# 13

# *Applications*

This chapter presents two applications which seek to illustrate the use of the many different programming concepts presented previously in Part III.

The first application builds a library of graphic components, `Awi` (Application Window Interface). Next the library will be applied in a simple Francs to Euros converter. The components library reacts to user input by calling event handlers. Although this is a simple application algorithmically, it shows the benefits of using closures to structure the communication between components. Indeed the various event handlers share certain values via their environment. To appreciate the construction of `Awi` it is necessary to know the base library `Graphics` (see chapter 5, page 117).

The second application is a search for a least cost path in a directed graph. It uses Dijkstra's algorithm which calculates all the least cost paths from a source node to all the other nodes connected to this source. A cache mechanism implemented using a table of weak pointers (see page 265) is used to speed the search. The GC can free the elements of this table at any time but they can be recalculated as necessary. The graph visualization uses the simple button component of the `Awi` library for selecting the origin and destination nodes of the path sought. We then compare the efficiency of running the algorithm both with and without the cache. To facilitate timing measurements between the two versions a file with the description of the graph and the origin and destination nodes is passed as an argument to the search algorithm. Finally, a small graphical interface will be added to the search program.

## *Constructing a Graphical Interface*

The implementation of a graphical interface for a program is a tedious job if the tools at your disposal are not powerful enough, as this is the case with the `Graphics` library. The user-friendliness of a program derives in part from its interface. To ease the task of creating a graphical interface we will start by creating a new library called `Awi` which

sits on top of `Graphics` and then we will use it as a simple module to help us construct the interface for an application.

This graphical interface manipulates *components*. A component is a region of the main window which can be displayed in a certain graphical context and can handle events that are sent to it. There are basically two kinds of components: simple components, such as a confirmation button or a text entry field, and *containers* which allow other components to be placed within them. A component can only be attached to a single container. Thus the interface of an application is built as a tree whose root corresponds to the main container (the graphics window), the nodes are also containers and the leaves are simple components or empty containers. This treelike structure helps us to propagate events arising from user interaction. If a container receives an event it checks whether one of its children can handle it, if so then it sends the event to that child, otherwise it deals with the event using its own handler.

The component is the essential element in this library. We define it as a record which contains details of size, a graphic context, the parent and child components along with functions for display and for handling events. Containers include a function for displaying their components. To define the *`component`* type, we build the types for the graphics context, for events and for initialization options. A graphical context is used to contain the details of "graphical styles" such as the colors of the background and foreground, the size of the characters, the current location of the component and the fonts that have been chosen. Then must we define the kinds of events which can be sent to the component. These are more varied than those in the `Graphics` library on which they are based. We include a simple option mechanism which helps us to configure graphics contexts or components. One implementation difficulty arises in positioning components within a container.

The general event handling loop receives physical events from the input function of the `Graphics` library, decides whether other events should be generated as a result of these physical events, and then sends them to the root container. We shall consider the following components: text display, buttons, list boxes, input regions and enriched components. Next we will show how the components are assembled to construct graphical interfaces, illustrating this with a program to convert between Francs and Euros. The various components of this application communicate with each other over a shared piece of state.

## *Graphics Context, Events and Options*

Let's start by defining the base types along with the functions to initialize and modify graphics contexts, events and options. There is also an option type to help us parametrize the functions which create graphical objects.

### *Graphics Context*

The graphics context allows us to keep track of the foreground and background colors, the font, its size, the current cursor position, and line width. This results in the following type.

```
type  g_context = {
      mutable bcol : Graphics.color;
      mutable fcol : Graphics.color;
      mutable font :  string;
      mutable font_size : int;
      mutable lw : int;
      mutable x : int;
      mutable y : int };;
```

The `make_default_context` function creates a new graphics context containing default values [1].

```
# let default_font = "fixed"
  let default_font_size = 12
  let make_default_context () =
    { bcol = Graphics.white; fcol = Graphics.black;
      font = default_font;
      font_size = default_font_size;
      lw = 1;
      x = 0; y = 0;};;
val default_font : string = "fixed"
val default_font_size : int = 12
val make_default_context : unit -> g_context = <fun>
```

Access functions for the individual fields allow us to retrieve their values without knowing the implementation of the type itself.

```
# let get_gc_bcol gc  = gc.bcol
  let get_gc_fcol gc  = gc.fcol
  let get_gc_font  gc  = gc.font
  let get_gc_font_size  gc  = gc.font_size
  let get_gc_lw gc  = gc.lw
  let get_gc_cur gc = (gc.x,gc.y);;
val get_gc_bcol : g_context -> Graphics.color = <fun>
val get_gc_fcol : g_context -> Graphics.color = <fun>
val get_gc_font : g_context -> string = <fun>
```

1. The name of the character font may vary according to the system being used.

```
val get_gc_font_size : g_context -> int = <fun>
val get_gc_lw : g_context -> int = <fun>
val get_gc_cur : g_context -> int * int = <fun>
```

The functions to modify those fields work on the same principle.

```
# let set_gc_bcol gc c = gc.bcol <- c
  let set_gc_fcol gc c = gc.fcol <- c
  let set_gc_font  gc f = gc.font <- f
  let set_gc_font_size  gc s = gc.font_size <- s
  let set_gc_lw gc i = gc.lw <- i
  let set_gc_cur gc (a,b) = gc.x<- a; gc.y<-b;;
val set_gc_bcol : g_context -> Graphics.color -> unit = <fun>
val set_gc_fcol : g_context -> Graphics.color -> unit = <fun>
val set_gc_font : g_context -> string -> unit = <fun>
val set_gc_font_size : g_context -> int -> unit = <fun>
val set_gc_lw : g_context -> int -> unit = <fun>
val set_gc_cur : g_context -> int * int -> unit = <fun>
```

We can thus create new contexts, and read and write various fields of a value of the *g_context* type.

The use_gc function applies the data of a graphic context to the graphical window.

```
# let use_gc gc =
      Graphics.set_color (get_gc_fcol gc);
      Graphics.set_font (get_gc_font gc);
      Graphics.set_text_size (get_gc_font_size gc);
      Graphics.set_line_width (get_gc_lw gc);
      let (a,b) = get_gc_cur gc in Graphics.moveto a b;;
val use_gc : g_context -> unit = <fun>
```

Some data, such as the background color, are not directly used by the Graphics library and do not appear in the use_gc function.

## *Events*

The Graphics library only contains a limited number of interaction events: mouse click, mouse movement and key press. We want to enrich the kind of event that arises from interaction by integrating events arising at the component level. To this end we define the type *rich_event*:

```
# type rich_event =
```

```
    MouseDown | MouseUp | MouseDrag | MouseMove
  | MouseEnter | MouseExit | Exposure
  | GotFocus | LostFocus | KeyPress | KeyRelease;;
```

To create such events it is necessary to keep a history of previous events. The `MouseDown` and `MouseMove` events correspond to mouse events (clicking and moving) which are created by `Graphics`. Other mouse events are created by virtue of either the previous event `MouseUp`, or the last component which handled a physical event `MouseExit`. The `Exposure` event corresponds to a request to redisplay a component. The concept of *focus* expresses that a given component is interested in a certain kind of event. Typically the input of text to a component which has grabbed the focus means that this component alone will handle `KeyPress` and `KeyRelease` events. A `MouseDown` event on a text input component hands over the input focus to it and takes it away from the component which had it before.

These new events are created by the event handling loop described on page 360.

## *Options*

A graphical interface needs rules for describing the creation options for graphical objects (components, graphics contexts). If we wish to create a graphics context with a certain color it is currently necessary to construct it with the default values and then to call the two functions to modify the color fields in that context. In the case of more complex graphic objects this soon becomes tedious. Since we want to extend these options as we build up the components of the library, we need an "extensible" sum type. The only one provided by Objective Caml is the *exn* type used for exceptions. Because using *exn* for handling options would affect the clarity of our programs we will only use this type for real exceptions. Instead, we will simulate an extensible sum type using pseudo constructors represented by character strings. We define the type *opt_val* for the values of these options. An option is a tuple whose first element is the name of the option and the second its value. The *lopt* type encompasses a list of such options.

```
# type opt_val = Copt of Graphics.color | Sopt of string
                 | Iopt of int | Bopt of bool;;
# type lopt = (string * opt_val) list ;;
```

The decoding functions take as input a list of options, an option name and a default value. If the name belongs to the list then the associated value is returned, if not then we get the default value. We show here only the decoding functions for integers and booleans, the others work on the same principle.

```
# exception OptErr;;
exception OptErr
# let theInt lo name default =
    try
```

```
        match List.assoc name lo with
          Iopt i → i
        | _  → raise OptErr
      with Not_found → default;;
val theInt : ('a * opt_val) list -> 'a -> int -> int = <fun>
# let theBool lo name default =
      try
        match List.assoc name lo with
          Bopt b → b
        | _  → raise OptErr
      with Not_found → default;;
val theBool : ('a * opt_val) list -> 'a -> bool -> bool = <fun>
```

We can now write a function to create a graphics context using a list of options in the following manner:

```
# let set_gc gc lopt =
      set_gc_bcol gc (theColor lopt "Background" (get_gc_bcol gc));
      set_gc_fcol gc (theColor lopt "Foreground" (get_gc_fcol gc));
      set_gc_font gc (theString lopt "Font" (get_gc_font gc));
      set_gc_font_size gc (theInt lopt "FontSize" (get_gc_font_size gc));
      set_gc_lw gc (theInt lopt "LineWidth" (get_gc_lw gc));;
val set_gc : g_context -> (string * opt_val) list -> unit = <fun>
```

This allows us to ignore the order in which the options are passed in.

```
# let dc = make_default_context () in
      set_gc dc [ "Foreground", Copt Graphics.blue;
                  "Background", Copt Graphics.yellow];
      dc;;
- : g_context =
{bcol=16776960; fcol=255; font="fixed"; font_size=12; lw=1; x=0; y=0}
```

This results in a fairly flexible system which unfortunately partially evades the type system. The name of an option is of the type *string* and nothing prevents the construction of a nonexistant name. The result is simply that the value is ignored.

## *Components and Containers*

The component is the essential building block of this library. We want to be able to create components and then easily assemble them to construct interfaces. They must be able to display themselves, to recognize an event destined for them, and to handle

that event. Containers must be able to receive events from other components or to hand them on. We assume that a component can only be added to one container.

## *Construction of Components*

A value of type *component* has a size (`w` and `h`), an absolute position in the main window (`x` and `y`), a graphics context used when it is displayed (`gc`), a flag to show whether it is a container (`container`), a parent - if it is itself attached to a container (`parent`), a list of child components (`children`) and four functions to handle positioning of components. These control how children are positioned within a component (`layout`), how the component is displayed (`display`), whether any given point is considered to be within the area of the component (`mem`) and finally a function for event handling (`listener`) which returns `true` if the event was handled and `false` otherwise. The parameter of the `listener` is of type (type *rich_status*) and contains the name of the event the lowlevel event information coming from the `Graphics` module, information on the keyboard focus and the general focus, as well as the last component to have handled an event. So we arrive at the following mutually recursive declarations:

```
# type  component =
    { mutable info : string;
      mutable x : int;  mutable y : int;
      mutable w :int ; mutable h : int;
      mutable gc : g_context;
      mutable container : bool;
      mutable parent :   component list;
      mutable children :  component list;
      mutable layout_options : lopt;
      mutable layout : component →  lopt → unit;
      mutable display : unit  → unit;
      mutable mem : int * int → bool;
      mutable listener : rich_status  → bool }
  and rich_status =
    { re : rich_event;
      stat : Graphics.status;
      mutable key_focus : component;
      mutable gen_focus : component;
      mutable last : component};;
```

We access the data fields of a component with the following functions.

```
# let get_gc c = c.gc;;
val get_gc : component -> g_context = <fun>
# let is_container c = c.container;;
val is_container : component -> bool = <fun>
```

The following three functions define the default behavior of a component. The function to test whether a given mouse position applies to a given component (`in_rect`) checks that the coordinate is within the rectangle defined by the coordinates of the component.

