

objectiveValuation.pas

March 9, 2018

Contents

1	objectivevaluation	3
2	Purpose	4
3	Comand line interface	4
3.1	iotable	4
3.2	planray	5
3.3	reseourcevec	5
3.4	results	5
4	rf	7
5	kantor	7
6	IndexedMethod	12
7	kantorovichMethod	15
8	findFirstinput	16
9	claimHeapSpace	17
10	freeheapSpace	17
11	marginalgain	17
12	mpp	18
13	mostProductiveTechniqueFor	19
14	ComputeTotalsEtc	19
15	somePositiveTargetsStillZero	21
16	numpostargets	21
17	renormaliseMultipliers	21
18	LowerNonEquatedMultipliers	21

19 LowerNonEquatedIntensities	22
20 getavailability	22
21 RedistributeResources	22
22 rescaleall	24
23 PaulsOptimiser	24
24 KantorovichsOptimiser	24
25 findScoop	27
26 csvfilereader	28
27 Introduction	28
27.1 Background	28
27.2 CSV File Format	29
27.2.1 CSV File Structure	29
28 CSV File Rules	31
28.1 File Size	31
28.2 CSV Records	31
29 CSV Record Rules	31
29.1 CSV Field Column Rules	31
29.2 Header Record Rules	32
30 getdatamatrix	33
31 recursedown	33
32 recurse	33
33 getcolheaders	34
34 recurse	34
35 getrowheaders	35
36 recurse	35
37 colcount	35
38 getcell	36
39 removetrailingnull	36
40 onlynulls	36
41 rowcount	37

42	isint	37
43	printcsv	37
44	parsecsvfile	38
45	thetoken	39
46	peek	39
47	isoneof	39
48	nextsymbol	39
49	have	39
50	haveoneof	40
51	initialise	40
52	resolvealpha	40
53	resolvedigits	41
54	resolvetoken	41
55	markbegin	42
56	markend	42
57	setalpha	42
58	emptyfield	42
59	parsebarefield	43
60	parsedelimitedfield	43
61	appendcurrentchar	43
62	parsefield	44
63	parserecord	44
64	parseheader	44
65	parsewholefile	44

1 objectivevaluation

```

program objectiveValuation ;
uses kantor ,csvfilereader ;

```

2 Purpose

The programme applies the Objective Valuation technique of the Soviet Nobelist Kantorovich to data from input output tables. Kantorovich derived a set of numbers, which he termed Objectively Determined Valuations, from a combination of

1. the technology available
2. the objectives of the socialist plan
3. the available stock of material resources

This programme can potentially be used in studies of the comparative efficiency of labour values, prices of production or Kantorovich valuations in economic planning.

3 Comand line interface

Usage : `objectiveValuation iotable.csv planray.csv resourcevec.csv`
`> result.csv`

The files should be laid out with a first line and first column made up of text fields labeling the rows and columns. All other cells should be numeric. Discussion of matrix sizes in what follows refers exclusively to the rectangular subarray of numeric values.

3.1 *iotable*

The first file, the *iotable* one, should contain an N by M io table in standard column form, with a column corresponding to an industry so that cell at row i col j contains the amount of output from the ith industry used by the jth industry.

The last line must contain the outputs of each industry.

3.2 planray

This contains the target level of gross outputs in the next period to be obtained using the technology of the iotable. In line with Kantorovich practice this corresponds to a ray in N-1 dimensional space along which the outputs are supplied in fixed proportions. This is also sometimes called a Leontief demand function. The format, excluding headers, is a 1 by N-1 matrix, or N-1 element row vector. For a typical io table this implies that even though labour is not a produced input it should appear as a zero output target in the plan.

Headers should correspond to the column headers of the IO table.

3.3 reseourcevec

This is a N-1 row vector of available resources, the headers should correspond to the row titles of the IO table.

3.4 results

The output is a 2 by M matrix in csv format. The two rows encode the intensity with which each column technology in the IO table must be operated and the objective valuations associated with each output.

```
type
  pmat = ^ matrix ;
  channel = record
    p : pcsv;
    r;
    m : ↑ matrix ;
  end ;
procedure rf ( var ch :channel ;param :integer ); (see Section 4 )
var
  Let iot, rv, pr, results ∈ channel;
  Let tech ∈ ^technologytable;
  Let odvs ∈ ^ODVvec;
  Let X ∈ ^intensityvec;
```

```

Let target, startingresource ∈  $\hat{\text{resourcevec}}$ ;
Let outputs ∈  $\hat{\text{resourcevec}}$ ;
Let i, j ∈ integer;
begin
  rf (iot, 1);
  rf (pr, 2);
  rf (rv, 3);

  new ( tech, iot .m ^ .cols, iot .m ^ .rows -2);
  tech↑ ← 0;

  for i ← 1 to tech↑.ω do
    begin
      for j ← agriculture to tech↑.amusements do
        begin
          tech↑[i, src, j] ← - iot.m↑[j + 1, i] ;
        end ;
      tech↑[i, dest, i - 1] ← iot.m↑[iot.m↑.rows, i];
    end ;

    new ( target, pr .m ^ .cols -1);
    target↑ ← pr.m↑[1, 1 +  $\iota_0$ ];
    new ( startingresource, rv .m ^ .cols -1);
    new ( odvs, rv .m ^ .cols -1);

    new ( X, iot .m ^ .cols );

    startingresource↑ ← rv.m↑[1, 1 +  $\iota_0$ ];
    optimiser ← pauls;
    kantorovichMethod ( tech ^ , target ^ ,
      startingresource ^ ,
      odvs ^ , X ^ );

    for i ← 1 to iot.m↑.cols do
      write ( ' ,', iot .c ^ [i] ^ .textual ^ );
      writeln;
      write( 'output' );
      for i ← 1 to iot.m↑.cols do
        write ( ' ,', x ^ [i] * iot .m ^ [iot .m ^ .rows, i]:1:4);
        writeln;
        write( 'odv' );
        for i ← 1 to iot.m↑.cols do
          write ( ' ,', odvs ^ [i -1]:1:4);
          writeln;

```

omit last row and allow for
the base of 2nd dim of tech-
niques to be 0

inputs go in transposed with
minus sign

outputs from last row of io
matrix

there is to be an odv for each
resource
there is an intensity for each
production technique, thus 1
for each column in the io ta-
ble

end .

4 rf

procedure *rf* (**var** *ch* :*channel* ;*param* :*integer*);

Read in one of the file parameters and extract the data from it

```
begin
  with ch do
    begin
      p ← parsecsvfile (paramstr (param));
      if p = nil then
        begin
          writeln( 'error opening or parsing file ' , paramstr (param));
          halt (2);
        end
      else ;
        r ← getrowheaders (p);
        c ← getcolheaders (p);
        m ← getdatamatrix (p);
      end ;
    end ;
end ;
```

5 kantor

What follows is a library written in Vector Pascal. The text in roman font that follows are comments. The program code is generally in san-serif font, and the whole, is in the literate programming output format generated by the compiler.

unit *Kantor* ;

* The library provides a procedure which generalises the algorithm of Kantorovich that he first presented in the context of excavators working on soils of different types, so that it will work for a general plan optimisation for the whole economy by his method of resolving multipliers. It starts out from the data provided in an i/o Table. In the table the *techniques* for for different production techniques are shown in italics. In this context a technique means the expected output per year of the technique. In the case of Kantorovich's original algorithm

the techniques were represented as a matrix of outputs of different soils and machines. In an input output table a technique is conventionally represented by a column (Leontief notation) or a row (Sraffa notation) of a matrix.

