

Documentation of `rubikrotation.pl`

version 4.0

Nickalls RWD (dick@nickalls.org)
and
Syropoulos A (asyropoulos@yahoo.com)

Last revised: 03 March 2017
This file: `rubikrotationPL.pdf`

Contents

1	Overview	3
2	The code	6
2.1	rotation	19
2.2	random	47
2.3	writestate	49
2.4	ErrorMessage	53
2.5	gprint	53
2.6	checkstate	54
2.7	rr-overview	56
2.8	rrR	57
2.9	rrSr	60
2.10	rrLp	61
2.11	rrU	64
2.12	rrSu	66
2.13	rrDp	67
2.14	rrF	70
2.15	rrSf	72
2.16	rrBp	73
2.17	rr-derivative subs	75
2.18	rrRp	76
2.19	rrRw	76
2.20	rrRwp	76
2.21	rrRs	76
2.22	rrRsp	76
2.23	rrRa	76
2.24	rrRap	76
2.25	rrL	77
2.26	rrLw	77
2.27	rrLwp	77
2.28	rrLs	77
2.29	rrLsp	77
2.30	rrLa	77
2.31	rrLap	77
2.32	rrUp	77
2.33	rrUw	77
2.34	rrUwp	78

2.35	rrUs	78
2.36	rrUsp	78
2.37	rrUa	78
2.38	rrUap	78
2.39	rrD	78
2.40	rrDw	78
2.41	rrDwp	78
2.42	rrDs	78
2.43	rrDsp	79
2.44	rrDa	79
2.45	rrDap	79
2.46	rrFw	79
2.47	rrFp	79
2.48	rrFwp	79
2.49	rrFs	79
2.50	rrFsp	79
2.51	rrFa	79
2.52	rrFap	80
2.53	rrB	80
2.54	rrBw	80
2.55	rrBwp	80
2.56	rrBs	80
2.57	rrBsp	80
2.58	rrBa	80
2.59	rrBap	80
2.60	rrSup	80
2.61	rrSd	81
2.62	rrSdp	81
2.63	rrSl	81
2.64	rrSlp	81
2.65	rrSrp	81
2.66	rrSfp	81
2.67	rrSb	81
2.68	rrSbp	81
2.69	rubikmod	82
2.70	cleanstring	84
2.71	cutinfoblock	84
2.72	fixrepeatelement	86
2.73	repeat	88
2.74	quitprogram	90
2.75	showarray	90
2.76	cleanarray	91
2.77	restorebrackets	91
2.78	infoblockcolon	92
2.79	RemoveAllSpaces	95
2.80	CheckSyntax	95
2.81	inverse	100

1 Overview

This Perl program (`rubikrotation.pl`), is part of the L^AT_EX `rubikrotation` package, which itself is part of the Rubik bundle (available from <http://www.ctan.org/pkg/rubik>). A general overview of the interaction between `rubikrotation.sty` and this Perl program is presented in the RUBIKROTATION documentation (see Section *General overview*).

This Perl program reads (as input) a formatted ‘data’ file (typically the file `rubikstate.dat` —see below). This data-file defines the current state of the Rubik cube; each line beginning with a specified ‘keyword’. After processing the input data-file, the program writes the final state to a text-file (typically the file `rubikstateNEW.dat` when used with the `rubikcube` package), and writes any error messages to the file `rubikstateERRORS.dat`.

Usage: Version 3 onwards uses the input and output filenames specified by the `CALLing` command-line, the usage for the executable form being as follows:

```
$ rubikrotation -i <input filename> [-o <output filename>]
```

If no output filename is specified, then the default filename `rubikOUT.txt` is used. When used in conjunction with the `rubikcube` package, then `rubikcube.sty` calls the program and sets the input filename as `rubikstate.dat` and the output filename as `rubikstateNEW.dat`.

