.c -width $cwidth -height $cheight -background blue
pack .f.c -padx 10 -pady 10
label .f.aufforderung -text "Bitte abgelesenen Code eingeben!"
pack .f.aufforderung
entry .f.eingabe -textvariable yourcode -bg white -fg black -width 7
pack .f.eingabe -padx 5 -pady 5
focus .f.eingabe
frame .f.buttons
pack .f.buttons -padx 10 -pady 10
button .f.buttons.nochmal -text "Neuer Code" -command {
   .f.c delete all; captcha_create .f.c }
pack .f.buttons.nochmal -side left -padx 5
button .f.buttons.quit -text "Abbrechen" -command {exit}
```

```
pack .f.buttons.quit -side left -padx 5
button .f.buttons.send -text "Abschicken" -command {
  if {$codestring == $yourcode} {
    message .m -width 400 -justify center \
      -padx 20 -bg green -fg white \
      -text "Codeeingabe korrekt"
    pack .m -padx 20 -pady 20
    after 2000 exit
  } else {
    .f.aufforderung configure \
      -text "Eingabe falsch - bitte nochmal!"
    set yourcode ""
    focus .f.eingabe
    .f.c delete all; captcha_create .f.c }
  }
pack .f.buttons.send -side left -padx 5

set alphabet {
  a b c d e f g h i j
  k m n p q r s t u v
  w x y z A B C D E F
  G H I J K L M N P Q
  R S T U V W X Y Z 2
  3 4 5 6 7 8 9 + = % }
set codestring ""

proc captcha_create {cw} {
  global alphabet codestring
  set codestring ""
  for {set i 0} {$i <=4} {incr i} {
    set index [expr int(rand() * 60)]
    set glyph [lindex $alphabet $index]
    set codestring $codestring$glyph
    set xoffset [expr int(rand() * 30) - 15]
    set yoffset [expr int(rand() * 40) - 20]
    if {[expr $i % 2] == 0} {set style "i"} else {set style "r"}
    $cw create text [expr 50 + 50*$i + $xoffset] [expr 50 + $yoffset] \
      -text $glyph -fill white \
      -font -*-helvetica-bold-$style-normal-*-*-600-*
  } ;# end for

  for {set j 0} {$j <= 7} {incr j} { # polka dots
    set r [expr 10 + int(rand() * 15)]
    set xrand [expr int(rand() * 280) + 10]
    set yrand [expr int(rand() * 80) + 10]
    $cw create oval $xrand $yrand [expr $xrand + $r] \
      [expr $yrand + $r] -fill white -outline ""
  } ;# end for
};# end proc captcha_create
captcha_create .f.c
```

Hinweis: Die Funktion **`rand()`** liefert eine Gleitkommazahl x mit $0 \leq x < 1$. Die Funktion **`int()`** schneidet Nachkommastellen ab und liefert einen Integer.

(b) Nach welcher Regel erzeugt das Programm ein **kursives** Zeichen statt eines **nicht-kursiven** Zeichens?

(c) Um wieviele Pixel differiert die tiefste von der höchsten Grundlinie eines Zeichens?

(d) Wieviele Kreise werden mindestens, wieviele höchstens gezeichnet?

(e) Was ist der kleinste, was der größte Durchmesser der Kreise zum Verfremden des Bildes?

(f) Was passiert genau, wenn nach Druck von „Abschicken“ der eingegebene Code mit dem angezeigten übereinstimmt.

(g) Was passiert genau, wenn nach Druck von „Abschicken“ der eingegebene Code **nicht** mit dem angezeigten übereinstimmt?

(h) Wie wird im Fall (g) erreicht, dass das Eingabefeld zurückgesetzt wird. Geben Sie die Codezeile dazu an!

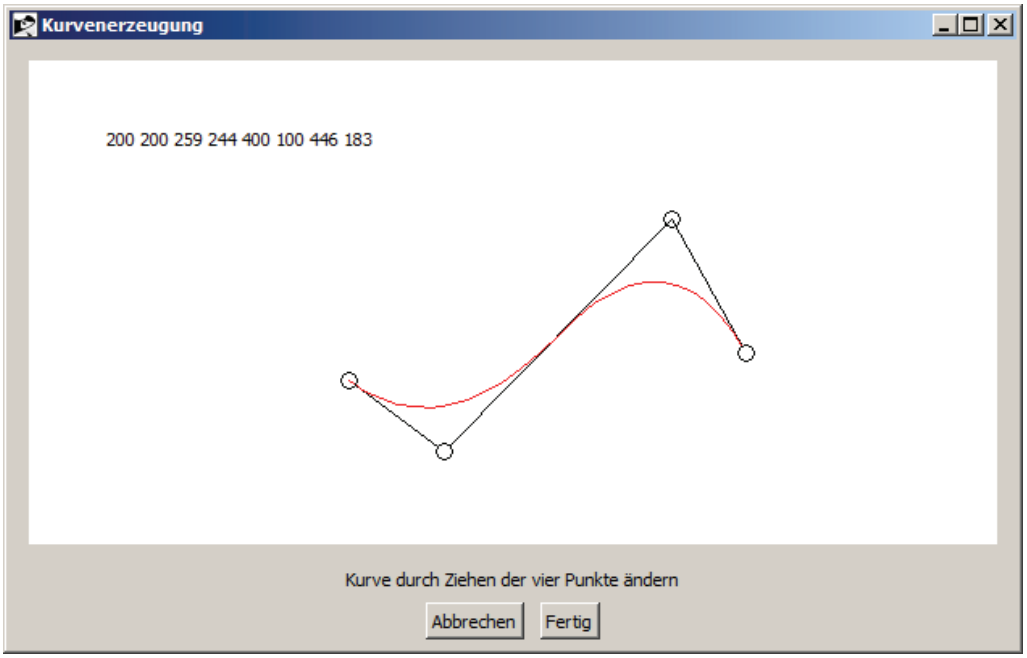
(i) Nennen Sie mindestens zwei im üblichen Alphabet vorhandene Zeichen, die im Zeichenvorrat des Programms ausgelassen wurden. Warum fehlen diese wohl?

(j) Was kann ein Anwender machen, wenn der angezeigte Code beim besten Willen nicht lesbar ist, weil die Kreise so ungünstig darüberliegen?

(k) Werden zuerst die Codezeichen und dann die Kreise gemalt, oder umgekehrt? Spielt das in dieser Anwendung eine Rolle? Begründung!

Aufgabe 3:

Wir basteln uns einen einfachen, interaktiven Kurveneditor. Hierzu gibt es das Line-Item mit der Option „-smooth bezier“. Für diese Klausur interessiert aber nur die Technik der Bindings.




```
#!/usr/bin/wish
wm title . "Kurvenerzeugung"
frame .f
pack .f
set cwidth 600
set cheight 300
canvas .f.c -width $cwidth -height $cheight -background white
pack .f.c -padx 10 -pady 10
label .f.aufforderung \
   -text "Kurve durch Ziehen der vier Punkte ändern"
pack .f.aufforderung
frame .f.b
button .f.b.cancel -text Abbrechen -command {exit}
button .f.b.ok -text Fertig -command {exit}
pack .f.b
pack .f.b.cancel -padx 5 -pady 5 -side left
pack .f.b.ok -padx 5 -pady 5 -side left

proc BewegePunkt {P Xneu Yneu} {
  global XP YP PID
  set XP($P) $Xneu; set YP($P) $Yneu
  .f.c coords thecurve "$XP(P0) $YP(P0) $XP(P1) $YP(P1) \
    $XP(P2) $YP(P2) $XP(P3) $YP(P3)"
  .f.c coords HelpLine "$XP(P0) $YP(P0) $XP(P1) $YP(P1) \
    $XP(P2) $YP(P2) $XP(P3) $YP(P3)"
  .f.c coords $P [expr $Xneu - 5] [expr $Yneu - 5] \
   [expr $Xneu + 5] [expr $Yneu + 5]
}
```

```
#Anfangskoordinaten
set XP(P0) 200; set YP(P0) 200
set XP(P1) 200; set YP(P1) 100
set XP(P2) 400; set YP(P2) 100
set XP(P3) 400; set YP(P3) 200
.f.c create line $XP(P0) $YP(P0) $XP(P1) $YP(P1) \
  $XP(P2) $YP(P2) $XP(P3) $YP(P3) -tag HelpLine

.f.c create line "$XP(P0) $YP(P0) $XP(P1) $YP(P1) \
    $XP(P2) $YP(P2) $XP(P3) $YP(P3)" \
  -smooth bezier -fill red -tag thecurve

for {set i 0} {$i <=3} {incr i} {
  set thisx $XP(P$i); set thisy $YP(P$i)
  set currentID [.f.c create oval [expr $thisx - 5] \
  [expr $thisy - 5] [expr $thisx + 5] [expr $thisy + 5] \
  -outline black -fill "" -tag "P$i Punkt"]
  set PID($currentID) P$i
}; # for i

.f.c bind Punkt <Enter> {
    .f.c itemconfigure [.f.c find withtag current] -fill blue}

.f.c bind Punkt <Leave> {
    .f.c itemconfigure [.f.c find withtag current] -fill white}

.f.c bind Punkt <B1-Motion> {
    set ThisP $PID([.f.c find withtag current])
    BewegePunkt $ThisP %x %y}

.f.c bind Punkt <ButtonRelease-1> {
    set ThisP $PID([.f.c find withtag current])
    BewegePunkt $ThisP %x %y}
```

(a) Ergänzen Sie das Programm oben so, dass die Punkte durch das Anklicken mit der linken Maustaste und während des Ziehens mit gedrückter Maustaste eine rote Füllung erhalten, die sich nach dem Loslassen der linken Maustaste wieder zu blau ändert und beim Verlassen des Punkts zu weiß zurückwechselt. Hinweis1: Sie brauchen dazu ein neues Binding und eine kleine Ergänzung zum Ereignis **<ButtonRelease-1>**. Hinweis2: **find withtag current** ist eine Canvas-Methode, die den ItemId des obersten Items liefert, in dessen Fläche der Mauszeiger momentan ist.

(b) Im Moment bewirkt das Drücken des Abbrechen- oder Fertig-Knopfs nur das sofortige Programmende. Ändern Sie das Verhalten des Fertig-Knopfs dahingehend, dass zunächst die Hilfslinie und die vier kleinen Kreise für die Punkte verschwinden und erst dann 5 Sekunden später das Programm ganz aufhört.

```
button .f.b.ok -text Fertig -command {




}
```

(c) Ergänzen Sie das Programmfragment zur Wandlung der vier Koordinatenpaare in **`XP(P0)`**, **`YP(P0)`**, ..., **`XP(P3)`**, **`YP(P3)`** in eine Koordinatenliste **`pstring`**.

```
set pstring ""
for {set j 0} {$j <= 3} {incr j} {
    lappend ____________________________________________
}
```

(d) In der linken oberen Ecke der Leinwand in der Abbildung oben sieht man ein Textitem mit den vier X- und Y-Koordinaten als Liste. Der zugehörige Programmcode zur Erzeugung fehlt. Ergänzen Sie ihn!

Hinweis: Legen Sie das Textitem auf der Leinwand zunächst mit den Anfangskoordinaten als Text an. In der Prozedur **`BewegePunkt`** rufen Sie für dieses Textitem ein **`itemconfigure`** mit dem neuen Text (den aktuellen Koordinaten) auf.

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2009

Nachname: Vorname:

Matr.Nr.: Studiengang:

MUSTERLÖSUNG

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 120 Minuten.

Aufgabe	Punkte max.	Punkte	Aufgabe	Punkte max.	Punkte
1a	2		2b - g	2+2+2+2+2+2	
1b	4		2h - k	2+2+2+2	
1c	3		3a	4	
2a	5		3b - d	4+3+5	
Zw.-Summe	14		Summe	14 + 36 = 50	

Zu Aufgabe 1:

Hier ist eine Aufstellung der gängigen Listen-Kommandos.

lindex *list i*
Gibt Listenelement mit Index *i* zurück
ACHTUNG: Das erste Element hat Index 0

llength *list*
Gibt Anzahl der Listenelemente zurück

lrange *list i j*
Gibt die Listenelemente zwischen
Index *i* und *j* (inklusive) als Liste zurück

lappend *listVar arg ?...?*
Fügt neue Elemente an die Liste mit Variablennamen
listVar an

linsert *list i arg*
Fügt vor Position *i* das neue Element *arg* ein. Gibt
die modifizierte Liste zurück

lsearch *?options? list pattern*
Sucht in der Liste nach dem Muster und liefert den Index
des ersten passenden Elements, sonst -1. Die Optionen bestimmen
die Suche, ohne Angabe wird -glob (wie string match) genommen.

lreplace *list first last ?element element ...?*
Ersetzt Elemente der Liste durch neue Elemente und liefert
die neue Liste zurück. *first* und *last* bestimmen den ersten und
letzten Index der zu ersetzenden Elemente. Die optional angegebenen
Elemente treten an deren Stelle. Fehlt diese Angabe, werden die
Elemente zwischen *first* und *last* einfach entfernt.

lsort *?options? list*
Sortiert die Elemente der Liste und liefert eine neue sortierte
Liste zurück. Die Standardvorgabe ist ASCII-Sortierung, eine
numerische Sortierung erreicht man mit der Option -integer.

Aufgabe 1:

Hinweis: Für die folgenden Teilaufgaben (a) - (c) verwenden Sie die Listenkommandos auf der gegenüberliegenden Seite.

(a) Nehmen wir an, eine Liste enthalte die X- und Y-Koordinaten einer Linie, d. h. die ersten beiden Werte bilden den ersten Punkt, das nächste Paar den zweiten Punkt usw. Vergleichen Sie auch die Liste in der Abbildung zu Aufgabe 3. Natürlich muss so eine Liste eine gerade Anzahl an Werten enthalten. Vervollständigen Sie die Prozedur **geradeAnzahl**, die prüft, ob das Argument eine Liste mit einer geraden Anzahl von Werten ist.

```
proc geradeAnzahl {folge} {
  if {[expr [llength $folge] % 2] != 0} {
    return 0 } else {
    return 1 }
} ;# proc
```

(b) Vervollständigen Sie die folgende Prozedur **Trenne**, die eine Liste, die als Argument übergeben wird, in zwei Listen **XPunkte** und **YPunkte** mit der offensichtlichen Bedeutung trennt. Wir verwenden – wenig elegant – **XPunkte** und **YPunkte** als globale Variablen.

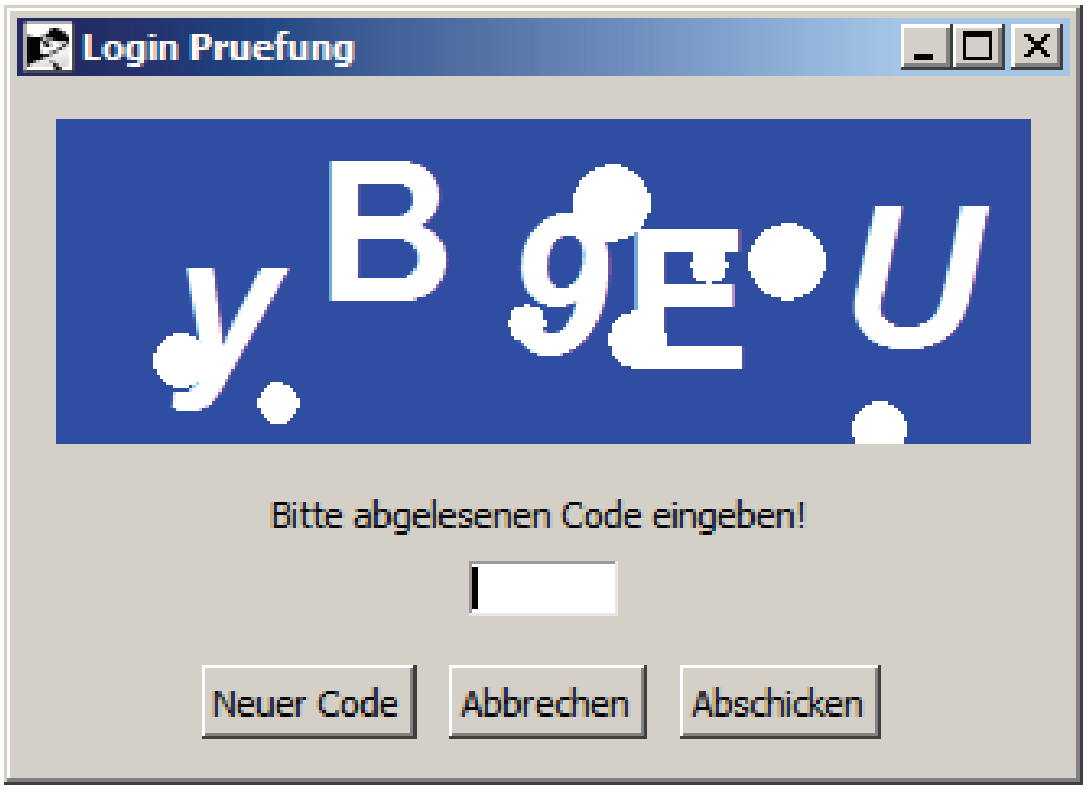
```
proc Trenne {folge} {
  global XPunkte YPunkte
  for {set i 0} {$i < [llength $folge] } {incr i} {
    if {  $i  % 2 == 0} {
      lappend XPunkte [lindex $folge $i]
    } else {
      lappend YPunkte [lindex $folge $i]
    }
  };# for
} ;# proc
```

(c) Die **XPunkte**-Liste sollte aufsteigende Werte haben. Prüfen Sie das mit einer Prozedur **sorted**, die genau dann 1 (wahr) liefert, wenn das Argument eine Liste mit aufsteigend angeordneten Werten ist, sonst 0 (false). Vervollständigen Sie die Prozedur!

```
proc sorted {folge} {
  for {set i 0} {$i < [expr [ llength $folge ] - 1]} {incr i} {
    if {[ lindex $folge $i] > [ lindex $folge [expr $i + 1]]} {
      return 0 ;#false}
  };# end for
  return 1 ;#true
} ;# proc sorted
```

Aufgabe 2:

Viele Web-Portale, bei denen man sich anmelden muss, schützen sich durch ein sog. CAPTCHA vor bösartigen Angriffen durch „Bots“. Ein CAPTCHA ist eine zufällig ausgewählte Zeichenfolge mit meist 5 Zeichen, die so verfremdet angezeigt wird, dass sie in der Regel nur ein Mensch lesen kann. Der Anwender muss diese Zeichenfolge dann eingeben, worauf das Anmeldeprogramm die Übereinstimmung der Eingabe mit dem vorgegebenen Codewort prüft.



(a) Hier ist das Programm, mit dem wir Fenster der obigen Art erzeugen (der richtige Code wäre yB9EU). Der untere Teil ist ausgeblendet. Zeichnen Sie die fehlenden Teile ein!

```
#!/usr/bin/wish
wm title . "Login Pruefung"
frame .f
pack .f
set cwidth 300
set cheight 100
canvas .f.c -width $cwidth -height $cheight -background blue
pack .f.c -padx 10 -pady 10
label .f.aufforderung -text "Bitte abgelesenen Code eingeben!"
pack .f.aufforderung
entry .f.eingabe -textvariable yourcode -bg white -fg black -width 7
pack .f.eingabe -padx 5 -pady 5
focus .f.eingabe
frame .f.buttons
pack .f.buttons -padx 10 -pady 10
button .f.buttons.nochmal -text "Neuer Code" -command {
   .f.c delete all; captcha_create .f.c }
pack .f.buttons.nochmal -side left -padx 5
button .f.buttons.quit -text "Abbrechen" -command {exit}
```

```
pack .f.buttons.quit -side left -padx 5
button .f.buttons.send -text "Abschicken" -command {
  if {$codestring == $yourcode} {
    message .m -width 400 -justify center \
      -padx 20 -bg green -fg white \
      -text "Codeeingabe korrekt"
    pack .m -padx 20 -pady 20
    after 2000 exit
  } else {
    .f.aufforderung configure \
      -text "Eingabe falsch - bitte nochmal!"
    set yourcode ""
    focus .f.eingabe
    .f.c delete all; captcha_create .f.c }
  }
pack .f.buttons.send -side left -padx 5

set alphabet {
  a b c d e f g h i j
  k m n p q r s t u v
  w x y z A B C D E F
  G H I J K L M N P Q
  R S T U V W X Y Z 2
  3 4 5 6 7 8 9 + = % }
set codestring ""

proc captcha_create {cw} {
  global alphabet codestring
  set codestring ""
  for {set i 0} {$i <=4} {incr i} {
    set index [expr int(rand() * 60)]
    set glyph [lindex $alphabet $index]
    set codestring $codestring$glyph
    set xoffset [expr int(rand() * 30) - 15]
    set yoffset [expr int(rand() * 40) - 20]
    if {[expr $i % 2] == 0} {set style "i"} else {set style "r"}
    $cw create text [expr 50 + 50*$i + $xoffset] [expr 50 + $yoffset] \
      -text $glyph -fill white \
      -font -*-helvetica-bold-$style-normal-*-*-600-*
  } ;# end for

  for {set j 0} {$j <= 7} {incr j} { # polka dots
    set r [expr 10 + int(rand() * 15)]
    set xrand [expr int(rand() * 280) + 10]
    set yrand [expr int(rand() * 80) + 10]
    $cw create oval $xrand $yrand [expr $xrand + $r] \
      [expr $yrand + $r] -fill white -outline ""
  } ;# end for
};# end proc captcha_create
captcha_create .f.c
```

Hinweis: Die Funktion **`rand()`** liefert eine Gleitkommazahl x mit $0 \leq x < 1$. Die Funktion **`int()`** schneidet Nachkommastellen ab und liefert einen Integer.

(b) Nach welcher Regel erzeugt das Programm ein **kursives** Zeichen statt eines **nicht-kursiven** Zeichens?

Jedes zweite Zeichen erscheint kursiv, beginnend mit dem ersten an Position 0.

(c) Um wieviele Pixel differiert die tiefste von der höchsten Grundlinie eines Zeichens?

Um 39 Pixel (auch als richtig gewertet: 40 Pixel)

(d) Wieviele Kreise werden mindestens, wieviele höchstens gezeichnet?

Es werden genau 8 Kreise gemalt.

(e) Was ist der kleinste, was der größte Durchmesser der Kreise zum Verfremden des Bildes?

Der Durchmesser variiert von 10 bis 24 Pixel (auch als richtig gewertet: bis 25 Pixel).

(f) Was passiert genau, wenn nach Druck von „Abschicken“ der eingegebene Code mit dem angezeigten übereinstimmt.

Es erscheint „Codeeingabe korrekt" zentriert unterhalb des Buttons. Der Text ist weiß auf grünem Hintergrund. Nach 2s beendet sich das Programm selbst.

(g) Was passiert genau, wenn nach Druck von „Abschicken“ der eingegebene Code **nicht** mit dem angezeigten übereinstimmt?

Der Text „Bitte abgelesenen Code eingeben!" ändert sich zu „Eingabe falsch - bitte nochmal!". Die Eingabemaske (entry) wird gelöscht und fokussiert. Die Leinwand (canvas) wird gelöscht und ein neues Captcha wird generiert.

(h) Wie wird im Fall (g) erreicht, dass das Eingabefeld zurückgesetzt wird. Geben Sie die Codezeile dazu an!

set yourcode ""

(i) Nennen Sie mindestens zwei im üblichen Alphabet vorhandene Zeichen, die im Zeichenvorrat des Programms ausgelassen wurden. Warum fehlen diese wohl?

Es fehlen z. B. der Buchstabe O und die Ziffer Null, auch das kleine l und die Ziffer 1, da diese Paare sich zu ähnlich sehen und damit unnötige Lesefehler erzeugen.

(j) Was kann ein Anwender machen, wenn der angezeigte Code beim besten Willen nicht lesbar ist, weil die Kreise so ungünstig darüberliegen?

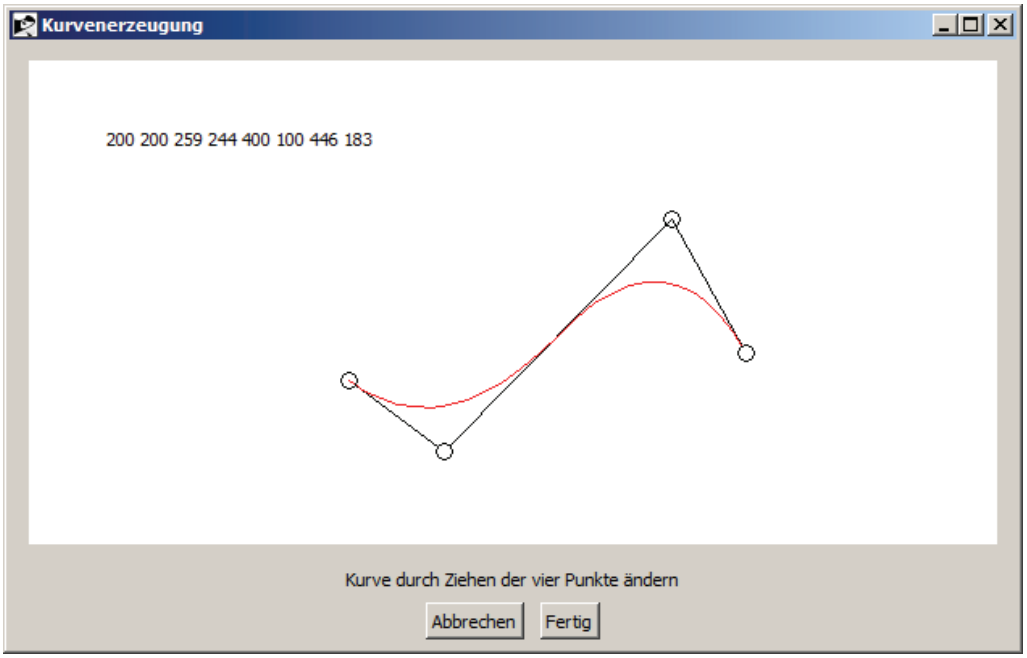
Durch Klicken des „Neuer Code"-Knopfes kann er sich ein neues Captcha erzeugen lassen.

(k) Werden zuerst die Codezeichen und dann die Kreise gemalt, oder umgekehrt? Spielt das in dieser Anwendung eine Rolle? Begründung!

Es werden zuerst die Zeichen und dann die Kreise gemalt. Da beide in gleicher Farbe erscheinen, spielt die Reihenfolge keine Rolle. Bei unterschiedlicher Farbe müssen die Codezeichen aber immer zuerst gezeichnet werden.

Aufgabe 3:

Wir basteln uns einen einfachen, interaktiven Kurveneditor. Hierzu gibt es das Line-Item mit der Option „-smooth bezier“. Für diese Klausur interessiert aber nur die Technik der Bindings.




