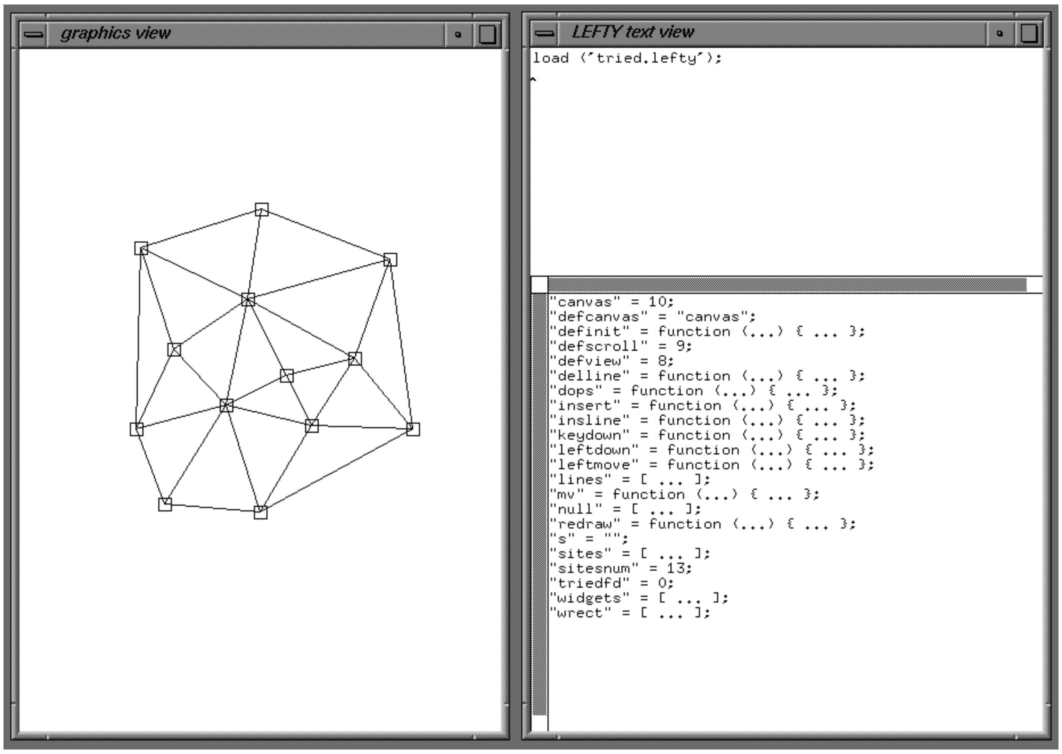
Editing Pictures with *lefty*

Eleftherios Koutsofios

AT&T Bell Laboratories
Murray Hill, NJ

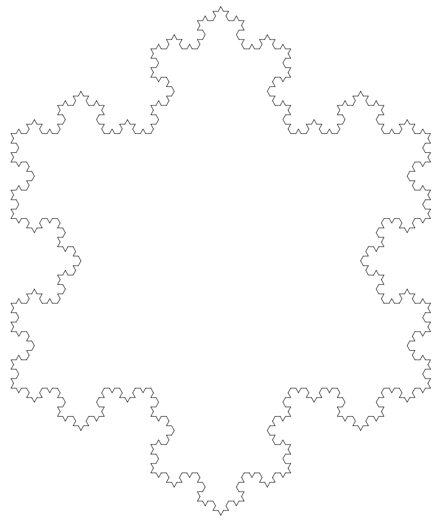
lefty is a two-view graphics editor for technical pictures. This editor has no hardwired knowledge about specific picture layouts or editing operations. Each picture is described by a program that contains functions to draw the picture and functions to perform editing operations that are appropriate for the specific picture. Primitive user actions, like mouse and keyboard events, are also bound to functions in this program. Besides the graphical view of the picture itself, the editor presents a textual view of the program that describes the picture. Programmability and the two-view interface allow the editor to handle a variety of pictures, but are particularly useful for pictures used in technical contexts, e.g., graphs and trees. Also, *lefty* can communicate with other processes. This feature allows it to use existing tools to compute specific picture layouts and allows external processes to use the editor as a front end to display their data structures graphically. The figure below shows a typical snapshot of *lefty* in use. The editor has been programmed to edit delaunay triangulations. The window on the left shows the actual picture. The user can use the mouse to insert or move cites and the triangulation is kept up to date by the editor (which uses an external process to compute the triangulation). The window on the right shows the program view of the picture.



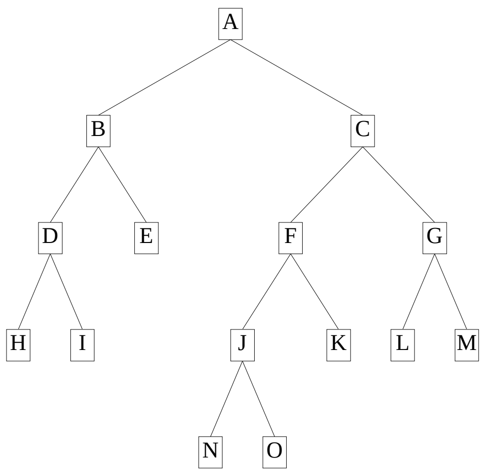
1 Introduction

lefty is an editor designed to handle technical pictures. Technical pictures are pictures used in technical contexts, e.g. program call graphs, binary search trees, fractals, and networks. Although there are many different types of technical pictures, they all share several properties.

One such property is accuracy. Since the reason for drawing a technical picture is to display some abstract object in a way that is easy to understand, it is very important that the positions, sizes, and other graphical attributes of the graphical primitives in a picture follow strict rules. Figure 1 shows two technical pictures. The fractal in Figure 1a consists of equal-size line segments arranged in a path that is computed using a simple recursive function. The binary tree in Figure 1b consists of nodes and edges. All nodes of the same depth are drawn along the same horizontal line and parent nodes are centered over their children nodes.



(a)



(b)

Figure 1: Two technical pictures

Accuracy, however, is just the end result of the more fundamental property *structure*. In Figure 1b, the hierarchy of the tree constrains the graphical representation. `F` and `G` are both children of `C`; if `C` is moved to the right, `F` and `G` must also move to the right to preserve the symmetry. Other parts of the tree also have to move.

Most graphics editors provide ways for drawing pictures accurately, but very few provide ways to maintain the consistency of technical pictures. If the user moves `C` to the right, the editor could move `F` and `G` automatically. Most existing editors, however, do not provide this kind of functionality. *lefty*, on the other hand, was designed with this kind of functionality in mind.

lefty implements a procedural programming language. This language can be used to specify all aspects of picture editing: how to draw the picture, how to edit it, and how to bind user actions to editing operations. Essentially, a picture is treated as an object that contains methods for operating on it. For example, for the tree in Figure 1b this program would include functions to draw nodes and edges, functions to insert nodes and edges, and finally functions to bind mouse actions to editing operations. If the tree is supposed to have some specific semantics then the functions that insert nodes and edges may be modified to perform consistency checks. For example, the function for inserting edges may check if adding an edge would violate the semantics of the picture and if so print an error message. By having a language, *lefty* can be programmed to handle many different kinds of pictures, and at the same time provide in-depth support for each kind of picture.

A picture in *lefty* is shown in two ways. One view is the usual "what you see is what you get view" (WYSIWYG). The other view is a textual view of the program that controls the picture. Users can perform operations on either view. The WYSIWYG view is more intuitive, while the program view is more functional.

Another way in which *lefty* differs from other editors is in the way it can be used. *lefty* can be used as a standalone editor, to prepare pictures for printing, but it can also communicate with other processes. This allows *lefty* to act as a graphical front end for other processes. In this mode a picture becomes a user interface object. For example, as some system changes state, its picture may change, but at the same time the system can be changed by changing the picture. *lefty*'s programmability makes building graphical front ends for other tools easy. In many cases, these front ends perform better than tools that implement all the functionality in a single program.

2 Overview

This section presents an overview of the editor. We use an example that demonstrates how pictures are described and built using *lefty*. This example is too simple to take advantage of the editor's programmability because the picture has no structure, but it demonstrates the basic picture-design principles.

The picture in this example is a collection of rectangles of various sizes positioned randomly. The complete program for this program can be found in Appendix C.

Picture descriptions consist of two parts:

data structures that hold information about the picture. For this example, the data structures tell how many boxes are in the picture and their locations and sizes.

functions that implement operations on the data structures. This example has functions to insert, delete, move, and draw boxes.

