```
/*************************************************************************/
/**                                                                     **/
/**    Documentation for lp_solve                                       **/
/**                                                                     **/
/**    (C) Hartmut Schwab       Hartmut.Schwab@IWR.Uni-Heidelberg.De    **/
/**                                                                     **/
/**                                                                     **/
/*************************************************************************/
```


In compiling this information, I have drawn on my own knowledge of the field. This file has been read by Michel Berkelaar, the author of lp_solve but nevertheless the errors in this document came into it by me. I give my thanks to all those who have offered advice and support.

Suggestions, corrections, topics you'd like to see covered, and additional material, are all solicited. Send email to

Hartmut.Schwab@IWR.Uni-Heidelberg.De

The newest version of this document is available at the same location than lp_solve.

Version: @(#) lp_solve.doc 1.18@(#), Creation date: 97/02/11.

```
/***************************************************************/
/**                                                           **/
/**                                                           **/
/**            Preface                                        **/
/**                                                           **/
/**                                                           **/
/***************************************************************/
```

This documentation would not exist, if Michel Berkelaar had not written the program lp_solve. Thanks to him, we have a version of the simplex algorithm, where the source code is available, which uses sparse matrix computations and which everybody can include into his or her own applications. More and more users in the Operations Research community are using lp_solve.

His program is finding more and more users in the OR community. The growing interest is easily proven by the still growing number of questions regarding lp_solve in the News. Questions where to find a source code for the simplex algorithm are usually answered with a hint to lp_solve.

It is also important to mention Jeroen Dirks who added a subroutine library to lp_solve. His work makes it easier to include lp_solve in own applications.

Until now, no documentation about the construction of the program had been available. This file wants to close this gap. It wants to provide the reader with some information about the internal structure of the program and add some documentation to lp_solve.

How is this document organised?

You will find a table of contents in the next section. Section "Introduction" describes the intention of this document. You also will find some information, what has been included into this document and what has been excluded from it. The later one is the more important one. This document does not contain every information and it doesn't want to contain every information.

A very important section is "Datastructures used internally by lp_solve". You can guess, what you will find there. If you want to do some extensions, if you detect some errors you probably have to look there to understand the way lp_solve organises all the information internally.

The following sections contain a short description of the functions, but only the MAIN IDEAS are written down. These sections are organised in the same way as the source files. Each source file has its own section.

For easier understanding the structure of the program, I also included the "Function calling tree". It gives an idea how the different functions work together.

If you want to get more information, which is not covered in this document, probably the hints you find in the References will be a good starting point.

This file has been prepared carefully. Nevertheless it probably will contain errors. The author is in no way responsible for any damage caused by using this file.

This document describes version 2.0 of lp_solve. However now major changes are expected to occur in the next time. Therefore this document should be valid also for newer versions of lp_solve.

```
/*****************************************************************/
/**                                                             **/
/**                                                             **/
/**              Contents                                       **/
/**                                                             **/
/**                                                             **/
/*****************************************************************/
```

Contents:
=========

```
- Introduction

- Datastructures used internally by lp_solve

- Short description of the functions, the MAIN IDEAS are written down.

- Function calling tree

- References




/****************************************************************/
/**                                                            **/
/**                                                            **/
/**              Introduction                                  **/
/**                                                            **/
/**                                                            **/
/****************************************************************/


Purpose:
========

   This documentation should help the readers to get a better
   understanding of the source code of lp_solve and its main ideas,
   to be able to make changes in the code and to locate errors in the code,
   if there appear any.
   It should also give the chance to make improvements to the code
   i.e. other product form or different Branch and Bound strategy.



This documentation does NOT describe the simplex algorithm. You can find a
description of the simplex algorithm in every book about linear programming.
If you are interested to find a description which is more adapted to the
sparse form of the simplex, check the literature given in the references.

Some keywords, which describe the implementation of lp_solve:

- selecting pivot variable with largest reduced costs.
  No devex or steepest edge.

- inverting the basis matrix is done in pure product form. No LU
  decomposition.

- Branch and Bound implemented as recursive function, this means pure
  Depth First. There are two strategies for selecting a branching variable.
  Branching on the first non integer variable, branching on a random
  selected variable.


The result is a relatively small and easy to grasp code.
On the other side it can run into numerical problems. More expenditure
has to be done to make the code numerical stable and fast on large
problems. Perhaps somebody wants to add some parts to improve the program.
However this always should be done in a modular way.















/****************************************************************/
```

```
/**                                                              **/
/**                                                              **/
/**             DATASTRUCTURES                                   **/
/**                                                              **/
/**                                                              **/
/*****************************************************************/
```

This section describes the most important data structures of the program. For arrays I tried to give the size of the array and give an idea of the contents of miscellaneous fields, if there are different parts in the array.
The first part of this section should give the main idea, how a linear programming problem is transferred into the internal data structures.
Then a more detailed description of the separate arrays follows.

How is the following linear program transferred to the internal data?

```
max   c'x
s.t.  A x <= b
      lower <= x <= upper
```

All data is placed into arrays which can be accessed via pointers in the structure lprec. There you can find information to the matrix, information, how to interpret the data and the inverted matrix (Eta file).

===================================

Something about numbering rows and columns:

The program is written in C, however indices for the data normally start with 1!

Columns are numbered from 1 to columns.
Rows are numbered from 1 to rows.
row[0] is objective row.

Maximisation or minimisation is only a change of sign in objective row.
This change is marked in ch_sign[0] and in lp->maximise.

===================================

Internally to lp_solve there exist only EQUALITIES or LESSEQUAL rows.
GREATEREQUAL rows are multiplied with -1 and the information, that this is done is written in the array ch_sign.

How to access the right hand side?
Read lp->orig_rhs.

How to access the bounds?
Read lp->orig_lowbo and lp->orig_upbo

How to know the sense of a constraint?
It can be accessed via the information on the bounds of a row. Internally there exist only Less-equal and Equal rows. Upper bound == 0 means row is equal row. Everything else means: It is a Less-Equal row. To distinguish between Less-Equal rows and Greater-Equal rows of the original problem you have to check ch_sign.

How to access information of the original problem, which is not mentioned above?
A good source to find this information is the routine "write_LP" which writes out the whole original problem. It therefore has to access all the original data. You can check this routine to get the requested information?

```
===================================

typedef struct _lprec
{
  nstring   lp_name;        /* the name of the lp */

  short     active;     /*TRUE if the globals point to this structure*/
  short     verbose;            /* ## Verbose flag */
  short     print_duals;        /* ## PrintDuals flag for PrintSolution */
  short     print_sol;          /* ## used in lp_solve */
  short     debug;              /* ## Print B&B information */
  short     print_at_invert;    /* ## Print information at every reinversion */
  short     trace;              /* ## Print information on pivot selection */
  short     anti_degen;     /* ## Do perturbations */

  int       rows;               /* Nr of constraint rows in the problem */
  int       rows_alloc;         /* The allocated memory for Rows sized data */
  int       columns;            /* The number of columns (= variables) */
  int       columns_alloc;
  int       sum;                /* The size of the variables + the slacks */
  int       sum_alloc;

  short     names_used;         /* Flag to indicate if names for rows and
                   columns are used */







  nstring   *row_name;      /* rows_alloc+1 */

 +-------------------+-------------------+-------------------+
 |  Objective_name(?)|  Row_name[1]      |  UNUSED           |
 +-------------------+-------------------+-------------------+
 \________ ________/         ^                              ^
          \/                 |                              |
     25 Bytes               rows                      rows_alloc+1


  Each field has a length of NAMELEN = 25 Byte; in total it has rows_alloc+1
  entries. The first entry seems to be unused.



  nstring   *col_name;      /* columns_alloc+1 */


 +-------------------+-------------------+-------------------+
 |  **************   |  Col_name[1]      |  UNUSED           |
 +-------------------+-------------------+-------------------+
 \________ ________/         ^                              ^
          \/                 |                              |
     25 Bytes             columns                   columns_alloc+1

  Similar to row_name:
  Each field has a length of NAMELEN = 25 Byte; in total it has columns_alloc+1
  entries. The first entry seems to be unused.


 /* Row[0] of the sparse matrix is the objective function */
```

```
  int       non_zeros;          /* The number of elements in the sparse matrix*/
  int       mat_alloc;      /* The allocated size for matrix sized
                   structures */
  matrec    *mat;               /* mat_alloc :The sparse matrix */
  int       *col_end;           /* columns_alloc+1 :Cend[i] is the index of the
                   first element after column i.
                   column[i] is stored in elements
                   col_end[i-1] to col_end[i]-1 */
  int       *col_no;            /* mat_alloc :From Row 1 on, col_no contains the
                   column nr. of the
                                nonzero elements, row by row */
  short     row_end_valid;  /* true if row_end & col_no are valid */
  int       *row_end;           /* rows_alloc+1 :row_end[i] is the index of the
                   first element in Colno after row i */
  REAL      *orig_rh;           /* rows_alloc+1 :The RHS after scaling & sign
                  changing, but before `Bound transformation' */
  REAL      *rh;        /* rows_alloc+1 :As orig_rh, but after Bound
                   transformation */
  REAL      *rhs;       /* rows_alloc+1 :The RHS of the current simplex
                   tableau */
  short     *must_be_int;       /* sum_alloc+1 :TRUE if variable must be
                   Integer */
  REAL      *orig_upbo;         /* sum_alloc+1 :Bound before transformations */
  REAL      *orig_lowbo;    /*  "       "                   */
  REAL      *upbo;              /*  "       "  :Upper bound after transformation
                   & B&B work*/
  REAL      *lowbo;             /*  "       "  :Lower bound after transformation
                   & B&B work */

  short     basis_valid;        /* TRUE if the basis is still valid */
  int       *bas;               /* rows_alloc+1 :The basis column list */
  short     *basis;             /* sum_alloc+1 : basis[i] is TRUE if the column
                   is in the basis */
  short     *lower;             /*  "       "  :TRUE if the variable is at its
                   lower bound (or in the basis), it is FALSE
                   if the variable is at its upper bound */


 The following

     max c'x


     s.t.  A*x <= b
           l <= x <= u

symbolically:

    +------------------------------+
    |             c                |
    +------------------------------+

    +------------------------------+    +-+
    |                              |    | |
    |             A                | <= |b|
    |                              |    | |
    |                              |    | |
    +------------------------------+    +-+


     +------------------------------+
     |          upper Bound         |
     +------------------------------+

     +------------------------------+
     |          lower Bound         |
     +------------------------------+



is transformed to
```

```
     +------------------------------+    +-+      +-+      +-+  \
     |             c                |    | |      | |      | |   |    := row[0]
     +------------------------------+    +-+      +-+      +-+   |
     |                              |    | |      | |      | |   \  rows_alloc+1
     |             A                | <= | |      | |      | |   /
     |                              |    | |      | |      | |   |
     |                              |    | |      | |      | |   |
     +------------------------------+    +-+      +-+      +-+  /
                                       orig_rh    rh      rhs
                                     scaled +    =orig_rh current rhs
                                     signchange  + Bound  dh. Basis values(?)
                                                 transform
```

