# harmonyplan.pas

December 11, 2018

## Contents

# 1 harmonyplan

**program** *harmonyplan* ;
**uses** *technologies* ,*harmony* ,*csvfilereader* ;

**type**
*pmat* =^ *matrix* ;
*channel* = **record**
*p* : *pcsv*;
*r* : *pheadervec*;
*c* : *pheadervec*;
*m* : ↑ *matrix* ;
**end** ;
**var**
matrices: **array** [1..4] **of** *pmat* ;
**procedure** *rf* ( **var** *ch* :*channel* ;*param* :*integer* ); (see Section 2 )

**var**
Let *flows*, *caps*, *deps*, *targs* ∈ channel;
Let *outputs*, *compressedDeprates*, *labour*, *initialResource*, *targets*, *intensities* ∈ pvec;
Let *i*, *j*, *k*, *lr*, *y*, *year*, *maxprod*, *years*, *capitals*, *cn* ∈ integer;
Let *yearXproductIntensityIndex* ∈ ^matrix;
Let *relativecapnum* ∈ ^matrix;
Let *flow* ∈ real;
Let *t* ∈ ptechnique;
Let *inputs*, *toutputs* ∈ presourcevec;
Let *capnumtoflownum* ∈ pintvec;
**var**
Let *C* ∈ ^TechnologyComplex ;
Let *si*, *sj*, *sy* ∈ string[10];
Let *start*, *stop* ∈ double;
**function** *deprate* ( *capitaltype* :*integer* ):*real* ; (see Section 3 )
**function** *countyears* ( *heads* :*pheadervec* ) :*integer* ; (see Section 4 )
**function** *outputrowinheaders* :*integer* ; (see Section 5 )
**function** *labourRow* :*integer* ; (see Section 6 )
**function** *capnum* ( *prod* , *year* , *maxcap* :*integer* ) :*integer* ; (see Section 7 )
**function** *countnonzero* ( **var** *m* :*matrix* ):*integer* ; (see Section 8 )
**function** *countinputsTo* ( *industry* :*integer* ) :*integer* ; (see Section 9 )
**function** *flownum* ( *prod* , *year* :*integer* ):*integer* ; (see Section 10 )
**function** *capname* ( *row* ,*col* ,*year* :*integer* ):**string** ; (see Section 11 )
**function** *productName* ( *prod* , *year* ,*internalcode* :*integer* ) :**string** ; (see Section 12 )

```
procedure flowsourcedfromTo ( year ,row ,col :integer ); (see Section 13 )
procedure flowsOriginatingIn ( year :integer ); (see Section 14 )
procedure printResults ( var c :TechnologyComplex ; var intensity , initialResource :vector ) ; (see Section 15
procedure setupintertermporalflow ; (see Section 16 )

var
    Let tv ∈ ptvec;
begin
    rf (flows, 1);
    rf (caps, 2);
    rf (deps, 3);
    rf (targs, 4);
    // go through the targets matrix and make sure no targets are actually zero - make them very
    small positive amounts
    for i← 1 to targs.m↑.rows do
        for j← 1 to targs.m↑.cols do
            if (targs.m↑[i]) then targs.m↑[i]← 1 - Harmony.capacitytarget;
    new ( outputs ,flows .m ^ .cols );
    new ( labour ,flows .m ^ .cols );
    outputs↑← flows.m↑[outputrowinheaders];
    labour↑← flows.m↑[labourRow];
    years← countyears (targs.r);

    maxprod← flows.m↑.cols;
    new ( yearXproductIntensityIndex ,years ,maxprod +2);

    capitals← countnonzero (caps.m↑);
    //writeln ( 'maxprod' , maxprod , 'capitals' , capitals , ' years' , years );
    // work out how many products the harmonizer will have to solve for
    // assume that we have N columns in our table and y years then
        // we have Ny year product combinations
        // in addition we have y labour variables
        // and caps .y capital stocks

        new ( C );
    //writeln ( 'call definecomplex(' ,( maxprod +capitals )*years , ')' );
    definecomplex (C↑, (maxprod + capitals + 1) × years);
    // writeln ( "productnum "+C .productCount ( )+" years "+years );
    // Assign identifiers to the outputs
    for i← 1 to maxprod + 1 do
        for year← 1 to years do
            addproduct (C↑, productName (i, year, flownum (i, year)), flownum (i, year));

    for i← 1 to maxprod do
        for j← 1 to maxprod do
            for year← 1 to years do
                if (caps.m↑[i]) then
                    addproduct (C↑, capname (i, j, year), capnum (round(relativecapnum↑[i]), year, capitals));
```

```
for year← 1 to years do
begin
  // add a production technology for each definite product
  for i← 1 to maxprod do
  begin
    // writeln ( 'product ' ,i , ' has ' ,countinputsTo ( i ), ' inputs to it ' );
    new ( inputs ,countinputsTo ( i ));
    new ( toutputs , 1);
    j← 1;
include labour
    for k← 1 to maxprod + 1 do
    begin
      if (flows.m↑[k]) then
      begin
        flow← flows.m↑[k];
        inputs↑[j].quantity← flow;
        inputs↑[j].product← findproduct (C↑, productName (k, year, flownum (k, year)));
        j← j + 1;
      end ;
      if (k ≤ maxprod) then  / / no labour row  for the capital matrix so we miss last row
        if (caps.m↑[k]) then
        begin
          flow← caps.m↑[k];
          inputs↑[j].quantity← flow;
          inputs↑[j].product← findproduct (C↑, capName (k, i, year));
          j← 1 + j;
        end ;
    end ;
    with toutputs↑[1] do
    begin
      quantity← outputs↑[i];
      { product:=C↑.index[flownum(i,year)];}
      product← findproduct (C↑, productName (i, year, flownum (i, year)));
    end ;
    //writeln ( 'year ' ,year , 'of' ,years , 'first call define techniques' );
    t← defineTechnique (C↑, inputs↑, toutputs↑);
    yearXproductIntensityIndex↑[year]← C↑.techniqueCount;
  end ;

end ;

setupintertermporalflow;
```

now set up the initial resource vector

```
new ( initialResource ,C ^ .productCount );
```

put in each years labour

```
lr← labourRow;


for y← 1 to years do
begin
    initialResource↑[flownum (lr, y)]← targs.m↑[y];
    C↑.nonproduced↑[flownum (lr, y)]← true;
    C↑.nonfinal↑[flownum (lr, y)]← true;
end ;
```

put in each years initial capital stock allowing for depreciation

```
for i← 1 to caps.m↑.rows do
begin
    for j← 1 to caps.m↑.rows do
    begin
        if (caps.m↑[i]) then
            for y← 1 to years do
            begin
                cn← capnum (round(relativecapnum↑[i]), y, capitals);
                if (verbose) then writeln ( i , ',' ,j , ',' ,y , ',' ,cn );
                if (y = 1) then C↑.nonproduced↑[cn]← true;
                C↑.nonfinal↑[cn]← true;
                initialResource↑[cn]← caps.m↑[i];
            end
    end
end ;
```

now set up the target vector

```
new ( targets ,C ^ .productCount );
// initialise to very small numbers to prevent divide by zero

targets↑← 0.03;
for y← 1 to years do
    for j← 1 to targs.m↑.cols - 1 do
        {do not include the labour col of the targets}
        targets↑[flownum (j, y)]← targs.m↑[y];
```

```
    if (verbose) then
    begin
        logComplex (C↑);
    end ;

    start← secs;

    intensities← balancePlan (targets↑, initialResource↑, C↑);
    stop← secs;
    printResults (C↑, intensities↑, initialResource↑);
    writeln( 'took ' , ((stop - start) × 0.01), ' sec' );


end .
```

## 2 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);
        matrices_param← m;
    end ;
end ;
```

## 3 deprate

capitaltype is the compressed capital index

```
function deprate ( capitaltype :integer ):real ;
begin
    deprate← compressedDeprates↑[capitaltype];
end ;
```

# 4 countyears

```
function countyears ( heads :pheadervec ) :integer ;
var
    Let i, j ∈ integer;
begin
    j← 0;
    for  i← 1  to  targs.r↑.max  do
    begin

        if  (heads↑[i] ≠ nil)  then
            then
                j← j + 1;
    end ;
    countyears← j;
end ;
```

# 5 outputrowinheaders

```
function outputrowinheaders :integer ;
var
    Let i, j ∈ integer;
begin
    j← 0;
    for  i← 1  to  flows.r↑.max  do
        if  (flows.r↑[i].textual↑ = ( 'output'  ) )  then  j← i;
    if  j = 0  then
    begin
        writeln( 'no output row found in flow matrix'  );
        halt (301);
    end ;
    outputrowinheaders← j;
end ;
```

# 6 labourRow

