CS405 Introduction to Empirical Modelling 2008

Some answers for Labsheet 2: Script mechanics - **`eden`**, **`donald`** and **`scout`**

Mechanics of definitive scripts

We have seen that a simple piece of script can have many different interpretations, and can be refined in many ways. For this reason, it is often useful to be able to extract subsets of definitions from existing scripts and incorporate them in your own models. In order to do this effectively, you need to understand the relationship between the `eden`, `donald` and `scout` notations and know how to tackle the problem of making sense of what may be a relatively complex script. In this laboratory, you will get some experience of this activity by looking at `roomviewerYung1991`. Understanding a script in this way is clearly analogous to what is described as "program comprehension", an activity that doesn't necessarily have much to do with understanding how a program is interpreted in its application context. In the case of a well-written definitive script, we can expect a closer connection between understanding the mechanics of a script and appreciating its relationship to an external referent.

Each exercise illustrates useful techniques that can be exploited in many models and comprises a set of relatively small questions. In tackling them, you may find it helpful to first address one or two questions from each.

Exercise 1: Understanding how `donald` is represented in `eden`

By examining the observables in the `roomviewer.s` script, answer the following questions:

- What is the current definition of `table/SW`?

```
Inspect donald definitions, or execute

?_table_SW;
```

- How would you print out the current value of `desk/drawer/k` every time that it is changed?

```
proc wddk : _desk_drawer_k {
        writeln(_desk_drawer_k);
}

%donald
desk/drawer/k = 3
```

- Why is the line that represents the cable dashed?

```
## ?A_cable;
A_cable="linestyle=dashed,dash=13";
A_cable ~> [P_cable]; /* A_cable last changed by input */
```

- The observable `cableIsShort` is set to `true` when the distance between the socket and the lamp exceeds the length of the cable. Determine what observables influence the value of `cableIsShort` and write down some representative examples of redefinitions that resolve the problem of the short cable, together with their interpretations in the referent.

```
%eden
## ?_cableIsShort; etc

_cableIsShort is dist(dot1(_cable), dot2(_cable)) > _cablelength;
/* current value of _cableIsShort is 0 */

_cableIsShort ~> [_, monCableStr, screen];
/* _cableIsShort last changed by input */

_cable is line(_plug, _table_lamp_centre);
/* current value of _cable is ['L',['C',500,100],['C',650,650]] */
_cable ~> [P_cable, _, _cableIsShort]; /* _cable last changed by input */

_plug is _plug1; /* current value of _plug is ['C',500,100] */
_plug ~> [P_plug, _cable, _, plugMenu]; /* _plug last changed by input */
```

```
_table_lamp_centre is scalar_div((vector_add(_table_SW, _table_NE)), 2);
/* current value of _table_lamp_centre is ['C',650,650] */
_table_lamp_centre ~> [P_table_lamp_centre, _table_lamp, _table_lamp_L1, _table_lamp_L2,
_table_lamp_L3, _table_lamp_L4, _table_lamp_L5, _table_lamp_L6, _table_lamp_L7,
_table_lamp_base, _cable]; /* _table_lamp_centre last changed by input */

_table_SW is scalar_div((vector_add(_SW, _NE)), 2);
/* current value of _table_SW is ['C',500,500] */
_table_SW ~> [P_table_SW, _table_S, _table_W, _table_SE, _table_NE, _table_NW, _table,
_table_lamp_centre]; /* _table_SW last changed by input */

_SW is cart(100, 100); /* current value of _SW is ['C',100,100] */
_SW ~> [P_SW, _S, _W, _SE, _NE, _NW, _table_SW, _desk_SW, _] /* _SW last changed by input */

_NE is vector_add(_SW, cart(_width, _length)); /* current value of _NE is ['C',900,900] */
_NE ~> [P_NE, _N2, _E, _table_SW, _]; /* _NE last changed by input */

_width is 800; /* current value of _width is 800 */
_width ~> [_SE, _NE, _]; /* _width last changed by input */

_length is 800; /* current value of _length is 800 */
_length ~> [_NE, _NW, _]; /* _length last changed by input */

_table_NE is vector_add(_table_SW, cart(_table_width, _table_length));
/* current value of _table_NE is ['C',800,800] */
_table_NE ~> [P_table_NE, _table_N, _table_E, _table, _table_lamp_centre];
/* _table_NE last changed by input */

_table_width is 300; /* current value of _table_width is 300 */
_table_width ~> [_table_SE, _table_NE, _table]; /* _table_width last changed by input */

_table_length is 300; /* current value of _table_length is 300 */
_table_length ~> [_table_NE, _table_NW, _table]; /* _table_length last changed by input */

## can display this with the DMT
```

- How would you adapt the model so that:
  - the cable is coloured red when it is not long enough?

