☒ **Gr. 1,** Dr. G. Kronberger **Name** PAPESH Konstantin **Aufwand in h** 12

☐ **Gr. 2,** Dr. H. Gruber

☐ **Gr. 3,** Dr. D. Auer **Punkte** _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

Spielbergs *Cartoon Studio* 1.0 (24 Punkte)

Stephen Spielberg hat endgültig genug von gierigen, launischen SchauspielerInnen und will nur noch Zeichentrickfilme (*cartoons*) machen. Nachdem sich Computeranimationen aber in Spielfilmen bewährt haben, will er seine Figuren nicht händisch zeichnen, sondern von Ihnen ein oo Programm entwickeln lassen, welches aus einer textuellen Beschreibung durch Interpretation den Trickfilm generiert. Dazu hat er sich auch schon eine Sprache einfallen lassen. Hier die Grammatik dafür:

```
Cartoon =        Statement { ';' Statement } .

Statement =      ident '=' ShapeDef        |
                 ( 'SHOW' | 'HIDE' ) ident |
                 Action .

ShapeDef =       'LINE'       number number number number |
                 'RECTANGLE'  number number number number |
                 'CIRCLE'     number number number        |
                 'PICTURE'    ident { '+' ident } .

Action =         'MOVE' ident number number .
```

Man kann damit vier Arten von grafischen Objekten (engl. *shapes*) benennen und definieren:

1. *Linien* durch Angabe der (*x*, *y*)-Koordinaten des Anfangs- und des Endpunkts,
2. *Rechtecke* durch Angabe der (*x*, *y*)-Koordinaten der Eckpunkte links-oben sowie rechts-unten,
3. *Kreise* durch Angabe der (*x*, *y*)-Koordinate des Mittelpunkts und des Radius und
4. *Bilder* durch Nennung aller zu einem Bild gehörenden grafischen Objekte.

Jedes grafische Objekt hat einen eindeutigen Namen, mit dem es später angezeigt (*SHOW*), gelöscht (*HIDE*), transformiert oder zu einem Bild hinzugefügt werden kann.

Für die (vorerst einzige) Verschiebeaktion ist mit zwei Werten anzugeben, wie weit das grafische Objekt auf der *x*- bzw. auf der *y*-Achse verschoben werden soll.

Testen Sie Ihr Programm ausführlich, indem Sie einen kleinen Trickfilm herstellen.

Mögliche Erweiterung: Falls Sie Zeit und Lust haben, können Sie sich weitere Aktionen ausdenken und diese dann noch implementieren.

ADF2x & PRO2X Algorithmen & Datenstrukturen und OOP – SS 2018
Übungsabgabe 10

Konstantin Papesh

20. Juni 2018

10.1 Spielbergs *Cartoon Studio 1.0*

10.1.1 Lösungsidee

Im Grunde werden zwei völlig verschiedene Module geschrieben und diese dann ineinander verwebt. Zum einen braucht es einen Parser für die EBNF-Grammatik, um *Videofiles* einlesen zu können, zum anderen einen grafischen Handler um die gegebenen Objekte darstellen zu können. Hierbei wird *WinGraph* eingesetzt, welches eine einfache Implementierung ermöglicht.

EBNF-Grammatik

Die EBNF wird nach dem Kochbuchrezept eingelesen und abgearbeitet. Danach werden die semantischen Aktionen eingewoben.

WinGraph

Die Implementierung von *WinGraph* findet in *ModShapes.pas* statt. Dieses ist objektorientiert geschrieben, um eine vereinfachte Handhabung mehrerer grafischen Objekte zu gewährleisten. Die verfügbaren grafischen Objekte[1] werden mithilfe der *init* instanziert und sind sofort sichtbar. Nach Instanzierung können diese Objekte auf der Bildfläche verschoben oder versteckt werden. Das Objekt *Picture* hat an sich keine grafische Repräsentation sondern agiert als ein Container, welcher mehrere andere grafische Objekte enthält. So können diese gruppiert werden und im Verbund verschoben werden. Ein Beispiel hierfür wäre ein aus mehreren Teilen bestehendes Auto, welches als gesamtes *Picture* über die Bildfläche geschoben werden kann.

