

```

1  /*****
2  /**
3  /**      Documentation for lp_solve
4  /**
5  /**      (C) Hartmut Schwab      Hartmut.Schwab@IWR.Uni-Heidelberg.De
6  /**
7  /**
8  /**
9  /*****/
10
11
12
13  This documentation is Copyright 1996 by Hartmut Schwab. It may be freely
14  redistributed in its entirety provided that this copyright notice and the
15  disclaimer is not removed. It may not be sold for profit or incorporated in
16  commercial documents without the written permission of the copyright holder.
17  Permission is expressly granted for this document to be made available for
18  file transfer from installations offering unrestricted anonymous file
19  transfer on the Internet. Its intent is to promote widespread usage.
20
21      Notice must be given of the location of the
22      availability of the unmodified current source code
23      of lp_solve
24      e.g.,
25          ftp://ftp.es.ele.tue.nl/pub/lp\_solve
26
27
28
29  While all information in this documentation is believed to be correct at the
30  time of writing, it is provided "as is" with no warranty implied.
31
32
33
34  In compiling this information, I have drawn on my own knowledge of the
35  field. This file has been read by Michel Berkelaar, the author of lp_solve but
36  nevertheless the errors in this document came into it by me. I give my
37  thanks to all those who have offered advice and support.
38
39  Suggestions, corrections, topics you'd like to see covered, and additional
40  material, are all solicited. Send email to
41
42      Hartmut.Schwab@IWR.Uni-Heidelberg.De
43
44
45
46  The newest version of this document is available at the same location than
47  lp_solve.
48
49
50
51  Version: @(#) lp_solve.doc 1.18@(#), Creation date: 97/02/11.
52
53
54
55
56
57
58
59  /*****/
60  /**
61  /**
62  /**      Preface
63  /**
64  /**
65  /**
66  /*****/
67
68  This documentation would not exist, if Michel Berkelaar had not written the
69  program lp_solve. Thanks to him, we have a version of the simplex algorithm,
70  where the source code is available, which uses sparse matrix computations
71  and which everybody can include into his or her own applications. More and
72  more users in the Operations Research community are using lp_solve.
73
74  His program is finding more and more users in the OR community. The growing

```

```

74 interest is easily proven by the still growing number of questions regarding
75 lp_solve in the News. Questions where to find a source code for the simplex
76 algorithm are usually answered with a hint to lp_solve.
77
78 It is also important to mention Jeroen Dirks who added a subroutine library
79 to lp_solve. His work makes it easier to include lp_solve in own
80 applications.
81
82 Until now, no documentation about the construction of the program had been
83 available. This file wants to close this gap. It wants to provide the reader
84 with some information about the internal structure of the program and add
85 some documentation to lp_solve.
86
87
88 How is this document organised?
89
90 You will find a table of contents in the next section. Section
91 "Introduction" describes the intention of this document. You also will find
92 some information, what has been included into this document and what has
93 been excluded from it. The later one is the more important one. This document
94 does not contain every information and it doesn't want to contain every
95 information.
96
97 A very important section is "Datastructures used internally by lp_solve".
98 You can guess, what you will find there. If you want to do some extensions,
99 if you detect some errors you probably have to look there to understand the
100 way lp_solve organises all the information internally.
101
102 The following sections contain a short description of the functions, but
103 only the MAIN IDEAS are written down. These sections are organised in the
104 same way as the source files. Each source file has its own section.
105
106 For easier understanding the structure of the program, I also included the
107 "Function calling tree". It gives an idea how the different functions work
108 together.
109
110 If you want to get more information, which is not covered in this document,
111 probably the hints you find in the References will be a good starting point.
112
113
114
115
116 This file has been prepared carefully. Nevertheless it probably will contain
117 errors. The author is in no way responsible for any damage caused by using
118 this file.
119
120
121
122 This document describes version 2.0 of lp_solve. However now major changes
123 are expected to occur in the next time. Therefore this document should be
124 valid also for newer versions of lp_solve.
125
126
127
128
129 /*****
130 /**
131 /**
132 /**          Contents
133 /**
134 /**
135 /*****/
136
137
138 Contents:
139 =====
140
141 - Copyright and distribution
142
143 - Preface
144
145 - Contents
146

```

```

147 - Introduction
148
149 - Datastructures used internally by lp_solve
150
151 - Short description of the functions, the MAIN IDEAS are written down.
152
153 - Function calling tree
154
155 - References
156
157
158
159
160 /*****/
161 /**/
162 /**/
163 /**/      Introduction      /**/
164 /**/      /**/
165 /**/      /**/
166 /*****/
167
168
169 Purpose:
170 =====
171
172     This documentation should help the readers to get a better
173     understanding of the source code of lp_solve and its main ideas,
174     to be able to make changes in the code and to locate errors in the code,
175     if there appear any.
176     It should also give the chance to make improvements to the code
177     i.e. other product form or different Branch and Bound strategy.
178
179
180
181     This documentation does NOT describe the simplex algorithm. You can find a
182     description of the simplex algorithm in every book about linear programming.
183     If you are interested to find a description which is more adapted to the
184     sparse form of the simplex, check the literature given in the references.
185
186     Some keywords, which describe the implementation of lp_solve:
187
188     - selecting pivot variable with largest reduced costs.
189       No devex or steepest edge.
190
191     - inverting the basis matrix is done in pure product form. No LU
192       decomposition.
193
194     - Branch and Bound implemented as recursive function, this means pure
195       Depth First. There are two strategies for selecting a branching variable.
196       Branching on the first non integer variable, branching on a random
197       selected variable.
198
199
200     The result is a relatively small and easy to grasp code.
201     On the other side it can run into numerical problems. More expenditure
202     has to be done to make the code numerical stable and fast on large
203     problems. Perhaps somebody wants to add some parts to improve the program.
204     However this always should be done in a modular way.
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219 /*****/

```

```

220  /**                                                    **/
221  /**                                                    **/
222  /**            DATASTRUCTURES                        **/
223  /**                                                    **/
224  /**                                                    **/
225  /*******/
226
227
228  This section describes the most important data structures of
229  the program. For arrays I tried to give the size of the array
230  and give an idea of the contents of miscellaneous fields, if there
231  are different parts in the array.
232  The first part of this section should give the main idea, how a
233  linear programming problem is transferred into the internal data
234  structures.
235  Then a more detailed description of the separate arrays follows.
236
237
238
239  How is the following linear program transferred to the internal data?
240
241  max    c'x
242  s.t.   A x <= b
243         lower <= x <= upper
244
245
246
247
248  All data is placed into arrays which can be accessed via pointers in the
249  structure lprec. There you can find information to the matrix, information,
250  how to interpret the data and the inverted matrix (Eta file).
251
252
253  =====
254
255  Something about numbering rows and columns:
256
257  The program is written in C, however indices for the data normally start
258  with 1!
259
260  Columns are numbered from 1 to columns.
261  Rows are numbered from 1 to rows.
262  row[0] is objective row.
263
264
265  Maximisation or minimisation is only a change of sign in objective row.
266  This change is marked in ch_sign[0] and in lp->maximise.
267
268
269  =====
270
271
272  Internally to lp_solve there exist only EQUALITIES or LESSEQUAL rows.
273  GREATEREQUAL rows are multiplied with -1 and the information, that
274  this is done is written in the array ch_sign.
275
276
277  How to access the right hand side?
278  Read lp->orig_rhs.
279
280
281  How to access the bounds?
282  Read lp->orig_lowbo and lp->orig_upbo
283
284
285  How to know the sense of a constraint?
286  It can be accessed via the information on the bounds of a row. Internally
287  there exist only Less-equal and Equal rows. Upper bound == 0 means row is
288  equal row. Everything else means: It is a Less-Equal row. To distinguish
289  between Less-Equal rows and Greater-Equal rows of the original problem you
290  have to check ch_sign.
291
292

```

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 */
```

```

366
367 int non_zeros; /* The number of elements in the sparse matrix*/
368 int mat_alloc; /* The allocated size for matrix sized
369 structures */
370 matrec *mat; /* mat_alloc :The sparse matrix */
371 int *col_end; /* columns_alloc+1 :Cend[i] is the index of the
372 first element after column i.
373 column[i] is stored in elements
374 col_end[i-1] to col_end[i]-1 */
375 int *col_no; /* mat_alloc :From Row 1 on, col_no contains the
376 column nr. of the
377 nonzero elements, row by row */
378 short row_end_valid; /* true if row_end & col_no are valid */
379 int *row_end; /* rows_alloc+1 :row_end[i] is the index of the
380 first element in Colno after row i */
381 REAL *orig_rh; /* rows_alloc+1 :The RHS after scaling & sign
382 changing, but before 'Bound transformation' */
383 REAL *rh; /* rows_alloc+1 :As orig_rh, but after Bound
384 transformation */
385 REAL *rhs; /* rows_alloc+1 :The RHS of the current simplex
386 tableau */
387 short *must_be_int; /* sum_alloc+1 :TRUE if variable must be
388 Integer */
389 REAL *orig_upbo; /* sum_alloc+1 :Bound before transformations */
390 REAL *orig_lowbo; /* " " */
391 REAL *upbo; /* " " :Upper bound after transformation
392 & B&B work*/
393 REAL *lowbo; /* " " :Lower bound after transformation
394 & B&B work */
395
396 short basis_valid; /* TRUE if the basis is still valid */
397 int *bas; /* rows_alloc+1 :The basis column list */
398 short *basis; /* sum_alloc+1 : basis[i] is TRUE if the column
399 is in the basis */
400 short *lower; /* " " :TRUE if the variable is at its
401 lower bound (or in the basis), it is FALSE
402 if the variable is at its upper bound */
403
404

```

The following

max  $c'x$

s.t.  $Ax \leq b$   
 $l \leq x \leq u$

symbolically:

```

415 +-----+
416 |           c           |
417 +-----+
418
419 +-----+ +--+
420 |           A           | <= |b|
421 |           |           |
422 |           |           |
423 |           |           |
424 +-----+ +--+
425
426
427 +-----+
428 |   upper Bound   |
429 +-----+
430
431 +-----+
432 |   lower Bound   |
433 +-----+
434
435
436

```

is transformed to

437  
438