The default display function (`display_rect`) fills the rectangle of the component with the background color found in the graphic context of that component. The default layout function (`direct_layout`) places components relatively within their containers. Valid options are `"PosX"` and `"PosY"`, corresponding to the coordinates relative to the container.

```
# let in_rect  c  (xp,yp)   =
    (xp >= c.x) && (xp < c.x + c.w) && (yp >= c.y) && (yp < c.y + c.h) ;;
val in_rect : component -> int * int -> bool = <fun>
# let display_rect c ()   =
    let gc = get_gc c in
      Graphics.set_color (get_gc_bcol gc);
      Graphics.fill_rect c.x c.y c.w c.h ;;
val display_rect : component -> unit -> unit = <fun>
# let direct_layout c c1 lopt =
    let px = theInt lopt "PosX" 0
    and py = theInt lopt "PosY" 0 in
      c1.x <- c.x + px; c1.y <- c.y + py ;;
val direct_layout :
  component -> component -> (string * opt_val) list -> unit = <fun>
```

It is now possible to define a component using the function `create_component` which takes width and height as parameters and uses the three preceding functions.

```
# let create_component iw ih  =
    let dc =
       {info="Anonymous";
        x=0; y=0; w=iw; h=ih;
        gc = make_default_context() ;
        container = false;
        parent = []; children = [];
        layout_options = [];
        layout = (fun a b → ());
        display = (fun () → ());
        mem = (fun s → false);
        listener = (fun s  → false);}
    in
      dc.layout <- direct_layout dc;
      dc.mem <- in_rect dc;
      dc.display <- display_rect dc;
      dc ;;
val create_component : int -> int -> component = <fun>
```

We then define the following empty component:

```
# let empty_component = create_component 0 0 ;;
```

This is used as a default value when we construct values which need to contain at least one component (for example a value of type *rich_status*).

## *Adding Child Components*

The difficult part of adding a component to a container is how to position the component within the container. The `layout` field contains this positioning function. It takes a component (a child) and a list of options and calculates the new coordinates of the child within the container. Different options can be used according to the positioning function. We describe several layout functions when we talk about about the *panel* component (see *below*, page 366). Here we simply describe the mechanism for propagating the display function through the tree of components, coordinate changes, and propagating events. The propagation of actions makes intensive use of the `List.iter` function, which applies a function to all the elements of a list.

The function `change_coord` applies a relative change to the coordinates of a component and those of all its children.

```
# let rec change_coord  c (dx,dy)   =
    c.x <- c.x + dx; c.y <- c.y + dy;
    List.iter (fun s → change_coord s (dx,dy) ) c.children;;
val change_coord : component -> int * int -> unit = <fun>
```

The `add_component` function checks that the conditions for adding a component have been met and then joins the parent (`c`) and the child (`c1`). The list of positioning options is retained in the child component, which allows us to reuse them when the positioning function of the parent changes. The list of options passed to this function are those used by the positioning function. There are three conditions which need to be prohibited: the child component is already a parent, the parent is not a container or the child is too large for parent

```
# let add_component  c c1 lopt   =
    if c1.parent <> [] then failwith "add_component: already a parent"
    else
      if not (is_container c ) then
        failwith "add_component: not a container"
      else
        if (c1.x + c1.w > c.w) || (c1.y + c1.h > c.h)
        then failwith "add_component: bad position"
        else
          c.layout  c1 lopt;
          c1.layout_options <- lopt;
          List.iter (fun s → change_coord s (c1.x,c1.y)) c1.children;
          c.children <- c1 :: c.children;
          c1.parent <-  [c] ;;
val add_component : component -> component -> lopt -> unit = <fun>
```

The removal of a component from some level in the tree, implemented by the following function, entails both a change to the link between the parent and the child and also a change to the coordinates of the child and all its own children:

```
# let remove_component c c1 =
      c.children <- List.filter ((!=) c1)  c.children;
      c1.parent <- List.filter ((!=) c) c1.parent;
      List.iter (fun s → change_coord s (- c1.x, - c1.y)) c1.children;
      c1.x <- 0; c1.y <- 0;;
val remove_component : component -> component -> unit = <fun>
```

A change to the positioning function of a container depends on whether it has any children. If it does not the change is immediate. Otherwise we must first remove the children of the container, modify the container's positioning function and then add the components back in with the same options used when they were originally added.

```
# let set_layout f c =
    if c.children = [] then c.layout <- f
    else
      let ls = c.children in
        List.iter (remove_component c) ls;
        c.layout <- f;
        List.iter (fun s → add_component c s s.layout_options) ls;;
val set_layout : (component -> lopt -> unit) -> component -> unit = <fun>
```

This is why we kept the list of positioning options. If the list of options is not recognized by the new function it uses the defaults.

When a component is displayed, the display event must be propagated to its children. The container is displayed behind its children. The order of display of the children is unimportant because they never overlap.

```
# let rec display  c  =
    c.display ();
    List.iter (fun cx → display cx ) c.children;;
val display : component -> unit = <fun>
```

## *Event Handling*

The handling of physical events (mouse click, key press, mouse movement) uses the `Graphics.wait_next_event` function (see page 132) which returns a physical status (of type `Graphics.status`) following any user interaction. This physical status is used to calculate a rich status (of type *rich_status*) containing the event type (of type *rich_event*), the physical status, the components possessing the keyboard focus and the general focus along with the last component which successfully handled such an event. The general focus is a component which accepts all events.

Next we describe the functions for the manipulating of rich events, the propagation of this status information to components for them to be handled, the creation of the information and the main event-handling loop.

### *Functions used on Status*

The following functions read the values of the mouse position and the focus. Functions on focus need a further parameter: the component which is capturing or losing that focus.

```
# let get_event e = e.re;;
# let get_mouse_x e = e.stat.Graphics.mouse_x;;
# let get_mouse_y e = e.stat.Graphics.mouse_y;;
# let get_key e = e.stat.Graphics.key;;

# let has_key_focus e c = e.key_focus == c;;
# let take_key_focus e c = e.key_focus <- c;;
# let lose_key_focus e c = e.key_focus <- empty_component;;
# let has_gen_focus e c = e.gen_focus == c;;
# let take_gen_focus e c = e.gen_focus <- c;;
# let lose_gen_focus e c = e.gen_focus <- empty_component;;
```

### *Propagation of Events*

A rich event is sent to a component to be handled. Analogous to the display mechanism discussed earlier, child components have priority over their parents for handling simple mouse movement. If a component receives status information associated with an event, it looks to see if it has a child which can handle it. If so, the child returns `true` otherwise `false`. If no child can handle the event, the parent component tries to use the function in its own `listener` field.

Status information coming from keyboard activity is propagated differently. The parent component looks to see if it possesses the keyboard focus, and if so it handles the event, otherwise it propagates to its children.

Some events are produced as a result of handling an initial event. For example, if one component captures the focus, then this means another has lost it. Such events are handled immediately by the target component. This is the same with the entry and exit events caused when the mouse is moved between different components.

The `send_event` function takes a value of type *`rich_status`* and a component. It returns a boolean indicating whether the event was handled or not.

```
# let rec send_event rs c =
    match get_event rs with
      MouseDown | MouseUp | MouseDrag | MouseMove →
       if c.mem(get_mouse_x rs, get_mouse_y rs) then
         if  List.exists (fun sun → send_event rs sun) c.children then true
         else ( if c.listener rs then (rs.last <-c; true) else false )
       else false
    | KeyPress | KeyRelease →
       if has_key_focus rs c then
```

```
          ( if c.listener rs then (rs.last<-c; true)
            else false )
        else List.exists (fun sun → send_event rs sun) c.children
    | _  → c.listener rs;;
val send_event : rich_status -> component -> bool = <fun>
```

Note that the hierarchical structure of the components is really a tree and not a cyclic graph. This guarantees that the recursion in the `send_event` function cannot cause an infinite loop.

## *Event Creation*

We differentiate between two kinds of events: those produced by a physical action (such as a mouse click) and those which arise from some action linked with the previous history of the system (such as the movement of the mouse cursor out of the screen area occupied by a component). As a result we define two functions for creating rich events.

The function which deals with the former kind constructs a rich event out of two sets of physical status information:

```
# let compute_rich_event s0 s1  =
    if s0.Graphics.button <> s1.Graphics.button then
    begin
      if s0.Graphics.button then MouseDown else MouseUp
    end
    else if s1.Graphics.keypressed then KeyPress
    else if (s0.Graphics.mouse_x <> s1.Graphics.mouse_x  ) ||
            (s0.Graphics.mouse_y <> s1.Graphics.mouse_y  ) then
    begin
      if s1.Graphics.button then MouseDrag else MouseMove
    end
    else raise Not_found;;
val compute_rich_event : Graphics.status -> Graphics.status -> rich_event =
  <fun>
```

The function creating the latter kind of event uses the last two rich events:

```
# let send_new_events res0 res1 =
      if res0.key_focus <> res1.key_focus then
      begin
        ignore(send_event  {res1 with re = LostFocus} res0.key_focus);
        ignore(send_event  {res1 with re = GotFocus} res1.key_focus)
      end;
      if (res0.last <> res1.last) &&
         (( res1.re = MouseMove) || (res1.re = MouseDrag)) then
      begin
        ignore(send_event  {res1 with re = MouseExit} res0.last);
        ignore(send_event  {res1 with re = MouseEnter} res1.last )
```

```
      end;;
val send_new_events : rich_status -> rich_status -> unit = <fun>
```

We define an initial value for the *rich_event* type. This is used to initialize the history of the event loop.

```
# let initial_re =
    { re = Exposure;
      stat = { Graphics.mouse_x=0; Graphics.mouse_y=0;
               Graphics.key = ' ';
               Graphics.button = false;
               Graphics.keypressed = false };
      key_focus = empty_component;
      gen_focus = empty_component;
      last = empty_component } ;;
```

## *Event Loop*

The event loop manages the sequence of interactions with a component, usually the ancestor component for all the components of the interface. It is supplied with two booleans indicating whether the interface should be redisplayed after every physical event has been handled (`b_disp`) and whether to handle mouse movement (`b_motion`). The final argument (`c`), is the root of the component tree.

```
# let loop b_disp b_motion  c =
    let res0 = ref initial_re  in
    try
      display c;
      while true do
        let lev = [Graphics.Button_down; Graphics.Button_up;
                   Graphics.Key_pressed] in
        let flev = if b_motion then (Graphics.Mouse_motion) :: lev
                   else lev in
        let s = Graphics.wait_next_event flev
        in
          let res1 = {!res0 with stat = s} in
           try
              let res2 = {res1 with
                          re = compute_rich_event !res0.stat res1.stat} in
                ignore(send_event res2 c);
                send_new_events !res0 res2;
                res0 := res2;
               if b_disp then display c
           with Not_found → ()
      done
    with e → raise e;;
val loop : bool -> bool -> component -> unit = <fun>
```

The only way out of this loop is when one of the handling routines raises an exception.

