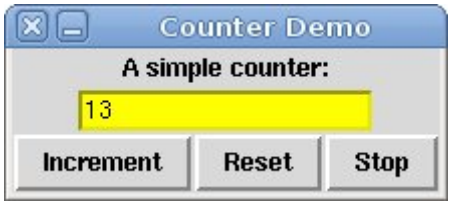
## 4.8 Anwendungen

In diesem Abschnitt wollen wir auf einige Anwendungen eingehen, bei denen die Eigenschaften der logisch-funktionalen Programmierung nützlich sind. Prinzipiell können logisch-funktionale Sprachen überall da eingesetzt werden, wo auch rein logische oder rein funktionale Sprachen sinnvoll sind. Dies haben wir schon am Beispiel der Constraint-Programmierung im Kapitel 4.7.1 gesehen. Bei einigen Anwendungen ist aber die Kombination beider Sprachparadigmen sinnvoll, wie wir nachfolgend sehen werden.

### 4.8.1 GUI-Programmierung

Reale Anwendungen haben häufig graphische Benutzerschnittstellen (GUIs) zur Interaktion. Leider ist die Programmierung dieser Schnittstellen ein notwendiges, aber häufig auch aufwändiges Übel. In diesem Abschnitt wollen wir zeigen, dass die Kombination funktionaler und logischer Konstrukte geeignet ist, um die Programmierung von GUIs einfach und abstrakt zu halten.
Betrachten wir als einfaches Beispiel einen interaktiven Zähler mit folgender GUI:




Hieraus kann man sehen, dass GUIs typischerweise wie folgt zusammengesetzt sind:

- **Widgets**: elementare Interaktionselemente wie
  - Eingabetextfeld
  - Schaltfläche (Button)
  - Textanzeige
  - Menü
  - Auswahl (check button, radio button)
  - ...
- Komposition größerer Einheiten:
  - vertikale Komposition (Spalten)
  - horizontale Komposition (Zeilen)
  - Matrixkomposition

Somit haben GUIs einen hierarchischen Aufbau, der in einer funktionalen Sprache direkt als Datenstruktur beschreibbar ist, z.B. als:

```
data Widget = Entry         [ConfItem]
            | Button        [ConfItem]
            | Label         [ConfItem]
            | TextEdit      [ConfItem]
            | Scale Int Int [ConfItem]
            ⋮
            | Row    [ConfCollection] [Widget]
            | Col    [ConfCollection] [Widget]
            | Matrix [ConfCollection] [[Widget]]
```

Hierbei enthält der Typ `ConfItem` verschiedene Parameter für Widgets, wie z.B. Textinhalt, Hintergrundfarbe etc.:

```
data ConfItem = Text String
              | Background String
              ⋮
```

Der Typ `ConfCollection` dient zur Beschreibung der Ausrichtung bei Kompositionen:

```
data ConfCollection =
   CenterAlign | LeftAlign | RightAlign | TopAlign | BottomAlign
```

Damit könnten wir die Struktur unserer Zähler-GUI wie folgt als Datenterm spezifizieren:

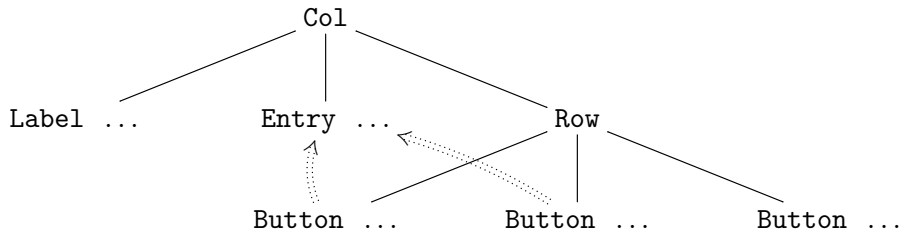
```
Col [] [
  Label [Text "A simple counter:"],
  Entry [Text "0", Background "yellow"],
  Row [] [Button [Text "Increment"],
          Button [Text "Reset"],
          Button [Text "Stop"]]]
```

Es fehlt allerdings noch die eigentliche Funktionalität der GUI. Zum Beispiel soll durch Betätigung der Increment-Schaltfläche der Text in dem `Entry`-Widget (der aktuelle Zählerstand) verändert werden. Hierzu sind zwei Dinge notwendig:

1. Die GUI-Funktionalität soll als „Event Handler“ beschrieben werden: jeder Button (oder ein anderes Widget mit Interaktionsmöglichkeiten) enthält einen „Event Handler“, d.h. eine Funktion, die bei Auslösung eines Ereignisses (durch den Benutzer) aufgerufen wird.