We can convert the Kantorovich table

105	107	64
56	66	38
56	83	53

Where each row corresponds to a soil type and each column to an excavator into an extended Sraffa format as

105	0	0	-1	0	0
107	0	0	0	-1	0
64	0	0	0	0	-1
0	56	0	-1	0	0
0	66	0	0	-1	0
0	38	0	0	0	-1
0	0	56	-1	0	0
0	0	83	0	-1	0
0	0	53	0	0	-1

where the first 3 columns correspond to the outputs of different soils and the next 3 to the outputs of excavators. The outputs of excavators are negative since a given process uses up an excavator.

In practice we split this into two distinct tables one the **dest** table which lists the outputs and the other the **src** table which lists the inputs. Thus:

dest

105	0	0	0	0	0
107	0	0	0	0	0
64	0	0	0	0	0
0	56	0	0	0	0
0	66	0	0	0	0
0	38	0	0	0	0
0	0	56	0	0	0
0	0	83	0	0	0
0	0	53	0	0	0

src

0	0	0	-1	0	0
0	0	0	0	-1	0
0	0	0	0	0	-1
0	0	0	-1	0	0
0	0	0	0	-1	0
0	0	0	0	0	-1
0	0	0	-1	0	0
0	0	0	0	-1	0
0	0	0	0	0	-1

Why reorganise like this?

Because the general economic planning problem involves the production and consumption of many of the same goods. Energy is both used and produced, for example. It is thus convenient to show the inputs and outputs in two different tables.

The pair composed of a destination row and the corresponding src row we term a technique.

In principle we could have a problem with joint production, but for now we are sticking to the simpler single product per technique problem. Thus each row of the destination column contains only one non zero element.

The objective was given, by Kantorovich in a final column of his table, in our format we would have row vector of resources available at the start of the period.

Kantorovich's algorithm demanded equal quantities of all outputs which is not the general case. We keep the Kantorovich approach within this library, which means that the units of measurement of all inputs must be scaled in order to result in an output that is the same, in these units of measurement, for all products.

Thus if we wanted to produce 2 cubic meters of soil type 1 relative to every cubic meter of soil type 2 and 3 we use 2 cubic meter as the standard unit for 1.

We handle this by dividing the appropriate column of the technology matrix by 2 and similarly scaling any corresponding entries for the available resources. More generally, given a targets vector we divide all other row vectors in the problem by this.

$$20000 \quad 20000 \quad 20000 \quad 0 \quad 0 \quad 0$$

And an initial resource vector of

$$0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1$$

indicating we have 1 machine of each type.

My first step in developing the programme in this new format was to take the programme that I had previously published in Intervention, and to reformulate it in this more general form. The aim is to check that given the new data format the algorithm still gives the same answers as Kantorovich. Following on from that we can provide a more general version that will work on arbitrary sized input output tables.

Following the columns of the techniques Kantorovich gives the optimal allocation of machine times to activities to minimise overall time taken to do the digging. The program will reproduce this result by applying his method of resolving multipliers or objectively determined valuations. We first introduce our domain of discourse resources, techniques and units of usage and output. We simply number the resources 0 to 5 with resources in this case including soil and machines.

```

interface
  const
    alpha =1;
    epsilon =0.001;
    agriculture =0;

```

first entry in US IO table is
agriculture

Kantorovich's Objectively
Determined Valuations

last entry of US IO table is
amusements

```

    dest =0;
    src =1;
    growthconst =0.1;
    annealrate =0.99;
type
    optimizer =( kantorovichs ,pauls );
    flow = dest ..src ;
    resource = integer ;
    units =real ;
    intensity = real ;
    ODV = real ;

    technique = integer ;
    resourcevec(amusements:resource ) = array [agriculture ..amusements ] of units ;

```

A technology is defined as a flow of src inputs to a flow of dest outputs, with each flow being a vector.

```

    technologytable(omega,amusements:integer ) = array [alpha .. omega ,dest ..src ,agricul-
    ture .. amusements ] of units ;
    ODVvec =resourcevec ;

    intensityvec =vector ;
    volume =real ;
var
    Let growthfactor ∈ real;
    Let optimiser ∈ optimizer;
procedure kantorovichMethod ( var techniques :technologytable ; var target ,startingresource :resourcevec

```

The method must be supplied with a technology table and a vector of targets and available resources. It outputs a vector of resolving multipliers L and an vector of technique intensities X which specify how much each technique will be used.

implementation

type

```

    productionIndex (omega:integer )= array [alpha .. omega ] of resource ;
    pproductionIndex = ^ productionIndex ;

procedure IndexedMethod ( var produces :productionindex ;var techniques :technologytable ; var targets ,s

```

```

unit initialisation
    procedure kantorovichMethod ( var techniques :technologytable ; var target ,startingresource ,L :ODVvec ;
    begin
        growthfactor ← growthconst;
        optimiser ← kantorovichs;
    end .
unit kantor

```

6 IndexedMethod

```

procedure IndexedMethod ( var produces :productionindex ;var techniques :technologytable ;
var targets ,startingresource ,L :ODVvec ; var X :intensityvec );

```

The indexed method is a nested procedure which will work using an index of what each technique produces. Many techniques may produce same output.

The algorithm manipulates a vector x which will encode the time for which each technique is operated, this corresponds in Kantorovich's case to the time a given type of machine spends on a given soil. Later it may have a more general impliation. Next L , a vector of objectively determined valuations of resources¹.

The standardised output of each technique is obtained by applying resolving multipliers to the outputs types it produces.

```

type
    outputmatrix(omega,amusements:integer ) = array [alpha .. omega ,agriculture .. amuse-
        ments ] of units ;
var
    Let targetintensity ∈ real;
    Let achievableintensities ∈  $\hat{\text{resourcevec}}$ ;
    Let dx ∈ intensity;
    Let standardisedoutput ∈  $\hat{\text{outputmatrix}}$ ;
    Let available ∈  $\hat{\text{resourcevec}}$ ;
    Let outputs ∈  $\hat{\text{intensityvec}}$ ;
    Let totals ∈  $\hat{\text{odvvec}}$ ;
    Let ok ∈ boolean;
    Let greatestProducedOutput , leastProducedOutput , deltam ∈ real;