Location and size data are stored in `objarray`, which is an array of key-value pairs, and the number of boxes is given by `objnum`. Figure 2 shows a snapshot of the data structures for two boxes. Each element of `objarray` specifies the origin and corner of a box in fields `rect[0]` and `rect[1]` respectively. Section 3.1 describes the *lefty* language in more detail. Figure 3 is the WYSIWYG view.

```
objnum = 2;
objarray = [
    0 = [
        'id' = 0;
        'rect' = [
            0 = [ 'x' = 50; 'y' = 50; ]; 1 = [ 'x' = 200; 'y' = 200; ];
        ];
    ];
    1 = [
        'id' = 1;
        'rect' = [
            0 = [ 'x' = 150; 'y' = 250; ]; 1 = [ 'x' = 350; 'y' = 450; ];
        ];
    ];
];
```

Figure 2: A snapshot of the data structure

We need functions to draw the picture, to change it, and to bind user events to changes to the picture. `drawbox` and `redrawboxes` in Figure 4 do the actual drawing; `drawbox` draws a single box, and `redrawboxes` clears the display and draws all the boxes. `box` does the actual rendering; it is a built-in function that draws a rectangle. Built-in functions provide access to window and operating system resources. They are described in Section 3.5. `canvas` is

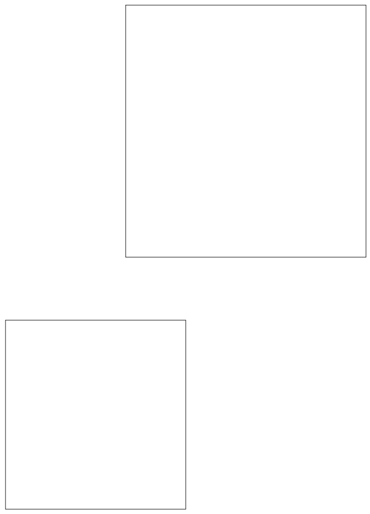


Figure 3: The WYSIWYG view of the picture

```
drawbox = function (obj, color) {
    box (canvas, obj, obj.rect, ['color' = color;]);
};
redrawboxes = function () {
    local i;
    clear (canvas);
    for (i = 0; i < objnum; i = i + 1)
        drawbox (objarray[i], 1);
};
```

Figure 4: Drawing functions

the id for the drawing area. In this example there is only one drawing area and its id is assigned to the global variable `canvas`.

Figure 5 shows various editing functions. `new` adds a new box to `objarray`. `reshape` changes the shape of an existing box, and `move` moves a box. `pointadd` is a function that adds two points and returns the resulting point. `remove` is a built-in function that removes array elements; here it removes entry `objnum - 1` from `objarray`.

Figure 6 shows some of the functions that bind user events to editing operations. `leftdown` is called when the user presses the left mouse button. In this picture, if the user presses the left button over white space (not inside any box) a new—zero size—box is created and the drawing mode is set to `xor`. As the user moves the mouse while holding the left button down, `leftmove` is called. `leftmove` creates a rubberband effect, by reshaping the box so that the corner of the box follows the mouse. When the user releases the

```
new = function (rect) {
    objarray[objnum] = [
        'rect' = rect;
        'id' = objnum;
    ];
    objnum = objnum + 1;
    return objarray[objnum - 1];
};
reshape = function (obj, rect) {
    obj.rect = rect;
    return obj;
};
move = function (obj, p) {
    obj.rect[0] = pointadd (obj.rect[0], p);
    obj.rect[1] = pointadd (obj.rect[1], p);
    return obj;
};
delete = function (obj) {
    if (obj.id ~= objnum - 1) {
        objarray[obj.id] = objarray[objnum - 1];
        objarray[obj.id].id = obj.id;
    }
    remove (objnum - 1, objarray);
    objnum = objnum - 1;
};
```

Figure 5: Editing functions

left button, `leftup` is called. `leftup` sets the drawing mode back to normal (mode `src`). Functions `middledown`, `middlemove`, and `middleup`, allow the user to grab an existing box and move it. Finally, `rightup` deletes a box. The names of these functions are special. When a mouse or keyboard event occurs, *lefty* searches its data structures for a function that corresponds to the event. If such a function is defined, *lefty* calls it with one argument, `data`. `data` is a table that contains information about the user event. `data.pos` is the position of the mouse at the time of the event. `data.ppos` is present only for move or up events and it holds the position of the mouse at the time of the corresponding down event. `data.obj` is the object that the user "selected". In this example, each rectangle on the screen is associated with an entry in `objarray`. This is done by `drawbox` which calls `box` with the corresponding entry in `objarray` as the second argument. When the user presses a mouse button over a rectangle, *lefty* locates that rectangle and from that locates the corresponding object. `setgattr` is a built-in function that changes the graphics state of a drawing area. `clearpick` is a built-in function that makes its object argument unselectable from the WYSIWYG view, by removing it from the data structure used by *lefty* to locate graphical primitives.

The user can edit the picture from either the program or the WYSIWYG view. From

```
leftdown = function (data) {
    if (data.obj ~= null)
        return;
    leftbox = new (rectof (data.pos, data.pos));
    drawbox (leftbox, 1);
    setgfxattr (canvas, ['mode' = 'xor';]);
};
leftmove = function (data) {
    if (~leftbox)
        return;
    drawbox (leftbox, 1);
    clearpick (canvas, leftbox);
    reshape (leftbox, rectof (data.ppos, data.pos));
    drawbox (leftbox, 1);
};
leftup = function (data) {
    if (~leftbox)
        return;
    drawbox (leftbox, 1);
    clearpick (canvas, leftbox);
    reshape (leftbox, rectof (data.ppos, data.pos));
    setgfxattr (canvas, ['mode' = 'src';]);
    drawbox (leftbox, 1);
    remove ('leftbox');
};
```

Figure 6: User Interface functions

the WYSIWYG view, the user can create a new box by pressing the left button, then—while holding the button down—moving the mouse to another location and releasing the button. From the program view, the user can perform similar operations by entering expressions in the editor's language. For example, the user can create a new box by typing the following commands:

```
newbox = new ([0 = ['x' = 250; 'y' = 120;]; 1 = ['x' = 390; 'y' = 240;];]);
drawbox (newbox, 1);
```

The program view shows the data structures and functions of the current program. By default, entries in this view are shown abstracted, with each entry taking up a single line. The user can selectively expand individual entries to their full size.

Figure 8 shows the WYSIWYG view after moving one of the boxes and Figure 7 shows the corresponding data structures.

Generating postscript output is easy. `canvas` can be set to an id corresponding to a file. In this case, built-ins such as `box`, will append the appropriate postscript expressions

```
objnum = 3;
objarray = [
    0 = [
        'id' = 0;
        'rect' = [
            0 = [ 'x' = 50; 'y' = 50; ]; 1 = [ 'x' = 200; 'y' = 200; ];
        ];
    ];
    1 = [
        'id' = 1;
        'rect' = [
            0 = [ 'x' = 150; 'y' = 250; ]; 1 = [ 'x' = 350; 'y' = 450; ];
        ];
    ];
    2 = [
        'id' = 2;
        'rect' = [
            0 = [ 'x' = 250; 'y' = 120; ]; 1 = [ 'x' = 390; 'y' = 240; ];
        ];
    ];
];
```

Figure 7: A snapshot of the data structure after editing the picture

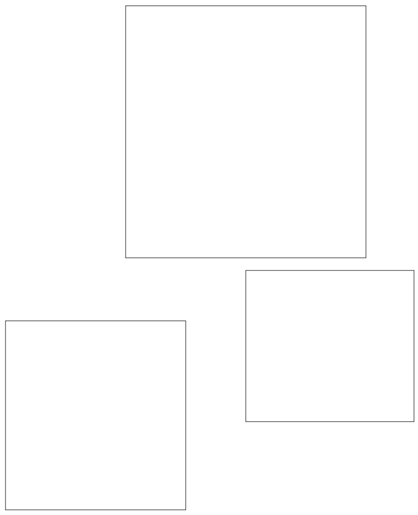


Figure 8: The WYSIWYG view of the picture after editing

to this file. Function `dops`, shown in Figure 9 creates a postscript file, sets `canvas` to the id of the file, draws the picture, closes the file and restores `canvas` to its original value.

```
dops = function () {
    local s;
    s = ['x' = 8 * 300; 'y' = 10.5 * 300;];
    canvas = createwidget (-1, ['type' = 'ps'; 'size' = s;]);
    setwidgetattr (canvas, ['window' = wrect;]);
    redraw (canvas);
    destroywidget (canvas);
    canvas = defcanvas;
};
```

Figure 9: Function for generating postscript output

3 System Components

This section presents the components of the editor.

3.1 The Language

Since *lefty* is interactive, the language is designed to allow for fast parsing and execution. The language was inspired by *EZ* [FH85]. Appendix B specifies the language in detail.

The language supports *scalars* and *tables*. A scalar is a number or a character string of arbitrary length. A table is a one-dimensional array indexed by numbers or strings.

For example, `objarray` in Figure 2 is a two-entry table indexed by `0` and `1`. Each of these entries is a table with entries for the center and the size of each box and an `id` for each box.

Variables are either global, i.e., part of a global name table, or local to a function. Expressions may or may not return a value. For example, `a + b` does not return a value when either `a` or `b` are not defined.

The smallest program unit is the expression. User actions on the WYSIWYG view result in the execution of expressions. User-typed text in the program view is a sequence of expressions. Each user action results in the immediate evaluation of an expression. For example, if the user enters `num = sqrt (4);` in the program view, `sqrt` is called and its return value, 2, is assigned to `num`. Once executed, the input is discarded; the only change in the program's state is that it now contains `num`. To specify code that is meant to be executed later, the user must define a function, e.g.,

```
afunction = function (n) {
    num = sqrt (n);
};
```

"Executing" a function declaration adds the name of the function to the global name table. Calling `afunction` assigns a value to `num`, e.g., `afunction (4);` assigns `2` to `num`.