```
function labourRow :integer ;
var
    Let i, j ∈ integer;
begin
    j← 0;
    for  i← 1  to  flows.r↑.max  do
        if  (flows.r↑[i].textual↑ = ( 'labour'  ) )  then  j← i;
    if  j = 0  then
    begin
        writeln( 'no labour row found in flow matrix'  );
        halt (301);
```

```
    end ;
    labourrow← j;
end ;
```

# 7 capnum

```
function capnum ( prod , year , maxcap :integer ) :integer ;
begin
    capnum← (prod) + (year - 1) × (maxcap) + years × (maxprod + 1);
end ;
```

# 8 countnonzero

```
function countnonzero ( var m :matrix ):integer ;
var
    Let t, i, j ∈ integer;
begin
    t← 1;
    new ( relativecapnum ,m .rows ,m .cols );
    for i← 1 to m.rows do
        for j← 1 to m.cols do
            if (m_{i,j} > 0) then
            begin
                relativecapnum↑[i]← t;
                t← t + 1;
            end ;
            new ( capnumtoflownum ,t -1);
            new ( compressedDeprates ,t -1);
            // pass through again filling in the backwardvector
            //writeln ( t );
            t← 1;
            for i← 1 to m.rows do
                for j← 1 to m.cols do
                    if (m_{i,j} > 0) then
                    begin
                        // writeln ( i ,j ,t );
                        capnumtoflownum↑[t]← i;
                        compressedDeprates↑[t]← deps.m↑[i];
                        t← t + 1;
                    end ;
                    countnonzero← t - 1;
end ;
```

## 9 countinputsTo

```
function countinputsTo ( industry :integer ) :integer ;
var
   Let total, i ∈ integer;
begin
   total← 0;
   for i← 1 to maxprod + 1 do
      if (flows.m↑[i]) then total← total + 1;
   for i← 1 to maxprod do
      if (caps.m↑[i]) then total← total + 1;
   countinputsTo← total;
end ;
```

## 10 flownum

```
function flownum ( prod , year :integer ):integer ;
begin
   flownum← (prod) + (year - 1) × (maxprod + 1);
end ;
```

## 11 capname

```
function capname ( row ,col ,year :integer ):string ;
begin
   capname← 'C[' + int2str (row) + ']['  + int2str (col) + ']Y'  + int2str (year);
end ;
```

## 12 productName

```
function productName ( prod , year ,internalcode :integer ) :string ;
begin
   productname← flows.r↑[prod].textual↑ + 'Y'  + int2str (year) + '{'  + int2str (internalcode)
   + '}'  ;
end ;
```

## 13 flowsourcedfromTo

```
procedure flowsourcedfromTo ( year ,row ,col :integer );
```

Generate investment technique starting from the specified year, directed at the specified row and col , with possible joint production

```
var
    Let src, dest ∈ presourcevec;
    Let outputyears, i ∈ integer;
    Let t ∈ ptechnique;
begin


    outputyears← years - year;
    if outputyears > 0 then
    begin
        new ( src ,1);
        src↑[1].product← findproduct (C↑, productName (row, year, flownum (row, year)));
        src↑[1].quantity← 1;
        new ( dest ,outputyears );
        for i← 1 to outputyears do
            with dest↑[i] do
            begin
                product← findproduct (C↑, capName (row, col, year + i));
                quantity← (1 - deps.m↑[row])^(i−1);
            end ;
            t← defineTechnique (C↑, src↑, dest↑);
            // dispose ( dest );
            dispose ( src );
    end ;
end ;
```

# 14 flowsOriginatingIn

```
procedure flowsOriginatingIn ( year :integer );
```

This generates all investment flows generated in 'year' .

```
var
    Let r, c ∈ integer;
begin
    for r← 1 to maxprod do
        for c← 1 to maxprod do
            if caps.m↑[r] then flowsourcedfromTo (year, r, c);
end ;
```

# 15 printResults

```
procedure printResults ( var c :TechnologyComplex ; var intensity , initialResource :vector ) ;
var
```

```
    Let netoutput, gross, usage, produced ∈ pvec;
    Let toth ∈ real;
    Let year ∈ integer;
procedure writecsvln ( var s :headervec ) ; (see Section 17 )
procedure writecsvvec ( var s :vector ) ; (see Section 18 )
var
    Let row, col, index ∈ integer;
    Let howmuch, h ∈ real;
begin
    netoutput← computeNetOutput (C, intensity, initialResource);
    gross← computeGrossAvail (C, intensity, initialResource);
    writeln ( 'iter, useweight, phase2, temp' );
    writeln ( ' ' ,iters , ',' ,useweight , ',' ,phase2adjust , ',' , startingtemp );
    write ( 'year,headings' );
    writecsvln (flows.c↑);
    toth← 0;
    for year← 1 to years do
        begin new ( usage ,maxprod );
        new ( produced ,maxprod +1);

        writeln ( year , ',flow matrix' );
        for row← 1 to outputrowinheaders do

        begin
            write(year);
            write ( ',' ,flows .r ^ [row ].textual ^ );
            for col← 1 to flows.c↑.max do

            begin
                index← round(yearXproductIntensityIndex↑[year]);
                howmuch← intensities↑[index] × flows.m↑[row];
                write ( ',' ,howmuch );
                if (row < maxprod) then
                    begin usage ^ [row ]:= usage ^ [row ]+howmuch ;
                end
                else
                    begin produced ^ [col ]:=howmuch ;
                end ;
            end ;
            writeln( '' );
        end ;

        write ( year , ',' );
        write( 'productive consumption' );
        writecsvvec (usage↑);
        write ( year , ',' );
        write( 'accumulation ' );
        for col← 1 to usage↑.cols do
        begin
            write ( ',' ,( produced ^ [col ]-netoutput ^ [flownum ( col ,year )]-usage ^ [col ]));
```

```
        end ;
        writeln( '' );
        write ( year , ',' );
        write( 'netoutput ' );
        for col← 1 to flows.c↑.max do
        begin
            write ( ',' ,netoutput ^ [flownum ( col ,year )]);
        end ;
        writeln( '' );
        write ( year , ',' );
        write( 'target ' );
    for col← 1 to flows.c↑.max do begin begin
            write ( ',' ,targs .m ^ [year ][col ]);
        end ;
        writeln( '' );
        write ( year , ',' );
        write( 'netoutput/target ' );
    for col← 1 to flows.c↑.max do begin begin
            write ( ',' ,( netoutput ^ [flownum ( col ,year )]/targs .m ^ [year ][col ]));
        end ;
        writeln( '' );

        write ( '' ,year , ',' );
        write( 'harmony ' );

    for col← 1 to flows.c↑.max do begin begin
            h← Harmony.H (targs.m↑[year], netoutput↑[flownum (col, year)]);
            write ( ',' ,h );
            toth← h + toth;
        end ;
        writeln( '' );
        writeln ( '' ,year , ',capital use matrix' );
        for row← 1 to labourrow - 1 do
        begin
            write( '' , year);
            write ( ',' ,flows .r ^ [row ].textual ^ );
        for col← 1 to flows.c↑.max do begin begin
                index← round(yearXproductIntensityIndex↑[year]);
                howmuch← intensities↑[index];
                write ( ',' ,howmuch *caps .m ^ [row ][col ]);
            end ;
            writeln( '' );
        end ;
        writeln( '' );
    end ;
    writeln ( 'totalharmony ,' ,toth );
end ;
```

# 16 setupintertermporalflow

**procedure** *setupintertermporalflow* ;

the aim of this procedure is to create techniques which represent investment flows, in general these will be joint production techniques. We will have one technique for each type of non zero capital good, for each year other than the last one.

```
var
   Let y ∈ integer;
begin
   for y← 1 to years do flowsOriginatingIn (y);
end ;
```

# 17 writecsvln

```
procedure writecsvln ( var s :headervec ) ;
var
   Let i ∈ integer;
   Let c ∈ csvcell;
begin
   for i← 1 to s.max do
   begin
      c← s_i↑;
      with c do
         write ( ',' ,textual ^ );
   end ;
   writeln;
end ;
```

# 18 writecsvvec

```
procedure writecsvvec ( var s :vector ) ;
var
   Let i ∈ integer;
begin
   for i← 1 to s.cols do
   begin
      write ( ',' ,s [i ]);
   end ;
   writeln;
end ;
```

# 19 threadlib

# 20 technologies

A library to represent a set of production technologies in a more compact form than as an input output table or matrix. It can take advantage of the sparse character of large io tables.



