```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  File:     PCrecPic.dem (For PC computers)
%  Purpose:  User interface for Recursive Picture
%  Author:   Jeff Edmonds
%  Location: York University
%  Email:    jeff@cs.yorku.ca
%  Date:     Jan, 1996
% --------------------------------

import
    % Mouse in "PClib/widget/Mouse",
    GUIControl in "PClib/widget/gui.tu",
    PicRadioButtonWidget in "PClib/widget/picrbtn.tu",
    PicButtonWidget in "PClib/widget/picbtn.tu",
    RadioWidget in "PClib/widget/radio.tu",
    NewMenu in "PClib/widget/menu.tu",
    WindowClass in "PClib/predefs/windowcl.tu",
    RecursivePicture in "recPic.tu"

% ButtonWidget in "PClib/widget/button.tu",
% CheckBoxWidget in "PClib/widget/checkbox.tu",


var rPic, reprPic : ^RecursivePicture
new rPic
rPic -> Initialize
new reprPic
reprPic -> Initialize
var fileName : string := ""

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Odd routines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure ColorTest ()
    for i : 1 .. maxcolor
        Draw.Line (i * 10, 0, i * 10, maxy, i)
    end for
end ColorTest

% ColorTest ()

procedure str2int (str : string, var x, y : int)
    var tempstr : string := ""
    var str2 : string := str + " "
    var j : int := 1
    loop
        exit when str2 (j) = " "
        tempstr += str2 (j)
        j += 1
    end loop
    x := strint (tempstr)
    tempstr := ""
    loop
        exit when str2 (j) not= " "
        j += 1
    end loop
    loop
        exit when str2 (j) = " "
        tempstr += str2 (j)
        j += 1
    end loop
    y := strint (tempstr)
end str2int

function ArcAng (xc, yc, x, y : int) : int
    if x - xc = 0 then
        if y > yc then
            result 90
        else
            result 270
        end if
    else
        var ang := round (arctand ( (y - yc) / (x - xc)))
        if x < xc then
            ang += 180
        end if
        result ang mod 360
    end if
end ArcAng

% Given three points returns the center radius and angles of the circle
% or begin and end of line
procedure Center (x1, y1, x2, y2, x3, y3 : int,
        var xc, yc, r, a1, a2 : int, var isLine : boolean)
    var denom := (x3 * y2 - x3 * y1 - x1 * y2 - x2 * y3 + x2 * y1 + x1 *
        y3)
        * 2
    if denom = 0 then
        isLine := true
    else
        isLine := false
        xc := ( - y2 ** 2 * y3 + y2 ** 2 * y1 - y2 * y1 ** 2
            + x1 ** 2 * y3 - x2 ** 2 * y3 + x2 ** 2 * y1
            + y2 * y3 ** 2 - y1 * y3 ** 2 + y3 * y1 ** 2
            - x1 ** 2 * y2 + x3 ** 2 * y2 - x3 ** 2 * y1)
            div denom
        yc := - ( - x3 * y2 ** 2 + x1 * y2 ** 2 - y3 ** 2 * x1
            + x3 ** 2 * x2 - x2 * y1 ** 2 + x3 * y1 ** 2
            - x3 ** 2 * x1 + y3 ** 2 * x2 - x2 ** 2 * x3
            - x1 ** 2 * x2 + x1 ** 2 * x3 + x2 ** 2 * x1)
            div denom
        const maxVal : int := round (sqrt (maxint) / 10)
        if xc > maxVal or yc > maxVal then
            isLine := true
            return
        end if
        r := round (sqrt ( (xc - x1) ** 2 + (yc - y1) ** 2))
        a1 := ArcAng (xc, yc, x1, y1)
        a2 := ArcAng (xc, yc, x2, y2)
        var a3 := ArcAng (xc, yc, x3, y3)
        if (a3 < a1 and a1 < a2) or
                (a1 < a2 and a2 < a3) or
                (a2 < a3 and a3 < a1) then
            var temp := a1
            a1 := a2
            a2 := temp
        end if
    end if
end Center


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Setting Window Info
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Constants for the window
const titleBarHeight : int := 21
const windowEdgeSize : int := 13
```

```
% Constants for the screen
var screenWidth := Config.Display (cdScreenWidth)
var screenHeight := Config.Display (cdScreenHeight)

% Create the new output window
var window : ^WindowClass
new window

% Position the new output window
window -> SetSize (screenWidth * 2 div 3 - windowEdgeSize,
    screenHeight * 2 div 3 - windowEdgeSize - titleBarHeight)
window -> SetPosition (0, 0)

setscreen ("nocursor,noecho")

const cXOR : int := 8
const cDraw : int := 16
const cMouse : int := 13
const cGrid : int := 12


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set up menu and buttons
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Create the menus

NewMenu.AddMenu ("File", 0)
NewMenu.AddItem ("New base/nonbase", "", " ", "New", 1, 0)
NewMenu.AddItem ("Open base/nonbase", "", " ", "Open", 1, 1)
NewMenu.AddItem ("Save base/nonbase", "", " ", "Save", 1, 2)
NewMenu.AddItem ("Save base/nonbase As", "", " ", "SaveAs", 1, 3)
NewMenu.AddItem ("Clear Screen", "", " ", "Clear", 1, 4)
NewMenu.AddItem ("Quit", "", " ", "Quit", 1, 5)

const edit : int := 1
const draw : int := 2
const handin : int := 3
const help : int := 4
const reprMode : int := 5
var mode := edit

NewMenu.AddMenu ("Modes", 1)
NewMenu.AddItem ("Edit", "", " ", "Edit", 2, 0)
NewMenu.AddItem ("Draw", "", " ", "Draw", 2, 1)
NewMenu.AddItem ("Hand In", "", " ", "HandIn", 2, 2)
NewMenu.AddItem ("Help", "", " ", "Help", 2, 3)

const maxEx := 11
var examples : array 1 .. maxEx of string (20)
    := init ("face", "person", "three", "leaf", "fallLine", "cake",
    "tilt", "squareR", "spiral", "recur1", "recur2")
NewMenu.AddMenu ("Examples", 2)
for i : 1 .. maxEx
    NewMenu.AddItem (examples (i), "", " ", examples (i), 3, i - 1)
end for

const maxRepr := 3
type reprType :
    record
        menu, file : string (20)
        x1, y1, x2, y2 : real
        n : int
    end record
var repr : array 1 .. maxRepr of reprType := init (
    init ("triangle", "triangle", 0.5, 0.5, 0.8, 0.5, 60),
    init ("faceS", "faceS", 0.3, 0.3, 0.3, 0.7, 60),
    init ("squareC", "squareC", 0.35, 0.5, 0.65, 0.5, 5))
NewMenu.AddMenu ("Reproduce", 3)
for i : 1 .. maxRepr
    NewMenu.AddItem (repr (i).file, "", " ", repr (i).file, 4, i - 1)