Assignment is done either by value (scalars), or by reference (tables). For example, after the sequence `a = 1; b = a;`, `a` and `b` point to two different values, while the sequence `a = [ ]; b = a;` results in both `a` and `b` pointing to the same table. Functions are stored and treated like scalars.

3.2 The Program View

The program view is a textual representation of the picture state. It displays the name and value of each global object.

The textual representation can be long, so the editor presents an abbreviated view by default: each name, value pair is displayed on a line. Figure 10a shows a few of the entries in the program view for the picture in Figure 3. Only the value for `objnum` is displayed, as it can fit in a single line. Other variables have an abstract representation, which indicates whether they are functions or tables.

```
'delete' = function (...) { ... };
'drawbox' = function (...) { ... };
'leftdown' = function (...) { ... };
'leftmove' = function (...) { ... };
'leftup' = function (...) { ... };
'move' = function (...) { ... };
'new' = function (...) { ... };
'objarray' = [ ... ];
'objnum' = 2;
'redraw' = function (...) { ... };
'reshape' = function (...) { ... };
```

(a) All entries closed

```
'delete' = function (...) { ... };
'drawbox' = function (...) { ... };
'leftdown' = function (...) { ... };
'leftmove' = function (...) { ... };
'leftup' = function (...) { ... };
'move' = function (...) { ... };
'new' = function (...) { ... };
'objarray' = [
    0 = [ ... ];
    1 = [ ... ];
];
'objnum' = 2;
'redraw' = function (...) { ... };
'reshape' = function (...) { ... };
```

(b) Opening entry `objarray`

```
'delete' = function (...) { ... };
'drawbox' = function (...) { ... };
'leftdown' = function (...) { ... };
'leftmove' = function (...) { ... };
'leftup' = function (...) { ... };
'move' = function (...) { ... };
'new' = function (...) { ... };
'objarray' = [
    0 = [
        'id' = 0;
        'rect' = [ ... ];
    ];
    1 = [ ... ];
];
'objnum' = 2;
'redraw' = function (...) { ... };
'reshape' = function (...) { ... };
```

(c) Opening entry `objarray[0]`

```
'delete' = function (...) { ... };
'drawbox' = function (...) { ... };
'leftdown' = function (...) { ... };
'leftmove' = function (...) { ... };
'leftup' = function (...) { ... };
'move' = function (...) { ... };
'new' = function (...) { ... };
'objarray' = [
    0 = [
        'id' = 0;
        'rect' = [ ... ];
    ];
    1 = [ ... ];
];
'objnum' = 2;
'redraw' = function (...) { ... };
'reshape' = function (obj, rect) {
    obj.rect = rect;
    return obj;
};
'zarray' = objarray;
```

(d) Opening entry `reshape`

Figure 10: Various levels of abstraction on the program view

For a more detailed view of an object, the user clicks on the line describing the object.

For example, clicking on the line for `objarray` causes the editor to expand it, as shown in Figure 10b, to show that `objarray` has two entries indexed by `0` and `1`. Clicking on the `0` entry of `objarray` causes the editor to expand that entry as shown in Figure 10c. Entry `1` remains the same, but entry `0` is expanded. Function entries behave similarly: clicking on a function displays the function's body. Clicking on `reshape`, for example, results in the display shown in Figure 10d.

Clicking on an expanded entry replaces that entry with its abstracted version.

If an entry points to the same value as another entry, the second entry is shown differently. Rather than showing the same value twice, the editor shows the duplication. For example, if we execute `zarray=objarray;`, `zarray` will be shown as in Figure 10d. This display semantic makes it clear how much unique information is available.

Unlike in the WYSIWYG view, where changes are controlled by the program that describes the picture, the user can do anything in the program view, including getting the program into an inconsistent state. All the functions and tables are visible and can be edited. This flexibility is necessary, since a conceptual change to the program or the data usually requires a sequence of modifications to the text of the program. Although the sequence of modifications leaves the editor in a consistent state, individual modifications can put the editor in an inconsistent state temporarily. For example, the user can add a box to Figure 3 by typing in the sequence of commands executed by function `new` in Figure 5. After the user has typed in the assignment for `objarray[objnum]`, the program is inconsistent: `objarray` has three entries, but the value for `objnum` is still `2`. The program becomes consistent after the user types in the command to increment `objnum`.

3.3 The WYSIWYG View

The WYSIWYG view is the graphical representation of the picture. The program that describes a picture controls the WYSIWYG view; all the objects are drawn by the program, and all user actions are handled by the program. The WYSIWYG view can consist of one or more widgets, such as drawing areas, buttons, lists, text areas, and scrollable widgets.

Widgets can be manipulated using built-in functions. When a new widget is created, *lefty* adds an entry to a global table called `widgets`. Each of these entries is a table that can be used to customize the behavior of the widget. When the user generates an event, e.g. clicks a mouse button, *lefty* searches the corresponding entry in `widgets` for the appropriate callback function. If the function cannot be found in that table, *lefty* then searches for it in the global namespace.

The most interesting type of widget in *lefty* is the drawing area. Drawing inside such a widget is handled by a set of built-in functions. The supported graphical primitives are lines, polygons, splinegons, elliptic arcs, and text. Each drawing area maintains its own graphics state. The built-in functions and state variables are described in Section 3.5

When an event occurs inside a drawing area, for example, a mouse button is pressed or released, the editor checks if a function corresponding to this event exists. The possibilities are:

`leftdown`	`leftmove`	`leftup`
`middledown`	`middlemove`	`middleup`
`rightdown`	`rightmove`	`rightup`
`keydown`		`keyup`

There is no restriction on what these functions do. The programmer must define them as appropriate for the current picture. *lefty* searches for these functions first in the drawing area's entry in `widgets`, then in the global namespace. If a function is found, it is called with a single argument. This argument is a table that contains information about the event. It has the following fields.

`obj`	the object that the user selected with this event, or `null` if no such object could be found
`pos`	a table with two entries, `x` and `y`, that hold the mouse coordinates at the time of this event
`pobj`	(only for `move` and `up` events) the object selected by the preceding `down` event.
`ppos`	(only for `move` and `up` events) the mouse coordinates at the time of the preceding `down` event
`widget`	the widget id of the drawing area where this event occurred
`key`	(only for `key` events) the ascii character of the key

Determining the selected object at a button press or release has two phases. The editor determines if the mouse coordinates select a graphical primitive. Closed shapes, for example, boxes and ellipses, are selected if the mouse coordinates lie inside the shape. If such a primitive can be found, the editor finds the *lefty* data object associated with it.

Finding the selected graphical primitive is straightforward. The editor maintains a data structure of all the graphical primitives in the WYSIWYG view. When an event is received, the coordinates are used to search through this data structure for the selected primitive. The only complication is when two or more primitives overlap. In the box example in Section 2, boxes could overlap. The editor does not resolve these kinds of ambiguities. One solution would be to allow the user to rotate through all the objects that could potentially be selected. The editor does provide a way to resolve ambiguities created by design, such as when an object is drawn using more than one graphical primitives. In the tree figure in Section 1, a tree node is drawn as a rectangle enclosing a label.

Finding the data object that corresponds to the selected primitive is slightly more complex. The data object must be specified as an argument to the rendering primitive. All rendering functions take as their second argument the object to associate with the primitive they draw. This argument can also be `null`, which effectively makes the primitive unselectable. For the tree example, the text label of a node could be associated with `null` and that would leave the node's box as the only selectable primitive occupying that area of the display. The box would have to be associated with the table that represents the corresponding node of the tree. The mapping between objects and graphical primitives is manipulated with two functions:

`clearpick` (*canvas*, *object*)
`setpick` (*canvas*, *object*, *rectangle*)

`clearpick` removes *object* from the mapping, and `setpick` associates the rectangular area *rectangle* with *object*. Finally, clearing the WYSIWYG view clears the mapping.

When a drawing area is resized, or when its window is redrawn, *lefty* searches for a function called `redraw`. If such a function is found in `widgets` or in the global namespace, it is called with a table as an argument. This table contains an entry `widget` which is the widget id of the drawing area.

The label widget can display a piece of text in its rectangular area. It provides a subset of the user-interface functions provided by the drawing area; it provides all the `up` and `down` functions mentioned above, but does not provide the `move` functions. The table passed as argument to these functions contains the `widget` entry and—for the `keyup` and `keydown` functions—the `key` entry.

The button widget provides a single function, `pressed`. It is called when the user clicks on the button. If such a function is found it is called with just one entry, `widget`.

When the user presses `CR` in an input text widget, *lefty* looks for a function called `oneline`. This function, if found, is called with two arguments, `widget` and `text`. `text` contains the line of text that the user just entered.

