

labourValuation.pas

June 7, 2018

Contents

1	labourvaluation	3
2	Purpose	3
3	Comand line interface	3
3.1	iotable	3
4	rf	5
5	csvfilereader	5
6	Introduction	5
6.1	Background	5
6.2	CSV File Format	6
6.2.1	CSV File Structure	6
7	CSV File Rules	8
7.1	File Size	8
7.2	CSV Records	8
8	CSV Record Rules	8
8.1	CSV Field Column Rules	8
8.2	Header Record Rules	9
9	getdatamatrix	10
10	recursedown	10
11	recurse	10
12	getcolheaders	11
13	recurse	11
14	getrowheaders	12
15	recurse	12
16	colcount	12

17	getcell	13
18	removetrailingnull	13
19	onlynulls	13
20	rowcount	14
21	isint	14
22	printcsv	14
23	parsecsvfile	15
24	thetoken	16
25	peek	16
26	isoneof	16
27	nextsymbol	16
28	have	16
29	haveoneof	17
30	initialise	17
31	resolvealpha	17
32	resolvedigits	18
33	resolvetoken	18
34	markbegin	19
35	markend	19
36	setalpha	19
37	emptyfield	19
38	parsebarefield	20
39	parsedelimitedfield	20
40	appendcurrentchar	20
41	parsefield	21
42	parserecord	21
43	parseheader	21

1 labourvaluation

```
program labourValuation ;  
uses csvfilereader ;
```

2 Purpose

The programme applies the labour value theory to data from input output tables.

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 : labourValuation *iotable.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.

```

type
    pmat =  $\wedge$  matrix ;
    channel = record
        p : pcsv;
        r;
        m :  $\uparrow$  matrix ;
    end ;
procedure rf ( var ch : channel ; param : integer ); (see Section 4 )
var
    Let iot  $\in$  channel;
    Let L, V, O, O1  $\in$   $\wedge$ vector;
    Let square  $\in$   $\wedge$ matrix;
    Let i, j  $\in$  integer;
begin
    rf (iot, 1);

    new ( L , iot . m  $\wedge$  . cols );
    new ( V , iot . m  $\wedge$  . cols );
    new ( O , iot . m  $\wedge$  . cols );
    new ( O1 , iot . m  $\wedge$  . cols );
    L $\uparrow$  $\leftarrow$  1;
    V $\uparrow$  $\leftarrow$  iot . m $\uparrow$ [iot . m $\uparrow$ .rows - 1];
    O $\uparrow$  $\leftarrow$  iot . m $\uparrow$ [iot . m $\uparrow$ .rows];

    new ( square , iot . m  $\wedge$  . cols , iot . m  $\wedge$  . cols );
    square $\uparrow$  $\leftarrow$  (iot . m $\uparrow$ )T;

    for i $\leftarrow$  1 to 20 do
    begin

        o1 $\uparrow$  $\leftarrow$  V $\uparrow$  +  $\sum$  (L $\uparrow$   $\times$  square $\uparrow$  ) ;
        L $\uparrow$  $\leftarrow$  O1 $\uparrow$  / O $\uparrow$  ;
    end ;
    writeln(O1 $\uparrow$  : 12 : 1 , O $\uparrow$  : 12 : 1 );

end .

```

initialise all labour values to
 1
 extract the variable capital
 vector from the IO table
 Extract the final output
 price vector from the IO ta-
 ble

Create a transposed version
 of the IO table central ma-
 trix

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 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>

6 Introduction

6.1 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.

6.2 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

6.2.1 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

7 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

7.1 File Size

Maximum csv file size should be 30 MB.