### *Test Functions*

We define the following two functions to create by hand status information corresponding to mouse and keyboard events.

```
# let make_click e x y =
    {re = e;
     stat = {Graphics.mouse_x=x; Graphics.mouse_y=y;
             Graphics.key = ' '; Graphics.button = false;
             Graphics.keypressed = false};
     key_focus = empty_component;
     gen_focus = empty_component;
     last = empty_component}

  let make_key e ch c =
    {re = e;
     stat = {Graphics.mouse_x=0; Graphics.mouse_y=0;
             Graphics.key = c; Graphics.button = false;
             Graphics.keypressed = true};
     key_focus = empty_component;
     gen_focus = empty_component;
     last = empty_component};;
val make_click : rich_event -> int -> int -> rich_status = <fun>
val make_key : rich_event -> 'a -> char -> rich_status = <fun>
```

We can now simulate the sending of a mouse event to a component for test purposes.

## *Defining Components*

The various mechanisms for display, coordinate change and, propagating event are now in place. It remains for us to define some components which are both useful and easy to use. We can classify components into the following three categories:

- simple components which do not handle events, such as text to be displayed;
- simple components which handle events, such as text entry fields;
- containers and their various layout strategies.

Values are passed between components, or between a component and the application by modification of shared data. The sharing is implemented by closures which contain in their environment the data to be modified. Moreover, as the behavior of the component can change as a result of event handling, components also contain an internal state in the closures of their handling functions. For example the handling function for an input field has access to text while it is being written. To this end we implement components in the following manner:

- define a type to represent the internal state of the component;
- declare functions for the manipulation of this state;
- implement the functions for display, testing whether a coordinate is within the component and handling events;
- implement the function to create the component, thereby associating those closures with fields in the component;
- test the component by simulating the arrival of events.

Creation functions take a list of options to configure the graphics context. The calculation of the size of a component when it is created needs to make use of graphics context of the graphical window in order to determine the width of the text to be displayed.

We describe the implementation of the following components:

- simple text (`label`);
- simple container (`panel`);
- simple button (`button`);
- choice among a sequence of strings (`choice`);
- text entry field (`textfield`);
- rich component (`border`).

## *The Label Component*

The simplest component, called a *label*, displays a string of characters on the screen. It does not handle events. We will start by describing the display function and then the creation function.

Display must take account of the foreground and background colors and the character font. It is the job of the `display_init` function to erase the graphical region of the component, select the foreground color and position the cursor. The function `display_label` displays the string passed as a parameter immediately after the call to `display_init`.

```
# let display_init c =
    Graphics.set_color (get_gc_bcol (get_gc c)); display_rect  c ();
    let gc= get_gc c in
      use_gc gc;
      let (a,b) = get_gc_cur gc in
        Graphics.moveto (c.x+a) (c.y+b)
  let display_label s c () =
      display_init c; Graphics.draw_string s;;
val display_init : component -> unit = <fun>
val display_label : string -> component -> unit -> unit = <fun>
```

As this component is very simple it is not necessary to create any internal state. Only the function `display_label` knows the string to be displayed, which is passed by the

creation function.

```
# let create_label s lopt =
    let gc = make_default_context () in   set_gc gc lopt; use_gc gc;
      let (w,h) = Graphics.text_size s in
        let u = create_component w h  in
          u.mem <- (fun x → false);  u.display <- display_label s  u;
          u.info <- "Label"; u.gc <- gc;
          u;;
val create_label : string -> (string * opt_val) list -> component = <fun>
```

If we wish to change the colors of this component, we need to manipulate its graphic context directly.

The display of *label* `l1` below is depicted in figure 13.1.

```
# let courier_bold_24 = Sopt "*courier-bold-r-normal-*24*"
  and courier_bold_18  = Sopt "*courier-bold-r-normal-*18*";;
# let l1 = create_label "Login: " ["Font", courier_bold_24;
                                   "Background", Copt gray1];;
```


Figure 13.1: Displaying a *label*.

### *The panel* Component, Containers and Layout

A *panel* is a graphical area which can be a container. The function which creates a panel is very simple. It augments the general function for creating components with a boolean indicating whether it is a container. The functions for testing location within the *panel* and for display are those assigned by default in the `create_component` function.

```
# let create_panel b  w h  lopt =
    let u = create_component w h   in
      u.container <- b;
      u.info <- if b then "Panel container" else "Panel";
      let gc = make_default_context () in set_gc gc lopt; u.gc <- gc;
      u;;
val create_panel :
  bool -> int -> int -> (string * opt_val) list -> component = <fun>
```

The tricky part with containers lies in the positioning of their child components. We define two new layout functions: `center_layout` and `grid_layout`. The first, `center_layout` places a component at the center of a container:

```
# let center_layout c c1 lopt =
      c1.x <- c.x + ((c.w -c1.w) /2); c1.y <- c.y + ((c.h -c1.h) /2);;
```

```
val center_layout : component -> component -> 'a -> unit = <fun>
```

The second, `grid_layout` divides a container into a grid where each box has the same size. The layout options in this case are `"Col"` for the column number and `"Row"` for the row number.

```
# let grid_layout (a, b)  c c1 lopt =
    let px = theInt lopt "Col" 0
    and py = theInt lopt "Row" 0 in
     if (px >= 0) && (px < a) && ( py >=0) && (py < b) then
       let lw = c.w /a
       and lh = c.h /b in
            if (c1.w > lw) || (c1.h > lh) then
              failwith "grid_placement: too big component"
            else
              c1.x <- c.x + px * lw + (lw - c1.w)/2;
              c1.y <- c.y + py * lh + (lh - c1.h)/2;
       else  failwith "grid_placement: bad position";;
val grid_layout :
  int * int -> component -> component -> (string * opt_val) list -> unit =
  <fun>
```

It is clearly possible to define more. One can also customize a container by changing its layout function (`set_layout`). Figure 13.2 shows a *panel*, declared as a container, in which two *labels* have been added and which corresponds to the following program:

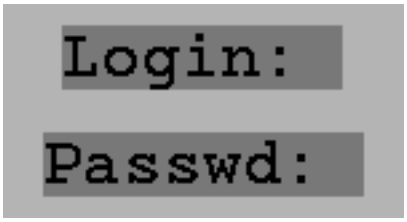


Figure 13.2: A *panel* component.

```
# let l2 = create_label "Passwd: " ["Font", courier_bold_24;
                                    "Background", Copt gray1] ;;
# let p1 = create_panel true 150 80 ["Background", Copt gray2] ;;
# set_layout  (grid_layout (1,2) p1) p1;;
# add_component p1 l1 ["Row", Iopt 1];;
# add_component p1 l2 ["Row", Iopt 0];;
```

Since the components need at least one parent so that they can be integrated into the interface, and since the `Graphics` library only supports one window, we must define a

principle window which is a container.

```
# let open_main_window w h =
    Graphics.close_graph();
    Graphics.open_graph (" "^(string_of_int w)^"x"^(string_of_int h));
    let u = create_component  w h in
       u.container <- true;
       u.info <- "Main Window";
       u;;
val open_main_window : int -> int -> component = <fun>
```

## *The Button Component*

A *button* is a component which displays a text in its graphical region and reacts to mouse clicks which occur there. To support this behavior it retains a piece of state, a value of type *button_state*, which contains the text and the mouse handling function.

```
# type button_state =
   { txt : string; mutable action :  button_state → unit } ;;
```

The function `create_bs` creates this state. The `set_bs_action` function changes the handling function and the function `get_bs_text` retrieves the text of a button.

```
# let create_bs s = {txt = s; action = fun x → ()}
  let set_bs_action bs f = bs.action <- f
  let get_bs_text bs = bs.txt;;
val create_bs : string -> button_state = <fun>
val set_bs_action : button_state -> (button_state -> unit) -> unit = <fun>
val get_bs_text : button_state -> string = <fun>
```

The display function is similar to that used by *labels* with the exception that the text derives this time from the state information belonging to the button. By default the listening function activates the action function when a mouse button is pressed.

```
# let display_button  c bs  () =
     display_init c;  Graphics.draw_string (get_bs_text bs)
  let listener_button c bs  e = match get_event e with
    MouseDown → bs.action bs; c.display (); true
  | _ → false;;
val display_button : component -> button_state -> unit -> unit = <fun>
val listener_button : component -> button_state -> rich_status -> bool =
  <fun>
```

We now have all we need to define the creation function for simple buttons:

```
# let create_button s lopt     =
```

```
    let bs = create_bs s in
      let gc = make_default_context () in
        set_gc gc lopt; use_gc gc;
        let w,h = Graphics.text_size (get_bs_text bs) in
          let u = create_component w h   in
            u.display <- display_button u bs;
            u.listener <- listener_button u bs;
            u.info <- "Button";
            u.gc <- gc;
            u,bs;;
val create_button :
  string -> (string * opt_val) list -> component * button_state = <fun>
```

This returns a tuple of which the first element is the button which has been created and the second is the internal state of the button. The latter value is particularly useful if we want to change the action function of the button since the button state is not accessible via the button function.

Figure 13.3 shows a *panel* to which a button has been added. We have associated an action function which displays the string contained by the button on the standard output.

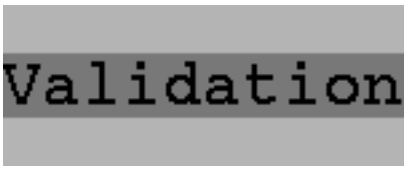


Figure 13.3: Button display and reaction to a mouseclick.

```
# let b,bs = create_button "Validation" ["Font", courier_bold_24;
                                         "Background", Copt gray1];;
# let p2 = create_panel true 150 60 ["Background", Copt gray2];;
# set_bs_action bs (fun bs → print_string ( (get_bs_text bs)^ "...");
                             print_newline());;
# set_layout  (center_layout p2) p2;;
# add_component p2 b  [];;
```

In contrast to *labels*, a button component knows how to react to a mouse click. To test this feature we send the event "mouse click" to a central position on the *panel* `p2`, which is occupied by the button. This causes the action associated with the button to be carried out:

```
# send_event (make_click MouseDown 75 30) p2;;
Validation...
- : bool = true
```

and returns the value `true` showing that the event has indeed been handled.

### *The choice* Component

The *choice* component allows us to select one of the choices offered using a mouse click. There is always a current choice. A mouse click on another choice causes the current choice to change and causes an action to be carried out. We use the same technique we used previously for simple buttons. We start by defining the state needed by a choice list:

```
# type choice_state =
    { mutable ind : int; values : string array; mutable sep : int;
      mutable height : int; mutable action : choice_state → unit } ;;
```

The index `ind` shows which string is to be highlighted in the list of `values`. The `sep` and `height` fields describe in pixels the distance between two choices and the height of a choice. The action function takes an argument of type *choice_state* as an input and does its job using the index.