end for


% Edit Buttons

const line : int := 1
const circle : int := 2
const circleE : int := 3
const arc : int := 4
const recurse : int := 5
const delete : int := 6
const maxEdit : int := 6
var editMode := line

const size : int := 10
const border : int := 3
const buttonSize : int := size + 2 * border + 6

var editButtons : array 1 .. maxEdit of ^PicRadioButtonWidget
for i : 1 .. maxEdit
    new editButtons (i)
    editButtons (i) -> Move (0, maxy - titleBarHeight - i * buttonSize)
    editButtons (i) -> SetWidth (buttonSize)
    editButtons (i) -> SetHeight (buttonSize)
    if i not= 1 then
        editButtons (i) -> Join (editButtons (i - 1))
    end if
end for

const butR := 1
% The line
Draw.Line (border, border, border + size, border + size, black)
Draw.FillOval (border, border, butR, butR, black)
Draw.FillOval (border + size, border + size, butR, butR, black)
var pic := Pic.New (0, 0, 2 * border + size, 2 * border + size)
editButtons (line) -> SetPic (pic)
Draw.FillBox (0, 0, border + 2 * size, border + 2 * size, white)

% The circle
const radius : int := size div 2
Draw.Oval (border + radius, border + radius, radius, radius, black)
Draw.FillOval (border + radius, border + radius, butR, butR, black)
Draw.FillOval (border + radius, border + radius, butR, butR, black)
pic := Pic.New (0, 0, 2 * border + size, 2 * border + size)
editButtons (circle) -> SetPic (pic)
Draw.FillBox (0, 0, border + 2 * size, border + 2 * size, white)

% The circle edge
Draw.Oval (border + radius, border + radius, radius, radius, black)
Draw.FillOval (border, border + radius, butR, butR, black)
Draw.FillOval (border + 2 * radius, border + radius, butR, butR, black)
pic := Pic.New (0, 0, 2 * border + size, 2 * border + size)
editButtons (circleE) -> SetPic (pic)
Draw.FillBox (0, 0, border + 2 * size, border + 2 * size, white)

% The arc
Draw.Arc (border + radius, border + radius, radius, radius, 180, 360,
    black)
```

```
Draw.FillOval (border, border + radius, butR, butR, black)
Draw.FillOval (border + 2 * radius, border + radius, butR, butR, black)
Draw.FillOval (border + radius, border, butR, butR, black)
pic := Pic.New (0, 0, 2 * border + size, 2 * border + size)
editButtons (arc) -> SetPic (pic)
Draw.FillBox (0, 0, border + 2 * size, border + 2 * size, white)

% The recurse
rPic -> DrawArrow (border, border, border + size, border + size, black)
pic := Pic.New (0, 0, 2 * border + size, 2 * border + size)
editButtons (recurse) -> SetPic (pic)
Draw.FillBox (0, 0, border + 2 * size, border + 2 * size, white)

% The delete
Draw.Line (border, border, border + size, border + size, black)
Draw.Line (border, border + size, border + size, border, black)
Draw.FillOval (border + (size div 2), border + (size div 2), 3, 3, black)
pic := Pic.New (0, 0, 2 * border + size, 2 * border + size)
editButtons (delete) -> SetPic (pic)
Draw.FillBox (0, 0, border + 2 * size, border + 2 * size, white)

editMode := line
editButtons (editMode) -> Select

% The Undo Button
const undoNone := 1
const undoBase := 2
const undoNonbase := 3
const undoRec := 4
var undoMode : int := undoNone

var undoButton : ^PicButtonWidget
new undoButton
undoButton -> Move (0, maxy - titleBarHeight
    - (maxEdit + 2) * buttonSize)
undoButton -> SetWidth (buttonSize)
undoButton -> SetHeight (buttonSize)
Draw.Line (border, border, border + size, border + size, black)
Draw.Line (border, border + size, border + size, border, black)
pic := Pic.New (0, 0, 2 * border + size, 2 * border + size)
undoButton -> SetPic (pic)
Draw.FillBox (border, border, border + size, border + size, white)


% The grid buttons

const noGrid := 1
const maxGrid := 5
var gridVal : array 1 .. maxGrid of int := init (0, 4, 5, 6, 7)
var gridButtons : array 1 .. maxGrid of ^RadioWidget
var grid : int
for i : 1 .. maxGrid
    new gridButtons (i)
    if i = 1 then
        gridButtons (i) -> SetLabel ("no grid")
    else
        gridButtons (i) -> SetLabel (intstr (gridVal (i)) + " grid")
        gridButtons (i) -> Join (gridButtons (i - 1))
    end if
end for
gridButtons (1) -> Move (0, maxy - titleBarHeight
    - (maxEdit + 4) * buttonSize)
grid := gridVal (noGrid)
gridButtons (noGrid) -> Select
var roundButtons : array 1 .. 2 of ^RadioWidget
var roundGrid : boolean := false
new roundButtons (1)
roundButtons (1) -> SetLabel ("Any Where")
new roundButtons (2)
roundButtons (2) -> SetLabel ("On Grid")
roundButtons (2) -> Join (roundButtons (1))
roundButtons (1) -> Move (0, maxy - titleBarHeight
    - round ( (maxEdit + maxGrid + 3.5) * buttonSize))
roundButtons (1) -> Select


% The n buttons

const basecase := 1
const nonbasecase := 2
const maxn := 13
var n : int

var nButtons : array 1 .. maxn of ^RadioWidget
var nVal : array 1 .. maxn of int := init (1, 0, 2, 3, 4, 5, 6, 10, 20,
    30,
    40, 50, 60)
var nStr : array 1 .. maxn of string (8)
    := init ("n=base=1", "nonbase", "n=2", "n=3", "n=4", "n=5", "n=6",
    "n=10", "n=20", "n=30", "n=40", "n=50", "n=60")
for i : 1 .. maxn
    new nButtons (i)
    nButtons (i) -> SetLabel (nStr (i))
    if i not= 1 then
        nButtons (i) -> Join (nButtons (i - 1))
    end if
end for
nButtons (1) -> Move (maxx - nButtons (basecase) -> Width,
    maxy - nButtons (basecase) -> Height - titleBarHeight)
n := nVal (basecase)
nButtons (basecase) -> Select


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Redisplay the screen in the correct mode
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var dividerX : int

var gridW : real
var gridBaseX, gridNonbaseX, gridY : int

procedure DrawGrid (x1, x2, y : int)
    if grid not= 0 then
        gridW := (x2 - x1) / grid
        var gridY := grid
        for i : - 1 .. grid + 1
            Draw.Line (round (x1 + i * gridW), round (y + gridY * gridW),
                round (x1 + i * gridW), round (y - gridY * gridW), cGrid)
        end for