    ```
    %eden
    A_cable is "linestyle=dashed,dash=13,color=" // ((_cableIsShort) ? "red": "black");
    ```

  - the door is coloured green when open and red when shut?

    ```
    A_door_door is "color=" // ((_door_open) ? "green" : "red");
    ```

  - in the improved version of the desk those parts of the drawer that are invisible are displayed as dashed lines?

    ```
    ## the fixed desk (from Lab 1):

    %donald
    within desk {
      within drawer {
            real howopen
         E = [NE, SE]
         W = [NW, SW]
         S = [SW, SE]
         N = [NW, NE]
         SE = SW + {length, 0}
         SW = NW - {0, width}
         NE = NW + {length, 0}
         NW = (~/NE  * howopen + ~/NW * (1-howopen))
            length = ~/width
         width = ~/length div 3
        }
    }
    desk/drawer/howopen = 1.0       ## open
    desk/drawer/howopen = 0.0       ## closed

    within desk {
        within drawer {
                    line outside
                    line inside
                    point meetsdeskedge
                    meetsdeskedge = intersect(S, ~/E)
                    outside = [meetsdeskedge,SE]
                    inside = [SW, meetsdeskedge]
            }
    }

    A_desk_drawer_inside is "linestyle=dashed,dash=1,color=grey";
    A_desk_drawer_W is "linestyle=dashed,dash=1,color=grey";

    A_desk_drawer_inside is "color=grey";
    A_desk_drawer_W is "color=grey";
    ```

```
## am thinking that whether S or the segments inside, outside get displayed
## is fortuitous, as no layering in donald. May be OK here as S ~> inside, outside
## alternatively can attempt to hide S side of drawer (these fail!):

within desk {
    within drawer {
        S = [SW, SW]
        }
}

## a mistake, as there S no longer intersects ~/E


within desk {
    within drawer {
        S = [SW, SE]
        }
}

%eden
proc P_desk_drawer_S {};
## this suppresses the visualisation of the south side of the drawer
## but seems to have undesirable side-effects
```

- Place a small circle at the centre of the lamp to indicate a bulb, and set its attributes so that it is coloured solid yellow or black according to whether the lamp is on or off.

```
within table/lamp {
        circle bulb
        bulb = circle(centre, 10)
}
%eden
A_table_lamp_bulb is "fill=solid,color=yellow");
A_table_lamp_bulb is "fill=solid,color=" // ((lamp_on) ? "yellow" : "black");
```

- Attach to the centre of the table a `donald` label that displays its current coordinates.

```
%donald
label tablePos
%eden
_tablePos is label("X", scalar_mult(vector_add(_table_NE, _table_SW), 0.5));
labelStr is str(tail(vector_add(_table_NE, _table_SW)));
_tablePos is label(labelStr, scalar_mult(vector_add(_table_NE, _table_SW), 0.5));
## more elegant solutions might define label in donald and use escape to eden
## also good to define the centre of the table as an observable
## could use int() to convert result to integer values for display purposes
```

- Explain why nothing happens if you modify the `roomviewerYung1991` model by introducing the definition of the rotated table in the Appendix to this labsheet. How do you resolve this problem? [Hint: you may need to make use of `eden` to define some `donald` observables.]

```
## table/centre is newly declared, hence undefined
## to define it without moving the table, must use an assignment - so need eden:
%eden
_table_centre = scalar_mult(vector_add(_table_NE, _table_SW), 2.0);
## assuming that table/centre is already declared
## can then introduce the rotated table code
## can move table e.g. by
table/centre = {500,500}
## can rotate table e.g. by
table/tablerotangle = pi div 3
```

Exercise 2: Understanding how `scout` describes screen layout

- Trace the set (or a representative subset) of the dependencies that determines the windows that make up the screen display.

```
## Can use
?screen;
## in scout and then trace the definitions of the displays and windows encountered
## can do the same in eden, and use the output in the DMT to get a graphical trace
```

- Identify which observables determine the size and location of the lefthand window containing the `donald` floorplan of the room.

```
window don1 = {
    type: DONALD
    box: [p1, q1]
    pict: "view"
    border: 1
    sensitive: ON
};

## the observables are p1 and q1 (points in scout)
```