We now define the function to create a set of status information and the function to change to way it is handled.

```
# let create_cs sa = {ind = 0; values = sa; sep = 2;
                      height = 1; action = fun x → ()}
  let set_cs_action cs f = cs.action <- f
  let get_cs_text cs = cs.values.(cs.ind);;
val create_cs : string array -> choice_state = <fun>
val set_cs_action : choice_state -> (choice_state -> unit) -> unit = <fun>
val get_cs_text : choice_state -> string = <fun>
```

The display function shows the list of all the possible choices and accentuates the current choice in inverse video. The event handling function reacts to a release of the mouse button.

```
# let display_choice c cs  () =
     display_init c;
     let (x,y) = Graphics.current_point()
     and nb = Array.length cs.values in
       for i = 0 to nb-1 do
         Graphics.moveto x (y + i*(cs.height+ cs.sep));
         Graphics.draw_string cs.values.(i)
       done;
       Graphics.set_color (get_gc_fcol (get_gc c));
       Graphics.fill_rect x (y+ cs.ind*(cs.height+ cs.sep)) c.w cs.height;
       Graphics.set_color (get_gc_bcol (get_gc c));
       Graphics.moveto x  (y + cs.ind*(cs.height + cs.sep));
       Graphics.draw_string cs.values.(cs.ind) ;;
val display_choice : component -> choice_state -> unit -> unit = <fun>

# let listener_choice c cs e = match e.re with
    MouseUp →
```

```
      let x = e.stat.Graphics.mouse_x
      and y = e.stat.Graphics.mouse_y in
        let cy = c.y in
          let i = (y - cy) / ( cs.height + cs.sep) in
            cs.ind <- i; c.display ();
            cs.action cs; true
  |   _  → false ;;
val listener_choice : component -> choice_state -> rich_status -> bool =
  <fun>
```

To create a list of possible choices we take a list of strings and a list of options, and we return the component itself along with its internal state.

```
# let create_choice lc lopt  =
    let sa =  (Array.of_list (List.rev lc)) in
    let cs = create_cs sa in
    let gc = make_default_context () in
      set_gc gc lopt;  use_gc gc;
        let awh = Array.map (Graphics.text_size) cs.values in
        let w = Array.fold_right (fun (x,y) → max x)  awh 0
        and h = Array.fold_right (fun (x,y) → max y)  awh 0 in
        let h1 = (h+cs.sep) * (Array.length sa) + cs.sep  in
        cs.height <- h;
        let u = create_component w h1   in
          u.display <- display_choice u cs;
          u.listener <- listener_choice u cs ;
          u.info <- "Choice "^ (string_of_int (Array.length cs.values));
          u.gc <- gc;
          u,cs;;
val create_choice :
  string list -> (string * opt_val) list -> component * choice_state = <fun>
```

The sequence of three pictures in figure 13.4 shows a *panel* to which a list of choices has been added. To it we have bound an action function which displays the chosen string to the standard output. The pictures arise from mouse clicks simulated by the following program.

```
# let c,cs = create_choice ["Helium"; "Gallium"; "Pentium"]
                           ["Font", courier_bold_24;
                            "Background", Copt gray1];;
# let p3 = create_panel true 110 110  ["Background", Copt gray2];;
# set_cs_action cs (fun cs → print_string ( (get_cs_text cs)^"...");
                             print_newline());;
# set_layout  (center_layout p3) p3;;
# add_component p3 c [];;
```


Figure 13.4: Displaying and selecting from a choice list.

Here also we can test the component straight away by sending several events. The following changes the selection, as is shown in the central picture in figure 13.4.

```
# send_event (make_click MouseUp 60 55 ) p3;;
Gallium...
- : bool = true
```

The sending of the following event selects the first element in the choice list

```
# send_event (make_click MouseUp 60 90 ) p3;;
Helium...
- : bool = true
```

### *The textfield* Component

The text input field, or *textfield*, is an area which enables us to input a text string. The text can be aligned to the left or (typically for a calculator) the right. Furthermore a cursor shows where the next character will be entered. Here we need a more complex internal state. This includes the text which is being entered, the direction of the justification, a description of the cursor, a description of how the characters are displayed and the action function.

```
# type textfield_state =
    { txt : string;
      dir : bool; mutable ind1 : int; mutable ind2 : int; len : int;
      mutable visible_cursor : bool; mutable cursor : char;
      mutable visible_echo : bool; mutable echo : char;
      mutable action : textfield_state → unit } ;;
```

To create this internal state we need the initial text, the number of characters available for the text input field and the justification of the text.

```
# let create_tfs txt size dir  =
    let l = String.length txt in
      (if size < l then failwith "create_tfs");
      let ind1 = if dir then 0 else size-1-l
```

```
      and ind2 = if dir then l else size-1 in
      let n_txt = (if dir then (txt^(String.make (size-l) ' '))
                          else ((String.make (size-l) ' ')^txt )) in
    {txt = n_txt; dir=dir; ind1 = ind1; ind2 = ind2; len=size;
     visible_cursor  = false;  cursor = ' '; visible_echo =  true; echo = ' ';
     action= fun x → ()};;
val create_tfs : string -> int -> bool -> textfield_state = <fun>
```

The following functions allow us to access various fields, including the displayed text.

```
# let set_tfs_action tfs f = tfs.action <- f
  let set_tfs_cursor b c tfs =  tfs.visible_cursor <- b; tfs.cursor <- c
  let set_tfs_echo b c tfs =  tfs.visible_echo <- b; tfs.echo <- c
  let get_tfs_text tfs =
    if tfs.dir then String.sub tfs.txt tfs.ind1 (tfs.ind2 - tfs.ind1)
    else String.sub tfs.txt (tfs.ind1+1) (tfs.ind2 - tfs.ind1);;
```

The `set_tfs_text` function changes the text within the internal state `tfs` of the component `tf` with the string `txt`.

```
# let set_tfs_text tf tfs txt =
     let l = String.length txt in
       if l > tfs.len then failwith "set_tfs_text";
       String.blit  (String.make tfs.len ' ') 0 tfs.txt 0 tfs.len;
       if tfs.dir then (String.blit txt 0 tfs.txt 0 l;
                    tfs.ind2 <- l )
       else   ( String.blit txt 0 tfs.txt (tfs.len -l) l;
                tfs.ind1 <- tfs.len-l-1 );
       tf.display ();;
val set_tfs_text : component -> textfield_state -> string -> unit = <fun>
```

Display operations must take account of how the character is echoed and the visibility of the cursor. The `display_textfield` function calls the `display_cursor` function which shows where the cursor is.

```
# let display_cursor c tfs =
    if tfs.visible_cursor then
      ( use_gc (get_gc c);
        let (x,y) = Graphics.current_point() in
        let (a,b) = Graphics.text_size " " in
        let shift =  a *  (if tfs.dir then max (min (tfs.len-1) tfs.ind2)  0
                           else tfs.ind2) in
          Graphics.moveto (c.x+x + shift) (c.y+y);
          Graphics.draw_char tfs.cursor);;
val display_cursor : component -> textfield_state -> unit = <fun>
# let display_textfield c tfs  () =
     display_init c;
     let s = String.make tfs.len ' '
```

```
      and txt = get_tfs_text tfs in
        let nl = String.length txt in
        if (tfs.ind1 >= 0) && (not tfs.dir) then
           Graphics.draw_string (String.sub s 0 (tfs.ind1+1) );
        if tfs.visible_echo  then (Graphics.draw_string (get_tfs_text tfs))
        else Graphics.draw_string (String.make (String.length txt) tfs.echo);
        if (nl > tfs.ind2) && (tfs.dir)
          then Graphics.draw_string (String.sub s tfs.ind2 (nl-tfs.ind2));
        display_cursor c tfs;;
val display_textfield : component -> textfield_state -> unit -> unit = <fun>
```

The event-listener function for this kind of component is more complex. According to the input direction (left or right justified) we may need to move the string which has already been input. Capture of focus is achieved by a mouse click in the input zone.

```
# let listener_text_field u tfs e =
       match e.re with
         MouseDown → take_key_focus e u ; true
       | KeyPress →
           ( if Char.code (get_key e)  >= 32 then
             begin
               ( if tfs.dir then
                 ( ( if tfs.ind2 >= tfs.len then (
                       String.blit tfs.txt 1 tfs.txt 0 (tfs.ind2-1);
                       tfs.ind2 <- tfs.ind2-1) );
                     tfs.txt.[tfs.ind2] <- get_key e;
                     tfs.ind2 <- tfs.ind2 +1 )
                 else
                 ( String.blit tfs.txt 1 tfs.txt 0 (tfs.ind2);
                   tfs.txt.[tfs.ind2] <- get_key e;
                   if tfs.ind1 >= 0 then tfs.ind1 <- tfs.ind1 -1
                 );
               )
             end
             else (
             ( match Char.code (get_key e) with
                 13 → tfs.action tfs
               |  9 → lose_key_focus e u
               |  8 → if (tfs.dir && (tfs.ind2 > 0))
                         then tfs.ind2 <- tfs.ind2 -1
                         else if (not tfs.dir) && (tfs.ind1 < tfs.len -1)
                              then tfs.ind1 <- tfs.ind1+1
               | _ → ()
             ))); u.display(); true
       | _ → false;;
val listener_text_field :
  component -> textfield_state -> rich_status -> bool = <fun>
```

The function which creates text entry fields repeats the same pattern we have seen in the previous components.

```
# let create_text_field  txt size dir lopt  =
    let tfs = create_tfs txt size dir
    and l = String.length txt in
     let gc = make_default_context () in
      set_gc gc lopt; use_gc gc;
      let (w,h) = Graphics.text_size (tfs.txt) in
        let u = create_component w h   in
          u.display <- display_textfield u tfs;
          u.listener <-  listener_text_field u tfs ;
          u.info <- "TextField"; u.gc <- gc;
          u,tfs;;
val create_text_field :
  string ->
  int -> bool -> (string * opt_val) list -> component * textfield_state =
  <fun>
```

This function returns a tuple consisting of the component itself, and the internal state of that component. We can test the creation of the component in figure 13.5 as follows:

```
# let tf1,tfs1 = create_text_field "jack"  8 true ["Font", courier_bold_24];;
# let tf2,tfs2 = create_text_field "koala" 8  false ["Font", courier_bold_24];;
# set_tfs_cursor true '_' tfs1;;
# set_tfs_cursor true '_' tfs2;;
# set_tfs_echo false '*' tfs2;;
# let p4 = create_panel true 140 80 ["Background", Copt gray2];;
# set_layout  (grid_layout (1,2) p4) p4;;
# add_component p4 tf1 ["Row", Iopt 1];;
# add_component p4 tf2 ["Row", Iopt 0];;
```

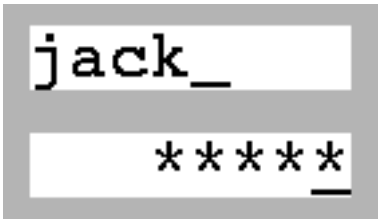


Figure 13.5: Text input component.