```
#!/usr/bin/wish
wm title . "Kurvenerzeugung"
frame .f
pack .f
set cwidth 600
set cheight 300
canvas .f.c -width $cwidth -height $cheight -background white
pack .f.c -padx 10 -pady 10
label .f.aufforderung \
   -text "Kurve durch Ziehen der vier Punkte ändern"
pack .f.aufforderung
frame .f.b
button .f.b.cancel -text Abbrechen -command {exit}
button .f.b.ok -text Fertig -command {exit}
pack .f.b
pack .f.b.cancel -padx 5 -pady 5 -side left
pack .f.b.ok -padx 5 -pady 5 -side left

proc BewegePunkt {P Xneu Yneu} {
  global XP YP PID
  set XP($P) $Xneu; set YP($P) $Yneu
  .f.c coords thecurve "$XP(P0) $YP(P0) $XP(P1) $YP(P1) \
    $XP(P2) $YP(P2) $XP(P3) $YP(P3)"
  .f.c coords HelpLine "$XP(P0) $YP(P0) $XP(P1) $YP(P1) \
    $XP(P2) $YP(P2) $XP(P3) $YP(P3)"
  .f.c coords $P [expr $Xneu - 5] [expr $Yneu - 5] \
   [expr $Xneu + 5] [expr $Yneu + 5]
}
```

```
#Anfangskoordinaten
set XP(P0) 200; set YP(P0) 200
set XP(P1) 200; set YP(P1) 100
set XP(P2) 400; set YP(P2) 100
set XP(P3) 400; set YP(P3) 200
.f.c create line $XP(P0) $YP(P0) $XP(P1) $YP(P1) \
  $XP(P2) $YP(P2) $XP(P3) $YP(P3) -tag HelpLine

.f.c create line "$XP(P0) $YP(P0) $XP(P1) $YP(P1) \
    $XP(P2) $YP(P2) $XP(P3) $YP(P3)" \
  -smooth bezier -fill red -tag thecurve

for {set i 0} {$i <=3} {incr i} {
  set thisx $XP(P$i); set thisy $YP(P$i)
  set currentID [.f.c create oval [expr $thisx - 5] \
  [expr $thisy - 5] [expr $thisx + 5] [expr $thisy + 5] \
  -outline black -fill "" -tag "P$i Punkt"]
  set PID($currentID) P$i
}; # for i

.f.c bind Punkt <Enter> {
    .f.c itemconfigure [.f.c find withtag current] -fill blue}

.f.c bind Punkt <Leave> {
    .f.c itemconfigure [.f.c find withtag current] -fill white}

.f.c bind Punkt <B1-Motion> {
    set ThisP $PID([.f.c find withtag current])
    BewegePunkt $ThisP %x %y}

.f.c bind Punkt <ButtonRelease-1> {
    set ThisP $PID([.f.c find withtag current])
    BewegePunkt $ThisP %x %y ;# Klammer hier weg
    .f.c itemconfigure $ThisP -fill blue}


.f.c bind Punkt <ButtonPress-1> {
    .f.c itemconfigure [.f.c find withtag current] -fill red}
```

(a) Ergänzen Sie das Programm oben so, dass die Punkte durch das Anklicken mit der linken Maustaste und während des Ziehens mit gedrückter Maustaste eine rote Füllung erhalten, die sich nach dem Loslassen der linken Maustaste wieder zu blau ändert und beim Verlassen des Punkts zu weiß zurückwechselt. Hinweis1: Sie brauchen dazu ein neues Binding und eine kleine Ergänzung zum Ereignis **<ButtonRelease-1>**. Hinweis2: **find withtag current** ist eine Canvas-Methode, die den ItemId des obersten Items liefert, in dessen Fläche der Mauszeiger momentan ist.

(b) Im Moment bewirkt das Drücken des Abbrechen- oder Fertig-Knopfs nur das sofortige Programmende. Ändern Sie das Verhalten des Fertig-Knopfs dahingehend, dass zunächst die Hilfslinie und die vier kleinen Kreise für die Punkte verschwinden und erst dann 5 Sekunden später das Programm ganz aufhört.

```
button .f.b.ok -text Fertig -command {
    .f.c delete HelpLine
    .f.c delete Punkt
    after 5000 exit
}
```

(c) Ergänzen Sie das Programmfragment zur Wandlung der vier Koordinatenpaare in **XP(P0)**, **YP(P0)**, ..., **XP(P3)**, **YP(P3)** in eine Koordinatenliste **pstring**.

```
set pstring ""
for {set j 0} {$j <= 3} {incr j} {
    lappend pstring $XP(P$j) $YP(P$j)
}
```

(d) In der linken oberen Ecke der Leinwand in der Abbildung oben sieht man ein Textitem mit den vier X- und Y-Koordinaten als Liste. Der zugehörige Programmcode zur Erzeugung fehlt. Ergänzen Sie ihn!

Hinweis: Legen Sie das Textitem auf der Leinwand zunächst mit den Anfangskoordinaten als Text an. In der Prozedur **BewegePunkt** rufen Sie für dieses Textitem ein **itemconfigure** mit dem neuen Text (den aktuellen Koordinaten) auf.

```
.f.c create text 10 10 -text "$XP(P0) $YP(P0) $XP(P1) $YP(P1) \
      $XP(P2) $YP(P2) $XP(P3) $YP(P3)" -tag coords -anchor w
```

in BewegePunkt:

```
    .f.c itemconfigure coords -text "$XP(P0) $YP(P0) $XP(P1) $YP(P1) \
         $XP(P2) $YP(P2) $XP(P3) $YP(P3)"
```

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2010

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

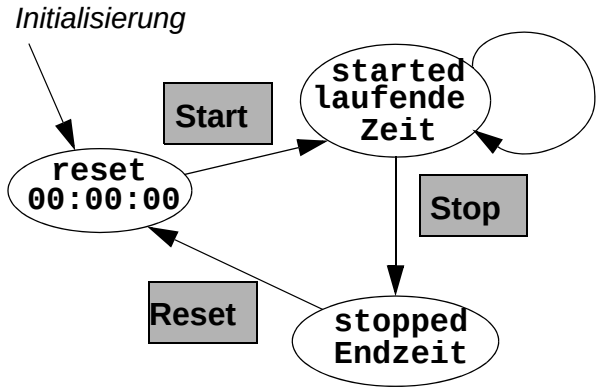
Aufgabe	Punkte maximal	Punkte erreicht
1	12	
2	4+2+2+2	
3	4	
4	2+2+2+4+4	
Summe	40	

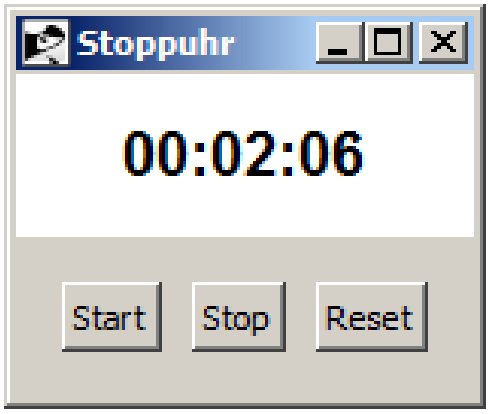
Aufgabe 1:

Wir basteln uns eine Stoppuhr. Diese hat, wie gezeigt, eine Anzeige (als Label realisiert) und darunter drei Knöpfe für Start, Stop und Reset (Rücksetzen). Zu Beginn zeigt die Anzeige 00:00:00, wird Start gedrückt, merken wir uns die Systemuhrzeit als Startzeit und die Anzeige läuft. Angezeigt wird dann der Wert „momentane Uhrzeit“ minus Startzeit. Wird bei laufender Uhr Stop gedrückt, halten wir die Endzeit fest und als Anzeige bleibt Endzeit - Startzeit stehen. Nur mit Reset springt die Anzeige zurück nach Null und es kann wieder Start gedrückt werden.

Die eigentliche Zeitmessung regeln wir über die Systemuhrzeit (**`clock seconds`** liefert die Unix-Zeit in Sekunden). Für die Formatierung in Stunden:Minuten:Sekunden können wir das **`clock format`** Kommando nutzen (**`-gmt true`** stellt sicher, dass wir keine Stunde für die Ortszeit hinzuaddiert bekommen).

Die Uhr kennt drei Zustände: **`reset, started, stopped`**. Die Uhr wird im Zustand **`reset`** initialisiert. Die folgenden Übergänge zeigt das Diagramm.





Fügen Sie den fehlenden Reset-Knopf ein und ergänzen Sie alle Lücken im Programmtext.

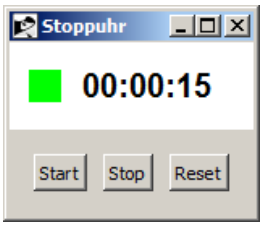
```
#!/usr/bin/wish
wm title . "Stoppuhr"
#oben ein Label zur Zeitanzeige
#darunter ein Start-, Stop- und Resetknopf

set state reset
frame .f -padx 10 -pady 10 -bg white
pack .f -expand yes -fill both
frame .g -padx 10 -pady 10
pack .g -expand yes -fill both
label .f.zeit -text "00:00:00" -bg white -font "Arial 15 bold"
pack .f.zeit -side left -expand yes

button .g.start -text "Start" -command {
  if {$state == "reset"} {
    set stime [clock seconds]
    set state "started"
    ShowTime
  }; #else ignore button press
}
```

```
pack .g.start -side left -padx 5 -pady 5 -expand yes -fill both

button .g.stop -text "Stop" -command {
  if {$state == "started"} {
    set etime [clock seconds]
    set state "stopped"
    ShowTime; #could be omitted
  }; #else ignore button press
}
pack .g.stop -side left -padx 5 -pady 5 -expand yes -fill both

# hier den Reset-Knopf hin
```

```
proc ShowTime {} {
  global stime etime state

  switch $state {
    "reset" {set s ____________________}

    "stopped" {set s [_________________________________________]}

    "started" {set s [_________________________________________]
        after 100 ShowTime }
  }
  .f.zeit configure -text [clock format $s -format %T -gmt true]
}
```

Aufgabe 2:

In die Stoppuhr aus Aufgabe 1 kommt vor die Anzeige der Zeit ein Rechteck (als Frame **`.f.square`**) mit genau 20 Pixel Höhe und Breite. Die Füllfarbe soll grau (**`grey`**) sein im Zustand **`reset`**, grün (**`green`**) im Zustand **`started`** und rot (**`red`**) im Zustand **`stopped`** (vgl. Abbildung). Markieren Sie im Programm von Aufgabe 1 am Rand die Einfügungen als Nummer (1), (2), ... und geben Sie die Programmstücke unten an.



Einfügung/Änderung (1)

Einfügung/Änderung (2)

Einfügung/Änderung (3)

Einfügung/Änderung (4)

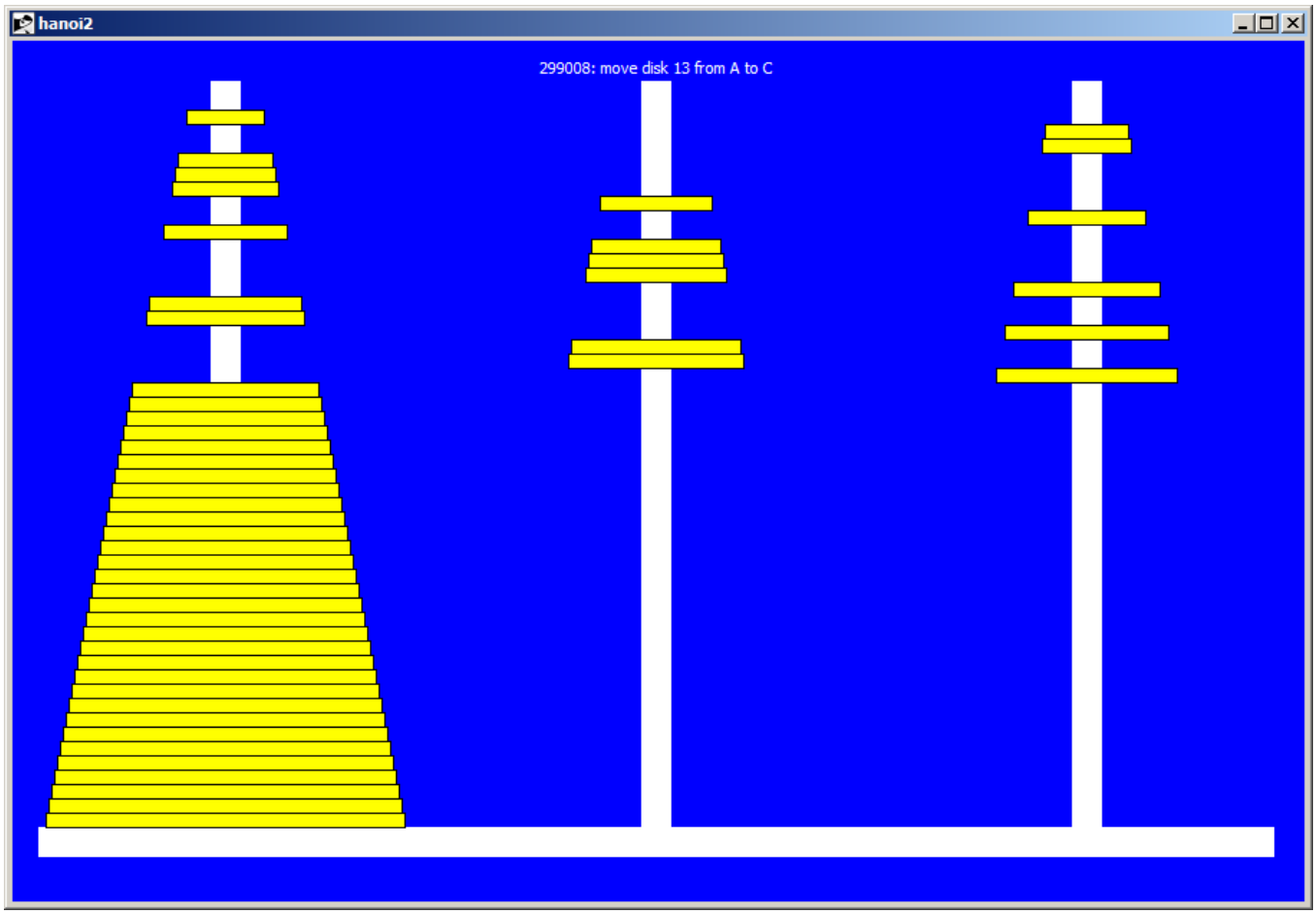
Aufgabe 3:

Betrachten Sie das folgende Programmstück. Was zeigt das Label **`.f.zeit`** an, wenn **`s`** den Wert 3666 hat?

```
set m [expr ($s % 3600) / 60]
set sec [expr $s % 60]
.f.zeit configure \
  -text "[expr $s/3600]:[expr {$m<10?0:""}]$m:[expr {$sec<10?0:""}]$sec"
```

Aufgabe 4:

Die *Türme von Hanoi* sind ein Spiel, bei dem es gilt, eine Pyramide von *n* Scheiben von der Stange *A* links auf die mittlere Stange *B* unter Zuhilfenahme der dritten Stange *C* umzuschichten. Dabei darf nie eine Scheibe auf einer kleineren zu liegen kommen.



Die Aufgabe gilt als Paradebeispiel für eine rekursive Lösung, weil man *n* Scheiben von *A* nach *B* bringt, indem man die obersten *n*-1 Scheiben von *A* nach *C* bringt, dann die *n*-te Scheibe von *A* nach *B*, worauf die *n*-1 Scheiben von *C* nach *B* zurück gehen. Die Unteraufgabe des Umschichtens der *n*-1 Scheiben löst man nach der selben Methode, d. h. die Stapel *A*, *B*, und *C* wechseln ihre Rolle als Quelle (*von*), Ziel (*to*) und Zwischenspeicher (*via*).

Das Bild oben zeigt unsere Lösung in Tcl/Tk, wobei zur besseren (und einfacheren) Visualisierung die Lücken in den Stapeln nicht geschlossen werden, die Scheiben sich also nur waagrecht bewegen. Es folgt der Quellcode unseres Programms. Beantworten Sie die danach folgenden Fragen, bzw. lösen Sie die Aufgaben.

```
canvas .c -width 900 -height 600 -background blue
pack .c

.c create rectangle 140 30 160 550 -fill white -outline white
.c create rectangle 440 30 460 550 -fill white -outline white
.c create rectangle 740 30 760 550 -fill white -outline white
.c create rectangle 20 550 880 570 -fill white -outline white

set numdiscs 50
```

```
for {set i $numdiscs} {$i>0} {incr i -1} {
  .c create rectangle [expr 25 + ($numdiscs - $i)*2] \
    [expr 540 - 10*($numdiscs - $i)] \
    [expr 275 - ($numdiscs - $i)*2] \
    [expr 550 - 10*($numdiscs - $i)] \
    -fill yellow -tag disc$i
}

.c create text 450 20 -tag banner -fill white -text ""

set count 1
proc hanoi { n {from A} {to B} {via C} } {
  global count numdiscs
  if {$n > 0} {
    hanoi [expr $n-1] $from $via $to
  # puts "$count: move from $from to $to"
  if {$n > 10} {
    .c itemconfigure banner \
      -text "$count: move disk $n from $from to $to"
  }; #show only for bigger disks
  incr count
  set nn [expr $numdiscs - $n]
  switch $to {
  A {.c coords disc$n [expr 25 + $nn*2] \
    [expr 540 - 10*$nn] \
    [expr 275 - $nn*2] \
    [expr 550 - 10*$nn]
    }
  B {.c coords disc$n [expr 325 + $nn*2] \
    [expr 540 - 10*$nn] \
    [expr 575 - $nn*2] \
    [expr 550 - 10*$nn]
    }
  C {.c coords disc$n [expr 625 + $nn*2] \
    [expr 540 - 10*$nn] \
    [expr 875 - $nn*2] \
    [expr 550 - 10*$nn]
    }
  }; #end of switch
  update
  hanoi [expr $n-1] $via $to $from
  }; #end if $n > 0
}

hanoi $numdiscs
```

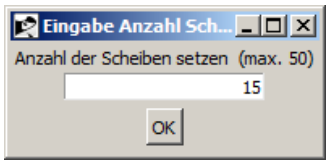
(a) Welchen Tag hat die größte Scheibe?

(b) Welche x0 y0 x1 y1 Koordinaten hat die größte Scheibe in der Ausgangslage?

(c) Wie müsste der Aufruf der Prozedur im Hauptprogramm lauten, wenn die Scheiben am Ende auf *C* liegen sollen?

(d) Beim Erzeugen der Scheiben soll jede zweite pink gefärbt werden, damit man die Bewegungen besser sieht. Geben Sie den Programmcode dazu an und markieren Sie die Stelle, wo er eingefügt werden soll.

(e) Die Anzahl der Scheiben soll interaktiv nach Aufruf des Programms in einem Toplevel-Fenster **`.t`** eingegeben werden. Vervollständigen Sie den angegebenen Programmtext.




```
proc  AnzahlSetzen {} {
  global numdiscs
  toplevel .t
  wm title _____________ "Eingabe Anzahl Scheiben"
  grab .t
  label .t.neuerwert -text "Anzahl der Scheiben setzen  (max. 50)"
  set ewert ""
  _________________ .t.eingabe -textvariable ewert \
      -background white -width 20 -justify right
  pack .t.neuerwert
  pack .t.eingabe
  focus ____________________________
  button .t.but -text "OK" -command "
      set numdiscs \_____________________________; destroy .t"
  pack .t.but -padx 5 -pady 5
}

set numdiscs 0
AnzahlSetzen
vwait ___________________________________
```

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2010

Nachname: Vorname:

Matr.Nr. Studiengang:

MUSTERLÖSUNG

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

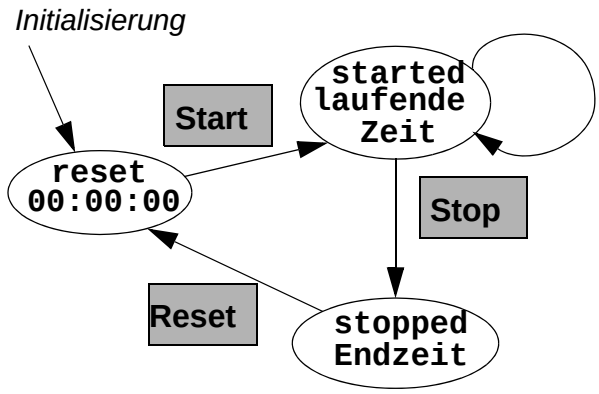
Aufgabe	Punkte maximal	Punkte erreicht
1	12	
2	4+2+2+2	
3	4	
4	2+2+2+4+4	
Summe	40	

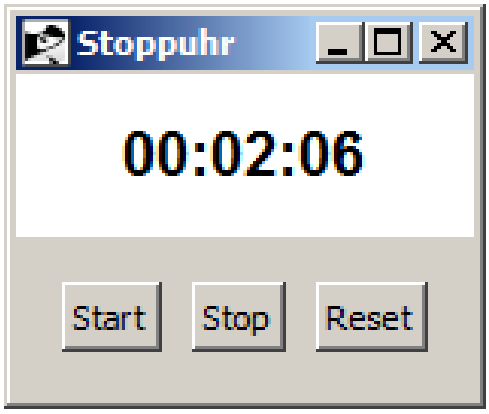
Aufgabe 1:

Wir basteln uns eine Stoppuhr. Diese hat, wie gezeigt, eine Anzeige (als Label realisiert) und darunter drei Knöpfe für Start, Stop und Reset (Rücksetzen). Zu Beginn zeigt die Anzeige 00:00:00, wird Start gedrückt, merken wir uns die Systemuhrzeit als Startzeit und die Anzeige läuft. Angezeigt wird dann der Wert „momentane Uhrzeit“ minus Startzeit. Wird bei laufender Uhr Stop gedrückt, halten wir die Endzeit fest und als Anzeige bleibt Endzeit - Startzeit stehen. Nur mit Reset springt die Anzeige zurück nach Null und es kann wieder Start gedrückt werden.

Die eigentliche Zeitmessung regeln wir über die Systemuhrzeit (**`clock seconds`** liefert die Unix-Zeit in Sekunden). Für die Formatierung in Stunden:Minuten:Sekunden können wir das **`clock format`** Kommando nutzen (**`-gmt true`** stellt sicher, dass wir keine Stunde für die Ortszeit hinzuaddiert bekommen).

Die Uhr kennt drei Zustände: **`reset, started, stopped`**. Die Uhr wird im Zustand **`reset`** initialisiert. Die folgenden Übergänge zeigt das Diagramm.





Fügen Sie den fehlenden Reset-Knopf ein und ergänzen Sie alle Lücken im Programmtext.

```
#!/usr/bin/wish
wm title . "Stoppuhr"
#oben ein Label zur Zeitanzeige
#darunter ein Start-, Stop- und Resetknopf

set state reset
frame .f -padx 10 -pady 10 -bg white
pack .f -expand yes -fill both
frame .g -padx 10 -pady 10
pack .g -expand yes -fill both
label .f.zeit -text "00:00:00" -bg white -font "Arial 15 bold"
pack .f.zeit -side left -expand yes

button .g.start -text "Start" -command {
  if {$state == "reset"} {
    set stime [clock seconds]
    set state "started"
    ShowTime
  }; #else ignore button press
}
```

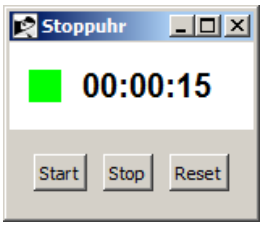
(1) >

(2) >

```
pack .g.start -side left -padx 5 -pady 5 -expand yes -fill both

button .g.stop -text "Stop" -command {
  if {$state == "started"} {
    set etime [clock seconds]
(3) >   set state "stopped"
    ShowTime; #could be omitted
  }; #else ignore button press
}
pack .g.stop -side left -padx 5 -pady 5 -expand yes -fill both

# hier den Reset-Knopf hin


button .g.reset -text "Reset" -command {
  if {$state == "stopped"} {
(4) >   set state "reset"
    ShowTime
  }; #else ignore button press
}
pack .g.reset -side left -padx 5 -pady 5 -expand yes \
  -fill both



proc ShowTime {} {
  global stime etime state

  switch $state {
    "reset" {set s 0}

    "stopped" {set s [expr $etime - $stime]}

    "started" {set s [expr [clock seconds] - $stime]
        after 100 ShowTime }
  }
  .f.zeit configure -text [clock format $s -format %T -gmt true]
}
```

Aufgabe 2:

In die Stoppuhr aus Aufgabe 1 kommt vor die Anzeige der Zeit ein Rechteck (als Frame **`.f.square`**) mit genau 20 Pixel Höhe und Breite. Die Füllfarbe soll grau (**`grey`**) sein im Zustand **`reset`**, grün (**`green`**) im Zustand **`started`** und rot (**`red`**) im Zustand **`stopped`** (vgl. Abbildung). Markieren Sie im Programm von Aufgabe 1 am Rand die Einfügungen als Nummer (1), (2), ... und geben Sie die Programmstücke unten an.



Einfügung/Änderung (1)

```
frame .f.square -width 20 -height 20 -bg grey
pack .f.square -side left -expand yes
```

Einfügung/Änderung (2)

```
.f.square configure -bg green
```

Einfügung/Änderung (3)

```
.f.square configure -bg red
```

Einfügung/Änderung (4)

```
.f.square configure -bg grey
```

Aufgabe 3:

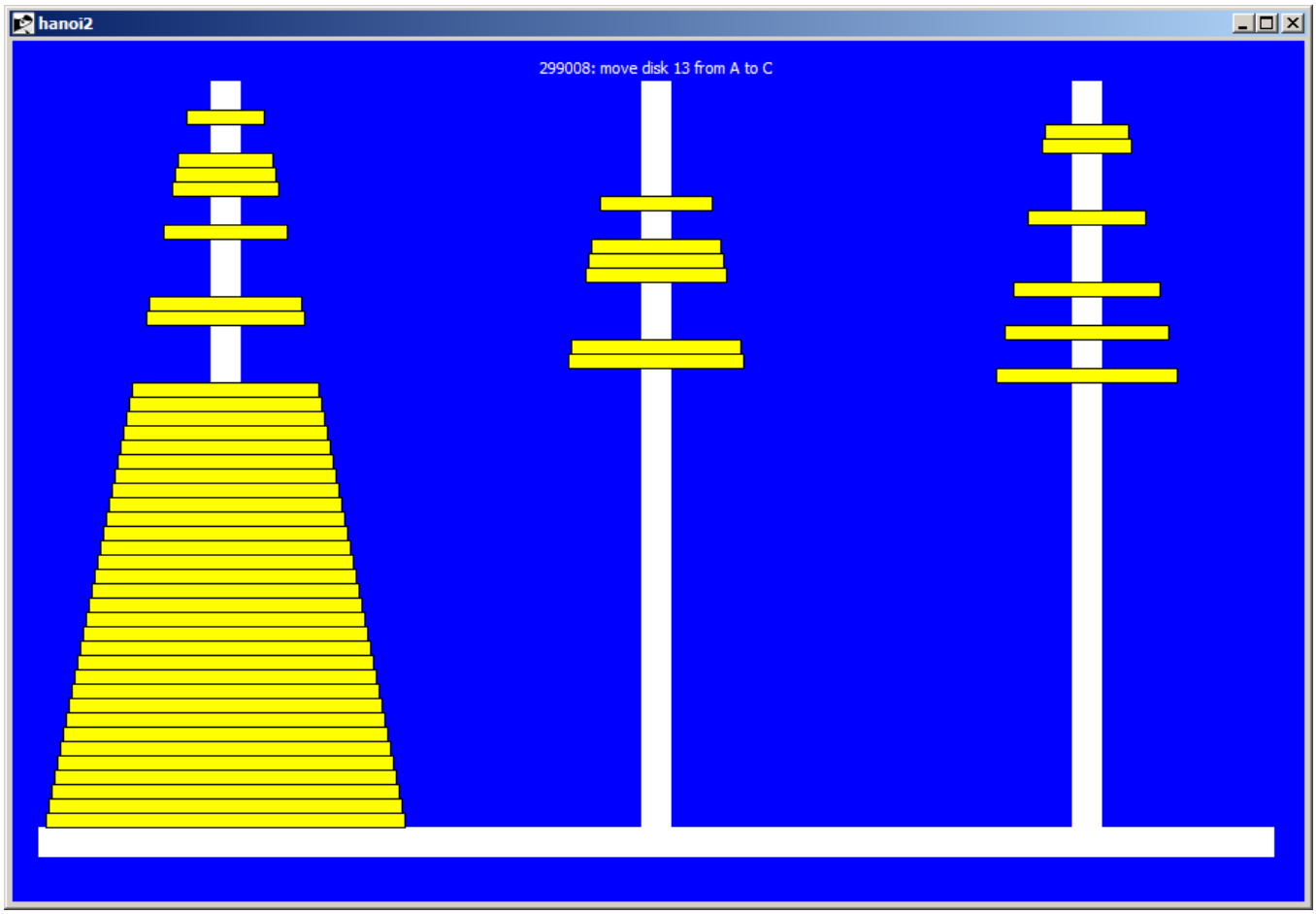
Betrachten Sie das folgende Programmstück. Was zeigt das Label **`.f.zeit`** an, wenn **`s`** den Wert 3666 hat?