```
unit technologies ;
interface

    const
        namelen = 35;
    type
        pvec =^ vector ;
        resourceid = string [namelen ];
        ptechniquelist = ^ techniquelist ;
        resourcerec = record
            id : resourceid;
            productNumber : integer;
```

```
        users;
    end ;
    presource = ↑ resourcerec ;
    resourceindex ( max :integer )=array [1..max ] of presource ;
    presourceindex = ↑ resourceindex ;
    iopair = record
        product : presource;
        quantity : real;
    end ;
    resourcevec ( max :integer )=array [1..max ] of iopair ;
    presourcevec = ↑ resourcevec ;
    technique = record
        produces;
        techniqueno : integer;
    end ;
    ptechnique = ↑ technique ;
    techniquelist = record
        tech : ptechnique;
        next : ptechniquelist;
    end ;
    pproductlist = ↑ productlist ;
    productlist = record
        product : presource;
        next : pproductlist;
    end ;
Hash table of products
    productindex ( max :integer )=array [0..max ] of pproductlist ;
    pproductindex = ↑ productindex ;
    intvec(maxi:integer )= array [1..maxi] of integer;
    pintvec = ↑ intvec ;
    techvec(maxt:integer )= array [1..maxt] of ptechnique;
    ptvec = ↑ techvec ;
    producervec(maxv:integer )= array [1..maxv] of ptvec;
    pdvec = ↑ producervec ;
    bvec ( max :integer )= array [1..max ] of boolean ;
    pbvec = ↑ bvec ;
    pcomplex = ↑ technologycomplex ;
    technologycomplex = record
        techniqueslist : ptechniquelist;
        techniquesvec : ptvec;
        index : pproductindex;
        producerIndex;
        nonfinal : pbvec;
        nonproduced : pbvec;
        techniquecount;

        allresourceindex : presourceindex;
    end ;
    tc = technologycomplex;

procedure logComplex ( var ct :tc ); (see Section 21 )
```

```
    function techniques ( var ct :tc ):ptvec ; (see Section 22 )
    function produces ( var ct :tc ;t :technique ;productNumber :integer ):boolean ; (see Section 23 )
    function buildProducerIndex ( var ct :tc ):pdvec ; (see Section ?? )
    function buildUserIndex ( var ct :tc ):pdvec ; (see Section ?? )
    function buildIndex ( var ct :tc ;produces :boolean ):pdvec ; (see Section ?? )
    function defineTechnique ( var ct :tc ;var inputs ,outputs :resourcevec ):ptechnique ; (see Section ?? )
    procedure addproduct ( var ct :tc ; name :string ;number :integer ); (see Section ?? )
    function findproduct ( var ct :tc ; name :string ):presource ; (see Section ?? )
    function defineResource ( var ct :tc ;name :resourceid ;number :integer ):presource ; (see Section ?? )
    function defineproductlist ( var ct :tc ;name :string ; p :pproductlist ;number :integer ):pproductlist ; (see

    procedure defineComplex ( var ct :tc ;numberofproducts :integer ); (see Section ?? )
    function rateOfHarmonyGain ( var t :technique ;var derivativeOfProductHarmony :vector ) :real ; (see Sect
    function marginalphysicalcoproducts ( var t :technique ; input :presource ) :pvec ; (see Section 26 )
    function getCoproductionCodes ( var t :technique ):pintvec ; (see Section 24 )
    implementation
    procedure logComplex ( var ct :tc ); (see Section 21 )

    function techniques ( var ct :tc ):ptvec ; (see Section 22 )
    function produces ( var ct :tc ;t :technique ;productNumber :integer ):boolean ; (see Section 23 )
    function getCoproductionCodes ( var t :technique ):pintvec ; (see Section 24 )
    function findIlna ( i :presource ;var a : resourcevec ) :integer ; (see Section 25 )
    function marginalphysicalcoproducts ( var t :technique ; input :presource ) :pvec ; (see Section 26 )
    function rateOfHarmonyGain ( var t :technique ;var derivativeOfProductHarmony :vector ) :real ; (see Sect
    end ;
function hash ( s :string ):integer ; (see Section 28 )
```

# 21 logComplex

```
procedure logComplex ( var ct :tc );
var
    Let f ∈ text;
    Let i, j ∈ integer;
    ui :pdvec
procedure printtechnique ( var t :technique ); (see Section 29 )
procedure rect ( te :ptechniquelist ); (see Section 30 )
begin
with ct do begin
        assign (f, 'complex.csv' );
        rewrite (f);
        writeln(f, 'Technology Complex' );
        writeln ( f , 'index.max,nonproduced.max,nonfinal.max,allresourceindex.max,techniquecount,
        productcount' );
        writeln ( f ,index ^ .max , ',' ,nonproduced ^ .max , ',' ,nonfinal ^ .max , ',' ,allresourceindex
        ^ .max , ',' ,techniquecount , ',' ,productcount );
        write(f, 'Resource number' );
        for i← 1 to allresourceindex↑.max do write ( f , ',' ,i );
        writeln(f);
```

```
        write(f, 'Resource id' );
        for i← 1 to allresourceindex↑.max do
            if allresourceindex↑[i] = nil then write ( f , ', ' ) else write ( f , ',' ,allresourceindex
            ^ [i ]^ .id );
        writeln(f);
        { now list all techniques }
        rect (techniqueslist);
        { now the user index }
        writeln(f, 'User index' );
        ui← buildUserIndex (ct);
        for i← 1 to ui↑.maxv do
        begin
            write ( f , 'Product,' ,i , ',is used by technique' );
            for j← 1 to ui↑[i]↑.maxt do
                write ( f , ',' ,ui ^ [i ]^ [j ]^ .techniqueno );
            writeln(f);
        end ;
        writeln(f, 'Producer index' );
        ui← buildproducerIndex (ct);
        for i← 1 to ui↑.maxv do
        begin
            write ( f , 'Product,' ,i );
            if nonproduced↑[i] then write ( f , ', is an initial input and produced by' ) else write
            ( f , ',is produced by technique' );
            for j← 1 to ui↑[i]↑.maxt do
                write ( f , ',' ,ui ^ [i ]^ [j ]^ .techniqueno );
            writeln(f);
        end ;
        close (f);
    end ;
end ;
```

# 22 techniques

```
function techniques ( var ct :tc ):ptvec ;
var
    Let i ∈ integer;
    Let list ∈ ptechniquelist;
begin
with ct do begin begin
        if techniquesvec = nil then
        begin
            new ( techniquesvec ,techniquecount );
            list← techniqueslist;
            for i← 1 to techniquecount do
            begin
                techniquesvec↑[i]← list↑.tech;
                list← list↑.next;
```

```
            end ;
        end ;
        techniques← techniquesvec;
    end ;
end ;
```

## 23 produces

```
function produces ( var ct :tc ;t :technique ;productNumber :integer ):boolean ;
var
    Let i ∈ integer;
    Let ok ∈ boolean;
begin
    with t do with ct do
    begin
        ok← false;
        for i← 1 to produces↑.max do
            if produces↑[i].product↑.productnumber = productNumber then ok← true;
    end ;
    produces← ok;
end ;
```

## 24 getCoproductionCodes

```
function getCoproductionCodes ( var t :technique ):pintvec ;
var
    Let p ∈ pintvec;
begin
    with t do
    begin
        new ( p ,produces ^ .max );
        p↑← produces↑[iota0].product↑.productNumber;
        getCoproductionCodes← p;
    end ;
end ;
```

## 25 findiIna

```
function findiIna ( i :presource ;var a : resourcevec ) :integer ;
label 99;
var
    Let j ∈ integer;
begin
    for j← 1 to a.max do
        if (aj.product↑.id = i↑.id) then
```

```
        begin
            findiIna← j;
            goto 99;
        end ;
        findiIna← - 1 ;
        99:
end ;
```

# 26 marginalphysicalcoproducts

```
function marginalphysicalcoproducts ( var t :technique ; input :presource ) :pvec ;
var
    Let mpp ∈ pvec;
    Let pos, i ∈ integer;
begin
    new ( mpp , t .produces ^ .max );

    pos← findiIna (input, t.consumes↑);
    if pos < 1 then
    begin
        writeln( 'findiIna returns ' , pos);
        if input = nil then write( 'input was nil' ) else writeln( 'input non nil' );
        writeln( ' in technique ' , t.techniqueno);
        writeln( 'could not find ' , input↑.productnumber, input↑.id);
        writeln( 'the technique actually consumes the following' );
        for i← 1 to t.consumes↑.max do
            write ( t .consumes ^ [i ].product ^ .id , ', ' );
        halt (405);
    end ;
    with t do
        for i← 1 to mpp↑.cols do
            mpp↑[i]← produces↑[i].quantity/consumes↑[pos].quantity;

    marginalphysicalcoproducts← mpp;
end ;
```

# 27 rateOfHarmonyGain

```
function rateOfHarmonyGain ( var t :technique ;var derivativeOfProductHarmony :vector )
:real ;
var
    Let gain, cost ∈ real;
    Let j ∈ integer;
begin with t do begin
        gain← 0;
        for j← 1 to produces↑.max do
```