## *Enriched Components*

Beyond the components described so far, it is also possible to construct new ones, for example components with bevelled edges such as those in the calculator on page 136. To create this effect we construct a *panel* larger than the component, fill it out in a certain way and add the required component to the center.

```
# type border_state =
    {mutable relief : string; mutable line : bool;
     mutable bg2 : Graphics.color; mutable size : int};;
```

The creation function takes a list of options and constructs an internal state.

```
# let create_border_state lopt =
    {relief = theString lopt "Relief" "Flat";
     line = theBool lopt "Outlined" false;
     bg2 = theColor lopt "Background2" Graphics.black;
     size = theInt lopt "Border_size" 2};;
val create_border_state : (string * opt_val) list -> border_state = <fun>
```

We define the profile of the border used in the boxes of figure 5.6 (page 130) by defining the options `"Top"`, `"Bot"` and `"Flat"`.

```
# let display_border bs c1 c () =
    let x1 = c.x  and y1 = c.y in
    let x2 = x1+c.w-1 and y2 = y1+c.h-1 in
    let ix1 = c1.x and iy1 =  c1.y in
    let ix2 = ix1+c1.w-1 and iy2 = iy1+c1.h-1 in
    let border1 g = Graphics.set_color g;
      Graphics.fill_poly [| (x1,y1);(ix1,iy1);(ix2,iy1);(x2,y1) |] ;
      Graphics.fill_poly [| (x2,y1);(ix2,iy1);(ix2,iy2);(x2,y2) |]
    in
      let border2 g =  Graphics.set_color g;
        Graphics.fill_poly [| (x1,y2);(ix1,iy2);(ix2,iy2);(x2,y2) |] ;
        Graphics.fill_poly [| (x1,y1);(ix1,iy1);(ix1,iy2);(x1,y2) |]
      in
        display_rect c ();
        if bs.line then (Graphics.set_color (get_gc_fcol (get_gc c));
                         draw_rect x1 y1 c.w c.h);
       let b1_col = get_gc_bcol ( get_gc c)
       and b2_col = bs.bg2 in
        match bs.relief with
          "Top" →  (border1 b1_col; border2 b2_col)
        |  "Bot" → (border1 b2_col; border2 b1_col)
        |  "Flat" →  (border1 b1_col; border2 b1_col)
        |  s → failwith ("display_border: unknown relief: "^s)
  ;;
val display_border : border_state -> component -> component -> unit -> unit =
  <fun>
```

The function which creates a border takes a component and a list of options, it constructs a *panel* containing that component.

```
# let create_border c lopt =
      let bs = create_border_state lopt in
      let p = create_panel true (c.w + 2 * bs.size)
                                (c.h + 2 * bs.size) lopt in
        set_layout (center_layout p) p;
        p.display <- display_border bs c p;
        add_component p c [] ; p;;
val create_border : component -> (string * opt_val) list -> component = <fun>
```

Now we can test creating a component with a border on the *label* component and the text entry field `tf1` defined by in our previous tests. The result is show in figure 13.6.

```
# remove_component   p1 l1;;
# remove_component   p4 tf1;;
# let b1 = create_border l1 [] ;;
# let b2 = create_border tf1 ["Relief", Sopt "Top";
                              "Background", Copt Graphics.red;
                              "Border_size", Iopt 4];;
# let p5 = create_panel true 140 80 ["Background", Copt gray2];;
# set_layout  (grid_layout (1,2) p5) p5;;
# add_component p5 b1 ["Row", Iopt 1];;
# add_component p5 b2 ["Row", Iopt 0];;
```

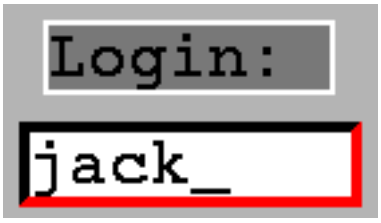


Figure 13.6: An enriched component.

## *Setting up the* `Awi` *Library*

The essential parts of our library have now been written. All declarations [2] of types and values which we have seen so far in this section can be grouped together in one file. This library consists of one single module. If the file is called `awi.ml` then we get

2. except for those used in our test examples

a module called `Awi`. The link between the name of the file and that of the module is described in chapter 14.

Compiling this file will produce a compiled interface file `awi.cmi` and, depending on the compiler being used, the bytecode itself `awi.cmo` or else the native machine code `awi.cmx`. To use the bytecode compiler we enter the following command

```
ocamlc -c awi.ml
```

To use it at the interactive toplevel, we need to load the bytecode of our new library with the command `#load "awi.cmo";;` having also previously ensured that we have loaded the `Graphics` library. We can then start calling functions from the module to create and work with components.

```
# open Awi;;
# create_component;;
- : int -> int -> Awi.component = <fun>
```

The result type of this function is `Awi.component`, chapter 14 explains more about this.

## *Example: A Franc-Euro Converter*

We will now build a currency converter between Francs and Euros using this new library. The actual job of conversion is trivial, but the construction of the interface will show how the components communicate with each other. While we are getting used to the new currency we need to convert in both directions. Here are the components we have chosen:

- a list of two choices to describe the direction of the conversion;
- two text entry fields for inputting values and displaying converted results;
- a simple button to request that the calculation be performed;
- two *labels* to show the meaning of each text entry field.

These different components are shown in figure 13.7.

Communication between the components is implemented by sharing state. For this purpose we define the type *state_conv* which hold the fields for francs (`a`), euros (`b`), the direction in which the conversion is to be performed (`dir`) and the conversion factors (`fa` and `fb`).

```
# type state_conv =
    { mutable a:float; mutable b:float; mutable dir : bool;
      fa : float; fb : float } ;;
```

We define the initial state as follows:

```
# let e =  6.55957074
```

```
  let fe = { a =0.0; b=0.0; dir = true; fa = e; fb = 1./. e};;
```

The conversion function returns a floating result following the direction of the conversion.

```
# let calculate fe =
    if fe.dir then fe.b <- fe.a /. fe.fa else fe.a <- fe.b /. fe.fb;;
val calculate : state_conv -> unit = <fun>
```

A mouse click on the list of two choices changes the direction of the conversion. The text of the choice strings is `"->"` and `"<-"`.

```
# let action_dir fe cs = match get_cs_text cs with
    "->" → fe.dir <- true
  | "<-" → fe.dir <- false
  | _ → failwith "action_dir";;
val action_dir : state_conv -> choice_state -> unit = <fun>
```

The action associated with the simple button causes the calculation to be performed and displays the result in one of the two text entry fields. For this to be possible we pass the two text entry fields as parameters to the action.

```
# let action_go fe tf_fr tf_eu tfs_fr tfs_eu  x =
    if fe.dir then
      let r = float_of_string (get_tfs_text tfs_fr) in
        fe.a <- r; calculate fe;
        let sr = Printf.sprintf "%.2f" fe.b in
            set_tfs_text tf_eu tfs_eu  sr
    else
      let r = float_of_string (get_tfs_text tfs_eu) in
        fe.b <- r; calculate fe;
        let sr = Printf.sprintf "%.2f" fe.a in
            set_tfs_text tf_fr tfs_fr sr;;
val action_go :
  state_conv ->
  component -> component -> textfield_state -> textfield_state -> 'a -> unit =
  <fun>
```

It now remains to build the interface. The following function takes a width, a height and a conversion state and returns the main container with the three active components.

```
# let create_conv w h fe =
  let gray1 = (Graphics.rgb 120 120 120) in
  let m = open_main_window w h
  and  p = create_panel true (w-4) (h-4)  []
  and  l1 = create_label "Francs" ["Font", courier_bold_24;
                                   "Background", Copt gray1]
```

```
  and l2 = create_label "Euros" ["Font", courier_bold_24;
                                 "Background", Copt gray1]
  and c,cs = create_choice ["->"; "<-"] ["Font", courier_bold_18]
  and tf1,tfs1 = create_text_field  "0" 10 false ["Font", courier_bold_18]
  and tf2,tfs2 = create_text_field "0" 10 false ["Font", courier_bold_18]
  and b,bs = create_button " Go " ["Font", courier_bold_24]
  in
    let gc = get_gc m in
      set_gc_bcol gc gray1;
      set_layout (grid_layout (3,2) m ) m;
      let tb1 = create_border tf1  []
      and tb2 = create_border tf2  []
      and bc = create_border c  []
      and bb =
        create_border  b
          ["Border_size", Iopt 4; "Relief", Sopt "Bot";
           "Background", Copt gray2; "Background2", Copt Graphics.black]
      in
        set_cs_action cs (action_dir fe);
        set_bs_action bs (action_go fe tf1 tf2 tfs1 tfs2);
        add_component m l1 ["Col",Iopt 0;"Row",Iopt 1];
        add_component m l2 ["Col",Iopt 2;"Row",Iopt 1];
        add_component m bc ["Col",Iopt 1;"Row",Iopt 1];
        add_component m tb1 ["Col",Iopt 0;"Row",Iopt 0];
        add_component m tb2 ["Col",Iopt 2;"Row",Iopt 0];
        add_component m bb  ["Col",Iopt 1;"Row",Iopt 0];
        m,bs,tf1,tf2;;
val create_conv :
  int ->
  int -> state_conv -> component * button_state * component * component =
  <fun>
```

The event handling loop is started on the container `m` constructed below. The resulting display is shown in figure 13.7.

```
# let (m,c,t1,t2) = create_conv 420 150 fe ;;
# display m ;;
```

One click on the choice list changes both the displayed text and the direction of the conversion because all the event handling closures share the same state.

## *Where to go from here*

Closures allow us to register handling methods with graphical components. It is however impossible to "reopen" these closures to extend an existing handler with additional behavior. We need to define a completely new handler. We discuss the possibilities for

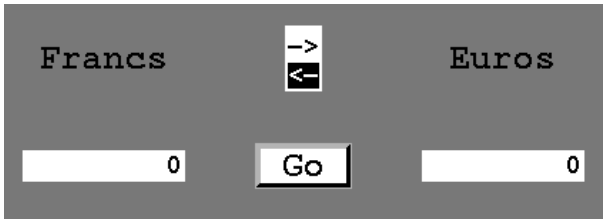


Figure 13.7: Calculator window.

extending handlers in chapter 16 where we compare the functional and object-oriented paradigms.

In our application many of the structures declared have fields with identical names (for example `txt`). The last declaration masks all previous occurences. This means that it becomes difficult to use the field names directly and this is why we have declared a set of access functions for every type we have defined. Another possibility would be to cut our library up into several modules. From then on field names could be disambiguated by using the module names. Nonetheless, with the help of the access functions, we can already make full use of the library. Chapter 14 returns to the topic of type overlaying and introduces abstract data types. The use of overlaying can, among other things, increase robustness by preventing the modification of sensitive data fields, such as the parent child relationships between the components which should not allow the construction of a circular graph.

There are many possible ways to improve this library.

One criterion in our design for components was that it should be possible to write new ones. It is fairly easy to create components of an arbitrary shape by using new definitions of the `mem` and `display` functions. In this way one could create buttons which have an oval or tear-shaped form.

The few layout algorithms presented are not as helpful as they could be. One could add a grid layout whose squares are of variable size and width. Or maybe we want to place components alongside each other so long as there is enough room. Finally we should anticipate the possibility that a change to the size of a component may be propagated to its children.

# *Finding Least Cost Paths*

Many applications need to find least cost paths through weighted directed graphs. The problem is to find a path through a graph in which non-negative weights are associated with the arcs. We will use Dijkstra's algorithm to determine the path.

This will be an opportunity to use several previously introduced libraries. In the order of appearance, the following modules will be used: `Genlex` and `Printf` for input and

output, the module `Weak` to implement a cache, the module `Sys` to track the time saved by a cache, and the `Awi` library to construct a graphical user interface. The module `Sys` is also used to construct a standalone application that takes the name of a file describing the graph as a command line argument.

## *Graph Represention*s

A weighted directed graph is defined by a set of nodes, a set of edges, and a mapping from the set of edges to a set of values. There are numerous implementations of the data type *weighted directed graph*.

- adjacency matrices:
  each element $(m(i,j))$ of the matrix represents an edge from node $i$ to $j$ and the value of the element is the weight of the edge;
- adjacency lists:
  each node $i$ is associated with a list $[(j_1,w_1);..;(j_n,w_n)]$ of nodes and each triple $(i,j_k,w_k)$ is an edge of the graph, with weight $w_k$;
- a triple:
  a list of nodes, a list of edges and a function that computes the weights of the edges.