        for i : - gridY .. gridY
            Draw.Line (round (x1 - gridW), round (y + i * gridW),
                round (x2 + gridW), round (y + i * gridW), cGrid)
        end for
    end if
end DrawGrid

% Rounds a point to the nearest grid point
procedure GridRound (var x, y : int)
```

```
    if mode = edit and editMode ~= delete and roundGrid and grid ~= 0 then
        var gridX := gridBaseX
        if x >= dividerX then
            gridX := gridNonbaseX
        end if
        var i : int := round ( (x - gridX) / gridW)
        var j : int := round ( (y - gridY) / gridW)
        x := round (gridX + i * gridW)
        y := round (gridY + j * gridW)
    end if
end GridRound


procedure DrawBaseNonBase (x1, x2 : int)
    const r := 3
    const width : real := 0.5

    dividerX := (x1 + x2) div 2
    var edgeW : int := round ( (x2 - x1) * (1 - width) / 4)
    var baseX1 := x1 + edgeW
    var baseX2 := dividerX - edgeW
    var nonBaseX1 := dividerX + edgeW
    var nonBaseX2 := x2 - edgeW
    var y := (maxy - titleBarHeight) div 2
    Draw.FillOval (baseX1, y, r, r, cGrid)
    Draw.FillOval (baseX2, y, r, r, cGrid)
    Draw.FillOval (nonBaseX1, y, r, r, cGrid)
    Draw.FillOval (nonBaseX2, y, r, r, cGrid)
    rPic -> InitializeAdd (baseX1, y, baseX2, y, nonBaseX1, y,
        nonBaseX2, y)

    DrawGrid (baseX1, baseX2, y)
    DrawGrid (nonBaseX1, nonBaseX2, y)
    gridBaseX := baseX1
    gridNonbaseX := nonBaseX1
    gridY := y

    var del : boolean := false
    if mode = edit and editMode = delete then
        del := true
    end if
    rPic -> DrawRec (baseX1, y, baseX2, y, 1, del, cDraw)
    rPic -> DrawRec (nonBaseX1, y, nonBaseX2, y, 0, del, cDraw)
end DrawBaseNonBase

procedure Redisplay ()
    cls
    NewMenu.DrawBar
    case mode of
        label edit :
            editButtons (1) -> Show
            undoButton -> Show
            gridButtons (1) -> Show
            roundButtons (1) -> Show
            nButtons (1) -> Hide
            DrawBaseNonBase (editButtons (1) -> Width, maxx)
            Draw.Line (dividerX, 0, dividerX, maxy - titleBarHeight,
                cGrid)
        label draw :
            editButtons (1) -> Hide
            undoButton -> Hide
            gridButtons (1) -> Hide
            roundButtons (1) -> Hide
            nButtons (1) -> Show
        label handin :
            editButtons (1) -> Hide
            undoButton -> Hide
            gridButtons (1) -> Hide
            roundButtons (1) -> Hide
            nButtons (1) -> Show
            const divider := (maxx - nButtons (nonbasecase) ->
                Width) div 2
            DrawBaseNonBase (0, divider)
            Draw.Line (divider, 0, divider, maxy - titleBarHeight,
                cGrid)
        label help :
            editButtons (1) -> Hide
            undoButton -> Hide
            gridButtons (1) -> Hide
            roundButtons (1) -> Hide
            nButtons (1) -> Hide
            Text.Locate (4, 1)
            put "Read the file help.txt"
        label reprMode :
            editButtons (1) -> Hide
            undoButton -> Hide
            gridButtons (1) -> Hide
            nButtons (1) -> Hide
        label :
    end case
end Redisplay

procedure EditMode ()
    mode := edit
    editMode := line
    editButtons (editMode) -> Select
    undoMode := undoNone
    Redisplay ()
end EditMode


procedure DrawMode (newMode : int)
    mode := newMode
    if rPic -> NumRec () >= 2 and n >= 10 then
        n := nVal (5)
        nButtons (5) -> Select
    end if
    Redisplay ()
end DrawMode


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Open File
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure Open ()
    cls
    NewMenu.DrawBar
    setscreen ("cursor,echo")
    Text.Locate (4, 1)
    loop
        put "Enter file name (or done): " ..
        get fileName
        if fileName = "done" then
            rPic -> Initialize
            exit
        end if
        var preStr : array 1 .. 2 of string := init ("", "fig/")
        var postStr : array 1 .. 2 of string := init ("", ".rec")
        var suc : int
```

```
        for i : 1 .. 2
            for j : 1 .. 2
                suc := rPic -> Get (preStr (i) + fileName +
                    postStr (j))
                if suc = 0 then
                    fileName := preStr (i) + fileName + postStr (j)
                    exit
                elsif suc = 2 then
                    put "File ",
                        preStr (i) + fileName + postStr (j),
                        " had the wrong format"
                end if
            end for
            exit when suc not= 1
        end for
        if suc = 0 then
            exit
        elsif suc = 1 then
            put "Could open none of: " ..
            for i : 1 .. 2
                for j : 1 .. 2
                    put preStr (i) + fileName + postStr (j), " " ..
                end for
            end for
            put ""
        end if
    end loop
    setscreen ("nocursor,noecho")

    DrawMode (handin)
end Open

procedure SaveAs ()
    cls
    NewMenu.DrawBar
    setscreen ("cursor,echo")
    Text.Locate (4, 1)
    loop
        put "Enter file name (or done): " ..
        get fileName
        if fileName = "done" then
            fileName := ""
            exit
        end if
        exit when rPic -> Put (fileName)
        put "Could not open ", fileName
    end loop
    Redisplay ()
end SaveAs


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mouse Handler
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure DrawMouse (x, y : int)
    const l := 5
    Draw.Line (x - l, y - l, x + l, y + l, cMouse)
    Draw.Line (x - l, y + l, x + l, y - l, cMouse)
end DrawMouse

% While the mouse is held down, this is the figure that is drawn
% from point (x1,y1) to (x2,y2)   and (x3,y3) if necessary
% c = color.