```

¹Presumably Kantorovich labels his multipliers L because of their similarity to Lagrangian multipliers

first input resource

Output weighted by L

```

    Let best ∈ real;
    e , e2 , Scoop ,
    Let A ∈ resource;
    Let r, s, j ∈ resource;
    Let weightedTotalOutput ∈ real;
    Let  $\lambda$ , f, t ∈ real;
    Let least, m, jt, st ∈ technique;
    Let count, k ∈ integer;
    Let equated ∈  $\mathbb{R}^n$ ;
function findfirstinput :resource ; (see Section 8 )

procedure claimHeapSpace ; (see Section 9 )
procedure freeheapSpace ; (see Section 10 )

function marginalgain ( s :technique ; d :resource ):volume ; (see Section 11 )

function mpp ( s :technique ; d :resource ):volume ; (see Section 12 )

function mostProductiveTechniqueFor ( e :resource ):technique ; (see Section 13 )

procedure ComputeTotalsEtc ; (see Section 14 )

function somePositiveTargetsStillZero :boolean ; (see Section 15 )
function numpostargets :integer ; (see Section 16 )
procedure renormaliseMultipliers ; (see Section 17 )
procedure LowerNonEquatedMultipliers ; (see Section 18 )
procedure LowerNonEquatedIntensities ; (see Section 19 )
procedure getavailability ; (see Section 20 )
procedure RedistributeResources ; (see Section 21 )
procedure rescaleall ; (see Section 22 )

procedure PaulsOptimiser ; (see Section 23 )

procedure KantorovichsOptimiser ; (see Section 24 )
var
    Let av ∈ real;
begin
    claimHeapSpace;

    initialise multipliers

    L ← 1;
    ok ← false;
    count ← 0;
    f ← 0.3;

    Iterate the following steps until we have a satisfactory answer.

    while not ok do
    begin

```

default assumption

```

available↑ ← startingresource;
x ← 0;
redistributeResources;
totals↑ ← 0;

```

Compute the total output of each product in value terms

```

for k ← α to techniques.ω do
  totals↑[producesk] ← totals↑[producesk] + (techniquesk,dest,producesk) × xk × Lproducesk;
greatestProducedOutput ← \max totals↑;
leastProducedOutput ← \min totals↑;
if somePositivetargetsStillZero then
  begin

```

check if any resource has a zero output and raise its value if it has

s iterates over outputs

```

  for s ← agriculture to techniques.amusements do
    if targetss > 0 then
      if totals↑[s] ≤ 0 then L[s] := L[s] * (1.0 + growthfactor * (random and 7)/8 +
        growthfactor);
      renormaliseMultipliers;
    end
  else ok ← true;
  count ← count + 1;
  {writeln(X, L/L[0], count);}
  growthfactor ← growthfactor × annealrate;
  if count > techniques.omega × 10000 then
    begin
      writeln('after ', count, ' trys could not find initial resolving multipliers' );
      halt (999);
    end ;
  end ;
  {writeln('initial multipliers found after ', count, ' trys');}
  writeln(L);}
  count ← 0;

```

At this point our estimate of the resolving multipliers is accurate enough to ensure that some of each output is now being produced, but we have not yet met the requirement that the target amount of each output must be produced. We now try to get a more precise estimate of the resolving multipliers and in the process we adjust the amounts of each output being made. It is important to note at this point that any further adjustments must come by de-specialising some of the inputs so that they are used more than one output type. The resolving multipliers have until now been used to weight the outputs of different types in order to assign each input to the output it is best suited to. If an input is no longer specialised, that is if it produces more than one output, then the weights must be such that it is no longer *best* at one particular output type. The multipliers

must be set so that the marginal weighted output of the resource on any of the outputs on which it is employed are the same. Thus if a machine k is employed on two resources i, j then $\text{standardisedoutput}[k,j]=\text{standardisedoutput}[k,i]$.

In turn this implies that for any machine that is employed to make two resources the ratio of the resolving multipliers must be the inverse of the ratio of the techniques.

The algorithm will work output a time bringing ever more outputs into equality with their targets. We define the set of resources whose outputs has been brought into equality with target as the equated set.

For those outputs in the equated set, the resolving multipliers of the outputs will have been corrected so that for any machine moving more than one resource they stand in inverse ratio to that machine's productivity.

```

growthfactor ← growthconst;
computeTotalsEtc;
repeat
  case optimiser of
    kantorovichs : KantorovichsOptimiser;
    pauls : PaulsOptimiser;
  end ;
  computeTotalsEtc;
  count ← count + 1;

until ((  $\sum \text{equated} \uparrow$  ) = numpostargets) ∨ (count > (500)) ∨ (growthfactor <  $\epsilon$ );
{ writeln('second pass terminated after ', count);}
rescaleall;
freeHeapSpace;
end ;

```

Indexed method

7 kantorovichMethod

```

procedure kantorovichMethod ( var techniques : technologytable ; var target , startingresource
, L : ODVvec ; var X : intensityvec );
var

```

Kantorovich method

```

Let  $index \in \hat{\text{productionindex}}$ ;
Let  $t \in \text{technique}$ ;
Let  $r \in \text{resource}$ ;
Let  $targets \in \hat{\text{odvvec}}$ ;
begin
  new (  $index$  ,  $techniques$  .  $\omega$  );

   $growthfactor \leftarrow growthconst$ ;
  new (  $targets$  ,  $techniques$  .  $\text{amusements}$  );
   $targets \uparrow \leftarrow \begin{cases} 0.0 & \text{if } target = 0.0 \\ target/target_0 & \text{otherwise} \end{cases}$  ;
  for  $t \leftarrow \alpha$  to  $techniques.\omega$  do
    for  $r \leftarrow agriculture$  to  $techniques.\text{amusements}$  do
      if  $techniques_{t,dest,r} > 0$  then  $index \uparrow[t] \leftarrow r$ ;

```

prescale by targets

```

 $techniques \leftarrow \begin{cases} techniques/targets \uparrow & \text{if } target \neq 0 \\ techniques & \text{otherwise} \end{cases}$  ;
 $startingresource \leftarrow \begin{cases} startingresource/targets \uparrow & \text{if } target \neq 0 \\ startingresource & \text{otherwise} \end{cases}$  ;

 $indexedMethod (index \uparrow, techniques, targets \uparrow, startingresource, L, X)$ ;

```

return scale of the input data to its original value

```

 $techniques \leftarrow \begin{cases} techniques \times targets \uparrow & \text{if } target \neq 0 \\ techniques & \text{otherwise} \end{cases}$  ;
 $startingresource \leftarrow \begin{cases} startingresource \times targets \uparrow & \text{if } target \neq 0 \\ startingresource & \text{otherwise} \end{cases}$  ;
dispose (  $index$  );
dispose (  $targets$  );
end ;

```

Kantorovich method

8 findfirstinput

```

function  $findfirstinput : resource$  ;
var
  Let  $t \in \text{technique}$ ;
  Let  $r, f \in \text{resource}$ ;
begin
   $f \leftarrow techniques.\text{amusements}$ ;
  for  $t \leftarrow \alpha$  to  $techniques.\omega$  do
    for  $r \leftarrow f$  downto 0 do
      if  $techniques_{t,src,r} > 0$  then  $f \leftarrow r$ ;
   $findfirstinput \leftarrow f$ 

```


Find first input

end ;

9 claimHeapSpace

procedure *claimHeapSpace* ;

var

Let *maxtechnique*, *maxresource* \in integer;

begin

maxtechnique \leftarrow *techniques.w*;

maxresource \leftarrow *techniques.amusements*;

new (*achievableintensities* ,*maxresource*);

new (*standardisedoutput* ,*maxtechnique* ,*maxresource*);

new (*available* ,*startingresource* .*amusements*);

new (*outputs* ,*maxtechnique*);

new (*totals* ,*maxresource*);

new (*equated* ,*maxresource*);