For array widgets, *lefty* tries to call a function named `resize` whenever their size changes (either through user actions, or program control). This function is called with two arguments, `widget` and `size`. `size` is an (x,y) table containing the new size of the widget. This function is expected to return an array containing new sizes for all of the widget's children. This array must be indexed by the widget ids of the children, and each element must be an (x,y) table containing the size of a child.

lefty can monitor open file descriptors. Built-in function `monitor` takes as an argument the id of an input channel (generated by `openio`), and adds it to the list of file descriptors being monitored. When there is something to read from that file descriptor, *lefty* searches the global namespace for a function called `monitorfile`. If such a function is found, it is called with a table as an argument. This table contains an entry `fd` which is the file descriptor that is ready for reading.

Finally, when there are no X or file I/O events to handle, *lefty* can optionally execute a function called `idle`. This feature can be turned on or off using the `idlerun` built-in.

3.4 Inter-process Communication

lefty provides built-ins for communicating with other processes. This capability can be used in several ways.

Purely for output. A process can use the editor to display some data structures; the process does not need any code for graphical layout.

For both input and output. The editor can be used to specify the input and to display the result of some processing of that input. Debugging is an example; instead of printing data structures as text or writing code to draw them, the process being debugged simply connects to the editing server and sends the data structures to the server for display.

As an extension to the editor itself. There are tools displaying trees [TdBB88], DAGs [GNV88], delaunay triangulations [GS85], and VLSI layouts [Uni85]. These tools are usually large software packages, and duplicating their functionality in the editor is a major undertaking. Instead, the editor communicates with these tools as separate processes. Whenever some aspect of a layout needs to be updated, the editor sends a message asking for instructions on how to perform the update to the appropriate process.

lefty communicates with other processes by exchanging ASCII strings. This allows *lefty* to communicate with many existing tools, without having to modify these tools at all. For example, the layout for the tree in Figure 1b is generated by a call to `complayout`. This function does all the calculations. If the layout were to be generated by a separate process this function could be rewritten as follows.

```
complayout = function () {
    ...
    writeline (treefd, 'compute layout');
    while ((s = readline (treefd)) ~= '') {
        t = split (s, ' ');
        nodearray[ston (t[0])] = ['x' = ston (t[0]); 'y' = ston (t[0]);];
    }
    ...
};
```

`complayout` sends a message to the external process, using `writeline`, requesting a new layout. The `while` loop reads back the response from the process. Each line would consist of 3 numbers: the id of a node and its x and y coordinates. The process must send an empty line at the end of the transmission. `treefd` is the file descriptor for communicating with the other process.

The response from a process can also be a *lefty* expression. The `while` loop above could be replaced with the following loop.

```
complayout = function () {
    ...
    while ((s = readline (treefd)) ~= '')
        run (s);
    ...
};
```

`run` is a built-in. It parses and execute the *lefty* expression specified by the string `s`. A sample string could be `nodearray[i].p = ['x' = 10; 'y' = 20;];`.

This form of remote procedure call gives processes access to *lefty* functions and data structures and should help minimize the amount of work needed to interface a process with *lefty*.

The technique of communicating by sending programs has been used in several other systems, most notably in window systems [PLR85, SUN88].

3.5 Built-in Functions

3.5.1 Widget Functions

widgetid = `createwidget` (*parentid, attr*)
`setwidgetattr` (*widgetid, attr*)
attr = `getwidgetattr` (*widgetid, keys*)
`destroywidget` (*widgetid*)

These functions are used to create, modify, and destroy widgets. `createwidget` creates a new widget and returns its id. This id is a small integer. `createwidget` creates a new table, indexed by *widgetid*, under the `widgets` global table. *parentid* is the id of the parent widget. *attr* is a table of attributes, such as type, size, name, etc. Attribute `type` must be specified, but if some other attributes are not set, default values are used instead. `setwidgetattr` sets one or more attributes for the specified widget (except for `type`). `getwidgetattr` returns the current values of the attributes specified by *keys*. *keys* is an indexed array of attribute names. For example, if *keys* is set to `[0 = 'name'; 1 = 'size';]`, the returned *attr* table will contain two entries, `['name' = ...; 'size' = ...;]`. `destroywidget` destroys the specified widget and any children that it might have.

Tables 1 and 2 show the available widgets.

3.5.2 Graphics Functions

`clear` (*canvasid*)
`clearpick` (*canvasid, object*)
`setpick` (*canvasid, object, rect*)

`clear` clears the drawing area whose id is *canvasid* and the table that contains the mapping between data objects and graphical objects. `clearpick` removes *object* from the mapping table, and `setpick` maps the rectangular area specified by *rect* to *object*.

item = `displaymenu` (*widgetid, menu*)

`displaymenu` pops up the menu specified by *menu* inside the widget specified by *widgetid* (which must be either a canvas or a label widget). *menu* must be a table of number-string pairs. When the user selects one of the string entries `displaymenu` returns the number associated with that string. If the user dismisses the menu, `-1` is returned.

Type	Attributes	Attr. type	Description

`view`	`origin`
	`size`
	`name`
	`zorder`
		table of (x, y)
		table of (x, y)
		string
		string
			A top level window. It may contain exactly one child. `zorder` can be used to push / pop the view (values `"top"`, `"bottom"`).

`text`	`size`
	`borderwidth`
	`text`
	`mode`
	`appendtext`
		table of (x, y)
		integer
		string
		string
		string
			A widget that can display (and optionally edit) text. `mode` can be one of `"oneline"`, `"input"`, or `"output"`. For mode `line`, *lefty* tries to execute the `func` callback whenever `CR` is pressed. `appendtext` appends a string to the string already displayed by the widget.

`scroll`	`size`
	`borderwidth`
	`childcenter`
	`mode`
		table of (x, y)
		integer
		table of (x, y)
		string
			A widget that can contain a—potentially larger—child widget and let the user scroll through it. `childcenter` may not be specified until the scroll widget has a child widget. `childcenter` aligns the child so that the child's `childcenter` coordinates are at the center of the scroll widget. `mode` can be set to `"forcebars"` to make scrollbars appear even when the child widget is small enough to fit inside the scroll widget.

`array`	`size`
	`borderwidth`
	`mode`
	`layout`
		table of (x, y)
		integer
		string
		string
			A widget that can take a list of children widgets and display them either as a horizontal or a vertical list. `mode` can be one of `"horizontal"`, or `"vertical"`. `layout` controls whether the widget rearranges its children every time there is some change. If set to `"off"` the widget will stop rearranging its children until `layout` is set to `"on"` again.

Table 1: Widget types part 1

reply `= ask (`*prompt*`)`

Prompts the user for information; it displays the *prompt* string in a dialog box and waits for the user to type a reply, which is returned as the value of `ask`.

`setgfxattr (`*canvasid, attr*`)`
attr `= getgfxattr (`*canvasid, keys*`)`

`setgfxattr` sets attributes in the graphics state. Each drawing area has its own state variables. `setgfxattr` sets these attributes permanently. These attributes can also be set on a per rendering call basis. `getgfxattr` returns the current values of the attributes specified by keys. *keys* is an indexed array of attribute names. For example, if *keys* is set to

Type	Attributes	Attr. type	Description

`button`	`size`
	`borderwidth`
	`text`
		table of (x, y)
		integer
		string
			A widget that can display a text label and execute the callback `pressed` when it is selected.

`canvas`	`size`
	`borderwidth`
	`cursor`
	`color`
	
	`viewport`
	`window`
		table of (x, y)
		integer
		string
		array of (r, g, b) and strings
		table of (x, y)
		2 tables of (x, y)
			A drawing area. `cursor` must be the name of a cursor bitmap, e.g. `"watch"` or `"default"`. `color` is an array of RGB values and color names. Colors `0` and `1` are predefined to be the background and foreground colors. `viewport` sets the size in pixels of the drawing area. `window` sets the mapping between drawing coordinates and pixel coordinates. The default value for `window` is (0,0) - (1,1). The origin is at the lower left side.

`label`	`size`
	`borderwidth`
	`text`
		table of (x, y)
		integer
		string
			A widget that can display a text label and execute several callbacks depending on the mouse or keyboard buttons used.

`ps`	`origin`
	`size`
	`name`
	`mode`
	`color`
	`window`
		table of (x, y)
		table of (x, y)
		string
		string
		array of (r, g, b)
		2 tables of (x, y)
			A postscript file. `name` is the file name. `mode` can be `"landscape"`.

Table 2: Widget types part 2

`[0 = 'mode'; 1 = 'width';]`, the returned *attr* table will contain two entries, `['mode' = ...; 'width' = ...;]`.

The graphics state consists of the variables shown in Table 3.

Name	Type	Range	Default	Description
`color`	integer	`0-255`	`1`	The current drawing color.
`width`	integer	`>= 0`	`0`	The current line width.
`mode`	string	`'src'` `'xor'`	`'src'`	The current drawing mode.
`fill`	string	`'on'` `'off`	`'off'`	Whether polygons and arcs should be drawn filled or outlined.

`style`	string	`'solid'`
		`'dashed'`
		`'dotted'`
			`'solid'`	The current line style.