% action =
const trackDown := 1 % draw while button down
const trackUp := 2 % draw while button up
const trackDraw := 3 % draw when mouse complete
const trackAdd := 4 % add to fig

procedure DrawTrack (x1, y1, x2, y2, x3, y3, c, action : int)
    var rad : int
    if action = trackDown then
        DrawMouse (x2, y2)
    elsif action = trackUp then
        DrawMouse (x3, y3)
    end if
    case mode of
        label edit :
            case editMode of
                label line :
                    if action = trackDown or action = trackDraw then
                        Draw.Line (x1, y1, x2, y2, c)
                    elsif action = trackAdd then
                        rPic -> AddLine (x1, y1, x2, y2, (x1 < dividerX))
                    end if

                label circle :
                    rad := round (sqrt ( (x2 - x1) ** 2 + (y2 - y1) ** 2))
                    if action = trackDown or action = trackDraw then
                        Draw.Oval (x1, y1, rad, rad, c)
                    elsif action = trackAdd then
                        rPic -> AddCircle (x1, y1, rad, (x1 < dividerX))
                    end if

                label circleE :
                    rad := round (
                        sqrt ( (x2 - x1) ** 2 + (y2 - y1) ** 2) / 2)
                    if action = trackDown or action = trackDraw then
                        Draw.Oval ( (x1 + x2) div 2, (y1 + y2) div 2, rad,
                            rad, c)
                    elsif action = trackAdd then
                        rPic -> AddCircle ( (x1 + x2) div 2, (y1 + y2) div 2,
                            rad, (x1 < dividerX))
                    end if

                label arc :
                    if action = trackDown then
                        Draw.Line (x1, y1, x2, y2, c)
                    else
                        const radDot : int := 2
                        var xc, yc, ang1, ang2 : int
                        var isLine : boolean
                        Center (x1, y1, x2, y2, x3, y3, xc, yc, rad,
                            ang1, ang2, isLine)
                        if not isLine then
                            if action = trackUp then
                                Draw.Arc (xc, yc, rad, rad, ang1, ang2, c)
                                Draw.FillOval (xc, yc, radDot, radDot, c)
                            elsif action = trackDraw then
                                Draw.Arc (xc, yc, rad, rad, ang1, ang2, c)
                            elsif action = trackAdd then
                                rPic -> AddArc (xc, yc, rad, ang1, ang2,
                                    (x1 < dividerX))
                            end if
                        else
                            if action = trackUp or action = trackDraw then
                                Draw.Line (x1, y1, x2, y2, c)
                            elsif action = trackAdd then
                                rPic -> AddLine (x1, y1, x2, y2, (x1 <
```

```
                                dividerX))
                        end if
                    end if
                end if


            label recurse :
                if x1 >= dividerX then
                    if action = trackDown or action = trackDraw then
                        rPic -> DrawArrow (x1, y1, x2, y2, c)
                    elsif action = trackAdd then
                        rPic -> AddRecurse (x1, y1, x2, y2)
                    end if
                end if

            label delete :
                if action = trackAdd then
                    if rPic -> Delete (x1, y1) then
                        Redisplay ()
                    end if
                end if

            label :
        end case

    label draw, handin :
        var nn : int := n
        if nn not= 1 then
            nn := 0
        end if
        if action = trackDown then
            rPic -> DrawRec (x1, y1, x2, y2, nn, false, c)
        elsif action = trackDraw then
            rPic -> DrawRec (x1, y1, x2, y2, n, false, c)
        end if
    label help :
    label :
  end case
end DrawTrack


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Event Handler
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Redisplay ()
var mx, my, mx2, my2, mb : int
Mouse.Where (mx, my, mb)
setscreen ("xor")
DrawMouse (mx, my)
setscreen ("noxor")
loop
    Mouse.Where (mx2, my2, mb)
    if mx ~= mx2 or my ~= my2 then
        setscreen ("xor")
        DrawMouse (mx, my) % erase mouse
        DrawMouse (mx2, my2) % redraw mouse
        setscreen ("noxor")
        mx := mx2
        my := my2
    end if

    const event := GUIControl.GetEvent
    case event.kind of
        label "menu" :
            case event.value of
                label "New" :
                    rPic -> Initialize ()
                    EditMode ()
                label "Open" :
                    Open ()
                label "Save" :
                    if fileName not= "" then
                        var s := rPic -> Put (fileName)
                    else
                        SaveAs ()
                    end if
                label "SaveAs" :
                    SaveAs ()
                label "Clear" :
                    Redisplay ()
                label "Quit" :
                    window -> Close
                    exit

                label "Edit" :
                    EditMode ()
                label "Draw" :
                    DrawMode (draw)
                label "HandIn" :
                    DrawMode (handin)
                label "Help" :
                    mode := help
                    Redisplay ()

                label :
            end case

            % Example Menu
            var e : int := 1
            loop
                exit when e > maxEx
                exit when event.value = examples (e)
                e += 1
            end loop
            if e <= maxEx then
                if rPic -> Get ("fig/" + examples (e) + ".rec") = 0 then
                    fileName := ""
                    DrawMode (handin)
                else
                    Text.Locate (4, 1)
                    put "Could not open."
                end if
            end if

            % Reproduce Menu
            e := 1
            loop
                exit when e > maxRepr
                exit when event.value = repr (e).file
                e += 1
            end loop
            if e <= maxRepr then
                mode := reprMode
                Redisplay ()
                if reprPic -> Get ("fig/" + repr (e).file + ".rec") = 0 then
                    reprPic -> DrawRec (
                        round (repr (e).x1 * maxx),
                        round (repr (e).y1 * maxy),
                        round (repr (e).x2 * maxx),
```

```
                round (repr (e).y2 * maxy),
                repr (e).n, false, cDraw)
            reprPic -> Initialize ()
        else
            Text.Locate (4, 1)
            put "Could not open."
        end if
    end if

label "widget" :

    % Check Edit Buttons
    var newMode : int := 1
    loop
        exit when newMode > maxEdit
        exit when event.widget = editButtons (newMode)
        newMode += 1
    end loop
    if newMode <= maxEdit then
        if (editMode not= delete and newMode = delete) or
                (editMode = delete and newMode not= delete)
                then
            editMode := newMode
            Redisplay ()
        end if
        editMode := newMode
    end if

    % Check Undo Button
    if event.widget = undoButton then
        rPic -> Undo (undoMode)
        undoMode := undoNone
        Redisplay ()
    end if

    % Check Grid Buttons
    newMode := 1
    loop
        exit when newMode > maxGrid
        exit when event.widget = gridButtons (newMode)
        newMode += 1
    end loop
    if newMode <= maxGrid then
        grid := gridVal (newMode)
        Redisplay ()
    end if

    % Check round Buttons
    if event.widget = roundButtons (1) then
        roundGrid := false
    elsif event.widget = roundButtons (2) then
        roundGrid := true
    end if

    % Check n Buttons
    newMode := 1
    loop
        exit when newMode > maxn
        exit when event.widget = nButtons (newMode)
        newMode += 1
    end loop
    if newMode <= maxn then
        n := nVal (newMode)
        var numR := rPic -> NumRec ()
        if numR >= 2 and n >= 10 then
            Text.Locate (10, 1)
            put "You are asking to print the base case ",
                numR, "**", n - 1, " times."