A \leftarrow *findfirstinput*;

end ;

claimHeapSpace;

claimHeapSpace

10 freeheapspace

procedure *freeheapspace* ;

begin

dispose (*achievableintensities*);

dispose (*standardisedoutput*);

dispose (*available*);

dispose (*outputs*);

dispose (*totals*);

dispose (*equated*);

end ;

11 marginalgain

function *marginalgain* (*s* :*technique* ; *d* :*resource*):*volume* ;

This computes the marginal gain, of a small shift of the resource *d* time to the specified technique type *s*. We compute the effect of multiplying all current time allocations to $1 - \epsilon$ whilst increasing the allocation of time to resource *s* by ϵ .

var

Let *currentoutput*, *currentuse*, *shift*, *marginalIncreaseinS* \in real;

Let *valueofincrease*, *valueofloss* \in real;

Let *t* \in technique;

marginal gain

begin

check if technique s uses resource d

```
if techniquess,src,d = 0 then
begin
    marginalgain ← - maxint ;
end
else
begin
```

get total current use of resource d

current use is negative !

shift will be negative too
marginal increase will be
+ve, since we divide by a
negative

```
currentuse ← ∑ x × techniquest0,src,d ;
shift ← ε × currentuse;
marginalIncreaseInS ←  $\frac{shift \times techniques_{s,dest,produces_s}}{techniques_{s,src,d}}$ ;
```

```
valueofincrease ← marginalIncreaseInS × Lproducess;
```

we next have to work out the total value of output currently produced in all
sectors by d

```
valueofloss ← 0;
for t ← α to techniques.ω do
    if techniquest,src,d ≠ 0 then
        valueofloss ← valueofloss + techniquest,dest,producest × xt × ε × Lproducest;
```

if it uses d

```
marginalgain ← valueofincrease - valueofloss
```

```
end
```

marginalgain

```
end ;
```

12 mpp

```
function mpp ( s : technique ; d : resource ): volume ;
```

This computes the marginal physical product or the output of s with respect to
input d

mpp

begin

check if technique s uses resource d

```

if  $techniques_{s,src,d} = 0$  then
begin
   $mpp \leftarrow -maxint$  ;
end
else
begin
   $mpp \leftarrow \frac{techniques_{s,dest,produces_s}}{techniques_{s,src,d}}$  ;
end
end ;

```

mpp

13 mostProductiveTechniqueFor

function *mostProductiveTechniqueFor* ($e : resource$): *technique* ;

This determines within which technique resource e produces the most of. under current valuations.

```

var
  Let  $v \in volume$ ;
  Let  $j, s \in technique$ ;
begin
   $v \leftarrow 0.0$ ;
   $s \leftarrow \alpha$ ;
  for  $j \leftarrow \alpha$  to  $techniques.\omega$  do
    if  $techniques_{j,src,e} < 0$  then
      begin
        if  $v < \frac{L_{produces_j} \times techniques_{j,dest,produces_j} \times X_j}{-techniques_{j,src,e}}$  then
          begin
             $v \leftarrow \frac{L_{produces_j} \times techniques_{j,dest,produces_j} \times X_j}{-techniques_{j,src,e}}$ ;
             $s \leftarrow j$ ;
          end ;
        end ;
      end ;
   $mostProductiveTechniqueFor \leftarrow s$ ;
mostProductiveTechniqueFor end ;

```

it uses this input

$mostProductiveTechniqueFor$ **end ;**

14 ComputeTotalsEtc

procedure *ComputeTotalsEtc* ;

Work out how much is being produced, which resource is being produced least and which resources outputs are equals to this,

```

var
  Let  $i, j \in \text{integer}$ ;
  Let  $least \in \text{real}$ ;
begin
   $totals \uparrow \leftarrow 0$ ;
  for  $i \leftarrow \alpha$  to  $techniques.\omega$  do
    begin
       $totals \uparrow [\text{produces}_i] \leftarrow totals \uparrow [\text{produces}_i] + (techniques_{i,dest,produces_i}) \times x_i$ ;
    end ;

```

$totals$ now has a vector of how much of each resource is produced at the end relative to the target

```

 $weightedTotalOutput \leftarrow L.totals \uparrow$ ;
 $leastProducedOutput \leftarrow \backslash min (totals \uparrow)$  ;

```

Find the resource that is least produced or most in shortage

```

for  $s \leftarrow agriculture$  to  $techniques.amusements$  do
  if  $(totals \uparrow [s]) = leastProducedOutput$  then  $least \leftarrow s$ ;

```

Find the ones on the plan ray

```

 $equated \uparrow \leftarrow 0$ ;
 $least \leftarrow maxint$ ;
for  $i \leftarrow agriculture$  to  $techniques.amusements$  do
  if  $targets_i > 0$  then
    if  $totals \uparrow [i] < least$  then  $least \leftarrow totals \uparrow [i]$ ;

for  $i \leftarrow agriculture$  to  $techniques.amusements$  do
  if  $targets_1 > 0$  then
    if  $(totals \uparrow [i] - least) < \epsilon$  then  $equated \uparrow [i] \leftarrow 1$ ;

 $equated \uparrow \leftarrow \text{ord}(((totals \uparrow - least) < \epsilon) \wedge ((totals \uparrow - least) > -\epsilon))$ ;
{writeln('t',totals $\hat{\phantom{x}}$ );
writeln('e',equated $\hat{\phantom{x}}$ );writeln('x',x);
writeln('least',least);
{write('press return');readln;{}}

```

```

end ;

```

15 somePositiveTargetsStillZero

function *somePositiveTargetsStillZero* : *boolean* ;

returns true if some products which should have positive output are still zero

begin

somePositiveTargetsStillZero \leftarrow \ or $((totals \uparrow \leq 0) \wedge (targets > 0))$;
end ;

16 numpostargets

function *numpostargets* : *integer* ;

begin

end ;

17 renormaliseMultipliers

procedure *renormaliseMultipliers* ;

In order to prevent uncontrolled growth or shrinkage in multipliers, express them with respect to their own average.

var

Let $av \in \text{real}$;

begin

$av \leftarrow \sum \frac{L}{1 + techniques.amusements}$;

$L \leftarrow \frac{L}{av}$;

end ;

find the average

renormalise L

18 LowerNonEquatedMultipliers

procedure *LowerNonEquatedMultipliers* ;

begin

$L \leftarrow L \times (equated \uparrow \times growthfactor + 1 - growthfactor)$;

renormaliseMultipliers;

end ;

those in the equated set are
unchanged by this others are
reduced

19 LowerNonEquatedIntensities

```
procedure LowerNonEquatedIntensities ;  
begin  
     $X \leftarrow X \times (\text{equated}\uparrow[\text{produces}] \times \text{growthfactor} + 1 - \text{growthfactor});$   
end ;
```

20 getavailability

```
procedure getavailability ;  
var  
    Let  $i \in \text{integer}$ ;  
begin  
     $\text{available}\uparrow \leftarrow \text{startingresource};$   
    for  $i \leftarrow \alpha$  to  $\text{techniques.}\omega$  do  
         $\text{available}\uparrow \leftarrow \text{available}\uparrow + \text{techniques}_{i,\text{src}} \times X_i ;$   
end ;
```