2. GUIs haben neben der Layout-Struktur noch eine innere logische Struktur: beispielsweise müssen die Button-Handler auf den Entry-Widget Bezug nehmen, um den Textinhalt zu ändern.

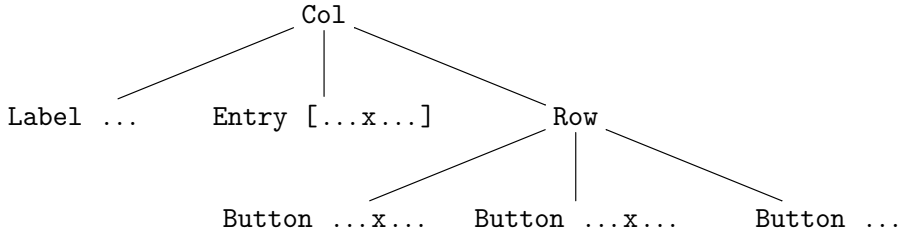
Skizze:




Wie kann man diese zusätzliche logische Struktur in einer Programmiersprache darstellen? Hierzu gibt es folgende Alternativen:

- Skriptsprachen (z.B. Tcl/Tk): Hier werden Widgets durch fest gewählte Strings identifiziert. Die Nachteile hiervon sind die Fehleranfälligkeit (z.B. Tippfehler bei Widgetnamen) und fehlende Kompositionalität (durch Namenskonflikte bei der Zusammensetzung von Widgets).
- Zeigerstrukturen (z.B. GUI-Bibliotheken in Haskell): Hier wird bei der Erzeugung eines Widgets eine Referenz auf dieses zurückgegeben. Diese Referenzen kann man dann zu größeren Einheiten zusammensetzen. Der Nachteil hiervon ist, dass eine GUI dann schrittweise wie in einem imperativen Programm zusammengesetzt wird, statt die Struktur wie oben als hierarchische Struktur zu spezifizieren.
- In logisch-funktionalen Sprachen wie Curry gibt es eine weitere Alternative: die Verwendung logischer Variablen als Referenz.

  Skizze:




```
where x free
```

  Die Vorteile hiervon sind:

  - Die Layout-Struktur kann hierarchisch spezifiziert werden (und muss nicht schrittweise mit Befehlen zusammengesetzt werden).
  - Logische Variablen werden als Referenzen benutzt und sind somit vom Compiler prüfbar (d.h. Vermeidung der Probleme der Skriptsprachen).
  - Das Vorgehen ist kompositionell: es gibt keine Namenskonflikte bei der Komposition von GUIs.

Zur Umsetzung dieser Idee ergänzen wir den Typ der Widget-Konfigurationen:

```
data ConfItem = ...
              ⋮
              | WRef WidgetRef
```

Hierbei ist `WidgetRef` ein abstrakter Datentyp, der keine expliziten Konstruktoren exportiert. Aus diesem Grund kann bei der GUI-Konstruktion nur eine freie Variable als `WRef`-Parameter angegeben werden. Intern wird in der Bibliotheksimplementierung diese freie Variable mit konkreten Strings für die Kommunikation mit Tcl/Tk belegt.