```

439
440
441      +-----+      +-+      +-+      +-+  \
442      |          c          |  |  |  |  |  |      := row[0]
443      +-----+      +-+      +-+      +-+  |
444      |          A          |  |  |  |  |  |  \ rows_alloc+1
445      |          |          |  |  |  |  |  |  /
446      |          |          |  |  |  |  |  |
447      |          |          |  |  |  |  |  |
448      +-----+      +-+      +-+      +-+  /
449
450                                orig_rh      rh      rhs
451                                scaled +      =orig_rh current rhs
452                                signchange + Bound dh. Basis values(?)
453                                transform
454
455 sum_alloc = rows_alloc + columns_alloc;
456 Be careful: There is a difference between "rows" and "rows_alloc". We
457 allocate "rows_alloc" elements, but use only "rows" elements. This means,
458 the space between "rows" and "rows_alloc" is empty/not used.
459
460 The same is true for "columns" and "columns_alloc".
461 The pair for matrix is called "non_zeros" and "mat_alloc".
462
463
464
465 rows
466 |
467 rows_alloc+1          sum_alloc+1. Index [0] seems to be unused.
468 | |
469 slack v          v
470 +-----+
471 | |          must_be_int |
472 +-----+
473
474 slack
475 +-----+
476 | |          orig_upbo | before Bound transform
477 +-----+
478
479 slack
480 +-----+
481 | |          orig_lowbo | before Bound transform
482 +-----+
483
484 slack
485 +-----+
486 | |          upbo | after Bound transform and in B+B
487 +-----+
488
489 slack
490 +-----+
491 | |          lowbo | after Bound transform and in B+B
492 +-----+
493
494
495 slack
496 +-----+
497 | |          Basis | TRUE, if column is in Basis.
498 +-----+          FALSE otherwise.
499
500 slack
501 +-----+
502 | |          lower | TRUE, if column is in Basis or
503 +-----+          nonbasic and at lower bound.
504 ^          FALSE otherwise.
505 |
506 0
507
508
509 +-+
510 | |
511 +-+

```

```

512 | |
513 | | indices of columns which are in Basis.
514 | |
515 | |
516 +-+
517 bas
518
519
520 The matrix is stored in sparse form in the usual way.
521
522
523 int non_zeros; /* The number of elements in the sparse matrix*/
524 int mat_alloc; /* The allocated size for matrix sized
525 structures */
526 matrec *mat; /* mat_alloc :The sparse matrix */
527 int *col_end; /* columns_alloc+1 :Cend[i] is the index of the
528 first element after column i.
529 column[i] is stored in elements
530 col_end[i-1] to col_end[i]-1 */
531 int *col_no; /* mat_alloc :From Row 1 on, col_no contains the
532 column nr. of the
533 nonzero elements, row by row */
534 short row_end_valid; /* true if row_end & col_no are valid */
535 int *row_end; /* rows_alloc+1 :row_end[i] is the index of the
536 first element in Colno after row i */
537
538 +-----+-----+-----+-----+-----+-----+-----+
539 | 1 | 3 | 7 | 1 | *** | *** | *** | (row_nr)
540 +-----+-----+-----+-----+-----+-----+ mat
541 | 2.5 | 4.7 | 1.0 | 2.0 | *** | *** | *** | (value)
542 +-----+-----+-----+-----+-----+-----+
543 | | | | | ^ | ^ |
544 | | | | | | |
545 | | | | | non_zeros mat_alloc
546
547 Entry Zero is valid.
548
549
550
551
552 +-----+-----+-----+-----+
553 | *** | 3 | 4 | | col_end (in fact beginning of next column)
554 +-----+-----+-----+-----+
555 | | | | ^ | ^ |
556 | | | | | |
557 | | | | columns columns_alloc+1
558
559 Entry Zero is NOT valid.(?)
560
561
562
563 +-----+-----+-----+-----+-----+-----+
564 | 1 | 2 | 5 | 1 | *** | *** | *** | col_no: Which columns appear in
565 +-----+-----+-----+-----+-----+-----+ row[i]. Row[i] starts at
566 | | | | | ^ | ^ | row_end[i-1] and ends at
567 | | | | | non_zeros mat_alloc row_end[i] - 1.
568
569
570 ATTENTION: Documentation in header file seems to be wrong!!!
571 col_no[0] is not used! row[i] starts in row_end[i-1]+1 and ends in row_end[i].
572 In array there are used (non_zero - number of coefficients in objective row +1)
573 elements used.
574
575 Col_no is used in invert. (And nowhere else, but set in IsValid)
576
577 +-+
578 | |
579 +-+
580 | |
581 | | How many coefficients are in rows 1 to i. Equivalent:
582 | | row_end[i] is the index of the first element in col_no after row i.
583 | |
584 +-+

```



```

585         row_end
586
587
588
589 How is sense of the constraints/rows coded?
590
591 Look at slack variable of the row. If the orig_upbo[i] < infinite (this
592 should be: orig_upbo[i] == 0) then we have an equality row. This comparison
593 can be found in write_MPS(), where all Rows with upper bound == infinity are
594 "L" or "G" rows. All other rows are "E" rows. See also write_LP().
595
596 In the other cases, that means orig_upbo[[i] == infinite, we have to look
597 at ch_sign[i]. If ch_sign[i] == TRUE, we have a greater equal row, if
598 ch_sign == FALSE, we have a less equal row.
599
600 =====
601
602
603 short      eta_valid;          /* TRUE if current Eta structures are valid */
604 int        eta_alloc;          /* The allocated memory for Eta */
605 int        eta_size;           /* The number of Eta columns */
606 int        num_inv;            /* The number of real pivots */
607 int        max_num_inv;        /* ## The number of real pivots between
608                                reinversions */
609 REAL       *eta_value;         /* eta_alloc :The Structure containing the
610                                values of Eta */
611 int        *eta_row_nr;        /* " " :The Structure containing the Row
612                                indexes of Eta */
613 int        *eta_col_end;       /* rows_alloc + MaxNumInv : eta_col_end[i] is
614                                the start index of the next Eta column */
615
616 +-----+-----+-----+-----+-----+-----+-----+-----+
617 |          |          eta_col_end          | Startindex of next Eta column
618 +-----+-----+-----+-----+-----+-----+-----+
619 rows_alloc      max_num_inv      ^
620 this is needed  we can have      eta_size
621 for first      maximal so many
622 invert         inverts, i.e. etamatrices.
623                until next inversion.
624
625
626 +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
627 | | | | | | | | | | | | | | | | eta_value
628 +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
629
630 +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
631 | | | | | | | | | | | | | | | | eta_row_nr
632 +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
633                                     ^
634                                     |
635                                     eta_alloc
636
637
638 Normal way of sparse matrix representation. But how to built one Eta
639 Matrix? I do not know, in which column to put eta-column.
640
641 Guess:
642 This is one eta_matrix. Last entry contains index of the column and value
643 on the diagonal.
644 +---+---+---+---+---+---+
645 |1.3|1.5 |1.8 |1.7 |2.5| eta_value
646 +---+---+---+---+---+---+
647
648 +---+---+---+---+---+---+
649 | 1 | 3 | 7 | 8 | 6 | eta_row_nr
650 +---+---+---+---+---+---+
651
652
653 The matrix in dense form would be:
654
655 +---+---+---+---+---+---+
656 | 1 |   |   |   |   |1.3|   |   |
657 +---+---+---+---+---+---+

```

658			1							
659	+	---	+	---	+	---	+	---	+	---
660				1				.5		
661	+	---	+	---	+	---	+	---	+	---
662					1					
663	+	---	+	---	+	---	+	---	+	---
664						1				
665	+	---	+	---	+	---	+	---	+	---
666							2.5			
667	+	---	+	---	+	---	+	---	+	---
668							.8	1		
669	+	---	+	---	+	---	+	---	+	---
670							.7		1	
671	+	---	+	---	+	---	+	---	+	---
672							^			
673										
674							column 6			

```

675
676
677
678
679
680
681 short      bb_rule;          /* what rule for selecting B&B variables */
682
683 short      break_at_int;      /* TRUE if stop at first integer better than
684                               break_value */
685 REAL       break_value;
686
687 REAL       obj_bound;         /* ## Objective function bound for speedup of
688                               B&B */
689 int        iter;              /* The number of iterations in the simplex
690                               solver (LP) */
691 int        total_iter;        /* The total number of iterations (B&B) (ILP)*/
692 int        max_level;         /* The Deepest B&B level of the last solution */
693 int        total_nodes;       /* total number of nodes processed in b&b */
694
695
696
697
698
699
700
701
702 REAL       *solution;         /* sum_alloc+1 :The Solution of the last LP,
703                               0 = The Optimal Value,
704                               1..rows The Slacks,
705                               rows+1..sum The Variables */
706 REAL       *best_solution;     /* " " :The Best 'Integer' Solution */
707 REAL       *duals;            /* rows_alloc+1 :The dual variables of the
708                               last LP */
709
710
711                               sum_alloc+1. Index [0] is optimal solution value.
712                               |
713 slack                               v
714 +-----+-----+-----+-----+
715 |      |               solution      |
716 +-----+-----+-----+-----+
717 ^      ^                               ^
718 |      |                               |
719 |  rows                               sum=rows+columns
720 Optimal
721 value
722 = Index 0
723
724
725 slack
726 +-----+-----+-----+-----+
727 |      |               best_solution      | Best integer solution so far.
728 +-----+-----+-----+-----+
729
730 +-+
```

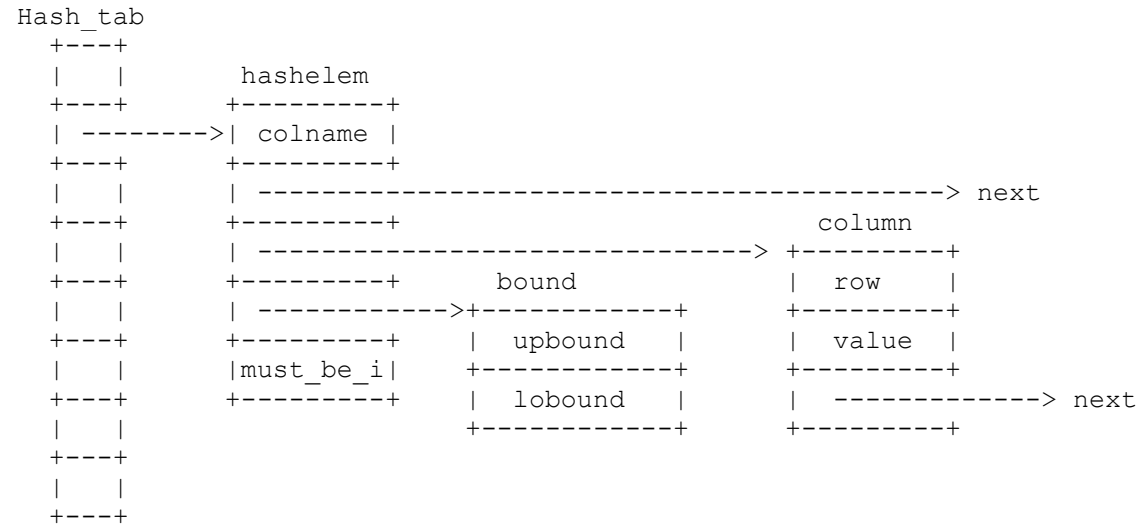
```