21 RedistributeResources

```
procedure RedistributeResources ;
```

move the free resources to the place where they are most valuable

```
begin
```

Use the L to get a standardised performance for each process.

```
 $\text{standardisedoutput}\uparrow \leftarrow (\text{techniques}_{\iota_0, \text{dest}} \times L);$   
getavailability;
```

For each input find the technique for which it has the best performance

```
for  $e \leftarrow \text{agriculture}$  to  $\text{techniques.amusements}$  do  
    if  $\text{available}\uparrow[e] > 0$  then  
        begin
```

test it is an input

find the best performance of the input on any technique

try every technique
which uses resource e

```

outputs↑← 0;
for s←  $\alpha$  to techniques. $\omega$  do
  if techniquess,src,e < 0 then
    begin
      t← 0;
      for r← agriculture to techniques.amusements do
        t← t + standardisedoutput↑[s, r];
      outputs↑[s]← t;
    end ;
  best← \max outputs↑ ;

```

allocate each resource to the technique it is best at

```

for s←  $\alpha$  to techniques. $\omega$  do
  if outputs↑[s] = best then
    begin

```

set this technique to have an intensity sufficient to use 50 percent of the resource e that is available subject to the availability of the other resources technique s needs. The achievable intensities specifies how much more output this technique could produce subject to the constraint of each type of resource's availability. Thus it is indexed by the available resource types.

```

achievableintensities ^ := if techniques [s ,src ]=0.0 then maxreal
else - available↑ / (techniquess,src) ;

targetintensity← 0.5 × \min achievableintensities↑ ;

```

We find the lowest of the intensities that can be achieved with remaining resources.

add it into x

```

xS← xS + targetintensity;

```

update the available resources , we add the source weighted by intensity, note that these numbers will be negative and will reduce availability

```

available↑← available↑ + (targetintensity × techniquess,src) ;
end ;

end ;

end ;

```

22 rescaleall

```
procedure rescaleall ;
(* this, having found the best ratio of intensities, scales them all to the
limit of the available resources by finding which is the rate limiting resource *)
var
  Let usage  $\in$   $\hat{\text{resourcevec}}$ ;
  Let lowestsurplus  $\in$  real;
begin
  getavailability;
  new ( usage ,techniques .amusements );
  usage $\uparrow$  $\leftarrow$  startingresource - available $\uparrow$ ;
  lowestsurplus $\leftarrow$   $\backslash \min$  (  $\frac{\textit{startingresource}}{\textit{usage}}$  $\uparrow$  ) ;
  X $\leftarrow$  X  $\times$  lowestsurplus;
  dispose ( usage );
end ;
```

23 PaulsOptimiser

```
procedure PaulsOptimiser ;
(* ! this is an alternative algorithmic step for finding the optimal resolving
multipliers
once an initial set has been found.
```

Its basic steps are:

```
\begin{enumerate}
\item Lower the resolving multipliers of the non equated outputs relative to the
equated ones.
\item Lower the intensities (X) of all processes producing non equated outputs.
\item Redistribute the free resources to the processes whose outputs are in the
equated set.
\end{enumerate}
*)
```

```
begin
```

```
  LowerNonEquatedMultipliers;
  LowerNonEquatedIntensities;
  RedistributeResources;
  rescaleall;
  growthfactor $\leftarrow$  growthfactor  $\times$  annealrate;
```

```
end ;
```

24 KantorovichsOptimiser

```
procedure KantorovichsOptimiser ;
```


This optimising step is called repeatedly once an initial set of resolving multipliers has been found. It is the one originally proposed by Kantorovich himself.

function *findScoop* (**var** *s* : *technique*) : *resource* ; (see Section 25)
begin

Find which input is 2nd best at producing output least in the non equated set under current resolving multipliers. Call this input Scoop. The parameter least is an output of the function call.

Scoop ← *findScoop* (*least*);
m ← *mostProductiveTechniqueFor* (*Scoop*);

Adjust the resolving multiplier ratio between Scoops resource and the least produced resource to ratio of Scoops techniques.

$$L_{produces_m} \leftarrow \frac{L_{produces_{least}} \times mpp(least, scoop)}{mpp(m, scoop)};$$

It is now necessary to reduce the output of scoop on scoops output and increase it on the least produced output. It is necessary to compute how much to reduce scoops resource by. The resolving multipliers give us substitution ratios between different resource outputs. Suppose that we want to reduce output of resource *m* by one unit and increase the output of resource *j*, the increase in *j* we get is

$$\Delta_j = \frac{L_m}{L_j}$$

. If we want to reduce the output of *i* and we have two other resources *j, k* which we want to increase equally then we have

$$\Delta_k = \Delta_j$$

where Δ_x means change in *x* and

$$-\Delta_m L_m = \Delta_k L_k + \Delta_j L_j = \Delta_k (L_j + L_k)$$

. Thus

$$\Delta_m = -\Delta_k \frac{L_j + L_k}{L_m}$$

Let C_m, C_j, C_k be the current outputs of each resource; given that $C_j = C_k$ we have to chose the Δ s so that

$$C_m + \Delta_m = C_j + \Delta_j = C_k + \Delta_k$$

It follows that

$$C_m - \Delta_k \frac{L_j + L_k}{L_m} = C_k + \Delta_k$$

and

$$C_m - C_k = \Delta_k \frac{L_j + L_k}{L_m} + \Delta_k = \Delta_k \left(1 + \frac{L_j + L_k}{L_m}\right)$$

so

$$\Delta_j = \frac{C_m - C_j}{1 + \frac{L_j + L_k}{L_m}}$$

We next compute the reduction to be made in resource m from the formula

$$\Delta_m = -\Delta_k \frac{L_j + L_k}{L_m}$$

substituting we get

$$\Delta_m = -\frac{C_m - C_j}{1 + \frac{L_j + L_k}{L_m}} \left(\frac{L_j + L_k}{L_m}\right)$$

Translating this to the variables used in the program we have:

$j \leftarrow \text{produces}_{least};$

$\lambda \leftarrow \frac{L \cdot (\text{equated}_{\uparrow})}{L_{\text{produces}_m}};$

we have used the cross product with the equated set, represented as a vector of 0 or 1 to sum over the equated set.

$\text{deltam} \leftarrow \frac{(-1) \times \lambda \times (\text{totals}_{\uparrow}[\text{produces}_m] - \text{totals}_{\uparrow}[j])}{1 + \lambda};$

Note that in the line above we are generalising the term $L_j + L_k$ to an arbitrary number of multipliers (1 or 2 in this program) by computing the inner product between the equated vector and the multipliers. This works because the equated vector has a 1 for all resources in the equated set. We now compute the change in intensity that Scoop spends on its best resource (dx) by scaling deltam by Scoops technique output for resource m.