Außerdem wird die Widget-Definition von `Button` so abgeändert, dass dieser als zusätzlichen Parameter einen Event Handler hat, wobei ein Event Handler eine I/O-Operation ist, die eine Verbindung zu einer GUI (Datentyp `GuiPort`) als Parameter hat:

```
data Widget = ...
            | Button (GuiPort → IO ()) [ConfItem]
            ⋮
```

Die Manipulation einzelner Widgets in einer GUI erfolgt in der einfachsten Form durch das Lesen oder Setzen des Widget-Wertes (z.B. ist dies bei einem `Entry`-Widget der Eingabetext). Hierzu stehen die folgenden Basisoperationen zur Verfügung:

```
--- Gets the (String) value of a widget in a GUI.
getValue :: WidgetRef → GuiPort → IO String

--- Sets the (String) value of a widget in a GUI.
setValue :: WidgetRef → String → GuiPort → IO ()

--- Updates the (String) value of a widget w.r.t. to an update function.
updateValue :: (String →String) → WidgetRef → GuiPort → IO ()
updateValue upd wref gp = do
  val <- getValue wref gp
  setValue wref (upd val) gp

--- Terminating the GUI.
exitGUI :: GuiPort → IO ()
```

Damit können wir nun unsere Zähler-GUI vollständig definieren, denn die gerade vorgestellten Elemente sind in der Curry-Bibliothek `Graphics.UI` vordefiniert:[13]

```
import Graphics.UI

counterGUI =
  Col [] [
```

[13] Die Bibliothek `Graphics.UI` befindet sich im Paket `gui`, welches mit Hilfes des Curry Package Managers durch "`cypm add gui`" installiert werden kann.

```
    Label [Text "A simple counter:"],
    Entry [WRef val, Text "0", Background "yellow"],
    Row [] [Button (updateValue incrText val) [Text "Increment"],
            Button (setValue val "0")         [Text "Reset"],
            Button exitGUI                    [Text "Stop"]]]
where
 val free

 incrText s = show ((read s :: Int) + 1)
```

Zur Ausführung einer GUI steht die Operation

```
runGUI :: String → Widget → IO ()
```

zur Verfügung, der man einen Titel und eine Widget-Spezifikation als Parameter übergibt. Somit können wir nun unsere Zähler-GUI starten:

```
> runGUI "Counter Demo" counterGUI
```

Hier hat die Verwendung einer logisch-funktionalen Sprache folgende Vorteile:

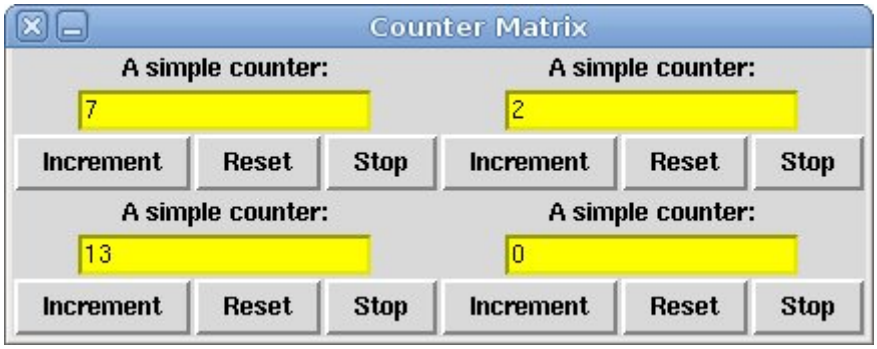
- deklarative Spezifikation von GUIs, die ausführbar ist
- algebraische Datentypen für die Layout-Beschreibung
- Funktionen als Bürger 1. Klasse für Event-Handler in Datenstrukturen
- logische Variablen als prüfbare(!) Referenzen

Somit haben erst die kombinierten Konstrukte der logisch-funktionalen Programmierung die Realisierung dieser GUI-API ermöglicht.

Ein weiterer Vorteil liegt in der Kompositionalität und Wiederverwendbarkeit von GUIs. Wenn wir z.B. mehrere Zähler in einer größeren GUI benötigen, können wir einfach unsere Zähler-GUI wiederverwenden:

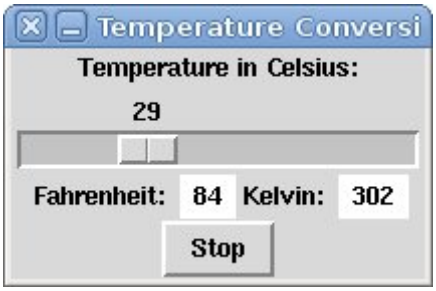
```
> runGUI "Counter Matrix"
         (Matrix [] [[counterGUI,counterGUI],[counterGUI,counterGUI]])
```

Dies führt zu einer GUI mit vier unabhängigen Zählern:




Diese Lösung wäre z.B. in Tcl/Tk problematisch wegen der festen Namen, die für jedes Widget gewählt werden müssen.