```
        gain← gain + derivativeOfProductHarmony produces↑[j].product↑.productNumber × pro-
        duces↑[j].quantity;
    cost← 0;
    for j← 1 to consumes↑.max do
        cost← cost + derivativeOfProductHarmony consumes↑[j].product↑.productNumber × con-
        sumes↑[j].quantity;

    writeln ( techniqueno , ',' ,produces ^ [1].product ^ . productNumber , ', ' ,gain , ',' ,cost
    );
    rateofharmonygain← gain−cost/cost;
end ;
```

# 28 hash

```
function hash ( s :string ):integer ;
var
    Let i, j ∈ integer;
begin
    Let j ∈ =1;
    for i := 1 to length ( s ) do
        Let j ∈ = (j*11 +ord(s[i]) ) and maxint;
    Let hash ∈ =j;
end ;

function defineResource ( var ct :tc ;name :resourceid ;number :integer ):presource ; (see Section 31 )

    defineResource← t;
end ;
function defineproductlist ( var ct :tc ;name :string ; p :pproductlist ;number :integer ):pproductlist ; (see Sec
```

# 29 printtechnique

```
procedure printtechnique ( var t :technique );
var
    Let i ∈ integer;
begin
    with t do
    begin
        writeln ( f , 'technique,' ,techniqueno );
        write(f, 'inputs' );
        for i← 1 to consumes↑.max do
            write ( f , ',' ,consumes ^ [i ].product ^ .productnumber );
        writeln(f);
        write(f, 'outputs' );
        for i← 1 to produces↑.max do
            write ( f , ',' ,produces ^ [i ].product ^ .productnumber );
        writeln(f);
```

```
    end ;
end ;
```

## 30 rect

```
procedure rect ( te :ptechniquelist );
begin
    else
    begin
        rect (te↑.next);
        printtechnique (te↑.tech↑);
    end ;
end ;
```

## 31 defineResource

```
function defineResource ( var ct :tc ;name :resourceid ;number :integer ):presource ;
var
    Let t ∈ presource;
begin
    new ( t );
    with t ^ do with ct do
    begin
        id← name;
        productcount← number;
        productNumber← productcount;
        users← nil;
        producers← nil;

        allresourceindex↑[productnumber]← t;

    end ;
```

## 32 defineproductlist

```
function defineproductlist ( var ct :tc ;name :string ; p :pproductlist ;number :integer ):ppro-
ductlist ;
var
    Let pntr ∈ pproductlist;
begin
    new ( pntr );
    Let pntr↑.next ∈ =p;
    pntr ^ .product :=defineResource ( ct ,name ,number );
    Let defineproductlist ∈ =pntr;
end ;
```

**function** *findproduct* ( **var** *ct* :*tc* ; **name** :**string** ):*presource* ; (see Section 33 )
**end** ;

**procedure** *addproduct* ( **var** *ct* :*tc* ; **name** :**string** ;*number* :*integer* ); (see Section 34 )
**end** ;

**function** *buildIndex* ( **var** *ct* :*tc* ;*produces* :*boolean* ):*pdvec* ; (see Section 35 )
**end** ;
**function** *buildProducerIndex* ( **var** *ct* :*tc* ):*pdvec* ; (see Section 36 )

# 33 findproduct

```
function findproduct ( var ct :tc ; name :string ):presource ;
var
    Let h ∈ integer;
    Let p ∈ pproductlist;
    Let ok ∈ boolean;
begin
    Let h ∈ =hash(name) ;
    with  ct  do
    begin
        h← h rem index ↑.max;

        p← index↑[h];
        ok← p ≠ nil;
        while  ok  do
        begin

            ok← not(p↑.product↑.id = name);
            if  ok  then
            begin
                p← p↑.next;
                ok← p ≠ nil;
            end ;
        end ;
        if  p = nil  then
        begin
            writeln( 'product '  , name,  ' not found'  );
            exit ( 401);
        end ;

        if  p = nil  then  findproduct← nil
        else  findproduct← p↑.product;
    end ;
```

## 34 addproduct

```
procedure addproduct ( var ct :tc ; name :string ;number :integer );
var
    Let h ∈ integer;
    Let p ∈ pproductlist;
    Let ok ∈ boolean;
begin
    Let h ∈ =hash(name) ;
    with   ct   do
    begin
        h← h rem index ↑.max;

        p← index↑[h];
        ok← p ≠ nil;
        while  ok  do
        begin
            ok← p↑.product↑.id ≠ name;
            if  ok  then
            begin
                p← p↑.next;
                ok← p ≠ nil;
            end ;
        end ;
        if  p = nil  then  index↑[h]← defineproductlist (ct, name, index↑[h], number);
        { if p<>nil then product already defined }
    end ;
```

## 35 buildIndex

```
function buildIndex ( var ct :tc ;produces :boolean ):pdvec ;
var
    Let locindex ∈ pdvec;
    Let l, j, k, l ∈ integer;
    Let p ∈ ptechniquelist;
    Let t ∈ technique;
    Let producercount ∈ pintvec;
begin
    with   ct   do
    begin

        if  ( ( producerIndex =nil )  and   produces )or ( ( userIndex =nil )  and   not produces )
        then
        begin
            // if produces then writeln ( 'buildproducerindex' ) else writeln ( 'builduserindex' );
            new ( locindex ,productcount );
            new ( producercount , productcount );
            p← techniqueslist;
            producercount↑← 0;
```

```
            while p ≠ nil do
            begin
                t← p↑.tech↑;
                if produces then l← t.produces↑.max else l← t.consumes↑.max;
                // writeln ( 'technique ' , t .techniqueno , 'length of ' ,( if produces then 'produces' else
                'consumes' ), ' list =' ,l );
                for i← 1 to l do
                begin
                    if produces then k← t.produces↑[i].product↑.productnumber
                    else k← t.consumes↑[i].product↑.productnumber;
                    producercount↑[k]← producercount↑[k] + 1;
                end ;
                p← p↑.next;
            end ;
            // writeln ( 'producercounts' );
            // writeln ( producercount ^ );
            p← techniqueslist;
create a vector of producers
            for i← 1 to productcount do new(locindex↑[i], producercount↑[i]);
            while p ≠ nil do
            begin
                t← p↑.tech↑;
                if produces then l← t.produces↑.max else l← t.consumes↑.max;
                for i← 1 to l do
                begin
                    if produces then j← t.produces↑[i].product↑.productnumber
                    else j← t.consumes↑[i].product↑.productnumber;
                    k← producercount↑[j];

                    locindex↑[j]↑[k]← p↑.tech;
                    producercount↑[j]← k - 1;
                end ;
                p← p↑.next;
            end ;
            if produces then
                producerIndex← locindex
            else userindex← locindex
        end ;
        if produces then
            buildIndex← producerIndex
        else buildIndex← userindex;
    end ;
```

# 36 buildProducerIndex

```
function buildProducerIndex ( var ct :tc ):pdvec ;
begin buildproducerindex := buildIndex ( ct ,true );
end ;
function buildUserIndex ( var ct :tc ):pdvec ; (see Section 37 )
```

## 37 buildUserIndex

```
function buildUserIndex ( var ct :tc ):pdvec ;
begin builduserindex := buildindex ( ct ,false );
end ;
procedure defineComplex ( var ct :tc ;numberofproducts :integer ); (see Section 38 )

end ;

function defineTechnique ( var ct :tc ;var inputs ,outputs :resourcevec ):ptechnique ; (see Section 39 )
end ;
begin

end .
```

## 38 defineComplex

```
procedure defineComplex ( var ct :tc ;numberofproducts :integer );
var
   Let complex ∈ technologycomplex;
begin
   // writeln ( 'definecomplex ' ,numberofproducts );
   with ct do
   begin
      new ( index ,( numberofproducts div 2)+1);

      new ( nonproduced , numberofproducts );
      nonproduced↑← false;
      new ( nonfinal ,numberofproducts );
      nonfinal↑← false;
      techniquesvec← nil;
      techniqueslist← nil;
      techniquecount← 0;
      productcount← 0;

      new ( allresourceindex ,numberofproducts );
      allresourceindex↑← nil;
      index↑← nil
   end ;
```

# 39 defineTechnique

```
function defineTechnique ( var ct :tc ;var inputs ,outputs :resourcevec ):ptechnique ;
var
    Let t ∈ ptechnique;
    Let i ∈ integer;
var  Let tl ∈ ptechniquelist;
procedure adduser ( product :presource ); (see Section 40 )
    product↑.users← l;
end ;
procedure addproducer ( product :presource ); (see Section 41 )
product↑.producers← l;
end ;
begin
with  ct  do
begin
    new ( t );
    techniqueCount← techniqueCount + 1;
    //writeln ( 'def tech ' ,techniquecount , ' with ' , inputs .max , ' inputs and ' ,outputs
    .max , ' outputs' );

    with  t↑  do
    begin
        techniqueno← techniquecount;
        new ( produces ,outputs .max );

        new ( consumes ,inputs .max );
        produces↑← outputs;
        consumes↑← inputs;
        for  i← 1  to  outputs.max  do
            if  outputs_i.product = nil  then
                begin writeln ( 'null product in outputs' ,i ) ;
            halt (300);
        end
        else
            addproducer (outputs_i.product);
        for  i← 1  to  inputs.max  do
            if  inputs_i.product = nil  then
                begin writeln ( 'null product in inputs' ,i ) ;
            halt (300);
    end else
        adduser (inputs_i.product);
    end ;

    new ( tl );
    tl↑.tech← t;
    tl↑.next← techniqueslist;
    techniqueslist← tl;
    definetechnique← t;
end ;
```

# 40 adduser