$dx \leftarrow \frac{\text{deltam}}{-\text{mpp}(m, \text{scoop})};$

reallocate this time to Scoops best resource in the equated set which we will now call j

$\text{best} \leftarrow - \text{maxint};$

$jt \leftarrow \alpha;$

for $st \leftarrow \alpha$ **to** $\text{techniques}.\omega$ **do**

```

    if  $mpp(st, scoop) \times equated \uparrow [produces_{st}] > best$  then
    begin
         $best \leftarrow mpp(st, scoop)$ ;
         $jt \leftarrow st$ ;
    end ;

     $x_m \leftarrow x_m + dx$ ;
     $x_{jt} \leftarrow x_{jt} - dx$ ;
end ;

```

25 findScoop

function *findScoop* (**var** *s* : *technique*) : *resource* ;

search all outputs which are not yet reaching target to find a pair of input and output = produces[s] where shifting input time to producing that output would produce the best result. Return the input as the function result and the technique to use in the parameter s

```

var
    Let gain ∈ volume;
    Let m ∈ resource;
    Let j, t ∈ technique;
    Let d, Scoop ∈ resource;
begin
    Scoop ← A;
    m ← 0;
    gain ← - maxint ;
    s ←  $\alpha$ ;
    for d ← agriculture to techniques.amusements do
    begin
        if startingresourced > 0 then
        begin

```

find Scoop

that is to say we iterate over all goods used as inputs

m stands for Main output of
d

$m \leftarrow produces_{mostProductiveTechniqueFor(d)}$

```

    for j ←  $\alpha$  to techniques. $\omega$  do
        if  $equated \uparrow [produces_j] > 0$  then

```

find a technique j, whose output is doing ok that potentially could have resource reallocated to make d

```

begin
  if marginalgain (j, d) > gain then
    begin

```

if this is the best marginal gain so far, record the technique *j* and the input *d*

```

      if equated↑[m] < 1 then
        begin
          gain ← marginalgain (j, d);
          { writeln('gain',gain,'j',j,'d',d,'m',m); }
          Scoop ← d;
          s ← j;
        end ;
      end ;
    end ;
  end ;
  { writeln('best gain',gain,'s',s,'scoop',scoop,'best output from scoop',m); }
  findScoop ← Scoop;
end ;

```

26 csvfilereader

```
unit csvfilereader ;
```

This parses csv files meeting the official UK standard for such files The following text is imported from that definition at <https://www.ofgem.gov.uk/sites/default/files/docs/2013/01/csvfilefor>

27 Introduction

27.1 Background

The comma separated values (CSV) format is a widely used text file format often used to exchange data between applications. It contains multiple records (one per line), and each field is delimited by a comma.

27.2 CSV File Format

The primary function of CSV file is to separate each field values by comma separated and transport text - based data to one or more target application. A source application is one which creates or appends to a CSV file and a target application is one which reads a CSV file

27.2.1 CSV File Structure

The CSV file structure use following two notations

FS (Field Separator) i.e. comma separated

FD (Field Delimiter) i.e. Always use a double - quote.

Each line feed in CSV file represents one record and each line is terminated by any valid NL (New line i.e. Carriage Return (CR) ASCII (13) and Line Feed (LF) ASCII (10)) feed. Each record contains one or more fields and the fields are separated by the FS character (i.e. Comma) A field is a string of text characters which will be delimited by the FD character (i.e. double - quote (")) Any field may be quoted (with double quotes).

Fields containing a line - break, double - quote, and/or commas should be quoted. (If they are not, the file will likely be impossible to process correctly).

The FS character (i.e. comma) may appear in a FD delimited field and in this case it is not treated as the field separator. If a field's value contains one or more commas, double - quotes, CR or LF characters, then it MUST be delimited by a pair of double - quotes (ASCII 0x22).

DO NOT apply double - quote protection where it is not required as applying double quotes on every field or on empty field would takes more file space If a field requires Excel protection, its value MUST be prefixed with a single tilde character .

See example below:

FS =,

FD ="

Data Record:

Test1,Test2,, "Test3,Test4", "Test5 ~"Test6"~ Test7", "Test8, "" ,", Test9"

Indicates the following four fields

Test1	5 characters
Test2	5 characters
	0 characters
Test3,Test4	11 characters
Test5 ~"Test6"~ Test7	20 characters
Test8, "	8 characters
,Test9	6 characters

28 CSV File Rules

- The file type extension MUST be set to .CSV
- The character set used by data contained in the file MUST be an 8 - bit (UTF - 8).
- No binary data should be transported in CSV file.
- A CSV file MUST contain at least one record.
- No limit to the number of data records
- The End of Record must be set to CR +LF (i.e. Carriage Return and Line Feed)
- Do not use whitespaces in the file name
- The EOR marker MUST NOT be taken as being part of the CSV record
- EOF (End of File) character indicates a logical EOF (SUB - ASCII 0x1A) and not the physical end .
- A logical EOF marker cannot be double - quote protected.
- Any record appears after the EOF will be ignored

28.1 File Size

Maximum csv file size should be 30 MB.

28.2 CSV Records

A CSV record consists of two elements, a data record followed by an end - of - record marker (EOR). The EOR is a data record delivery marker and does not form part of the data delivered by the record

29 CSV Record Rules

Pls. note this rule applies to every CSV record including the last record in the file.

29.1 CSV Field Column Rules

- Each record within the same CSV file MUST contain the same number of field columns . The header record describes how many fields the application should expect to process.
- Field columns MUST be separated from each other by a single separation character
- A field column MUST NOT have leading or trailing whitespace

29.2 Header Record Rules

A header record allows the Ofgem IT systems to guard against the potential issues such as missing column or additional column that are not in scope

- The header record MUST be the first record in the file.
- A CSV file MUST contain one header record only .
- Header labels MUST NOT be blank.
- Use single word only
- Do not use spaces (Use _ if words needs to be separated)

interface

const

textlen =80;

type

pcsv = ^ *csvcell* ;

celltype =(*linestart* ,*numeric* ,*alpha*);

textfield =*textline* ;

csvcell = **record**

right : *pcsv*;

case *tag* : *celltype* **of**

linestart : (*down* : *pcsv*);

numeric : (*number* : *real*);

α : (*textual* : *pstring*);

end ;

headervec (**max** :*integer*) =**array** [1..**max**]**of** *pcsv* ;

pheadervec = ↑ *headervec* ;

procedure *printcsv* (**var** *f* :*text* ;*p* :*pcsv*); (see Section ??)

function *parsecsvfile* (**name** :*textline*):*pcsv* ; (see Section ??)

function *rowcount* (*p* :*pcsv*):*integer* ; (see Section ??)

function *getdatamatrix* (*p* :*pcsv*): ^ *matrix* ; (see Section 30)

function *getcell* (*p* :*pcsv* ;*row* ,*col* :*integer*):*pcsv* ; (see Section ??)

function *getrowheaders* (*p* :*pcsv*): ^ *headervec* ; (see Section ??)

function *getcolheaders* (*p* :*pcsv*): ^ *headervec* ; (see Section ??)

function *colcount* (*p* :*pcsv*):*integer* ; (see Section ??)

returns nil for file that can not be opened, otherwise returns pointer to tree of csvcells.

implementation

const

FD = 34;

FS = 44;

RS = 10;

field delimiter

field separator

record separator


```

    EOI = $1a;
    CR = 13;
type
    token = (FDsym);
    tokenset = set of token ;
var
    categorisor: array [byte] of token;

function getdatamatrix ( p :pcsv ): ^ matrix ; (see Section 30 )

```

30 getdatamatrix

```

function getdatamatrix ( p :pcsv ): ^ matrix ;

```

extract the column headers as a vector of strings

```

var
    m : ↑ matrix ;
procedure recursedown ( j :integer ;q :pcsv ); (see Section 31 )