Table 3: Graphics state

`arrow` (*canvasid, object, p1, p2 [, attr]*)
`line` (*canvasid, object, p1, p2 [, attr]*)

`box` (*canvasid, object, rect [, attr]*)
`polygon` (*canvasid, object, pointarray [, attr]*)
`splinegon` (*canvasid, object, pointarray [, attr]*)
`arc` (*canvasid, object, center, size [, attr]*)
`text` (*canvasid, object, pos, string, fontname, fontsize, just [,attr]*)
size = `textsize` (*canvasid, object, fontname, fontsize*)

The final argument in most of these functions can be used to change the graphics state for the execution of that function. `splinegon` draws a piecewise bezier spline curve. `fontname` must be an X font name or a postscript font name. For ISO style font names, if the name contains the sequence `%d`, this sequence will be replaced by the appropriate font size. `just` is a two letter string that controls the justification of the string. The first letter may be `l`, `c`, or `r` for left, center, or right justified strings. The second letter specifies the vertical justification and can be one of `u`, `c`, `d`.

3.5.3 Input / Output functions

id = `openio` (*type, name, mode [, format]*)
`closeio` (*id [, flag]*)
string = `readline` (*id*)
string = `read` (*id*)
`writeline` (*id*)
table = `readgraph` (*id*)
table = `parsegraphlabel` (*label, rects*)

These functions handle input and output for a variety of connections, such as regular files, pipes, and sockets. *type* is a string whose value can be one of `'file'`, `'pipe'`, `'socket'`, or `'cs'`. For regular files, file *name* is opened for reading or writing, depending on *mode*. *mode* can be one of `'r'`, `'w'`, or `'w+'`. For pipes and sockets, *name* is the name of an executable. If the name does not begin with `/`, or `.`, the executable is searched, first in the path defined by the environment variable `LEFTYPATH`, then in `PATH`. Finally, if *format* is specified, it customizes the way an executable is invoked. In *format*, a `%` followed by a letter specifies a formatting directive. The following directives are currently recognized.

- `%e` the full path name for the executable.
- `%i` the input file descriptor (for pipes).
- `%o` the output file descriptor (for pipes).
- `%h` the hostname (for sockets).
- `%p` the port number (for sockets).

An arbitrary shell command can be executed by calling `openio` with *name* set to `"ksh"` (or any other shell) and *format* set to `concat ('%e -c "', ` *cmd*`, '"')`, where *cmd* is the shell command. For sockets, *lefty* creates an internet socket, starts up the executable, then waits for the executable to connect to that socket. `'cs'` can be used to establish a *libcs*-style connection. `name` in this case is the *libcs* name for a service. The optional `flag` parameter can be set to `"kill"` to make *lefty* send the kill signal to the child process (if such exists) after it has closed the file descriptor. `readline` reads a full line and returns it (stripping the newline character). `readgraph` reads a graph in *dot*'s language and returns it as a table. `parsegraphlabel` takes a *dot*-style record label and the corresponding string of coordinates for the record fields and returns a hierachical table. Each entry in this table contains either the text and coordinates of a field, or a sub-table of fields.

3.5.4 Math Functions

value = `atan` (*y, x*)
value = `cos` (*angle*)
value = `sin` (*angle*)
value = `sqrt` (*number*)
value = `random` (*number*)
integer = `toint` (*number*)

`angle` is assumed to be in degrees. `toint` truncates the decimal part of *number*.

3.5.5 Miscellaneous Functions

`dump` (*...*)
`echo` (*arg1, ...*)

`echo` prints out each arguments by appending them one after the other in the same line. `echo` does not handle tables or functions. `dump` prints its arguments separated by newlines. It can handle any type of *lefty* object. If `dump` is called with no arguments, it prints the entire namespace.

object2 = `copy` (*object1*)
`remove` (*key [, table]*)
size = `tablesize` (*table*)
size = `strlen` (*string*)
table = `split` (*string, delimiter*)
string = `concat` (*arg1 [, ...]*)
string = `ntos` (*number*)
number = `ston` (*string*)

`copy` makes a complete copy of *object1*. This is useful for assigning tables by value. `remove` removes *key*, either from *table*, or from the global namespace. `tablesize` returns the number of entries in a table. `strlen` returns the number of characters in *string*. `split` splits *string* in words and returns a table (indexed from 0 and up), where each entry is a word. *delimiter* is a one character string that is used to break *string* into words. Each occurance of *delimiter* separates two words. The only exception is when *delimiter* is the space character; all leading and trailing spaces are ignored and multiple spaces are treated as a single space. `concat` concatenates all its arguments into one string. `ntos` converts a number to a string. `ston` converts a string to a number.

`load` (*string*)
`run` (*string*)
`exit` ()

`load` parses and executes *lefty* statements from the file specified by *string*. If the file does not start with `/` or `.`, it is searched in the path specified by the environment variable `LEFTYPATH`. `run` parses and executes the *lefty* statements in *string*. `exit` quits the editor.

`txtview` (*mode*)

`txtview` turns the program view on or off. *mode* can be one of `'on'` or `'off'`.

`monitor` (*fileid, mode*)

`monitor` turns on or off the monitoring of a file for input. *mode* can be one of `'on'` or `'off'`. *fileid* is an id returned from `openio`. When a file descriptor becomes ready for reading, *lefty* calls the `monitorfile` callback.

`idlerun` (*mode*)

`idlerun` can be used to control what *lefty* does when there are no events to handle. *mode* can be one of `'on'` or `'off'`. The default mode if `'off'`. Setting `mode` to `'on'`, instructs *lefty* to keep running the `idle` callback unless there are X or file events to handle. Setting `mode` to `'off'`, instructs *lefty* to just block waiting for events to handle.

`sleep` (*seconds*)

`sleep` pauses execution for *seconds* seconds.

`system` (*string*)

`system` executes *string* as a shell command. It waits until the command finishes.

4 Examples

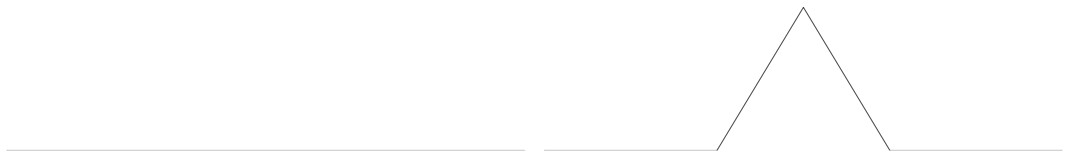
4.1 Fractals

This is an example of a type of figure easily described in a procedural language. Fractals are usually created by starting from a basic figure and recursively replacing parts of it with more complex constructs.

In this example, the basic figure is the equilateral triangle; `drawfractal` "draws" the three sides of the triangle:

```
drawfractal = function () {
    ...
    fractal (0, length, fractalangle +  60);
    fractal (0, length, fractalangle -  60);
    fractal (0, length, fractalangle - 180);
    ...
};
```

The replacement rule is to replace each line segment with four:



original replacement

`fractal` does the recursive replacement:

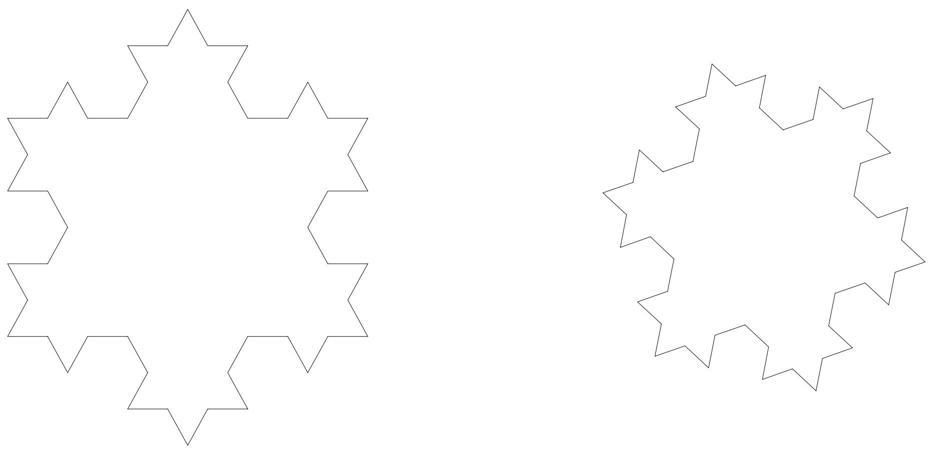
```
fractal = function (level, length, angle) {
    local nlength, newpenpos;

    if (level >= maxlevel) {
        newpenpos.x = penpos.x + length * cos (angle);
        newpenpos.y = penpos.y + length * sin (angle);
        line (canvas, null, penpos, newpenpos, ['color' = 1;]);
        penpos = newpenpos;
        return;
    }
    nlength = length / 3;
    fractal (level + 1, nlength, angle);
    fractal (level + 1, nlength, angle + 60);
    fractal (level + 1, nlength, angle - 60);
    fractal (level + 1, nlength, angle);
};
```

Recursion is controlled by `level`. If `level` exceeds `maxlevel`, `fractal` returns, otherwise it makes the four recursive calls to itself. The fractal in Figure 1a was drawn with `maxlevel` set to `4`.