```
procedure adduser ( product :presource );
var
   Let l ∈ ptechniquelist;
begin
   // writeln ( 'adduser ' ,product ^ .productnumber , product ^ .id );
   new ( l );
   with  l ^  do
   begin
      tech← t;
      next← product↑.users;
   end ;
```

# 41 addproducer

```
procedure addproducer ( product :presource );
var
   Let l ∈ ptechniquelist;
begin
   // writeln ( 'addproducer ' ,product ^ .productnumber , product ^ .id );
   new ( l );
   with  l ^  do
   begin
      tech← t;
      next← product↑.producers;
   end ;
```

# 42 harmony

```
unit harmony ;
```

A class to optimise a set of linear production technologies to meet a Kantorovich style output target and having a pregiven set of initial resources.¡p¿

It produces an output file of the plan in lp-solve format on standard out¡p¿ Class to provide optimisation of plans using the algorithm in Towards a New Socialism ¡p¿ Copyright (C) 2018 William Paul Cockshott

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see https://www.gnu.org/licenses/.

```
interface
    uses technologies ;
    function H ( target , netoutput :real ) :real ; (see Section 45 )

    function dH ( target , netoutput :real ) :real ; (see Section ?? )
    const
        useweight :real = 5;
        phase2adjust :real = 0.3 ;
        capacitytarget :real =0.98;
        startingtemp :real =0.23;
        meanh :real =0;
        phase1rescale :boolean =true ;
        phase2rescale :boolean =true ;
        iters :integer =80;
        verbose :boolean =true ;

    function balancePlan ( var planTargets , initialresource :vector ;var C :technologycomplex ):pvec ; (see Sec

    procedure printstateS ( var netOutput ,productHarmonyDerivatives ,productHarmony :vector ; (see Section
    var
        Let C ∈ TechnologyComplex ;
    var
        Let intensity ∈ vector);
    function mean ( var m :vector ; var C :TechnologyComplex ):real ; (see Section 51 )
    function nonfinalHarmonyDerivativeMax ( var netOutput :vector ; nonfinal :integer ;var dharmonies :vecto
    function computeGrossAvail ( var C :TechnologyComplex ;var intensity , initial :vector ):pvec ; (see Sectio
```

**function** *computeNetOutput* ( **var** *C* :*TechnologyComplex* ; **var** *intensity* ,*initial* :*vector* ):*pvec* ; (see Sectio
**procedure** *rescaleIntensity* ( **var** *intense* :*vector* ;**var** *C* :*TechnologyComplex* ; **var** *initialresource* :*vector* ); (
**function** *sigmoid* ( *d* :*real* ):*real* ; (see Section 58 )
**procedure** *initialiseIntensities* ( **var** *intensity* :*vector* ;*C* :*TechnologyComplex* ;**var** *initialresource* :*vector* ); (
**procedure** *equaliseHarmony* ( **var** *intensity* , (see Section 56 )
*derivativeOfProductHarmony* ,
Let *netproduct* ∈ vector;
Let *temperature* ∈ real;
**var**
Let *C* ∈ TechnologyComplex ;
**var**
Let *h* ∈ vector;
**var**
Let *index* ∈ pdvec;
**var** Let *initialresource* ∈ vector );

**var**
Let *productHarmony* ∈ pvec;

**implementation**
**procedure** *rescaleIntensity* ( **var** *intense* :*vector* ;**var** *C* :*TechnologyComplex* ; **var** *initialresource* :*vector* ); (

the derivative of the harmony function * evaluated numerically so as to be independent of the H function

**function** *fdH* ( *target* , *netoutput* :*real* ) :*real* ; (see Section 44 )

**function** *H* ( *target* , *netoutput* :*real* ) :*real* ; (see Section 45 )

Forward procedure declarations

**function** *meanv* ( **var** *m* :*vector* ):*real* ; (see Section 53 )
*forward*;
**function** *computeHarmonyDerivatives* ( **var** *netOutput* , *planTargets* :*vector* ;*C* :*TechnologyComplex* ;**var** *in*
*forward*;

gives the vector of total amount produced or available in initial resource vector
- does not deduct productive consumption

**function** *computeGrossAvail* ( **var** *C* :*TechnologyComplex* ;**var** *intensity* , *initial* :*vector* ):*pvec* ; (see Sectio

C is a technology complex, fixed resources should be added as nonproduced products¡p¿ planTargets is the target output of each product¡p¿ returns a vector of technology intensities

**function** *balancePlan* ( **var** *planTargets* , *initialresource* :*vector* ;**var** *C* :*technologycomplex* ):*pvec* ; (see Sec

compute the derivatives of the harmonies of all products with repect to marginal increase in output in terms of actual output units not intensities

**function** *computeHarmonyDerivatives* ( **var** *netOutput* , *planTargets* :*vector* ;*C* :*TechnologyComplex* ;**var** *i*

**procedure** *printstateS* ( **var** *netOutput* ,*productHarmonyDerivatives* ,*productHarmony* :*vector* ;**var** *C* :*Tech*

for non final goods we make derivatives their harmonies the maximum of the derivatives of the harmonies of their users

**function** *nonfinalHarmonyDerivativeMax* ( **var** *netOutput* :*vector* ; *nonfinal* :*integer* ;**var** *dharmonies* :*vecto*

**function** *mean* ( **var** *m* :*vector* ;**var** *C* :*TechnologyComplex* ):*real* ; (see Section 51 )
**function** *sdev* ( **var** *m* :*vector* ;**var** *C* :*TechnologyComplex* ):*real* ; (see Section 52 )
**function** *meanv* ( **var** *m* :*vector* ):*real* ; (see Section 53 )
**function** *stdev* ( **var** *m* :*vector* ; *av* :*real* ) :*real* ; (see Section 54 )

shrink or expand all industries in order to not exceed target level of use of the critical fixed reource

**procedure** *initialiseIntensities* ( **var** *intensity* :*vector* ;*C* :*TechnologyComplex* ;**var** *initialresource* :*vector* ); (
**procedure** *equaliseHarmony* ( **var** *intensity* , (see Section 56 )

**function** *computeNetOutput* ( **var** *C* :*TechnologyComplex* ;**var** *intensity* ,*initial* :*vector* ):*pvec* ; (see Sectio
**function** *sigmoid* ( *d* :*real* ):*real* ; (see Section 58 )

**begin**
**end** .

# 43 rescaleIntensity

**procedure** *rescaleIntensity* ( **var** *intense* :*vector* ;**var** *C* :*TechnologyComplex* ; **var** *initialresource* :*vector* );

```
var
    Let netoutput ∈ pvec;
    Let amountused, shortfallratio, maxfrac, resource, usage, fractionaluse, expansionratio, weight
    ∈ real;
    Let i, j ∈ integer;
    Let grossAvail, shrinkby ∈ pvec;
    Let users, pt ∈ ptvec;
    Let t ∈ ptechnique;
    Let allpositive ∈ boolean;
    Let ui ∈ pdvec;
begin
    netoutput← computeNetOutput (C, intense, initialresource);
    if (verbose) then
    begin
        writeln( 'post phase0' );
        writeln( 'netoutput' );
        writeln(netoutput↑);

    end ;
    maxfrac← 0;
    for i← 1 to C.nonproduced↑.max do
        if (C.nonproduced↑[i]) then
        begin
            resource← initialresource_i;
            usage← resource - netoutput↑[i];
            fractionaluse← usage/resource;
            if (fractionaluse > maxfrac) then maxfrac← fractionaluse;
        end ;
        expansionratio← capacitytarget/maxfrac;
        if (phase1rescale) then
            // expand overall scale of production to balance
            intense← expansionratio × intense;
    // writeln ( 'dispose(netoutput)' );
    dispose ( netoutput );
    // now make sure no other resource has a negative output
    netoutput← computeNetOutput (C, intense, initialresource);
    if (verbose) then
    begin
        writeln( 'post phase1' );
        writeln( 'netoutput' );
        writeln(netoutput↑);
        writeln( 'intensity' );
        writeln(intense);
    end ;
    allpositive← \∧ ((netoutput↑) ≥ 0) ;

    if (not allpositive ) then
        if (phase2rescale) then
        begin
```

```
ui← buildUserIndex (C);
grossAvail← computeGrossAvail (C, intense, initialresource);
new ( shrinkby , C .techniqueCount );
shrinkby↑← 1;
for i← 1 to netoutput↑.cols do
if (netoutput↑[i] < 0) then begin
        amountused← grossAvail↑[i] - netoutput↑[i];
        shortfallratio← capacitytarget×(grossAvail↑[i]) / amountused;
        users← ui↑[i];
```

Users is now a vector of all techqniques that use product i