sum_alloc = rows_alloc + columns_alloc;
Be careful: There is a difference between "rows" and "rows_alloc". We allocate "rows_alloc" elements, but use only "rows" elements. This means, the space between "rows" and "rows_alloc" is empty/not used.

The same is true for "columns" and "columns_alloc".
The pair for matrix is called "non_zeros" and "mat_alloc".

```
 rows
    |
  rows_alloc+1                 sum_alloc+1. Index [0] seems to be unused.
    | |                             |
slack v                             v
+----+------------------------------+
|    |          must_be_int         |
+----+------------------------------+

slack
+----+------------------------------+
|    |          orig_upbo           | before Bound transform
+----+------------------------------+

slack
+----+------------------------------+
|    |          orig_lowbo          | before Bound transform
+----+------------------------------+

slack
+----+------------------------------+
|    |               upbo           | after Bound transform and in B+B
+----+------------------------------+

slack
+----+------------------------------+
|    |              lowbo           | after Bound transform and in B+B
+----+------------------------------+


slack
+----+------------------------------+
|    |          Basis               |  TRUE, if column is in Basis.
+----+------------------------------+  FALSE otherwise.

slack
+----+------------------------------+
|    |          lower               |  TRUE, if column is in Basis or
+----+------------------------------+  nonbasic and at lower bound.
 ^                                     FALSE otherwise.
 |
 0


     +-+
     | |
     +-+
```

```
    | |
    | |   indices of columns which are in Basis.
    | |
    | |
    +-+
    bas
```

The matrix is stored in sparse form in the usual way.

```
  int       non_zeros;          /* The number of elements in the sparse matrix*/
  int       mat_alloc;      /* The allocated size for matrix sized
                   structures */
  matrec    *mat;               /* mat_alloc :The sparse matrix */
  int       *col_end;           /* columns_alloc+1 :Cend[i] is the index of the
                   first element after column i.
                   column[i] is stored in elements
                   col_end[i-1] to col_end[i]-1 */
  int       *col_no;            /* mat_alloc :From Row 1 on, col_no contains the
                   column nr. of the
                                nonzero elements, row by row */
  short     row_end_valid;  /* true if row_end & col_no are valid */
  int       *row_end;           /* rows_alloc+1 :row_end[i] is the index of the
                   first element in Colno after row i */

  +-----+-----+-----+-----+-----+-----+-----+
  |  1  |  3  |  7  |  1  | *** | *** | *** |        (row_nr)
  +-----+-----+-----+-----+-----+-----+-----+   mat
  | 2.5 | 4.7 | 1.0 | 2.0 | *** | *** | *** |        (value)
  +-----+-----+-----+-----+-----+-----+-----+
                           ^               ^
                           |               |
                       non_zeros         mat_alloc
```

Entry Zero is valid.

```
  +-----+-----+-----+-----+
  | *** |  3  |  4  |     |    col_end  (in fact beginning of next column)
  +-----+-----+-----+-----+
               ^         ^
               |         |
            columns   columns_alloc+1
```

Entry Zero is NOT valid.(?)

```
  +-----+-----+-----+-----+-----+-----+-----+
  |  1  |  2  |  5  |  1  | *** | *** | *** |  col_no: Which columns appear in
  +-----+-----+-----+-----+-----+-----+-----+          row[i]. Row[i] starts at
                           ^               ^          row_end[i-1] and ends at
                           |               |          row_end[i] - 1.
                       non_zeros         mat_alloc
```

ATTENTION: Documentation in header file seems to be wrong!!!
col_no[0] is not used! row[i] starts in row_end[i-1]+1 and ends in row_end[i].
In array there are used (non_zero - number of coefficients in objective row +1)
elements used.

Col_no is used in invert. (And nowhere else, but set in Isvalid)

```
    +-+
    | |
    +-+
    | |
    | |  How many coefficients are in rows 1 to i. Equivalent:
    | |  row_end[i] is the index of the first element in col_no after row i.
    | |
    +-+
```

```
    row_end
```

How is sense of the constraints/rows coded?

Look at slack variable of the row. If the orig_upbo[i] < infinite (this should be: orig_upbo[i] == 0) then we have an equality row. This comparison can be found in write_MPS(), where all Rows with upper bound == infinity are "L" or "G" rows. All other rows are "E" rows. See also write_LP().

In the other cases, that means orig_upbo[[i] == infinite, we have to look at ch_sign[i]. If ch_sign[i] == TRUE, we have a greater equal row, if ch_sign == FALSE, we have a less equal row.

=================================

```
  short     eta_valid;          /* TRUE if current Eta structures are valid */
  int       eta_alloc;          /* The allocated memory for Eta */
  int       eta_size;           /* The number of Eta columns */
  int       num_inv;            /* The number of real pivots */
  int       max_num_inv;        /* ## The number of real pivots between
                    reinvertions */
  REAL      *eta_value;         /* eta_alloc :The Structure containing the
                    values of Eta */
  int       *eta_row_nr;         /*  "     "  :The Structure containing the Row
                    indexes of Eta */
  int       *eta_col_end;       /* rows_alloc + MaxNumInv : eta_col_end[i] is
                    the start index of the next Eta column */

     +-------+---------------------------+
     |       |    eta_col_end            |   Startindex of next Eta column
     +-------+---------------------------+
     rows_alloc     max_num_inv       ^
  this is needed    we can have      eta_size
  for first         maximal so many
  invert            inverts, i.e. etamatrices.
                    until next inversion.


     +---+---+---+---+---+---+---+---+---+---+---+---+---+
     |   |   |   |   |   |   |   |   |   |   |   |   |   | eta_value
     +---+---+---+---+---+---+---+---+---+---+---+---+---+

     +---+---+---+---+---+---+---+---+---+---+---+---+---+
     |   |   |   |   |   |   |   |   |   |   |   |   |   | eta_row_nr
     +---+---+---+---+---+---+---+---+---+---+---+---+---+
                                                         ^
                                                         |
                                                     eta_alloc
```

Normal way of sparse matrix representation. But how to built one Eta Matrix? I do not know, in which column to put eta-column.