7.2 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

8 CSV Record Rules

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

8.1 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

8.2 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 = \wedge *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 = \uparrow *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*): \wedge *matrix* ; (see Section 9)

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

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

function *getcolheaders* (*p* : *pcsv*): \wedge *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 9 )

```

9 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 10 )

```

10 recursedown

```

procedure recursedown ( j :integer ;q :pcsv );
procedure recurse ( i :integer ;q :pcsv ); (see Section 11 )

```

11 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 12 )

```

12 getcolheaders

```

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

```

extract the column headers

```

var
     $M$ ;
     $h : \uparrow headervec$  ;
procedure  $recurse ( i : integer ; q : pcsv )$  ; (see Section 13 )

```

13 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 14 )

```

14 getrowheaders

```

function getrowheaders ( p :pcsv ): ^ headervec ;

```

extract the rows headers

```

var
    M;
    h : ↑ headervec ;
procedure recurse ( i :integer ;q :pcsv ); (see Section 15 )

```

15 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 16 )

```

16 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 17 )
```

17 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 18 )
```

18 removetrailingnull

```
procedure  $removetrailingnull ( var p : pcsv );$ 
function  $onlynulls ( q : pcsv ) : boolean ;$  (see Section 19 )
```

19 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 20 )

```

20 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 21 )

```

21 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 22 )

```

22 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 23 )

```

23 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 24 )

```

24 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 25 )
```

25 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 26 )
```

26 isoneof

```
function isoneof ( s :tokenset ):boolean ;  
begin  
  isoneof  $\leftarrow$  thetoken  $\in$  s  
end ;  
procedure nextsymbol ; (see Section 27 )
```

27 nextsymbol

```
procedure nextsymbol ;  
begin  
  if currentchar  $\leq$  fs then currentchar  $\leftarrow$  currentchar + 1  
end ;  
function have ( c :token ):boolean ; (see Section 28 )
```

28 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 29 )

```

29 haveoneof

```

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

procedure initialise ; (see Section 30 )

```

30 initialise

```

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

end ;
procedure resolvealpha ; (see Section 31 )

```

31 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 32)

32 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 33 )

```

convert to binary

33 resolvetoken

```

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

procedure markbegin ; (see Section 34 )

```

34 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 35 )
```

35 markend

marks the end of a field

```
procedure markend ;  
begin  
    tokend ← currentchar;  
    resolvetoken;  
end ;  
procedure setalpha ( s : textfield ); (see Section 36 )
```

36 setalpha

```
procedure setalpha ( s : textfield );  
begin  
    lastfield↑.tag ←  $\alpha$ ;  
    new ( lastfield ^ .textual );  
    lastfield↑.textual↑ ← s;  
end ;  
procedure emptyfield ; (see Section 37 )
```

37 emptyfield

```
procedure emptyfield ;  
begin  
  
    markbegin;  
    setalpha ( ' ' );  
  
end ;  
  
procedure parsebarefield ; (see Section 38 )
```

38 parsebarefield

```
procedure parsebarefield ;  
begin  
  if isoneof ([RSsym, EOFsym, FSsym]) then emptyfield  
else begin begin  
    markbegin;  
    while haveoneof ([any, space]) do ;  
    markend;  
  end ;  
end ;  
procedure parsedelimitedfield ; (see Section 39 )
```

skip over the field

39 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 40 )
```

40 appendcurrentchar

```
procedure appendcurrentchar ;  
begin  
  s ← s + chr(bp↑[currentchar]);  
  nextsymbol;  
end ;  
begin  
  markbegin;  
  s ← ' ' ;  
  continue ← true;  
  repeat  
    while isoneof ([FSsym..any]) do  
      begin  
        appendcurrentchar;  
      end ;  
    have (FDsym);  
    continue ← peek (FDsym) ∧ (length (s) < textlen);  
    if continue then appendcurrentchar;  
  until (not continue );
```

eat what may be closing
quotes

```

    setalpha (s);
end ;
procedure parsefield ; (see Section 41 )

```

41 parsefield

```

procedure parsefield ;
begin
    if have (FDsym) then parsedelimitedfield
    else parsebarefield
end ;
procedure parserecord ; (see Section 42 )

```

42 parserecord

```

procedure parserecord ;
begin
    parsefield;
    while have (FSym) do parsefield;
end ;
procedure parseheader ; (see Section 43 )

```

43 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 44 )

```

44 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 .

```