731      | |
732      +--+
733      | |
734      | |    dual variables
735      | |
736      | |
737      +--+
738      duals
739
740
741
742
743
744
745
746      short      maximise;          /* TRUE if the goal is to maximise the
747                      objective function */
748      short      floor_first;        /* TRUE if B&B does floor bound first */
749      short      *ch_sign;            /* rows_alloc+1 :TRUE if the Row in the matrix
750                      has changed sign
751                      (a`x > b, x>=0) is translated to
752                      s + -a`x = -b with x>=0, s>=0) */
753
754
755
756
757      +--+
758      | |
759      +--+
760      | |
761      | |    TRUE or FALSE
762      | |
763      | |
764      +--+
765      ch_sign      (compare sense of a row described above)
766
767      short      scaling_used;        /* TRUE if scaling is used */
768      short      columns_scaled;      /* TRUE is the columns are scaled too, Only use
769                      if all variables are non-integer */
770      REAL      *scale;               /* sum_alloc+1 :0..Rows the scaling of the Rows,
771                      Rows+1..Sum the scaling of the columns */
772
773
774
775                      sum_alloc+1
776                      |
777      rows          columns          v
778      +---+-----+
779      |   |                               | scale: Scaling factors for rows and columns
780      +---+-----+
781      ^   ^                               ^
782      |   |                               |
783      0 rows                          sum=rows+columns
784
785
786
787
788
789      int      nr_lagrange;           /* Nr. of Langrangian relaxation constraints */
790      REAL      **lag_row;            /* NumLagrange, columns+1:Pointer to pointer of
791                      rows */
792      REAL      *lag_rhs;             /* NumLagrange :Pointer to pointer of Rhs */
793      REAL      *lambda;              /* NumLagrange :Lambda Values */
794      short      *lag_con_type;        /* NumLagrange :TRUE if constraint type EQ */
795      REAL      lag_bound;            /* the lagrangian lower bound */
796
797      short      valid;               /* Has this lp passed the 'test' */
798      REAL      infinite;             /* ## numerical stuff */
799      REAL      epsilon;              /* ## */
800      REAL      epsb;                 /* ## */
801      REAL      epsd;                 /* ## */
802      REAL      epsel;                /* ## */
803  } lprec;

```

804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876

The HASH structure:  
=====

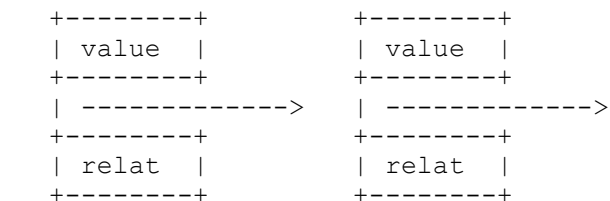


```
typedef struct _hashelem
{
    nstring      colname;
    struct _hashelem *next;
    struct _column *col;
    struct _bound  *bnd;
    int          must_be_int;
} hashelem;
```

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

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

First\_rside



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

```

877 {
878     REAL          value;
879     struct _rside *next;
880     short         relat;
881 } rside;
882
883
884
885
886
887     First_constraint_name
888
889     +-----+          +-----+
890     | name   |          | name   |
891     +-----+          +-----+
892     | row    |          | row    |
893     +-----+          +-----+
894     | ----->        | ----->
895     +-----+          +-----+
896
897
898 typedef struct _constraint_name
899 {
900     char          name[NAMELEN];
901     int           row;
902     struct _constraint_name *next;
903 } constraint_name;
904
905
906
907
908
909
910 typedef struct _tmp_store_struct
911 {
912     nstring name;
913     int      row;
914     REAL     value;
915     REAL     rhs_value;
916     short    relat;
917 } tmp_store_struct;
918
919
920
921
922
923
924
925
926 /******
927 /**
928 /**
929 /**          Routines in file "solve.c"
930 /**
931 /**
932 /******
933
934
935
936
937
938 This file contains the routines which are important to use the
939 SIMPLEX algorithm. For example you find routines to do basis
940 exchange, to select new pivot element etc. in this file.
941
942
943 set_globals()    Copy/initialize the global variables for a special LP.
944
945 ftran (int start,
946        int end,
947        REAL *pcol)          /* The column which will be used in ftran */
948        /* multiply the column with all matrices in etafile */
949

```

```

950     for every matrix between start and end
951         calculate End of the matrix
952         r = number of column in Eta matrix
953         theta = pcol[r]
954         for one matrix
955             multiply pcol with the matrix (?)
956         update pcol[r]
957     round values in pcol
958
959 btran (REAL *row)
960     For all Eta matrices, Starting with the highest number
961     k = number of column in Eta-matrix
962     for one matrix
963         do multiplication
964     round result
965     set row[Eta_row_nr[k]] = result
966
967
968 Isvalid (lprec *lp)
969     Calculate the structures row_end[] and
970     col_no[].
971     internally two arrays are used:
972     row_nr[], which contains the number of coefficients
973     in row[i] and num[], which is a working array and
974     contains the already used part of col_no in row[i].
975     The array col_no is written at several positions
976     at the same time. So it could look like
977
978     +-----+
979     |**          ****          *      **      |
980     +-----+
981
982     The second part of the routine uses two arrays
983     rownum[] and colnum[]. It tests, if there are some
984     empty columns in the matrix and prints a
985     Warning message in this case.
986
987     In detail:
988     if (!lp->row_end_valid)
989         malloc space for arrays num and rownum.
990         initialise with zero
991         count in rownum[i] how many coefficients are in row i
992         set row_end (ATTENTION: documentation of row_end
993             seems to be wrong. row_end points to LAST
994             coefficient in row. But this is never used??)
995         col_no[0] is not used!!!
996         loop through all the columns,
997         forget row[0] = objective row
998         write column index in array col_no.
999         free num, rownum
1000         row_end_valid = TRUE
1001     if (!lp->valid)
1002         Calloc rownum, colnum.
1003         for all columns
1004             colnum[i]++, for every coefficient in column.
1005
1006         if colnum[i] = 0, print warning.
1007
1008
1009 resize_eta()          simple REALLOC
1010
1011
1012
1013
1014 condens ecol(int row_nr, REAL *pcol)
1015     if necessary:
1016         resize_eta()
1017     For all rows
1018         if i <> row_nr && pcol[i] <> 0
1019             Eta_row_nr = i
1020             Eta_value = pcol[i]
1021             elnr++
1022

```

```

1023      /* Last Action: write element for diagonal */
1024      Eta_row_nr = row_nr
1025      Eta_value = pcol[row_nr]
1026
1027      update Eta_col_end
1028
1029
1030  addetacol()
1031      Determine Begin and End of last Eta matrix.
1032      calculate theta = 1/ Eta_value[Diagonal element]
1033      multiply all coefficients in last matrix with -theta
1034      JustInverted = FALSE
1035
1036
1037
1038
1039  setpivcol(short lower, int varin, REAL *pcol)
1040  /* Main idea: take one column from original matrix and call ftran */
1041
1042      /* init */
1043      ...
1044      pcol[i] = 0   for all i
1045
1046      If variable      /* This means not surplus/slack variable */
1047          copy column coefficients into pcol[]
1048          Actualise pcol[0] -= Extrad*f
1049      else              /* surplus/slack variable */
1050          /* This column is column from identity matrix */
1051          pcol[varin] = 1 or -1
1052
1053      ftran(1, Etasize, pcol)
1054
1055
1056
1057  minoriteration(colnr, row_nr)
1058      set varin
1059      elnr = wk = Eta_col_end[Eta_size]  /* next free element */
1060      Eta_size++
1061      /* Eta_size = number of matrices in Eta file */
1062      /* Eta_col_end = End of one matrix */
1063
1064      Do something if Extrad <> 0          /* I do not know what to do
1065                                          and what is Extrad */
1066
1067      For all coefficients in column
1068          set row_nr
1069          If Objective_row and Extrad
1070              Eta_value[Eta_col_end[Eta_size -1]] += Mat[j].value
1071          else if k <> row_nr
1072              Eta_row_nr[elnr] = k
1073              Eta_value[elnr] = Mat[j].value
1074              elnr++
1075          else
1076              piv = Mat[j].value
1077
1078      /* Last action: Write element on diagonal */
1079      insert row_nr and 1/piv
1080
1081      theta = rhs[row_nr] / piv
1082      Rhs[row_nr] = theta
1083
1084      For all coefficients of last eta matrix without diagonal element
1085          Rhs[Eta_row_nr[i]] -= theta*Eta_value[i]
1086
1087      /* set administration data for Basis */
1088      varout =
1089      Bas[row_nr] = varin
1090      Basis[varout] = FALSE
1091      Basis[varin] = TRUE
1092
1093      For all coefficients of last eta matrix without diagonal element
1094          Eta_value[i] /= -piv
1095
1096      /* update Eta_col_end */

```