The behavior of the representations depends on the size and the number of edges in the graph. Since one goal of this application is to show how to cache certain previously executed computations without using all of memory, an adjacency matrix is used to represent weighted directed graphs. In this way, memory usage will not be increased by list manipulations.

```
# type  cost  = Nan | Cost of float;;
# type  adj_mat =  cost array array;;
# type 'a graph = { mutable ind : int;
                    size : int;
                    nodes : 'a array;
                    m : adj_mat};;
```

The field `size` indicates the maximal number of nodes, the field `ind` the actual number of nodes.

We define functions to let us create graphs edge by edge.

The function to create a graph takes as arguments a node and the maximal number of nodes.

```
# let create_graph n s =
    { ind = 0; size = s; nodes = Array.create s n;
      m = Array.create_matrix s s Nan } ;;
val create_graph : 'a -> int -> 'a graph = <fun>
```

The function `belongs_to` checks if the node `n` is contained in the graph `g`.

```
# let belongs_to n g =
    let rec aux i =
      (i < g.size) & ((g.nodes.(i) = n) or (aux (i+1)))
    in aux 0;;
val belongs_to : 'a -> 'a graph -> bool = <fun>
```

The function `index` returns the index of the node `n` in the graph `g`. If the node does not exist, a `Not_found` exception is thrown.

```
# let index n g =
    let rec aux i =
      if i >= g.size then raise Not_found
      else if g.nodes.(i) = n then i
           else aux (i+1)
    in aux 0 ;;
val index : 'a -> 'a graph -> int = <fun>
```

The next two functions are for adding nodes and edges of cost `c` to graphs.

```
# let add_node n g =
    if g.ind = g.size then failwith "the graph is full"
    else if belongs_to n g then failwith "the node already exists"
    else (g.nodes.(g.ind) <- n; g.ind <- g.ind + 1) ;;
val add_node : 'a -> 'a graph -> unit = <fun>
# let add_edge e1 e2 c g =
    try
      let x = index e1 g and y = index e2 g in
        g.m.(x).(y) <- Cost c
    with Not_found → failwith "node does not exist" ;;
val add_edge : 'a -> 'a -> float -> 'a graph -> unit = <fun>
```

Now it is easy to create a complete weighted directed graph starting with a list of nodes and edges. The function `test_aho` constructs the graph of figure 13.8:

```
# let test_aho () =
    let g = create_graph "nothing" 5 in
      List.iter (fun x → add_node x g) ["A"; "B"; "C"; "D"; "E"];
      List.iter (fun (a,b,c) → add_edge a b c g)
        ["A","B",10.;
         "A","D",30.;
         "A","E",100.0;
         "B","C",50.;
         "C","E",10.;
         "D","C",20.;
         "D","E",60.];
      for i=0 to g.ind -1 do g.m.(i).(i) <- Cost 0.0 done;
       g;;
```

```
val test_aho : unit -> string graph = <fun>
# let a = test_aho();;
val a : string graph =
  {ind=5; size=5; nodes=[|"A"; "B"; "C"; "D"; "E"|];
   m=[|[|Cost 0; Cost 10; Nan; Cost 30; Cost ...|]; ...|]}
```

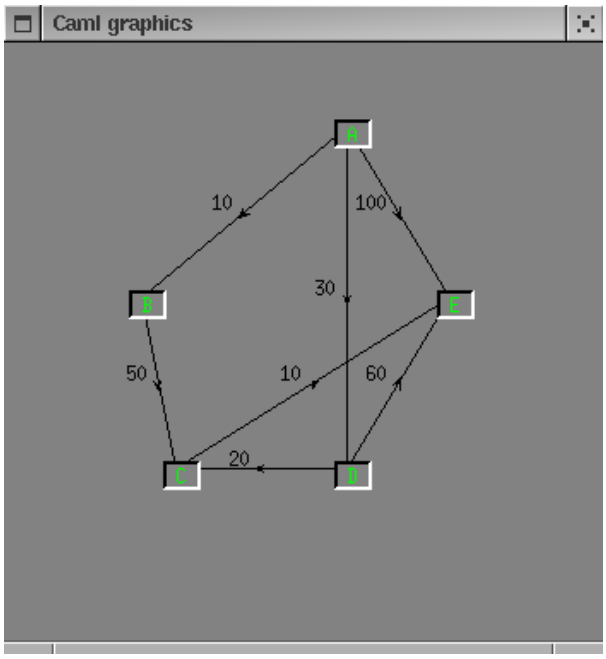


Figure 13.8: The test graph

## *Constructing Graphs*

It is tedious to directly construct graphs in a program. To avoid this, we define a concise textual representation for graphs. We can define the graphs in text files and construct them in applications by reading the text files.

The textual representation for a graph consists of lines of the following forms:

- the number of nodes: `SIZE number`;
- the name of a node: `NODE name`;
- the cost of an edge: `EDGE name1 name2 cost`;
- a comment: `# comment`.

For example, the following file, `aho.dat`, describes the graph of figure 13.8 :

```
SIZE 5
```

```
NODE A
NODE B
NODE C
NODE D
NODE E
EDGE A B 10.0
EDGE A D 30.0
EDGE A E 100.0
EDGE B C 50.
EDGE C E 10.
EDGE D C 20.
EDGE D E 60.
```

To read graph files, we use the lexical analysis module `Genlex`. The lexical analyser is constructed from a list of keywords `keywords`.

The function `parse_line` executes the actions associated to the key words by modifying the reference to a graph.

```
# let keywords = [ "SIZE"; "NODE"; "EDGE"; "#"];;
val keywords : string list = ["SIZE"; "NODE"; "EDGE"; "#"]
# let lex_line l = Genlex.make_lexer keywords (Stream.of_string l);;
val lex_line : string -> Genlex.token Stream.t = <fun>
# let parse_line g s = match  s with parser
    [< '(Genlex.Kwd  "SIZE"); '(Genlex.Int n) >] →
        g := create_graph "" n
  | [< '(Genlex.Kwd  "NODE"); '(Genlex.Ident name) >] →
        add_node name !g
  | [< '(Genlex.Kwd  "EDGE"); '(Genlex.Ident e1);
       '(Genlex.Ident  e2); '(Genlex.Float c) >] →
        add_edge e1 e2 c !g
  | [< '(Genlex.Kwd  "#") >] → ()
  | [<>] → () ;;
val parse_line : string graph ref -> Genlex.token Stream.t -> unit = <fun>
```

The analyzer is used to define the function creating a graph from the description in the file:

```
# let create_graph name =
    let g = ref {ind=0; size=0; nodes =[||]; m = [||]} in
    let ic = open_in name in
     try
       print_string ("Loading "^name^": ");
       while true do
        print_string ".";
         let l = input_line ic in parse_line g (lex_line l)
       done;
       !g
     with End_of_file → print_newline(); close_in ic; !g ;;
```

```
val create_graph : string -> string graph = <fun>
```
The following command constructs a graph from the file `aho.dat`.
```
# let b = create_graph "PROGRAMMES/aho.dat" ;;
Loading PROGRAMMES/aho.dat: ..............
val b : string graph =
  {ind=5; size=5; nodes=[|"A"; "B"; "C"; "D"; "E"|];
   m=[|[|Nan; Cost 10; Nan; Cost 30; Cost 100|]; ...|]}
```

## *Dijkstra's Algorithm*

Dijkstra's algorithm finds a least cost path between two nodes. The cost of a path between node $n1$ and node $n2$ is the sum of the costs of the edges on that path. The algorithm requires that costs always be positive, so there is no benefit in passing through a node more than once.

Dijkstra's algorithm effectively computes the minimal cost paths of all nodes of the graph which can be reached from a source node $n1$. The idea is to consider a set containing only nodes of which the least cost path to $n1$ is already known. This set is enlarged successively, considering nodes which can be accessed directly by an edge from one of the nodes already contained in the set. From these candidates, the one with the best cost path to the source node is added to the set.

To keep track of the state of the computation, the type *comp_state* is defined, as well as a function for creating an initial state:
```
# type comp_state = { paths : int array;
                      already_treated : bool array;
                      distances : cost array;
                      source : int;
                      nn : int};;
# let create_state () =  { paths = [||]; already_treated = [||]; distances = [||];
                           nn = 0; source = 0};;
```
The field `source` contains the start node. The field `already_treated` indicates the nodes whose optimal path from the source is already known. The field `nn` indicates the total number of the graph's nodes. The vector `distances` holds the minimal distances between the source and the other nodes. For each node, the vector `path` contains the preceding node on the least cost path. The path to the source can be reconstructed from each node by using `path`.

### *Cost Functions*

Four functions on costs are defined: `a_cost` to test for the existence of an edge, `float_of_cost` to return the floating point value, `add_cost` to add two costs and `less_cost` to check if one cost is smaller than another.

```
# let a_cost c = match c with Nan → false | _-> true;;
```

```
val a_cost : cost -> bool = <fun>
# let float_of_cost c = match c with
    Nan → failwith "float_of_cost"
  | Cost x → x;;
val float_of_cost : cost -> float = <fun>
# let add_cost  c1 c2 = match (c1,c2) with
    Cost x, Cost y → Cost (x+.y)
  | Nan, Cost y → c2
  | Cost x, Nan → c1
  | Nan, Nan →  c1;;
val add_cost : cost -> cost -> cost = <fun>
# let less_cost  c1 c2 = match (c1,c2) with
    Cost x, Cost y → x < y
  | Cost x, Nan → true
  | _, _ →  false;;
val less_cost : cost -> cost -> bool = <fun>
```

The value `Nan` plays a special role in the computations and in the comparison. We will come back to this when we have presented the main function (page 388).

## *Implementing the Algorithm*

The search for the next node with known least cost path is divided into two functions. The first, `first_not_treated`, selects the first node not already contained in the set of nodes with known least cost paths. This node serves as the initial value for the second function, `least_not_treated`, which returns a node not already in the set with a best cost path to the source. This path will be added to the set.

```
# exception Found of int;;
exception Found of int
# let first_not_treated cs =
    try
      for i=0 to cs.nn-1 do
        if not cs.already_treated.(i) then raise (Found i)
      done;
      raise Not_found;
      0
    with Found i → i ;;
val first_not_treated : comp_state -> int = <fun>
# let least_not_treated p cs =
    let ni = ref p
    and nd = ref cs.distances.(p) in
      for i=p+1 to cs.nn-1 do
        if not cs.already_treated.(i) then
         if less_cost cs.distances.(i)  !nd then
           ( nd := cs.distances.(i);
             ni := i )
      done;
      !ni,!nd;;
```

```
val least_not_treated : int -> comp_state -> int * cost = <fun>
```

The function one_round selects a new node, adds it to the set of treated nodes and computes the distances for any next candidates.

```
# exception No_way;;
exception No_way
# let one_round cs  g =
    let p = first_not_treated cs  in
    let np,nc = least_not_treated p cs in
      if not(a_cost nc  ) then raise No_way
      else
      begin
        cs.already_treated.(np) <- true;
        for i = 0 to cs.nn -1 do
          if not cs.already_treated.(i) then
          if a_cost g.m.(np).(i) then
            let ic = add_cost cs.distances.(np)  g.m.(np).(i) in
              if less_cost ic cs.distances.(i)   then (
                cs.paths.(i) <- np;
                cs.distances.(i) <- ic
               )
        done;
        cs
      end;;
val one_round : comp_state -> 'a graph -> comp_state = <fun>
```

The only thing left in the implementation of Dijkstra's algorithm is to iterate the preceding function. The function dij takes a node and a graph as arguments and returns a value of type comp_state, with the information from which the least cost paths from the source to all the reachable nodes of the graph can be deduced.

```
# let dij s g =
    if belongs_to s g then
    begin
      let i = index s g in
      let cs = { paths = Array.create g.ind (-1) ;
                 already_treated = Array.create g.ind false;
                 distances = Array.create g.ind Nan;
                 nn = g.ind;
                 source = i}  in
        cs.already_treated.(i) <- true;
        for j=0 to g.ind-1 do
          let c = g.m.(i).(j) in
            cs.distances.(j) <- c;
            if a_cost c then cs.paths.(j) <- i
        done;
        try
```

```
          for k = 0 to cs.nn-2 do
            ignore(one_round cs g)
          done;
          cs
        with No_way → cs
    end
    else failwith "dij: node unknown";;
val dij : 'a -> 'a graph -> comp_state = <fun>
```

`Nan` is the initial value of the distances. It represents an infinite distance, which conforms to the comparison function `less_cost`. In contrast, for the addition of costs (function `add_cost`), this value is treated as a zero. This allows a simple implementation of the table of distances.