Functionality: Version 4 allows additional functionality with regard to the RubikRotation sequence of rotation-codes, namely: (a) repeating elements of the form: `...([repeatname],U,D2,F3)n...` where n is an integer, (b) a terminal ‘info-block’ for carrying special information regarding the sequence (meta data), and delimited by a balanced pair of angle brackets; for example, `...<(pf*) //(C2){h2}>` (see the RUBIKROTATION documentation for details). All delimiters in the rotation sequence must be balanced, otherwise the program will terminate. If the keyword ‘inverse’ appears inside an info-block, then this triggers the program to generate the (mathematically) inverse of the sequence of rotation-codes (see the RUBIKROTATION documentation for details).

Input datafile (`rubikstate.dat`)

This Perl program reads as input the plain-text file `rubikstate.dat` which is output by the file `rubikrotation.sty` in response to a `\RubikRotation` command in the `.tex` file. A typical form is as follows:

```
%% filename: rubikstate.dat
%% written by rubikrotation.sty=v3.4 (2016/10/30)
up,W,W,W,W,W,W,W,W
down,Y,Y,Y,Y,Y,Y,Y,Y
left,B,B,B,B,B,B,B,B
right,G,G,G,G,G,G,G,G
front,O,O,O,O,O,O,O,O
back,R,R,R,R,R,R,R,R
checkstate
rotation,[seqA],L,R2,([triple],U,D)3,F,B,<(pf*)>
```

Each line of the input datafile is a comma-separated sequence of elements (codes); the first element of each line is a ‘keyword’ which directs the Perl program. This data-file defines the current (initial) state of the Rubik cube (using the ‘face’ keywords `up`, `down`, `left`, `right`, `front`, `back`), and includes special keywords (`checkstate`, `rotation`, `random`) which trigger the relevant subroutines to process the input data accordingly.

Note that the keyword ‘rotation’ is associated with two different line formats which are distinguished according to whether or not the second element is the word ‘random’; one format is shown above. The other format, which is associated with a random rotation, has its second element as the word ‘random’ and the third element as an integer; for example:

`rotation,random,120`

Brackets: All brackets on a line must be balanced. Brackets of the same type must not be nested. Balanced pairs of brackets can be positioned anywhere. In the event of any unbalanced bracket, the program will terminate with an appropriate error message.

Square brackets are not processed as rotations, and hence can be used to indicate a name (only the first such bracket will be allocated as a ‘name’), or tag. Square brackets must not contain commas.

Curved brackets indicate a repeated rotation sequence (a ‘repeat block’); a trailing integer n ($n \geq 0$) is the repeat number, and if absent it is actioned as $n = 1$.

Angle brackets are used to carry sequence information (an ‘info-block’). Although info-blocks can be placed anywhere, the macros in the ‘rubikpatterns’ database typically has the info-blocks positioned as the final argument of a rotation sequence. There are no content restrictions for balanced angle brackets (= info-blocks)

Program actions

The Perl program is called by a system command issued by the `rubikrotation.sty` and triggered by a `\RubikRotation` command in the `.tex` file. The Perl program proceeds by reading the file `rubikstate.dat` line-by-line, and acting according to the keyword (first element of the line). Each input line is checked for syntax at an early stage. Significant syntax errors (unbalanced or nested brackets, missing commas) result in the program issuing appropriate error messages and then terminating cleanly.

If the keyword is either `up`, `down`, `left`, `right`, `front`, `back` then the subsequent colour codes are written to the associated face cubie variables.

If the keyword is `checkstate` then a simple error check of the state of the Rubik cube colour configuration is undertaken.

If the first two elements are `rotation,random` then a random sequence of rotations is carried out.

If the keyword is `rotation` and the second element is *not* ‘random’, then the elements are processed as a sequence of rotations, possible ‘repeat-blocks’, and a possible terminal ‘info-block’.

The final rubik state, together with a lot of general information and error messages are all written to the file `rubikstateNEW.dat`. All lines containing the word ‘ERROR’ are also written to the file `rubikstateERRORS.dat`. Since the file `rubikstateNEW.dat` is automatically input by the `rubikrotation.sty` as the `\RubikRotation` command terminates, then all this information also appears in the L^AT_EX log file.