```

1096     Eta_col_end[Eta_size] = elnr
1097
1098
1099
1100 rhsmincol(REAL theta, int row_nr, int varin)
1101     Error test
1102     Find for last matrix in etafile the begin and end
1103     for all coefficients in last eta matrix without diagonal coefficient
1104         calculate rhs[eta_row_nr] -= theta * etavalue[i]
1105     rhs[row_nr] = theta
1106
1107     varout = bas[row_nr]
1108     Bas[row_nr] = varin
1109     Basis[varout] = FALSE
1110     Basis[varin] = TRUE
1111
1112
1113
1114 invert()
1115     allocate
1116
1117     +---+---+---+---+---+---+
1118     | 0 |   |   |   |   |   |   | rownum
1119     +---+---+---+---+---+---+
1120                                     ^
1121                                     |
1122                                     Rows+1
1123
1124     +---+---+---+---+---+---+
1125     |   |   |   |   |   |   |   | col
1126     +---+---+---+---+---+---+
1127
1128     +---+---+---+---+---+---+
1129     |   |   |   |   |   |   |   | row
1130     +---+---+---+---+---+---+
1131
1132     +---+---+---+---+---+---+
1133     |   |   |   |   |   |   |   | pcol
1134     +---+---+---+---+---+---+
1135
1136     +---+---+---+---+---+---+
1137     |TRU|   |   |   |   |   |   | frow
1138     +---+---+---+---+---+---+
1139
1140     +---+---+---+---+---+---+
1141     |FAL|   |   |   |   |   |   |   | fcol
1142     +---+---+---+---+---+---+
1143                                     ^
1144                                     |
1145                                     columns+1
1146
1147     +---+---+---+---+---+---+
1148     | 0 |   |   |   |   |   |   |   | colnum
1149     +---+---+---+---+---+---+
1150                                     /* count number of
1151                                     Coefficients which
1152                                     appear in basis matrix */
1153
1154 change frow and fcol depending on Bas[i]
1155     frow = FALSE if Bas[i] <= Rows
1156     fcol = TRUE  if Bas[i] > Rows
1157
1158 set
1159     Bas[i] = i    for all i
1160     Basis[i] = TRUE    for all slack variables
1161                     FALSE for all other variables
1162     Rhs[i] = Rh[i]  for all i    /* initialise original Rhs */
1163
1164 Correct Rhs for all Variables on upper bound
1165 Correct Rhs for all slack variables on upper bound (if necessary)
1166
1167 Etasize =0
1168 v = 0
1169 row_nr = 0

```



```

1169 Num_inv = 0
1170 numit = 0
1171
1172 Look for rows with only one Coefficient (while)
1173 if found
1174 look for column of coefficient
1175 set fcol[colnr - 1] = FALSE /* This col is no longer
1176                               in Basis */
1177 colnum[colnr] = 0
1178 correct rownum counter
1179 set frow[row_num] = FALSE /* This row is no longer
1180                             in Basis */
1181 minoriteration(colnr, row_nr)
1182
1183 Look for columns with only one Coefficient (while)
1184 if found
1185
1186 set frow[row_num] = FALSE /* This row is no longer
1187                             in Basis */
1188 rownum[] = 0
1189 update column[]
1190 numit++ /* counter how many iterations to do */
1191 /* at end */
1192 col[numit] = colnr /* replaces minoriteration. But this */
1193 /* is done later and we need arrays */
1194 row[numit] = row_nr /* col and row therefore */
1195
1196 /* real invertation */
1197 for all columns (From beginning to end )
1198 if fcol
1199 set fcol[] = FALSE
1200 setpivcol ( Lower[Rows + j] , Rows + j, pcol)
1201 Loop through all the rows to find coefficient with
1202 frow[row_nr] && pcol[row_nr]
1203 /* Interpretation:
1204 Look for first coefficient in (partly inverted)
1205 Basis matrix which is nonzero and use it for pivot.*/
1206
1207 /* comparison for pcol is dangerous, but ok after
1208 rounding */
1209
1210 Error conditions
1211
1212 /* Now we know pivot element */
1213
1214 frow[row_nr] = FALSE
1215 condensecol (row_nr, pcol)
1216 rhsmocol (theta, row_nr, Rows + j)
1217 addetacol()
1218
1219 For all stored actions /* compare numit */
1220 set colnr / varin
1221 row_nr
1222 init pcol with 0
1223 set pcol[]
1224 actualize pcol[0]
1225 condensecol(row_nr, pcol)
1226 rhsmocol(theta, row_nr, varin)
1227 addetacol()
1228
1229 Round Rhs
1230 print info
1231 Justinverted = TRUE
1232 Doinvert = FALSE
1233
1234 free()
1235
1236
1237
1238 colprim(int *colnr,
1239 short minit,
1240 REAL* drow)
1241 /* Each, colprimal - rowprimal and rowdual - coldual form a couple */

```

```

1242     btran( 1,0,0,0,0,0,...)
1243     update result depending on variables at upper bound
1244
1245         look for variable with negative reduced costs
1246
1247     =====
1248     More detailed:
1249
1250     init *colnr = 0
1251         dpiv = set to a small negative number.
1252     if NOT minit
1253         drow = 1,0,0,0,0,0.....
1254         btran(drow)
1255         For variables at upper bound we have to calculate the
1256         reduced cost differently:
1257         multiply each coefficient in column with reduced cost of row=
1258         slackvariable and sum. This is new reduced cost.
1259         round reduced costs "drow"
1260     Look for variable which has upper bound Greater than Zero, which
1261     is nonbasic.
1262         Perhaps correct sign of reduced costs of variables
1263         at upper bound.
1264         take variable with most negative reduced costs.
1265         save reduced costs in dpiv and
1266         col_nr in *col_nr
1267     print trace info
1268     if *col_nr == 0
1269         set some variables, that indicate that we are optimal.
1270     return True, if *col_nr > 0
1271
1272
1273
1274     rowprim(int colnr,
1275             int * row_nr,
1276             REAL * theta,
1277             REAL * pcol)          /* contains current column from var conr */
1278
1279     /* search for good candidate in a column for pivot */
1280     First look for big entries
1281     Second ( this means first failed ) look also for smaller entries
1282     Warning numerical instability
1283
1284     Determine UNBOUNDED
1285     Perhaps shift variable to its upper bound
1286
1287     Aim: determin valid pivot element
1288
1289     print some info
1290
1291     Return true, if we had been successful finding a pivot element.
1292
1293     =====
1294     More detailed:
1295
1296     init *row_nr = 0
1297         *theta = infinity
1298     loop through all the rows
1299         gout = maximal steplength = *theta
1300         *row_nr = number of that row.
1301         /* At first look only for Steps which are not calculated with
1302         very small divisors. If no such steps found, take also
1303         small divisors in consideration */
1304     Perhaps we found numerical problems. Print warning in this case
1305
1306     If we did not find a limiting row, we are perhaps unbounded.
1307     (upperbound on that variable = infinity)
1308     The case that we have an upper bound is treated separately
1309
1310     print some trace info
1311
1312     return (*row_nr > 0)
1313
1314

```

```

1315
1316 rowdual(int *row_nr)
1317     look for infeasibilities
1318
1319     init *row_nr = 0
1320         minrhs = a little bit negative
1321
1322     loop through all the rows
1323         if we find a variable which is not zero, but has to be
1324         then we break this loop. *row_nr = i
1325
1326         calculate distance between rhs[i] and upperbound[i]
1327         take smaller one
1328
1329         |-----|-----|
1330             0      rhs[i]      upperbound[i]
1331             =g
1332
1333
1334         minrhs is smallest g
1335         *row_nr is corresponding rownumber.
1336
1337     print some trace info
1338     return (*row_nr > 0)
1339
1340
1341 coldual(int row_nr,
1342     int *colnr,
1343     short minit,
1344     REAL *prow,
1345     REAL *drow)
1346     looks also for a candidate for pivot.
1347
1348
1349
1350
1351 iteration (int row_nr,
1352     int varin,
1353     REAL *theta,
1354     REAL up,
1355     short *minit,
1356     short *low,
1357     short primal,
1358     REAL *pcol)
1359
1360     execute one iteration
1361
1362
1363
1364
1365 solvelp()
1366         First check if right hand side is positive everywhere
1367         and smaller than possible upper bound of this row.
1368         In this case we start with a feasible basis.
1369
1370
1371
1372     ATTENTION:
1373     If we want to use solvelp() directly, skipping
1374     solve() and milpsolve() we have to be very careful.
1375     e.g. solve() sets the global variables!!!
1376
1377 is_int(REAL value)
1378     simple routine, checks, if a REAL value is integer.
1379
1380 construct_solution(REAL *sol)
1381     The routine does exactly, what its name says.
1382     There are two parts, with and without scaling.
1383
1384
1385     First set all variables to their lower bounds.
1386     Then set all basis variables to their true values, i.e.
1387     the right hand side is added to the lower bound.

```

```

1388         (The reason is that all variables have been transformed
1389         to have lower bound zero) ## Autor fragen!!
1390         Finally set the non basic variables, which are not at
1391         their lower bound to their upper bound.
1392         Calculate values of the slack variables of a row.
1393
1394
1395
1396 calculate_duals()
1397         In fact calculate the reduced costs of the slack variables
1398         and correct values.
1399
1400
1401
1402 milpsolve(REAL    *upbo,
1403          REAL    *lowbo,
1404          short   *sbasis,
1405          short   *slower,
1406          int     *sbas)
1407         First of all: copy the arrays upbo and lowbo
1408         to the pointers of Upbo and Lowbo. (Memory
1409         is allocated for these arrays. Pointers point
1410         to lp->upbo and lp->lowbo)
1411         (size of memory is updated, if new columns are
1412         added.)
1413         These arrays came from solve() as ORIGINAL
1414         bounds. Therefore no shifting of transformed bounds
1415         necessary in lpkit.c if solve() is called.
1416
1417         if (LP->anti_degen)
1418             disturb lower and upper bound a little bit.
1419         if (!LP->eta_valid)
1420             shift lower bounds to zero. This means:
1421             Orig_lowbo    ... unchanged
1422             Orig_upbo     ... unchanged
1423             lowbo         ... unchanged (implicit in code = 0)
1424             upbo          ... mainly upbo_old - lowbo.
1425
1426         solvelp()
1427
1428         if (LP->anti_degen)
1429             restore upbo, lowbo, Orig_rh and solve again.
1430
1431         if (OPTIMAL solution of LP)
1432             check, if we can cutoff branch with LP value.
1433             look for noninteger variable (look for first
1434             or look random)
1435             if (noninteger variables)
1436                 setup two new problems.
1437                 Malloc new memory
1438                 memcpy the data
1439                 solve problems recursively (Floor_first/ceiling_first)
1440                 set return values
1441             else
1442                 /* all required values are int */
1443                 check, if better solution found.
1444                 /* Yes */
1445                 memcpy data
1446                 perhaps break B+B
1447
1448
1449         Recursive Function. Pure depth first search.
1450             No easily accessible nodelist, because of depth
1451             first search. (Also less active nodes)
1452         Branching on first noninteger variablen or
1453             on a randomly selected variable.
1454         Avoid inverting if possible.
1455
1456
1457
1458 solve(lprec *lp)
1459         init BEST-Solution, init perhaps basis, call milpsolve
1460

```