```
set m [expr ($s % 3600) / 60]
set sec [expr $s % 60]
.f.zeit configure \
  -text "[expr $s/3600]:[expr {$m<10?0:""}]$m:[expr {$sec<10?0:""}]$sec"
```

1:01:06

Aufgabe 4:

Die *Türme von Hanoi* sind ein Spiel, bei dem es gilt, eine Pyramide von *n* Scheiben von der Stange *A* links auf die mittlere Stange *B* unter Zuhilfenahme der dritten Stange *C* umzuschichten. Dabei darf nie eine Scheibe auf einer kleineren zu liegen kommen.



Die Aufgabe gilt als Paradebeispiel für eine rekursive Lösung, weil man *n* Scheiben von *A* nach *B* bringt, indem man die obersten *n*-1 Scheiben von *A* nach *C* bringt, dann die *n*-te Scheibe von *A* nach *B*, worauf die *n*-1 Scheiben von *C* nach *B* zurück gehen. Die Unteraufgabe des Umschichtens der *n*-1 Scheiben löst man nach der selben Methode, d. h. die Stapel *A*, *B*, und *C* wechseln ihre Rolle als Quelle (*von*), Ziel (*to*) und Zwischenspeicher (*via*).

Das Bild oben zeigt unsere Lösung in Tcl/Tk, wobei zur besseren (und einfacheren) Visualisierung die Lücken in den Stapeln nicht geschlossen werden, die Scheiben sich also nur waagrecht bewegen. Es folgt der Quellcode unseres Programms. Beantworten Sie die danach folgenden Fragen, bzw. lösen Sie die Aufgaben.

```
canvas .c -width 900 -height 600 -background blue
pack .c

.c create rectangle 140 30 160 550 -fill white -outline white
.c create rectangle 440 30 460 550 -fill white -outline white
.c create rectangle 740 30 760 550 -fill white -outline white
.c create rectangle 20 550 880 570 -fill white -outline white

set numdiscs 50
```

(d) >

```
for {set i $numdiscs} {$i>0} {incr i -1} {
  .c create rectangle [expr 25 + ($numdiscs - $i)*2] \
    [expr 540 - 10*($numdiscs - $i)] \
    [expr 275 - ($numdiscs - $i)*2] \
    [expr 550 - 10*($numdiscs - $i)] \
    -fill yellow -tag disc$i
}

.c create text 450 20 -tag banner -fill white -text ""

set count 1
proc hanoi { n {from A} {to B} {via C} } {
  global count numdiscs
  if {$n > 0} {
    hanoi [expr $n-1] $from $via $to
  # puts "$count: move from $from to $to"
  if {$n > 10} {
    .c itemconfigure banner \
      -text "$count: move disk $n from $from to $to"
  }; #show only for bigger disks
  incr count
  set nn [expr $numdiscs - $n]
  switch $to {
  A {.c coords disc$n [expr 25 + $nn*2] \
    [expr 540 - 10*$nn] \
    [expr 275 - $nn*2] \
    [expr 550 - 10*$nn]
    }
  B {.c coords disc$n [expr 325 + $nn*2] \
    [expr 540 - 10*$nn] \
    [expr 575 - $nn*2] \
    [expr 550 - 10*$nn]
    }
  C {.c coords disc$n [expr 625 + $nn*2] \
    [expr 540 - 10*$nn] \
    [expr 875 - $nn*2] \
    [expr 550 - 10*$nn]
    }
  }; #end of switch
  update
  hanoi [expr $n-1] $via $to $from
  }; #end if $n > 0
}

hanoi $numdiscs
```

(a) Welchen Tag hat die größte Scheibe?

disc50

(b) Welche x0 y0 x1 y1 Koordinaten hat die größte Scheibe in der Ausgangslage?

25 540 275 550

(c) Wie müsste der Aufruf der Prozedur im Hauptprogramm lauten, wenn die Scheiben am Ende auf *C* liegen sollen?

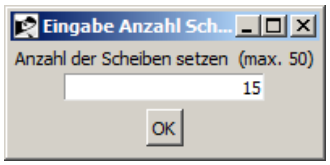
hanoi $numdiscs A C B

(d) Beim Erzeugen der Scheiben soll jede zweite pink gefärbt werden, damit man die Bewegungen besser sieht. Geben Sie den Programmcode dazu an und markieren Sie die Stelle, wo er eingefügt werden soll.

```
if { $i % 2 == 0 } { .c itemconfigure disc$i -fill pink }
```

Eingefügt bei (d). Andere Lösungen möglich

(e) Die Anzahl der Scheiben soll interaktiv nach Aufruf des Programms in einem Toplevel-Fenster **.t** eingegeben werden. Vervollständigen Sie den angegebenen Programmtext.




```
proc  AnzahlSetzen {} {
  global numdiscs
  toplevel .t
  wm title .t "Eingabe Anzahl Scheiben"
  grab .t
  label .t.neuerwert -text "Anzahl der Scheiben setzen  (max. 50)"
  set ewert ""
  entry .t.eingabe -textvariable ewert \
       -background white -width 20 -justify right
  pack .t.neuerwert
  pack .t.eingabe
  focus .t.eingabe
  button .t.but -text "OK" -command "
       set numdiscs \$ewert; destroy .t"
  pack .t.but -padx 5 -pady 5
}

set numdiscs 0
AnzahlSetzen
vwait numdiscs
```

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ zum Wintersemester 2011/12

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 120 Minuten.

Aufgabe	**Punkte max.**	**Punkte erreicht**
1	6	
2	4	
3	6	
4	6	
Summe	**22**	

Aufgabe 1:

Bei einem sog. *Annuitätendarlehen* zahlt der Kreditnehmer (Kunde) das Darlehen in jährlich gleichen Raten (meist monatlich) dem Kreditgeber (Bank) zurück. Die Zahlung (Annuität) enthält einen Sollzinsanteil und einen Tilgungsanteil. Weil die Darlehensschuld durch die Tilgung immer kleiner wird, sinkt der Zinsanteil mit der Zeit und der Tilgungsanteil der monatlichen Raten steigt. Der Sollzins wird meist für die Dauer der Laufzeit fest vereinbart.

Wir stellen hier einen sehr einfachen Tilgungsrechner vor, mit dem man leicht sehen kann, wie sich die Darlehensschuld mit der Zeit bei gegebenem Zins reduziert. Weil Banken die monatlichen Raten anders verrechnen, weicht der Rechner hier geringfügig vom bankenüblichen Berechnungsschema ab und sollte für tatsächliche Tilgungspläne nicht verwendet werden.

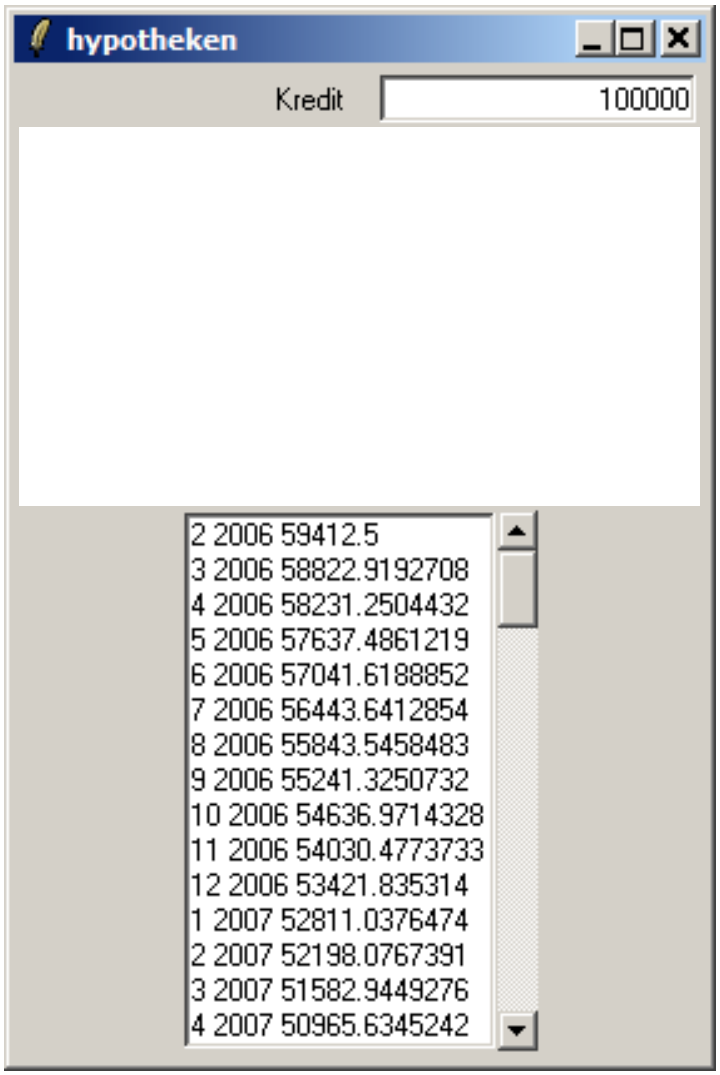
Füllen Sie die Lücken im Programm!

```
#kleines Hypotheken-Tilgungsprogramm - L. Wegner 2006

set kredit 100000
set restschuld 60000
set startmonat 1
set startjahr 2006
set annuitaet 800
set zinssatz 4.25

label .klabel -text "Kredit"
entry .kentry -width 20 -textvariable _____________ -justify right
grid .klabel .kentry -padx 4 -pady 4 -sticky e

label .rslabel -text "Restschuld"
entry .rsentry -width 20 -textvariable _____________ -justify right
grid .rslabel .rsentry -padx 4 -pady 4 -sticky e

frame .m
frame .y
grid .m .y -padx 4 -pady 4

label .m.mlabel -text "zum Monat (mm)"
entry .m.mentry -width 2 -textvariable _____________________
pack .m.mlabel .m.mentry -side left -padx 4 -pady 4

label .y.ylabel -text "Jahr (yyyy)"
entry .y.yentry -width 4 -textvariable _____________________
pack .y.ylabel .y.yentry -side left -padx 4 -pady 4

label .annlabel -text "Annuität"
entry .annentry -width 20 -textvariable annuitaet \
  -justify right
```

```
grid .annlabel .annentry -padx 4 -pady 4 -sticky e

label .zinslabel -text "Zinssatz nominal in \%"
entry .zinsentry -width 5 -textvariable zinssatz -justify right
grid .zinslabel .zinsentry -padx 4 -pady 4 -sticky e

button .start -width 20 -bg "blue" -fg "white" -text "Start" \
  -command {.r.lbox delete 0 end; berechne}
button .exit -width 20 -bg "red" -fg "white" -text "Ende" \
  -command {exit}
grid .start .exit -padx 4 -pady 4

frame .r
grid .r -padx 4 -pady 4
grid configure .r -columnspan 2

listbox .r.lbox  -yscroll ".r.scroll set" -setgrid 1 \
    -height 15
scrollbar .r.scroll -command ".r.lbox yview"
pack .r.scroll -side right -fill y
pack .r.lbox -side left -expand 1 -fill both

.r.lbox insert 0 "$restschuld"

proc berechne {} {
  global restschuld annuitaet zinssatz startjahr startmonat
  set monat $startmonat
  set jahr $startjahr
  set rest $restschuld

  while {$rest > 0} {
    set zins [expr $rest * $zinssatz / 1200.0]
    set tilgung [expr $annuitaet - $zins]
    set rest [_____________________________________________]
    incr monat
    if {$monat > 12} {set monat 1; incr jahr}
    .r.lbox _____________________ end "$monat $jahr $rest"
  }
  return $rest
}
```

Aufgabe 2:

Wie sieht die Eingabemaske des Programm aus? Ergänzen Sie die fehlenden Teile! Der untere Teil des Screenshots zeigt übrigens den Ablauf für die Voreinstellung des Programms (60.000,- Schuld, 4.25% Zins, usw.).



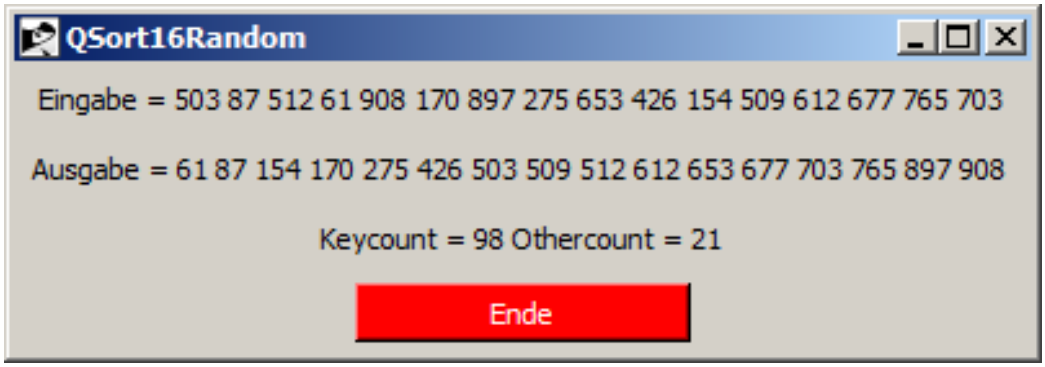
Aufgabe 3:

Das Sortierverfahren Quicksort teilt bekanntlich eine Folge von Sätzen in drei Teile: alle Sätze mit Schlüssel kleiner dem sog. Pivotelement, alle Sätze gleich dem Pivotelement und alle Sätze mit Schlüssel größer dem Pivotelement. Die Sätze, die gleich dem Pivotelement sind, kommen in die Mitte, die Folgen mit kleineren bzw. größeren Werten werden dann rekursiv sortiert. Als Pivotelement kann man das erste, letzte, mittlere oder einen Median-von-dreien wählen.

Betrachten Sie die Implementierung hier mit Tcl/Tk (aus www.rosettacode.org) und ergänzen Sie die Lücken! Die Ausgabe sieht man unten.

```
#Rosetta Code mit wish Rahmen zur Anzeige
set keycount 0; set othercount 0
proc quicksort {m} {
  global keycount othercount
  incr othercount
  if {[_____________________________] <= 1} {#Listenlaenge 0 oder 1;
      return $m
  }
  set pivot [lindex $m 0]
  set less [set equal [set greater [list]]]; #drei leere Listen
  foreach _____________ $m {
    if {$x < $pivot} {incr keycount 1} else {incr keycount 2}
    lappend [expr {$x < $pivot ? "less" : _________________ ? \
        "greater" : "equal"}] $x
  }
  return [concat [_____________________________] $equal \
    [_____________________________]]
}

set ein "503 87 512 61 908 170 897 275 653 426 154 509 612 677 765
703"
label .inlabel -text "Eingabe = $ein"
pack .inlabel -padx 4 -pady 4

label .outlabel -text "Ausgabe = [_____________________________] "
pack .outlabel -padx 4 -pady 4

label .zaehler -text "Keycount = $keycount Othercount = $othercount"
pack .zaehler -padx 4 -pady 4

button .exit -width 20 -bg "red" -fg "white" -text "Ende" \
```

```
  -command {exit}
pack .exit -padx 4 -pady 4
```

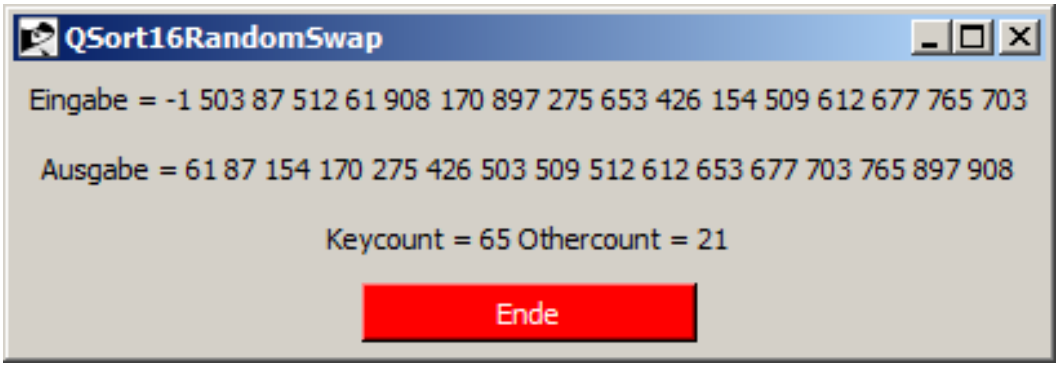


Aufgabe 4:

Hier stellen wir eine eher traditionelle Quicksort-Implementierung vor, die mit einem Array arbeitet. Ergänzen Sie die Lücken. Die Ausgabe sehen Sie wieder auf der nächsten Seite.

```
#Tcl Quicksort mit Array und Rahmen zur Anzeige
set keycount 0; set othercount 0
proc qsort {m el r} { #array m in den Grenzen el bis r
  global keycount othercount
  upvar __________________ a
  incr othercount
  if {$r > $el} {
    set i [expr $el - 1]; set j $r
    set pivot $a($r)
    while {true} {
      while {true} {
        incr i; incr keycount; if {$a($i) >= $pivot} {break} }
      while {true} {
        incr j -1; incr keycount; if {$a($j) <= $pivot} {break} }
      if {$i >= $j} {break}
      set aux $a($i); set a($i) $a($j); set a($j) $aux
    }
    set aux $a($i); set a($i) $a($r); set a($r) $aux
    qsort a $el [expr $i - 1]
    qsort a [expr $i + 1] _______________
  }
}

set a(0) -1; #stopper
set ein "-1 503 87 512 61 908 170 897 275 653 426 154 509 612 677 765
703"
for {set i 1} {$i <= 16} {incr i} {
  set __________ [lindex $ein _______]
}
label .inlabel -text "Eingabe = $ein"
pack .inlabel -padx 4 -pady 4

qsort a "1" "16"
```

```
set aus {}
for {set i 1} {$i <= 16} {incr i} {
  lappend __________ _____________________
}

label .outlabel -text "Ausgabe = $aus"
pack .outlabel -padx 4 -pady 4

label .zaehler -text "Keycount = $keycount Othercount = $othercount"
pack .zaehler -padx 4 -pady 4

button .exit -width 20 -bg "red" -fg "white" -text "Ende" \
  -command {exit}
pack .exit -padx 4 -pady 4
```



ENDE DER KLAUSUR

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ zum Wintersemester 2011/12

Nachname: Musterlösung Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 120 Minuten.

Aufgabe	Punkte max.	Punkte erreicht
1	6	
2	4	
3	6	
4	6	
Summe	**22**	

Aufgabe 1:

Bei einem sog. *Annuitätendarlehen* zahlt der Kreditnehmer (Kunde) das Darlehen in jährlich gleichen Raten (meist monatlich) dem Kreditgeber (Bank) zurück. Die Zahlung (Annuität) enthält einen Sollzinsanteil und einen Tilgungsanteil. Weil die Darlehensschuld durch die Tilgung immer kleiner wird, sinkt der Zinsanteil mit der Zeit und der Tilgungsanteil der monatlichen Raten steigt. Der Sollzins wird meist für die Dauer der Laufzeit fest vereinbart.

Wir stellen hier einen sehr einfachen Tilgungsrechner vor, mit dem man leicht sehen kann, wie sich die Darlehensschuld mit der Zeit bei gegebenem Zins reduziert. Weil Banken die monatlichen Raten anders verrechnen, weicht der Rechner hier geringfügig vom bankenüblichen Berechnungsschema ab und sollte für tatsächliche Tilgungspläne nicht verwendet werden.

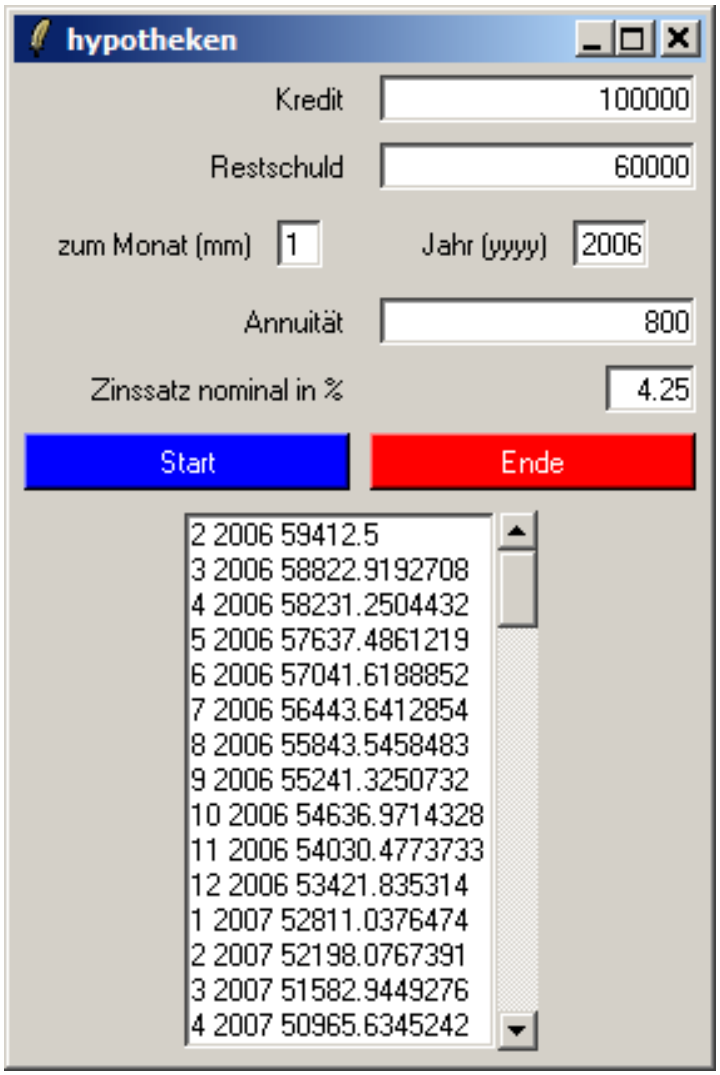
Füllen Sie die Lücken im Programm!

```
#kleines Hypotheken-Tilgungsprogramm - L. Wegner 2006

set kredit 100000
set restschuld 60000
set startmonat 1
set startjahr 2006
set annuitaet 800
set zinssatz 4.25

label .klabel -text "Kredit"
entry .kentry -width 20 -textvariable ___kredit____ -justify right
grid .klabel .kentry -padx 4 -pady 4 -sticky e

label .rslabel -text "Restschuld"
entry .rsentry -width 20 -textvariable __restschuld__ -justify right
grid .rslabel .rsentry -padx 4 -pady 4 -sticky e

frame .m
frame .y
grid .m .y -padx 4 -pady 4

label .m.mlabel -text "zum Monat (mm)"
entry .m.mentry -width 2 -textvariable __startmonat__
pack .m.mlabel .m.mentry -side left -padx 4 -pady 4

label .y.ylabel -text "Jahr (yyyy)"
entry .y.yentry -width 4 -textvariable __startjahr__
pack .y.ylabel .y.yentry -side left -padx 4 -pady 4

label .annlabel -text "Annuität"
entry .annentry -width 20 -textvariable annuitaet \
  -justify right
```

```
grid .annlabel .annentry -padx 4 -pady 4 -sticky e

label .zinslabel -text "Zinssatz nominal in \%"
entry .zinsentry -width 5 -textvariable zinssatz -justify right
grid .zinslabel .zinsentry -padx 4 -pady 4 -sticky e

button .start -width 20 -bg "blue" -fg "white" -text "Start" \
  -command {.r.lbox delete 0 end; berechne}
button .exit -width 20 -bg "red" -fg "white" -text "Ende" \
  -command {exit}
grid .start .exit -padx 4 -pady 4

frame .r
grid .r -padx 4 -pady 4
grid configure .r -columnspan 2

listbox .r.lbox  -yscroll ".r.scroll set" -setgrid 1 \
    -height 15
scrollbar .r.scroll -command ".r.lbox yview"
pack .r.scroll -side right -fill y
pack .r.lbox -side left -expand 1 -fill both

.r.lbox insert 0 "$restschuld"

proc berechne {} {
  global restschuld annuitaet zinssatz startjahr startmonat
  set monat $startmonat
  set jahr $startjahr
  set rest $restschuld

  while {$rest > 0} {
    set zins [expr $rest * $zinssatz / 1200.0]
    set tilgung [expr $annuitaet - $zins]
    set rest [__expr $rest - $tilgung__]
    incr monat
    if {$monat > 12} {set monat 1; incr jahr}
    .r.lbox __insert__ end "$monat $jahr $rest"
  }
  return $rest
}
```

Aufgabe 2:

Wie sieht die Eingabemaske des Programm aus? Ergänzen Sie die fehlenden Teile! Der untere Teil des Screenshots zeigt übrigens den Ablauf für die Voreinstellung des Programms (60.000,- Schuld, 4.25% Zins, usw.).



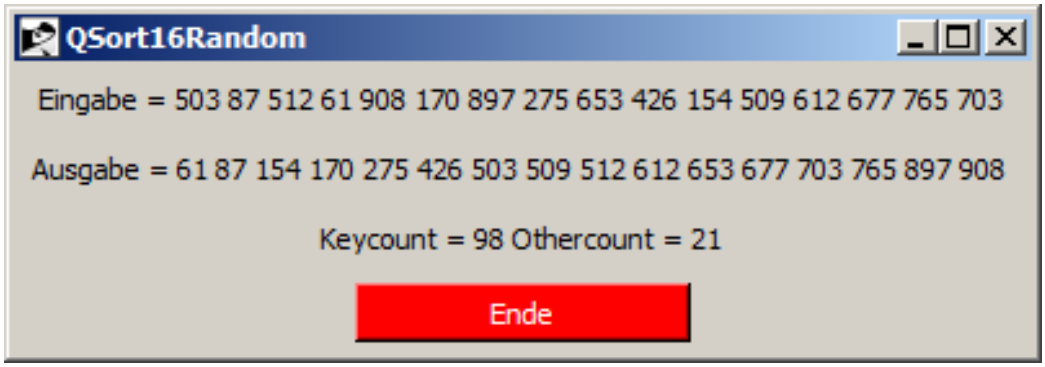
Aufgabe 3:

Das Sortierverfahren Quicksort teilt bekanntlich eine Folge von Sätzen in drei Teile: alle Sätze mit Schlüssel kleiner dem sog. Pivotelement, alle Sätze gleich dem Pivotelement und alle Sätze mit Schlüssel größer dem Pivotelement. Die Sätze, die gleich dem Pivotelement sind, kommen in die Mitte, die Folgen mit kleineren bzw. größeren Werten werden dann rekursiv sortiert. Als Pivotelement kann man das erste, letzte, mittlere oder einen Median-von-dreien wählen.

Betrachten Sie die Implementierung hier mit Tcl/Tk (aus www.rosettacode.org) und ergänzen Sie die Lücken! Die Ausgabe sieht man unten.

```
#Rosetta Code mit wish Rahmen zur Anzeige
set keycount 0; set othercount 0
proc quicksort {m} {
  global keycount othercount
  incr othercount
  if {[__llength $m__] <= 1} {#Listenlaenge 0 oder 1;
      return $m
  }
  set pivot [lindex $m 0]
  set less [set equal [set greater [list]]]; #drei leere Listen
  foreach _x__ $m {
    if {$x < $pivot} {incr keycount 1} else {incr keycount 2}
    lappend [expr {$x < $pivot ? "less" : __$x > $pivot__ ? \
        "greater" : "equal"}] $x
  }
  return [concat [__quicksort $less__] $equal \
    [__quicksort $greater__]]
}

set ein "503 87 512 61 908 170 897 275 653 426 154 509 612 677 765
703"
label .inlabel -text "Eingabe = $ein"
pack .inlabel -padx 4 -pady 4

label .outlabel -text "Ausgabe = [__quicksort $ein__] "
pack .outlabel -padx 4 -pady 4

label .zaehler -text "Keycount = $keycount Othercount = $othercount"
pack .zaehler -padx 4 -pady 4

button .exit -width 20 -bg "red" -fg "white" -text "Ende" \
```

```
  -command {exit}
pack .exit -padx 4 -pady 4
```

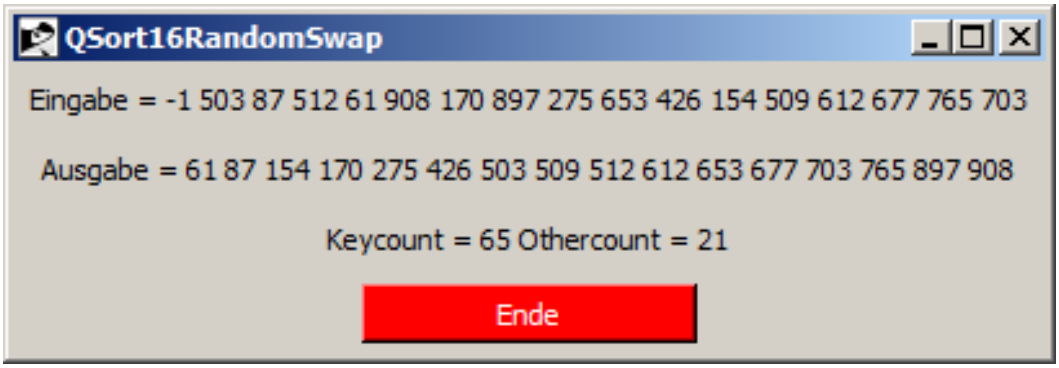


Aufgabe 4:

Hier stellen wir eine eher traditionelle Quicksort-Implementierung vor, die mit einem Array arbeitet. Ergänzen Sie die Lücken. Die Ausgabe sehen Sie wieder auf der nächsten Seite.