Processing a RubikRotation sequence

We will now focus on the key actions associated with processing a typical rotation sequence.

- 1: The sequence is checked for syntax errors at an early stage (see above).
- 2: A copy of the original sequence \rightarrow `SequenceShort`
- 3: Infoblock(s) are removed \rightarrow `SequenceInfo`
- 4: Check for any ‘repeat blocks’. If detected, then we convert any associated () , are converted to { } ; respectively. Any repeat-blocks can now be safely processed as separate rotation elements and can be fed into the ‘rotation’ subroutine for further processing.
- 5: The sequence of comma separated elements (some of which may be ‘reformulated’ repeat-blocks) is now processed into an array, the elements of which are then passed to the ‘rotation’ subroutine for further processing. Any repeat-blocks are then detected and their elements sent back into the ‘rotation’ subroutine the appropriate number of times.
- 6: Any square-bracket elements are detected: the first such \rightarrow `SequenceName`
- 7: All rotation codes are processed mod (4), and each instance is then appended to a so-called ‘Long’ string \rightarrow `SequenceLong`

8: The four `$Sequence...` variables (see above) are written to the file `rubikstateNEW.dat` as the macros `\SequenceInfo`, `\SequenceName`, `\SequenceShort`, `\SequenceLong`. Note that this makes them available to the \LaTeX document, since the `\RubikRotation` command (in the `rubikrotation.sty`) automatically inputs the file `rubikstateNEW.dat` as control passes back to \LaTeX when the Perl program terminates. See also the code for the `\RubikRotation` command below.

9: All output data is written to the file `rubikstateNEW.dat`. This file is input by \LaTeX (see above) and hence all error messages also appear in the log file.

10: Error messages are collected in an array and written to the file `rubikstateNEW.dat` and also to the file `rubikstateERRORS.dat`. The \LaTeX command `\ShowRubikErrors` results in a verbatim copy of the `rubikstateERRORS.dat` file being input into the document.

RubikRotation command

The following `\RubikRotation` command (from the file `rubikrotation.sty`), shows the code for writing the file `rubikstate.dat`, calling the Perl program, and finally, the inputting of the file `rubikstateNEW.dat`.

```
\newcommand{\RubikRotation}[2][1]{%
  \typeout{---TeX process (rubikrotation.sty)-----}%
  \typeout{---NEW rotation command-----}%
  \typeout{---command = RubikRotation[#1]{#2}}%
  \typeout{---writing current Rubik state to file rubikstate.dat}%
  \@openstatefile% open data file
  \@print{\@comment filename: rubikstate.dat}%
  \@print{\@comment written by rubikrotation.sty%
    =v\RRfileversion\space (\RRfiledate)}%
  \@printrubikstate%
%% countingloop code from Feuersaenger (2015)
  \newcount\ourRRcounter%
  \@countingloop{\ourRRcounter} in 1:{#1}{%
    \immediate\write\outfile{rotation,#2}}%
  \@closestatefile% close data file
  \typeout{---CALLing Perl script (rubikrotation.pl)}%
  \immediate\write18{\rubikperlcmd}%
  \typeout{---inputting NEW datafile (data written by Perl script)}%
  \input{rubikstateNEW.dat}%
  \typeout{-----}%
}
```