```
        weight← 0;
        pt← techniques (C);
```

go through all techniques which use product i

```
        for j← 1 to users↑.maxt do
        begin
            t← users↑[j];
            if verbose then writeln( ' for product ' , i, 'user ' , j, 'is technique number
            t↑.techniqueno);
```

check that they do not actually make product i as output

```
            if not produces (C, t↑, i) then
            begin
```

reduce its intensity by the shortfall ratio

```
                if (shortfallratio < shrinkby↑[T↑.Techniqueno]) then
                    shrinkby↑[T↑.Techniqueno]← shortfallratio;
            end ;
        end ;

    end ;
    intense← intense × shrinkby↑;
    if (verbose) then
    begin
        writeln( 'postphase2' );
        writeln( 'netoutput' );
        writeln(netoutput↑);
        writeln( 'shrinkby' );
```

```
                writeln(shrinkby↑);
                writeln( 'intensity'  );
                writeln(intense);
            end ;
            dispose ( grossavail );
            dispose ( shrinkby );
        end ;


        dispose ( netoutput );


    end ;
```

# 44 fdH

```
function fdH ( target , netoutput :real ) :real ;
var
    Let ϵ, base, baseplusEpsilon ∈ real;
begin

    ϵ← 0.0004;
    base← H (target, netoutput);
    basePlusEpsilon← H (target, ϵ + netoutput);
    fdh← (basePlusEpsilon−base)/ϵ;

end ;
```

# 45 H

```
function H ( target , netoutput :real ) :real ;
var
    scale :real
begin
    scale← (netoutput−target)/target;
    if  (scale < 0)  then  H← scale - (scale × scale) × 0.5
    else  H← ln(scale + 1);
end ;
```

# 46 computeGrossAvail

```
function computeGrossAvail ( var C :TechnologyComplex ;var intensity , initial :vector ):pvec
;
var
    Let outputv ∈ pvec;
```

```
    Let j, i, p ∈ integer;
    Let ltrav ∈ ptvec;
    Let t ∈ ptechnique;
    Let f ∈ real;
begin
    new ( outputv ,C .productCount );
    outputv↑← initial;
    ltrav← techniques (C);
    for j← 1 to C.techniqueCount do

    begin
        t← ltrav↑[j];

        for i← 1 to t↑.produces↑.max do
        begin
            p← t↑.produces↑[i].product↑.productNumber;
            f← t↑.produces↑[i].quantity;
            outputv↑[p]← outputv↑[p] + f × intensity t↑.techniqueno;

            if verbose then
            begin
                {writeln(t↑.techniqueno,p,f,intensity[t↑.techniqueno]);}
            end ;
        end ;

    end ;
    computeGrossAvail← outputv;
end ;
```

# 47 balancePlan

```
function balancePlan ( var planTargets , initialresource :vector ;var C :technologycomplex
):pvec ;
label 99;


var
    Let producerindex ∈ pdvec;
    Let intensity, netoutput, productHarmonyDerivatives ∈ pvec;
    Let t, meanh ∈ real;
    Let i ∈ integer;
function computeHarmony ( var netOutput , planTargets :vector ;C :TechnologyComplex ;var intentsity :vecto
procedure printstate ( var intensity :vector ;C :TechnologyComplex ;var initial , targets :vector ) ; (see Sectio
procedure adjustIntensities ( var intensity :vector ; (see Section 61 )

begin
if verbose then begin begin
        writeln( 'balancePlan' );
```

```
        writeln(planTargets, initialresource);
    end ;
    if (planTargets.cols ≠ C.productCount) then
    begin
        writeln ( 'plan target has length ' ,planTargets .cols ,
        ' but the number of products in TechnologyComplex is ' , C .productCount );
        balancePlan← nil;
        goto 99;
    end ;
    producerIndex← buildProducerIndex (C);
    new ( intensity , C .techniqueCount );
    initialiseIntensities (intensity↑, C, initialresource);
    if (verbose) then begin write ( 'initialised intensity' );
        writeln(intensity↑);
    end ;
    t← startingtemp;
    new ( productHarmony , C .productCount );
for i← 0 to iters - 1 do begin
        netOutput← computeNetOutput (C, intensity↑, initialresource);
        productHarmony↑← H (planTargets, netOutput↑);
        meanh← mean (productHarmony↑, C);
        productHarmonyDerivatives← computeHarmonyDerivatives (netOutput↑, planTargets, C, in-
        tensity↑);
        adjustIntensities ( intensity ^ ,
        productHarmonyDerivatives
        t
        C
        productHarmony ^ ,
        producerIndex
        initialresource ,planTargets );
        if (verbose) then printstate (intensity↑, C, initialresource, planTargets);
        dispose ( netOutput );
        dispose ( productHarmonyDerivatives );
    end ;
    balancePlan← intensity;
    99:
end ;
```

# 48 computeHarmonyDerivatives

```
function computeHarmonyDerivatives ( var netOutput , planTargets :vector ;C :Technology-
Complex ;var intentsity :vector ):pvec ;
var
    Let dh ∈ pvec;
    Let i, solve ∈ integer;
begin
    new ( dh , netOutput .cols );
    dh↑← fdH (planTargets, netOutput);
```

```
    for solve← 1 to 2 do
        {$par}
        for i← 1 to C.nonfinal↑.max do
            if (C.nonfinal↑[i]) then
            begin
                dh↑[i]← (dh↑[i]+useweight×nonfinalHarmonyDerivativeMax(netOutput,i,dh↑,C)) / useweight+1;
            end ;
            computeHarmonyDerivatives← dh;
end ;
```

weighted average of derivative due to shortage and due to potential other use

# 49 printstateS

```
procedure printstateS ( var netOutput ,productHarmonyDerivatives ,productHarmony :vector
;var C :TechnologyComplex ;var intensity :vector );
var
    Let expansionrate, gainrate ∈ pvec;
    Let i, pn ∈ integer;
    Let t ∈ ptvec;
begin
    writeln( 'netoutput ' );
    write(netOutput);
    writeln( 'intensity' );
    writeln(intensity);
    writeln ( 'h ,' );
    writeln(productHarmony);

    writeln( 'productHarmonyDerivatives' );
    writeln(productHarmonyDerivatives);
    new ( expansionrate ,C .techniquecount );
    new ( gainrate ,C .techniquecount );
    t← techniques (C);
    for i← 1 to C.techniquecount do
    begin
        pn← t↑[i]↑.techniqueno;
        gainrate↑[pn]← rateOfHarmonyGain (t↑[i]↑, productHarmonyDerivatives);
        expansionrate↑[pn]← 1 + sigmoid (gainrate↑[pn]) × startingtemp × phase2adjust;
    end ;
    write ( 'gainrates, ' );
    writeln(gainrate↑);
    write ( 'expansionrates,' );
    writeln(expansionrate↑);
    dispose ( expansionrate );
    dispose ( gainrate );
end ;
```

# 50 nonfinalHarmonyDerivativeMax

```
function nonfinalHarmonyDerivativeMax ( var netOutput :vector ; nonfinal :integer ;var dharmonies :vector ;var C :TechnologyComplex ):real ;
var
    max ,total ,d :real ;
    Let best ∈ integer;
    Let userIndex ∈ pdvec;
    Let users ∈ ptvec;
    Let i, techno ∈ integer;
    Let t ∈ ptechnique;
    Let mpp ∈ pvec;
    Let codes ∈ pintvec;
    Let pt ∈ ptvec;
begin
    userIndex← buildUserIndex (C);
    max := -1e22;
    total← 0;
    d← 0;
    best← 0;
    users← userIndex↑[nonfinal];
    if users = nil then
    begin
        write( 'userIndex[' );
        halt (405);
    end ;
    pt← techniques (C);
    for i← 1 to users↑.maxt do
    begin
        t← users↑[i];
        mpp← marginalphysicalcoproducts (t↑, C.allresourceindex↑[nonfinal]);
        codes← getCoproductionCodes (t↑);
        d← ∑ dharmonies_{codes↑} × mpp↑ ;
        dispose ( mpp );
        dispose ( codes );
        total← total + d;
        if ((d) > max) then
            max :=d ;
    end ;
    nonfinalHarmonyDerivativeMax← total/users↑.maxt;

end ;
```

# 51 mean

```
function mean ( var m :vector ;var C :TechnologyComplex ):real ;
var
    Let sum ∈ real;
```