            put "We recommend using a smaller value of n"
            put "Hit any key and <ret> to continue"
            var ch : string (1)
            getch (ch)
            Redisplay ()
            newMode := 5
            n := nVal (newMode)
            nButtons (newMode) -> Select
        end if
    end if

label "buttondown" :
    % Mouse button down, but not on a widget
    setscreen ("xor")
    DrawMouse (mx, my) % erase mouse
    var x1, y1, x2, y2, x3, y3, xnew, ynew, b1, b2 : int
    str2int (event.value, x1, y1)
    GridRound (x1, y1)
    x2 := x1
    y2 := y1
    DrawTrack (x1, y1, x2, y2, 0, 0, cXOR, trackDown)
    loop
        exit when Mouse.ButtonMoved ("up")
        Mouse.Where (xnew, ynew, b1)
        GridRound (xnew, ynew)
        if x2 ~= xnew or y2 ~= ynew then
            % erase
            DrawTrack (x1, y1, x2, y2, 0, 0, cXOR, trackDown)
            % redraw
            DrawTrack (x1, y1, xnew, ynew, 0, 0, cXOR, trackDown)
            x2 := xnew
            y2 := ynew
        end if
    end loop
    % Final erase of track figure
    DrawTrack (x1, y1, x2, y2, 0, 0, cXOR, trackDown)
    Mouse.ButtonWait ("up", x2, y2, b1, b2)
    GridRound (x2, y2)

    x3 := x2
    y3 := y2
    if mode = edit and editMode = arc then
        DrawTrack (x1, y1, x2, y2, x3, y3, cXOR, trackUp)
        loop
            exit when Mouse.ButtonMoved ("down")
            Mouse.Where (xnew, ynew, b1)
            if x3 ~= xnew or y3 ~= ynew then
                % erase
                DrawTrack (x1, y1, x2, y2, x3, y3, cXOR, trackUp)
                % Redraw
                DrawTrack (x1, y1, x2, y2, xnew, ynew, cXOR,
                    trackUp)
                x3 := xnew
                y3 := ynew
            end if
        end loop
        % Final erase of track figure
        DrawTrack (x1, y1, x2, y2, x3, y3, cXOR, trackUp)
        Mouse.ButtonWait ("down", x3, y3, b1, b2)
    end if

    setscreen ("noxor")
```

```
            % Draw final version
            DrawTrack (x1, y1, x2, y2, x3, y3, cDraw, trackDraw)
            DrawTrack (x1, y1, x2, y2, x3, y3, 0, trackAdd)

            if mode = edit then
                case editMode of
                    label delete :
                        undoMode := undoNone
                    label recurse :
                        undoMode := undoRec
                    label :
                        if x1 < dividerX then
                            undoMode := undoBase
                        else
                            undoMode := undoNonbase
                        end if
                end case
            end if

            setscreen ("xor")
            Mouse.Where (mx, my, b1)
            DrawMouse (mx, my) % Draw first outter loop mouse
            setscreen ("noxor")

        label "keystroke" :
            % Keystroke received
            % Text.Locate (1, 1)
            % put event.value
        label "" :
            % Nothing happened

    end case
end loop




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  File:     recPic.tu
%  Class:    RecursivePicture
%  Purpose:  Draws a recursive picture specified from a base and non-base
%            case figure.
%  Author:   Jeff Edmonds
%  Location: York University
%  Email:    jeff@cs.yorku.ca
%  Date:     Jan, 1996
% --------------------------------

unit
class RecursivePicture

    export Initialize, Get, Put, InitializeAdd, AddLine, AddCircle,
        AddArc, AddRecurse, Delete, Undo, NumRec, DrawRec, DrawArrow

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % The following variable types specify points, lengths, angles,
    % orientations, and handles within the Spec coordinate system.
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    type pntInSpecType :
        record
            x, y : real
        end record

    type lengInSpecType : real

    type angInSpecType : real

    type orientInSpecType :
        record
            p1, p2 : pntInSpecType
        end record

    type handleInSpecType :
        record
            p1, p2 : pntInSpecType
        end record

    % The following variable types specify points, lengths, angles, and
    % orientations within the Pic coordinate system.

    type pntInPicType :
        record
            x, y : real
        end record

    type lengInPicType : real

    type angInPicType : real

    type orientInPicType :
        record
            p1, p2 : pntInPicType
        end record


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %-The following routines expand, rotate and translate points, lengths,
    %-angles, and orientations from the Spec coordinate system to the Pic
    %-coordinate system. The routines each have three input parameters.  The
    %-second parameter is the variable "handle". It specifies the handle
    %-orientation within the Spec coordinate system (found in first line of
    %-the file). The second parameter is the variable "where". It specifies
    %-the orientation within the Pic coordinate system. The Spec coordinate
    %-system is expanded, rotated, and translated so that the two handle
    %-points are moved to the two where points.  The first parameter to each
    %-routine is a point, length, angle, or orientation as the case may
    %-be. It is in the Spec coordinate system. As the Spec coordinate
    %-system is expanded, rotated, and translated, this objected is
    %-expanded, rotated, and translated too. The output of the routine is
    %-the resulting object.
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    function Pnt_SpecToPic (pntInSpec : pntInSpecType,
            handle : handleInSpecType,
            where : orientInPicType)
            : pntInPicType

        % Translate everything so that p1 of where and handle are at the
        % origin
        var v1, pnt : pntInSpecType
        var v2 : pntInPicType
        v2.x := where.p2.x - where.p1.x
        v2.y := where.p2.y - where.p1.y
        v1.x := handle.p2.x - handle.p1.x
        v1.y := handle.p2.y - handle.p1.y
        pnt.x := pntInSpec.x - handle.p1.x
        pnt.y := pntInSpec.y - handle.p1.y

        % Calculate the factors by which to rotate v1 to v2
```

```
    var l1Sqr := (v1.x) ** 2 + (v1.y) ** 2
    var v1DOTv2 := v1.x * v2.x + v1.y * v2.y
    var v1WIERDv2 := v1.x * v2.y - v1.y * v2.x
    var lCosA := v1DOTv2 / l1Sqr
    var lSinA := v1WIERDv2 / l1Sqr

    % Rotate pnt by factor
    var pntInPic : pntInPicType
    pntInPic.x := lCosA * pnt.x - lSinA * pnt.y
    pntInPic.y := lSinA * pnt.x + lCosA * pnt.y

    % Translate the point back.
    pntInPic.x += where.p1.x
    pntInPic.y += where.p1.y

    % put "handle: (", handle.p1.x, ",", handle.p1.y, ") (",
    %         handle.p2.x, ",", handle.p2.y, ")"
    % put "where: (", where.p1.x, ",", where.p1.y, ") (",
    %         where.p2.x, ",", where.p2.y, ")"
    % put "translate: (", pntInSpec.x, ",", pntInSpec.y, ") (",
    %         pntInPic.x, ",", pntInPic.y, ")"

    result pntInPic
end Pnt_SpecToPic


function Leng_SpecToPic (lengInSpec : lengInSpecType,
        handle : handleInSpecType,
        where : orientInPicType)
        : lengInPicType

    var l1 := sqrt ( (handle.p2.x - handle.p1.x) ** 2
        + (handle.p2.y - handle.p1.y) ** 2)
    var l2 := sqrt ( (where.p2.x - where.p1.x) ** 2
        + (where.p2.y - where.p1.y) ** 2)
    result lengInSpec * (l2 / l1)
end Leng_SpecToPic


function Ang_SpecToPic (angInSpec : angInSpecType,
        handle : handleInSpecType,
        where : orientInPicType)
        : angInPicType

    % Translate everything so that p1 of where and handle are
    % at the origin
    var v1, pnt : pntInSpecType
    var v2 : pntInPicType
    v2.x := where.p2.x - where.p1.x
    v2.y := where.p2.y - where.p1.y
    v1.x := handle.p2.x - handle.p1.x
    v1.y := handle.p2.y - handle.p1.y

    % Calculate the angle from v1 to v2
    var l1 := sqrt ( (v1.x) ** 2 + (v1.y) ** 2)
    var l2 := sqrt ( (v2.x) ** 2 + (v2.y) ** 2)
    var v1DOTv2 := v1.x * v2.x + v1.y * v2.y
    var v1WIERDv2 := v1.x * v2.y - v1.y * v2.x
    var ang : int
    if v1DOTv2 = 0 then
        if v1WIERDv2 > 0 then
            ang := 90
        else
            ang := 270
        end if
    else
        % oot does not have an arccos program!