Guess:
This is one eta_matrix. Last entry contains index of the column and value on the diagonal.

```
     +---+---+---+---+---+
     |1.3|.5 |.8 |.7 |2.5|   eta_value
     +---+---+---+---+---+

     +---+---+---+---+---+
     | 1 | 3 | 7 | 8 | 6 |   eta_row_nr
     +---+---+---+---+---+
```

The matrix in dense form would be:

```
    +---+---+---+---+---+---+---+---+
    | 1 |   |   |   |1.3|   |   |
    +---+---+---+---+---+---+---+---+
```

```
     |   | 1 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+---+
     |   |   | 1 |   |   |.5 |   |   |
     +---+---+---+---+---+---+---+---+
     |   |   |   | 1 |   |   |   |   |
     +---+---+---+---+---+---+---+---+
     |   |   |   |   | 1 |   |   |   |
     +---+---+---+---+---+---+---+---+
     |   |   |   |   |   |2.5|   |   |
     +---+---+---+---+---+---+---+---+
     |   |   |   |   |   |.8 | 1 |   |
     +---+---+---+---+---+---+---+---+
     |   |   |   |   |   |.7 |   | 1 |
     +---+---+---+---+---+---+---+---+
                           ^
                           |
                      column 6






  short     bb_rule;        /* what rule for selecting B&B variables */

  short     break_at_int;       /* TRUE if stop at first integer better than
                                   break_value */
  REAL      break_value;

  REAL      obj_bound;          /* ## Objective function bound for speedup of
                   B&B */
  int       iter;               /* The number of iterations in the simplex
                   solver (LP) */
  int       total_iter;         /* The total number of iterations (B&B) (ILP)*/
  int       max_level;          /* The Deepest B&B level of the last solution */
  int       total_nodes;    /* total number of nodes processed in b&b */








  REAL      *solution;          /* sum_alloc+1 :The Solution of the last LP,
                   0 = The Optimal Value,
                                   1..rows The Slacks,
                   rows+1..sum The Variables */
  REAL      *best_solution;     /*  "        "  :The Best 'Integer' Solution */
  REAL      *duals;             /* rows_alloc+1 :The dual variables of the
                   last LP */


                              sum_alloc+1. Index [0] is optimal solution value.
                                    |
 slack                              v
 +----+-----------------------------+
 |    |          solution           |
 +----+-----------------------------+
  ^  ^                             ^
  |  |                             |
  |  rows                       sum=rows+columns
 Optimal
 value
 = Index 0


 slack
 +----+-----------------------------+
 |    |          best_solution      | Best integer solution so far.
 +----+-----------------------------+

      +-+
```

```
       | |
       +-+
       | |
       | |  dual variables
       | |
       | |
       +-+
       duals






    short     maximise;           /* TRUE if the goal is to maximise the
                     objective function */
    short     floor_first;        /* TRUE if B&B does floor bound first */
    short     *ch_sign;           /* rows_alloc+1 :TRUE if the Row in the matrix
                     has changed sign
                                  (a`x > b, x>=0) is translated to
                     s + -a`x = -b with x>=0, s>=0) */




       +-+
       | |
       +-+
       | |
       | | TRUE or FALSE
       | |
       | |
       +-+
       ch_sign    (compare sense of a row described above)

    short     scaling_used;   /* TRUE if scaling is used */
    short     columns_scaled;     /* TRUE is the columns are scaled too, Only use
                     if all variables are non-integer */
    REAL      *scale;             /* sum_alloc+1 :0..Rows the scaling of the Rows,
                     Rows+1..Sum the scaling of the columns */



                                 sum_alloc+1
                                     |
  rows            columns            v
  +----+-----------------------------+
  |    |                             | scale: Scaling factors for rows and columns
  +----+-----------------------------+
   ^  ^                             ^
   |  |                             |
   0 rows                        sum=rows+columns





    int       nr_lagrange;    /* Nr. of Langrangian relaxation constraints */
    REAL      **lag_row;      /* NumLagrange, columns+1:Pointer to pointer of
                     rows */
    REAL      *lag_rhs;       /* NumLagrange :Pointer to pointer of Rhs */
    REAL      *lambda;        /* NumLagrange :Lambda Values */
    short     *lag_con_type;      /* NumLagrange :TRUE if constraint type EQ */
    REAL      lag_bound;      /* the lagrangian lower bound */

    short     valid;      /* Has this lp passed the 'test' */
    REAL      infinite;           /* ## numerical stuff */
    REAL      epsilon;            /* ## */
    REAL      epsb;               /* ## */
    REAL      epsd;               /* ## */
    REAL      epsel;              /* ## */
  } lprec;
```

The HASH structure:
===================

```
 Hash_tab
   +---+
   |   |       hashelem
   +---+      +----------+
   | -------->| colname  |
   +---+      +----------+
   |   |      | -------------------------------------------> next
   +---+      +----------+                         column
   |   |      | -------------------------------> +----------+
   +---+      +----------+      bound            |  row     |
   |   |      | ------------>+-------------+     +----------+
   +---+      +----------+   |  upbound    |     |  value   |
   |   |      |must_be_i|    +-------------+     +----------+
   +---+      +----------+   |  lobound    |     |  -------------> next
   |   |                     +-------------+     +----------+
   +---+
   |   |
   +---+
```

```
typedef struct _hashelem
{
  nstring          colname;
  struct _hashelem *next;
  struct _column   *col;
  struct _bound    *bnd;
  int              must_be_int;
} hashelem;


typedef struct _column
{
  int            row;
  float          value;
  struct _column *next ;
} column;

typedef struct _bound
{
  REAL          upbo;
  REAL          lowbo;
} bound;
```

```
   First_rside

   +---------+        +---------+
   | value   |        | value   |
   +---------+        +---------+
   | ------------->   | ------------->
   +---------+        +---------+
   | relat   |        | relat   |
   +---------+        +---------+
```

```
typedef struct _rside /* contains relational operator and rhs value */
```

```
{
  REAL          value;
  struct _rside *next;
  short         relat;
} rside;





   First_constraint_name

   +--------+         +--------+
   | name   |         | name   |
   +--------+         +--------+
   | row    |         | row    |
   +--------+         +--------+
   | ------------->   | ------------->
   +--------+         +--------+


typedef struct _constraint_name
{
  char                  name[NAMELEN];
  int                   row;
  struct _constraint_name *next;
} constraint_name;






typedef struct _tmp_store_struct
{
  nstring name;
  int     row;
  REAL    value;
  REAL    rhs_value;
  short   relat;
} tmp_store_struct;









/*****************************************************************/
/**                                                             **/
/**                                                             **/
/**             Routines in file "solve.c"                      **/
/**                                                             **/
/**                                                             **/
/*****************************************************************/





This file contains the routines which are important to use the
SIMPLEX algorithm. For example you find routines to do basis
exchange, to select new pivot element etc. in this file.


set_globals()   Copy/initialize the global variables for a special LP.

ftran (int start,
       int end,
       REAL *pcol)         /* The column which will be used in ftran */
        /* multiply the column with all matrices in etafile */