```
    Let num, i ∈ integer;
begin
    sum← 0;
    num← 0;
    for  i← 1  to  C.nonproduced↑.max  do
        if  (not C.nonproduced ↑[i])  then
        begin
            sum← sum + m_i;
            num← num + 1;
        end ;
        mean← sum/num;
end ;
```

## 52 sdev

```
function sdev ( var m :vector ;var C :TechnologyComplex ):real ;
var
    Let sum, av ∈ real;
    Let num, i ∈ integer;
begin
    sum← 0;
    av← mean (m, C);
    num← 0;
    for  i← 1  to  C.nonproduced↑.max  do
        if  (not C.nonproduced ↑[i])  then
        begin
            sum← sum + (m_i - av) × (m_i - av);
            num← num + 1;
        end ;
        sdev← √(sum/num);
end ;
```

## 53 meanv

```
function meanv ( var m :vector ):real ;
var
    Let sum ∈ real;
    Let num, i ∈ integer;
begin
    sum← 0;
    num← 0;
    for  i← 1  to  m.cols  do

    begin
        sum← sum + m_i;
        num← num + 1;
```

```
    end ;
    meanv← sum/num;
end ;
```

## 54 stdev

```
function stdev ( var m :vector ; av :real ) :real ;
var
    Let sum, av ∈ real;
    Let num, i ∈ integer;
begin
    sum← 0;
    av← meanv (m);
    num← 0;
    for i← 1 to m.cols do
    begin
        sum← sum + (m_i - av) × (m_i - av);
        num← num + 1;
    end ;
    stdev← √(sum/num);
end ;
```

## 55 initialiseIntensities

```
procedure initialiseIntensities ( var intensity :vector ;C :TechnologyComplex ;var initialresource
:vector );
var
    Let i ∈ integer;
begin
    intensity← 0.1;
    rescaleIntensity (intensity, C, initialresource);
end ;
```

## 56 equaliseHarmony

```
procedure equaliseHarmony ( var intensity ,
derivativeOfProductHarmony
netproduct : vector;
temperature : real;
var
    Let C ∈ TechnologyComplex ;
var
    Let h ∈ vector;
var
    Let index ∈ pdvec;
```

```
var
    Let initialresource ∈ vector );
var
    Let mh, divisor ∈ real;
    Let excessh, changeoutput, fractionalchange ∈ real;
    Let k, j, i ∈ integer;
    Let productionset ∈ ptvec;
begin

    mh← mean (h, C);
    for k← 1 to h.cols do
        if (not C.nonproduced ↑[k]) then
        if (not C.nonfinal ↑[k]) then begin begin
```

work out how much to change its output to get it on the mean

```
                excessH← (h_k - mh);
```

divide this by the derivative to get change in output

```
                changeOutput← temperature × excessH;
                if netproduct_k = 0.0 then divisor← 1.0 else divisor← netproduct_k;
                if derivativeofproductharmony_k ≠ 0 then
            else begin begin
                    writeln ( 'error, zero harmony derivative for product ' ,k );
                    halt (406);
                end ;
                productionSet← index↑[k];
                if productionset = nil then
                begin
                    writeln( 'corrupt index in equalise harmony' );
                    halt (402);
                end
                else
                for i← 1 to productionset↑.maxt do begin begin
                    if (productionset↑[i] = nil) then begin begin
                            writeln( 'productionset[' );
                            halt (404);
                        end ;
                        j← productionset↑[i]↑.techniqueno;
                        (* sign is negative since we reduce the high harmonies*)
                        intensity_j← intensity_j × (1 - fractionalchange);
                        if (intensity_j < 0) then
                        begin
                            writeln ( 'IllegalIntensity ' ,j , ' went negative, fractional change = ' ,fractionalcha
signal the pascal arithmetic
overflow error
                            halt (215);
                        end ;
```

```
            end ;
        end ;
    end ;
```

## 57 computeNetOutput

```
function computeNetOutput ( var C :TechnologyComplex ;var intensity ,initial :vector ):pvec
;
var
    Let outputv ∈ pvec;
    Let k, k2, i, j ∈ integer;
    Let t ∈ ptechnique;
    Let pt ∈ ptvec;
    Let it ∈ real;
begin
    writeln( 'in compute net output' );
    outputv← computeGrossAvail (C, intensity, initial);
    pt← techniques (C);
if (verbose) then begin begin
        writeln( 'output' );
        writeln(outputv↑);
    end ;
    for j← 1 to C.techniqueCount do
    begin
        t← pt↑[j];
        it← intensity t↑.techniqueno;

        for k← 1 to t↑.consumes↑.max do
            outputv↑[t↑.consumes↑[k].product↑.productnumber]← outputv↑[t↑.consumes↑[k].product↑.productnum
            -it *t ^ .consumes ^ [k ].quantity ;

    end ;
    writeln( 'leavecomputenetoutput' );
    computeNetOutput← outputv;
end ;
```

## 58 sigmoid

```
function sigmoid ( d :real ):real ;
begin
    if (d > 0) then sigmoid← d/(1+d) else
        if (d = 0) then sigmoid← 0 else begin begin
                d← - d ;
                sigmoid← - (d/(1+d)) ;
            end
end ;
```

## 59 computeHarmony

**function** *computeHarmony* ( **var** *netOutput* , *planTargets* :*vector* ;*C* :*TechnologyComplex* ;**var** *intentsity* :*vector* ):*pvec* ;
**var**
Let *lh* ∈ pvec;
Let *i* ∈ integer;
**begin**
**new** ( *lh* ,*netOutput .cols* );

*lh*↑← *H* (*planTargets*, *netOutput*);
*computeHarmony*← *lh*;
**end** ;

## 60 printstate

**procedure** *printstate* ( **var** *intensity* :*vector* ;*C* :*TechnologyComplex* ;**var** *initial* , *targets* :*vector* ) ;
**var**
Let *netoutput*, *h*, *hd* ∈ pvec;
**begin**
*netOutput*← *computeNetOutput* (*C*, *intensity*, *initial*);
*h*← *computeHarmony* (*netOutput*↑, *targets*, *C*, *intensity*);
*hd*← *computeHarmonyDerivatives* (*netOutput*↑, *targets*, *C*, *intensity*);
*printstateS* (*netOutput*↑, *hd*↑, *h*↑, *C*, *intensity*);
// **writeln** ( 'dispose in printstate' );
**dispose** ( *h* );
**dispose** ( *hd* );
**dispose** ( *netOutput* );
**end** ;

## 61 adjustIntensities

**procedure** *adjustIntensities* ( **var** *intensity* :*vector* ;
**var**
Let *derivativeOfProductHarmony* ∈ pvec;
Let *temperature* ∈ real;
**var**
Let *C* ∈ technologycomplex;
**var**
Let *h* ∈ vector;
**var**
Let *index* ∈ pdvec;
**var**

```
    initialresource ,
    Let planTargets ∈ vector);
var
    Let netOutput ∈ pvec;
    Let expansionrate ∈ pvec;
    Let ltechniques ∈ ptvec;
    Let t ∈ ptechnique;
    Let meane, adjustedexp ∈ real;
    Let i, j ∈ integer;
begin
    netOutput← computeNetOutput (C, intensity, initialresource);
if (verbose) then begin begin
        writeln( 'preequalisation' );
        printstate (intensity, C, initialresource, planTargets);
    end ;
    equaliseHarmony ( intensity ,
    derivativeOfProductHarmony ^ ,
    netOutput ^ ,
    temperature
    C
    h
    index
    initialresource );

    // dispose ( netOutput );
    netOutput← computeNetOutput (C, intensity, initialresource);
    derivativeOfProductHarmony← computeHarmonyDerivatives (netOutput↑, planTargets, C, in-
    tensity);
if (verbose) then begin begin
        writeln( 'prereallocation' );
        printstate (intensity, C, initialresource, planTargets);
    end ;
    new ( expansionrate , C .techniquecount );
    ltechniques← techniques (C);
    for i← 1 to C.techniquecount do
    begin
        t← ltechniques↑[i];
        expansionrate↑[i]← rateOfHarmonyGain (t↑, derivativeOfProductHarmony↑);

    end ;
    meane← meanv (expansionrate↑);

    for i← 1 to C.techniquecount do
    begin

        adjustedexp← sigmoid (expansionrate↑[i]) × temperature × phase2adjust;
```

absolute limit to shrink rate shrink or expand in proportion to gains

```
        intensity_i ← intensity_i × (1 + adjustedexp);
        if (intensity_i < 0) then
        begin
            writeln ( ' intensity ' ,i , ' went negative, adjustedexp=' ,adjustedexp );
            goto 99;
        end ;
    end ;
    dispose ( netOutput );
    netOutput ← computeNetOutput (C, intensity, initialresource);
    dispose ( derivativeOfProductHarmony );
    derivativeOfProductHarmony ← computeHarmonyDerivatives (netOutput↑, planTargets, C, in-
    tensity);
    if (verbose) then
    begin
        writeln( 'postreallocation' );
        printstate (intensity, C, initialresource, planTargets);
    end ;
    rescaleIntensity (intensity, C, initialresource);
end ;
```

# 62 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

# 63 Introduction

## 63.1 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.

## 63.2 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

### 63.2.1 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 ch aracter (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 (AS CII 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"
```

Indicat es 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 |

# 64 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 m ust 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 en d .
- A logical EOF marker cannot be double - quote protected.
- Any record appears after the EOF will be ignored

### 64.1 File Size

Maximum csv file size should be 30 MB.