The picture is drawn using the concept of the *pen*. Drawing is done relative to `pen`, which holds the current pen coordinates, and `pen` is updated after each line is drawn.

`transformfractal` changes the size and orientation of the fractal. It takes two arguments, `prevpoint` and `currpoint`, and rotates and scales the fractal, relative to its center, so that `prevpoint` is mapped to `currpoint`. `transformfractal` is called from `leftup`. `prevpoint` is set to the mouse coordinates during the down event, while `currpoint` is set to the coordinates during the up event. Neither `prevpoint` nor `currpoint` need lie on the fractal outline; rather than making the vertices or edges of the fractal selectable, `setpick` is used to make the entire view a single selectable object. Figure 11 shows how the picture changes as the user moves the cursor.



(a) (b)

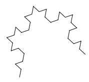
Figure 11: Moving the cursor scales and rotates the picture

Figure 12 shows a trace of the functions executed when the user transforms the fractal in Figure 11a to the one in Figure 11b. For brevity, the trace does not show the lowest level of recursion for `fractal`. Each level one `fractal` call makes four calls to itself, each of which makes a call to `line`. The pictures to the right of the trace depict the state of the WYSIWYG view as the program executes.

4.2 Trees

The program in this example draws trees of arbitrary degree. Each node of the tree is represented by an entry in `nodearray`. Each entry contains a string label and information about the children of the node. `inode` and `iedge` insert new nodes and edges; `inode` is called from `leftdown`, while `iedge` is called from `middleup`. To insert an edge, the user

```
leftup (['ppos' = ['x' = 200; 'y' = 50;];
        'pos' = ['x' = 250; 'y' = 100;];])
    transformfractal (['x' = 200; 'y' = 50;],
            ['x' = 250; 'y' = 100;])
    drawfractal ()
        fractal (0, 237.170825, 78.434949)
            fractal (1, 79.056942, 78.434949)
            fractal (1, 79.056942, 138.434949)
            fractal (1, 79.056942, 18.434949)
            fractal (1, 79.056942, 78.434949)
        fractal (0, 237.170825, -41.565051)
            fractal (1, 79.056942, -41.565051)
            fractal (1, 79.056942, 18.434949)
            fractal (1, 79.056942, -101.565051)
            fractal (1, 79.056942, -41.565051)
        fractal (0, 237.170825, -161.565051)
            fractal (1, 79.056942, -161.565051)
            fractal (1, 79.056942, -101.565051)
            fractal (1, 79.056942, -221.565051)
            fractal (1, 79.056942, -161.565051)
```



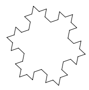


Figure 12: A trace of the execution sequence that draws a fractal

pressed the middle button over the parent node, moves the mouse over the child node and releases the button. `iedge` performs the following steps.

```
iedge = function (node1, node2) {
    node1.ch[node1.chn] = node2;
    node1.chn = node1.chn + 1;
    node2.depth = node1.depth + 1;
    complayout ();
    clear (canvas);
    drawtree (tree);
};
```

Whenever an edge is inserted, the program recomputes the tree layout. This is done by `complayout`. The layout algorithm assigns distinct `x`-coordinates to each leaf node, and positions each intermediate node midway between its leftmost and rightmost children. `drawnode` and `drawedge` draw the nodes and edges of the tree. The program contains two functions, `boxnode` and `circlenode`, which draw a node either as a box or a circle. Which one is used is controlled by assigning the appropriate function as a value to `drawnode`. `changenode` can be used to switch between the two styles, i.e., typing

```
changenode (boxnode);
```

sets `drawnode` to `boxnode`, clears the display, and draws the tree using box-style nodes. Switching between styles could also be done from the WYSIWYG view, using a menu. Adding such a menu facility would require the following additions to the program.

```
menu = [
    0 = 'box';
    1 = 'circle';
];
rightdown = function (data) {
    local i;
    if ((i = displaymenu (canvas, menu)) == 0)
        changenode (boxnode);
    else if (i == 1)
        changenode (circlenode);
};
```

Another modification to the tree program would be to allows it to draw binary search trees. `dolayout` is modified to position each intermediate node so that it lies to the right of all the nodes in its left subtree and to the left of all the nodes in its right subtree. Figures 13 and 14 show two such trees; these were copied from Figures 17.5 and 14.11 in Reference [Sed88].

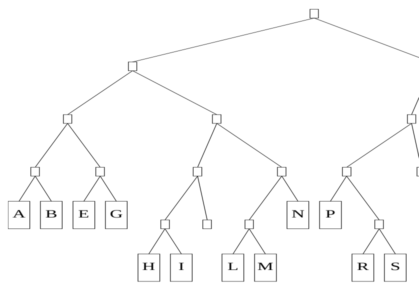


Figure 13: A radix search tree

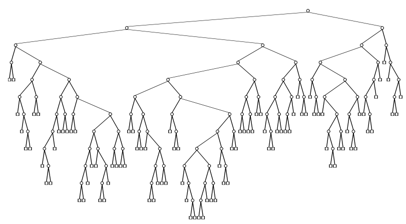


Figure 14: A larger binary search tree

4.3 Delaunay Triangulations

In this example, an external process is used to maintain the delaunay triangulation of a set of sites. The user can insert new sites or move existing sites to new positions. `insert` inserts a site at position `p`:

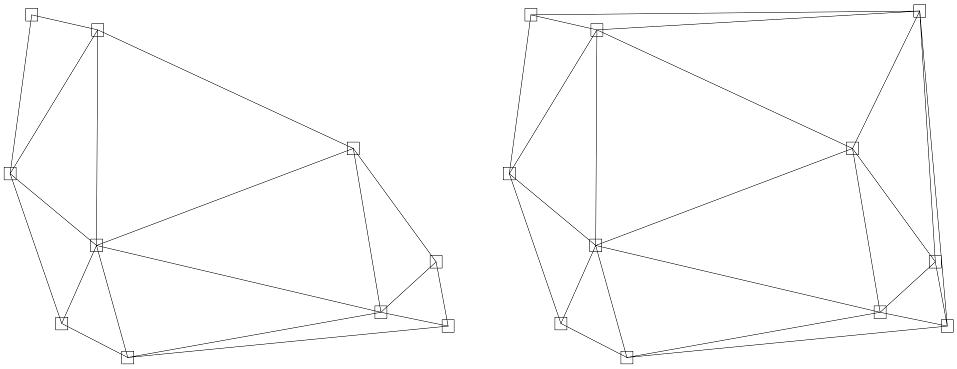
```
insert = function (point) {
    local s;
    sites[sitesnum].num = sitesnum;
    sites[sitesnum].point = point;
    s = concat ('new ', sitesnum, ' ', point.x, ' ', point.y);
    writeline (triedfd, s);
    sitesnum = sitesnum + 1;
    while ((s = readline (triedfd)) ~= '')
        run (s);
    box (canvas, sites[sitesnum - 1],
            [0 = ['x' = point.x - 5; 'y' = point.y - 5;];
             1 = ['x' = point.x + 5; 'y' = point.y + 5;];
            ]);
};
```

`insert` updates the editor's data structures, sends a message to the process indicating that a new site was inserted, processes its response, and draws the new site as a box. The process responds with a sequence of calls to `insline` and `delline`, which insert or delete an edge between two sites:

```
insline = function (i, j) {
    lines[i][j].f = sites[i];
    lines[i][j].l = sites[j];
    line (canvas, null, ['x' = sites[i].point.x; 'y' = sites[i].point.y;],
            ['x' = sites[j].point.x; 'y' = sites[j].point.y;]);
};
delline = function (i, j) {
    remove (j, lines[i]);
    if (tablesize (lines[i]) == 0)
        remove (i, lines);
    line (canvas, null, ['x' = sites[i].point.x; 'y' = sites[i].point.y;],
            ['x' = sites[j].point.x; 'y' = sites[j].point.y;],
            ['color' = 0;]);
};
```

Because the picture is a triangulation, i.e., there are no line intersections, the view can be updated incrementally by drawing and erasing lines. For example, `delline` removes an edge from the screen by drawing it in the background color. Incremental techniques are also used to compute how the triangulation itself changes when a new site is inserted. For deletion, however, the incremental techniques are not significantly faster than recomputing the complete triangulation.

Figure 15 shows a sample triangulation and how it changes when a new site is added to the upper right corner.



(a) (b)

Figure 15: Adding a new site to the triangulation in (a) produces (b)

4.4 Directed Acyclic Graphs

In this example, *lefty* is programmed to handle graphs. It can control several windows with a different graph displayed in each one. *lefty* uses *dot* [GKNV93] as a layout server for the graphs. The resulting tool is called *dotty*. *dotty* allows the user to read in graphs in *dot*'s language and to edit them in various ways.