Now the search with Dijkstra's algorithm can be tested.

```
# let g = test_aho ();;
# let r = dij "A" g;;
```

The return values are:

```
# r.paths;;
- : int array = [|0; 0; 3; 0; 2|]
# r.distances;;
- : cost array = [|Cost 0; Cost 10; Cost 50; Cost 30; Cost 60|]
```

## *Displaying the Results*

To make the results more readable, we now define a display function.

The table `paths` of the state returned by `dij` only contains the last edges of the computed paths. In order to get the entire paths, it is necessary to recursively go back to the source.

```
# let display_state f (g,st)  dest =
    if belongs_to dest g then
       let d = index dest g in
         let rec aux is =
            if is = st.source then Printf.printf "%a"  f g.nodes.(is)
            else (
              let old = st.paths.(is) in
              aux old;
              Printf.printf " -> (%4.1f) %a" (float_of_cost g.m.(old).(is))
                                             f g.nodes.(is)
            )
         in
           if not(a_cost st.distances.(d)) then Printf.printf "no way\n"
           else (
```

```
            aux d;
            Printf.printf " = %4.1f\n" (float_of_cost st.distances.(d));;
val display_state :
  (out_channel -> 'a -> unit) -> 'a graph * comp_state -> 'a -> unit = <fun>
```

This recursive function uses the command stack to display the nodes in the right order. Note that the use of the format `"a"` requires the function parameter `f` to preserve the polymorphism of the graphs for the display.

The optimal path between the nodes "A" (index 0) and "E" (index 4) is displayed in the following way:

```
# display_state (fun x y → Printf.printf "%s!" y) (a,r) "E";;
A! -> (30.0) D! -> (20.0) C! -> (10.0) E! = 60.0
- : unit = ()
```

The different nodes of the path and the costs of each route are shown.

## Introducing a Cache

Dijkstra's algorithm computes all least cost paths starting from a source. The idea of preserving these least cost paths for the next inquiry with the same source suggests itself. However, this storage could occupy a considerable amount of memory. This suggests the use of "weak pointers." If the results of a computation starting from a source are stored in a table of weak pointers, it will be possible for the next computation to check if the computation has already been done. Because the pointers are weak, the memory occupied by the states can be freed by the garbage collector if needed. This avoids interrupting the rest of the program through the allocation of too much memory. In the worst case, the computation has to be repeated for a future inquiry.

### Implementing a Cache

A new type *'a comp_graph* is defined:

```
# type 'a comp_graph =
    { g : 'a graph; w : comp_state Weak.t } ;;
```

The fields `g` and `w` correspond to the graph and to the table of weak pointers, pointing to the computation states for each possible source.

Such values are constructed by the function `create_comp_graph`.

```
# let create_comp_graph g =
    { g = g;
      w = Weak.create g.ind } ;;
val create_comp_graph : 'a graph -> 'a comp_graph = <fun>
```

The function `dij_quick` checks to see if the computation has already been done. If it has, the stored result is returned. Otherwise, the computation is executed and the result is registered in the table of weak pointers.

```
# let dij_quick s cg =
```

```
    let i = index s cg.g in
      match Weak.get cg.w i with
        None → let cs = dij s cg.g in
                  Weak.set cg.w i (Some cs);
                  cs
      | Some cs → cs;;
val dij_quick : 'a -> 'a comp_graph -> comp_state = <fun>
```

The display function still can be used:

```
# let cg_a = create_comp_graph a in
  let r = dij_quick "A" cg_a in
   display_state (fun x y → Printf.printf "%s!" y) (a,r) "E" ;;
A! -> (30.0) D! -> (20.0) C! -> (10.0) E! = 60.0
- : unit = ()
```

## *Performance Evaluation*

We will test the performance of the functions `dij` and `dij_quick` by iterating each one on a random list of sources. In this way an application which frequently computes least cost paths is simulated (for example a railway route planning system).

We define the following function to time the calculations:

```
# let exe_time f g ss =
    let t0 = Sys.time() in
     Printf.printf "Start (%5.2f)\n" t0;
     List.iter (fun s → ignore(f s g)) ss;
     let t1 = Sys.time() in
      Printf.printf "End (%5.2f)\n" t1;
      Printf.printf "Duration = (%5.2f)\n" (t1 -. t0) ;;
val exe_time : ('a -> 'b -> 'c) -> 'b -> 'a list -> unit = <fun>
```

We create a random list of 20000 nodes and measure the performance on the graph `a`:

```
# let ss =
   let ss0 = ref [] in
   let i0 = int_of_char 'A' in
   let new_s i = Char.escaped (char_of_int (i0+i)) in
   for i=0 to 20000 do ss0 := (new_s (Random.int a.size))::!ss0 done;
   !ss0 ;;
val ss : string list =
  ["A"; "B"; "D"; "A"; "E"; "C"; "B"; "B"; "D"; "E"; "B"; "E"; "C"; "E"; "E";
   "D"; "D"; "A"; "E"; ...]
# Printf.printf"Function dij :\n";
   exe_time dij a ss ;;
Function dij :
Start ( 1.09)
```

```
End ( 1.41)
Duration = ( 0.32)
- : unit = ()
# Printf.printf"Function dij_quick :\n";
   exe_time dij_quick (create_comp_graph a) ss ;;
Function dij_quick :
Start ( 1.41)
End ( 1.44)
Duration = ( 0.03)
- : unit = ()
```

The results confirm our assumption. The direct access to a result held in the cache is considerably faster than a second computation of the result.

## A Graphical Interface

We use the `Awi` library to construct a graphical interface to display graphs. The interface allows selection of the source and destination nodes of the path. When the path is found, it is displayed graphically. We define the type *'a gg*, containing fields describing the graph and the computation, as well as fields of the graphical interface.

```
#  #load "PROGRAMMES/awi.cmo";;

# type 'a gg = { mutable src : 'a * Awi.component;
                 mutable dest : 'a * Awi.component;
                 pos : (int * int) array;
                 cg : 'a comp_graph;
                 mutable state : comp_state;
                 mutable main : Awi.component;
                 to_string : 'a → string;
                 from_string : string → 'a } ;;
```

The fields `src` and `dest` are tuples (node, component), associating a node and a component. The field `pos` contains the position of each component. The field `main` is the main container of the set of components. The two functions `to_string` and `from_string` are conversion functions between type *'a* and strings. The elements necessary to construct these values are the graph information, the position table and the conversion functions.

```
# let create_gg cg vpos ts fs =
    {src = cg.g.nodes.(0),Awi.empty_component;
     dest = cg.g.nodes.(0),Awi.empty_component;
     pos = vpos;
     cg = cg;
     state = create_state () ;
```

```
      main = Awi.empty_component;
      to_string = ts;
      from_string = fs};;
val create_gg :
  'a comp_graph ->
  (int * int) array -> ('a -> string) -> (string -> 'a) -> 'a gg = <fun>
```

## *Visualisation*

In order to display the graph, the nodes have to be drawn, and the edges have to be traced. The nodes are represented by button components of the `Awi` library. The edges are traced directly in the main window. The function `display_edge` displays the edges. The function `display_shortest_path` displays the found path in a different color.

**Drawing Edges** An edge connects two nodes and has an associated weight. The connection between two nodes can be represented by a line. The main difficulty is indicating the orientation of the line. We choose to represent it by an arrow. The arrow is rotated by the angle the line has with the abscissa (the $x$-axis) to give it the proper orientation. Finally, the costs are displayed beside the edge.

To draw the arrow of an edge we define the functions `rotate` and `translate` which care respectively for rotation and shifting. The function `display_arrow` draws the arrow.

```
# let rotate l a =
    let ca = cos a  and sa = sin a in
      List.map (function (x,y) → ( x*.ca +. -.y*.sa, x*.sa +. y*.ca)) l;;
val rotate : (float * float) list -> float -> (float * float) list = <fun>
# let translate l (tx,ty) =
    List.map (function (x,y) → (x +. tx, y +. ty)) l;;
val translate :
  (float * float) list -> float * float -> (float * float) list = <fun>
# let display_arrow (mx,my) a =
    let triangle = [(5.,0.); (-3.,3.); (1.,0.); (-3.,-3.); (5.,0.)] in
    let tr =  rotate triangle a in
    let ttr = translate tr (mx,my) in
    let tt = List.map (function (x,y) → (int_of_float x, int_of_float y)) ttr
    in
      Graphics.fill_poly (Array.of_list tt);;
val display_arrow : float * float -> float -> unit = <fun>
```

The position of the text indicating the weight of an edge depends on the angle of the edge.

```
# let display_label (mx,my) a lab =
    let (sx,sy) = Graphics.text_size lab in
    let pos = [ float(-sx/2),float(-sy) ] in
    let pr = rotate pos a in
```

```
    let pt = translate pr (mx,my) in
      let px,py = List.hd pt in
        let ox,oy = Graphics.current_point () in
          Graphics.moveto ((int_of_float mx)-sx-6)
                          ((int_of_float my) );
          Graphics.draw_string lab;
          Graphics.moveto ox oy;;
val display_label : float * float -> float -> string -> unit = <fun>
```

The preceding functions are now used by the function `display_edge`. Parameters are the graphical interface `gg`, the nodes `i` and `j`, and the color (`col`) to use.

```
# let display_edge gg col i j =
    let g = gg.cg.g in
    let x,y = gg.main.Awi.x,gg.main.Awi.y in
      if a_cost g.m.(i).(j) then (
        let (a1,b1) = gg.pos.(i)
        and (a2,b2) = gg.pos.(j) in
          let x0,y0 = x+a1,y+b1   and x1,y1 = x+a2,y+b2 in
          let rxm =  (float(x1-x0)) /. 2.  and rym =  (float(y1-y0)) /. 2. in
          let xm = (float x0) +. rxm and ym = (float y0) +. rym in
            Graphics.set_color col;
            Graphics.moveto x0 y0;
            Graphics.lineto x1 y1;
            let a =  atan2 rym rxm in
            display_arrow (xm,ym) a;
            display_label (xm,ym) a
              (string_of_float(float_of_cost g.m.(i).(j))));;
val display_edge : 'a gg -> Graphics.color -> int -> int -> unit = <fun>
```

**Displaying a Path** To display a path, all edges along the path are displayed. The graphical display of a path towards a destination uses the same technique as the textual display.

```
# let rec display_shortest_path gg col dest =
    let g = gg.cg.g in
      if belongs_to dest g then
        let d = index dest g in
         let rec aux is =
            if is = gg.state.source then ()
            else (
              let old = gg.state.paths.(is) in
                display_edge gg col old is;
                aux old )
         in
           if not(a_cost gg.state.distances.(d)) then Printf.printf "no way\n"
           else aux d;;
```

```
val display_shortest_path : 'a gg -> Graphics.color -> 'a -> unit = <fun>
```

**Displaying a Graph** The function `display_gg` displays a complete graph. If the destination node is not empty, the path between the source and the destination is traced.

```
#  let display_gg gg () =
     Awi.display_rect gg.main ();
     for i=0 to gg.cg.g.ind -1 do
       for j=0 to gg.cg.g.ind -1 do
         if i<> j then display_edge gg (Graphics.black) i j
       done
     done;
     if snd gg.dest != Awi.empty_component then
        display_shortest_path gg Graphics.red (fst gg.dest);;
val display_gg : 'a gg -> unit -> unit = <fun>
```

## *The Node Component*

The nodes still need to be drawn. Since the user is allowed to choose the source and destination nodes, we define a component for nodes.