        var tanA := v1WIERDv2 / v1DOTv2
        ang := round (arctand (tanA))
        if v1DOTv2 < 0 then
            ang += 180
        end if
    end if

    result angInSpec + ang

    % put "handle: (", handle.p1.x, ",", handle.p1.y, ") (",
    %     handle.p2.x, ",", handle.p2.y, ")"
    % put "where: (", where.p1.x, ",", where.p1.y, ") (",
    %     where.p2.x, ",", where.p2.y, ")"
    % put "translate: from ", angInSpec, ", by ", ang,
    %     ", to ", angInSpec + ang

end Ang_SpecToPic


function Orient_SpecToPic (orientInSpec : orientInSpecType,
        handle : handleInSpecType,
        where : orientInPicType)
        : orientInPicType
    var orientInPic : orientInPicType
    orientInPic.p1 := Pnt_SpecToPic (orientInSpec.p1, handle, where)
    orientInPic.p2 := Pnt_SpecToPic (orientInSpec.p2, handle, where)
    result orientInPic
end Orient_SpecToPic



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This section defines the data structure for storing the data
% from a RecFile.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

type objTypes : enum (line, circle, arc, recurse)

type objRec :
    record
        typeOf : objTypes
        p1, p2 : pntInSpecType
        r : lengInSpecType
        a1, a2 : angInSpecType
        del : pntInPicType % pnt at which to delete obj
    end record

const maxObjects := 200

type objListType :
    record
        n : int
        list : array 1 .. maxObjects of objRec
    end record

type recRec :
    record
        where : orientInSpecType
        del : pntInPicType % pnt at which to delete obj
    end record

type recListType :
    record
```

```
        n : int
        list : array 1 .. maxObjects of recRec
    end record

type recPictureType :
    record
        handle : handleInSpecType
        base, nonbase : objListType
        recurse : recListType
    end record

var recPicture, recArrow : recPictureType
% Used to draw the arrow for n=0

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure ClearPicture (var recPic : recPictureType)
    recPic.handle.p1.x := 0
    recPic.handle.p1.y := 0
    recPic.handle.p2.x := 1
    recPic.handle.p2.y := 0
    recPic.base.n := 0
    recPic.nonbase.n := 0
    recPic.recurse.n := 0
end ClearPicture

procedure Initialize ()
    var obj : objRec

    ClearPicture (recPicture)
    ClearPicture (recArrow)

    obj.typeOf := objTypes.line
    obj.p1.x := 0.0
    obj.p1.y := 0.0
    obj.p2.x := 0.3
    obj.p2.y := 0.0
    recArrow.base.list (1) := obj

    obj.p1.x := 0.25
    obj.p1.y := 0.05
    obj.p2.x := 0.3
    obj.p2.y := 0.0
    recArrow.base.list (2) := obj

    obj.p1.x := 0.25
    obj.p1.y := - 0.05
    obj.p2.x := 0.3
    obj.p2.y := 0.0
    recArrow.base.list (3) := obj

    obj.p1.x := 0.7
    obj.p1.y := 0.0
    obj.p2.x := 1.0
    obj.p2.y := 0.0
    recArrow.base.list (4) := obj

    obj.p1.x := 0.7
    obj.p1.y := 0.0
    obj.p2.x := 0.65
    obj.p2.y := 0.05
    recArrow.base.list (5) := obj

    obj.p1.x := 0.7
    obj.p1.y := 0.0
    obj.p2.x := 0.65
    obj.p2.y := - 0.05
    recArrow.base.list (6) := obj

    recArrow.base.n := 6
end Initialize


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This section read the data from a RecFile
%
% GetObjs reads from a file the following
%     { Object }*
%     "done"
%
% GetRecurse reads from a file the following
%     { "recurse Where }*
%     "done"
%
% Get reads the whole file
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure GetObjs (fileNumber : int, var objs : objListType)

    var command : string (20)
    var obj : objRec

    objs.n := 0
    loop
        objs.n += 1
        get : fileNumber, command
        case command of
            label "line" :
                obj.typeOf := objTypes.line
                get : fileNumber,
                    obj.p1.x, obj.p1.y, obj.p2.x, obj.p2.y
            label "circle" :
                obj.typeOf := objTypes.circle
                get : fileNumber,
                    obj.p1.x, obj.p1.y, obj.r
            label "arc" :
                obj.typeOf := objTypes.arc
                get : fileNumber,
                    obj.p1.x, obj.p1.y, obj.r, obj.a1, obj.a2
            label "done" :
                objs.n -= 1
                return
            label :
                put "File formated incorrectly: "
                quit
        end case
        objs.list (objs.n) := obj
    end loop
end GetObjs


procedure GetRecurse (fileNumber : int, var recurse : recListType)

    var command : string (20)
    var where : orientInSpecType

    recurse.n := 0
    loop
        get : fileNumber, command
```

```
        if command = "recurse" then
            get : fileNumber,
                where.p1.x, where.p1.y, where.p2.x, where.p2.y
            recurse.n += 1
            recurse.list (recurse.n).where := where

        elsif command = "done" then
            return

        else
            put "File formated incorrectly: "
            quit
        end if
    end loop
end GetRecurse


function Get (fileName : string) : int
    % 0 fine so far, 1 cant open, 2 bad format
    var fileNumber : int % used for reading in file
    var command : string (20)

    open : fileNumber, fileName, get
    if fileNumber = 0 then
        result 1
    end if

    get : fileNumber, command
    if command not= "handle" then
        result 2
    end if

    get : fileNumber,
        recPicture.handle.p1.x, recPicture.handle.p1.y,
        recPicture.handle.p2.x, recPicture.handle.p2.y

    GetObjs (fileNumber, recPicture.base)
    GetObjs (fileNumber, recPicture.nonbase)
    GetRecurse (fileNumber, recPicture.recurse)
    close : fileNumber
    result 0
end Get



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This section writes the data to a RecFile
%
% PutObjs writes to a file the following
%     { Object }*
%     "done"
%
% Putrecurse writes to a file the following
%     { "recurse Where }*
%     "done"
%
% Put writes the whole file
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure PutObjs (fileNumber : int, objs : objListType)
    var obj : objRec
    for i : 1 .. objs.n
        obj := objs.list (i)
        case obj.typeOf of
            label objTypes.line :
                put : fileNumber, "line " ..
                put : fileNumber,
                    obj.p1.x, " ", obj.p1.y, " ", obj.p2.x, " ", obj.p2.y
            label objTypes.circle :
                put : fileNumber, "circle " ..
                put : fileNumber,
                    obj.p1.x, " ", obj.p1.y, " ", obj.r
            label objTypes.arc :
                put : fileNumber, "arc " ..