### 64.2 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

## 65 CSV Record Rules

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

### 65.1 CSV Field Column Rules

- Each recor d 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

### 65.2 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 recor d 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 66 )
    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

field delimitor     const
                      FD = 34;
field separator       FS = 44;
record separator      RS = 10;
                      EOI = $1a;
                      CR = 13;
                    type
                      token = (FDsym);
                      tokenset = set of token ;
                    var
                      categorisor:  array  [byte]  of  token;
```

**function** *getdatamatrix ( p :pcsv ):^ matrix ;* (see Section 66 )

# 66 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 67 )
```

# 67 recursedown

```
procedure recursedown ( j :integer ;q :pcsv );
procedure recurse ( i :integer ;q :pcsv ); (see Section 68 )
```

# 68 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 ;

begin
    if  q ≠ nil  then
    begin
        recurse (0, q↑.right);
        recursedown (j + 1, q↑.down);
    end
end ;
begin
```

```
    if  p = nil  then  getdatamatrix← nil
    else
    begin
        new ( m ,rowcount ( p )-1,colcount ( p )-1);
        recursedown (1, p↑.down);
        getdatamatrix← m;
    end ;
end ;
function getcolheaders ( p :pcsv ):^ headervec ; (see Section 69 )
```

# 69 getcolheaders

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

extract the column headers

```
var
    M;
    h : ↑ headervec ;
procedure recurse ( i :integer ;q :pcsv ); (see Section 70 )
```

# 70 recurse

```
procedure recurse ( i :integer ;q :pcsv );
begin
    if  q ≠ nil  then
    begin

        if  i ≥ 1  then  h↑[i]← q;
        recurse (i + 1, q↑.right);
    end
end ;
begin

    if  p = nil  then  getcolheaders← nil
    else
    begin
        new ( h ,colcount ( p )-1);
        recurse (0, p↑.right);
        getcolheaders← h;
    end ;
end ;
function getrowheaders ( p :pcsv ):^ headervec ; (see Section 71 )
```

# 71 getrowheaders

```
function getrowheaders ( p :pcsv ):^ headervec ;
```

extract the rows headers

```
var
    M;
    h : ↑ headervec ;
procedure recurse ( i :integer ;q :pcsv ); (see Section 72 )
```

# 72 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 73 )
```

# 73 colcount

```
function colcount ( p :pcsv ):integer ;
```

return the number of columns in the spreadsheet

```
begin
    if  p = nil  then  colcount← 0
    else
        case  p↑.tag  of
            linestart : colcount← colcount (p↑.right);
        end
```

```
end ;
function getcell ( p :pcsv ;row ,col :integer ):pcsv ; (see Section 74 )
```

# 74 getcell

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

return the cell at position row,col in the spredsheet

```
begin
    if  p = nil  then  getcell← nil
    else  if  row = 1  then
    begin
        else  if  col = 1  then  getcell← p
    end
end ;

procedure removetrailingnull ( var p :pcsv ); (see Section 75 )
```

# 75 removetrailingnull

```
procedure removetrailingnull ( var p :pcsv );
function onlynulls ( q :pcsv ):boolean ; (see Section 76 )
```

# 76 onlynulls

```
function onlynulls ( q :pcsv ):boolean ;
begin
    if  q = nil  then  onlynulls← false  false
    else
        if  q↑.tag = α  then
        begin
        end
        else  onlynulls← false  false
end ;
begin
    if  p ≠ nil  then
        case  p↑.tag  of
            linestart :
            or ( ( p ^ .down =nil )and onlynulls ( p ^ .right )) then p :=nil
            else  removetrailingnull (p↑.down);
        end
end ;
function rowcount ( p :pcsv ):integer ; (see Section 77 )
```

# 77 rowcount

```
function rowcount ( p :pcsv ):integer ;
begin
    if p = nil then rowcount← 0
    else
        case p↑.tag of
            linestart : rowcount← 1 + rowcount (p↑.down);
            numeric← 1
        end
end ;
function isint ( r :real ):boolean ; (see Section 78 )
```

# 78 isint

```
function isint ( r :real ):boolean ;
var
    i : integer;
begin
    i← round(r);
    isint← (i × 1.0) = r
end ;
procedure printcsv ( var f :text ;p :pcsv ); (see Section 79 )
```

# 79 printcsv

```
procedure printcsv ( var f :text ;p :pcsv );
begin
    if p ≠ nil then
        with p↑ do
        begin
            if tag = linestart then
            begin
                printcsv (f, right);
                if down ≠ 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
        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 ;
function parsecsvfile ( name :textfield ):pcsv ; (see Section 80 )
```

# 80 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 81 )
```

# 81 thetoken

```
function thetoken :token ;
begin
    if currentchar ≤ fs then
        thetoken← categorisor bp↑[currentchar]
    else thetoken← EOFsym
end ;
```

**function** *peek ( c :token ):boolean* ; (see Section 82 )

## 82 peek

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

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

```
begin
    peek← c = thetoken
end ;
function isoneof ( s :tokenset ):boolean ; (see Section 83 )
```

## 83 isoneof

```
function isoneof ( s :tokenset ):boolean ;
begin
    isoneof ← thetoken ∈ s
end ;
procedure nextsymbol ; (see Section 84 )
```

## 84 nextsymbol

```
procedure nextsymbol ;
begin
    if currentchar ≤ fs then currentchar← currentchar + 1
end ;
function have ( c :token ):boolean ; (see Section 85 )
```

## 85 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 86 )

# 86 haveoneof

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

**procedure** *initialise* ; (see Section 87 )

# 87 initialise

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

end ;
```

**procedure** *resolvealpha* ; (see Section 88 )

# 88 resolvealpha

```
procedure resolvealpha ;
var
    i;
begin
    with lastfield↑ do
    begin
        tag← α;
        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 89 )
```

# 89 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 ;
convert to binary
        val (s, number, l);
    end ;
end ;
procedure resolvetoken ; (see Section 90 )
```

# 90 resolvetoken

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

procedure markbegin ; (see Section 91 )
```

# 91 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 92 )
```

## 92 markend

marks the end of a field

```
procedure markend ;
begin
    tokend← currentchar;
    resolvetoken;
end ;
procedure setalpha ( s :textfield ); (see Section 93 )
```

## 93 setalpha

```
procedure setalpha ( s :textfield );
begin
    lastfield↑.tag← α;
    new ( lastfield ^ .textual );
    lastfield↑.textual↑← s;
end ;
procedure emptyfield ; (see Section 94 )
```

## 94 emptyfield

```
procedure emptyfield ;
begin

    markbegin;
    setalpha ( '' );

end ;

procedure parsebarefield ; (see Section 95 )
```

## 95 parsebarefield

```
procedure parsebarefield ;
begin
    if isoneof ([RSsym, EOFsym, FSsym]) then emptyfield
else begin begin
        markbegin;
```

skip over the field

```
        while haveoneof ([any, space]) do   ;
```

```
        markend;
    end ;
end ;
procedure parsedelimitedfield ; (see Section 96 )
```

# 96 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 97 )
```

# 97 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 );
    setalpha (s);
end ;
procedure parsefield ; (see Section 98 )
```

eat what may be closing quotes

# 98 parsefield

```
procedure parsefield ;
```

```
begin
    if have (FDsym) then parsedelimitedfield
    else parsebarefield
end ;
procedure parserecord ; (see Section 99 )
```

## 99 parserecord

```
procedure parserecord ;
begin
    parsefield;
    while have (FSsym) do parsefield;
end ;
procedure parseheader ; (see Section 100 )
```

## 100 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 101 )
```

## 101 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;
the default case of failure    parsecsvfile← nil;

        assign (f, name);
open file for reading    reset (f);
ioresult =0 if opened ok    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;
```

We now have the csv file in memory - parse it

```
                    parsewholefile;
                    removetrailingnull (firstrecord);
                    parsecsvfile← firstrecord;
                end ;
                dispose ( bp );
                close (f);
            end ;
        end ;
    end ;
    begin
        categorisor← any;
        categorisor_FD← FDsym;
        categorisor_FS← FSsym;
        categorisor_RS← RSsym;
        categorisor_EOI← EOFsym;
        categorisor_ord( ' ' )← space;
        categorisor_CR← space;
        {writeln('fs=',fs,'fd=',fd,'rs=',rs);
        writeln(categorisor);}
    end .
```