```
#Tcl Quicksort mit Array und Rahmen zur Anzeige
set keycount 0; set othercount 0
proc qsort {m el r} { #array m in den Grenzen el bis r
  global keycount othercount
  upvar __$m__ a
  incr othercount
  if {$r > $el} {
    set i [expr $el - 1]; set j $r
    set pivot $a($r)
    while {true} {
      while {true} {
        incr i; incr keycount; if {$a($i) >= $pivot} {break} }
      while {true} {
        incr j -1; incr keycount; if {$a($j) <= $pivot} {break} }
      if {$i >= $j} {break}
      set aux $a($i); set a($i) $a($j); set a($j) $aux
    }
    set aux $a($i); set a($i) $a($r); set a($r) $aux
    qsort a $el [expr $i - 1]
    qsort a [expr $i + 1] __$r__
  }
}

set a(0) -1; #stopper
set ein "-1 503 87 512 61 908 170 897 275 653 426 154 509 612 677 765
703"
for {set i 1} {$i <= 16} {incr i} {
  set __a($i)_ [lindex $ein __$i__]
}
label .inlabel -text "Eingabe = $ein"
pack .inlabel -padx 4 -pady 4

qsort a "1" "16"
```

```
set aus {}
for {set i 1} {$i <= 16} {incr i} {
  lappend __aus__ __$a($i)__
}

label .outlabel -text "Ausgabe = $aus"
pack .outlabel -padx 4 -pady 4

label .zaehler -text "Keycount = $keycount Othercount = $othercount"
pack .zaehler -padx 4 -pady 4

button .exit -width 20 -bg "red" -fg "white" -text "Ende" \
  -command {exit}
pack .exit -padx 4 -pady 4
```



ENDE DER KLAUSUR

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2012

Nachname: Vorname:

Matr.Nr.: Studiengang:

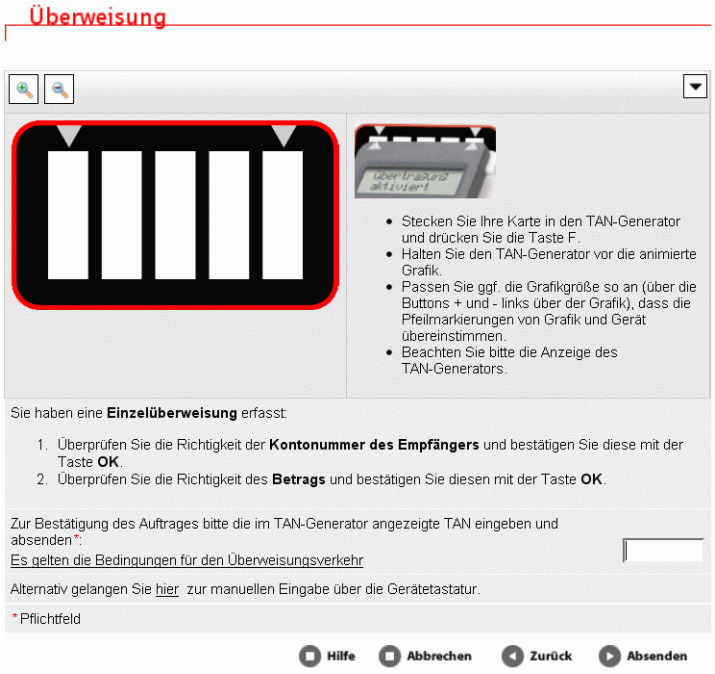
Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte max.	Punkte
1	4	
2	4+2+2	
3	2+2+2	
4	4	
5	6	
6	6	
7	4	
8	2	
Summe	40	

Aufgabe 1:

In dieser Klausur geht es um das ChipTAN-Verfahren der Banken und Sparkassen für Online-Überweisungen, bei dem eine Zeichenfolge als sog. „Flickercode“ am Bildschirm angezeigt wird. Mit einem TAN-Generator, der fünf Lichtsensoren besitzt, kann man sich diese Zeichenfolge, die u.a. das Empfängerkonto und den Betrag enthält, auf das kleine Gerät übertragen lassen und dort auf dessen Display das Zielkonto und den Betrag nochmals prüfen. Der TAN-Generator berechnet dann über die eingesteckte EC-Karte eine vom Konto, Betrag und anderen Werten abhängige TAN, die man am Rechner eingibt. Wegen des getrennten Geräts und der optischen Übertragung schützt das Verfahren vor Manipulationen durch Trojaner auf dem Rechner des Online-Kunden.

Für die Beantwortung der Fragen brauchen Sie dieses Verfahren nicht vollständig zu verstehen. Vielmehr werden wir Teilaspekte betrachten und Lösungen für ein Simulationsprogramm, das Flickercodes produziert, entwickeln.



Wir beginnen mit der Anzeige der fünf Balken (vier Datenbit und ein Takt) im schwarzen Feld mit rotem Rahmen, die wir durch die Prozedur **`AnzeigeAnlegen`** auf einem Canvas (Parameter **`worauf`**) mit einer Anfangsskalierung von 1.0 (Parameter **`faktor`**) erzeugen. Die Methode **`scale`** des Canvas-Widgets skaliert die Items auf der Leinwand, deren Tag angegeben wird. Die Balken erhalten zusätzlich einzelne Tags, damit wir sie später gezielt von weiß auf schwarz und umgekehrt umschalten können.

(a) Ergänzen Sie die Lücken!

```
proc AnzeigeAnlegen {worauf faktor} {
  #flicker Rahmen, Dreiecke und Balken lt Sparkassenabbildung
  $worauf create polygon 20 20 230 20 230 140 20 140 -fill red \
    -joinstyle round -outline red -width 40 -tag display
  $worauf create polygon 25 25 225 25 225 135 25 135 -fill black \
    -joinstyle round -outline black -width 40 -tag display
  set startx 35; set i 0
  foreach _______ {takt bit0 bit1 bit2 bit3} {
    $worauf create rectangle [expr $startx+$i*38] 35 \
      [expr $startx+($i+1)*38 - 8] 125 -fill white \
```

```
      -outline "" -tags "$bar display"
    ____________________
  }
  #Positionierdreiecke Mitte erster und letzter Balken anlegen
  $worauf create polygon [expr $startx+5] 18 [expr $startx+25] 18 \
    [expr $startx+15] 35 -fill lightgrey -outline "" -tag display
  set startx [expr $startx + ____________________________]
  $worauf create polygon [expr $startx+5] 18 [expr $startx+25] 18 \
    [expr $startx+15] 35 -fill lightgrey -outline "" -tag display
  #auf gewuenschte Groesse skalieren, 0 0 ist Ursprung
  $worauf scale _________________ 0 0 $faktor $faktor
}
```

Aufgabe 2:

Jetzt betrachten wir eine Prozedur zum Skalieren dieser Anzeige, die wir später mit zwei Zoom-Knöpfen verbinden. So können Kunden die Anzeige auf unterschiedlichen Monitoren mit den Pfeilen ihres TAN-Generators abgleichen.

Dabei stellt es sich heraus, dass es besser ist, alle Items auf der Leinwand zu löschen und mit der gewünschten Skalierung (globale Variable **`scale`**) neu anzulegen, als diese wiederholt mit 1.05 bzw. 0.95 zu skalieren, weil sich nach mehreren Schritten durch Rundung unschöne Effekte einstellen.

(a) Ergänzen Sie die Lücken!

```
#Skalierung der Flickeranzeige
proc Skaliere {was ____________________} {
  global scale
  set scale [expr {$scale+$increment < 0.1?0.1:$scale + $increment}]
  _____________ delete all
  $was configure -width [expr int(250*$scale)] \
    -height [expr int(160*$scale)]
  AnzeigeAnlegen __________ _______________
}
```

(b) Nehmen wir an, nach dem ersten **`set scale [...]`** wollten wir einen zweiten **`set`**-Befehl einfügen, der die größte zugelassene Skalierung auf 5.0 begrenzt. Wie lautet der Befehl, der **`scale`** unverändert lässt, wenn der Wert ≤ 5.0 ist, mit der Syntax analog zu oben?

(c) Warum heißt es oben **`$was configure ...`** und nicht **`$was itemconfigure ...`** ? Kurze Erläuterung auf der Rückseite des Blatts.

Aufgabe 3:

Jetzt geht es darum, die Balken zum Blinken zu bringen. Wie aus der Prozedur **AnzeigeAnlegen** bereits ersichtlich, ist der linke Balken der Takt, daran schließen sich von links nach rechts die Nutzbits mit den Wertigkeiten 2^0, 2^1, 2^2, 2^3 an. Damit lässt sich je Taktzyklus eine Hexadezimalziffer (Tetrade, Nibble) übertragen. Ein gesetztes Bit (1) entspricht einem weißen Balken, 0 ist schwarz. Das Taktbit ist zu Beginn der Übertragung einer Tetrade weiß und wechselt in der Mitte der Übertragung der Tetrade zu schwarz. Damit blinkt der Taktgeber doppelt so schnell wie die Nutzbits.

(a) Ergänzen Sie die Lücken!

```
proc ShowNibble {worauf b0 b1 b2 b3} {
  $worauf itemconfigure takt -fill _____________________
  $worauf itemconfigure bit0 -fill [expr {$b0?"white":"black"}]
  $worauf itemconfigure bit1 -fill [expr {$b1?"white":"black"}]
  $worauf itemconfigure bit2 -fill [expr {$b2?"white":"black"}]
  $worauf itemconfigure bit3 -fill [expr {$b3?"white":"black"}]
  update
  after 50
  $worauf itemconfigure takt -fill _____________________
  update
  after 50
}
```

(b) Die zu übertragende Zeichenfolge wird in einer Endlosschleife angezeigt. Wenn die Zeichenfolge aus 40 Hexadezimalziffern besteht, wie lange dauert dann - bei der oben eingestellten Taktrate - ein Durchlauf der Zeichenfolge?

(c) Die Zeichenfolge beginnt mit zwei Synchronisationsbytes 0FFF. Erst wenn diese erkannt wurden, beginnt der eigentliche Lesevorgang. Wie lange dauert es insgesamt nach dem Anlegen des TAN-Generators am Monitor, bis der Lesevorgang im schlechtesten Fall abgeschlossen wurde für die o. g. Zeichenfolge von 40 Tetraden (darin seien die beiden Synchronisationsbytes schon enthalten)?

Aufgabe 4:

Liegt eine Bytefolge vor, dann werden die beiden Tetraden jedes Bytes in vertauschter Reihenfolge übertragen, d. h. eine zu sendende Folge 0F FF 12 04 87 1A ... wird in der Reihenfolge F0 FF 21 40 78 A1 ... übertragen.

Für unser Simulationsprogramm arbeiten wir mit einer Zeichenketteneingabe, d. h. jede gedachte Tetrade wird als ASCII-Zeichen (character) gehalten. Die Prozedur **ShowNibble** verlangt die Tetrade als 4 Bitparameter. Die Umsetzung erfolgt etwas handwerklich schlicht über ein Array **H**, das wir wie folgt vorbelegen:

```
set H(0) {0 0 0 0}; set H(1) {1 0 0 0}; set H(2) {0 1 0 0}
...
set H(C) {0 0 1 1}; set H(D) {1 0 1 1}; set H(E) {0 1 1 1}
set H(F) {1 1 1 1}
```

Ergänzen Sie die Lücken unten, wobei **tfinal** die zu sendende Zeichenfolge ist.

```
while {$weiter} {
  for {set i 0} {$i < [string length $tfinal]} {incr i 2} {
    set c1 [string index ____________________ ____________________]
    set c2 [string index ____________________ [____________________]]
    #anzeigen in umgekehrter Reihenfolge!
    ShowNibble $worauf [lindex $H($c2) 0] \
      [lindex $H($c2) 1] [lindex $H($c2) 2] \
      [lindex $H($c2) 3]
    update
    ShowNibble $worauf [lindex $H($c1) 0] \
      [lindex $H($c1) 1] [lindex $H($c1) 2] \
      [lindex $H($c1) 3]
    update
    if {!$weiter} {break}
  };#end for
};#end while
```

Aufgabe 5:

Jetzt entwerfen wir ein kleines Rahmenprogramm mit Eingabemöglichkeiten für Kontonummer des Empfängers, Betrag und einen sog. Startcode (5-stellige Zahl). Darüber kommen die Flickercodeanzeige und die beiden Zoom-Knöpfe, darunter drei Knöpfe für Start, Stop und Ende.

Ergänzen Sie die Lücken (nur die Lücken mit Unterstreichung)!

```
frame .f -bg lightgrey
pack .f
...
frame .f.zooms
pack .f.zooms -padx 40 -anchor w

image create photo zout -file "zoom_out.gif"
button .f.zooms.out -image zout -command {_____________ ___________ -0.05}
pack .f.zooms.out -side left
image create photo zin -file "zoom_in.gif"
button .f.zooms.in -image zin -command {_____________ _____________ 0.05}
pack .f.zooms.in -side left

set scale 1.0
canvas .f.s -width 250 -height 160 -bg lightgray
pack .f.s -padx 40 -anchor nw

#Display auf canvas mit Faktor 100% anzeigen
AnzeigeAnlegen _________________ _________________________

frame .f.e
pack .f.e -fill both -padx 40 -pady 10 -anchor w
label .f.e.konto -text "_____________________________________"
entry .f.e.kentry -textvariable EKonto \
      -background white
label .f.e.betrag -text "Betrag"
entry .f.e.bentry -textvariable Betrag \
      -background white
label .f.e.startcode -text "Startcode"
entry .f.e.sentry -textvariable  startcode \
      -background white
pack .f.e.konto .f.e.kentry .f.e.betrag .f.e.bentry \
      ____________________ ____________________ -side left

frame .f.buttons
pack .f.buttons -padx 40 -pady 10 -anchor w
button .f.buttons.start -text "Start" -width 7 \
      -command {set weiter true; showCode .f.s}
button .f.buttons.stop -text "Stop" -width 7 \
      -command {set weiter false}
button .f.buttons.quit -text "Ende" -width 7 \
      -command {exit}
pack .f.buttons.start .f.buttons.stop .f.buttons.quit -side \
      left -padx 5

label .f.use -text "Eine Kontonummer, Betrag und Startcode eingeben\
      , dann Start drücken." -font "Times 12 normal" -justify left
pack .f.use -anchor w -padx 40 -pady 10
```

Aufgabe 6:

Die Aufbereitung der zu übertragenden Zeichenkette ist aufwändig und erfolgt auf dem Server des Kreditinstituts. U. a. wird eine Kontonummer, die eine ungerade Anzahl an Ziffern aufweist, um ein rechtsstehendes „F“ ergänzt.

Sei **$EKonto** die Ausgangszeichenkette und **ek** die Zielvariable. Ergänzen Sie den Code unten! Hinweis: **append** *v s1 s2* ... fügt die Zeichenketten *s1*, *s2*, ... an den Wert von *v* an, wobei die Variable *v* neu angelegt wird, wenn sie noch nicht existiert.

```
if {([string _________________ _________________ ] % 2) != 0} {
  append _________ _________________ _________
  } else {set ______ _________________}
```

Aufgabe 7:

Eine weitere Aufbereitung erfolgt durch Anhängen einer XOR-Prüfsummenziffer (Hexadezimalziffer), die sich aus dem Argument (Zeichenkette) **from** in der Prozedur **computeXOR** unten berechnet. Dazu kann man in Tcl den XOR-Operator „^“ in **expr** verwenden, der hier einen dezimalen Operanden erhalten soll.

Ergänzen Sie die Lücken!

```
set j 0
foreach hex {0 1 2 3 4 5 6 7 8 9 A B C D E F} {
  set hex2int($hex) $j
  set int2hex($j) $hex
  incr j
}


proc computeXOR {from} {
  global hex2int int2hex
  set x 0
  for {set i 0} {$i < [string length $from]} {incr i} {
    set x [expr $x ^ $hex2int([string ___________ _____________ _________ ])]
  }
  return $int2hex(_____________)
}
```

Aufgabe 8:

Zur Veranschaulichung wird in unserem Simulationsprogramm mittels des Labels **.f.use** der zu übertragende String (Wert von **tfinal**) angezeigt (vgl. letzte Zeile im Bild zu Aufgabe 5). Wie lautet der Befehl zur Änderung des anzuzeigenden Texts?

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2012

MUSTERLÖSUNG

Nachname: Vorname:

Matr.Nr.: Studiengang:

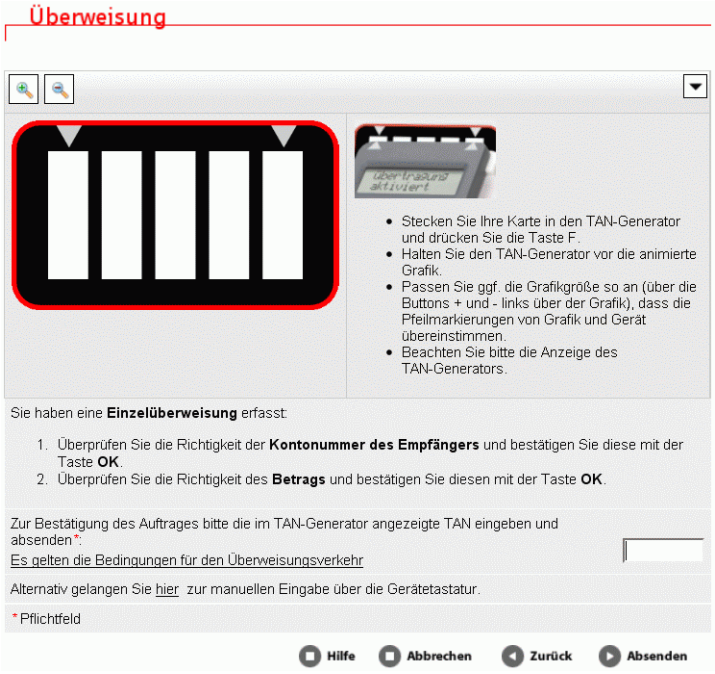
Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte max.	Punkte
1	4	
2	4+2+2	
3	2+2+2	
4	4	
5	6	
6	6	
7	4	
8	2	
Summe	40	

Aufgabe 1:

In dieser Klausur geht es um das ChipTAN-Verfahren der Banken und Sparkassen für Online-Überweisungen, bei dem eine Zeichenfolge als sog. „Flickercode“ am Bildschirm angezeigt wird. Mit einem TAN-Generator, der fünf Lichtsensoren besitzt, kann man sich diese Zeichenfolge, die u.a. das Empfängerkonto und den Betrag enthält, auf das kleine Gerät übertragen lassen und dort auf dessen Display das Zielkonto und den Betrag nochmals prüfen. Der TAN-Generator berechnet dann über die eingesteckte EC-Karte eine vom Konto, Betrag und anderen Werten abhängige TAN, die man am Rechner eingibt. Wegen des getrennten Geräts und der optischen Übertragung schützt das Verfahren vor Manipulationen durch Trojaner auf dem Rechner des Online-Kunden.

Für die Beantwortung der Fragen brauchen Sie dieses Verfahren nicht vollständig zu verstehen. Vielmehr werden wir Teilaspekte betrachten und Lösungen für ein Simulationsprogramm, das Flickercodes produziert, entwickeln.



Wir beginnen mit der Anzeige der fünf Balken (vier Datenbit und ein Takt) im schwarzen Feld mit rotem Rahmen, die wir durch die Prozedur **`AnzeigeAnlegen`** auf einem Canvas (Parameter **`worauf`**) mit einer Anfangsskalierung von 1.0 (Parameter **`faktor`**) erzeugen. Die Methode **`scale`** des Canvas-Widgets skaliert die Items auf der Leinwand, deren Tag angegeben wird. Die Balken erhalten zusätzlich einzelne Tags, damit wir sie später gezielt von weiß auf schwarz und umgekehrt umschalten können.

(a) Ergänzen Sie die Lücken!

```
proc AnzeigeAnlegen {worauf faktor} {
  #flicker Rahmen, Dreiecke und Balken lt Sparkassenabbildung
  $worauf create polygon 20 20 230 20 230 140 20 140 -fill red \
    -joinstyle round -outline red -width 40 -tag display
  $worauf create polygon 25 25 225 25 225 135 25 135 -fill black \
    -joinstyle round -outline black -width 40 -tag display
  set startx 35; set i 0
  foreach _bar_ {takt bit0 bit1 bit2 bit3} {
    $worauf create rectangle [expr $startx+$i*38] 35 \
      [expr $startx+($i+1)*38 - 8] 125 -fill white \
```

```
      -outline "" -tags "$bar display"
    __incr i__
  }
  #Positionierdreiecke Mitte erster und letzter Balken anlegen
  $worauf create polygon [expr $startx+5] 18 [expr $startx+25] 18 \
    [expr $startx+15] 35 -fill lightgrey -outline "" -tag display
  set startx [expr $startx + __4 * 38__]
  $worauf create polygon [expr $startx+5] 18 [expr $startx+25] 18 \
    [expr $startx+15] 35 -fill lightgrey -outline "" -tag display
  #auf gewuenschte Groesse skalieren, 0 0 ist Ursprung
  $worauf scale __display__ 0 0 $faktor $faktor
}
```

Aufgabe 2:

Jetzt betrachten wir eine Prozedur zum Skalieren dieser Anzeige, die wir später mit zwei Zoom-Knöpfen verbinden. So können Kunden die Anzeige auf unterschiedlichen Monitoren mit den Pfeilen ihres TAN-Generators abgleichen.

Dabei stellt es sich heraus, dass es besser ist, alle Items auf der Leinwand zu löschen und mit der gewünschten Skalierung (globale Variable **`scale`**) neu anzulegen, als diese wiederholt mit 1.05 bzw. 0.95 zu skalieren, weil sich nach mehreren Schritten durch Rundung unschöne Effekte einstellen.

(a) Ergänzen Sie die Lücken!

```
#Skalierung der Flickeranzeige
proc Skaliere {was __increment__} {
  global scale
  set scale [expr {$scale+$increment < 0.1?0.1:$scale+$increment}]
  __$was__ delete all
  $was configure -width [expr int(250*$scale)] \
    -height [expr int(160*$scale)]
  AnzeigeAnlegen __$was__ __$scale__
}
```

(b) Nehmen wir an, nach dem ersten **`set scale [...]`** wollten wir einen zweiten **`set`**-Befehl einfügen, der die größte zugelassene Skalierung auf 5.0 begrenzt. Wie lautet der Befehl, der **`scale`** unverändert lässt, wenn der Wert ≤ 5.0 ist, mit der Syntax analog zu oben?

```
set scale [expr {$scale > 5.0?5.0:$scale}]
```

(c) Warum heißt es oben **`$was configure ...`** und nicht **`$was itemconfigure ...`** ? Kurze Erläuterung auf der Rückseite des Blatts.

Die Leinwand (canvas) selbst wird neu konfiguriert, nicht die Items auf der Leinwand.

Aufgabe 3:

Jetzt geht es darum, die Balken zum Blinken zu bringen. Wie aus der Prozedur **AnzeigeAnlegen** bereits ersichtlich, ist der linke Balken der Takt, daran schließen sich von links nach rechts die Nutzbits mit den Wertigkeiten 2^0, 2^1, 2^2, 2^3 an. Damit lässt sich je Taktzyklus eine Hexadezimalziffer (Tetrade, Nibble) übertragen. Ein gesetztes Bit (1) entspricht einem weißen Balken, 0 ist schwarz. Das Taktbit ist zu Beginn der Übertragung einer Tetrade weiß und wechselt in der Mitte der Übertragung der Tetrade zu schwarz. Damit blinkt der Taktgeber doppelt so schnell wie die Nutzbits.

(a) Ergänzen Sie die Lücken!

```
proc ShowNibble {worauf b0 b1 b2 b3} {
  $worauf itemconfigure takt -fill __white__
  $worauf itemconfigure bit0 -fill [expr {$b0?"white":"black"}]
  $worauf itemconfigure bit1 -fill [expr {$b1?"white":"black"}]
  $worauf itemconfigure bit2 -fill [expr {$b2?"white":"black"}]
  $worauf itemconfigure bit3 -fill [expr {$b3?"white":"black"}]
  update
  after 50
  $worauf itemconfigure takt -fill __black__
  update
  after 50
}
```

(b) Die zu übertragende Zeichenfolge wird in einer Endlosschleife angezeigt. Wenn die Zeichenfolge aus 40 Hexadezimalziffern besteht, wie lange dauert dann - bei der oben eingestellten Taktrate - ein Durchlauf der Zeichenfolge?

40 * 100 ms = 4000 ms = 4 Sekunden

(c) Die Zeichenfolge beginnt mit zwei Synchronisationsbytes 0FFF. Erst wenn diese erkannt wurden, beginnt der eigentliche Lesevorgang. Wie lange dauert es insgesamt nach dem Anlegen des TAN-Generators am Monitor, bis der Lesevorgang im schlechtesten Fall abgeschlossen wurde für die o. g. Zeichenfolge von 40 Tetraden (darin seien die beiden Synchronisationsbytes schon enthalten)?

8 Sekunden

Aufgabe 4:

Liegt eine Bytefolge vor, dann werden die beiden Tetraden jedes Bytes in vertauschter Reihenfolge übertragen, d. h. eine zu sendende Folge 0F FF 12 04 87 1A ... wird in der Reihenfolge F0 FF 21 40 78 A1 ... übertragen.

Für unser Simulationsprogramm arbeiten wir mit einer Zeichenketteneingabe, d. h. jede gedachte Tetrade wird als ASCII-Zeichen (character) gehalten. Die Prozedur **ShowNibble** verlangt die Tetrade als 4 Bitparameter. Die Umsetzung erfolgt etwas handwerklich schlicht über ein Array **H**, das wir wie folgt vorbelegen:

```
set H(0) {0 0 0 0}; set H(1) {1 0 0 0}; set H(2) {0 1 0 0}
...
set H(C) {0 0 1 1}; set H(D) {1 0 1 1}; set H(E) {0 1 1 1}
set H(F) {1 1 1 1}
```

Ergänzen Sie die Lücken unten, wobei **`tfinal`** die zu sendende Zeichenfolge ist.

```
while {$weiter} {
  for {set i 0} {$i < [string length $tfinal]} {incr i 2} {
    set c1 [string index __$tfinal__ __$i__]
    set c2 [string index __$tfinal__ [__expr $i+1__]]
    #anzeigen in umgekehrter Reihenfolge!
    ShowNibble $worauf [lindex $H($c2) 0] \
      [lindex $H($c2) 1] [lindex $H($c2) 2] \
      [lindex $H($c2) 3]
    update
    ShowNibble $worauf [lindex $H($c1) 0] \
      [lindex $H($c1) 1] [lindex $H($c1) 2] \
      [lindex $H($c1) 3]
    update
    if {!$weiter} {break}
  };#end for
};#end while
```

Aufgabe 5:

Jetzt entwerfen wir ein kleines Rahmenprogramm mit Eingabemöglichkeiten für Kontonummer des Empfängers, Betrag und einen sog. Startcode (5-stellige Zahl). Darüber kommen die Flickercodeanzeige und die beiden Zoom-Knöpfe, darunter drei Knöpfe für Start, Stop und Ende.

Ergänzen Sie die Lücken (nur die Lücken mit Unterstreichung)!