The *lefty* program consists of two parts. One part handles the interface to *dot*, while the other handles the user interface and the drawing of the graphs. For example, there are two functions that are called when a user tries to insert a new node, `dotty.insertnode` and `dot.insertnode`. `dot.insertnode` inserts the node into the graph data structure. `dotty.insertnode` first calls `dot.insertnode`, then draws the node in the window.

```
dot.insertnode = function (graph, name, attr) {
    local nid, node, aid;
    if (~graph)
        return null;
    nid = graph.maxnid;
    if (~name) {
        while (graph.nodedict[(name = concat ('n', nid))] >= 0)
            nid = nid + 1;
    } else if (graph.nodedict[name] >= 0) {
        dot.message (0, concat ('node: ', name, ' exists'));
        return null;
    }
    graph.nodedict[name] = nid;
    graph.maxnid = nid + 1;
    graph.nodes[nid] = [
        dot.knid   = nid;
        dot.kname  = name;
        dot.kattr  = copy (dot.nodeattr);
        dot.kedges = [];
    ];
    node = graph.nodes[nid];
    for (aid in attr)
        node.attr[aid] = copy (attr[aid]);
    return node;
};


dotty.insertnode = function (gt, pos, name, attr) {
    local node, size;
    if (~attr)
        attr = [];
    if (~attr.label)
        attr.label = '\N';
    if (~pos)
        pos = ['x' = 10; 'y' = 10;];
    size = ['x' = strlen (attr.label) * 30; 'y' = 30;];
    if (~(node = dot.insertnode (gt.graph, name, attr)))
        return null;
    node.pos = pos;
    node.size = size;
    dotty.unpacknodeattr (gt, node);
    dotty.drawnode (gt.views, node);
    return node;
};
```

`dotty.computelayout` generates a new layout. `dot.writegraph` is used to send the graph to *dot*, which is running as a separate process. `dot.readgraph` is used to get the result from *dot*.

```
dotty.computelayout = function (gt) {
    ...
    graph = gt.graph;
    dot.writegraph (dotty.dotfd, graph, 1);
    if (~(g = readgraph (dotty.dotfd))) {
        ...
    }
    ...
};
```

Figure 16 shows a sample DAG.

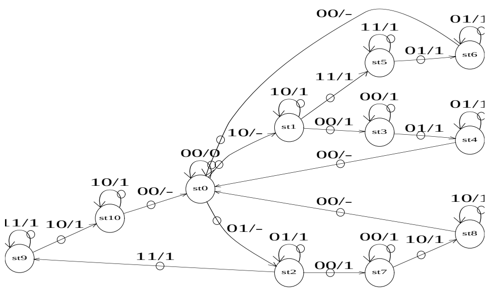


Figure 16: A Directed Acyclic Graph

dotty itself has become the basis for further customizations [KN93].

5 Conclusions

A unique feature of *lefty* is the use of a single language to describe all aspects of picture handling. Editing operations and layout algorithms are not hardwired in the editor; they are part of the picture specification. This allows the editor to handle a variety of pictures and still provide, for each type of picture, functionality comparable to that of dedicated tools.

Providing two views, each of which presents information at a different level of abstraction, gives users more flexibility in editing a picture. Some changes are easier to describe in one view than in another. Also, users have preferences; some prefer describing operations with programs, while others prefer using the mouse.

The editor's ability to communicate with external processes allows it to make use of existing tools whose functionality would be difficult to duplicate. This extensibility also makes it possible to edit pictures for which the editor's procedural description is not desirable. For example, a constraint-based editing environment can be implemented as an external process. Such a process can display both the picture and the constraints and allow the user to edit both. This arrangement simplifies the implementation of a constraint-based system because the editor already provides support for the user interface, and allows the constraint solver to be written in any language.

Using *lefty* to construct graphical front ends for existing tools is fast and convenient since the existing tools do not need to be modified. *lefty*, however, can also be used for building new applications. This can be a good alternative to building applications by integrating the user interface with the main application into a single program. Implementing the user interface as a separate process helps make it clear what functionality belongs to the user interface and what belongs to the main application. Having a programmable front end makes it easier to experiment with different approaches. Debugging is also easier, since the main application can be driven by a text file. In an integrated application one would have to perform the sequence of mouse and keyboard events that lead to the problem, and this can be tedious and error-prone. The speed disadvantage of an interpreted system, and the cost of inter-process communication can—in some case—be prohibitive. In most cases, however, these disadvantages do not affect the response time, which is dominated by window system operations. In fact, having separate processes can improve performance, since the processes can execute to some extent in parallel.

A Running *lefty*

lefty can be started by issuing the command:

```
lefty [options] [file]
```

The file name is optional. It may be -, for reading from standard input. *lefty* uses two environment variables, LEFTYPATH and LEFTYOPTIONS. LEFTYPATH is a colon separated list of directories. When *lefty* tries to open a file, it searches that path for the file. When *lefty* tries to start up another process, it searches LEFTYPATH first, then the standard PATH variable. LEFTYOPTIONS can be used to set specific options. Options specified on the command line override options set through this variable. Table 4 shows the supported options.

Option	Range	Default	Description
-x			Instructs the editor to exit after processing file.
-e <expr>	*lefty* expr.		expression is parsed and executed.
-el <num>	0-5	0	Set error reporting level. 0 never prints any messages. 1 prints severe errors, such as trying to return from a non function. 2 is the most useful: it reports function calls that cannot be executed, either because there is no function, or because of argument mismatches. 3 also warns about bad variable names. 4,5 warn about expressions that do not return a value. Only level 1 messages are real errors. The rest arise from legal *lefty* statements, but may be cased by some logic errors.
-sd <num>	0-2	2	Specifies how much of the stack to show, when an error message is to be printed. With 0, no part of the stack is shown. With 1, only the top stack frame is printed. With 2, the full stack is printed.
-sb <num>	0-2	2	Specifies how much of each function in the stack to show, when an error message is to be printed. With 0, no part of the function is shown. With 1, only the line around the error is printed. With 2, the full function body is printed.
-df <string>		""	Sets the default font. This font is used whenever a requested font cannot be found. The string must be a legal X font. If string is "", *lefty* will draw small boxes instead of text.
-ps <file>		out.ps	Specifies a default file name for postscript files. This name is used when no name is specified in the createwidget call.
-V			Prints the version in stderr

Table 4: Command line options

B Language Specification

In the formal specification of the language below, keywords are shown in typewriter font, alternatives are separated by vertical bars, parentheses indicate grouping, optional clauses are indicated by brackets, and optional repetition is indicated by braces.

expression:
scalar-constant
variable [`=` *expression*]
expression (`|` | `&`) *expression*
expression (`==` | `~=` | `<` | `<=` | `>` | `>=`) *expression*
expression (`+` | `-` | `*` | `/` | `%`) *expression*
function-declaration
`[` { *expression* `=` *expression* `;` } `]`
variable `(` [*expression* { `,` *expression* }] `)`
`(` *expression* `)`

variable:
identifier
variable `.` *identifier*
variable `[` *expression* `]`

function-declaration:
`function` *identifier* `(` [*identifier* { `,` *identifier* }] `)` `{`
{ `local` [*identifier* { `,` *identifier* }] `;` }
{ *statement* }
`}`

statement:
expression `;`
`{` { *statement* } `}`
`if` `(` *expression* `)` *statement* [`else` *statement*]
`while` `(` *expression* `)` *statement*
`for` `(` *expression* `;` *expression* `;` *expression* `)` *statement*
`for` `(` *variable* `in` *expression* `)` *statement*
`break` `;`
`continue` `;`
`return` [*expression*] `;`

A scalar constant is a number or a quoted character string. The language does not separate integer and real types; all numbers are reals.

The dot syntax *variable* `.` *identifier* is just a shorthand for *variable* `[` `"` *identifier* `"` `]`.

Assignment evaluates the right-hand side expression and assigns the resulting value to the variable on the left-hand side. If evaluation of the right-hand side expression returns no value, the left-hand side variable retains its previous value. Only the last component of the left-hand side variable may be undefined, otherwise the assignment fails. For example, `a.b.c = 10` succeeds if at least `a.b` is defined.

Assignment of tables is by reference. For example, if `b` holds a table, `a = b` results in `a` pointing to the same table.

The order of evaluation for `&` and `|` is left-to-right and evaluation terminates once the result is determined. For example, evaluation of the expression:

```
0 == 1 | 1 == 1 | a ()
```

begins by evaluating `0 == 1`. This is false, so execution proceeds with `1 == 1`. Since this comparison is true, the whole expression is true, so evaluation terminates. `a` is never called.

If the two sides of a comparison have different types, the result is false.

For arithmetic operations, if any of the expressions is not a number, evaluation aborts. For `%`, the two expressions must have no fractional part.

For table construction, each of the left-hand side expressions must evaluate to a scalar.

Functions are stored as scalars. There are also built-in functions; they provide functionality that cannot be written in the language itself. Section 3.5 describes the built-in functions.

For function calls, if the evaluation of an argument returns no value, the function body is not executed.