10.1.2 Implementierung

[1] Line, Circle, Rectangle, Picture

Listing 10.1: Main.pas

```
program Main;
uses ModLex, ModSyn, Crt, Windows, WinGraph;

var
    inputFileName : string;

begin
    writeLn('Lets go!');
    inputFileName := '';
    if ParamCount > 0 then
        inputFileName := ParamStr(1);
    initLex(inputFileName);
newSy;
    redrawproc := S; // read a sentence using procedure for sentence symbol S
    WGMain;
    writeLn('Success: ', success);
end.
```

Listing 10.2: ModShapes.pas

```
unit modShapes;

interface

USES Windows, WinGraph, sysutils; (* for HDC *)

const
    MAX = 254;

type
  pointRec = record
        x : integer;
        y : integer;
            end;
    shape = ^shapeObj;
  shapeObj = object
                visible : boolean;
                name : string;
        procedure move(mx, my : integer); virtual; abstract; (* abstract methods have
    no definition *)
        procedure write; virtual; abstract;
        procedure draw(dc : HDC); virtual; abstract;
                function contains(ident : string) : shape; virtual;
                procedure setVisible(visible : boolean); virtual;
      end;
  shapeArray = array[1..MAX] of shape;
    line = ^lineObj;
  lineObj = object(shapeObj)
        private
          startP, endP : PointRec;
        public
          constructor init(tStartP, tEndP : pointRec; ident : string);
          procedure move(mx, my : integer); virtual;
                    procedure write; virtual;
                    procedure draw(dc : HDC); virtual;
      end;
```

```
36    rectangle = ^rectangleObj;
37  rectangleObj = object(shapeObj)
38          private
39            p0, p1, p2, p3 : pointRec;
40          public
41            constructor init(lt, rb : pointRec; ident : string);
42            procedure move(mx, my : integer); virtual;
43                        procedure write; virtual;
44                        procedure draw(dc : HDC); virtual;
45        end;
46    circle = ^circleObj;
47  circleObj = object(shapeObj)
48          private
49            center : pointRec;
50            radius : integer;
51          public
52            constructor init(c : pointRec;
53                                      r : integer;
54                                      ident : string
55                                     );
56            procedure move(mx, my : integer); virtual;
57                        procedure write; virtual;
58                        procedure draw(dc : HDC); virtual;
59        end;
60    picture = ^pictureObj;
61  pictureObj = object(shapeObj)
62          private
63            shapes : shapeArray;
64            numShapes : integer;
65          public
66            constructor init(ident : string);
67            procedure move(mx, my : integer); virtual;
68            procedure add(s : shape);
69                        procedure write; virtual;
70                        procedure draw(dc : HDC); virtual;
71                        function contains(ident : string) : shape; virtual;
72      procedure setVisible(visible : boolean);
73 virtual;
74                    end;
75
76 implementation
77
78 procedure addToPoint(var p : pointRec; x, y : integer);
79 begin
80   p.x := p.x + x;
81   p.y := p.y + y;
82 end;
83
84 (**************Shape**************)
85 function shapeObj.contains(ident : string) : shape;
86 begin
87     ident := upperCase(ident);
88     if(name = ident) then
89         contains := @self
90     else
91         contains := NIL;
92 end;
```

```
93 procedure shapeObj.setVisible(visible : boolean);
94 begin
95     self.visible := visible;
96 end;
97 (***************Line***************)
98 constructor lineObj.init(tStartP, tEndP : PointRec; ident : string);
99 begin
100     self.name := upperCase(ident);
101   self.startP := tStartP;
102   self.endP := tEndP;
103   visible := TRUE;
104 end;
105
106 procedure lineObj.move(mx, my : integer);