```

1461
1462
1463 lag_solve(lprec *lp, REAL start_bound, int num_iter, short verbose)
1464     Lagrangean solver.
1465
1466
1467
1468
1469
1470
1471 /*****
1472 /**
1473 /**
1474 /**          Routines in file "debug.c"
1475 /**
1476 /**
1477 *****/
1478
1479
1480 static void print_indent(void)
1481 void debug_print_solution()
1482 void debug_print_bounds(REAL *upbo, REAL *lowbo)
1483 void debug_print(char *format, ...)
1484
1485
1486 =====
1487
1488 static void print_indent(void)
1489     Used for printing the branch and bound tree. For every node the depth
1490     in the tree is shown with some ASCII graphic.
1491
1492 void debug_print_solution()
1493     For all columns
1494         print_indent()
1495         print the variable name (true or artificial) and its value.
1496
1497 void debug_print_bounds(REAL *upbo, REAL *lowbo)
1498     For all columns
1499         Print the lower bounds if they are different from zero with true or
1500         artificial name.
1501         Print the upper bounds if they are different from infinity with true or
1502         artificial name.
1503
1504 void debug_print(char *format, ...)
1505
1506
1507
1508
1509 /*****
1510 /**
1511 /**
1512 /**          Routines in file "lp_solve.c"
1513 /**
1514 /**
1515 *****/
1516
1517
1518 void print_help(char *argv[])
1519     Print usage message. If the program is called with option "-h" the usage
1520     message is printed. The usage message gives the options which can be
1521     given when calling the program.
1522
1523 int main(int argc, char *argv[])
1524     Initialise some data. Read all the options and make use of them.
1525     (Options have to be given separately, non existing options are just ignored.)
1526
1527     Read MPS file or lp_file from stdin.
1528
1529     Perhaps print out some information about LP and do some manipulations on
1530     LP (i.e. scaling).
1531
1532     call solve(lp)
1533

```

```

1534     Check return status:
1535     If (OPTIMAL)
1536         Print out solution and some statistics.
1537     else print solution status.
1538
1539
1540
1541
1542     /*****
1543     /**
1544     /**
1545     /**          Routines in file "lpkit.c"
1546     /**
1547     /**
1548     *****/
1549
1550     The main purpose of this file is to give several "manipulation" routines to
1551     the user. The user should be able to read information from the current
1552     problem. But he/she should also be able to change information in the
1553     problem. So for example, it is possible to add new constraints to the
1554     problem, to change the bounds of the variables etc.
1555
1556
1557
1558     void error(char *format, ...)
1559     lprec *make_lp(int rows, int columns)
1560     void delete_lp(lprec *lp)
1561     lprec *copy_lp(lprec *lp)
1562     void inc_mat_space(lprec *lp, int maxextra)
1563     void inc_row_space(lprec *lp)
1564     void inc_col_space(lprec *lp)
1565     void set_mat(lprec *lp, int Row, int Column, REAL Value)
1566     void set_obj_fn(lprec *lp, REAL *row)
1567     void str_set_obj_fn(lprec *lp, char *row)
1568     void add_constraint(lprec *lp, REAL *row, short constr_type, REAL rh)
1569     void str_add_constraint(lprec *lp,
1570                           char *row_string,
1571                           short constr_type,
1572                           REAL rh)
1573     void del_constraint(lprec *lp, int del_row)
1574     void add_lag_con(lprec *lp, REAL *row, short con_type, REAL rhs)
1575     void str_add_lag_con(lprec *lp, char *row, short con_type, REAL rhs)
1576     void add_column(lprec *lp, REAL *column)
1577     void str_add_column(lprec *lp, char *col_string)
1578     void del_column(lprec *lp, int column)
1579     void set_upbo(lprec *lp, int column, REAL value)
1580     void set_lowbo(lprec *lp, int column, REAL value)
1581     void set_int(lprec *lp, int column, short must_be_int)
1582     void set_rh(lprec *lp, int row, REAL value)
1583     void set_rh_vec(lprec *lp, REAL *rh)
1584     void str_set_rh_vec(lprec *lp, char *rh_string)
1585     void set_maxim(lprec *lp)
1586     void set_minim(lprec *lp)
1587     void set_constr_type(lprec *lp, int row, short con_type)
1588     REAL mat_elm(lprec *lp, int row, int column)
1589     void get_row(lprec *lp, int row_nr, REAL *row)
1590     void get_column(lprec *lp, int col_nr, REAL *column)
1591     void get_reduced_costs(lprec *lp, REAL *rc)
1592     short is_feasible(lprec *lp, REAL *values)
1593     short column_in_lp(lprec *lp, REAL *testcolumn)
1594     void print_lp(lprec *lp)
1595     void set_row_name(lprec *lp, int row, nstring new_name)
1596     void set_col_name(lprec *lp, int column, nstring new_name)
1597     static REAL minmax_to_scale(REAL min, REAL max)
1598     void unscale_columns(lprec *lp)
1599     void unscale(lprec *lp)
1600     void auto_scale(lprec *lp)
1601     void reset_basis(lprec *lp)
1602     void print_solution(lprec *lp)
1603     void write_LP(lprec *lp, FILE *output)
1604     void write_MPS(lprec *lp, FILE *output)
1605     void print_duals(lprec *lp)
1606     void print_scales(lprec *lp)

```

```

1607
1608
1609 What is done in the routines:
1610 =====
1611
1612 void error(char *format, ...)
1613 lprec *make_lp(int rows, int columns)
1614
1615     Construct a new LP. Set all variables to some default values.
1616     The LP has "rows" rows and "columns" columns. The matrix contains
1617     no values, but space for one value. All arrays which depend on
1618     "rows" and "columns" are malloced.
1619
1620     The problem contains only continuous variables.
1621     Upper bounds are infinity, lower bounds are zero.
1622     The basis is true, all rows are in basis. All columns are nonbasic.
1623     The eta-file is valid. Solution, best_solution and duals are Zero.
1624     And some other default values.
1625
1626
1627
1628 void delete_lp(lprec *lp)
1629
1630     Delete ALL the malloced arrays. At last free the structure.
1631
1632
1633 lprec *copy_lp(lprec *lp)
1634
1635     Copy first the structure of the lp, this means especially, that all
1636     the constant values are copied.
1637     Copy all the arrays of the lp and set the pointers to the new arrays.
1638     Mainly use MALLOCCOPY for this, this means: malloc space and copy data.
1639
1640
1641 void inc_mat_space(lprec *lp, int maxextra)
1642
1643     Test if realloc necessary. If yes, realloc arrays "mat" and "col_no".
1644     If lp is active, set some global variables which could be changed by
1645     realloc.
1646
1647
1648 void inc_row_space(lprec *lp)
1649
1650     Test, if increment necessary.
1651     This routine increments the space for rows with 10 additional rows.
1652     Therefore one condition for correct work of this routine is that
1653     it is never necessary to increase the
1654     number of additionally rows in one step with more than 10!
1655     Several arrays are reallocated.
1656     At last, if LP is active, set some global variables new, because they could
1657     have changed.
1658
1659
1660 void inc_col_space(lprec *lp)
1661
1662     similar to routine increment row space. The problems are also the same.
1663     Several Arrays are reallocated, but no shift of values.
1664
1665
1666
1667
1668 void set_mat(lprec *lp, int Row, int Column, REAL Value)
1669
1670     set one element in matrix.
1671     Test, if row and column are in range. Scale value.
1672     If colnum is in basis and row not objective row set Basis_valid = FALSE
1673     Always set eta_valid = FALSE (is this necessary?)
1674
1675     Search in column for entry with correct rownumber.
1676     If row found scale value again but with other expression than first time.
1677     Perhaps change sign
1678
1679     If row not found:

```

```

1680     Increment mat_space for one additional element.
1681     Shift matrix and update col_end.
1682     Set new element "row" and scale value perhaps (same problem as above)
1683     Rowend is not valid any longer
1684     update number of nonzeros and copy this value if lp is active
1685
1686
1687
1688 void set_obj_fn(lprec *lp, REAL *row)
1689
1690     call in one loop for dense row the function set_mat().
1691     No test is done, if we want to include Elements with value "0".
1692     These values are included into the matrix!
1693
1694
1695
1696 void str_set_obj_fn(lprec *lp, char *row)
1697
1698     reserve space for one row
1699     try with "strtod()" to change all the strings to real values
1700     call set_obj_fn()
1701     free space
1702
1703
1704
1705 void add_constraint(lprec *lp, REAL *row, short constr_type, REAL rh)
1706
1707     first reserve space for integers for length of one row.
1708     Mark all the positions, which contain nonzeros and update non_zeros
1709     malloc space for a complete new matrix
1710     increment matrix space by null??
1711     rows++
1712     sum++
1713     increment row space
1714     if scaling
1715         shift the values
1716         and set scaling value for new row to 1
1717     if names used
1718         invent new name for row
1719     if columns are scaled
1720         scale coefficients
1721     calculate change_sign
1722     copy every column from old matrix to new matrix. Perhaps add new entry for
1723     new row.
1724     Update col_end
1725     copy new matrix back to old matrix.
1726     free the allocated arrays
1727
1728     shift orig_upper_bounds
1729         orig_lower_bounds
1730         basis
1731         lower
1732         must_be_int
1733
1734     update Basis info
1735     set bounds for slack variables
1736
1737     change_sign for rhs, but comparison is made with sense of constraint.
1738
1739     rows_end_valid = false
1740     put slackvariable for this row into basis
1741     if lp == active, set globals
1742     eta_file = non_valid.
1743
1744
1745
1746 void str_add_constraint(lprec *lp,
1747                        char *row_string,
1748                        short constr_type,
1749                        REAL rh)
1750
1751     This routine is similar to the routine str_set_obj_fn. The same idea,
1752     but call add_constraint.

```