- Find the definition that determines the text displayed on the door button (as defined by the `doorButton` `text` window in `scout`).

```
window doorButton = {
    type: TEXT
    frame: ([doorButtonPos, 1, strlen(doorMenu)])
    string: doorMenu
    border: 1
    sensitive: ON
};

## the definition is that of the string doorMenu defined in scout:
## doorMenu = if _door_open then "10:Close Door" else "10:Open Door" endif;
```

- How would you change the background colour and the colour of the text displayed on the door button?

```
window doorButton = {
    type: TEXT
    frame: ([doorButtonPos, 1, strlen(doorMenu)])
    string: doorMenu
    border: 1
    fgcolor: "red"
    bgcolor: "white"
    sensitive: ON
};
```

- How would you place a single rectangular window behind the windows displaying the floorplan and the enlarged floorplan in such a way as to provide a blue background?

```
window background = {
        type: DONALD
        bgcolor: "lightblue"
        box: [p1,q2]
};
## NB need to have this last semi-colon!!!

basicScreen = ;

window background = {
        type: DONALD
        bgcolor: "lightblue"
        box: [p1 - {10,10},q2 + {10,10}]
};
```

Exercise 3: Understanding how `donald` drawings are displayed in `scout` windows

- Inspect the definitions of the `scout` windows that display the floorplan and enlarged floorplan. Explain why the `pict` field is the same in both windows.

```
Both windows make use of the same donald room line drawing
```

- How would you interchange the drawings displayed in the two `scout` windows of type `DONALD`?

```
%scout
window don1 = {
    type: DONALD
    box: [p2, q2]
    pict: "view"
    border: 1
    sensitive: ON
};
```

```
window don2 = {
    type: DONALD
    box: [p1, q1]
    pict: "view"
    xmin: zoomPos.1 - zoomSize / 2
    ymin: zoomPos.2 - zoomSize / 2
    xmax: zoomPos.1 + zoomSize / 2
    ymax: zoomPos.2 + zoomSize / 2
    border: 1
    sensitive: ON
};
```

- How can you determine the `donald` coordinates of a point in the floorplan and enlarged floorplan displays solely by using mouse clicks?

```
## open the Command History window and read off
## the coordinates from a mouse click in the window don1 etc
~don1_mouse = [1,4,0,363.184079602, 462.686567164]; ## when depressed
~don1_mouse = [1,5,256,363.184079602, 462.686567164]; ## when released
```

- By assigning new values to the observables `zoomPos` and `zoomSize` and observing the impact on the `scout` window containing the enlarged floorplan, determine the significance of the `xmin`, `xmax`, `ymin` and `ymax` fields within the definition of the window.

```
The region of a donald picture that is displayed in a Scout window is rectangular.
Its SW corner is at {xmin,ymin} and its NE corner is at {xmax, ymax}
```

- Suppose that a `scout` window of type `DONALD` displays a line drawing without any distortion. What relationship must there be between the dimensions of the window and the values of the fields `xmin`, `xmax`, `ymin` and `ymax`?

```
Must have preservation of the aspect ratio
```

- Make use of `donald` variable of type `shape` to specify an image of the table rotated through an angle that can be freely specified via the input window. Revise the definition of the `scout` window that presently shows the enlarged floorplan so that it instead displays the rotated table.

```
%scout
window don2 = {
    type: DONALD
    box: [p2, q2]
    pict: "rotatedtable"
    xmin: zoomPos.1 - zoomSize / 2
    ymin: zoomPos.2 - zoomSize / 2
    xmax: zoomPos.1 + zoomSize / 2
    ymax: zoomPos.2 + zoomSize / 2
    border: 1
    sensitive: ON
};
%donald
viewport rotatedtable
real rotangle
shape rottable
rotangle = pi div 3
rottable = rot(table, (table/NE+table/SW) div 2, rotangle)
%eden
zoomPos is tail(scalar_mult(vector_add(_table_NE, _table_SW), 0.5));
## NB using the buttons to adjust zoomPos will remove this definition
```

Exercise 4: Understanding how GUI agents are specified in `scout` and `eden`.