107 begin
108   addToPoint(startP, mx, my);
109   addToPoint(endP, mx, my);
110 end;
111
112 procedure lineObj.write;
113 begin
114   //writeLn('Line from ', startP.x, ',', startP.y, ') to (', endP.x, ',' endP.y, ')'
        );
115 end;
116
117 procedure lineObj.draw(dc : HDC);
118 begin
119     if not visible then exit;
120     moveTo(dc, startP.x, startP.y);
121     lineTo(dc, endP.x, endP.y);
122 end;
123
124 (***********************RECTANGLE*****************)
125 constructor rectangleObj.init(lt, rb : PointRec; ident : string);
126 begin
127     self.name := ident;
128   p0 := lt;
129   p2 := rb;
130   p1.x := p2.x;
131   p1.y := p0.y;
132   p3.x := p0.x;
133   p3.y := p2.y;
134 end;
135
136 procedure rectangleObj.move(mx, my : integer);
137 begin
138   addToPoint(p0, mx, my);
139   addToPoint(p1, mx, my);
140   addToPoint(p2, mx, my);
141   addToPoint(p3, mx, my);
142 end;
143
144 procedure rectangleObj.write;
145 begin
146   writeLn('Rectangle: ');
147   writeLn('(', p0.x,',', p0.y,')');
148 end;
```

```
149
150 procedure rectangleObj.draw(dc : HDC);
151 begin
152     if not visible then exit;
153     moveTo(dc, p0.x, p0.y);
154     lineTo(dc, p1.x, p1.y);
155     lineTo(dc, p2.x, p2.y);
156     lineTo(dc, p3.x, p3.y);
157     lineTo(dc, p0.x, p0.y);
158 end;
159
160 (*********************Circle*****************************)
161 constructor circleObj.init(c: pointRec; r : integer; ident : string);
162 begin
163     self.name := ident;
164   self.center := c;
165   self.radius := r;
166   visible := TRUE;
167 end;
168
169 procedure circleObj.move(mx, my : integer);
170 begin
171   addToPoint(center, mx, my);
172 end;
173
174 procedure circleObj.write;
175 begin
176   writeLn('Circle with center: (', center.x, ',', center.y, ') radius: ', radius);
177 end;
178
179 procedure circleObj.draw(dc : HDC);
180 begin
181     if not visible then exit;
182     Ellipse(dc, center.x - radius, center.y - radius,
183           center.x + radius, center.y + radius);
184 end;
185
186 (*********************PICTURE*************************)
187 constructor pictureObj.init(ident : string);
188 begin
189     self.name := ident;
190   numShapes := 0;
191   visible := TRUE;
192 end;
193
194 procedure pictureObj.move(mx, my : integer);
195 var i : integer;
196 begin
197   for i := 1 to numShapes do
198     shapes[i]^.move(mx, my);
199 end;
200
201 procedure pictureObj.add(s : shape);
202 begin
203   if numShapes >= MAX then begin
204     writeLn('Picture is full');
205     halt;
```

```
206   end;
207   if s = @self then begin
208     writeLn('Cannot add picture to itself');
209     halt;
210   end;
211   inc(numShapes);
212   shapes[numShapes] := s;
213 end;
214
215 procedure pictureObj.write;
216 var i : integer;
217 begin
218   writeLn('Picture with ', numShapes, ' shapes: ');
219   for i := 1 to numShapes do
220     shapes[i]^.write;
221 end;
222
223 procedure pictureObj.draw(dc : HDC);
224 var i : integer;
225 begin
226     if not visible then exit;
227     for i := 1 to numShapes do begin
228         shapes[i]^.draw(dc); (* forward all messages *)
229     end;
230     sleep(5);
231 end;
232
233 procedure pictureObj.setVisible(visible : boolean);
234 var i : integer;
235 begin
236   i := 1;
237   while (i <= numShapes) do begin
238     shapes[i]^.setVisible(visible);
239     inc(i);
240   end;
241 end;
242
243 function pictureObj.contains(ident : string) : shape;
244 var i : integer;
245     tPointer : shape;
246 begin
247     tPointer := inherited contains(ident);
248     if(tPointer = NIL) then begin
249         i := 1;
250         while (i <= numShapes) and (tPointer = NIL) do begin
251             tPointer := shapes[i]^.contains(ident); (* forward all messages *)
252             inc(i);
253         end;
254     end;
255     contains := tPointer;
256 end;
257
258 begin
259 end.
```