```

31 recursedown

```

procedure recursedown ( j :integer ;q :pcsv );
procedure recurse ( i :integer ;q :pcsv ); (see Section 32 )

```

32 recurse

```

procedure recurse ( i :integer ;q :pcsv );
begin

    if q ≠ nil then
    begin
        if i ≥ 1 then
        begin
            if q↑.tag = numeric then
                m↑[j, i] ← q↑.number
            else m↑[j, i] ← 0.0
            end ;
            recurse ( i + 1, q↑.right);
        end
    end ;
end ;

begin

```

```

    if  $q \neq nil$  then
    begin
        recurse (0,  $q \uparrow .right$ );
        recursedown ( $j + 1$ ,  $q \uparrow .down$ );
    end
end ;
begin

    if  $p = nil$  then  $getdatamatrix \leftarrow nil$ 
    else
    begin
        new (  $m$ ,  $rowcount ( p ) - 1$ ,  $colcount ( p ) - 1$  );
        recursedown (1,  $p \uparrow .down$ );
         $getdatamatrix \leftarrow m$ ;
    end ;
end ;
function  $getcolheaders ( p : pcsv ) : ^ headervec$  ; (see Section 33 )

```

33 getcolheaders

```

function  $getcolheaders ( p : pcsv ) : ^ headervec$  ;

```

extract the column headers

```

var
     $M$ ;
     $h : \uparrow headervec$  ;
procedure  $recurse ( i : integer ; q : pcsv )$  ; (see Section 34 )

```

34 recurse

```

procedure  $recurse ( i : integer ; q : pcsv )$ ;
begin
    if  $q \neq nil$  then
    begin

        if  $i \geq 1$  then  $h \uparrow [i] \leftarrow q$ ;
        recurse ( $i + 1$ ,  $q \uparrow .right$ );
    end
end ;
begin

    if  $p = nil$  then  $getcolheaders \leftarrow nil$ 
    else
    begin

```

```

        new ( h , colcount ( p )-1);
        recurse (0, p↑.right);
        getcolheaders← h;
    end ;
end ;
function getrowheaders ( p :pcsv ): ^ headervec ; (see Section 35 )

```

35 getrowheaders

```

function getrowheaders ( p :pcsv ): ^ headervec ;

```

extract the rows headers

```

var
    M;
    h : ↑ headervec ;
procedure recurse ( i :integer ;q :pcsv ); (see Section 36 )

```

36 recurse

```

procedure recurse ( i :integer ;q :pcsv );
begin
    if q ≠ nil then
        begin
            h↑[i]← q↑.right;
            recurse ( i + 1, q↑.down);
        end
    end ;
begin
    if p = nil then getrowheaders← nil
    else
        begin
            new ( h , rowcount ( p )-1);
            recurse (1, p↑.down);
            getrowheaders← h;
        end ;
    end ;
function colcount ( p :pcsv ):integer ; (see Section 37 )

```

37 colcount

```

function colcount ( p :pcsv ):integer ;

```

return the number of columns in the spreadsheet

```
begin
  if  $p = nil$  then  $colcount \leftarrow 0$ 
  else
    case  $p \uparrow .tag$  of
       $linestart : colcount \leftarrow colcount (p \uparrow .right);$ 
    end
  end ;
function  $getcell ( p : pcsv ; row , col : integer ) : pcsv ;$  (see Section 38 )
```

38 getcell

```
function  $getcell ( p : pcsv ; row , col : integer ) : pcsv ;$ 
```

return the cell at position row,col in the spreadsheet

```
begin
  if  $p = nil$  then  $getcell \leftarrow nil$ 
  else if  $row = 1$  then
    begin
      else if  $col = 1$  then  $getcell \leftarrow p$ 
    end
  end ;

procedure  $removetrailingnull ( var p : pcsv );$  (see Section 39 )
```

39 removetrailingnull

```
procedure  $removetrailingnull ( var p : pcsv );$ 
function  $onlynulls ( q : pcsv ) : boolean ;$  (see Section 40 )
```

40 onlynulls

```
function  $onlynulls ( q : pcsv ) : boolean ;$ 
begin
  if  $q = nil$  then  $onlynulls \leftarrow false$  false
  else
    if  $q \uparrow .tag = \alpha$  then
      begin
        end
      else  $onlynulls \leftarrow false$  false
    end ;
```

```

begin
  if  $p \neq nil$  then
    case  $p \uparrow .tag$  of
      linestart :
        or ( (  $p \wedge .down = nil$  ) and onlynulls (  $p \wedge .right$  )) then  $p := nil$ 
        else removetrailingnull ( $p \uparrow .down$ );
      end
    end ;
function rowcount (  $p : pcsv$  ):integer ; (see Section 41 )

```

41 rowcount

```

function rowcount (  $p : pcsv$  ):integer ;
begin
  if  $p = nil$  then rowcount  $\leftarrow$  0
  else
    case  $p \uparrow .tag$  of
      linestart : rowcount  $\leftarrow$  1 + rowcount ( $p \uparrow .down$ );
      numeric  $\leftarrow$  1
    end
  end ;
function isint (  $r : real$  ):boolean ; (see Section 42 )

```

42 isint

```

function isint (  $r : real$  ):boolean ;
var
   $i : integer$ ;
begin
   $i \leftarrow \text{round}(r)$ ;
  isint  $\leftarrow$  ( $i \times 1.0$ ) =  $r$ 
end ;
procedure printcsv ( var  $f : text$  ;  $p : pcsv$  ); (see Section 43 )

```

43 printcsv

```

procedure printcsv ( var  $f : text$  ;  $p : pcsv$  );
begin
  if  $p \neq nil$  then
    with  $p \uparrow$  do
      begin
        if  $tag = linestart$  then
          begin
            printcsv ( $f, right$ );
            if  $down \neq nil$  then

```

```

        begin
            writeln(f);
            printcsv (f, down);
        end ;
    end
else
    if tag = numeric then
        begin
            else write(f, number : 1 : 6);
            if right ≠ nil then
                begin
                    write ( f , ',' );
                end
            end
        end
    else
        if tag = α then
            begin
                if textual ≠ nil then write(f, "" , textual↑, "" ) else write(f, 'nil' );
                if right ≠ nil then
                    begin
                        write ( f , ',' );
                    end
                end
            end
        end
    end
end ;
function parsecsvfile ( name :textfield ):pcsv ; (see Section 44 )

```

44 parsecsvfile

```

function parsecsvfile ( name :textfield ):pcsv ;
const
    megabyte = 1024 × 1024;
    maxbuf = 30 × megabyte;
type
    bytebuf = array [1..maxbuf ] of byte ;
var
    f : fileptr;
    bp : ↑ bytebuf ;
    fs;
    tokstart;
    firstfield;
function thetoken :token ; (see Section 45 )

```

45 thetoken

```
function thetoken :token ;  
begin  
  if currentchar  $\leq$  fs then  
    thetoken  $\leftarrow$  categorisorbp $\uparrow$ [currentchar]  
  else thetoken  $\leftarrow$  EOFsym  
end ;  
function peek ( c :token ):boolean ; (see Section 46 )
```

46 peek

```
function peek ( c :token ):boolean ;
```

matches current char against the token c returns true if it matches.

```
begin  
  peek  $\leftarrow$  c = thetoken  
end ;  
function isoneof ( s :tokenset ):boolean ; (see Section 47 )
```