```

```
    for every matrix between start and end
        calculate End of the matrix
        r = number of column in Eta matrix
        theta = pcol[r]
        for one matrix
            multiply pcol with the matrix (?)
        update pcol[r]
    round values in pcol

btran (REAL *row)
    For all Eta matrices, Starting with the highest number
        k = number of column in Eta-matrix
        for one matrix
            do multiplication
        round result
        set row[Eta_row_nr[k]] = result


Isvalid (lprec *lp)
        Calculate the structures row_end[] and
        col_no[].
        internally two arrays are used:
        row_nr[], which contains the number of coefficients
        in row[i] and num[], which is a working array and
        contains the already used part of col_no in row[i].
        The array col_no is written at several positions
        at the same time. So it could look like

        +--------------------------------------+
        |**          ****          *      **   |
        +--------------------------------------+

        The second part of the routine uses two arrays
        rownum[] and colnum[]. It tests, if there are some
        empty columns in the matrix and prints a
        Warning message in this case.

        In detail:
        if (!lp->row_end_valid)
           malloc space for arrays num and rownum.
           initialise with zero
           count in rownum[i] how many coefficients are in row i
           set row_end (ATTENTION: documentation of row_end
                                  seems to be wrong. row_end points to LAST
              coefficient in row. But this is never used??)
              col_no[0] is not used!!!
           loop through all the columns,
              forget row[0] = objective row
              write column index in array col_no.
           free num, rownum
           row_end_valid = TRUE
        if (!lp->valid)
           Calloc rownum, colnum.
           for all columns
              colnum[i]++, for every coefficient in column.

           if colnum[i] = 0, print warning.


resize_eta()        simple REALLOC




condensecol(int row_nr, REAL *pcol)
    if necessary:
        resize_eta()
    For all rows
        if i <> row_nr && pcol[i] <> 0
            Eta_row_nr = i
            Eta_value  = pcol[i]
            elnr++
```

```
    /* Last Action: write element for diagonal */
    Eta_row_nr = row_nr
    Eta_value = pcol[row_nr]

    update Eta_col_end


addetacol()
    Determine Begin and End of last Eta matrix.
    calculate theta = 1/ Eta_value[Diagonal element]
    multiply all coefficients in last matrix with -theta
    JustInverted = FALSE




setpivcol(short lower, int varin, REAL *pcol)
/* Main idea: take one column from original matrix and call ftran */

    /* init */
        ...
        pcol[i] = 0  for all i

    If variable   /* This means not surplus/slack variable */
        copy column coefficients into pcol[]
        Actualise pcol[0] -= Extrad*f
    else          /* surplus/slack variable */
        /* This column is column from identity matrix */
        pcol[varin] = 1 or -1

    ftran(1, Etasize, pcol)



minoriteration(colnr, row_nr)
    set varin
    elnr = wk = Eta_col_end[Eta_size]  /* next free element */
    Eta_size++
    /* Eta_size = number of matrices in Eta file */
    /* Eta_col_end = End of one matrix           */

    Do something if Extrad <> 0          /* I do not know what to do
                                            and what is Extrad  */
    For all coefficients in column
        set row_nr
        If Objective_row and Extrad
            Eta_value[Eta_col_end[Eta_size -1]] += Mat[j].value
        else if k <> row_nr
            Eta_row_nr[elnr] = k
            Eta_value[elnr] = Mat[j].value
            elnr++
        else
            piv = Mat[j].value

    /* Last action: Write element on diagonal */
    insert row_nr and 1/piv

    theta = rhs[row_nr] / piv
    Rhs[row_nr] = theta

    For all coefficients of last eta matrix without diagonal element
        Rhs[Eta_row_nr[i]] -= theta*Eta_value[i]

    /* set administration data for Basis */
    varout =
    Bas[row_nr] = varin
    Basis[varout] = FALSE
    Basis[varin] = TRUE

    For all coefficients of last eta matrix without diagonal element
        Eta_value[i] /= -piv

    /* update Eta_col_end */
```

```
    Eta_col_end[Eta_size] = elnr



rhsmincol(REAL theta, int row_nr, int varin)
    Error test
    Find for last matrix in etafile the begin and end
    for all coefficients in last eta matrix without diagonal coefficient
        calculate rhs[eta_row_nr] -= theta * etavalue[i]
    rhs[row_nr] = theta

    varout = bas[row_nr]
    Bas[row_nr] = varin
    Basis[varout] = FALSE
    Basis[varin]  = TRUE



invert()
    allocate

    +---+---+---+---+---+---+
    | 0 |   |   |   |   |   | rownum
    +---+---+---+---+---+---+
                             ^
                             |
                           Rows+1

    +---+---+---+---+---+---+
    |   |   |   |   |   |   | col
    +---+---+---+---+---+---+

    +---+---+---+---+---+---+
    |   |   |   |   |   |   | row
    +---+---+---+---+---+---+

    +---+---+---+---+---+---+
    |   |   |   |   |   |   | pcol              REAL pivot column??
    +---+---+---+---+---+---+

    +---+---+---+---+---+---+
    |TRU|   |   |   |   |   | frow              short
    +---+---+---+---+---+---+

    +---+---+---+---+---+---+---+---+
    |FAL|   |   |   |   |   |   |   | fcol      short
    +---+---+---+---+---+---+---+---+
                                     ^
                                     |
                                  columns+1

    +---+---+---+---+---+---+---+---+
    | 0 |   |   |   |   |   |   |   | colnum   /* count number of
    +---+---+---+---+---+---+---+---+             Coefficients which
                                                  appear in basis matrix */

    change frow and fcol depending on Bas[i]
        frow = FALSE if Bas[i] <= Rows
        fcol = TRUE  if Bas[i] >  Rows


    set
        Bas[i] = i   for all i
        Basis[i] = TRUE   for all slack variables
                   FALSE  for all other variables
        Rhs[i] = Rh[i]   for all i    /* initialise original Rhs */

    Correct Rhs for all Variables on upper bound
    Correct Rhs for all slack variables on upper bound (if necessary)

    Etasize =0
    v = 0
    row_nr = 0
```

```
    Num_inv = 0
    numit = 0

    Look for rows with only one Coefficient  (while)
        if found
        look for column of coefficient
        set     fcol[colnr - 1] = FALSE  /* This col is no longer
                                                        in Basis */
            colnum[colnr] = 0
        correct rownum counter
        set frow[row_num] = FALSE    /* This row is no longer
                                                    in Basis */
        minoriteration(colnr, row_nr)

    Look for columns with only one Coefficient  (while)
        if found

        set frow[row_num] = FALSE    /* This row is no longer
                                                    in Basis */
        rownum[] = 0
        update column[]
        numit++                /* counter how many iterations to do */
                                        /* at end                            */
        col[numit] = colnr     /* replaces minoriteration. But this */
                                        /* is done later and we need arrays  */
        row[numit] = row_nr    /* col and row therefore             */

    /* real invertation */
    for all columns   (From beginning to end )
        if fcol
            set fcol[] = FALSE
            setpivcol ( Lower[Rows + j] , Rows + j, pcol)
            Loop through all the rows to find coefficient with
                frow[row_nr] && pcol[row_nr]
            /* Interpretation:
               Look for first coefficient in (partly inverted)
                           Basis matrix which is nonzero and use it for pivot.*/

            /* comparison for pcol is dangerous, but ok after
                           rounding */

            Error conditions

            /* Now we know pivot element */

            frow[row_nr] = FALSE
            condensecol (row_nr, pcol)
            rhsmincol (theta, row_nr, Rows + j)
            addetacol()

    For all stored actions   /* compare numit */
        set colnr / varin
            row_nr
        init pcol with 0
        set pcol[]
        actualize pcol[0]
        condensecol(row_nr, pcol)
        rhsmincol(theta, row_nr, varin)
        addetacol()

    Round Rhs
    print info
    Justinverted = TRUE
    Doinvert = FALSE

    free()



colprim(int *colnr,
          short minit,
          REAL* drow)
    /* Each, colprimal - rowprimal and rowdual - coldual form a couple */
```

```
    btran( 1,0,0,0,0,0,...)
    update result depending on variables at upper bound

        look for variable with negative reduced costs

    ========================
    More detailed:

    init *colnr = 0
         dpiv = set to a small negative number.
    if  NOT minit
        drow = 1,0,0,0,0......
        btran(drow)
        For variables at upper bound we have to calculate the
        reduced cost differently:
        multiply each coefficient in column with reduced cost of row=
        slackvariable and sum. This is new reduced cost.
        round reduced costs "drow"
    Look for variable which has upper bound Greater than Zero, which
    is nonbasic.
        Perhaps correct sign of reduced costs of variables
        at upper bound.
        take variable with most negative reduced costs.
        save reduced costs in dpiv and
        col_nr in *col_nr
    print trace info
    if *col_nr == 0
        set some variables, that indicate that we are optimal.
    return True, if *col_nr > 0



rowprim(int colnr,
    int * row_nr,
    REAL * theta,
    REAL * pcol)            /* contains current column from var conr */

    /* search for good candidate in a column for pivot */
    First look for big entries
    Second ( this means first failed ) look also for smaller entries
    Warning numerical instability

    Determine UNBOUNDED
    Perhaps shift variable to its upper bound

    Aim: determin valid pivot element

    print some info

    Return true, if we had been successful finding a pivot element.

    ========================
    More detailed:

    init *row_nr = 0
         *theta = infinity
    loop through all the rows
        qout = maximal steplength = *thetha
        *row_nr = number of that row.
        /* At first look only for Steps which are not calculated with
           very small divisors. If no such steps found, take also
           small divisors in consideration */
    Perhaps we found numerical problems. Print warning in this case

    If we did not find a limiting row, we are perhaps unbounded.
    (upperbound on that variable = infinity)
    The case that we have an upper bound is treated separately

    print some trace info

    return (*row_nr > 0)
```

```
rowdual(int *row_nr)
    look for infeasibilities

    init *row_nr = 0
             minrhs = a little bit negative

    loop through all the rows
        if we find a variable which is not zero, but has to be
        then we break this loop. *row_nr = i

        calculate distance between rhs[i] and upperbound[i]
        take smaller one

        |-------|----------------|
                0      rhs[i]          upperbound[i]
                       =g


        minrhs is smallest g
                *row_nr is corresponding rownumber.

    print some trace info
    return (*row_nr > 0)


coldual(int row_nr,
    int *colnr,
    short minit,
    REAL *prow,
    REAL *drow)
    looks also for a candidate for pivot.




iteration (int row_nr,
           int varin,
           REAL *theta,
           REAL up,
           short *minit,
           short *low,
           short primal,
           REAL *pcol)

        execute one iteration




solvelp()
                First check if right hand side is positive everywhere
                and smaller than possible upper bound of this row.
                In this case we start with a feasible basis.



        ATTENTION:
        If we want to use solvelp() directly, skipping
        solve() and milpsolve() we have to be very careful.
        e.g. solve() sets the global variables!!!

is_int(REAL value)
        simple routine, checks, if a REAL value is integer.

construct_solution(REAL *sol)
                The routine does exactly, what its name says.
                There are two parts, with and without scaling.


                First set all variables to their lower bounds.
                Then set all basis variables to their true values, i.e.
                the right hand side is added to the lower bound.
```

```
                (The reason is that all variables have been transformed
                to have lower bound zero) ## Autor fragen!!
                Finally set the non basic variables, which are not at
                their lower bound to their upper bound.
                Calculate values of the slack variables of a row.



calculate_duals()
                In fact calculate the reduced costs of the slack variables
                and correct values.



milpsolve(REAL   *upbo,
          REAL   *lowbo,
          short  *sbasis,
          short  *slower,
          int    *sbas)
            First of all: copy the arrays upbo and lowbo
            to the pointers of Upbo and Lowbo. (Memory
            is allocated for these arrays. Pointers point
            to lp->upbo and lp->lowbo)
            (size of memory is updated, if new columns are
            added.)
            These arrays came from solve() as ORIGINAL
            bounds. Therefore no shifting of transformed bounds
            necessary in lpkit.c if solve() is called.

            if (LP->anti_degen)
               disturb lower and upper bound a little bit.
            if (!LP->eta_valid)
               shift lower bounds to zero. This means:
               Orig_lowbo   ... unchanged
               Orig_upbo    ... unchanged
               lowbo        ... unchanged (implicit in code = 0)
               upbo         ... mainly upbo_old - lowbo.

            solvelp()

            if (LP->anti_degen)
               restore upbo, lowbo, Orig_rh and solve again.

            if (OPTIMAL solution of LP)
               check, if we can cutoff branch with LP value.
               look for noninteger variable (look for first
               or look random)
               if (noninteger variables)
                  setup two new problems.
                  Malloc new memory
                  memcpy the data
                  solve problems recursively (Floor_first/ceiling_irst)
                  set return values
               else
                  /* all required values are int */
                  check, if better solution found.
                  /* Yes */
                  memcpy data
                  perhaps break B+B


            Recursive Function. Pure depth first search.
                        No easily accessible nodelist, because of depth
                        first search. (Also less active nodes)
            Branching on first noninteger variablen or
                        on a randomly selected variable.
            Avoid inverting if possible.



solve(lprec *lp)
            init BEST-Solution, init perhaps basis, call milpsolve
```

```
lag_solve(lprec *lp, REAL start_bound, int num_iter, short verbose)
        Lagrangean solver.





/*****************************************************************/
/**                                                             **/
/**                                                             **/
/**             Routines in file "debug.c"                      **/
/**                                                             **/
/**                                                             **/
/*****************************************************************/


static void print_indent(void)
void debug_print_solution()
void debug_print_bounds(REAL *upbo, REAL *lowbo)
void debug_print(char *format, ...)


===================================================

static void print_indent(void)
   Used for printing the branch and bound tree. For every node the depth
   in the tree is shown with some ASCII graphic.

void debug_print_solution()
   For all columns
      print_indent()
      print the variable name (true or artificial) and its value.

void debug_print_bounds(REAL *upbo, REAL *lowbo)
   For all columns
      Print the lower bounds if they are different from zero with true or
      artificial name.
      Print the upper bounds if they are different from infinity with true or
      artificial name.

void debug_print(char *format, ...)




/*****************************************************************/
/**                                                             **/
/**                                                             **/
/**             Routines in file "lp_solve.c"                   **/
/**                                                             **/
/**                                                             **/
/*****************************************************************/


void print_help(char *argv[])
   Print usage message. If the program is called with option "-h" the usage
   message is printed. The usage message gives the options which can be
   given when calling the program.

int main(int argc, char *argv[])
   Initialise some data. Read all the options and make use of them.
   (Options have to be given separately, non existing options are just ignored.)

   Read MPS file or lp_file from stdin.

   Perhaps print out some information about LP and do some manipulations on
   LP (i.e. scaling).

   call solve(lp)
```

```
   Check return status:
   If (OPTIMAL)
      Print out solution and some statistics.
   else print solution status.
```

```
/****************************************************************/
/**                                                            **/
/**                                                            **/
/**             Routines in file "lpkit.c"                     **/
/**                                                            **/
/**                                                            **/
/****************************************************************/
```