                put : fileNumber,
                    obj.p1.x, " ", obj.p1.y, " ",
                    obj.r, " ", obj.a1, " ", obj.a2
            label :
        end case
    end for
    put : fileNumber, "done"
end PutObjs


procedure PutRecurse (fileNumber : int, recurse : recListType)
    var where : orientInSpecType
    for i : 1 .. recurse.n
        where := recurse.list (i).where
        put : fileNumber, "recurse " ..
        put : fileNumber,
            where.p1.x, " ", where.p1.y, " ", where.p2.x, " ", where.p2.y
    end for
    put : fileNumber, "done"
end PutRecurse


function Put (fileName : string) : boolean
    var fileNumber : int % used for writing to the file
    open : fileNumber, fileName, put
    if fileNumber = 0 then
        result false
    end if
    put : fileNumber, "handle " ..
    put : fileNumber,
        recPicture.handle.p1.x, " ", recPicture.handle.p1.y," ",
        recPicture.handle.p2.x, " ", recPicture.handle.p2.y
    PutObjs (fileNumber, recPicture.base)
    PutObjs (fileNumber, recPicture.nonbase)
    PutRecurse (fileNumber, recPicture.recurse)
    close : fileNumber
    result true
end Put



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section rotates pnts, lengs and angles from Pic to Spec.
% It, however, uses the Spec to Pic routines by fudging the types.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var handleInPic : array 1 .. 2 of handleInSpecType
% really should be type orientInPicType

procedure InitializeAdd (
        x1base, y1base, x2base, y2base,
        x1nonbase, y1nonbase, x2nonbase, y2nonbase : int)
    handleInPic (1).p1.x := x1base
```

```
    handleInPic (1).p1.y := y1base
    handleInPic (1).p2.x := x2base
    handleInPic (1).p2.y := y2base
    handleInPic (2).p1.x := x1nonbase
    handleInPic (2).p1.y := y1nonbase
    handleInPic (2).p2.x := x2nonbase
    handleInPic (2).p2.y := y2nonbase
end InitializeAdd

function Pnt_PicToSpec (x, y : int, baseCase : boolean) : pntInSpecType
    var pntInPic : pntInSpecType % really pntInPicType
    var pntInSpecTemp : pntInPicType % really pntInSpecType
    var pntInSpec : pntInSpecType
    var handleInSpec : orientInPicType % really handleInSpecType

    var i : int
    if baseCase then
        i := 1
    else
        i := 2
    end if

    pntInPic.x := x
    pntInPic.y := y

    handleInSpec.p1.x := recPicture.handle.p1.x
    handleInSpec.p1.y := recPicture.handle.p1.y
    handleInSpec.p2.x := recPicture.handle.p2.x
    handleInSpec.p2.y := recPicture.handle.p2.y

    pntInSpecTemp := Pnt_SpecToPic (pntInPic,
        handleInPic (i), handleInSpec)

    pntInSpec.x := pntInSpecTemp.x
    pntInSpec.y := pntInSpecTemp.y
    result pntInSpec
end Pnt_PicToSpec


function Leng_PicToSpec (leng : int, baseCase : boolean) : lengInSpecType
    var lengInPic : lengInSpecType % really lengInPicType
    var lengInSpecTemp : lengInPicType % really lengInSpecType
    var lengInSpec : lengInSpecType
    var handleInSpec : orientInPicType % really handleInSpecType

    var i : int
    if baseCase then
        i := 1
    else
        i := 2
    end if

    lengInPic := leng

    handleInSpec.p1.x := recPicture.handle.p1.x
    handleInSpec.p1.y := recPicture.handle.p1.y
    handleInSpec.p2.x := recPicture.handle.p2.x
    handleInSpec.p2.y := recPicture.handle.p2.y

    lengInSpecTemp := Leng_SpecToPic (lengInPic,
        handleInPic (i), handleInSpec)

    lengInSpec := lengInSpecTemp
    result lengInSpec
end Leng_PicToSpec


function Ang_PicToSpec (ang : int, baseCase : boolean) : angInSpecType
    var angInPic : angInSpecType % really angInPicType
    var angInSpecTemp : angInPicType % really angInSpecType
    var angInSpec : angInSpecType
    var handleInSpec : orientInPicType % really handleInSpecType

    var i : int
    if baseCase then
        i := 1
    else
        i := 2
    end if

    angInPic := ang

    handleInSpec.p1.x := recPicture.handle.p1.x
    handleInSpec.p1.y := recPicture.handle.p1.y
    handleInSpec.p2.x := recPicture.handle.p2.x
    handleInSpec.p2.y := recPicture.handle.p2.y

    angInSpecTemp := Ang_SpecToPic (angInPic,
        handleInPic (i), handleInSpec)

    angInSpec := angInSpecTemp
    result angInSpec
end Ang_PicToSpec


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section adds objects
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure AddLine (x1, y1, x2, y2 : int, baseCase : boolean)
    var obj : objRec
    obj.typeOf := objTypes.line
    obj.p1 := Pnt_PicToSpec (x1, y1, baseCase)
    obj.p2 := Pnt_PicToSpec (x2, y2, baseCase)

    if baseCase then
        recPicture.base.n += 1
        recPicture.base.list (recPicture.base.n) := obj
    else
        recPicture.nonbase.n += 1
        recPicture.nonbase.list (recPicture.nonbase.n) := obj
    end if
end AddLine


procedure AddCircle (x, y, r : int, baseCase : boolean)
    var obj : objRec
    obj.typeOf := objTypes.circle
    obj.p1 := Pnt_PicToSpec (x, y, baseCase)
    obj.r := Leng_PicToSpec (r, baseCase)

    if baseCase then
        recPicture.base.n += 1
        recPicture.base.list (recPicture.base.n) := obj
    else
        recPicture.nonbase.n += 1
        recPicture.nonbase.list (recPicture.nonbase.n) := obj
    end if
```

```
end AddCircle


procedure AddArc (x, y, r, a1, a2 : int, baseCase : boolean)
    var obj : objRec
    obj.typeOf := objTypes.arc
    obj.p1 := Pnt_PicToSpec (x, y, baseCase)
    obj.r := Leng_PicToSpec (r, baseCase)
    obj.a1 := Ang_PicToSpec (a1, baseCase)
    obj.a2 := Ang_PicToSpec (a2, baseCase)

    if baseCase then
        recPicture.base.n += 1
        recPicture.base.list (recPicture.base.n) := obj
    else
        recPicture.nonbase.n += 1
        recPicture.nonbase.list (recPicture.nonbase.n) := obj
    end if
end AddArc


procedure AddRecurse (x1, y1, x2, y2 : int)
    var where : orientInSpecType
    where.p1 := Pnt_PicToSpec (x1, y1, false)
    where.p2 := Pnt_PicToSpec (x2, y2, false)
    recPicture.recurse.n += 1
    recPicture.recurse.list (recPicture.recurse.n).where := where
end AddRecurse


const delTol : real := 16

function DeleteObjs (var objs : objListType, x, y : int) : boolean
    var del := false
    for i : 1 .. objs.n
        if (objs.list (i).del.x - x) ** 2
                + (objs.list (i).del.y - y) ** 2 <= delTol then
            for j : i + 1 .. objs.n
                objs.list (j - 1) := objs.list (j)
            end for
            objs.n -= 1
            del := true
        end if
    end for
    result del
end DeleteObjs

function DeleteRec (var objs : recListType, x, y : int) : boolean
    var del := false
    for i : 1 .. objs.n
        if (objs.list (i).del.x - x) ** 2