The user's main action is choosing the end nodes of the path to be found. Thus a node must be a component that reacts to mouse clicks, using its state to indicate if it has been chosen as a source or destination. We choose the button component, which reacts to mouse clicks.

**Node Actions** It is necessary to indicate node selection. To show this, the background color of a node is changed by the function `inverse`.

```
# let inverse b =
     let gc = Awi.get_gc b in
       let fcol = Awi.get_gc_fcol gc
       and bcol = Awi.get_gc_bcol gc in
         Awi.set_gc_bcol gc fcol;
         Awi.set_gc_fcol gc bcol;;
val inverse : Awi.component -> unit = <fun>
```

The function `action_click` effects this selection. It is called when a node is clicked on by the mouse. As parameters it takes the node associated with the button and the graph to modify the source or the destination of the search. When both nodes are selected, the function `dij_quick` finds a least cost path.

```
# let action_click node gg b bs =
     let (s1,s) = gg.src
     and (s2,d) = gg.dest in
```

```
      if  s == Awi.empty_component then (
        gg.src  <- (node,b); inverse b )
      else
       if d == Awi.empty_component then (
         inverse b;
         gg.dest <- (node,b);
         gg.state <- dij_quick s1 gg.cg;
        display_shortest_path gg (Graphics.red) node
       )
       else (inverse s; inverse d;
             gg.dest <- (s2,Awi.empty_component);
             gg.src <- node,b; inverse b);;
val action_click : 'a -> 'a gg -> Awi.component -> 'b -> unit = <fun>
```

**Creating an Interface** The main function to create an interface takes an interface graph and a list of options, creates the different components and associates them with the graph. The parameters are the graph (`gg`), its dimensions (`gw` and `gh`), a list of graph and node options (`lopt`) and a list of node border options (`lopt2`).

```
# let main_gg gg gw gh lopt lopt2 =
    let gc = Awi.make_default_context () in
      Awi.set_gc gc lopt;
  (* compute the maximal button size *)
      let vs = Array.map gg.to_string gg.cg.g.nodes in
      let vsize = Array.map Graphics.text_size  vs in
        let w = Array.fold_right (fun (x,y) → max x)  vsize 0
        and h = Array.fold_right (fun (x,y) → max y)  vsize 0 in
  (* create the main panel *)
        gg.main <- Awi.create_panel true gw gh lopt;
        gg.main.Awi.display <- display_gg gg;
  (* create the buttons *)
          let vb_bs =
            Array.map (fun x → x,Awi.create_button (" "^(gg.to_string x)^" ")
                        lopt)
                      gg.cg.g.nodes in
            let f_act_b = Array.map (fun (x,(b,bs)) →
                                       let ac = action_click x  gg b
                                       in Awi.set_bs_action bs ac)  vb_bs in
            let bb =
              Array.map (function (_,(b,_)) → Awi.create_border b lopt2) vb_bs
            in
              Array.iteri
                (fun i (b) → let x,y = gg.pos.(i) in
                             Awi.add_component gg.main  b
                               ["PosX",Awi.Iopt (x-w/2);
                                "PosY", Awi.Iopt (y-h/2)]) bb;
```

```
            ();;
val main_gg :
  'a gg ->
  int ->
  int -> (string * Awi.opt_val) list -> (string * Awi.opt_val) list -> unit =
  <fun>
```

The buttons are created automatically. They are positioned on the main window.

**Testing the Interface** Everything is ready to create an interface now. We use a graph whose nodes are character strings to simplify the conversion functions. We construct the graph gg as follows:

```
# let id x = x;;
# let pos =  [| 200, 300; 80, 200 ; 100, 100; 200, 100;  260, 200 |];;
# let gg = create_gg  (create_comp_graph (test_aho())) pos id id;;
# main_gg  gg 400 400 ["Background", Awi.Copt  (Graphics.rgb 130 130 130);
               "Foreground",Awi.Copt  Graphics.green]
   [ "Relief", Awi.Sopt "Top";"Border_size", Awi.Iopt 2];;
```

Calling `Awi.loop true false gg.main;;` starts the interaction loop of the `Awi` library.

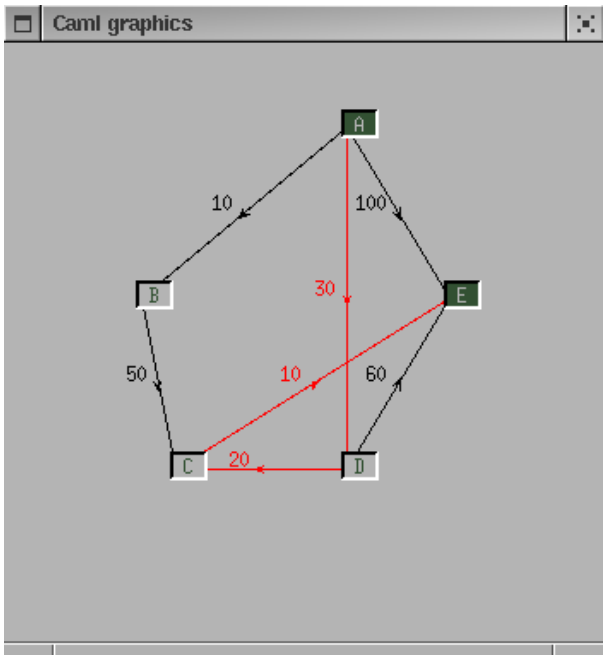


Figure 13.9: Selecting the nodes for a search

Figure 13.9 shows the computed path between the nodes "A" and "E". The edges on the path have changed their color.

## Creating a Standalone Application

We will now show the steps needed to construct a standalone application. The application takes the name of a file describing the graph as an argument. For standalone applications, it is not necesary to have an Objective Caml distribution on the execution machine.

### A Graph Description File

The file containes information about the graph as well as information used for the graphical interface. For the latter information, we define a second format. From this graphical description, we construct a value of the type *g_info*.

```
# type g_info = {npos : (int * int) array;
                 mutable opt : Awi.lopt;
                 mutable g_w : int;
                 mutable g_h : int};;
```

The format for the graphical information is described by the four key words of list key2.

```
# let key2 = ["HEIGHT"; "LENGTH"; "POSITION"; "COLOR"];;
val key2 : string list = ["HEIGHT"; "LENGTH"; "POSITION"; "COLOR"]
# let lex2 l = Genlex.make_lexer key2 (Stream.of_string l);;
val lex2 : string -> Genlex.token Stream.t = <fun>

# let pars2 g gi s = match s with parser
      [< '(Genlex.Kwd "HEIGHT"); '(Genlex.Int i) >] → gi.g_h <- i
  |   [< '(Genlex.Kwd "LENGTH"); '(Genlex.Int i) >] → gi.g_w <- i
  |   [< '(Genlex.Kwd "POSITION"); '(Genlex.Ident s);
         '(Genlex.Int i); '(Genlex.Int j) >] → gi.npos.(index s g) <- (i,j)
  |   [< '(Genlex.Kwd "COLOR"); '(Genlex.Ident s);
         '(Genlex.Int r); '(Genlex.Int g); '(Genlex.Int b) >] →
              gi.opt <- (s, Awi.Copt (Graphics.rgb r g b)) :: gi.opt
  | [<>] → ();;
val pars2 : string graph -> g_info -> Genlex.token Stream.t -> unit = <fun>
```

### Creating the Application

The function create_graph takes the name of a file as input and returns a couple composed of a graph and associated graphical information.

```
# let create_gg_graph name =
    let g = create_graph name in
```

```
      let gi = {npos = Array.create g.size (0,0); opt=[]; g_w =0; g_h = 0;} in
    let ic = open_in name in
    try
      print_string ("Loading (pass 2) " ^name ^" : ");
      while true do
        print_string ".";
        let l = input_line ic in pars2 g gi (lex2 l)
      done ;
      g,gi
    with End_of_file → print_newline(); close_in ic; g,gi;;
val create_gg_graph : string -> string graph * g_info = <fun>
```

The function `create_app` constructs the interface of a graph.

```
# let create_app name =
    let g,gi = create_gg_graph name in
    let size = (string_of_int gi.g_w) ^ "x" ^ (string_of_int gi.g_h) in
    Graphics.open_graph (" "^size);
    let gg = create_gg (create_comp_graph g) gi.npos id id in
    main_gg gg gi.g_w gi.g_h
      [ "Background", Awi.Copt (Graphics.rgb 130 130 130) ;
        "Foreground", Awi.Copt Graphics.green ]
      [ "Relief", Awi.Sopt "Top" ; "Border_size", Awi.Iopt 2 ] ;
    gg;;
val create_app : string -> string gg = <fun>
```

Finally, the function `main` takes the name of the file from the command line, constructs a graph with an interface and starts the interaction loop on the main component of the graph interface.

```
# let main () =
    if (Array.length Sys.argv ) <> 2
    then Printf.printf "Usage: dij.exe filename\n"
    else
      let gg = create_app Sys.argv.(1) in
        Awi.loop true false gg.main;;
val main : unit -> unit = <fun>
```

The last expression of that program starts the function `main`.

## *The Executable*

The motivation for making a standalone application is to support its distribution. We collect the types and functions described in this section in the file `dij.ml`. Then we compile the file, adding the different libraries which are used. Here is the command to compile it under Linux.

```
ocamlc -custom -o dij.exe graphics.cma awi.cmo graphs.ml \
    -cclib -lgraphics -cclib -L/usr/X11/lib -cclib -lX11
```

Compiling standalone applications using the `Graphics` library is described in chapters 5 and 7.

## *Final Notes*

The skeleton of this application is sufficiently general to be used in contexts other than the search for traveling paths. Different types of problems can be represented by a weighted graph. For example the search for a path in a labyrinth can be coded in a graph where each intersection is a node. Finding a solution corresponds to computing the shortest path between the start and the goal.

To compare the performance betwen C and Objective Caml, we wrote Dijkstra's algorithm in C. The C program uses the Objective Caml data structures to perform the calculations.

To improve the graphical interface, we add a *`textfield`* for the name of the file and two buttons to load and to store a graph. The user may then modify the positions of the nodes by mouse to improve the appearance.

A second improvement of the graphical interface is the ability to choose the form of the nodes. To display a button, a function tracing a rectangle is called. The display functions can be specialized to use polygons for nodes.