Listing 10.3: ModLex.pas

```
unit ModLex;

interface
type
    symbol = (numberSy, identSy,
              equalSy, semicolonSy, plusSy,
              showSy, hideSy,
              lineSy, rectangleSy,
              circleSy, pictureSy,
              moveSy,
              eofSy, noSy);
var
    curSy : symbol;
    numVal : integer; (* number value for semantic analysis *)
    identStr : string;

procedure newSy;
procedure initLex(inFileName : string);

implementation
const EOF_CH = chr(26);
    TAB_CH = chr(9);
    WNL_CH = chr(13);
    LNL_CH = chr(10);
var
    inFile : text;
    line : string;
    curChPos : integer;
    curCh : char;

procedure newCh; forward;

procedure initLex(inFileName : string);
begin
    assign(inFile, inFileName);
    reset(inFile);
    readLn(inFile, line);
    curChPos := 0;
    NewCh;
end;

procedure newSy;
begin
    (* skip whitespace *)
    while (curCh = ' ') or (curCh = TAB_CH) or (curCh = WNL_CH) or (curCh = LNL_CH)
    do newCh;
    case curCh of
        '=':    begin curSy := equalSy;     newCh; end;
        ';':    begin curSy := semicolonSy;    newCh; end;
        '+':    begin curSy := plusSy;    newCh; end;
        EOF_CH: begin curSy := eofSy;      newCh; end;
        'a'..'z', 'A'..'Z': BEGIN
          identStr := '';
          WHILE curCh IN ['a'.. 'z', 'A' ..'Z', '0'..'9', '_'] DO BEGIN
            identStr := Concat(identStr, UpCase(curCh));
            NewCh;
          END; (*WHILE*)
```

```
57            IF      identStr = 'SHOW' THEN
58              curSy := showSy
59            ELSE IF identStr = 'HIDE' THEN
60              curSy := hideSy
61            ELSE IF identStr = 'LINE' THEN
62              curSy := lineSy
63            ELSE IF identStr = 'RECTANGLE' THEN
64              curSy := rectangleSy
65            ELSE IF identStr = 'CIRCLE' THEN
66              curSy := circleSy
67            ELSE IF identStr = 'PICTURE' THEN
68              curSy := pictureSy
69            ELSE IF identStr = 'MOVE' THEN
70              curSy := moveSy
71            ELSE
72              curSy := identSy;
73          END;
74          '0'..'9': begin
75                  (* read a number *)
76                  numVal := Ord(curCh) - Ord('0'); (* value of digit*)
77                  newCh;
78                  while (curCh >= '0') and (curCh <= '9') do begin
79                      numVal := numVal * 10 + Ord(curCh) - Ord('0');
80                      newCh;
81                  end;
82                  curSy := numberSy;
83              end;
84          else begin curSy := noSy; newCh; end; (* default case *)
85      end; (* case *)
86  end;
87
88  procedure newCh;
89  begin
90      if curChPos < length(line) then begin
91          inc(curChPos);
92          curCh := line[curChPos]
93      end else begin
94          readLn(inFile, line);
95          if(length(line) = 0) then
96              curCh := EOF_CH
97          else begin
98              curChPos := 0;
99              inc(curChPos);
100             curCh := line[curChPos]
101         end;
102     end;
103 end;
104
105 begin
106 end.
```