2 The code

```
#!/usr/bin/env perl
##
use Carp;
use Fatal;
use warnings;
##
our $version = "v4.0 (03 March 2017)";
##
## rubikrotation.pl
## VERSION 4.0
##
## Copyright 03 March 2017,
## RWD Nickalls (dick@nickalls.org)
## A Syropoulos (asyropoulos@yahoo.com)
##
##-----
## changes in v3.6 (January 2017)
## (1) included Jaap Rm and Rc notation
## (2) new sub for improved expansion of mod-4 multiples of rotations (Oct 2016)
## (3) restructured to facilitate processing arrays through the rotation sub
## (4) included option for an <info> block
## (5) included Randelshofer superset ENG 3x3 notation
## (6) implemented an 'inverse' mode
## (7) improved syntax checking
## (8) used perltidy to polish the program layout
##      (but only when making the pdf documentation-- see file rubikrotationPL.pdf)
## (9) included a lot of new subroutines
##-----
## changes in v3.2:
## 1) changed program, prog --> script
## 2) added leading ... to the comments written by the <writestate> sub
##    (the ... code indicates that comments are written by the Perl script)
##
## 3) v3.2a added a \RubikSeqNEW{...} output line in the output file
##    to facilitate typesetting the rotation sequence (works OK just now)
##
```

```

## 4) v3.2c: added new commands:
##      \RotationSequenceName{}
##      \RotationSequenceClean{}
##
## 5) v3.2d: changed the returned command names (removed the Rotation part to keep it simple)
##      \Sequence{} = orig seq + NO NAME
##      \SequenceName{} = NAME only
##      \SequenceClean{} = clean seq + NO NAME
##
## 6) v3.2e: (25 Sept 2016)
##      changed some command names: use short & long for the Rubik R2 --> R,R code
##      (more intuitive than Clean)
##      \Sequence{} --> SequenceShort{}
##      \SequenceClean{} --> SequenceLong{}
##      removed the [ and ] around [name] variable
##
## 7) v3.2h: (2 Oct 2016)
##      improved the mod 4 routine using SUB rubikmod()
##      improved comments to log file re: rotation processing
##
##-----
## changes in v3.0:
## 1) accepts command-line arguments for input (mandatory) and output (optional) filenames
##    default output filename is: rubikOUT.txt
## 2) included the symbols [ and ] to denote a rotation-name label (ie as well as *)
## 3) fixed some of the variable definitions (as highlighted by <use strict> pragma)
##-----
## changes in v2.3:
## 1) accepts a single commandline argument (datafilename)
## 2) uses the standard modules Carp and Fatal (give extra line info on error)
##-----
## changes in v2.2:
## 1) changed licence --> LatexPP
## 2) included random n errors in ERROR messages (lines 492--495)
## 3) included version number in error message
##-----
#
# This file is part of the LaTeX rubikrotation package, and

```

```

# requires rubikcube.sty and rubikrotation.sty
#
# rubikrotation.pl is a Perl-5 program and free software:
# This program can be redistributed and/or modified under the terms
# of the LaTeX Project Public License Distributed from CTAN
# archives in directory macros/latex/base/lppl.txt; either
# version 1 of the License, or any later version.
#
# rubikrotation.pl is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

##-----
## OVERVIEW
## This program is part of the rubikrotation package, and is complementary to
## the LaTeX rubikcube package. It processes Rubik rotation sequences on-the-fly.
## The program reads a datafile (rubikstate.dat) output by the rubikcube package
## and writes the new state to the file rubikstateNEW.dat, which is then input
## by the TeX file. Further documentation accompanies the rubikrotation package.

## Note that all possible state changing rotations of a 3x3x3 cube are
## either combinations of, or the inverse of, just 9 different rotations,
## three associated with each XYZ axis.
##-----

##----- MAIN -----

## This main module opens three files, and
## sets up an array for collecting all errors (%error), and sets an error flag to "",
## reads in the rubik state data file =rubikstate.dat (output by TeXfile),
## and calls subs to write the TeX_OUT_FILE,
## and finally closes all files.
## Each line of the input file consists of a comma separated list of arguments.
## The first argument in each line of the file rubikstate.dat is the rubikkeyword.
## Program is documented in the rubikrotation.pdf (see section ‘Overview’)
##-----
## set autoflush for outputs