47 isoneof

```
function isoneof ( s :tokenset ):boolean ;  
begin  
  isoneof  $\leftarrow$  thetoken  $\in$  s  
end ;  
procedure nextsymbol ; (see Section 48 )
```

48 nextsymbol

```
procedure nextsymbol ;  
begin  
  if currentchar  $\leq$  fs then currentchar  $\leftarrow$  currentchar + 1  
end ;  
function have ( c :token ):boolean ; (see Section 49 )
```

49 have

```
function have ( c :token ):boolean ;  
begin  
  if peek (c) then
```

```

    begin
        nextsymbol;
        have ← true;

    end
    else
        have ← false;
    end ;
function haveoneof ( c : tokenset ):boolean ; (see Section 50 )

```

50 haveoneof

```

function haveoneof ( c : tokenset ):boolean ;
begin
    if isoneof ( c ) then
        begin
            nextsymbol;
            haveoneof ← true;
        end
    else
        haveoneof ← false;
    end ;

procedure initialise ; (see Section 51 )

```

51 initialise

```

procedure initialise ;
begin
    firstfield ← nil;
    lastfield ← nil;
    firstrecord ← nil;

end ;
procedure resolvealpha ; (see Section 52 )

```

52 resolvealpha

```

procedure resolvealpha ;
var
    i;
begin
    with lastfield↑ do
        begin
            tag ←  $\alpha$ ;

```



```

    new ( textual );
    textual↑← " ";
    l← tokend min(tokstart + textlen - 1) ;
    { copy field to string }
    for i← tokstart to l - 1 do
    begin
        textual↑← textual↑ + chr(bp↑[i]) ;
    end ;
    end ;
end ;

```

procedure *resolvedigits* ; (see Section 53)

53 resolvedigits

```

procedure resolvedigits ;
var
    i;
    s : string;
begin
    with lastfield↑ do
    begin
        tag← numeric;
        new ( textual );
        s← " ";
        l← tokend min(tokstart + textlen - 1) ;
        { copy field to a string }
        for i← tokstart to l do
        begin
            s← s + chr(bp↑[i]);
        end ;
        val (s, number, l);
    end ;
end ;
procedure resolvetoken ; (see Section 54 )

```

convert to binary

54 resolvetoken

```

procedure resolvetoken ;
begin
    if chr(bp↑[tokstart]) in [ '0' .. '9' ] then resolvedigits
    else resolvealpha
end ;

procedure markbegin ; (see Section 55 )

```

55 markbegin

mark start of a field

```
procedure markbegin ;  
begin  
    tokstart ← currentchar;  
    new ( lastfield ^ .right );  
    lastfield ← lastfield↑.right;  
    lastfield↑.right ← nil;  
end ;  
procedure markend ; (see Section 56 )
```

56 markend

marks the end of a field

```
procedure markend ;  
begin  
    tokend ← currentchar;  
    resolvetoken;  
end ;  
procedure setalpha ( s : textfield ); (see Section 57 )
```

57 setalpha

```
procedure setalpha ( s : textfield );  
begin  
    lastfield↑.tag ←  $\alpha$ ;  
    new ( lastfield ^ .textual );  
    lastfield↑.textual↑ ← s;  
end ;  
procedure emptyfield ; (see Section 58 )
```

58 emptyfield

```
procedure emptyfield ;  
begin  
  
    markbegin;  
    setalpha ( ‘ ’ );  
  
end ;  
  
procedure parsebarefield ; (see Section 59 )
```

59 parsebarefield

```
procedure parsebarefield ;  
begin  
  if isoneof ([RSym, EOFsym, FSym]) then emptyfield  
else begin begin  
    markbegin;  
    while haveoneof ([any, space]) do ;  
    markend;  
  end ;  
end ;  
procedure parsedelimitedfield ; (see Section 60 )
```

skip over the field

60 parsedelimitedfield

```
procedure parsedelimitedfield ;
```

parses a field nested between " chars converting escape chars as it goes

```
var  
  s : textfield;  
  i : integer;  
  continue : boolean;  
procedure appendcurrentchar ; (see Section 61 )
```

61 appendcurrentchar

```
procedure appendcurrentchar ;  
begin  
   $s \leftarrow s + \text{chr}(bp \uparrow [\text{currentchar}])$ ;  
  nextsymbol;  
end ;  
begin  
  markbegin;  
   $s \leftarrow \text{' '}$  ;  
  continue  $\leftarrow$  true;  
  repeat  
    while isoneof ([FSym..any]) do  
      begin  
        appendcurrentchar;  
      end ;  
    have (FDsym);  
    continue  $\leftarrow$  peek (FDsym)  $\wedge$  (length (s) < textlen);  
    if continue then appendcurrentchar;  
  until (not continue );
```

eat what may be closing
quotes

```

    setalpha (s);
end ;
procedure parsefield ; (see Section 62 )

```

62 parsefield

```

procedure parsefield ;
begin
    if have (FDsym) then parsedelimitedfield
    else parsebarefield
end ;
procedure parserecord ; (see Section 63 )

```

63 parserecord

```

procedure parserecord ;
begin
    parsefield;
    while have (FSym) do parsefield;
end ;
procedure parseheader ; (see Section 64 )

```

64 parseheader

```

procedure parseheader ;
begin
    { claim heap space for start of first line }
    new ( firstrecord );
    lastfield ← firstrecord;
    firstfield ← firstrecord;
    with firstrecord↑ do
    begin
        tag ← linestart;
        down ← nil;
        right ← nil;
    end ;
    parserecord;
end ;
procedure parsewholefile ; (see Section 65 )

```

65 parsewholefile

```

procedure parsewholefile ;

```

```

begin
  parseheader;

  while have (RSsym) do
    begin
      { claim heap space for the start of the new line }
      new ( firstfield ^ .down );
      firstfield ← firstfield↑.down;
      lastfield ← firstfield;
      with firstfield↑ do
        begin
          tag ← linestart;
          down ← nil;
          right ← nil;
        end ;
      parserecord;
    end ;
  end ;
begin
  initialise;
  parsecsvfile ← nil;

  assign (f, name);
  reset (f);
  if ioresult = 0 then
    begin
      fs ← filesize (f);
      if fs < maxbuf then
        begin
          new ( bp );
          blockread (f, bp↑[1], fs, rc);
          if rc = fs then
            begin
              currentchar ← 1;

```

the default case of failure

open file for reading

ioresult = 0 if opened ok

We now have the csv file in memory - parse it

```

      parsewholefile;
      removetrailingnull (firstrecord);
      parsecsvfile ← firstrecord;
    end ;
  dispose ( bp );
  close (f);
end ;
end ;
begin
  categorisor ← any;

```

```

categorisor  $_{FD} \leftarrow FDsym$ ;
categorisor  $_{FS} \leftarrow FSsym$ ;
categorisor  $_{RS} \leftarrow RSsym$ ;
categorisor  $_{EOI} \leftarrow EOFsym$ ;
categorisor ord( ' , ' )  $\leftarrow space$ ;
categorisor  $_{CR} \leftarrow space$ ;
{writeln('fs=' , fs , 'fd=' , fd , 'rs=' , rs);
writeln(categorisor);}
end .

```