- Determine what `eden` redefinition is executed when the door button is clicked. Confirm that this redefinition can be entered directly through the input window and explain why it has the effect of opening or shutting the door.

```
?doorButton_mouse_1;
doorButton_mouse_1=[1,5,256,23,6];
doorButton_mouse_1 ~> [doorButton_to_input, doorButton_mouseButtonPress];
proc doorButton_to_input: doorButton_mouse_1 {
        if (doorButton_mouse_1[2] == 4) input = 10;
}
## can achieve same effect in input window via:

input = 10;
?input;
proc handle_user_input: input {
        switch (input) {
```

```
        case 1: _table_SW = vector_add(_table_SW, cart(0, 100));        break;
        case 2: _table_SW = vector_sub(_table_SW, cart(0, 100));        break;
        case 3: _table_SW = vector_sub(_table_SW, cart(100, 0));        break;
        case 4: _table_SW = vector_add(_table_SW, cart(100, 0));        break;
        case 5: zoomPos = pt_add(zoomPos, [0, 100]);                    break;
        case 6: zoomPos = pt_subtract(zoomPos, [0, 100]);               break;
        case 7: zoomPos = pt_subtract(zoomPos, [100, 0]);               break;
        case 8: zoomPos = pt_add(zoomPos, [100, 0]);                    break;
        case 9: if (_plug == _plug1) {
                        _plug is _plug2;
                } else {
                        _plug is _plug1;
                }
                break;
        case 10: _door_open = !_door_open;                              break;
        }
}
```

- Modify the `eden` action associated with the door button so that it toggles the colour of the door from red to green. Explain why this is a less satisfactory way of specifying the colour of the door than using a definition as proposed in Exercise 1 above.

```
Determines state by side-effect, so more liable to get out of sync - e.g. if save and reload model
```

- Adapt the `eden` action associated with the door button so that it counts the number of times the door is opened. Modify the text on the door button so that it displays this count.

```
count=0;
proc doorButton_to_input: doorButton_mouse_1 {
   if (doorButton_mouse_1[2] == 4)
      {
         count++;
         input = 10;
      }
}

doorMenu is str(count);
```

- Identify the `eden` action that is triggered by clicking the mouse in the `scout` window displaying the floorplan. Explain how this action moves the table.

```
The action that is triggered is don1_to_tableSW. This action responds to mouse clicks in the don1
window. The action has two branches: one of which is executed when the second mouse button is
depressed (don1_mouse[2]==4) and one of which is executed when the second mouse button is released
(don1_mouse[2]==5). If the mouse is depressed within the region occupied by the table, the action
sets a flag (move_table) to 1, then records the current position of the table (old_table_SW) and of
the mouse (old_mouse_pos) When the mouse is released and the move_table flag is set to 1, the
action is triggered a second time and the second branch of the action is executed. The current
mouse position is then used to compute the intended displacement of the table from the previous
mouse position and to redefine the table position accordingly.
```

- Explain why the buttons that relocate the table no longer work correctly when the rotating table mentioned in Exercise 1 is introduced. Fix this problem.

```
## the buttons still act on the SW corner of the table
## they also break the dependency linking the SW corner of the table to its centre
## but its coordinates are now defined relatve to its centre

## adapt buttons on interface to act on the centre of the table

proc handle_user_input: input {
        switch (input) {
        case 1: _table_centre = vector_add(_table_centre, cart(0, 100));        break;
        case 2: _table_centre = vector_sub(_table_centre, cart(0, 100));        break;
        case 3: _table_centre = vector_sub(_table_centre, cart(100, 0));        break;
        case 4: _table_centre = vector_add(_table_centre, cart(100, 0));        break;
        case 5: zoomPos = pt_add(zoomPos, [0, 100]);                    break;
        case 6: zoomPos = pt_subtract(zoomPos, [0, 100]);               break;
        case 7: zoomPos = pt_subtract(zoomPos, [100, 0]);               break;
        case 8: zoomPos = pt_add(zoomPos, [100, 0]);                    break;
        case 9: if (_plug == _plug1) {
                        _plug is _plug2;
                } else {
                        _plug is _plug1;
                }
                break;
        case 10: _door_open = !_door_open;                              break;
        }
```

```
      }

      %donald
      ## restore the dependency
      within table {
              SW = centre - rot({width div 2, length div 2}, {0,0}, tablerotangle)
      }
```

Appendix to Labsheet 2

```
%donald

## first redefine coordinates of table relative to its centre

within table {
        point centre
        SW = centre - {width div 2, length div 2}
        NW = centre - {width div 2, -length div 2}
        SE = centre + {width div 2, -length div 2}
        NE = centre + {width div 2, length div 2}
}

## now define the rotation

within table {
        point centre
        real tablerotangle
        SW = centre - rot({width div 2, length div 2}, {0,0}, tablerotangle)
        NW = centre - rot({width div 2, -length div 2}, {0,0}, tablerotangle)
        SE = centre + rot({width div 2, -length div 2}, {0,0}, tablerotangle)
        NE = centre + rot({width div 2, length div 2}, {0,0}, tablerotangle)
}

## specify angles in radians - as floating point numbers

table/tablerotangle = pi div 4
```