```
frame .f -bg lightgrey
pack .f
...
frame .f.zooms
pack .f.zooms -padx 40 -anchor w

image create photo zout -file "zoom_out.gif"
button .f.zooms.out -image zout -command {__Skaliere__ __.f.s__ -0.05}
pack .f.zooms.out -side left
image create photo zin -file "zoom_in.gif"
button .f.zooms.in -image zin -command {__Skaliere__ __.f.s__ 0.05}
pack .f.zooms.in -side left

set scale 1.0
canvas .f.s -width 250 -height 160 -bg lightgray
pack .f.s -padx 40 -anchor nw

#Display auf canvas mit Faktor 100% anzeigen
AnzeigeAnlegen __.f.s___ __$scale__

frame .f.e
pack .f.e -fill both -padx 40 -pady 10 -anchor w
label .f.e.konto -text "__Empfängerkonto__"
entry .f.e.kentry -textvariable EKonto \
      -background white
label .f.e.betrag -text "Betrag"
entry .f.e.bentry -textvariable Betrag \
      -background white
label .f.e.startcode -text "Startcode"
entry .f.e.sentry -textvariable  startcode \
      -background white
pack .f.e.konto .f.e.kentry .f.e.betrag .f.e.bentry \
      __.f.e.startcode__ __.f.e.sentry__ -side left

frame .f.buttons
pack .f.buttons -padx 40 -pady 10 -anchor w
button .f.buttons.start -text "Start" -width 7 \
      -command {set weiter true; showCode .f.s}
button .f.buttons.stop -text "Stop" -width 7 \
      -command {set weiter false}
button .f.buttons.quit -text "Ende" -width 7 \
      -command {exit}
pack .f.buttons.start .f.buttons.stop .f.buttons.quit -side \
      left -padx 5

label .f.use -text "Eine Kontonummer, Betrag und Startcode eingeben\
      , dann Start drücken." -font "Times 12 normal" -justify left
pack .f.use -anchor w -padx 40 -pady 10
```

Aufgabe 6:

Die Aufbereitung der zu übertragenden Zeichenkette ist aufwändig und erfolgt auf dem Server des Kreditinstituts. U. a. wird eine Kontonummer, die eine ungerade Anzahl an Ziffern aufweist, um ein rechtsstehendes „F“ ergänzt.

Sei **$EKonto** die Ausgangszeichenkette und **ek** die Zielvariable. Ergänzen Sie den Code unten! Hinweis: **append** *v s1 s2* ... fügt die Zeichenketten *s1*, *s2*, ... an den Wert von *v* an, wobei die Variable *v* neu angelegt wird, wenn sie noch nicht existiert.

```
if {([string __length__ __$EKonto__ ] % 2) != 0} {
   append __ek__ __$EKonto__ __F__
   } else {set __ek__ __$EKonto__}
```

Aufgabe 7:

Eine weitere Aufbereitung erfolgt durch Anhängen einer XOR-Prüfsummenziffer (Hexadezimalziffer), die sich aus dem Argument (Zeichenkette) **from** in der Prozedur **computeXOR** unten berechnet. Dazu kann man in Tcl den XOR-Operator „^“ in **expr** verwenden, der hier einen dezimalen Operanden erhalten soll.

Ergänzen Sie die Lücken!

```
set j 0
foreach hex {0 1 2 3 4 5 6 7 8 9 A B C D E F} {
  set hex2int($hex) $j
  set int2hex($j) $hex
  incr j
}


proc computeXOR {from} {
  global hex2int int2hex
  set x 0
  for {set i 0} {$i < [string length $from]} {incr i} {
    set x [expr $x ^ $hex2int([string __index__ __$from__ __$i__ ])]
  }
  return $int2hex(__$x__)
}
```

Hier fehlte in der Klausur das Dollarzeichen, was bei der Korrektur berücksichtigt wird.

Aufgabe 8:

Zur Veranschaulichung wird in unserem Simulationsprogramm mittels des Labels **.f.use** der zu übertragende String (Wert von **tfinal**) angezeigt (vgl. letzte Zeile im Bild zu Aufgabe 5). Wie lautet der Befehl zur Änderung des anzuzeigenden Texts?

```
.f.use configure -text "tfinal = $tfinal"
```

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2013

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte max.	Punkte
1	6	
2	6+3	
3	4	
4	3	
5	3	
6	2	
7	3	
Summe	30	

Aufgabe 1:

Bei Navigationsgeräten wird für die Eingabe von Stadt, Straße, Hausnummer oder Postleitzahl eine Auswahlliste von noch möglichen Zeichen angeboten, die – in Abhängigkeit des bereits eingegebenen Anfangsstücks – sich aus der Liste der Städte, Straßen usw. ergibt. Bei einer Touchscreen-Eingabe bietet sich eine intelligente, virtuelle Tastatur an, bei der deaktivierte Tasten z. B. hellblau, noch wählbare – und somit aktive Tasten – dunkelblau erscheinen.

Die von uns für diese Aufgabe ausgewählte Datenstruktur und Verfeinerungsmethode funktionieren hinreichend schnell in Tcl/Tk, wäre aber für die Suche in einem Wörterbuch mit mehreren Hunderttausenden Einträgen zu langsam.

Als Ausgangslage ist eine sortierte Liste der Suchwörter (in Großbuchstaben, Leerzeichen, Bindestrich und „ß“) als Datei gegeben und wird in den Speicher als Tcl-Liste *lines* eingelesen. Listen in Tcl beginnen mit Index 0 und der letzte Eintrag hat den Index `[llength $lines] - 1`.

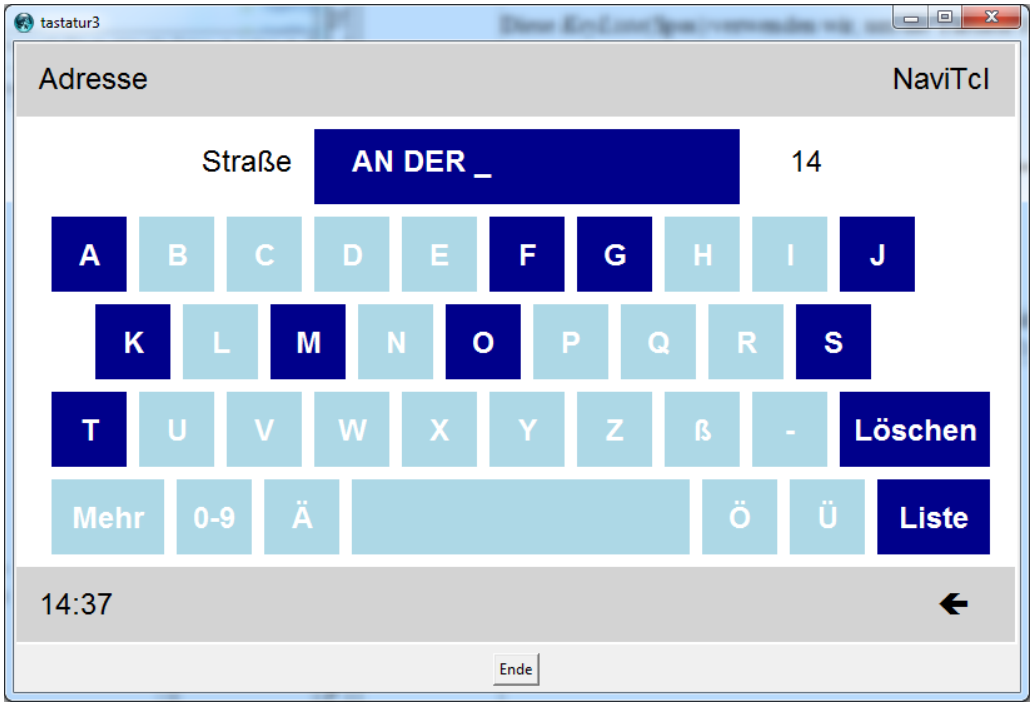
Die einzutippende Zeichenfolge wird in der Variablen *ziel* festgehalten. Zeichen in Zeichenketten (strings) beginnen auch mit dem Index 0. Generell liefert `[string index $ziel $i]` dann das *i*-te Zeichen in *ziel* für 0 <= *i* <= (|*ziel*| - 1).

Wir treffen die Vereinbarung, dass eine Variable (Zähler) *pos* die Iteration steuert. Wir beginnen mit *pos* = 0 für die Eingabe des ersten Zeichens. Zu diesem Zeitpunkt steht die gesamte Liste *lines* zur Auswahl, d. h. *Start*($*pos*) = 0 und *Ende*($*pos*) zeigen auf Anfang und Ende der gesamten Liste.

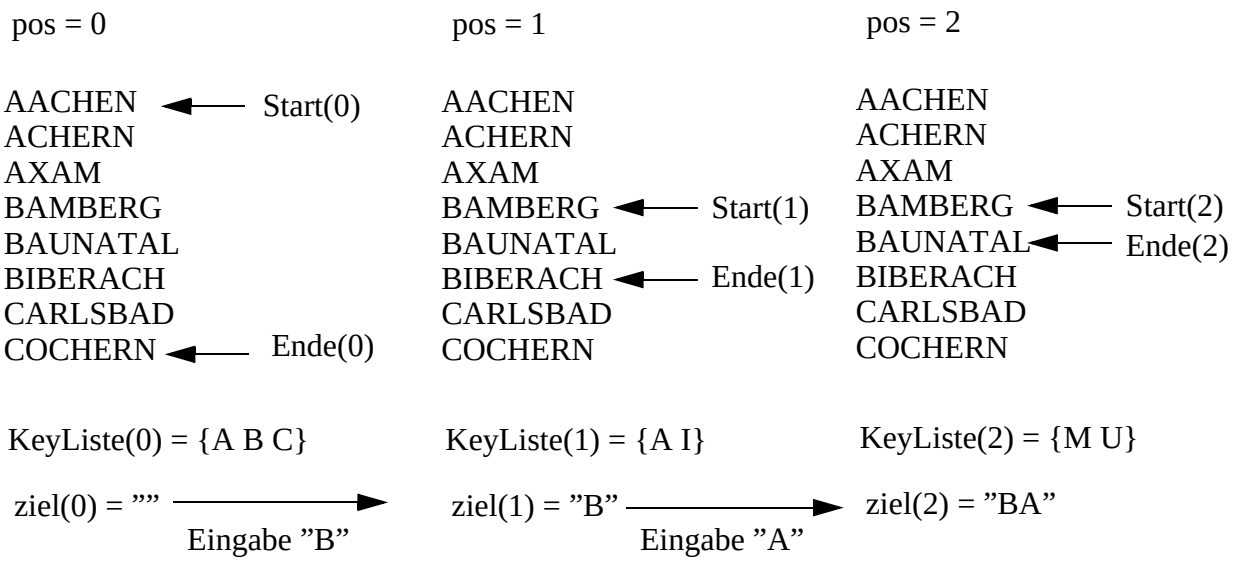
Aus der Liste der Ziele zwischen Start und Ende bestimmen wir das Alphabet (die Liste) der möglichen Anfangszeichen *KeyListe*($*pos*). Im Fall der Straßennamen von Kassel sind das z. B. alle Buchstaben außer „X“ „Ä“ „Ü“ (keine der 1347 Straßen in Kassel fängt mit X, Ä oder Ü an, ferner nie eine Straße mit „-“, „ß“ oder Leerzeichen). Das Alphabet *KeyListe*($*pos*) erstellen wir, indem wir in *lines* von *Start*($*pos*) bis *Ende*($*pos*) alle Einträge an der Zeichenkettenposition $*pos* scannen und gefundene Zeichen, sofern sie noch nicht in *KeyListe* sind, aufnehmen.

Diese *KeyListe*($*pos*) verwenden wir, um die Tastatur für die Eingabe des Zeichens für *ziel* (an

der Position $*pos*) neu zu konfigurieren. Alle zuvor aktiven Tasten werden deaktiviert, alle Tasten aus der neuen *KeyListe*($*pos*) werden dunkelblau eingefärbt und für `<ButtonPress>` aktiviert (mittels `bind`).



Das eingelesene Zeichen *key* wird an *ziel* angefügt und angezeigt. Die Liste *lines* wird jetzt für das neue Zeichen *key* eingeschränkt, indem nur die Einträge berücksichtigt werden, die auf Position $*pos* den Wert $*key* haben. Dies wird in *Start*($*pos* + 1) bzw. *Ende*($*pos* + 1) festgehalten. Für diese eingeschränkte Liste *lines* wird eine neue *KeyListe*($*pos* + 1) berechnet und zuletzt *pos* um eins erhöht. Allerdings erfolgt dies nur, wenn der noch abzusuchende Bereich von *liste* mehr als einen Eintrag hat, sonst wird dieser als Ergebnis angezeigt. In der Praxis würde man eher bei fünf Einträgen auf eine Listendarstellung wechseln. Die folgende Abbildung zeigt das Verfahren für eine kleine Städteliste mit 8 Einträgen.



Die einzige wirkliche Komplikation ergibt sich durch die Löschen-Taste, mit der wir das Zeichen auf der Position $*pos* - 1 löschen und den Vorgang mit der Eingabe für $*pos* - 1 fortsetzen. Dazu müssen wir auch *Start* und *Ende* wieder auf den Stand von $*pos* - 1 zurücksetzen, die *KeyLi-*

ste($*pos* - 1) entweder neu berechnen oder, falls gespeichert, wieder hernehmen und daraus die Tastatur neu aktivieren.

Ergänzen Sie die Lücken in der Prozedur zum Reduzieren der Suchliste und in der Prozedur, die die Tasten ein- und ausschaltet!

```
##Suche in der Suchliste von Start bis Ende alle Einträge, die an
#der Position P das Zeichen letter haben und liefere die neuen Start-
#und Endewerte zurück.
proc ShrinkList {Suchliste P Start Ende letter} {
  set s $Start
  if {$letter == "Space"} {set letter " "}
  while {[string compare [string index [lindex $Suchliste $s] $P] \
        "$letter"] < 0} {incr s}
  set e $Ende
  while ____________________________________________________________\
    ____________________________________________________________\
    {incr e -1}
  #Start-/Endewert als Zweierliste zurückliefern.
  return [list _____________________]
}

#alle alten Tasten in OldKList auf inaktiv setzen, neue aktivieren
proc AdjustKeyboard {OldKList NewKList} {
  #alle auf lightblue setzen mit leerem bind-Kommando.
  foreach ____________________________________ {
    ____________________________________________________________
    ____________________________________________________________
  }
  foreach tag $NewKList {
    .c itemconfigure $tag -fill darkblue
    .c bind "Sensor$tag" <ButtonPress> "add2ziel $tag"
  }
}
```

Aufgabe 2:

Die Touchscreen-Gestaltung ist dem tatsächlichen Kontroll-Panel eines Kraftfahrzeugs nachempfunden, das eine 7-Zoll-Diagonale (ca. 18 cm) und eine Auflösung von B x H 800 x 480 Pixel hat. Wir simulieren dies mit einem Canvas-Widget gleicher Dimension.

Das Grundmuster sieht eine Aufteilung in sieben Zeilen vor, jede mit 60 Pixel Höhe mit sechs Abständen zu je 10 Pixel dazwischen, damit 7 * 60 + 6 * 10 = 480 Pixel. Davon sind die oberste und unterste Reihe für Statusinformationen vorgesehen, die fünf mittleren Reihen lassen sich über farbliche Unterlegung in Tasten aufteilen, wie oben abgebildet.

Das Grundlayout der Zeilen wird in den fünf Listen *TLine*(0) bis *TLine*(4) festgelegt. Jede Taste besteht aus einem farbigen Rechteck, einer zentriert angeordneten Beschriftung und einem unsichtbaren Sensorfeld darüber. Jede Tastenzeile *i* hat einen Abstand *xdist*(*i*) für die erste Taste vom linken Rand.

(a) Ergänzen Sie den Teil für die Beschriftung. Der Font ist „Arial 18 bold“, die Farbe weiß. Der Tag beginnt immer mit Text..., also `TextA` für den Tag des Texts auf der A-Taste.

```
#Listenstruktur: Beschriftung = Tag, danach Tastenbreite in Pixel
set TLine(0) {Zielname 340}
set TLine(1) {A 60 B 60 C 60 D 60 E 60 F 60 G 60 H 60 I 60 J 60}
...
set TLine(4) {Mehr 90 0-9 60 Ä 60 Space 270 Ö 60 Ü 60 Liste 90}
set xdist(0) 240; set xdist(1) 30; set xdist(2) 65; set xdist(3) 30
set xdist(4) 30
#eine Reihe Tasten setzen, L Tastenliste, Tasten 60 hoch, x-Abstand 10
#xoffset Abstand links für erste Taste, yoffset Abstand von oben
proc setKeys {L xoffset yoffset} {
  set n [expr [llength $L] / 2]
  set x $xoffset
  for {set i 0} {$i < $n} {incr i} {
    set keywidth [lindex $L [expr $i*2 + 1]]
    .c create rectangle $x  $yoffset [expr $x + $keywidth] \
      [expr $yoffset + 60] \
      -outline "" -fill darkblue -tag "[lindex $L [expr $i*2]]"
    .c create text [expr $x + __________ / 2] [expr _____________ + 30] \
      -fill _______ -text [myprint [lindex $L [expr $i*2]]] \
      -font "__________________" -tag "Text[_____________________________]"
    #unsichtbares Rechteck drüberlegen für Bindings
    .c create rectangle $x  $yoffset [expr $x + $keywidth] \
      [expr $yoffset + 60] \
      -outline "" -fill "" -tag "Sensor[lindex $L [expr $i*2]]"
    set x [expr _________________________ + 10]
  }
}
```

(b) Rufen Sie `setKeys` mit den passenden Parametern für die fünf mittleren Zeilen *TLine*(*i*) auf (vgl. Abbildungen oben).

```
#oberste graue Statuszeile
.c create rectangle 0 0 800 60 -outline "" -fill lightgray
for {set i 0} {$i < 5} {incr i} {
  setKeys ____________________________________________________
}
```

Aufgabe 3:

Auf der untersten Statuszeile gibt es ganz rechts einen „Zurück-Pfeil“, der mittels

```
.c create text 750 450 -text "ç" -tag zurueck\
      -fill black -font "Wingdings 18 bold"
```

angelegt wurde. Legen Sie ein Sensorrechteck 60 x 60 mittig drüber und binden Sie die Taste an das `exit`-Kommando über einen Tag `Sensorzurueck`.

Aufgabe 4:

Auf der untersten Zeile sehen Sie links eine Zeitanzeige (Stunden:Minuten). Ergänzen Sie die Prozedur `setclock`, damit die Zeitangabe jede Sekunde gesetzt wird.

```
.c create text 20 450 -text "--:--" -tag __________ -fill black \
      -anchor w -font "Arial 18"

#Uhr unten links einmal pro Sekunde aktualisieren
proc setclock {} {
  .c itemconfigure __________ -text [clock format [clock seconds]\
        -format %R]
  update
  after ____________________________________________
}
setclock
```

Aufgabe 5:

Für die Simulation hier kommen die Straßennamen aus der Datei `sortedstrvz.txt`. Ergänzen Sie den Einlesevorgang und die Umwandlung in Großbuchstaben (in der Praxis wird man darauf achten, dass diese Umsetzung bereits erfolgt ist).

```
#Sortierte (!) Auswahlliste einlesen - hier Straßenverzeichnis Kassel
set fl [_____________ "sortedstrvz.txt"]
set data [read $fl];# liest alle Daten auf einmal
close $fl
#trennt grosse Zeichenkette in Liste per New Line
set lines [split $data \n]
#sofort auf Grossbuchstaben wandeln
set n [_____________________________];# Länge der Liste
for {set i 0} {$i < $n} {incr i} {
  set lines [lreplace $lines $i $i [string toupper \
              [_____________________________________]]]
}
```

Aufgabe 6:

In Aufgabe 2 (a) taucht eine Prozedur `myprint` auf, die den Text auf den Tasten modifiziert. Auslöser ist die Leertaste (`Space`) in *TLine*(4) und das Feld für `Zielnamen` in *TLine(*0), die ein Leerzeichen erhalten sollen. Ergänzen Sie die Lücken!

```
#Leertaste sollte leer sein, Leerzeichen als Tag schwierig
proc myprint {c} {
    return [expr {$c == "Space" || $c == "Zielname" ? ____ : ___ }]
}
```

Aufgabe 7:

Wir berechnen die Liste der zu aktivierenden Tasten.

```
#berechne eine Buchstabenliste für den Bereich der Suchliste,
#der von Start bis Ende geht; alle Zeichen links von Position P
#sind gleich (gleiches Präfix bis P-1)
#erstes Zeichen in einem String hat Index 0
#string index $S $j für j >= end liefert ""
#lindex auf leerer Liste liefert leeren String

proc ComputeKeyList {Suchliste P Start Ende} {
  set RListe {}
  for {set i $Start} {$i <= $Ende} {incr i} {
    set c [string index [_____________________________________] $P]
    set c [expr {$c == " " ? "Space" : $c}]
    if {[string compare [lindex $RListe end] $c] != 0} {
      lappend _____________________________
    }
  }
  return $RListe
```

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2013

MUSTERLÖSUNG

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte max.	Punkte
1	6	
2	6+3	
3	4	
4	3	
5	3	
6	2	
7	3	
Summe	30	

Aufgabe 1:

Bei Navigationsgeräten wird für die Eingabe von Stadt, Straße, Hausnummer oder Postleitzahl eine Auswahlliste von noch möglichen Zeichen angeboten, die – in Abhängigkeit des bereits eingegebenen Anfangsstücks – sich aus der Liste der Städte, Straßen usw. ergibt. Bei einer Touchscreen-Eingabe bietet sich eine intelligente, virtuelle Tastatur an, bei der deaktivierte Tasten z. B. hellblau, noch wählbare – und somit aktive Tasten – dunkelblau erscheinen.

Die von uns für diese Aufgabe ausgewählte Datenstruktur und Verfeinerungsmethode funktionieren hinreichend schnell in Tcl/Tk, wäre aber für die Suche in einem Wörterbuch mit mehreren Hunderttausenden Einträgen zu langsam.

Als Ausgangslage ist eine sortierte Liste der Suchwörter (in Großbuchstaben, Leerzeichen, Bindestrich und „ß“) als Datei gegeben und wird in den Speicher als Tcl-Liste *lines* eingelesen. Listen in Tcl beginnen mit Index 0 und der letzte Eintrag hat den Index `[llength $lines] - 1`.

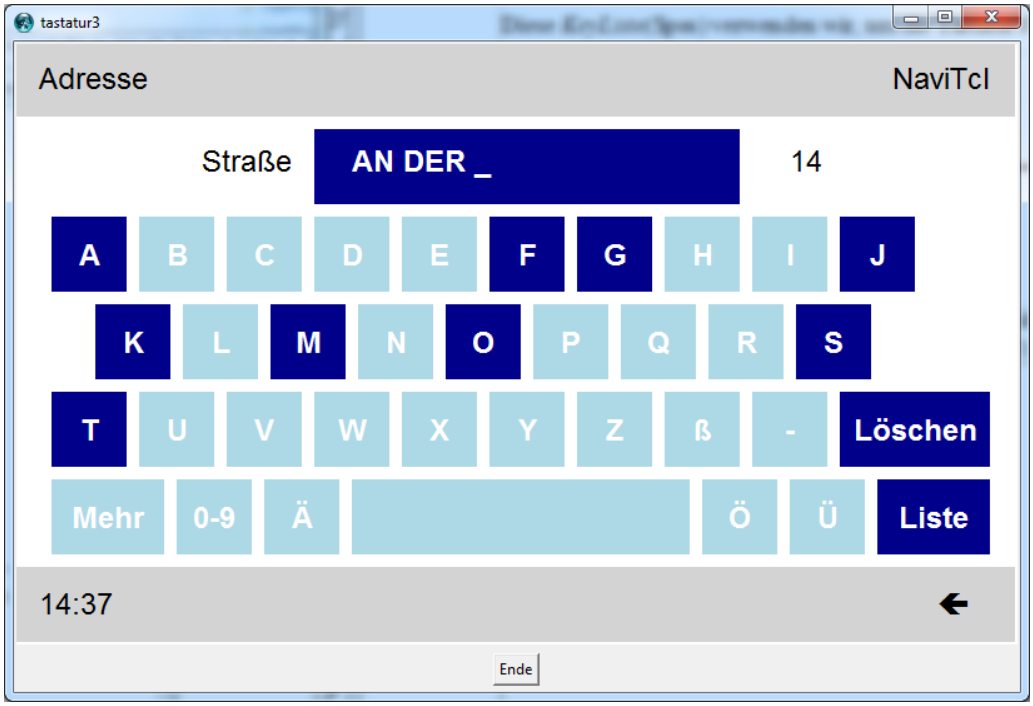
Die einzutippende Zeichenfolge wird in der Variablen *ziel* festgehalten. Zeichen in Zeichenketten (strings) beginnen auch mit dem Index 0. Generell liefert `[string index $ziel $i]` dann das *i*-te Zeichen in *ziel* für 0 <= *i* <= (|*ziel*| - 1).

Wir treffen die Vereinbarung, dass eine Variable (Zähler) *pos* die Iteration steuert. Wir beginnen mit *pos* = 0 für die Eingabe des ersten Zeichens. Zu diesem Zeitpunkt steht die gesamte Liste *lines* zur Auswahl, d. h. *Start*($*pos*) = 0 und *Ende*($*pos*) zeigen auf Anfang und Ende der gesamten Liste.

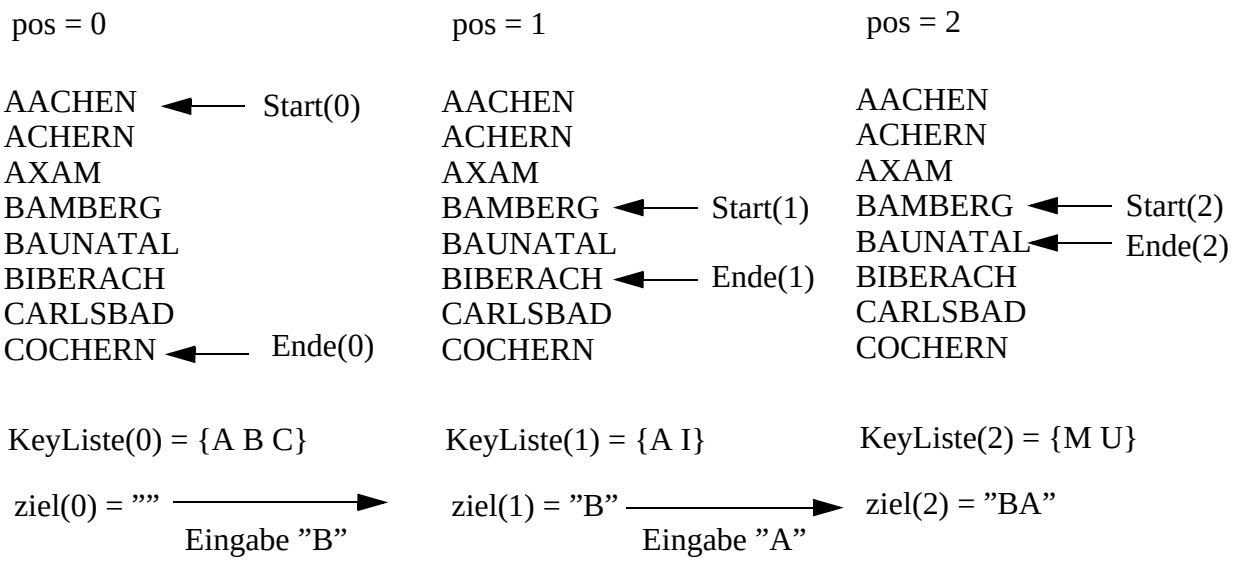
Aus der Liste der Ziele zwischen Start und Ende bestimmen wir das Alphabet (die Liste) der möglichen Anfangszeichen *KeyListe*($*pos*). Im Fall der Straßennamen von Kassel sind das z. B. alle Buchstaben außer „X“ „Ä“ „Ü“ (keine der 1347 Straßen in Kassel fängt mit X, Ä oder Ü an, ferner nie eine Straße mit „-“, „ß“ oder Leerzeichen). Das Alphabet *KeyListe*($*pos*) erstellen wir, indem wir in *lines* von *Start*($*pos*) bis *Ende*($*pos*) alle Einträge an der Zeichenkettenposition $*pos* scannen und gefundene Zeichen, sofern sie noch nicht in *KeyListe* sind, aufnehmen.

Diese *KeyListe*($*pos*) verwenden wir, um die Tastatur für die Eingabe des Zeichens für *ziel* (an

der Position $*pos*) neu zu konfigurieren. Alle zuvor aktiven Tasten werden deaktiviert, alle Tasten aus der neuen *KeyListe*($*pos*) werden dunkelblau eingefärbt und für `<ButtonPress>` aktiviert (mittels `bind`).



Das eingelesene Zeichen *key* wird an *ziel* angefügt und angezeigt. Die Liste *lines* wird jetzt für das neue Zeichen *key* eingeschränkt, indem nur die Einträge berücksichtigt werden, die auf Position $*pos* den Wert $*key* haben. Dies wird in *Start*($*pos* + 1) bzw. *Ende*($*pos* + 1) festgehalten. Für diese eingeschränkte Liste *lines* wird eine neue *KeyListe*($*pos* + 1) berechnet und zuletzt *pos* um eins erhöht. Allerdings erfolgt dies nur, wenn der noch abzusuchende Bereich von *liste* mehr als einen Eintrag hat, sonst wird dieser als Ergebnis angezeigt. In der Praxis würde man eher bei fünf Einträgen auf eine Listendarstellung wechseln. Die folgende Abbildung zeigt das Verfahren für eine kleine Städteliste mit 8 Einträgen.



Die einzige wirkliche Komplikation ergibt sich durch die Löschen-Taste, mit der wir das Zeichen auf der Position $*pos* - 1 löschen und den Vorgang mit der Eingabe für $*pos* - 1 fortsetzen. Dazu müssen wir auch *Start* und *Ende* wieder auf den Stand von $*pos* - 1 zurücksetzen, die *KeyLi-*

ste($*pos* - 1) entweder neu berechnen oder, falls gespeichert, wieder hernehmen und daraus die Tastatur neu aktivieren.

Ergänzen Sie die Lücken in der Prozedur zum Reduzieren der Suchliste und in der Prozedur, die die Tasten ein- und ausschaltet!