The main purpose of this file is to give several "manipulation" routines to the user. The user should be able to read information from the current problem. But he/she should also be able to change information in the problem. So for example, it is possible to add new constraints to the problem, to change the bounds of the variables etc.

```
void error(char *format, ...)
lprec *make_lp(int rows, int columns)
void delete_lp(lprec *lp)
lprec *copy_lp(lprec *lp)
void inc_mat_space(lprec *lp, int maxextra)
void inc_row_space(lprec *lp)
void inc_col_space(lprec *lp)
void set_mat(lprec *lp, int Row, int Column, REAL Value)
void set_obj_fn(lprec *lp, REAL *row)
void str_set_obj_fn(lprec *lp, char *row)
void add_constraint(lprec *lp, REAL *row, short constr_type, REAL rh)
void str_add_constraint(lprec *lp,
                        char *row_string,
                        short constr_type,
                        REAL rh)
void del_constraint(lprec *lp, int del_row)
void add_lag_con(lprec *lp, REAL *row, short con_type, REAL rhs)
void str_add_lag_con(lprec *lp, char *row, short con_type, REAL rhs)
void add_column(lprec *lp, REAL *column)
void str_add_column(lprec *lp, char *col_string)
void del_column(lprec *lp, int column)
void set_upbo(lprec *lp, int column, REAL value)
void set_lowbo(lprec *lp, int column, REAL value)
void set_int(lprec *lp, int column, short must_be_int)
void set_rh(lprec *lp, int row, REAL value)
void set_rh_vec(lprec *lp, REAL *rh)
void str_set_rh_vec(lprec *lp, char *rh_string)
void set_maxim(lprec *lp)
void set_minim(lprec *lp)
void set_constr_type(lprec *lp, int row, short con_type)
REAL mat_elm(lprec *lp, int row, int column)
void get_row(lprec *lp, int row_nr, REAL *row)
void get_column(lprec *lp, int col_nr, REAL *column)
void get_reduced_costs(lprec *lp, REAL *rc)
short is_feasible(lprec *lp, REAL *values)
short column_in_lp(lprec *lp, REAL *testcolumn)
void print_lp(lprec *lp)
void set_row_name(lprec *lp, int row, nstring new_name)
void set_col_name(lprec *lp, int column, nstring new_name)
static REAL minmax_to_scale(REAL min, REAL max)
void unscale_columns(lprec *lp)
void unscale(lprec *lp)
void auto_scale(lprec *lp)
void reset_basis(lprec *lp)
void print_solution(lprec *lp)
void write_LP(lprec *lp, FILE *output)
void write_MPS(lprec *lp, FILE *output)
void print_duals(lprec *lp)
void print_scales(lprec *lp)
```

What is done in the routines:
=============================

void error(char *format, ...)
lprec *make_lp(int rows, int columns)

Construct a new LP. Set all variables to some default values.
The LP has "rows" rows and "columns" columns. The matrix contains
no values, but space for one value. All arrays which depend on
"rows" and "columns" are malloced.

The problem contains only continuous variables.
Upper bounds are infinity, lower bounds are zero.
The basis is true, all rows are in basis. All columns are nonbasic.
The eta-file is valid. Solution, best_solution and duals are Zero.
And some other default values.

void delete_lp(lprec *lp)

Delete ALL the malloced arrays. At last free the structure.

lprec *copy_lp(lprec *lp)

Copy first the structure of the lp, this means especially, that all
the constant values are copied.
Copy all the arrays of the lp and set the pointers to the new arrays.
Mainly use MALLOCCOPY for this, this means: malloc space and copy data.

void inc_mat_space(lprec *lp, int maxextra)

Test if realloc necessary. If yes, realloc arrays "mat" and "col_no".
If Lp is active, set some global variables which could be changed by
realloc.

void inc_row_space(lprec *lp)

Test, if increment necessary.
This routine increments the space for rows with 10 additional rows.
Therefore one condition for correct work of this routine is that
it is never necessary to increase the
number of additionally rows in one step with more than 10!
Several arrays are realloced.
At last, if LP is active, set some global variables new, because they could
have changed.

void inc_col_space(lprec *lp)

similar to routine increment row space. The problems are also the same.
Several Arrays are realloced, but no shift of values.

void set_mat(lprec *lp, int Row, int Column, REAL Value)

set one element in matrix.
Test, if row and column are in range. Scale value.
If colnum is in basis and row not objective row set Basis_valid = FALSE
Always set eta_valid = FALSE (is this necessary?)