C Program Listings

C.1 Box Program

```
load ('def.lefty');
definit ();
#
# initialize window data
#
canvas = defcanvas;
wrect = [0 = ['x' = 0; 'y' = 0;]; 1 = ['x' = 400; 'y' = 500;];];
setwidgetattr (canvas, ['window' = wrect;]);
#
# data structures
#
objarray = [];
objnum = 0;
#
# misc functions
#
min = function (a, b) {
    if (a <= b)
        return a;
    return b;
};
max = function (a, b) {
    if (b <= a)
        return a;
    return b;
};
rectof = function (p1, p2) {
    return [
        0 = ['x' = min (p1.x, p2.x); 'y' = min (p1.y, p2.y);];
        1 = ['x' = max (p1.x, p2.x); 'y' = max (p1.y, p2.y);];
    ];
};
pointadd = function (p1, p2) {
    return ['x' = p2.x + p1.x; 'y' = p2.y + p1.y;];
};
pointsub = function (p1, p2) {
    return ['x' = p2.x - p1.x; 'y' = p2.y - p1.y;];
};
#
# rendering functions
#
drawbox = function (obj, color) {
```

```
    box (canvas, obj, obj.rect, ['color' = color;]);
};
redrawboxes = function () {
    local i;
    clear (canvas);
    for (i = 0; i < objnum; i = i + 1)
        drawbox (objarray[i], 1);
};
redraw = function (canvas) {
    redrawboxes ();
};
#
# editing functions
#
new = function (rect) {
    objarray[objnum] = [
        'rect' = rect;
        'id' = objnum;
    ];
    objnum = objnum + 1;
    return objarray[objnum - 1];
};
reshape = function (obj, rect) {
    obj.rect = rect;
    return obj;
};
move = function (obj, p) {
    obj.rect[0] = pointadd (obj.rect[0], p);
    obj.rect[1] = pointadd (obj.rect[1], p);
    return obj;
};
delete = function (obj) {
    if (obj.id ~= objnum - 1) {
        objarray[obj.id] = objarray[objnum - 1];
        objarray[obj.id].id = obj.id;
    }
    remove (objnum - 1, objarray);
    objnum = objnum - 1;
};
#
# user interface functions
#
# left mouse button creates new box
# middle button moves a box
# right button deletes a box
#
```

```
leftdown = function (data) {
    if (data.obj ~= null)
        return;
    leftbox = new (rectof (data.pos, data.pos));
    drawbox (leftbox, 1);
    setgfxattr (canvas, ['mode' = 'xor';]);
};
leftmove = function (data) {
    if (~leftbox)
        return;
    drawbox (leftbox, 1);
    clearpick (canvas, leftbox);
    reshape (leftbox, rectof (data.ppos, data.pos));
    drawbox (leftbox, 1);
};
leftup = function (data) {
    if (~leftbox)
        return;
    drawbox (leftbox, 1);
    clearpick (canvas, leftbox);
    reshape (leftbox, rectof (data.ppos, data.pos));
    setgfxattr (canvas, ['mode' = 'src';]);
    drawbox (leftbox, 1);
    remove ('leftbox');
};
middledown = function (data) {
    if (data.obj == null)
        return;
    middlebox = data.obj;
    middlepos = data.pos;
    setgfxattr (canvas, ['mode' = 'xor';]);
};
middlemove = function (data) {
    if (~middlebox)
        return;
    drawbox (middlebox, 1);
    clearpick (canvas, middlebox);
    move (middlebox, pointsub (middlepos, data.pos));
    middlepos = data.pos;
    drawbox (middlebox, 1);
};
middleup = function (data) {
    if (~middlebox)
        return;
    drawbox (middlebox, 1);
    clearpick (canvas, middlebox);
```

```
    move (middlebox, pointsub (middlepos, data.pos));
    setgfxattr (canvas, ['mode' = 'src';]);
    drawbox (middlebox, 1);
    remove ('middlepos');
    remove ('middlebox');
};
rightup = function (data) {
    if (data.pobj == null)
        return;
    drawbox (data.obj, 0);
    clearpick (canvas, data.obj);
    delete (data.obj);
};
dops = function () {
    local s;

    s = ['x' = 8 * 300; 'y' = 10.5 * 300;];
    canvas = createwidget (-1, ['type' = 'ps'; 'size' = s;]);
    setwidgetattr (canvas, ['window' = wrect;]);
    redraw (canvas);
    destroywidget (canvas);
    canvas = defcanvas;
};
```

C.2 Delaunay Triangulation Program

```
load ('def.lefty');
definit ();
# data structures
#
sitesnum = 0;
sites = [];
lines = [];
canvas = defcanvas;
wrect = [0 = ['x' = 0; 'y' = 0;]; 1 = ['x' = 400; 'y' = 500;];];
setwidgetattr (canvas, ['window' = wrect;]);
triedfd = openio ('pipe', 'tried', 'w+', '%e %i %o');

# drawing functions
#
redraw = function (id) {
    local i, j, rect, s;
    rect = [];
    clear (canvas);
    for (i in lines) {
        for (j in lines[i]) {
            s = lines[i][j];
            line (canvas, null, ['x' = s.f.point.x; 'y' = s.f.point.y;],
                    ['x' = s.l.point.x; 'y' = s.l.point.y;]);
        }
    }
    for (i = 0; i < sitesnum; i = i + 1) {
        rect[0] = [
            'x' = sites[i].point.x - 5; 'y' = sites[i].point.y - 5;
        ];
        rect[1] = [
            'x' = sites[i].point.x + 5; 'y' = sites[i].point.y + 5;
        ];
        box (canvas, sites[i], rect);
    }
};

# editing functions
#
insert = function (point) {
    local s;
    sites[sitesnum].num = sitesnum;
    sites[sitesnum].point = point;
    writeline (triedfd,
            concat ('new ', sitesnum, ' ', point.x, ' ', point.y));
```

```
    sitesnum = sitesnum + 1;
    while ((s = readline (triedfd)) ~= '')
        run (s);
    box (canvas, sites[sitesnum - 1],
            [0 = ['x' = point.x - 5; 'y' = point.y - 5;];
             1 = ['x' = point.x + 5; 'y' = point.y + 5;];
            ]);
};
mv = function (node, point) {
    local i;
    box (canvas, node, [
            0 = ['x' = node.point.x - 5; 'y' = node.point.y - 5;];
            1 = ['x' = node.point.x + 5; 'y' = node.point.y + 5;];
            ], ['color' = 0;]);
    clearpick (canvas, node);
    for (i = 0; i < sitesnum; i = i + 1) {
        if (lines[i][node.num])
            delline (i, node.num);
        if (lines[node.num][i])
            delline (node.num, i);
    }
    node.point = point;
    writeline (triedfd,
            concat ('mv ', node.num, ' ', point.x, ' ', point.y));
    while ((s = readline (triedfd)) ~= '')
        run (s);
    box (canvas, node, [
            0 = ['x' = point.x - 5; 'y' = point.y - 5;];
            1 = ['x' = point.x + 5; 'y' = point.y + 5;];
            ]);
};
insline = function (i, j) {
    lines[i][j].f = sites[i];
    lines[i][j].l = sites[j];
    line (canvas, null,
            ['x' = sites[i].point.x; 'y' = sites[i].point.y;],
            ['x' = sites[j].point.x; 'y' = sites[j].point.y;]);
};
delline = function (i, j) {
    remove (j, lines[i]);
    if (tablesize (lines[i]) == 0)
        remove (i, lines);
    line (canvas, null,
            ['x' = sites[i].point.x; 'y' = sites[i].point.y;],
            ['x' = sites[j].point.x; 'y' = sites[j].point.y;],
            ['color' = 0;]);
```

```
};

# user interface functions
#
leftdown = function (data) {
    if (~data.obj)
        insert (data.pos);
};
leftmove = function (data) {
    if (data.obj)
        mv (data.obj, data.pos);
};
keydown = function (data) {
    redraw (0);
};
dops = function () {
    local r;

    r = [0 = ['x' = 0; 'y' = 0;]; 1 = ['x' = 8 * 300; 'y' = 10.5 * 300;];];
    canvas = createwidget (-1, [
        'type' = 'ps';
        'origin' = r[0];
        'size' = r[1];
    ]);
    setwidgetattr (canvas, ['window' = wrect;]);
    redraw (0);
    destroywidget (canvas);
    canvas=defcanvas;
};
```

References

[FH85] C. W. Fraser and D. R. Hanson. High-level language facilities for low-level services. In *12th ACM Symp. on Prin. of Programming Languages*, pages 217–224, 1985.

[GKNV93] E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE-TSE*, March 1993.

[GNV88] E. R. Gansner, S. C. North, and K. P. Vo. DAG—A program that draws directed graphs. *Software—Practice and Experience*, 18(11):1047–1062, November 1988.

[GS85] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[KN93] Eleftherios Koutsofios and Stephen C. North. Viewing graphs with *dotty*. Technical report, AT&T Bell Laboratories, 1993.

[PLR85] R. Pike, B. Locanthi, and J. Reiser. Handware/software trade-offs for bitmap graphics on the blit. *Software—Practice and Experience*, 15(2):131–151, 1985.

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

[SUN88] SUN Microsystems Inc., 2550 Garcia Ave., Mountain View, CA 94043. *NeWS Manual*, 1988.

[TdBB88] R. Tamassia, G. di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, January/February 1988.

[Uni85] University of California at Berkeley. *Magic*, 1985.