```

```

## $|=1;
##-----
our $source_file      = "";
our $out_file         = "rubikOUT.txt";           #default
our $argc             = @ARGV;
our $commandLineArgs = join( " ", @ARGV );
our $showargs         = "\tcommandline args = $commandLineArgs\n";
our $usage =
"\tUsage: rubikrotation [-h|--help|-v|--version] -i <input file> [-o <out file>]\n";
our $rubikversion = "\tVersion: this is rubikrotation version $version\n";
#
## check for correct number of commandline arguments and allocate filenames
#
if ( $argc == 0 || $argc > 4 ) {    # croak if 0 or more than 4 arguments
    croak $rubikversion, $showargs, "\tWrong no of arguments\n", $usage;
}
else {
    SWITCHES:
    while ( $_ = $ARGV[0] ) {
        shift;
        if ( /^-h$/ || /^--help$/ ) {
            die $rubikversion, $usage,
                "\twhere,\n"
                . "\t[-h|--help]\tgives this help listing\n"
                . "\t[-v|--version]\tgives version\n"
                . "\t[-i]      \tcreates specified input file\n",
                "\t[-o]      \tcreates specified output file\n",
                "\tFor documentation see: rubikrotation.pdf,\n",
                "\trubikrotationPL.pdf and rubikcube.pdf.\n\n";
        }
        elsif ( /^-v$/ || /^--version$/ ) { die $rubikversion; }
        elsif ( /^-i$/ ) {
            if ( !@ARGV ) {
                croak $showargs, "\tNo input file specified!\n", $usage;
            }
            else {
                $source_file = $ARGV[0], shift;
            }
        }
    }
}

```

```

    }
    elsif (/^-o$/) {
        if ( !@ARGV ) {
            croak $showargs, "\tNo output file specified!\n", $usage;
        }
        else {
            $out_file = $ARGV[0], shift;
        }
    }
    elsif (/^-w+$/) {
        croak $showargs, "\t$_: Illegal command line switch!\n", $usage;
    }
    else {
        croak $showargs,
            "\tmissing filenames or ? missing -i or -o switch!\n",
            $usage;
    }
}    # end of while
}
;    # end of else

#####

open( IN_FILE, "<$source_file" )
|| croak "\tCan't open source file: $source_file\n";
open( TeX_OUT_FILE, ">$out_file" )
|| croak "\tCan't open output file: $out_file\n";

## create error file (for append)
open( ERROR_OUT_FILE, ">>rubikstateERRORS.dat" )
|| croak "ERROR: can't open file rubikstateERRORS.dat\n";

## use dots for Perl messages (I have used dashes for LaTeX messages in the .sty)
## gprint sub prints its argument (message) to both the screen and to the TeX_OUT_FILE

gprint("");    # newline
gprint("...PERL process.....");
gprint("...script = rubikrotation.pl $version");

```

```

## setup global error parameters, so we can write all the errors to a file as an array
our %error      = ();      # setup an array for error messages (was %)
our $erroralert = "";      # error flag
our $errornumber = 0;      #set number of errors to zero

```

```

gprint("...reading the current rubik state (from File: $source_file)");

```

```

our $dataline      = "";
our $newdataline   = "";
our $rubikkeyword   = "";
our $rotationcommand = "";
our @data          = ();

```

```

our $Sequence      = "";    ## will hold the original (SHORT) sequence
our $rotationseqNEW = "";    ## will hold the LONG sequence
our $RotationSequenceName = "";

```

11

```

our $SequenceName = "";
our $SequenceShort = "";
our $SequenceLong = "";
our $SequenceInfo = "";

```

```

#-----inverse mode-----
# a keyword INVERSE or inverse in an infoblock <..>
# FLAG is set (line 400) in response to detecting an infloblock.
# A set FLAG triggers (a) reversing rotation sequence (line 484),
# and (b) inverting each rotation (to generate the inverse sequence).
# Here we define direction FLAG for the INVERSE sequence of rotations.
# The conditional test is in the SUB rotation
our $inverse      = "INV";
our $directionflag = "";

```

```

#-----

```

```

LINE: while (<IN_FILE>) {
    next LINE if /^#/;      #skip comments
    next LINE if /^%/.;     #skip comments