Search in column for entry with correct rownumber.
If row found scale value again but with other expression than first time.
Perhaps change sign

If row not found:

```
     Increment mat_space for one additional element.
     Shift matrix and update col_end.
     Set new element "row" and scale value perhaps (same problem as above)
     Rowend is not valid any longer
     update number of nonzeros and copy this value if lp is active



void set_obj_fn(lprec *lp, REAL *row)

    call in one loop for dense row the function set_mat().
    No test is done, if we want to include Elements with value "0".
    These values are included into the matrix!



void str_set_obj_fn(lprec *lp, char *row)

    reserve space for one row
    try with "strtod()" to change all the strings to real values
    call set_obj_fn()
    free space



void add_constraint(lprec *lp, REAL *row, short constr_type, REAL rh)

    first reserve space for integers for length of one row.
    Mark all the positions, which contain nonzeros and update non_zeros
    malloc space for a complete new matrix
    increment matrix space by null??
    rows++
    sum++
    increment row space
    if scaling
       shift the values
       and set scaling value for new row to 1
    if names used
       invent new name for row
    if columns are scaled
       scale coefficients
    calculate change_sign
    copy every column from old matrix to new matrix. Perhaps add new entry for
    new row.
    Update col_end
    copy new matrix back to old matrix.
    free the allocated arrays

    shift orig_upper_bounds
          orig_lower_bounds
          basis
          lower
          must_be_int

    update Basis info
    set bounds for slack variables

    change_sign for rhs, but comparison is made with sense of constraint.

    rows_end_valid = false
    put slackvariable for this row into basis
    if lp == active, set globals
    eta_file = non_valid.



void str_add_constraint(lprec *lp,
                        char *row_string,
                        short constr_type,
                        REAL rh)

    This routine is similar to the routine str_set_obj_fn. The same idea,
    but call add_constraint.
```

```
void del_constraint(lprec *lp, int del_row)

   First check, if rownumber exists.
   For all columns
      For every coefficient in column
         if it is not rownumber,
         then shift elements to smaller nonzero index and perhaps correct row index.
         else  delete
         update col_end
   shift values for orig_rhs, ch_sign, bas, row_name down by one.
   Update values in bas
   shift values for lower, basis, orig_upbo, orig_lowbo, must_be_int, scaling down
      by one.
   update rows and sum
   set row_end_valid = FALSE
   if lp = active, set globals.
   eta_valid = FALSE
   basis_valid = FALSE



void add_lag_con(lprec *lp, REAL *row, short con_type, REAL rhs)

   Calloc/Realloc space for lag_row, lag_rhs, lambda, lag_con_type
   Fill arrays.

void str_add_lag_con(lprec *lp, char *row, short con_type, REAL rhs)

   Same idea as always. Reserve space for array, strtod values into this array,
   call add_lag_con and free array.



void add_column(lprec *lp, REAL *column)

   update columns and sums,
   increment space for columns and matrix
   if scaling used
      set scaling factor for column to "1" and scale all values with row[scaling].
   for all elements in (dense) column
      if value is not zero
         write it in matrix.
   update col_end
          orig_lowbo
          orig_upbo
          lower
          basis
          must_be_int
          invent perhaps name for column
   row_end_valid = FALSE
   if lp = active
      set sum, columns, non_zeros


void str_add_column(lprec *lp, char *col_string)

   Same idea as always. Reserve space for array, strtod values into this array,
   call add_column and free array.



void del_column(lprec *lp, int column)

   check, if column is in range
   if column in Basis set basis_valid to FALSE
      else update bas
   shift names_used,
         must_be_int
         orig_upbo
```

```
          orig_lowbo
          upbo
          lowbo
          basis
          lower
          scaling
    update lagrangean stuff
    copy elements in matrix down.
    update col_end
    update non_zeros
    row_end_valid = FALSE
    eta_valid = FALSE
    update sum
           column
    if lp = active
       set_globals()



void set_upbo(lprec *lp, int column, REAL value)

    Test if column number in range
    scale value
    Test, if new value is feasible (greater than lower bound)
    eta_valid = FALSE
    set orig_upbo

void set_lowbo(lprec *lp, int column, REAL value)

    Test if column number in range
    scale value
    Test, if new value is feasible (smaller than upper bound)
    eta_valid = FALSE
    set orig_lowbo

void set_int(lprec *lp, int column, short must_be_int)

    Test if column number in range
    set must_be_int
    If variable must be integer, unscale column


void set_rh(lprec *lp, int row, REAL value)

    Test, if row_number is in range
    Test, if row_number for objective row should be set, WARNING
    scale value and change sign.
    eta_valid = FALSE


void set_rh_vec(lprec *lp, REAL *rh)

    For all rows
       scale and change sign
       set orig_rh
    eta_valid = FALSE


void str_set_rh_vec(lprec *lp, char *rh_string)

    Same idea as always. Reserve space for array, strtod values into this array,
    call set_rh_vec and free array.


void set_maxim(lprec *lp)

    if maxim == FALSE
       multiply all Values in row[0] with -1
       eta_valid = FALSE
    set maximise = TRUE
        ch_sign[0] = TRUE
    if LP = active, set Maximise = TRUE
```

```
void set_minim(lprec *lp)

   if maxim == TRUE
      multiply all Values in row[0] with -1
      eta_valid = FALSE
   set maximise = FALSE
       ch_sign[0] = FALSE
   if LP = active, set Maximise = FALSE



void set_constr_type(lprec *lp, int row, short con_type)

   Test, if row_number is in range
   if type == EQUAL
      set upper bound on slackvariable to zero
      basis_valid == FALSE
      if change_sign[row]
         multiply all coefficients with -1
         eta_valid = FALSE
         change_sign = FALSE
         change sign of orig_rh
   if type == LESSEQUAL
      set upper bound on slackvariable to infinity
      basis_valid == FALSE
      if change_sign[row]
         multiply all coefficients with -1
         eta_valid = FALSE
         change_sign = FALSE
         change sign of orig_rh
   if type == GREATEREQUAL
      set upper bound on slackvariable to infinity
      basis_valid == FALSE
      if NOT change_sign[row]
         multiply all coefficients with -1
         eta_valid = FALSE
         change_sign = TRUE
         change sign of orig_rh
   else
      error wrong constraint type



REAL mat_elm(lprec *lp, int row, int column)
   /* get value of matrix element in row and column */


   Test, if row_number is in range
   Test, if col_number is in range
   value = 0
   loop through column
   if value found
      unscale and change_sign
   return value


void get_row(lprec *lp, int row_nr, REAL *row)
   /* this is dense form */

   Test, if row_number is in range
   for all columns
      initialise value with 0
      for all entries in column
         if row found, write value
      unscale value
   if change_sign
      multiply with -1



void get_column(lprec *lp, int col_nr, REAL *column)
```

```
   Test, if column is in range.
   /* column is dense*/
   initialise columnarray with 0
   for all elements in this colum, copy to dense array
   unscale and change sign


void get_reduced_costs(lprec *lp, REAL *rc)

   Basis has to be valid
   set_globals
   if eta_valid = FALSE
      invert
   initialise array with 0
   set rc[0] = 1
   btran(rc)
   For all columns
      if variable not in basis AND upper bound > 0
         rc[column] = SUM (over all elements in Column) mat.value * rc[row]
   round all values


short is_feasible(lprec *lp, REAL *values)

   Unscale values and look, if they are between orig_lower and orig_upper bounds
   allocate space for a new rhs
   With this values calculate rhs
   check if rhs is lessequal than orig rhs for LE rows and equal to orig_rhs
      for EQ rows.


   short column_in_lp(lprec *lp, REAL *testcolumn)

   for all columns
      for all elements in column
         unscale value and change_sign
         check if difference smaller than epsilon
   return TRUE or FALSE




void print_lp(lprec *lp)

   print rowwise in readable form.


void set_row_name(lprec *lp, int row, nstring new_name)

   Perhaps allocate memory for names and initialise with default names
   strcpy rowname


void set_col_name(lprec *lp, int column, nstring new_name)

   Perhaps allocate memory for names and initialise with default names
   strcpy colname



static REAL minmax_to_scale(REAL min, REAL max)

   calculate scaling factor depending on min and max


void unscale_columns(lprec *lp)

   for all columns
      for all coefficients in column
         unscale (columnscaling)
   for all columns
      unscale bounds
   set scaling vector to 1
```

```
   columns_scaled = FALSE
   eta_valid = FALSE



void unscale(lprec *lp)

   Work only if scaling used
   for all columns
      for all coefficients in column
         unscale (columnscaling)
   for all columns
      unscale bounds
   for all columns
      for all coefficients in column
         unscale (rowscaling)
   for all rows
      unscale orig_rhs
   free scale
   scaling_used = FALSE
   eta_valid = FALSE




void auto_scale(lprec *lp)

   find row maximum and row minimum. Use these values to scale problem.

void reset_basis(lprec *lp)

   basis_valid=FALSE


void print_solution(lprec *lp)

   Print solution to stdout
   Print all variables
   In some cases
      Print slack variables ???
      Print duals



void write_LP(lprec *lp, FILE *output)

   print LP rowwise in readable form.


void write_MPS(lprec *lp, FILE *output)

   The routine write_MPS seems to do no unscaling. However it uses internally
   the routine get_column() which does unscaling!



void print_duals(lprec *lp)

Print all duals


void print_scales(lprec *lp)

   Print all row scales
   print all column scales.
```

```
/*************************************************************/
/**                                                         **/
/**                                                         **/
/**             Routines in file "read.c"                   **/
/**                                                         **/
/**                                                         **/
/*************************************************************/



void yyerror(char *string)
void check_decl(char *str)
static int hashval(const char *string)
static hashelem *gethash(char *variable)
void add_int_var(char *name)
void init_read(void)
static column *getrow(column *p,
                      int row)
static bound *create_bound_rec(void)
void null_tmp_store(void)
static void store(char *variable,
                  int row,
                  REAL value)

void store_re_op(void)
void rhs_store(REAL value)
void var_store(char *var, int row, REAL value)
void store_bounds(void)
void add_constraint_name(char *name, int row)
void readinput(lprec *lp)
lprec *read_lp_file(FILE *input, short verbose, nstring lp_name)


===================================================

To understand the idea of this file you should also read carefully the
comments directly at the beginning of this file!



void yyerror(char *string)
   Output error string and line number

void check_decl(char *str)