```
##Suche in der Suchliste von Start bis Ende alle Einträge, die an
#der Position P das Zeichen letter haben und liefere die neuen Start-
#und Endewerte zurück.
proc ShrinkList {Suchliste P Start Ende letter} {
  set s $Start
  if {$letter == "Space"} {set letter " "}
  while {[string compare [string index [lindex $Suchliste $s] $P] \
        "$letter"] < 0} {incr s}
  set e $Ende
  while {[string compare [string index [lindex $Suchliste $e] $P] "$letter"] > 0} \
        {incr e -1}
  #Start-/Endewert als Zweierliste zurückliefern.
  return [list $s $e]
}

#alle alten Tasten in OldKList auf inaktiv setzen, neue aktivieren
proc AdjustKeyboard {OldKList NewKList} {
  #alle auf lightblue und leerem bind-Kommando setzen.
  foreach tag $OldKList {
    .c itemconfigure $tag -fill lightblue
    .c bind "Sensor$tag" <ButtonPress> {}
  }
  foreach tag $NewKList {
    .c itemconfigure $tag -fill darkblue
    .c bind "Sensor$tag" <ButtonPress> "add2ziel $tag"
  }
}
```

Aufgabe 2:

Die Touchscreen-Gestaltung ist dem tatsächlichen Kontroll-Panel eines Kraftfahrzeugs nachempfunden, das eine 7-Zoll-Diagonale (ca. 18 cm) und eine Auflösung von B x H 800 x 480 Pixel hat. Wir simulieren dies mit einem Canvas-Widget gleicher Dimension.

Das Grundmuster sieht eine Aufteilung in sieben Zeilen vor, jede mit 60 Pixel Höhe mit sechs Abständen zu je 10 Pixel dazwischen, damit 7 * 60 + 6 * 10 = 480 Pixel. Davon sind die oberste und unterste Reihe für Statusinformationen vorgesehen, die fünf mittleren Reihen lassen sich über farbliche Unterlegung in Tasten aufteilen, wie oben abgebildet.

Das Grundlayout der Zeilen wird in den fünf Listen *TLine*(0) bis *TLine*(4) festgelegt. Jede Taste besteht aus einem farbigen Rechteck, einer zentriert angeordneten Beschriftung und einem unsichtbaren Sensorfeld darüber. Jede Tastenzeile *i* hat einen Abstand *xdist*(*i*) für die erste Taste vom linken Rand.

(a) Ergänzen Sie den Teil für die Beschriftung. Der Font ist „Arial 18 bold“, die Farbe weiß. Der Tag beginnt immer mit Text..., also `TextA` für den Tag des Texts auf der A-Taste.

```
#Listenstruktur: Beschriftung = Tag, danach Tastenbreite in Pixel
set TLine(0) {Zielname 340}
set TLine(1) {A 60 B 60 C 60 D 60 E 60 F 60 G 60 H 60 I 60 J 60}
...
set TLine(4) {Mehr 90 0-9 60 Ä 60 Space 270 Ö 60 Ü 60 Liste 90}
set xdist(0) 240; set xdist(1) 30; set xdist(2) 65; set xdist(3) 30
set xdist(4) 30
#eine Reihe Tasten setzen, L Tastenliste, Tasten 60 hoch, x-Abstand 10
#xoffset Abstand links für erste Taste, yoffset Abstand von oben
proc setKeys {L xoffset yoffset} {
  set n [expr [llength $L] / 2]
  set x $xoffset
  for {set i 0} {$i < $n} {incr i} {
    set keywidth [lindex $L [expr $i*2 + 1]]
    .c create rectangle $x  $yoffset [expr $x + $keywidth] \
      [expr $yoffset + 60] \
      -outline "" -fill darkblue -tag "[lindex $L [expr $i*2]]"
    .c create text [expr $x + __$keywidth__ / 2] [expr __$yoffset__ + 30] \
      -fill _white_ -text [myprint [lindex $L [expr $i*2]]] \
      -font "_Arial 18 bold_" -tag "Text[lindex $L [expr $i * 2]]"
    #unsichtbares Rechteck drüberlegen für Bindings
    .c create rectangle $x  $yoffset [expr $x + $keywidth] \
      [expr $yoffset + 60] \
      -outline "" -fill "" -tag "Sensor[lindex $L [expr $i*2]]"
    set x [expr $x + $keywidth + 10]
  }
}
```

(b) Rufen Sie `setKeys` mit den passenden Parametern für die fünf mittleren Zeilen *TLine*(*i*) auf (vgl. Abbildungen oben).

```
#oberste graue Statuszeile
.c create rectangle 0 0 800 60 -outline "" -fill lightgray
for {set i 0} {$i < 5} {incr i} {
  setKeys __$TLine($i) $xdist($i) [expr $i*70 + 70]__
}
```

Aufgabe 3:

Auf der untersten Statuszeile gibt es ganz rechts einen „Zurück-Pfeil“, der mittels

```
.c create text 750 450 -text "ç" -tag zurueck\
      -fill black -font "Wingdings 18 bold"
```

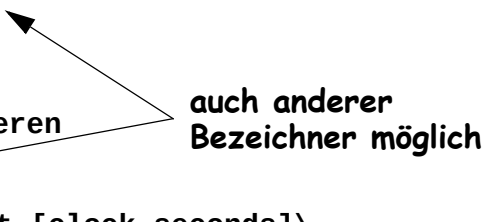
angelegt wurde. Legen Sie ein Sensorrechteck 60 x 60 mittig drüber und binden Sie die Taste an das `exit`-Kommando über einen Tag `Sensorzurueck`.

```
.c create rectangle 720 420 780 480 \
     -outline "" -fill "" -tag "Sensorzurueck"
.c bind "Sensorzurueck" <ButtonPress> "exit"
```

Aufgabe 4:

Auf der untersten Zeile sehen Sie links eine Zeitanzeige (Stunden:Minuten). Ergänzen Sie die Prozedur `setclock`, damit die Zeitangabe jede Sekunde gesetzt wird.

```
.c create text 20 450 -text "--:--" -tag _zeit_ -fill black \
      -anchor w -font "Arial 18"

#Uhr unten links einmal pro Sekunde aktualisieren
proc setclock {} {
  .c itemconfigure __zeit__ -text [clock format [clock seconds]\
        -format %R]
  update
  after __1000 setclock__
}
setclock
```



Aufgabe 5:

Für die Simulation hier kommen die Straßennamen aus der Datei `sortedstrvz.txt`. Ergänzen Sie den Einlesevorgang und die Umwandlung in Großbuchstaben (in der Praxis wird man darauf achten, dass diese Umsetzung bereits erfolgt ist).

```
#Sortierte (!) Auswahlliste einlesen - hier Straßenverzeichnis Kassel
set fl [__open__ "sortedstrvz.txt"]
set data [read $fl];# liest alle Daten auf einmal
close $fl
#trennt grosse Zeichenkette in Liste per New Line
set lines [split $data \n]
#sofort auf Grossbuchstaben wandeln
set n [__llength $lines__];# Länge der Liste
for {set i 0} {$i < $n} {incr i} {
  set lines [lreplace $lines $i $i [string toupper \
              [__lindex $lines $i__]]]
}
```

Aufgabe 6:

In Aufgabe 2 (a) taucht eine Prozedur `myprint` auf, die den Text auf den Tasten modifiziert. Auslöser ist die Leertaste (`Space`) in *TLine*(4) und das Feld für `Zielnamen` in *TLine(*0), die ein Leerzeichen erhalten sollen. Ergänzen Sie die Lücken!

```
#Leertaste sollte leer sein, Leerzeichen als Tag schwierig
proc myprint {c} {
    return [expr {$c == "Space" || $c == "Zielname" ? _" "_ : _$c_ }]
}
```

Aufgabe 7:

Wir berechnen die Liste der zu aktivierenden Tasten.

```
#berechne eine Buchstabenliste für den Bereich der Suchliste,
#der von Start bis Ende geht; alle Zeichen links von Position P
#sind gleich (gleiches Präfix bis P-1)
#erstes Zeichen in einem String hat Index 0
#string index $S $j für j >= end liefert ""
#lindex auf leerer Liste liefert leeren String

proc ComputeKeyList {Suchliste P Start Ende} {
  set RListe {}
  for {set i $Start} {$i <= $Ende} {incr i} {
    set c [string index [__lindex $Suchliste $i__] $P]
    set c [expr {$c == " " ? "Space" : $c}]
    if {[string compare [lindex $RListe end] $c] != 0} {
      lappend __RListe $c__
    }
  }
  return $RListe
```

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Wintersemester 2013/14

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte max.	Punkte
1	3	
2	3	
3	2	
4	3	
5	3 + 2	
6	3	
7	2	
8	4	
Summe	25	

Einleitung

Manche Klausuren sind das reinste Glücksspiel. In dieser Klausur betrachten wir eine stark vereinfachte Form des Kartenspiels **Black Jack** (verwandt mit **Siebzehn und Vier**). Bei uns spielt die **Bank** (*bank*) gegen genau einen **Spieler** (*player*). Gewonnen hat, wer am nächsten an 21 Punkten ist, ohne darüber zu liegen. Bei Punktgleichheit gewinnt die Bank.

Wir spielen mit einem sog. französischen **Blatt** (*deck*) zu **52 Karten** (*cards*). Dieses hat vier **Farben** (*suits*): **Pik** (*spade* **s**), **Herz** (*heart* **h**), **Caro** (*diamond* **d**), **Kreuz** (*club* **c**). Jede Farbe hat Karten mit den **Rängen** (*ranks*) **Zwei** (*two* **2**), **Drei** (*three* **3**), ..., **Neun** (*nine* **9**), **Zehn** (*ten* **t**), **Bube** (*jack* **j**), **Dame** (*queen* **q**), **König** (*king* **k**) und **Ass** (*ace* **a**).

Wir verwenden die einstelligen, englischen Abkürzungen für Farbe und Rang in der Reihenfolge **RangFarbe**, weil mit diesen Bezeichnungen in `card_img.tcl` Kartenbilder (*images*) für die Karten vorliegen. Also bezeichnet z. B. **3d** die Caro 3 (*3 of diamonds*), **th** die Herz 10 (*ten of hearts*), **qs** die Pik Dame (*queen of spades*). Die Abbildung unten zeigt die Spielsituation nach Austeilen der ersten vier Karten im **Spiel** (*game*).

Aufgabe 1:

Zunächst initialisieren wir einige Listen. Diese halten wir als Werte eines assoziativen Arrays (hashmap) **`data`**, den wir in allen Prozeduren als **`global`** deklarieren. In **`data(allcards)`** ist das Blatt (alle Karten) gespeichert, also „**`as ks qs ... 2c`**“.

Ergänzen Sie die Lücken!

```
set data(suits) "s h d c"
set data(ranks) "a k q j t 9 8 7 6 5 4 3 2"
set data(values) "11 10 10 10 10 9 8 7 6 5 4 3 2"
set data(allcards) {}
foreach suit ____________________________ {
  foreach rank ____________________________ {
    lappend data(allcards) "_____________________ "
  }
}
```

Aufgabe 2:

Zu Beginn eines Spiels werden die Karten gemischt. Das machen wir mit einer generischen Prozedur **`shuffleList`**. Der Befehl **lset** *varName ?index...? newValue* interpretiert *varName* als Name einer Listenvariablen und ersetzt darin an der Position *index* den Wert durch *newValue*. Ergänzen Sie die Lücken!

```
proc shuffleList { list } {
  set n [llength $list]
  for { set i 0 } { ____________________ } { incr i } {
    set j [expr { int( rand() * $n ) }]
    set temp [lindex ___________________________ ]
    lset list $i [lindex $list $j]
    lset list ___________________________
  }
  return $list
};# shuffleList
```

Aufgabe 3:

Durch welchen Befehl mit Aufruf von **`shuffleList`** ... kann man das Blatt **`$data(allcards)`** mischen und der Variablen **`data(deck)`** zuweisen?

Aufgabe 4:

Während des Spiels halten Bank und Spieler eine Liste von Karten, die wir in **`data(handbank)`** und **`data(handplayer)`** speichern. Wieviel Punkte (aus **`data(values)`**) eine Hand wert ist, prüfen wir mit der folgenden Prozedur, wobei nur der Rang der Karten zählt. Der Listenbefehl **lsearch** *?Optionen? Liste Muster* liefert uns dabei den Index für ein gesuchtes *Muster* in *Liste*. Der Stringbefehl **string index** *string index* liefert aus *string* das Zeichen an der Stelle *index* zurück (erstes Zeichen Index 0). Ergänzen Sie die Lücken!

```
proc getValue { hand } {
  global data
  set v 0
  foreach card $hand {
    set v [expr $v + [lindex ____________________________  \
      [lsearch -exact ____________  [string index $card ____ ]]]]
  }
  return $v
}
```

Aufgabe 5:

Die Spielkarten stellen wir als Bilditems auf der Leinwand (*canvas*) dar. Jedes Bild hat einen Tag „**tag$card**“, wobei **card** die Werte aus **data(allcards)** in der oben genannten Codierung annimmt. Zusätzlich gibt es noch ein Bild mit Tag „**back**“ für die Rückseite der Spielkarten, die natürlich immer gleich ist.

Beim Austeilen der Karten bewegen wir eine Karte verdeckt vom Stapel zum Zielort bei Bank oder Spieler. Am Zielort drehen wir die Karte um (außer die zweite bei der Bank, die zunächst verdeckt bleibt). Damit die Bewegung glatt und verlangsamt verläuft, gibt es eine Prozedur **moveCard**, die im Inneren die Leinwandmethode **move** verwendet.

(a) Ergänzen Sie die Lücken!

```
proc moveCard {c item tox toy {steps 10}} {
# modified from solitaire.tcl
  set orig [$c coords $item]
  set origx [lindex $orig 0]; set origy [lindex $orig 1]
  set diffx [expr {$tox - $origx}]
  set diffy [expr {$toy - $origy}]
  set stepx [expr {_____________________________________ }]
  set stepy [expr {_____________________________________ }]
  for {set i 1} {$i < $steps} {incr i} {
    $c move $item $stepx $stepy
    update idletasks
    after 50
  }
  $c coords ____________________ $tox $toy
};# move
```

(b) Warum werden hier – bei einer Bewegung in zehn Schritten – die ersten neun über **move** gemacht, der letzte Schritt aber über **coords**? Kurze Begründung genügt.

Aufgabe 6:

Zu Anfang eines Spiels legen wir 52 Rückseitenbilder übereinander. Jedes Bild hat den Tag der zugehörigen Spielkarte. Nachdem eine Karte zugedeckt an den Zielort bewegt wurde (siehe oben), rufen wir **`turnOver`** auf. Die Prozedur merkt sich den Ort des Bilds, wirft das Rückseitenbild weg und legt an seine Stelle das passende Vorderseitenbild an. Ergänzen Sie die Lücken!

```
proc turnOver { card } {
  global data
  set img ::img::$card; #Kartenbilder im Namensraum ::img

  set _____________  [.c coords "tag$card"]
  .c delete "tag_________________ "
  .c create image [lindex $xy 0] [lindex $xy 1] \
    -image _________________  -anchor nw -tags "tag$card"
}
```

Aufgabe 7:

Im Tcl-Manual zu Version 8.5 findet sich eine Prozedur **`lremove`** (kein Tcl-Befehl) zum Entfernen eines Listenwerts. Erklären Sie anhand des Programmcodes die Funktionsweise von **`lremove`** und den Unterschied zum Tcl-Befehl **`lreplace`**.

```
proc lremove {listVariable value} {
  upvar $listVariable var
  set idx [lsearch -exact $var $value]
  set var [lreplace $var $idx $idx]
}
```

Aufgabe 8:

Zuletzt betrachten wir noch die Routine **dealBank** zum Austeilen von Karten an die Bank. Die Routine teilt genau eine Karte aus, die vorne aus der Liste **data(deck)** entnommen wird und ans Ende der Liste **data(handbank)** angefügt wird. Die Zielkoordinaten für die animierte Kartenbewegung mit **moveCard** befinden sich für die erste Karte in **data(posx-bank)** und **data(posybank)**. Ergänzen Sie die Lücken!

```
proc dealBank { } {
  global data
  set card [lindex $data(deck) _________________]
  lappend data(handbank) _________________
  set len [llength ____________________________]
  lremove data(deck) $card
  .c raise "tag$card"
  moveCard .c "____________________" [expr {$data(posxbank) + \
    ($len - 1) * 20}] $data(posybank) 10; #Bewegung in 10 Schritten
  if {_____________________________ != 2} {
    #zweite Karte nicht umdrehen
    turnOver $card
  }
}
```

ENDE DER KLAUSUR

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Wintersemester 2013/14

Nachname: Vorname:

Matr.Nr. Studiengang:

Musterlösung

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte max.	Punkte
1	3	
2	3	
3	2	
4	3	
5	3 + 2	
6	3	
7	2	
8	4	
Summe	25	

Einleitung

Manche Klausuren sind das reinste Glücksspiel. In dieser Klausur betrachten wir eine stark vereinfachte Form des Kartenspiels **Black Jack** (verwandt mit **Siebzehn und Vier**). Bei uns spielt die **Bank** (*bank*) gegen genau einen **Spieler** (*player*). Gewonnen hat, wer am nächsten an 21 Punkten ist, ohne darüber zu liegen. Bei Punktgleichheit gewinnt die Bank.

Wir spielen mit einem sog. französischen **Blatt** (*deck*) zu **52 Karten** (*cards*). Dieses hat vier **Farben** (*suits*): **Pik** (*club* **c**), **Herz** (*heart* **h**), **Caro** (*diamond* **d**), **Kreuz** (*club* **c**). Jede Farbe hat Karten mit den **Rängen** (*ranks*) **Zwei** (*two* **2**), **Drei** (*three* **3**), ..., **Neun** (*nine* **9**), **Zehn** (*ten* **t**), **Bube** (*jack* **j**), **Dame** (*queen* **q**), **König** (*king* **k**) und **Ass** (*ace* **a**).

Wir verwenden die einstelligen, englischen Abkürzungen für Farbe und Rang in der Reihenfolge **RangFarbe**, weil mit diesen Bezeichnungen in `card_img.tcl` Kartenbilder (*images*) für die Karten vorliegen. Also bezeichnet z. B. **3d** die Caro 3 (*3 of diamonds*), **th** die Herz 10 (*ten of hearts*), **qs** die Pik Dame (*queen of spades*). Die Abbildung unten zeigt die Spielsituation nach Austeilen der ersten vier Karten im **Spiel** (*game*).

Aufgabe 1:

Zunächst initialisieren wir einige Listen. Diese halten wir als Werte eines assoziativen Arrays (hashmap) **`data`**, den wir in allen Prozeduren als **`global`** deklarieren. In **`data(allcards)`** ist das Blatt (alle Karten) gespeichert, also „**`as ks qs ... 2c`**“.

Ergänzen Sie die Lücken!

```
set data(suits) "s h d c"
set data(ranks) "a k q j t 9 8 7 6 5 4 3 2"
set data(values) "11 10 10 10 10 9 8 7 6 5 4 3 2"
set data(allcards) {}
foreach suit ____$data(suits)____ {
  foreach rank ____$data(ranks)____ {
    lappend data(allcards) "_____$rank$suit_____"
  }
}
```

Aufgabe 2:

Zu Beginn eines Spiels werden die Karten gemischt. Das machen wir mit einer generischen Prozedur **`shuffleList`**. Der Befehl **lset** *varName ?index...? newValue* interpretiert *varName* als Name einer Listenvariablen und ersetzt darin an der Position *index* den Wert durch *newValue*. Ergänzen Sie die Lücken!

```
proc shuffleList { list } {
  set n [llength $list]
  for { set i 0 } { ___$i < $n___ } { incr i } {
    set j [expr { int( rand() * $n ) }]
    set temp [lindex ___$list $i___]
    lset list $i [lindex $list $j]
    lset list ___$j $temp___
  }
  return $list
};# shuffleList
```

Aufgabe 3:

Durch welchen Aufruf von **`shuffleList`** ... kann man das Blatt **`$data(allcards)`** mischen und der Variablen **`data(deck)`** zuweisen?

```
set data(deck) [shuffleList $data(allcards)]
```

Aufgabe 4:

Während des Spiels halten Bank und Spieler eine Liste von Karten, die wir in **`data(handbank)`** und **`data(handplayer)`** speichern. Wieviel Punkte (aus **`data(values)`**) eine Hand wert ist, prüfen wir mit der folgenden Prozedur, wobei nur der Rang der Karten zählt. Der Listenbefehl **lsearch** *?Optionen? Liste Muster* liefert uns dabei den Index für ein gesuchtes *Muster* in *Liste*. Der Stringbefehl **string index** *string index* liefert aus *string* das Zeichen an der Stelle *index* zurück (erstes Zeichen Index 0). Ergänzen Sie die Lücken!

```
proc getValue { hand } {
  global data
  set v 0
  foreach card $hand {
    set v [expr $v + [lindex ___$data(values)____ \
      [lsearch -exact ___$data(ranks)___ [string index $card __0__]]]]
  }
  return $v
}
```

Aufgabe 5:

Die Spielkarten stellen wir als Bilditems auf der Leinwand (*canvas*) dar. Jedes Bild hat einen Tag „**tag$card**“, wobei **card** die Werte aus **data(allcards)** in der oben genannten Codierung annimmt. Zusätzlich gibt es noch ein Bild mit Tag „**back**“ für die Rückseite der Spielkarten, die natürlich immer gleich ist.

Beim Austeilen der Karten bewegen wir eine Karte verdeckt vom Stapel zum Zielort bei Bank oder Spieler. Am Zielort drehen wir die Karte um (außer die zweite bei der Bank, die zunächst verdeckt bleibt). Damit die Bewegung glatt und verlangsamt verläuft, gibt es eine Prozedur **moveCard**, die im Inneren die Leinwandmethode **move** verwendet.

(a) Ergänzen Sie die Lücken!

```
proc moveCard {c item tox toy {steps 10}} {
# modified from solitaire.tcl
  set orig [$c coords $item]
  set origx [lindex $orig 0]; set origy [lindex $orig 1]
  set diffx [expr {$tox - $origx}]
  set diffy [expr {$toy - $origy}]
  set stepx [expr {___$diffx / $steps___}]
  set stepy [expr {___$diffy / $steps___}]
  for {set i 1} {$i < $steps} {incr i} {
    $c move $item $stepx $stepy
    update idletasks
    after 50
  }
  $c coords ____$item___ $tox $toy
};# move
```

(b) Warum werden hier – bei einer Bewegung in zehn Schritten – die ersten neun über **move** gemacht, der letzte Schritt aber über **coords**? Kurze Begründung genügt.

Bei der Berechnung der Einzeldistanzen für die Bewegung kann es durch die ganzzahlige Division zu Abweichungen kommen, die sich bei 10 Schritten zu mehreren Pixeln Unterschied aufsummieren können. Daher bewegt der letzte Schritt das Item an die genaue Zielposition.

Aufgabe 6:

Zu Anfang eines Spiels legen wir 52 Rückseitenbilder übereinander. Jedes Bild hat den Tag der zugehörigen Spielkarte. Nachdem eine Karte zugedeckt an den Zielort bewegt wurde (siehe oben), rufen wir **turnOver** auf. Die Prozedur merkt sich den Ort des Bilds, wirft das Rückseitenbild weg und legt an seine Stelle das passende Vorderseitenbild an. Ergänzen Sie die Lücken!

```
proc turnOver { card } {
  global data
  set img ::img::$card; #Kartenbilder im Namensraum ::img
  set ___xy___ [.c coords "tag$card"]
  .c delete "tag$card"
  .c create image [lindex $xy 0] [lindex $xy 1] \
    -image ___$img___ -anchor nw -tags "tag$card"
}
```

Aufgabe 7:

Im Tcl-Manual zu Version 8.5 findet sich eine Prozedur **lremove** (kein Tcl-Befehl) zum Entfernen eines Listenwerts. Erklären Sie anhand des Programmcodes die Funktionsweise von **lremove** und den Unterschied zum Tcl-Befehl **lreplace**.

```
proc lremove {listVariable value} {
  upvar $listVariable var
  set idx [lsearch -exact $var $value]
  set var [lreplace $var $idx $idx]
}
```

lremove will als erstes Argument eine Listenvariable, als zweites Argument aber einen Wert in der Liste, keinen Listenindex. Der Wert wird dann über eine Suche mit lsearch, die ggf. den Index findet, und lreplace aus der Liste entfernt.
lreplace nimmt hingegen als erstes Argument eine Liste und fügt neue Elemente ein bzw. löscht Elemente in Abhängigkeit der Indexwerte.
Die geänderte Liste wird zurückgegeben.

Aufgabe 8:

Zuletzt betrachten wir noch die Routine **dealBank** zum Austeilen von Karten an die Bank. Die Routine teilt genau eine Karte aus, die vorne aus der Liste **data(deck)** entnommen wird und ans Ende der Liste **data(handbank)** angefügt wird. Die Zielkoordinaten für die animierte Kartenbewegung mit **moveCard** befinden sich für die erste Karte in **data(posx-bank)** und **data(posybank)**. Ergänzen Sie die Lücken!

```
proc dealBank { } {
  global data
  set card [lindex $data(deck) __0__]
  lappend data(handbank) __$card__
  set len [llength ___$data(handbank)___]
  lremove data(deck) $card
  .c raise "tag$card"
  moveCard .c "___tag$card___" [expr {$data(posxbank) + \
    ($len - 1) * 20}] $data(posybank) 10; #Bewegung in 10 Schritten
  if {___$len___ != 2} {
    #zweite Karte nicht umdrehen
    turnOver $card
  }
}
```

ENDE DER KLAUSUR

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2014

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

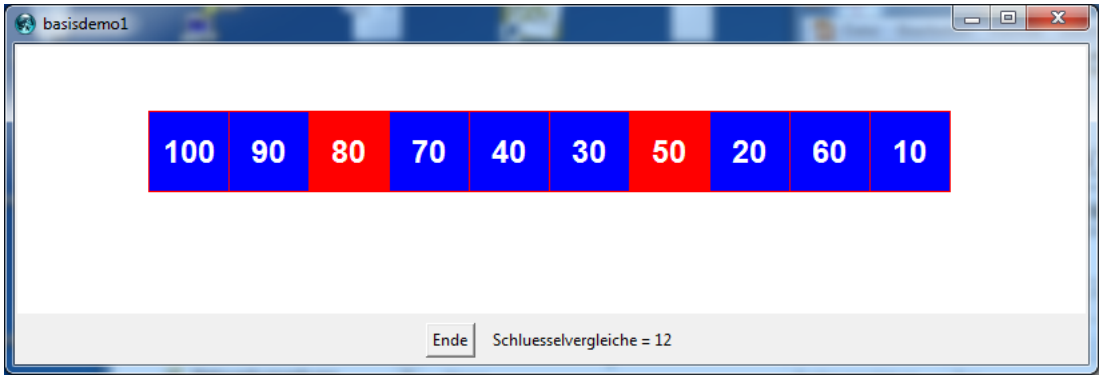
Aufgabe	Punkte max.	Punkte
1	4	
2	5	
3	5	
4	2	
5	4	
6	4	
7	3	
8	5	
Summe	32	

Aufgabe 1:

Visualisierungen von Algorithmen mittels animierter Grafiken sind sehr populär. Auf den Wikipedia-Seiten findet man *animated gifs* für verschiedene Sortierverfahren. Natürlich kann man solche Animationen auch gut mit Tcl/Tk produzieren.

Hier geht es um eine animierte Darstellung des Heapsort. Den Algorithmus müssen Sie nicht im Detail verstehen. Er arbeitet mit Schlüsselvergleichen und Vertauschungen von Arrayeinträgen. Wir stellen das Array dar und immer wenn der Algorithmus eine Vertauschung macht (Aufruf der Prozedur `swap(i, j)`), färben wir die Felder i und j neu.

In der *Base Edition* unserer Implementierung gibt es einen Ende-Knopf zum Beenden des Programms, eine Anzeige für die gemessene Anzahl von Schlüsselvergleichen und eine Leinwand mit dem Array (für das wir in Tcl eine Listenvariable nehmen). Die folgende Abbildung zeigt die Visualisierung dieser Sparversion mit einem Array von 10 Zahlen.



Wir beginnen mit dem Anlegen der Widgets und dem Initialisieren der Daten. Ergänzen Sie die Lücken!