```

1753
1754
1755
1756
1757 void del_constraint(lp prec *lp, int del_row)
1758
1759     First check, if rownumber exists.
1760     For all columns
1761         For every coefficient in column
1762             if it is not rownumber,
1763                 then shift elements to smaller nonzero index and perhaps correct row index.
1764             else delete
1765                 update col_end
1766     shift values for orig_rhs, ch_sign, bas, row_name down by one.
1767     Update values in bas
1768     shift values for lower, basis, orig_upbo, orig_lowbo, must_be_int, scaling down
1769         by one.
1770     update rows and sum
1771     set row_end_valid = FALSE
1772     if lp = active, set globals.
1773     eta_valid = FALSE
1774     basis_valid = FALSE
1775
1776
1777
1778 void add_lag_con(lp prec *lp, REAL *row, short con_type, REAL rhs)
1779
1780     Calloc/Realloc space for lag_row, lag_rhs, lambda, lag_con_type
1781     Fill arrays.
1782
1783 void str_add_lag_con(lp prec *lp, char *row, short con_type, REAL rhs)
1784
1785     Same idea as always. Reserve space for array, strtod values into this array,
1786     call add_lag_con and free array.
1787
1788
1789
1790 void add_column(lp prec *lp, REAL *column)
1791
1792     update columns and sums,
1793     increment space for columns and matrix
1794     if scaling used
1795         set scaling factor for column to "1" and scale all values with row[scaling].
1796     for all elements in (dense) column
1797         if value is not zero
1798             write it in matrix.
1799     update col_end
1800         orig_lowbo
1801         orig_upbo
1802         lower
1803         basis
1804         must_be_int
1805         invent perhaps name for column
1806     row_end_valid = FALSE
1807     if lp = active
1808         set sum, columns, non_zeros
1809
1810
1811 void str_add_column(lp prec *lp, char *col_string)
1812
1813     Same idea as always. Reserve space for array, strtod values into this array,
1814     call add_column and free array.
1815
1816
1817
1818 void del_column(lp prec *lp, int column)
1819
1820     check, if column is in range
1821     if column in Basis set basis_valid to FALSE
1822     else update bas
1823     shift names_used,
1824         must_be_int
1825         orig_upbo

```

```

1826         orig_lowbo
1827         upbo
1828         lowbo
1829         basis
1830         lower
1831         scaling
1832     update lagrangean stuff
1833     copy elements in matrix down.
1834     update col_end
1835     update non_zeros
1836     row_end_valid = FALSE
1837     eta_valid = FALSE
1838     update sum
1839         column
1840     if lp = active
1841         set_globals()
1842
1843
1844
1845 void set_upbo(lprec *lp, int column, REAL value)
1846
1847     Test if column number in range
1848     scale value
1849     Test, if new value is feasible (greater than lower bound)
1850     eta_valid = FALSE
1851     set_orig_upbo
1852
1853 void set_lowbo(lprec *lp, int column, REAL value)
1854
1855     Test if column number in range
1856     scale value
1857     Test, if new value is feasible (smaller than upper bound)
1858     eta_valid = FALSE
1859     set_orig_lowbo
1860
1861 void set_int(lprec *lp, int column, short must_be_int)
1862
1863     Test if column number in range
1864     set must_be_int
1865     If variable must be integer, unscale column
1866
1867
1868 void set_rh(lprec *lp, int row, REAL value)
1869
1870     Test, if row_number is in range
1871     Test, if row_number for objective row should be set, WARNING
1872     scale value and change sign.
1873     eta_valid = FALSE
1874
1875
1876 void set_rh_vec(lprec *lp, REAL *rh)
1877
1878     For all rows
1879         scale and change sign
1880         set_orig_rh
1881         eta_valid = FALSE
1882
1883
1884 void str_set_rh_vec(lprec *lp, char *rh_string)
1885
1886     Same idea as always. Reserve space for array, strtod values into this array,
1887     call set_rh_vec and free array.
1888
1889
1890 void set_maxim(lprec *lp)
1891
1892     if maxim == FALSE
1893         multiply all Values in row[0] with -1
1894         eta_valid = FALSE
1895     set_maximise = TRUE
1896     ch_sign[0] = TRUE
1897     if LP = active, set Maximise = TRUE
1898

```

```

1899
1900 void set_minim(lprec *lp)
1901
1902     if maxim == TRUE
1903         multiply all Values in row[0] with -1
1904         eta_valid = FALSE
1905     set maximise = FALSE
1906         ch_sign[0] = FALSE
1907     if LP = active, set Maximise = FALSE
1908
1909
1910
1911 void set_constr_type(lprec *lp, int row, short con_type)
1912
1913     Test, if row_number is in range
1914     if type == EQUAL
1915         set upper bound on slackvariable to zero
1916         basis_valid == FALSE
1917         if change_sign[row]
1918             multiply all coefficients with -1
1919             eta_valid = FALSE
1920             change_sign = FALSE
1921             change sign of orig_rh
1922     if type == LESSEQUAL
1923         set upper bound on slackvariable to infinity
1924         basis_valid == FALSE
1925         if change_sign[row]
1926             multiply all coefficients with -1
1927             eta_valid = FALSE
1928             change_sign = FALSE
1929             change sign of orig_rh
1930     if type == GREATEREQUAL
1931         set upper bound on slackvariable to infinity
1932         basis_valid == FALSE
1933         if NOT change_sign[row]
1934             multiply all coefficients with -1
1935             eta_valid = FALSE
1936             change_sign = TRUE
1937             change sign of orig_rh
1938     else
1939         error wrong constraint type
1940
1941
1942
1943 REAL mat_elm(lprec *lp, int row, int column)
1944     /* get value of matrix element in row and column */
1945
1946
1947     Test, if row_number is in range
1948     Test, if col_number is in range
1949     value = 0
1950     loop through column
1951     if value found
1952         unscale and change_sign
1953     return value
1954
1955
1956 void get_row(lprec *lp, int row_nr, REAL *row)
1957     /* this is dense form */
1958
1959     Test, if row_number is in range
1960     for all columns
1961         initialise value with 0
1962         for all entries in column
1963             if row found, write value
1964             unscale value
1965         if change_sign
1966             multiply with -1
1967
1968
1969
1970 void get_column(lprec *lp, int col_nr, REAL *column)
1971

```

```

1972     Test, if column is in range.
1973     /* column is dense*/
1974     initialise columnarray with 0
1975     for all elements in this column, copy to dense array
1976     unscale and change sign
1977
1978
1979 void get_reduced_costs(lprec *lp, REAL *rc)
1980
1981     Basis has to be valid
1982     set_globals
1983     if eta_valid = FALSE
1984         invert
1985     initialise array with 0
1986     set rc[0] = 1
1987     btran(rc)
1988     For all columns
1989         if variable not in basis AND upper bound > 0
1990             rc[column] = SUM (over all elements in Column) mat.value * rc[row]
1991     round all values
1992
1993
1994 short is_feasible(lprec *lp, REAL *values)
1995
1996     Unscale values and look, if they are between orig_lower and orig_upper bounds
1997     allocate space for a new rhs
1998     With this values calculate rhs
1999     check if rhs is lessequal than orig rhs for LE rows and equal to orig_rhs
2000     for EQ rows.
2001
2002
2003 short column_in_lp(lprec *lp, REAL *testcolumn)
2004
2005     for all columns
2006         for all elements in column
2007             unscale value and change_sign
2008             check if difference smaller than epsilon
2009     return TRUE or FALSE
2010
2011
2012
2013
2014 void print_lp(lprec *lp)
2015
2016     print rowwise in readable form.
2017
2018
2019 void set_row_name(lprec *lp, int row, nstring new_name)
2020
2021     Perhaps allocate memory for names and initialise with default names
2022     strcpy rowname
2023
2024
2025 void set_col_name(lprec *lp, int column, nstring new_name)
2026
2027     Perhaps allocate memory for names and initialise with default names
2028     strcpy colname
2029
2030
2031
2032 static REAL minmax_to_scale(REAL min, REAL max)
2033
2034     calculate scaling factor depending on min and max
2035
2036
2037 void unscale_columns(lprec *lp)
2038
2039     for all columns
2040         for all coefficients in column
2041             unscale (columnscaling)
2042     for all columns
2043         unscale bounds
2044     set scaling vector to 1

```

```

2045     columns_scaled = FALSE
2046     eta_valid = FALSE
2047
2048
2049 void unscale(lprec *lp)
2050
2051     Work only if scaling used
2052     for all columns
2053         for all coefficients in column
2054             unscale (columnscaling)
2055     for all columns
2056         unscale bounds
2057     for all columns
2058         for all coefficients in column
2059             unscale (rowscaling)
2060     for all rows
2061         unscale orig_rhs
2062     free scale
2063     scaling_used = FALSE
2064     eta_valid = FALSE
2065
2066
2067
2068
2069 void auto_scale(lprec *lp)
2070
2071     find row maximum and row minimum. Use these values to scale problem.
2072
2073 void reset_basis(lprec *lp)
2074
2075     basis_valid=FALSE
2076
2077
2078 void print_solution(lprec *lp)
2079
2080     Print solution to stdout
2081     Print all variables
2082     In some cases
2083         Print slack variables ???
2084     Print duals
2085
2086
2087
2088 void write_LP(lprec *lp, FILE *output)
2089
2090     print LP rowwise in readable form.
2091
2092
2093 void write_MPS(lprec *lp, FILE *output)
2094
2095     The routine write_MPS seems to do no unscaling. However it uses internally
2096     the routine get_column() which does unscaling!
2097
2098
2099
2100 void print_duals(lprec *lp)
2101
2102     Print all duals
2103
2104
2105 void print_scales(lprec *lp)
2106
2107     Print all row scales
2108     print all column scales.
2109
2110
2111
2112
2113
2114
2115
2116
2117

```