   We expect string "int". If this is not the case give error message.

static int hashval(const char *string)
   Calculate an integer hash value. (Modulo HASHSIZE).

static hashelem *gethash(char *variable)
   Returns a pointer to hashelement with name = variable.
   If this hashelement does not exist, gethash() returns a NULL pointer.

void add_int_var(char *name)
   Check if name exists. (if not, error message.)
   Check if it is the first time this variable was declared to be integer.
   Set flag for this variable to be integer.

void init_read(void)
   Init hashtable and globals.

static column *getrow(column *p,
                      int row)
   search in column-list (p is pointer to first element of column-list)
   for column->row = row.
   getrow() returns a pointer to this column structure.
   If not found a NULL-pointer is returned
   Follows one chain of pointers until correct element is found.

static bound *create_bound_rec(void)
   Creates a bound record.
```

```
   Calloc space.
   Set lowbo = 0 and upbo = Infinite
   Return pointer to this structure.

void null_tmp_store(void)
   clears the tmp_store variable after all information has been copied

static void store(char *variable,
                  int row,
                  REAL value)
   Store a value of the (sparse) matrix in data structure.
   If Value == 0, display warning.
   Three cases have to be distinguished:
   First: Variable does not exist
          Calloc space for info about variable
          update number of variables
          insert this element first into hashtable
          Calloc space for value
          insert rownumber and value into structure
   Second: Variable exists and has no value in that row yet
          Calloc space for value
          Insert rownumber and value into structure and insert into pointer
          chain.
   Third: Variable exists and has already a value in that row.
          add value to old value.

void store_re_op(void)
   switch yytext[0]
   case =
   case >
   case <
   default   error exit



void rhs_store(REAL value)
   Store RHS value in the rightside structure.
   Two cases are distinguished: Constraints with several variables have a right
   hand side, Constraints with only one variable are no constraints but bounds.

void var_store(char *var, int row, REAL value)
   Store all data in the right place.
   Distinguish between bound and constraint.

   error exit, if variable name is too long.
   update Lin_term_count carefully, because it could be a bound.
   If it is possible that constraint is only a bound, store its values in
   temporary space.
   If it is sure that it is NOT a bound, store values from temporary space
   first. Init temporary space with zero for further use.
   Store the values for the last read variable.

void store_bounds(void)
   The constraint was in fact a Bound. We store it now. The information for
   the variable can be found in temporary space.

   If value == 0: error exit.

   Check, if we know this variable already.
   If new variable found
      Calloc space
      update number of variables
      init space
      insert space into hashtable
   else
      Check if space for bounds exists already and if not, create.

   change perhaps direction of inequality, if negative coefficient in front of
   variable.

   Check, if bound is feasible. If not: error exit.
   Perhaps display warning, if upper bound is negative.
   Insert bound into structure, but check if there exists already stronger
```

```
   bound. In this case display warning.

   Check, if upperbound AND lower bound contradict each other. error exit.

   clear temporary space.

void add_constraint_name(char *name, int row)
   Store constraint name in structure. The first name has to be handled
   differently.

void readinput(lprec *lp)
   Transport the data from the  intermediate structure to the sparse matrix
   and free the intermediate structure. The routine tries not to waste memory
   and frees every data which is copied to the other structure.

   Copy all the given row names.
   For all Rows descending:
      store relational operator
      store rhs
      free memory
   For all equal rows change upper bound to zero.

   If some rows do not have names, generate a name.

   Read Hash structure   /* variables loose their original number */
      initialise col_end of the variable.
      Set must_be_int and the bounds of the variable.
      copy name of variable.
      put matrix values in sparse matrix.   /* No special sorting in a column */
         copy row index, value
         update number of nonzeros
         free space
   initialise col_end for last variable.

   if verbose
      print some statistic information
      print basically MPS file.

lprec *read_lp_file(FILE *input, short verbose, nstring lp_name)
   init some data
   call the parser yyparse()

   Calloc new lp structure and initialise lots of data (sizes of arrays etc.)
   Calloc arrays and insert into structure.

   Call readinput to get the information into the lp structure.

   check maximise
   set constraint type

   return pointer to lp.




/****************************************************************/
/**                                                            **/
/**                                                            **/
/**              Routines in file "readmps.c"                  **/
/**                                                            **/
/**                                                            **/
/****************************************************************/




int scan_line(char* line, char *field1, char *field2, char *field3,
              double *field4, char *field5, double *field6)
void addmpscolumn(void)
static int find_row(char *field)
static int find_var(char *field)
lprec *read_mps(FILE *input, short verbose)
```

```
==================================

This file reads in a MPS file. It describes MPS format in a comment at the
beginning.

int scan_line(char* line, char *field1, char *field2, char *field3,
              double *field4, char *field5, double *field6)
   input a MPS file line in "line", output the fields or 0/empty string in
   field?.
   Return value is number of read fields.
   For every field the following is done:
      - first check, if this field exists in inputstring
      - second copy this part of input "line" to a buffer
      - third use sscanf to read this field into "field?"
      - update number of items.

void addmpscolumn(void)
   This routine uses the global variable "Last_column" which is calloced and
   filled in read_mps.
   - add_column
   - set_col_name
   - set_int
   Reset Last_column to all entries = 0.

static int find_row(char *field)
   Given a name of a row in "field" return the index number of the row.
   This is done by cycling one time through the entries of the rows.
   (If name not found, there is considered the case "Unconstrained_rows_found".
    This means we found N-rows, which are ignored. Perhaps we are just looking
    for the name of an N-row which we ignored. Therefore we will not find its
    index.)
   Exit from program if name does not exist.

static int find_var(char *field)
   The routine is very similar to "find_row".
   Given name of a variable return the index or exit program if name does not
   exist.
   It is done by cycling one time through all column names.

lprec *read_mps(FILE *input, short verbose)
   This is the longest routine in this file. It does all the work for reading
   in a MPS file. It contains lot of if(Debug) statements to print out
   debugging information.


   Initialise an empty LP.
   Initialise various data.
   Start while loop to read one line after the other from the MPS file.
      First skip comments
      Check first character in line to determin if it is "special" line.
         NAME
            read problem name
         ROWS
            Set variable "section" to corresponding value.
         COLUMNS
            Set variable "section" to corresponding value.
         RHS
            addmpscolumn()
            Set variable "section" to corresponding value.
         BOUNDS
            Set variable "section" to corresponding value.
         RANGES
            Set variable "section" to corresponding value.
         ENDATA
            Do nothing.
         Error exit, if unknown Keyword.
      else (normal line)
         scan_line()
         switch to the correct section to use the fields.
         NAME: error
         ROWS: N-row: take first one as objective row, i.e. row[0].
                      Forget further N-rows. Set some Variables to  suppress
                      further error messages.
```

```
                L-row: str_add_constraint
                       set_row_name
                G-row: str_add_constraint
                       set_row_name
                E-row: str_add_constraint
                       set_row_name
                else: error exit
          COLUMNS:
                The line should have 4 or 6 fields!
                If line has 5 fields, it could be a MARKER row!
                If we receive a new column name, i.e. the name is different from
                Last_col_name AND Column_ready, we addmpscolumn() and also its
                name. Set Column_ready to TRUE.
                else
                copy field2 to Last_col_name
                Insert field 4 and perhaps field 6 into the correct position
                of array Last_column.
                If line has 5 fields:
                Check for 'MARKER' Keyword. Addmpscolumn().
                Check for 'INTORG' or 'INTEND'.
                Update variable "Int_section" depending on result.
                Ignore unknown markers. (Do not exit)
                If not 4, 5 or 6 fields: error exit.
          RHS:
                The line should have 4 or 6 fields!
                If not, error exit.
                Insert field4 and field6 into the correct position using set_rh.
          BOUNDS:
                No check for the number of fields in the line is done.
                The following Types are handled:
                UP: set upper bound
                LO: set lower bound
                FX: set upper bound
                    set lower bound
                PL: do nothing
                BV: set upper bound = 1
                    set integer = TRUE
                FR: split into two variables.
                    get_column, multiply with -1, add_column. Generate meaningful
                    name for column, add name. Nothing is done with current
                    lower and upper bounds.
                MI: change to positive variable by multiplying variable with -1.
                    i.e. get_column, del_column, multiply with -1, add_column,
                    generate column name, add name.
                else: error exit. Unsupported Type.
          RANGES:
                The line should have 4 or 6 fields!
                Error exit, if number of fields is wrong.
                Set bounds on row, i.e. writing the values directly into the
                array orig_upbo[].
    return Pointer to LP.
```

```
/*****************************************************************/
/**                                                             **/
/**                                                             **/
/**              File "lex.l"                                   **/
/**                                                             **/
/**                                                             **/
/*****************************************************************/
```