```
set keycounter 0
set xoffset 100; set yoffset 50
canvas .c -width 800 -height 200 -bg white
pack .c
frame .f
pack .f
button .f.cancel -text ___________ -command {_______________}
pack .f.cancel -padx 5 -pady 5 -side left
_________ .f.anzeige1 -text "_____________________________________________"
pack .f.anzeige1 -padx 5 -pady 5 -side left

set input {50 10 30 70 90 100 80 20 60 40}
set N [llength $input]
```

Aufgabe 2:

Jetzt geht es um die Prozedur zum Anlegen des „Arrays“. Der Aufruf mit den Daten von oben lautet **`createArray $input`**.

Füllen Sie die Lücken und achten Sie darauf, dass die Zahlen zentriert in den 60x60 Pixel großen Rechtecken stehen. Ergänzen Sie die Lücken zu den Tags, damit wir jedes Rechteck und den Text darin individuell ändern können.

```
#create the array as sequence of rectangles, say 60 x 60 with a
#slim border and number inside. Start on left with xoffset and
#yoffset from top.
proc createArray {L} {
  global xoffset yoffset
  set n [llength $L]
  set x $xoffset
  for {set i 0} {$i < $n} {incr i} {
    .c create rectangle $x  $yoffset [expr $x + __________] \
        [expr $yoffset + _____________] \
        -outline "red" -fill blue -tag "field-$i"
    .c create text [expr $x + ________] [expr $yoffset + ________] \
        -fill white -text [lindex _____ _____] -font "Arial 18 bold" \
        -tag "number-____________"
    set x [expr $x + 60]
  }
}
```

Aufgabe 3:

In der Liste **`L`** findet die tatsächliche Sortierung statt. Den jeweiligen Fortschritt zeigen wir mit der Routine **`showArray`**. Gleichzeitig aktualisieren wir den Stand des Schlüsselvergleichszählers. Ergänzen Sie die Lücken.

```
proc showArray {L} {
  global keycounter
  set n [llength $L]
  for {set i 0} {$i < $n} {incr i} {
    .c _______________ __________ -text "_____________________________"
  }
  .f.anzeige1 _____________ -text "_____________________________________"
  update
}
```

Aufgabe 4:

Wichtig für die Visualisierung ist das Umschalten der Farbe von blau zu rot für die Felder, deren Werte getauscht werden. Zuerst färben wir die beiden Felder auf rot, warten dann eine Sekunde, tauschen die Werte, warten wieder eine Sekunde, färben dann beide Felder zurück auf blau.

Ergänzen Sie die Lücken in den beiden Prozeduren zum Ändern der Farben. Die Aufrufe sehen Sie in der Prozedur **swap** in Aufgabe 5.

```
proc showSwapOn {i j} {
  .c itemconfigure "____________________" -fill "red"
  .c itemconfigure "____________________" -fill "red"
  update
}

proc showSwapOff {i j} {
  .c itemconfigure "____________________" -fill "blue"
  .c itemconfigure "____________________" -fill "blue"
  update
}
```

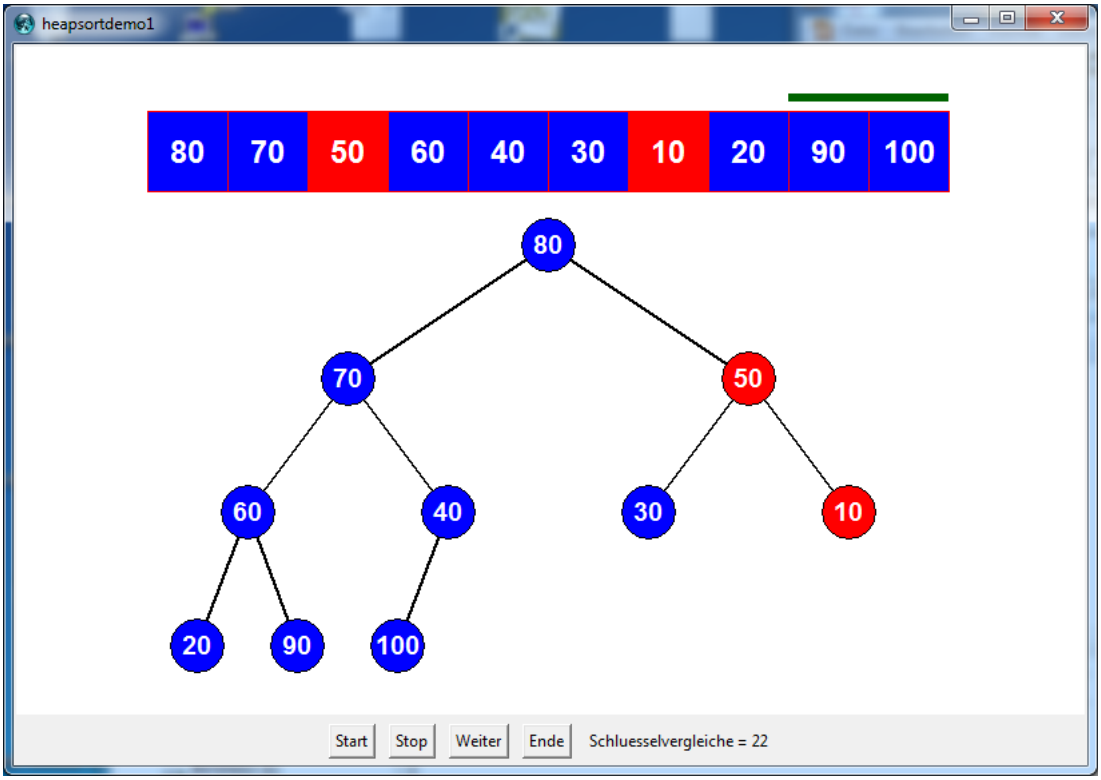
Aufgabe 5:

In der Prozedur **swap** vertauschen wir die beiden Werte an den Positionen **x** und **y** in der Liste **varName** und rufen die Prozeduren zur Farbumschaltung auf. Ergänzen Sie die Lücken! Hinweis: **lset** *varName index value* ersetzt in der Liste *varName* den Wert an der Position *index* durch *value*.

```
proc swap {varName x y} {
  upvar 1 _______________ _______________
  showSwapOn _____________ _______________
  after 1000
  set tmp [lindex $a $x]
  lset a $x [lindex $a $y]
  lset a $y $tmp
  showArray $a
  after 1000
  showSwapOff ____________ _______________
}
```

Aufgabe 6:

In der *Professional Edition* unserer Visualisierung zeigen wir auch den Heap, der ja ein vollständiger Binärbaum ist und ebenenweise gefüllt wird. Die Abbildung unten zeigt die Ausgabe des Programms, ein Balken oberhalb des Felds visualisiert die am rechten Ende bereits sortierte Folge. Über weitere Knöpfe kann die Visualisierung angehalten und danach wieder fortgesetzt werden.



Das Malen des Baums hat zwei Teile. Zum Einen berechnen wir in der Prozedur **`compute_node_locs`** die Positionen der Knoten im Baum (Zentren der Kreise). Diese speichern wir in zwei Arrays **`Xloc`** und **`Yloc`**, die Schlüssel sind die Zahlen von 0 bis N - 1, hier 9.

Der andere Teil ist das Anlegen von Knoten und Kanten mittels der Prozedur **`drawTree`** aus Aufgabe 8, wenn die Positionen der Knoten bekannt sind. Leicht ist dann auch das Ändern der Knotenfarben und angezeigten Werte (Texte): wir verwenden dieselben Tags wie für die Felder in der Liste darüber. Ein einziges **`itemconfigure`** ändert dann an beiden Stellen.

Erläutern Sie das Bildungsgesetz für die Berechnung der X-Y-Koordinaten in **`compute_node_locs`**. Im Beispiel der Abbildung oben ist **`xoffset`** 100, **`yoffset`** 50 und die Leinwand **`.c`** hat Breite 800 und Höhe 500.

```
proc compute_node_locs {n} {
#returns x, y node locations for a heap with n nodes
  global Xloc Yloc xoffset yoffset
  set w [expr [.c cget -width] - 2 * $xoffset]
  set y [expr {$yoffset + 100}]
  set i 0; set j 1
```

```
  while {$i < $n} {
    set b [expr $w/$j]
    for {set k 0} {$k < $j} {incr k} {
      set Xloc($i) [expr $b / 2 + $b*$k + $xoffset]
      set Yloc($i) $y
      incr i
    }
    set j [expr 2 * $j]
    set y [expr $y + 100]
  }
}
```

Erläuterung:

Aufgabe 7:

Das Anhalten mit dem Knopf **`Stop`**, bzw. Weitermachen mit **`Weiter`** regeln wir über **`vwait`** und eine Variable **`weiter`**, die wir anfangs z.B. auf **`true`** setzen. Ergänzen Sie die Lücken für die *command*-Skripten der beiden Knöpfe!

```
button .f.stop -text Stop -command {vwait _____________________}
button .f.continue -text Weiter -command {_____________________}
```

Aufgabe 8:

Das Malen der Knoten und Kanten ist leicht, wenn die Positionen (Zentren der Kreise) bekannt sind. Für Heaps gilt, dass die Söhne eines Knotens in Position *i* eines Arrays sich in den Positionen 2**i* und 2**i* + 1 befinden. Das gilt allerdings nur für Arrays, die von 1 an nummeriert sind. Daher berechnen wir die Positionen der Söhne über 2**j* - 1 und 2**j* mit *j* = *i* + 1. Beachten Sie, dass **`Xloc`** und **`Yloc`** von Null an laufen.

Ergänzen Sie die Lücken!

```
proc drawTree {a} {
  global Xloc Yloc
  set n [llength $a]
  for {set i 0} {$i < $n} {incr i} {
    set j [expr $i + 1]
    if {____________________ < $n} {
      .c create line $Xloc($i) $Yloc($i) \
        $Xloc([expr {2*$j - 1}]) $Yloc([expr {2*$j - 1}]) \
        -width 2
    }
    if {_____________________ < $n} {
      .c create line $Xloc($i) $Yloc($i) \
        $Xloc([expr {2*$j}]) $Yloc([expr {2*$j}]) \
        -width 2
    }
    .c create oval [expr {$Xloc($i) - 20}] [expr {$Yloc($i) - 20}] \
      [expr {$Xloc($i) + 20}] [expr {$Yloc($i)+20}] -fill blue \
      -tag "_____________________"
    .c create text _____________ _____________ -text [lindex $a ____] \
      -fill white -font "Arial 14 bold" -tag "___________________"
  }
}
```

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Sommersemester 2014

Nachname: Vorname:

Matr.Nr.: Studiengang:

MUSTERLÖSUNG

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

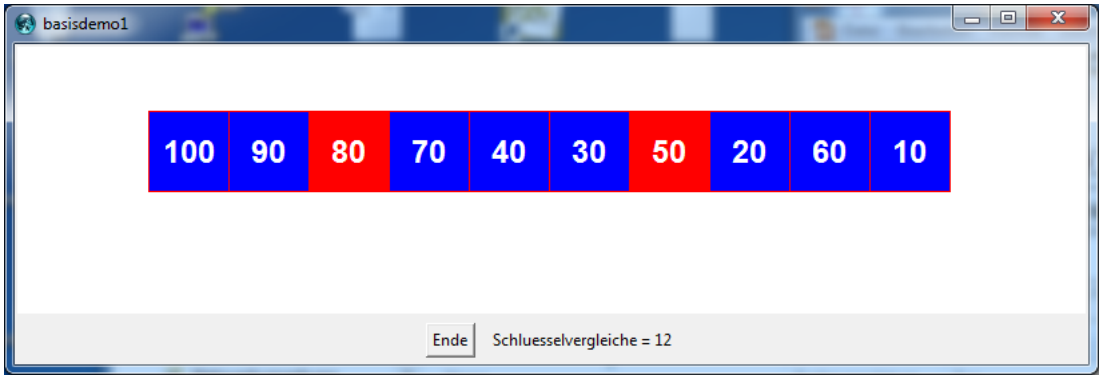
Aufgabe	Punkte max.	Punkte
1	4	
2	5	
3	5	
4	2	
5	4	
6	4	
7	3	
8	5	
Summe	32	

Aufgabe 1:

Visualisierungen von Algorithmen mittels animierter Grafiken sind sehr populär. Auf den Wikipedia-Seiten findet man *animated gifs* für verschiedene Sortierverfahren. Natürlich kann man solche Animationen auch gut mit Tcl/Tk produzieren.

Hier geht es um eine animierte Darstellung des Heapsort. Den Algorithmus müssen Sie nicht im Detail verstehen. Er arbeitet mit Schlüsselvergleichen und Vertauschungen von Arrayeinträgen. Wir stellen das Array dar und immer wenn der Algorithmus eine Vertauschung macht (Aufruf der Prozedur `swap(i, j)`), färben wir die Felder i und j neu.

In der *Base Edition* unserer Implementierung gibt es einen Ende-Knopf zum Beenden des Programms, eine Anzeige für die gemessene Anzahl von Schlüsselvergleichen und eine Leinwand mit dem Array (für das wir in Tcl eine Listenvariable nehmen). Die folgende Abbildung zeigt die Visualisierung dieser Sparversion mit einem Array von 10 Zahlen.



Wir beginnen mit dem Anlegen der Widgets und dem Initialisieren der Daten. Ergänzen Sie die Lücken!

```
set keycounter 0
set xoffset 100; set yoffset 50
canvas .c -width 800 -height 200 -bg white
pack .c
frame .f
pack .f
button .f.cancel -text __Ende__ -command {__exit__}
pack .f.cancel -padx 5 -pady 5 -side left
__label__ .f.anzeige1 -text "__Schluesselvergleiche = $keycounter__"
pack .f.anzeige1 -padx 5 -pady 5 -side left

set input {50 10 30 70 90 100 80 20 60 40}
set N [llength $input]
```

auch 0 oder leer

Aufgabe 2:

Jetzt geht es um die Prozedur zum Anlegen des „Arrays“. Der Aufruf mit den Daten von oben lautet **`createArray $input`**.

Füllen Sie die Lücken und achten Sie darauf, dass die Zahlen zentriert in den 60x60 Pixel großen Rechtecken stehen. Ergänzen Sie die Lücken zu den Tags, damit wir jedes Rechteck und den Text darin individuell ändern können.

```
#create the array as sequence of rectangles, say 60 x 60 with a
#slim border and number inside. Start on left with xoffset and
#yoffset from top.
proc createArray {L} {
  global xoffset yoffset
  set n [llength $L]
  set x $xoffset
  for {set i 0} {$i < $n} {incr i} {
    .c create rectangle $x  $yoffset [expr $x + __60__] \
        [expr $yoffset + __60__] \
        -outline "red" -fill blue -tag "field-$i"
    .c create text [expr $x + __30__] [expr $yoffset + __30__] \
        -fill white -text [lindex __$L__ __$i__] -font "Arial 18 bold" \
        -tag "number-___$i___"
    set x [expr $x + 60]
  }
}
```

Aufgabe 3:

In der Liste **`L`** findet die tatsächliche Sortierung statt. Den jeweiligen Fortschritt zeigen wir mit der Routine **`showArray`**. Gleichzeitig aktualisieren wir den Stand des Schlüsselvergleichszählers. Ergänzen Sie die Lücken.

```
proc showArray {L} {
  global keycounter
  set n [llength $L]                    auch ohne Anführungszeichen korrekt
  for {set i 0} {$i < $n} {incr i} {
    .c __itemconfigure__ __"number-$i"__ -text "__[lindex $L $i]__"
  }
  .f.anzeige1 __configure__ -text "__Schluesselvergleiche = $keycounter__"
  update
}
```

Aufgabe 4:

Wichtig für die Visualisierung ist das Umschalten der Farbe von blau zu rot für die Felder, deren Werte getauscht werden. Zuerst färben wir die beiden Felder auf rot, warten dann eine Sekunde, tauschen die Werte, warten wieder eine Sekunde, färben dann beide Felder zurück auf blau.

Ergänzen Sie die Lücken in den beiden Prozeduren zum Ändern der Farben. Die Aufrufe sehen Sie in der Prozedur **swap** in Aufgabe 5.

```
proc showSwapOn {i j} {
  .c itemconfigure "__field-$i__" -fill "red"
  .c itemconfigure "__field-$j__" -fill "red"
  update
}

proc showSwapOff {i j} {
  .c itemconfigure "__field-$i__" -fill "blue"
  .c itemconfigure "__field-$j__" -fill "blue"
  update
}
```

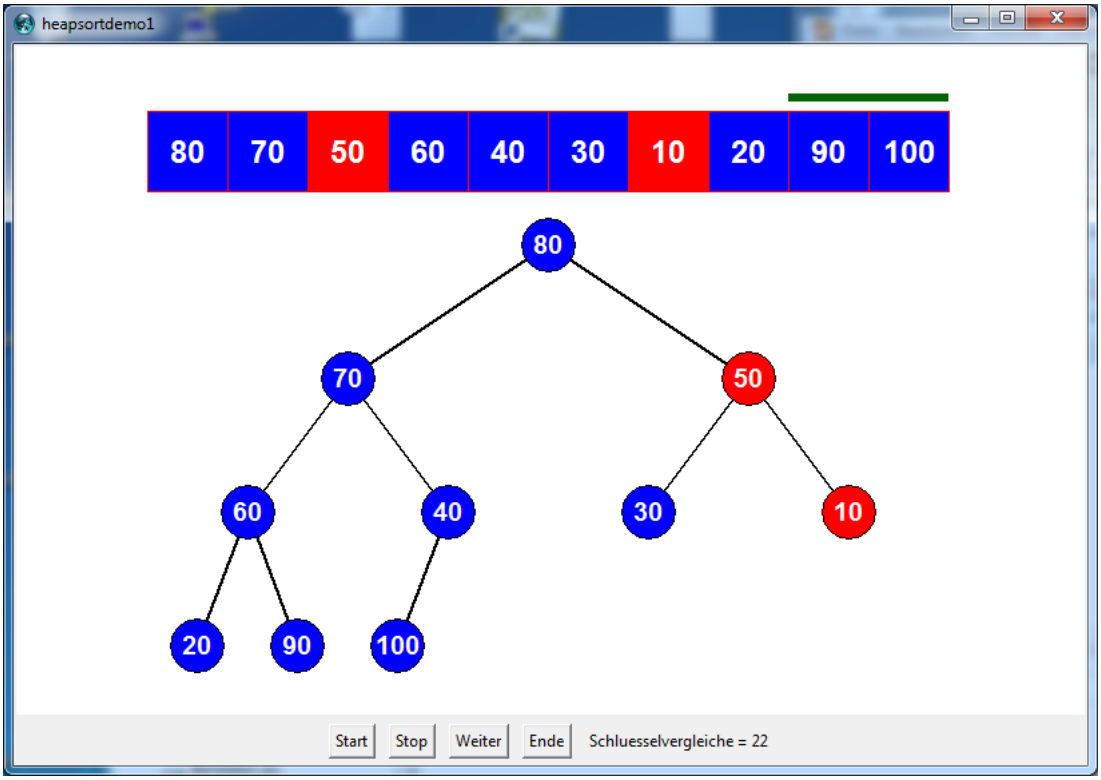
Aufgabe 5:

In der Prozedur **swap** vertauschen wir die beiden Werte an den Positionen **x** und **y** in der Liste **varName** und rufen die Prozeduren zur Farbumschaltung auf. Ergänzen Sie die Lücken! Hinweis: **lset** *varName index value* ersetzt in der Liste *varName* den Wert an der Position *index* durch *value*.

```
proc swap {varName x y} {
  upvar 1 __$varName__ __a__
  showSwapOn __$x__ __$y__
  after 1000
  set tmp [lindex $a $x]
  lset a $x [lindex $a $y]
  lset a $y $tmp
  showArray $a
  after 1000
  showSwapOff __$x__ __$y__
}
```

Aufgabe 6:

In der *Professional Edition* unserer Visualisierung zeigen wir auch den Heap, der ja ein vollständiger Binärbaum ist und ebenenweise gefüllt wird. Die Abbildung unten zeigt die Ausgabe des Programms, ein Balken oberhalb des Felds visualisiert die am rechten Ende bereits sortierte Folge. Über weitere Knöpfe kann die Visualisierung angehalten und danach wieder fortgesetzt werden.



Das Malen des Baums hat zwei Teile. Zum Einen berechnen wir in der Prozedur **`compute_node_locs`** die Positionen der Knoten im Baum (Zentren der Kreise). Diese speichern wir in zwei Arrays **`Xloc`** und **`Yloc`**, die Schlüssel sind die Zahlen von 0 bis N - 1, hier 9.

Der andere Teil ist das Anlegen von Knoten und Kanten mittels der Prozedur **`drawTree`** aus Aufgabe 8, wenn die Positionen der Knoten bekannt sind. Leicht ist dann auch das Ändern der Knotenfarben und angezeigten Werte (Texte): wir verwenden dieselben Tags wie für die Felder in der Liste darüber. Ein einziges **`itemconfigure`** ändert dann an beiden Stellen.

Erläutern Sie das Bildungsgesetz für die Berechnung der X-Y-Koordinaten in **`compute_node_locs`**. Im Beispiel der Abbildung oben ist **`xoffset`** 100, **`yoffset`** 50 und die Leinwand **`.c`** hat Breite 800 und Höhe 500.

```
proc compute_node_locs {n} {
#returns x, y node locations for a heap with n nodes
  global Xloc Yloc xoffset yoffset
  set w [expr [.c cget -width] - 2 * $xoffset]
  set y [expr {$yoffset + 100}]
  set i 0; set j 1
```

```
  while {$i < $n} {
    set b [expr $w/$j]
    for {set k 0} {$k < $j} {incr k} {
      set Xloc($i) [expr $b / 2 + $b*$k + $xoffset]
      set Yloc($i) $y
      incr i
    }
    set j [expr 2 * $j]
    set y [expr $y + 100]
  }
}
```

Erläuterung:

y-Koordinaten: beim Wurzelknoten $yoffset + 100 Pixel
jede weitere Ebene + 100 Pixel

x-Koordinaten: w gibt die Breite des Baumbereichs an,
b gibt den Abstand zwischen zwei Knoten derselben Ebene an (w geteilt durch Anzahl der Knoten der Ebene)
für den ersten Knoten jeder Ebene gilt
x = $xoffset + (b/2);
für jeden weiteren Knoten der Ebene +b Pixel

Aufgabe 7:

Das Anhalten mit dem Knopf **`Stop`**, bzw. Weitermachen mit **`Weiter`** regeln wir über **`vwait`** und eine Variable **`weiter`**, die wir anfangs z.B. auf **`true`** setzen. Ergänzen Sie die Lücken für die *command*-Skripten der beiden Knöpfe!

```
button .f.stop -text Stop -command {vwait __weiter__}
button .f.continue -text Weiter -command {__set weiter true__}
```

Aufgabe 8:

Das Malen der Knoten und Kanten ist leicht, wenn die Positionen (Zentren der Kreise) bekannt sind. Für Heaps gilt, dass die Söhne eines Knotens in Position *i* eines Arrays sich in den Positionen 2**i* und 2**i* + 1 befinden. Das gilt allerdings nur für Arrays, die von 1 an nummeriert sind. Daher berechnen wir die Positionen der Söhne über 2**j* - 1 und 2**j* mit *j* = *i* + 1. Beachten Sie, dass **`Xloc`** und **`Yloc`** von Null an laufen.

Ergänzen Sie die Lücken!

```
proc drawTree {a} {
  global Xloc Yloc
  set n [llength $a]
  for {set i 0} {$i < $n} {incr i} {
    set j [expr $i + 1]
    if {__2 * $j - 1__ < $n} {
      .c create line $Xloc($i) $Yloc($i) \
        $Xloc([expr {2*$j - 1}]) $Yloc([expr {2*$j - 1}]) \
        -width 2
    }
    if {__2 * $j__ < $n} {
      .c create line $Xloc($i) $Yloc($i) \
        $Xloc([expr {2*$j}]) $Yloc([expr {2*$j}]) \
        -width 2
    }
    .c create oval [expr {$Xloc($i) - 20}] [expr {$Yloc($i) - 20}] \
      [expr {$Xloc($i) + 20}] [expr {$Yloc($i)+20}] -fill blue \
      -tag "__field-$i__"
    .c create text __$Xloc($i)__ __$Yloc($i)__ -text [lindex $a __$i__] \
      -fill white -font "Arial 14 bold" -tag "__number-$i__"
  }
}
```

Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Wintersemester 2014/15

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte max.	Punkte
1	5	
2	2 + 3	
3	3 + 1	
4	3	
5	5	
Summe	22	

Für diese Klausur betrachten wir die vereinfachte Eingabe und Anzeige von Klausurergebnissen. Wir unterteilen das Programm in fünf Teilbereiche:

- Festlegen der Punktegrenzen für die „Eins“ und „nicht bestanden“ für die Notenzuordnung,
- Errechnen der Punkteintervalle für die möglichen Notenabstufungen,
- Erfassung und Speicherung der Teilnehmer mit Namen und erzielten Punkten,
- Berechnen der erzielten Note für jeden Teilnehmer,
- Anzeige einer Notenstatistik als Balkendiagramm.

Die folgenden Grundsätze sollen für die Klausurergebnisse gelten.

Es gibt eine maximal erreichbare Punktzahl *pmax*. Die kleinste Unterteilung bei der Benotung sind halbe Punkte. Für die interne Berechnung und Darstellung des Programms arbeitet das Programm mit einer Dezimal**punkt**notation, obwohl im Deutschen die Dezimal**komma**notation vorgeschrieben ist.

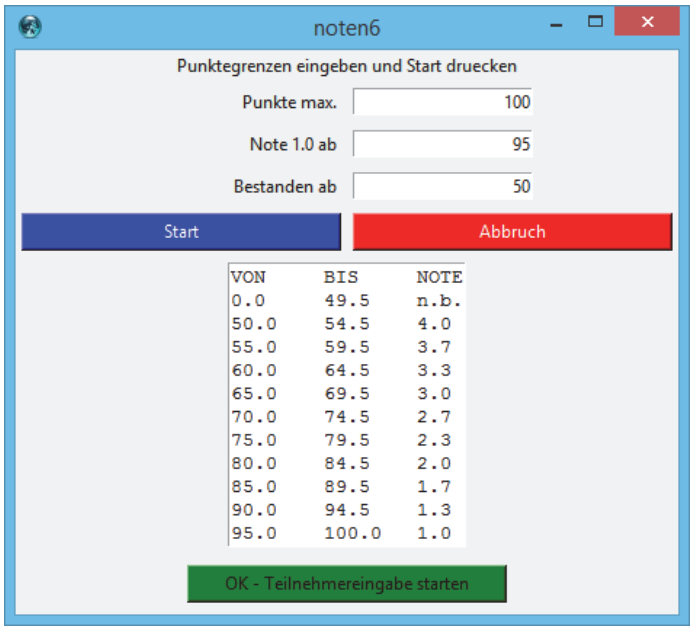
Die untere Grenze zum Bestehen der Klausur sei *gunten* und liege in der Regel bei 50 Prozent von *pmax*, kann aber wegen besonderer Umstände auch anders gesetzt werden. Klausuren mit Punkten < *gunten* sind als nicht bestanden zu werten, abgekürzt **n.b.** oder **5.0**.

Nach oben hin wird eine Grenze *goben* festgesetzt, ab der die höchste Note, nach unserer Prüfungsordnung eine *Eins*, numerisch die **1.0**, vergeben wird.

Zwischen ≥ *gunten* und < *goben* liegen 9 Notenstufen (4.0, 3.7, 3.3, 3.0, ..., 1.7, 1.3), die mit möglichst gleichen Abständen den Punkten zuzuordnen sind. Den Bereich ≥ *goben* haben wir bewusst ausgenommen, teilen also nicht *gunten* bis *pmax* gleichverteilt auf, weil man bei einer längeren, technisch anspruchsvollen Klausur auch kleinere Flüchtigkeitsfehler für eine sehr gute Leistung tolerieren kann.

Aufgabe 1:

Wir beginnen mit der Initialisierung einiger Variablen und mit einer Eingabemaske für Punktegrenzen wie gezeigt. Ergänzen Sie die Lücken!