```

2118
2119
2120 /*****
2121 /**
2122 /**
2123 /**      Routines in file "read.c"
2124 /**
2125 /**
2126 *****/
2127
2128
2129
2130 void yyerror(char *string)
2131 void check_decl(char *str)
2132 static int hashval(const char *string)
2133 static hashelem *gethash(char *variable)
2134 void add_int_var(char *name)
2135 void init_read(void)
2136 static column *getrow(column *p,
2137                        int row)
2138 static bound *create_bound_rec(void)
2139 void null_tmp_store(void)
2140 static void store(char *variable,
2141                  int row,
2142                  REAL value)
2143
2144 void store_re_op(void)
2145 void rhs_store(REAL value)
2146 void var_store(char *var, int row, REAL value)
2147 void store_bounds(void)
2148 void add_constraint_name(char *name, int row)
2149 void readinput(lprec *lp)
2150 lprec *read_lp_file(FILE *input, short verbose, nstring lp_name)
2151
2152
2153 =====
2154
2155 To understand the idea of this file you should also read carefully the
2156 comments directly at the beginning of this file!
2157
2158
2159
2160 void yyerror(char *string)
2161     Output error string and line number
2162
2163 void check_decl(char *str)
2164     We expect string "int". If this is not the case give error message.
2165
2166 static int hashval(const char *string)
2167     Calculate an integer hash value. (Modulo HASHSIZE).
2168
2169 static hashelem *gethash(char *variable)
2170     Returns a pointer to hashelement with name = variable.
2171     If this hashelement does not exist, gethash() returns a NULL pointer.
2172
2173 void add_int_var(char *name)
2174     Check if name exists. (if not, error message.)
2175     Check if it is the first time this variable was declared to be integer.
2176     Set flag for this variable to be integer.
2177
2178 void init_read(void)
2179     Init hashtable and globals.
2180
2181 static column *getrow(column *p,
2182                      int row)
2183     search in column-list (p is pointer to first element of column-list)
2184     for column->row = row.
2185     getrow() returns a pointer to this column structure.
2186     If not found a NULL-pointer is returned
2187     Follows one chain of pointers until correct element is found.
2188
2189 static bound *create_bound_rec(void)
2190     Creates a bound record.

```

```

2191     Calloc space.
2192     Set lowbo = 0 and upbo = Infinite
2193     Return pointer to this structure.
2194
2195 void null_tmp_store(void)
2196     clears the tmp_store variable after all information has been copied
2197
2198 static void store(char *variable,
2199                  int row,
2200                  REAL value)
2201     Store a value of the (sparse) matrix in data structure.
2202     If Value == 0, display warning.
2203     Three cases have to be distinguished:
2204     First: Variable does not exist
2205           Calloc space for info about variable
2206           update number of variables
2207           insert this element first into hashtable
2208           Calloc space for value
2209           insert rownumber and value into structure
2210     Second: Variable exists and has no value in that row yet
2211           Calloc space for value
2212           Insert rownumber and value into structure and insert into pointer
2213           chain.
2214     Third: Variable exists and has already a value in that row.
2215           add value to old value.
2216
2217 void store_re_op(void)
2218     switch yytext[0]
2219     case =
2220     case >
2221     case <
2222     default    error exit
2223
2224
2225
2226 void rhs_store(REAL value)
2227     Store RHS value in the rightside structure.
2228     Two cases are distinguished: Constraints with several variables have a right
2229     hand side, Constraints with only one variable are no constraints but bounds.
2230
2231 void var_store(char *var, int row, REAL value)
2232     Store all data in the right place.
2233     Distinguish between bound and constraint.
2234
2235     error exit, if variable name is too long.
2236     update Lin_term_count carefully, because it could be a bound.
2237     If it is possible that constraint is only a bound, store its values in
2238     temporary space.
2239     If it is sure that it is NOT a bound, store values from temporary space
2240     first. Init temporary space with zero for further use.
2241     Store the values for the last read variable.
2242
2243 void store_bounds(void)
2244     The constraint was in fact a Bound. We store it now. The information for
2245     the variable can be found in temporary space.
2246
2247     If value == 0: error exit.
2248
2249     Check, if we know this variable already.
2250     If new variable found
2251         Calloc space
2252         update number of variables
2253         init space
2254         insert space into hashtable
2255     else
2256         Check if space for bounds exists already and if not, create.
2257
2258     change perhaps direction of inequality, if negative coefficient in front of
2259     variable.
2260
2261     Check, if bound is feasible. If not: error exit.
2262     Perhaps display warning, if upper bound is negative.
2263     Insert bound into structure, but check if there exists already stronger

```

```

2264     bound. In this case display warning.
2265
2266     Check, if upperbound AND lower bound contradict each other. error exit.
2267
2268     clear temporary space.
2269
2270 void add_constraint_name(char *name, int row)
2271     Store constraint name in structure. The first name has to be handled
2272     differently.
2273
2274 void readinput(lpvec *lp)
2275     Transport the data from the intermediate structure to the sparse matrix
2276     and free the intermediate structure. The routine tries not to waste memory
2277     and frees every data which is copied to the other structure.
2278
2279     Copy all the given row names.
2280     For all Rows descending:
2281         store relational operator
2282         store rhs
2283         free memory
2284     For all equal rows change upper bound to zero.
2285
2286     If some rows do not have names, generate a name.
2287
2288     Read Hash structure /* variables loose their original number */
2289     initialise col_end of the variable.
2290     Set must_be_int and the bounds of the variable.
2291     copy name of variable.
2292     put matrix values in sparse matrix. /* No special sorting in a column */
2293     copy row index, value
2294     update number of nonzeros
2295     free space
2296     initialise col_end for last variable.
2297
2298     if verbose
2299         print some statistic information
2300         print basically MPS file.
2301
2302 lpvec *read_lp_file(FILE *input, short verbose, nstring lp_name)
2303     init some data
2304     call the parser yyparse()
2305
2306     Calloc new lp structure and initialise lots of data (sizes of arrays etc.)
2307     Calloc arrays and insert into structure.
2308
2309     Call readinput to get the information into the lp structure.
2310
2311     check maximise
2312     set constraint type
2313
2314     return pointer to lp.
2315
2316
2317
2318
2319 /******
2320 /**
2321 /**
2322 /**          Routines in file "readmps.c"
2323 /**
2324 /**
2325 /******
2326
2327
2328
2329
2330 int scan_line(char* line, char *field1, char *field2, char *field3,
2331             double *field4, char *field5, double *field6)
2332 void addmpscolumn(void)
2333 static int find_row(char *field)
2334 static int find_var(char *field)
2335 lpvec *read_mps(FILE *input, short verbose)
2336

```



```

2337 =====
2338
2339 This file reads in a MPS file. It describes MPS format in a comment at the
2340 beginning.
2341
2342 int scan_line(char* line, char *field1, char *field2, char *field3,
2343             double *field4, char *field5, double *field6)
2344     input a MPS file line in "line", output the fields or 0/empty string in
2345     field?.
2346     Return value is number of read fields.
2347     For every field the following is done:
2348         - first check, if this field exists in inputstring
2349         - second copy this part of input "line" to a buffer
2350         - third use sscanf to read this field into "field?"
2351         - update number of items.
2352
2353 void addmpscolumn(void)
2354     This routine uses the global variable "Last_column" which is calloced and
2355     filled in read_mps.
2356     - add_column
2357     - set_col_name
2358     - set_int
2359     Reset Last_column to all entries = 0.
2360
2361 static int find_row(char *field)
2362     Given a name of a row in "field" return the index number of the row.
2363     This is done by cycling one time through the entries of the rows.
2364     (If name not found, there is considered the case "Unconstrained_rows_found".
2365     This means we found N-rows, which are ignored. Perhaps we are just looking
2366     for the name of an N-row which we ignored. Therefore we will not find its
2367     index.)
2368     Exit from program if name does not exist.
2369
2370 static int find_var(char *field)
2371     The routine is very similar to "find_row".
2372     Given name of a variable return the index or exit program if name does not
2373     exist.
2374     It is done by cycling one time through all column names.
2375
2376 lprec *read_mps(FILE *input, short verbose)
2377     This is the longest routine in this file. It does all the work for reading
2378     in a MPS file. It contains lot of if(Debug) statements to print out
2379     debugging information.
2380
2381
2382     Initialise an empty LP.
2383     Initialise various data.
2384     Start while loop to read one line after the other from the MPS file.
2385         First skip comments
2386         Check first character in line to determin if it is "special" line.
2387             NAME
2388                 read problem name
2389             ROWS
2390                 Set variable "section" to corresponding value.
2391             COLUMNS
2392                 Set variable "section" to corresponding value.
2393             RHS
2394                 addmpscolumn()
2395                 Set variable "section" to corresponding value.
2396             BOUNDS
2397                 Set variable "section" to corresponding value.
2398             RANGES
2399                 Set variable "section" to corresponding value.
2400             ENDATA
2401                 Do nothing.
2402             Error exit, if unknown Keyword.
2403         else (normal line)
2404             scan_line()
2405             switch to the correct section to use the fields.
2406             NAME: error
2407             ROWS: N-row: take first one as objective row, i.e. row[0].
2408                 Forget further N-rows. Set some Variables to suppress
2409                 further error messages.

```