Listing 10.4: ModSyn.pas

```
1  unit ModSyn;
2
3  interface
4  uses Windows, Crt, WinGraph;
```

```
 5 var
 6     success : boolean;
 7 procedure S(tDc: HDC; tWnd: HWnd; tRe: TRect);
 8
 9 implementation
10 uses ModLex, ModShapes;
11
12 procedure cartoon; forward;
13 procedure statement; forward;
14 procedure shapeDef; forward;
15 procedure action; forward;
16 var pic : picture;
17     tName : string;
18     dc : HDC;
19 wnd : HWnd;
20 re : TRect;
21
22 procedure S(tDc: HDC; tWnd: HWnd; tRe: TRect);
23 begin
24     (* S = expr EOF. *)
25     dc := tDc;
26     wnd := tWnd;
27     re := tRe;
28     success := TRUE;
29     new(pic, init('origin'));
30     cartoon; if not success then exit;
31     if curSy <> eofSy then begin success := FALSE; exit; end;
32     newSy;
33     (* sem *)
34     pic^.write;
35 writeln('Wrote stuff');
36     (* endsem *)
37 end;
38
39 procedure cartoon;
40 begin
41     (* Statement [ ';' Statement }. *)
42     statement; if not success then exit;
43     while (curSy = semicolonSy)do begin
44         newSy;
45         statement; if not success then exit;
46     end; (* while *)
47 end;
48
49 procedure statement;
50 begin
51     (* ident '=' ShapeDef | ('SHOW' | 'HIDE') ident | Action. *)
52     if(curSy = identSy) then begin
53         tName := identStr;
54         newSy;
55         if curSy <> equalSy then begin success := FALSE; exit; end;
56         newSy;
57         shapeDef; if not success then exit;
58     end else if(curSy = showSy) or (curSy = hideSy) then begin
59         case curSy of
60             showSy : begin
61                     newSy;
```

```
62                      if curSy <> identSy then begin success := FALSE; exit; end;
63                      (*SEM*)
64                      if pic^.contains(identStr) <> NIL then
65                          pic^.contains(identStr)^.setVisible(TRUE)
66                      else
67                          writeLn(identStr, ' not declared!');
68                      (*ENDSEM*)
69                  end;
70              hideSy : begin
71                      newSy;
72                      if curSy <> identSy then begin success := FALSE; exit; end;
73                      (*SEM*)
74                      if pic^.contains(identStr) <> NIL then
75                          pic^.contains(identStr)^.setVisible(FALSE)
76                      else
77                          writeLn(identStr, ' not declared!');
78                      (*ENDSEM*)
79                  end;
80          end;
81          newSy;
82          pic^.draw(dc);
83      end else begin
84          action; if not success then exit;
85      end;
86  end;
87
88  procedure shapeDef;
89  var p, q : pointRec;
90      radius : integer;
91      ident : string;
92      l : line;
93      c : circle;
94      r : rectangle;
95      userPic : picture;
96  begin
97      case curSy of
98          lineSy : begin
99                  newSy;
100                 if curSy <> numberSy then begin success := FALSE; exit; end;
101                 (*SEM*)p.x := numVal;(*ENDSEM*)
102                 newSy;
103                 if curSy <> numberSy then begin success := FALSE; exit; end;
104                 (*SEM*)p.y := numVal;(*ENDSEM*)
105                 newSy;
106                 if curSy <> numberSy then begin success := FALSE; exit; end;
107                 (*SEM*)q.x := numVal;(*ENDSEM*)
108                 newSy;
109                 if curSy <> numberSy then begin success := FALSE; exit; end;
110                 (*SEM*)q.y := numVal;(*ENDSEM*)
111                 newSy;
112                 (*SEM*)
113                 new(l, init(p, q, tName));
114                 pic^.add(l);
115                 (*ENDSEM*)
116             end;
117         rectangleSy : begin
118                 newSy;
```

```
119                 if curSy <> numberSy then begin success := FALSE; exit; end;
120                 (*SEM*)p.x := numVal;(*ENDSEM*)
121                 newSy;
122                 if curSy <> numberSy then begin success := FALSE; exit; end;
123                 (*SEM*)p.y := numVal;(*ENDSEM*)
124                 newSy;
125                 if curSy <> numberSy then begin success := FALSE; exit; end;
126                 (*SEM*)q.x := numVal;(*ENDSEM*)
127                 newSy;
128                 if curSy <> numberSy then begin success := FALSE; exit; end;
129                 (*SEM*)q.y := numVal;(*ENDSEM*)
130                 newSy;
131                 (*SEM*)
132                 new(r, init(p, q, tName));
133                 pic^.add(r);
134                 (*ENDSEM*)
135             end;
136         circleSy : begin
137                 newSy;
138                 if curSy <> numberSy then begin success := FALSE; exit; end;
139                 (*SEM*)p.x := numVal;(*ENDSEM*)
140                 newSy;
141                 if curSy <> numberSy then begin success := FALSE; exit; end;
142                 (*SEM*)p.y := numVal;(*ENDSEM*)
143                 newSy;
144                 if curSy <> numberSy then begin success := FALSE; exit; end;
145                 (*SEM*)radius := numVal;(*ENDSEM*)
146                 newSy;
147                 (*SEM*)
148                 new(c, init(p, radius, tName));
149                 pic^.add(c);
150                 (*ENDSEM*)
151             end;
152         pictureSy : begin
153                 (*SEM*) new(userPic, init(tName)); (*ENDSEM*)
154                 newSy;
155                 if curSy <> identSy then begin success := FALSE; exit; end;
156                 (*SEM*)
157     //writeLn('Pointer:', integer(pic^.contains(identStr)));
158                 if pic^.contains(identStr) <> NIL then
159                     userPic^.add(pic^.contains(identStr))
160                 else
161                     writeLn(identStr, ' not declared!');
162                 (*ENDSEM*)
163                 newSy;
164                 while (curSy = plusSy)do begin
165                     newSy;
166                     if curSy <> identSy then begin success := FALSE; exit; end;
167                     (*SEM*)
168                     if pic^.contains(identStr) <> NIL then
169                         userPic^.add(pic^.contains(identStr))
170                     else
171                         writeLn(identStr, ' not declared!');
172                     (*ENDSEM*)
173                     newSy;
174                 end; (* while *)
175     pic^.add(userPic);
```

```
176     pic^.draw(dc);
177             end;
178         else begin
179             success := FALSE; exit;
180         end; (* else *)
181     end;(* case *)
182 end;
183
184 procedure action;
185 var ident : string;
186     xMove, yMove : integer;
187 begin
188     if curSy <> moveSy then begin success := FALSE; exit; end;
189     newSy;
190     if curSy <> identSy then begin success := FALSE; exit; end;
191     (*SEM*)ident := identStr; (*ENDSEM*)
192     newSy;
193     if curSy <> numberSy then begin success := FALSE; exit; end;
194     (*SEM*)xMove := numVal;(*ENDSEM*)
195     newSy;
196     if curSy <> numberSy then begin success := FALSE; exit; end;
197     (*SEM*)yMove := numVal;(*ENDSEM*)
198     newSy;
199     (*SEM*)
200     if pic^.contains(identStr) <> NIL then
201         pic^.contains(identStr)^.move(xMove, yMove)
202     else
203         writeLn(identStr, ' not declared!');
204     (*ENDSEM*)
205     FillRect(dc, re, 0);
206     pic^.draw(dc);
207 end;
208
209 begin
210 end.
```