```
set pmax 100
set goben 95
set gunten 50
set noten "n.b. 4.0 3.7 3.3 3.0 2.7 2.3 2.0 1.7 1.3 1.0"
# moegliche Notenstufen

label .ueberschrift -text "Punktegrenzen eingeben und Start druecken"
grid ____________________ -columnspan 2

label .pmaxlabel -text "Punkte max."
entry .pmaxentry -width 20 -textvariable pmax -justify right
grid .pmaxlabel -column 0 -row 1 -padx 4 -pady 4 -sticky ____
grid _________________ -column 1 -row 1 -padx 4 -pady 4 -sticky w

label .gobenlabel -text "Note 1.0 ab"
entry .gobenentry -width 20 -textvariable goben -justify right
grid .gobenlabel -column 0 -row 2 -padx 4 -pady 4 -sticky ____
grid _________________ -column 1 -row 2 -padx 4 -pady 4 -sticky w

label .guntenlabel -text "Bestanden ab"
entry .guntenentry -width 20 -textvariable gunten -justify right
grid .guntenlabel -column 0 -row 3 -padx 4 -pady 4 -sticky ____
grid _________________ -column 1 -row 3 -padx 4 -pady 4 -sticky w

button .start -width 30 -bg "blue" -fg "white" -text "Start" \
  _______________ {.text delete 1.0 end; berechne; showstat}
button .exit -width 30 -bg "red" -fg "white" -text "Abbruch" \
  _______________ {exit}
grid .start .exit -padx 4 -pady 4

text .text -width 20 -height 12 -wrap none
grid ____________ -columnspan 2 -pady 4

button .ok -width 30 -bg "green" -fg "black" \
  -text "OK - Teilnehmereingabe starten" -command {teilnehmereingabe}
grid ____________ -columnspan 2 -padx 4 -pady 8
```

Aufgabe 2:

(a) In der Prozedur **berechne** ermitteln wir die Punkteintervalle, die zu jeder Note gehören. Da wir nur mit halben Punkten arbeiten wollen, runden wir die Grenzen eines Intervalls immer auf 0.5 Punkte in der Prozedur **myround**. Hinweis: Die Funktion **round** rundet kaufmännisch auf eine ganze Zahl. Ergänzen Sie die Lücken!

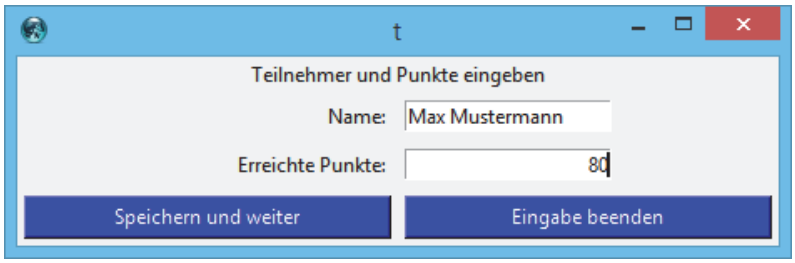
```
proc myround {________} {
  #round to nearest 0.5 value
  return [expr {round(_________ * 2.0) / 2.0}]
}
```

(b) Im eigentlichen Berechnungsteil geht es um die gleichmäßige Aufteilung der verfügbaren Punkte von *goben* - *gunten* auf die bei uns möglichen 9 Noten 4.0 bis 1.3 (n.b. und 1.0 spielen eine Sonderrolle, wie oben bereits erwähnt). Die Notengrenzen werden in den Arrays *pvon* und *pbis* abgelegt und zeilenweise in ein Text-Widget eingetragen (siehe Abbildung in Aufg. 1). Ergänzen Sie die Lücken!

```
proc berechne {} {
  global pvon pbis pmax gunten goben
  set pvon(n.b.) 0.0
  set pbis(n.b.) [expr [myround $gunten] - 0.5]
  set pvon(1.0) [myround $goben]
  set pbis(1.0) [myround $pmax]
  set intervall [expr ($goben - $gunten) / 9.0 ]
  .text insert end "VON\tBIS\tNOTE\n"
  .text insert end "$pvon(n.b.)\t$pbis(n.b.)\tn.b.\n"
  set starter $gunten
  foreach __________ "4.0 3.7 3.3 3.0 2.7 2.3 2.0 1.7 1.3" {
    set pvon($step) [myround $starter]
    set pbis($step) \
      [expr [myround [expr $starter + $intervall]] - 0.5]
    set starter [expr $starter + $intervall]
    ____________________________________________________________
  }
  .text insert end "$pvon(1.0)\t$pbis(1.0)\t1.0\n"
}
```

Aufgabe 3:

Für die Eingabe der Teilnehmer und deren Punkte entwerfen wir wieder eine Maske. Durch Klicken auf „Eingabe beenden“ kann das Fenster wieder geschlossen werden. Beachten Sie, dass dabei nur das Eingabefenster, nicht jedoch das komplette Programm geschlossen werden soll.




```
proc teilnehmereingabe {} {
  global pteilnehmer
  toplevel .t
  label .t.eingeben -text "Teilnehmer und Punkte eingeben"
  grid .t.eingeben -columnspan 2

  label .t.namelabel -text "Name:"
  entry .t.nameentry -width 20 -textvariable ____________ -justify left
  grid .t.namelabel -column 0 -row 1 -padx 4 -pady 4 -sticky e
  grid .t.nameentry -column 1 -row 1 -padx 4 -pady 4 -sticky w

  label .t.punktelabel -text "Erreichte Punkte:"
  entry .t.punkteentry -width 20 -textvariable _____________ -justify right
  grid .t.punktelabel -column 0 -row 2 -padx 4 -pady 4 -sticky e
  grid .t.punkteentry -column 1 -row 2 -padx 4 -pady 4 -sticky w

  button .t.weiter -width 30 -bg "blue" -fg "white" \
    -text "Speichern und weiter" -command {
    set pteilnehmer($tname) $tpunkte
    set tpunkte ""; set tname ""
    showstat }
  button .t.fertig -width 30 -bg "blue" -fg "white" \
    -text "_____________________" -command {_________________}
  grid .t.weiter .t.fertig -padx 4 -pady 4
}
```

(a) Ergänzen Sie die Lücken!

(b) Was passiert, wenn ein Teilnehmer mehrfach, ggf. auch mit unterschiedlichen Punkten, eingetragen wird?

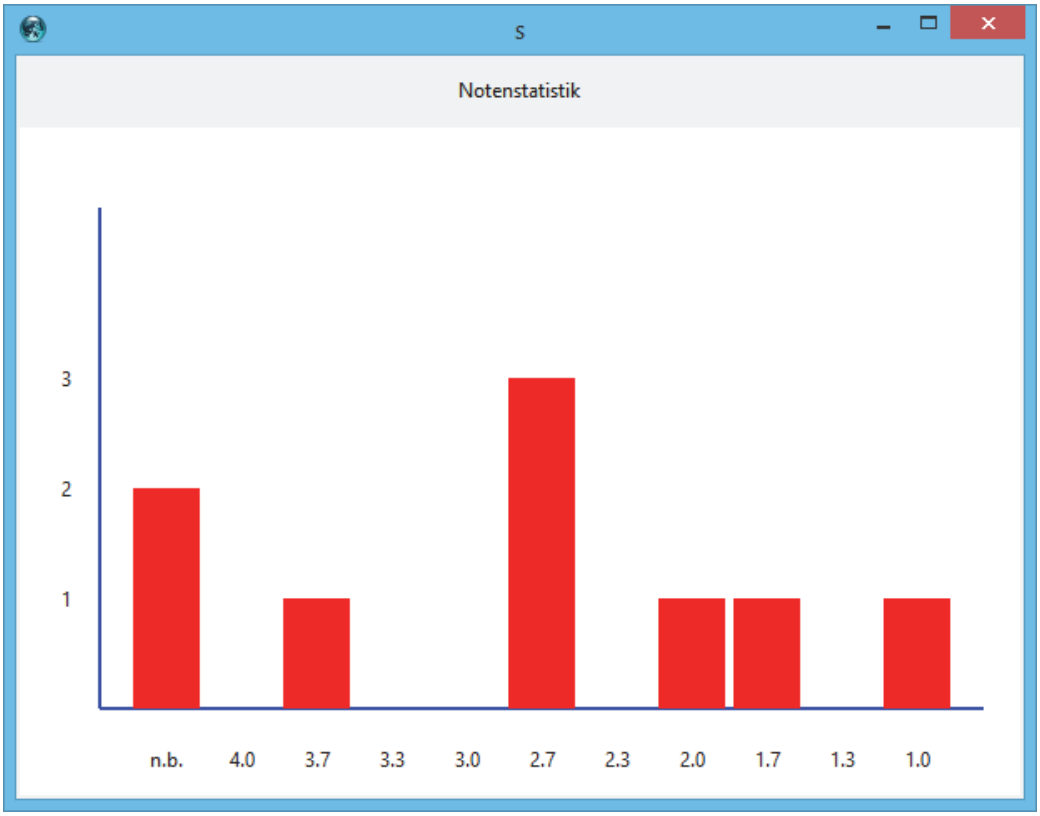
Aufgabe 4:

Stehen die Intervalle für jede Notenstufe erst einmal fest, ist die Umrechnung einer Punktzahl in die zugehörige Note leicht. Ergänzen Sie die Lücken!

```
proc p2n {p} {
  #berechne Note aus gegebener Punktzahl
  global pvon
  set n "n.b."
  foreach step "1.0 1.3 1.7 2.0 2.3 2.7 3.0 3.3 3.7 4.0" {
    if {_____________________________} {
      _____________________________
      break
    }
  }
  return $n
}
```

Aufgabe 5:

Bleibt zuletzt noch die Ausgabe der Notenverteilung in einer Balkengrafik. Der Programmtext unten umfasst das Toplevel-Widget mit der Leinwand als auch die Prozedur zum Zählen der Teilnehmer je Notenstufe und die Anzeige des zugehörigen Balkens.



Ergänzen Sie die Lücken!

```
toplevel .s
label .s.m -text "Notenstatistik"; pack .s.m -pady 10
canvas .s.c -width 600 -height 400 -bg white; pack .s.c

proc showstat {} {
  #je Teilnehmer Punkte in Note umrechnen und Statistik anzeigen
  global noten nteilnehmer pteilnehmer notenstatistik
  .s.c delete all
  .s.c create line 50 350 580 350 -width 2 -fill blue
  .s.c create line 50 350 50 50 -width 2 -fill blue

  foreach note $noten {set notenstatistik($note) 0}
  foreach teiln [array names pteilnehmer] {
    set nteilnehmer($teiln) [p2n $pteilnehmer($teiln)]
    ________ notenstatistik($nteilnehmer($teiln))
  }
  set maxjenote 0
  foreach note $noten {
    if {$notenstatistik($note) > $maxjenote} {
      set _____________ $notenstatistik($note)
    }
  }
  if {$maxjenote > 0} {set yincrement [expr 200 / $maxjenote]}
  for {set i 1} {$i <= $maxjenote} {incr i} {
    .s.c create text 30 [expr 350 - $yincrement * $i] -text _______
  }
  set i 90
  foreach note $noten {
    .s.c create text $i 380 -text __________
    if {$notenstatistik($note) > 0} {
      .s.c create rectangle [expr $i - 20] 350 [expr $i + 20] \
        [expr 350 - $yincrement * _________________________] \
        -fill red -outline ""
    }
    incr i 45
  }
}
```


Ende der Klausur

Klausur zur Vorlesung „Programmierung graphischer Benutzerschnittstellen mit Tcl/Tk“ im Wintersemester 2014/15

MUSTERLÖSUNG

Nachname: Vorname:

Matr.Nr.: Studiengang:

Bearbeiten Sie alle Aufgaben! Hilfsmittel sind nicht zugelassen. Die Bearbeitungszeit ist 90 Minuten.

Aufgabe	Punkte max.	Punkte
1	5	
2	2 + 3	
3	3 + 1	
4	3	
5	5	
Summe	22	

Für diese Klausur betrachten wir die vereinfachte Eingabe und Anzeige von Klausurergebnissen. Wir unterteilen das Programm in fünf Teilbereiche:

- Festlegen der Punktegrenzen für die „Eins“ und „nicht bestanden“ für die Notenzuordnung,
- Errechnen der Punkteintervalle für die möglichen Notenabstufungen,
- Erfassung und Speicherung der Teilnehmer mit Namen und erzielten Punkten,
- Berechnen der erzielten Note für jeden Teilnehmer,
- Anzeige einer Notenstatistik als Balkendiagramm.

Die folgenden Grundsätze sollen für die Klausurergebnisse gelten.

Es gibt eine maximal erreichbare Punktzahl *pmax*. Die kleinste Unterteilung bei der Benotung sind halbe Punkte. Für die interne Berechnung und Darstellung des Programms arbeitet das Programm mit einer Dezimal**punkt**notation, obwohl im Deutschen die Dezimal**komma**notation vorgeschrieben ist.

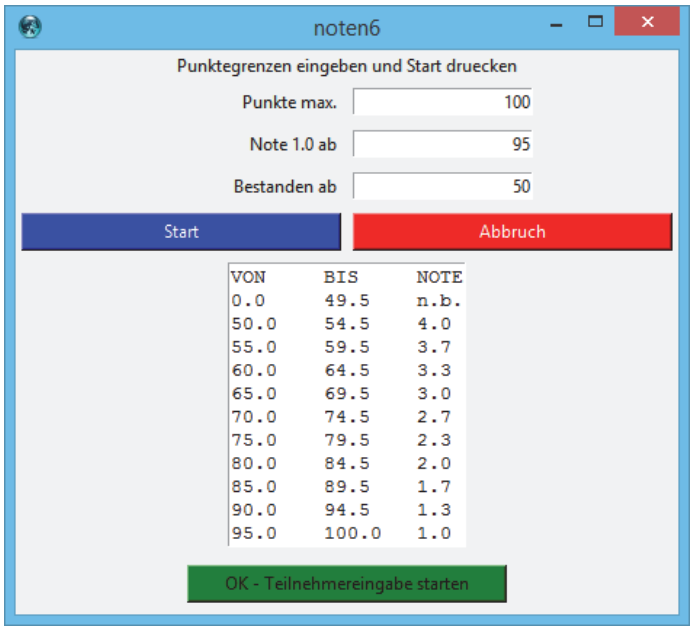
Die untere Grenze zum Bestehen der Klausur sei *gunten* und liege in der Regel bei 50 Prozent von *pmax*, kann aber wegen besonderer Umstände auch anders gesetzt werden. Klausuren mit Punkten < *gunten* sind als nicht bestanden zu werten, abgekürzt **n.b.** oder **5.0**.

Nach oben hin wird eine Grenze *goben* festgesetzt, ab der die höchste Note, nach unserer Prüfungsordnung eine *Eins*, numerisch die **1.0**, vergeben wird.

Zwischen ≥ *gunten* und < *goben* liegen 9 Notenstufen (4.0, 3.7, 3.3, 3.0, ..., 1.7, 1.3), die mit möglichst gleichen Abständen den Punkten zuzuordnen sind. Den Bereich ≥ *goben* haben wir bewusst ausgenommen, teilen also nicht *gunten* bis *pmax* gleichverteilt auf, weil man bei einer längeren, technisch anspruchsvollen Klausur auch kleinere Flüchtigkeitsfehler für eine sehr gute Leistung tolerieren kann.

Aufgabe 1:

Wir beginnen mit der Initialisierung einiger Variablen und mit einer Eingabemaske für Punktegrenzen wie gezeigt. Ergänzen Sie die Lücken!



```
set pmax 100
set goben 95
set gunten 50
set noten "n.b. 4.0 3.7 3.3 3.0 2.7 2.3 2.0 1.7 1.3 1.0"
# moegliche Notenstufen

label .ueberschrift -text "Punktegrenzen eingeben und Start druecken"
grid __.ueberschrift__ -columnspan 2

label .pmaxlabel -text "Punkte max."
entry .pmaxentry -width 20 -textvariable pmax -justify right
grid .pmaxlabel -column 0 -row 1 -padx 4 -pady 4 -sticky __e__
grid __.pmaxentry__ -column 1 -row 1 -padx 4 -pady 4 -sticky w

label .gobenlabel -text "Note 1.0 ab"
entry .gobenentry -width 20 -textvariable goben -justify right
grid .gobenlabel -column 0 -row 2 -padx 4 -pady 4 -sticky __e__
grid __.gobenentry__ -column 1 -row 2 -padx 4 -pady 4 -sticky w

label .guntenlabel -text "Bestanden ab"
entry .guntenentry -width 20 -textvariable gunten -justify right
grid .guntenlabel -column 0 -row 3 -padx 4 -pady 4 -sticky __e__
grid __.guntenentry__ -column 1 -row 3 -padx 4 -pady 4 -sticky w

button .start -width 30 -bg "blue" -fg "white" -text "Start" \
  __-command__ {.text delete 1.0 end; berechne; showstat}
button .exit -width 30 -bg "red" -fg "white" -text "Abbruch" \
  __-command__ {exit}
grid .start .exit -padx 4 -pady 4

text .text -width 20 -height 12 -wrap none
grid __.text__ -columnspan 2 -pady 4

button .ok -width 30 -bg "green" -fg "black" \
  -text "OK - Teilnehmereingabe starten" -command {teilnehmereingabe}
grid __.ok__ -columnspan 2 -padx 4 -pady 8
```

Aufgabe 2:

(a) In der Prozedur **berechne** ermitteln wir die Punkteintervalle, die zu jeder Note gehören. Da wir nur mit halben Punkten arbeiten wollen, runden wir die Grenzen eines Intervalls immer auf 0.5 Punkte in der Prozedur **myround**. Hinweis: Die Funktion **round** rundet kaufmännisch auf eine ganze Zahl. Ergänzen Sie die Lücken!

An Stelle von f kann auch ein beliebiger anderer Parametername gewählt werden.

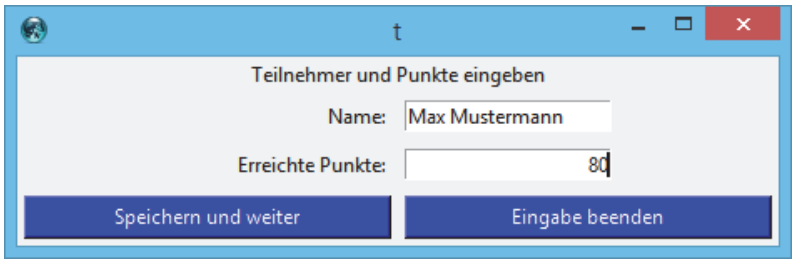
```
proc myround {__f__} {
  #round to nearest 0.5 value
  return [expr {round(__$f__ * 2.0) / 2.0}]
}
```

(b) Im eigentlichen Berechnungsteil geht es um die gleichmäßige Aufteilung der verfügbaren Punkte von *goben - gunten* auf die bei uns möglichen 9 Noten 4.0 bis 1.3 (n.b. und 1.0 spielen eine Sonderrolle, wie oben bereits erwähnt). Die Notengrenzen werden in den Arrays *pvon* und *pbis* abgelegt und zeilenweise in ein Text-Widget eingetragen (siehe Abbildung in Aufg. 1). Ergänzen Sie die Lücken!

```
proc berechne {} {
  global pvon pbis pmax gunten goben
  set pvon(n.b.) 0.0
  set pbis(n.b.) [expr [myround $gunten] - 0.5]
  set pvon(1.0) [myround $goben]
  set pbis(1.0) [myround $pmax]
  set intervall [expr ($goben - $gunten) / 9.0 ]
  .text insert end "VON\tBIS\tNOTE\n"
  .text insert end "$pvon(n.b.)\t$pbis(n.b.)\tn.b.\n"
  set starter $gunten
  foreach __step__ "4.0 3.7 3.3 3.0 2.7 2.3 2.0 1.7 1.3" {
    set pvon($step) [myround $starter]
    set pbis($step) \
      [expr [myround [expr $starter + $intervall]] - 0.5]
    set starter [expr $starter + $intervall]
    __.text insert end "$pvon($step)\t$pbis($step)\t$step\n"__
  }
  .text insert end "$pvon(1.0)\t$pbis(1.0)\t1.0\n"
}
```

Aufgabe 3:

Für die Eingabe der Teilnehmer und deren Punkte entwerfen wir wieder eine Maske. Durch Klicken auf „Eingabe beenden“ kann das Fenster wieder geschlossen werden. Beachten Sie, dass dabei nur das Eingabefenster, nicht jedoch das komplette Programm geschlossen werden soll.




```
proc teilnehmereingabe {} {
  global pteilnehmer
  toplevel .t
  label .t.eingeben -text "Teilnehmer und Punkte eingeben"
  grid .t.eingeben -columnspan 2

  label .t.namelabel -text "Name:"
  entry .t.nameentry -width 20 -textvariable __tname__ -justify left
  grid .t.namelabel -column 0 -row 1 -padx 4 -pady 4 -sticky e
  grid .t.nameentry -column 1 -row 1 -padx 4 -pady 4 -sticky w

  label .t.punktelabel -text "Erreichte Punkte:"
  entry .t.punkteentry -width 20 -textvariable __tpunkte__ -justify right
  grid .t.punktelabel -column 0 -row 2 -padx 4 -pady 4 -sticky e
  grid .t.punkteentry -column 1 -row 2 -padx 4 -pady 4 -sticky w

  button .t.weiter -width 30 -bg "blue" -fg "white" \
    -text "Speichern und weiter" -command {
    set pteilnehmer($tname) $tpunkte
    set tpunkte ""; set tname ""
    showstat }
  button .t.fertig -width 30 -bg "blue" -fg "white" \
    -text "__Eingabe beenden__" -command {__destroy .t__}
  grid .t.weiter .t.fertig -padx 4 -pady 4
}
```

(a) Ergänzen Sie die Lücken!

(b) Was passiert, wenn ein Teilnehmer mehrfach, ggf. auch mit unterschiedlichen Punkten, eingetragen wird?

__Die bisherige Punktezahl des Teilnehmers wird überschrieben.__

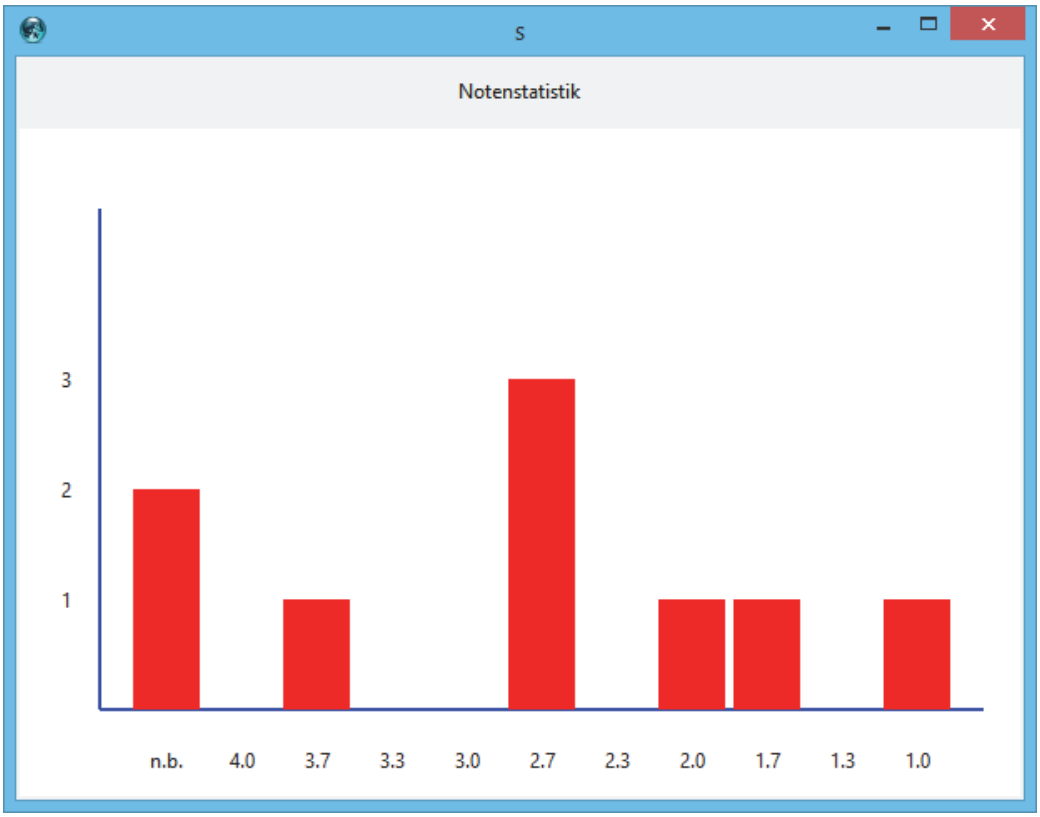
Aufgabe 4:

Stehen die Intervalle für jede Notenstufe erst einmal fest, ist die Umrechnung einer Punktzahl in die zugehörige Note leicht. Ergänzen Sie die Lücken!

```
proc p2n {p} {
  #berechne Note aus gegebener Punktzahl
  global pvon
  set n "n.b."
  foreach step "1.0 1.3 1.7 2.0 2.3 2.7 3.0 3.3 3.7 4.0" {
    if {__$p >= $pvon($step)__} {
      __set n $step__
      break
    }
  }
  return $n
}
```

Aufgabe 5:

Bleibt zuletzt noch die Ausgabe der Notenverteilung in einer Balkengrafik. Der Programmtext unten umfasst das Toplevel-Widget mit der Leinwand als auch die Prozedur zum Zählen der Teilnehmer je Notenstufe und die Anzeige des zugehörigen Balkens.



Ergänzen Sie die Lücken!

```
toplevel .s
label .s.m -text "Notenstatistik"; pack .s.m -pady 10
canvas .s.c -width 600 -height 400 -bg white; pack .s.c

proc showstat {} {
  #je Teilnehmer Punkte in Note umrechnen und Statistik anzeigen
  global noten nteilnehmer pteilnehmer notenstatistik
  .s.c delete all
  .s.c create line 50 350 580 350 -width 2 -fill blue
  .s.c create line 50 350 50 50 -width 2 -fill blue

  foreach note $noten {set notenstatistik($note) 0}
  foreach teiln [array names pteilnehmer] {
    set nteilnehmer($teiln) [p2n $pteilnehmer($teiln)]
    __incr__ notenstatistik($nteilnehmer($teiln))
  }
  set maxjenote 0
  foreach note $noten {
    if {$notenstatistik($note) > $maxjenote} {
      set __maxjenote__ $notenstatistik($note)
    }
  }
  if {$maxjenote > 0} {set yincrement [expr 200 / $maxjenote]}
  for {set i 1} {$i <= $maxjenote} {incr i} {
    .s.c create text 30 [expr 350 - $yincrement * $i] -text __$i__
  }
  set i 90
  foreach note $noten {
    .s.c create text $i 380 -text __$note__
    if {$notenstatistik($note) > 0} {
      .s.c create rectangle [expr $i - 20] 350 [expr $i + 20] \
        [expr 350 - $yincrement * __$notenstatistik($note)__] \
        -fill red -outline ""
    }
    incr i 45
  }
}
```


Ende der Klausur