```

2410         L-row: str_add_constraint
2411             set_row_name
2412         G-row: str_add_constraint
2413             set_row_name
2414         E-row: str_add_constraint
2415             set_row_name
2416     else: error exit
2417 COLUMNS:
2418     The line should have 4 or 6 fields!
2419     If line has 5 fields, it could be a MARKER row!
2420     If we receive a new column name, i.e. the name is different from
2421     Last_col_name AND Column_ready, we addmpscolumn() and also its
2422     name. Set Column_ready to TRUE.
2423     else
2424     copy field2 to Last_col_name
2425     Insert field 4 and perhaps field 6 into the correct position
2426     of array Last_column.
2427     If line has 5 fields:
2428     Check for 'MARKER' Keyword. Addmpscolumn().
2429     Check for 'INTORG' or 'INTEND'.
2430     Update variable "Int_section" depending on result.
2431     Ignore unknown markers. (Do not exit)
2432     If not 4, 5 or 6 fields: error exit.
2433 RHS:
2434     The line should have 4 or 6 fields!
2435     If not, error exit.
2436     Insert field4 and field6 into the correct position using set_rh.
2437 BOUNDS:
2438     No check for the number of fields in the line is done.
2439     The following Types are handled:
2440     UP: set upper bound
2441     LO: set lower bound
2442     FX: set upper bound
2443         set lower bound
2444     PL: do nothing
2445     BV: set upper bound = 1
2446         set integer = TRUE
2447     FR: split into two variables.
2448         get_column, multiply with -1, add_column. Generate meaningful
2449         name for column, add name. Nothing is done with current
2450         lower and upper bounds.
2451     MI: change to positive variable by multiplying variable with -1.
2452         i.e. get_column, del_column, multiply with -1, add_column,
2453         generate column name, add name.
2454     else: error exit. Unsupported Type.
2455 RANGES:
2456     The line should have 4 or 6 fields!
2457     Error exit, if number of fields is wrong.
2458     Set bounds on row, i.e. writing the values directly into the
2459     array orig_upbo[].
2460 return Pointer to LP.
2461
2462
2463
2464 /*****
2465 /**
2466 /**
2467 /**      File "lex.l"
2468 /**
2469 /**
2470 *****/
2471
2472 Of course lex.l is the descriptive file for lex. To understand it, you need
2473 some knowledge about lex. In this part you only find the different tokens
2474 that are described in lex.l and in some cases you also find a description
2475 how lex works on an input file in a correct language.
2476
2477 The input file can contain Comments "/* ... */" which are just ignored.
2478 White space (WS), i.e. Blanks, Tabulators and New line are also ignored from
2479 the input file.
2480
2481 COMMA:      ",",
2482 MINIMIZE:   "min" or "MIN" or "Min" or ...

```

```

2483 MAXIMIZE:      "max" or .....
2484 CONS:         a constant number. Its value is returned in global variable "f".
2485 SIGN:         basically "+" or "-". Variable "Sign" == TRUE, if "-".
2486 VAR:          variable. Its name is returned in global variable "Last_var".
2487              Basically variable names start with a letter and than can contain
2488              letters, digits, brackets and special characters.
2489 COLON:        ":"
2490 AR_M_OP:      "*"
2491 RE_OP:        "=", "<=" or ">="
2492 END_C:        ";"
2493

```

2494 It also detects errors in input file.

```

2498 /*****
2499 /**
2500 /**
2501 /**          File "lp.y"
2502 /**
2503 /**
2504 *****/

```

2505  
2506 Similar remarks are valid for lp.y than for lex.l. This is the descriptive  
2507 file for yacc. Therefore you need some knowledge about yacc to understand  
2508 it. It works closely together with lex. You will find a rough description  
2509 of the recognised language.

2510  
2511 An input file consists of  
2512 - an objective\_function  
2513 - constraints  
2514 - int\_declarations

2515  
2516 Constraints can be a single constraint or several constraints.

2517  
2518 In front of a constraint there can be a name separated by a colon.

2519  
2520 Basically a constraint consists of  
2521 - x\_linear\_sum  
2522 - RE\_OP an Relational Operator  
2523 - x\_linear\_sum  
2524 - ";"

2525  
2526  
2527 int\_declarations can be empty. They can consist of several int\_declaration's.  
2528 Each int\_declaration consists of  
2529 - an int\_declarator  
2530 - vars, i.e. several variables, which can be comma separated.  
2531 - and finally ";"

2532  
2533 Each x\_linear\_sum consists of several x\_linear\_term's with SIGN.  
2534 Each x\_linear\_term is a constant or a linear\_term.  
2535 A linear\_term is a variable or a constant followed by a variable or  
2536 a constant "\*" variable.

2537  
2538 The objective function starts with "max" or "min" followed by a linear\_sum and  
2539 ends with ";".  
2540 A linear\_sum consists of linear\_term's with signs.

2541  
2542 You will not find a more detailed description of the language. Best is to  
2543 look into an example to get an idea of the format.

```

2544
2545
2546
2547
2548
2549 /*****
2550 /**
2551 /**
2552 /**          Function calling tree
2553 /**
2554 /**
2555 *****/

```

```

2556
2557 The following lines have been produced using cflow. They give
2558 an overview of the calling structure of lp_solve. Refer to
2559 the cflow manual, if you have difficulties, to understand the output.
2560
2561
2562 The functions are arranged in levels. All the functions a single routine
2563 calls are intended one level more.
2564
2565 The line numbers refer to version 2.0 of lp_solve.
2566
2567 SOLVE.C
2568
2569
2570 1  lag_solve: int(), <solve.c 1516>
2571 2      malloc: <>
2572 3      exit: <>
2573 4      fprintf: <>
2574 5      calloc: <>
2575 6      memcpy: <>
2576 7      get_row: <>
2577 8      set_mat: <>
2578 9      print_lp: <>
2579 10     solve: int(), <solve.c 1462>
2580 11         set_globals: void(), <solve.c 16>
2581 12         lsvaild: short(), <solve.c 105>
2582 13             malloc: 2
2583 14             exit: 3
2584 15             fprintf: 4
2585 16             free: <>
2586 17             calloc: 5
2587 18     milpsolve: int(), <solve.c 1158>
2588 19         debug_print: <>
2589 20         memcpy: 6
2590 21         rand: <>
2591 22         invert: void(), <solve.c 320>
2592 23             fprintf: 4
2593 24             calloc: 5
2594 25             exit: 3
2595 26             minoriteration: void(), <solve.c 244>
2596 27             setpivcol: void(), <solve.c 215>
2597 28                 ftran: void(), <solve.c 66>
2598 29             error: <>
2599 30             condensecol: void(), <solve.c 174>
2600 31                 resize_eta: void(), <solve.c 162>
2601 32                     realloc: <>
2602 33                     exit: 3
2603 34                     fprintf: 4
2604 35             rhsmocol: void(), <solve.c 292>
2605 36                 fprintf: 4
2606 37                 exit: 3
2607 38             addetacol: void(), <solve.c 197>
2608 39             free: 16
2609 40     solvelp: int(), <solve.c 893>
2610 41         calloc: 5
2611 42         exit: 3
2612 43         fprintf: 4
2613 44         colprim: short(), <solve.c 502>
2614 45             btran: void(), <solve.c 87>
2615 46                 fprintf: 4
2616 47             setpivcol: 27
2617 48             rowprim: short(), <solve.c 564>
2618 49                 fprintf: 4
2619 50             condensecol: 30
2620 51             rowdual: short(), <solve.c 655>
2621 52                 fprintf: 4
2622 53             coldual: short(), <solve.c 712>
2623 54                 fprintf: 4
2624 55             iteration: void(), <solve.c 815>
2625 56                 addetacol: 38
2626 57                     fprintf: 4
2627 58             invert: 22
2628 59             free: 16

```

```

2629 60          fprintf: 4
2630 61          construct_solution: void(), <solve.c 1057>
2631 62          memset: <>
2632 63          debug_print_solution: <>
2633 64          is_int: short(), <solve.c 1044>
2634 65          floor: <>
2635 66          malloc: 2
2636 67          exit: 3
2637 68          debug_print_bounds: <>
2638 69          ceil: <>
2639 70          milpsolve: 18
2640 71          free: 16
2641 72          calculate_duals: void(), <solve.c 1129>
2642 73          btran: 45
2643 74          print_solution: <>
2644 75      print_solution: 74
2645 76          free: 16

```

```

2646
2647
2648
2649 READMPS.C

```

```

2650
2651
2652 1      read_mps: struct*(), <readmps.c 227>
2653 2          make_lp: <>
2654 3          strcpy: <>
2655 4          fgets: <>
2656 5          fprintf: <>
2657 6          sscanf: <>
2658 7          strcmp: <>
2659 8          calloc: <>
2660 9          exit: <>
2661 10         addmpscolumn: void(), <readmps.c 170>
2662 11             add_column: <>
2663 12                 set_col_name: <>
2664 13                 set_int: <>
2665 14         scan_line: int(), <readmps.c 101>
2666 15             strlen: <>
2667 16             strncpy: <>
2668 17             sscanf: 6
2669 18         set_row_name: <>
2670 19         str_add_constraint: <>
2671 20         find_row: int(), <readmps.c 188>
2672 21             strcmp: 7
2673 22             fprintf: 5
2674 23             exit: 9
2675 24         set_rh: <>
2676 25         find_var: int(), <readmps.c 210>
2677 26             strcmp: 7
2678 27             fprintf: 5
2679 28             exit: 9
2680 29         set_upbo: <>
2681 30         set_lowbo: <>
2682 31         set_int: 13
2683 32         get_column: <>
2684 33         add_column: 11
2685 34         strcat: <>
2686 35         set_col_name: 12
2687 36         del_column: <>

```

```

2688
2689
2690 DEBUG.C

```

```

2691
2692
2693 1      debug_print_solution: void(), <debug.c 21>
2694 2          print_indent: void(), <debug.c 7>
2695 3          fprintf: <>
2696 4          fprintf: 3
2697 5      debug_print_bounds: void(), <debug.c 41>
2698 6          print_indent: 2
2699 7          fprintf: 3
2700 8      debug_print: void(), <debug.c 72>
2701 9          print_indent: 2

```

```

2702 10      vfprintf: <>
2703 11      fputc: <>
2704
2705
2706
2707
2708
2709
2710
2711
2712  /*****
2713  /**
2714  /**
2715  /**      References
2716  /**
2717  /**
2718  /***/
2719
2720
2721
2722
2723
2724  Literatur:
2725      John Lawrence Nazareth,
2726          Computer Solution of Linear Programs,
2727          Oxford University Press 1987
2728
2729      Orchard-Hays, W.
2730          Advanced linear-programming computing techniques,
2731          McGraw-Hill, 1968
2732
2733      Chvatal, V.
2734          Linear Programming,
2735          W.H. Freeman and Company,
2736          New York and San Francisco 1983.
2737
2738      Nemhauser, G.L. and Wolsey, Laurence A.,
2739          Integer and Combinatorial Optimization,
2740          John Wiley & Sons, Inc. 1988
2741
2742
2743
2744

```