                + (objs.list (i).del.y - y) ** 2 <= delTol then
            for j : i + 1 .. objs.n
                objs.list (j - 1) := objs.list (j)
            end for
            objs.n -= 1
            del := true
        end if
    end for
    result del
end DeleteRec

function Delete (x, y : int) : boolean
    result
        DeleteObjs (recPicture.base, x, y) or
        DeleteObjs (recPicture.nonbase, x, y) or
        DeleteRec (recPicture.recurse, x, y)
end Delete

const undoNone := 1
const undoBase := 2
const undoNonbase := 3
const undoRec := 4

procedure Undo (undoMode : int)
    case undoMode of
        label undoNone :
        label undoBase :
            recPicture.base.n -= 1
        label undoNonbase :
            recPicture.nonbase.n -= 1
        label undoRec :
            recPicture.recurse.n -= 1
        label :
    end case
end Undo

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section draws the objects
% DrawObjs rotates the objects and draws them on the screen
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure DrawLine (pntInPic1, pntInPic2 : pntInPicType, col : int)
    var x1 := round (pntInPic1.x)
    var y1 := round (pntInPic1.y)
    var x2 := round (pntInPic2.x)
    var y2 := round (pntInPic2.y)
    Draw.Line (x1, y1, x2, y2, col)
end DrawLine


procedure DrawCircle (pntInPic : pntInPicType,
        radiusInPic : lengInPicType, col : int)
    var x := round (pntInPic.x)
    var y := round (pntInPic.y)
    var r := round (radiusInPic)
    Draw.Oval (x, y, r, r, col)
end DrawCircle


procedure DrawArc (pntInPic : pntInPicType,
        radiusInPic : lengInPicType,
        angInPic1, angInPic2 : angInPicType, col : int)
    var x := round (pntInPic.x)
    var y := round (pntInPic.y)
    var r := round (radiusInPic)
    var a1 := round (angInPic1)
    var a2 := round (angInPic2)
    Draw.Arc (x, y, r, r, a1, a2, col)
end DrawArc


procedure DrawDelPnt (pntInPic : pntInPicType, col : int)
    const r := 3
    var x := round (pntInPic.x)
    var y := round (pntInPic.y)
    Draw.FillOval (x, y, r, r, col)
end DrawDelPnt
```

```
procedure DrawObjs (var objs : objListType,
        handle : handleInSpecType,
        where : orientInPicType, del : boolean, col : int)

    var objInSpec : objRec

    var pntInPic1, pntInPic2 : pntInPicType
    var radiusInPic : lengInPicType
    var angInPic1, angInPic2 : angInPicType
    for i : 1 .. objs.n
        objInSpec := objs.list (i)
        case objInSpec.typeOf of
            label objTypes.line :
                pntInPic1 := Pnt_SpecToPic (objInSpec.p1, handle,
                    where)
                pntInPic2 := Pnt_SpecToPic (objInSpec.p2, handle,
                    where)
                DrawLine (pntInPic1, pntInPic2, col)
                if del then
                    objs.list (i).del.x
                        := (pntInPic1.x + pntInPic2.x) / 2
                    objs.list (i).del.y
                        := (pntInPic1.y + pntInPic2.y) / 2
                    DrawDelPnt (objs.list (i).del, col)
                end if

            label objTypes.circle :
                pntInPic1 := Pnt_SpecToPic (objInSpec.p1, handle,
                    where)
                radiusInPic := Leng_SpecToPic (objInSpec.r, handle,
                    where)
                DrawCircle (pntInPic1, radiusInPic, col)
                if del then
                    objs.list (i).del.x := pntInPic1.x
                    objs.list (i).del.y := pntInPic1.y
                    DrawDelPnt (objs.list (i).del, col)
                end if

            label objTypes.arc :
                pntInPic1 := Pnt_SpecToPic (objInSpec.p1, handle,
                    where)
                radiusInPic := Leng_SpecToPic (objInSpec.r, handle,
                    where)
                angInPic1 := Ang_SpecToPic (objInSpec.a1, handle,
                    where)
                angInPic2 := Ang_SpecToPic (objInSpec.a2, handle,
                    where)
                DrawArc (pntInPic1, radiusInPic, angInPic1,
                    angInPic2, col)
                if del then
                    var a : real := (angInPic1 + angInPic2) / 2
                    objs.list (i).del.x := pntInPic1.x + cosd (a) *
                        radiusInPic
                    objs.list (i).del.y := pntInPic1.y + sind (a) *
                        radiusInPic
                    DrawDelPnt (objs.list (i).del, col)
                end if
        end case
    end for
end DrawObjs


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section draws the recursive figure and an arrow (for recursion)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function NumRec () : int
    result recPicture.recurse.n
end NumRec


procedure DrawArrow (x1, y1, x2, y2, col : int)
    var where : orientInPicType
    where.p1.x := x1
    where.p1.y := y1
    where.p2.x := x2
    where.p2.y := y2
    DrawObjs (recArrow.base, recArrow.handle, where, false, col)
end DrawArrow


procedure DrawRecursive (where : orientInPicType, n : int,
        del : boolean, col : int)
    var whereInSpec : orientInSpecType
    var whereInPic : orientInPicType

    if n <= 0 then
        DrawObjs (recPicture.nonbase, recPicture.handle, where, del, col)

        for i : 1 .. recPicture.recurse.n
            whereInSpec := recPicture.recurse.list (i).where
            whereInPic := Orient_SpecToPic (whereInSpec,
                recPicture.handle,
                where)
            DrawObjs (recArrow.base, recArrow.handle,
                whereInPic, false, col)
            if del then
                recPicture.recurse.list (i).del.x
                    := (whereInPic.p1.x + whereInPic.p2.x) / 2
                recPicture.recurse.list (i).del.y
                    := (whereInPic.p1.y + whereInPic.p2.y) / 2
                DrawDelPnt (recPicture.recurse.list (i).del, col)
            end if
        end for
    elsif n = 1 then
        DrawObjs (recPicture.base, recPicture.handle, where, del, col)
    else
        DrawObjs (recPicture.nonbase, recPicture.handle, where, del, col)

        for i : 1 .. recPicture.recurse.n
            whereInSpec := recPicture.recurse.list (i).where
            whereInPic := Orient_SpecToPic (whereInSpec,
                recPicture.handle,
                where)
            DrawRecursive (whereInPic, n - 1, false, col)
        end for
    end if
end DrawRecursive


procedure DrawRec (x1, y1, x2, y2, n : int, del : boolean, col : int)
    var where : orientInPicType
    where.p1.x := x1
    where.p1.y := y1
    where.p2.x := x2
    where.p2.y := y2
```

```
        DrawRecursive (where, n, del, col)
    end DrawRec

end RecursivePicture
```