Als weiteres Beispiel betrachten wir einen Temperaturkonverter, bei dem man mit einem Schieberegler eine Temperatur in Celsius einstellen kann und diese sofort in eine Temperatur in Fahrenheit und Kelvin umgerechnet wird:

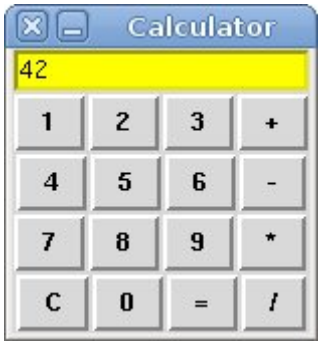



Ein Schieberegler wird durch das Widget `Scale` realisiert, das als Parameter das Minimum und Maximum der einzustellenden Werte erhält. Einem Schieberegler kann man mit dem Konfigurationsparameter `Cmd` auch einen Event-Handler zuordnen, der immer dann aktiviert wird, wenn sich an dem Wert des Schiebereglers etwas ändert. Damit können wir diese GUI wie folgt implementieren:

```
import Graphics.UI

tempGUI =
  Col [] [
    Label [Text "Temperature in Celsius:"],
    Scale 0 100 [WRef cels, Cmd convert],
    Row [] [Label [Text "Fahrenheit: "],
            Message [WRef fahr, Background "white"],
            Label [Text "Kelvin: "],
            Message [WRef kelv, Background "white"]],
    Button exitGUI [Text "Stop"]]

 where
```

```
    cels,fahr,kelv free

    convert gp = do cs <- getValue cels gp
                    let c = read cs :: Int
                    setValue fahr (show (c * 9 `div` 5 + 32)) gp
                    setValue kelv (show (c + 273)) gp

main = runGUI "Temperature Conversion" tempGUI
```

Als letztes Beispiel zur GUI-Programmierung betrachten wir einen einfachen Taschenrechner mit der üblichen Funktionalität:




Das Besondere an diesem Beispiel ist die Anforderung, dass die GUI auch einen internen Zustand halten muss, der von den Event Handlern berücksichtigt wird. Zum Beispiel muss bei Drücken der Taste "=" in der Anzeige das Ergebnis erscheinen, das natürlich von den zuvor gedrückten Tasten abhängig ist.
In Curry (wie auch in Haskell) gibt es „IO-Referenzen", um veränderbare Variablen zu realisieren. Hierzu gibt es (in der Bibliothek `Data.IORef`) die Basisoperationen

```
--- Creates a new IORef with an initial value.
newIORef :: a  → IO (IORef a)

--- Reads the current value of an IORef.
readIORef :: IORef a  → IO a

--- Updates the value of an IORef.
writeIORef :: IORef a  → a  → IO ()
```

Zur Realisierung unserer GUI müssen wir nur noch überlegen, was die GUI in ihrem Zustand halten muss. Da wir eine Zahl eingeben und diese dann über eine Operation verknüpfen, sind zwei Dinge für den Zustand relevant:

1. Die gerade eingegebene Zahl.
2. Eine Funktion, mit der die aktuelle Zahl verknüpft werden muss, wenn man z.B. "=" drückt.

Zum Beispiel ist nach der Eingabe von "`3+4`" der aktuelle Zustand `(4,(3+))`, d.h. die aktuelle Zahl ist `4` und diese muss mit der Operation `(3+)` verknüpft werden.
Mit diesen Überlegungen wird die Implementierung der GUI recht einfach: beim Drücken einer Zifferntaste muss die aktuelle Zahl vergrößert werden, und beim Drücken einer anderen Taste muss die Verknüpfungsoperation verändert werden. In jedem Fall muss auch die Anzeige modifiziert werden. Die vollständige Implementierung der Taschenrechner-GUI ist in Abbildung 4.1 angegeben.

In ähnlicher Form kann man auch eine API mit hohem Abstraktionsniveau für die Programmierung dynamischer Web-Seiten realisieren (vgl. Curry-Pakete `html` und `html2` und [Hanus 01, Hanus 21]).