Of course lex.l is the descriptive file for lex. To understand it, you need some knowledge about lex. In this part you only find the different tokens that are described in lex.l and in some cases you also find a description how lex works on an input file in a correct language.

The input file can contain Comments "/* ... */" which are just ignored. White space (WS), i.e. Blanks, Tabulators and New line are also ignored from the input file.

```
COMMA:          ","
MINIMIZE:       "min" or "MIN" or "Min" or ...
```

```
MAXIMIZE:      "max" or ...........
CONS:          a constant number. Its value is returned in global variable "f".
SIGN:          basically "+" or "-". Variable "Sign" == TRUE, if "-".
VAR:           variable. Its name is returned in global variable "Last_var".
               Basically variable names start with a letter and than can contain
               letters, digits, brackets and special characters.
COLON:         ":"
AR_M_OP:       "*"
RE_OP:         "=", "<=" or ">="
END_C:         ";"

It also detects errors in input file.



/****************************************************************/
/**                                                            **/
/**                                                            **/
/**              File "lp.y"                                   **/
/**                                                            **/
/**                                                            **/
/****************************************************************/

Similar remarks are valid for lp.y than for lex.l. This is the descriptive
file for yacc. Therefore you need some knowledge about yacc to understand
it. It works closely together with lex. You will find a rough description
of the recognised language.

An input file consists of
- an objective_function
- constraints
- int_declarations

Constraints can be a single constraint or several constraints.

In front of a constraint there can be a name separated by a colon.

Basically a constraint consists of
- x_lineair_sum
- RE_OP  an Relational Operator
- x_lineair_sum
- ";"


int_declarations can be empty. They can consist of several int_declaration's.
Each int_declaration consists of
- an int_declarator
- vars, i.e. several variables, which can be comma separated.
- and finally ";"

Each x_lineair_sum consists of several x_lineair_term's with SIGN.
Each x_lineair_term is a constant or a lineair_term.
A lineair_term is a variable or a constant followed by a variable or
a constant "*" variable.

The objective function starts with "max" or "min" followed by a lineair_sum and
ends with ";".
A lineair_sum consists of lineair_term's with signs.

You will not find a more detailed description of the language. Best is to
look into an example to get an idea of the format.





/****************************************************************/
/**                                                            **/
/**                                                            **/
/**              Function calling tree                         **/
/**                                                            **/
/**                                                            **/
/****************************************************************/
```

The following lines have been produced using cflow. They give an overview of the calling structure of lp_solve. Refer to the cflow manual, if you have difficulties, to understand the output.

The functions are arranged in levels. All the functions a single routine calls are intended one level more.

The line numbers refer to version 2.0 of lp_solve.

SOLVE.C

```
1   lag_solve: int(), <solve.c 1516>
2       malloc: <>
3       exit: <>
4       fprintf: <>
5       calloc: <>
6       memcpy: <>
7       get_row: <>
8       set_mat: <>
9       print_lp: <>
10      solve: int(), <solve.c 1462>
11          set_globals: void(), <solve.c 16>
12          Isvalid: short(), <solve.c 105>
13              malloc: 2
14              exit: 3
15              fprintf: 4
16              free: <>
17              calloc: 5
18          milpsolve: int(), <solve.c 1158>
19              debug_print: <>
20              memcpy: 6
21              rand: <>
22              invert: void(), <solve.c 320>
23                  fprintf: 4
24                  calloc: 5
25                  exit: 3
26                  minoriteration: void(), <solve.c 244>
27                  setpivcol: void(), <solve.c 215>
28                      ftran: void(), <solve.c 66>
29                  error: <>
30                  condensecol: void(), <solve.c 174>
31                      resize_eta: void(), <solve.c 162>
32                          realloc: <>
33                          exit: 3
34                          fprintf: 4
35                  rhsmincol: void(), <solve.c 292>
36                      fprintf: 4
37                      exit: 3
38                  addetacol: void(), <solve.c 197>
39                  free: 16
40              solvelp: int(), <solve.c 893>
41                  calloc: 5
42                  exit: 3
43                  fprintf: 4
44                  colprim: short(), <solve.c 502>
45                      btran: void(), <solve.c 87>
46                      fprintf: 4
47                  setpivcol: 27
48                  rowprim: short(), <solve.c 564>
49                      fprintf: 4
50                  condensecol: 30
51                  rowdual: short(), <solve.c 655>
52                      fprintf: 4
53                  coldual: short(), <solve.c 712>
54                      fprintf: 4
55                  iteration: void(), <solve.c 815>
56                      addetacol: 38
57                      fprintf: 4
58                  invert: 22
59                  free: 16
```

```
60              fprintf: 4
61              construct_solution: void(), <solve.c 1057>
62                  memset: <>
63              debug_print_solution: <>
64              is_int: short(), <solve.c 1044>
65                  floor: <>
66              malloc: 2
67              exit: 3
68              debug_print_bounds: <>
69              ceil: <>
70              milpsolve: 18
71              free: 16
72              calculate_duals: void(), <solve.c 1129>
73                  btran: 45
74              print_solution: <>
75          print_solution: 74
76          free: 16
```

READMPS.C

```
1   read_mps: struct*(), <readmps.c 227>
2       make_lp: <>
3       strcpy: <>
4       fgets: <>
5       fprintf: <>
6       sscanf: <>
7       strcmp: <>
8       calloc: <>
9       exit: <>
10      addmpscolumn: void(), <readmps.c 170>
11          add_column: <>
12          set_col_name: <>
13          set_int: <>
14      scan_line: int(), <readmps.c 101>
15          strlen: <>
16          strncpy: <>
17          sscanf: 6
18      set_row_name: <>
19      str_add_constraint: <>
20      find_row: int(), <readmps.c 188>
21          strcmp: 7
22          fprintf: 5
23          exit: 9
24      set_rh: <>
25      find_var: int(), <readmps.c 210>
26          strcmp: 7
27          fprintf: 5
28          exit: 9
29      set_upbo: <>
30      set_lowbo: <>
31      set_int: 13
32      get_column: <>
33      add_column: 11
34      strcat: <>
35      set_col_name: 12
36      del_column: <>
```

DEBUG.C

```
1   debug_print_solution: void(), <debug.c 21>
2       print_indent: void(), <debug.c 7>
3           fprintf: <>
4       fprintf: 3
5   debug_print_bounds: void(), <debug.c 41>
6       print_indent: 2
7       fprintf: 3
8   debug_print: void(), <debug.c 72>
9       print_indent: 2
```

```
10      vfprintf: <>
11      fputc: <>









/****************************************************************/
/**                                                            **/
/**                                                            **/
/**             References                                     **/
/**                                                            **/
/**                                                            **/
/****************************************************************/





Literatur:
           John Lawrence Nazareth,
                     Computer Solution of Linear Programs,
                     Oxford University Press 1987

           Orchard-Hays, W.
                     Advanced linear-programming computing techniques,
                     McGraw-Hill, 1968

           Chvatal, V.
                     Linear Programming,
                     W.H. Freeman and Company,
                     New York and San Francisco 1983.

           Nemhauser, G.L. and Wolsey, Laurence A.,
                     Integer and Combinatorial Optimization,
                     John Wiley & Sons, Inc. 1988
```