10.1.3 Ausgabe

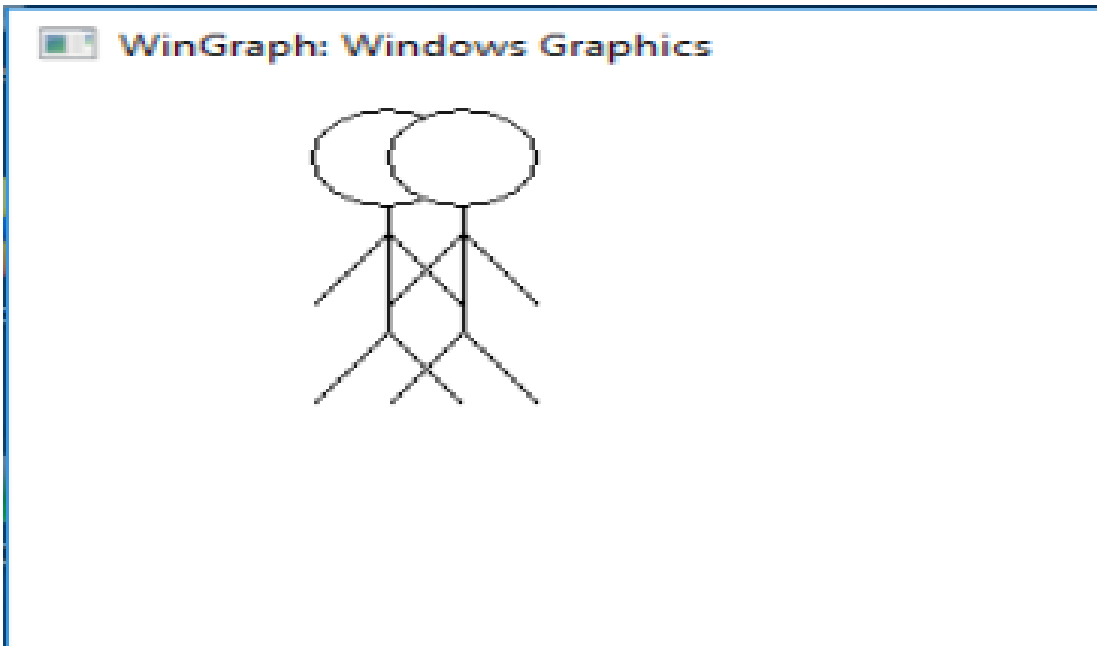

Abbildung 10.1: Beispiel für eine erzeugte Grafik des Programms