```

```

next LINE if /^$/;    #skip blank lines

print " \n TOP ----- (new line)\n\n";

$dataline = $_;      # grab the whole line as a string
chomp $dataline;     # remove the line-ending character

## now we can clean whitespace
$dataline = cleanstring($dataline);

#check syntax of the string
$rotationcommand = $dataline;    ## needed for error messages
CheckSyntax($dataline);

## form an array so we can process the (rubik)keywords.
@data = split( /,/ , $dataline );    # create an array called data

print " dataline array = @data\n";

#-----

## we have 10 fields (0--9)
## check for rubikkeyword= up,down,left,right,front,back,checkstate,rotation:
$rubikkeyword = $data[0];

if ( $rubikkeyword eq 'up' ) {
    gprint("...$dataline");
    $Ult[0] = $data[1], $Umt[0] = $data[2], $Urt[0] = $data[3],
    $Ulm[0] = $data[4], $Umm[0] = $data[5], $Urm[0] = $data[6],
    $Ulb[0] = $data[7], $Umb[0] = $data[8], $Urb[0] = $data[9];
    next LINE;
}

if ( $rubikkeyword eq 'down' ) {
    gprint("...$dataline");
    $Dlt[0] = $data[1], $Dmt[0] = $data[2], $Drt[0] = $data[3],
    $Dlm[0] = $data[4], $Dmm[0] = $data[5], $Drm[0] = $data[6],
    $Dlb[0] = $data[7], $Dmb[0] = $data[8], $Drb[0] = $data[9];
}

```

```

    next LINE;
}

if ( $rubikkeyword eq 'left' ) {
    gprint("...$dataline");
    $Llt[0] = $data[1], $Lmt[0] = $data[2], $Lrt[0] = $data[3],
    $Llm[0] = $data[4], $Lmm[0] = $data[5], $Lrm[0] = $data[6],
    $Llb[0] = $data[7], $Lmb[0] = $data[8], $Lrb[0] = $data[9];
    next LINE;
}

if ( $rubikkeyword eq 'right' ) {
    gprint("...$dataline");
    $Rlt[0] = $data[1], $Rmt[0] = $data[2], $Rrt[0] = $data[3],
    $Rlm[0] = $data[4], $Rmm[0] = $data[5], $Rrm[0] = $data[6],
    $Rlb[0] = $data[7], $Rmb[0] = $data[8], $Rrb[0] = $data[9];
    next LINE;
}

if ( $rubikkeyword eq 'front' ) {
    gprint("...$dataline");
    $Flt[0] = $data[1], $Fmt[0] = $data[2], $Frt[0] = $data[3],
    $Flm[0] = $data[4], $Fmm[0] = $data[5], $Frm[0] = $data[6],
    $Flb[0] = $data[7], $Fmb[0] = $data[8], $Frb[0] = $data[9];
    next LINE;
}

if ( $rubikkeyword eq 'back' ) {
    gprint("...$dataline");
    $Blt[0] = $data[1], $Bmt[0] = $data[2], $Brt[0] = $data[3],
    $Blm[0] = $data[4], $Bmm[0] = $data[5], $Brm[0] = $data[6],
    $Blb[0] = $data[7], $Bmb[0] = $data[8], $Brb[0] = $data[9];
    next LINE;
}

## if the rubikkeyword is 'checkstate'
## we just check the state and write the output data to a file.
if ( $rubikkeyword eq 'checkstate' ) {

```

```

    gprint("...");
    $rotationcommand = $dataline;    ## used in output message
    gprint("...command=$rotationcommand");
    checkstate();
    next LINE;
}

## IF the rubikkeyword is 'rotation'
##   we first check to see if the second argument=random.
##   ---if so, then we check that the third argument is an integer,
##   ---if it is an integer n --> random => random(n)
## ELSE   it must be a rotation sequence --> send elements to rotation sub.

#-----
if ( $rubikkeyword eq 'rotation' ) {
    gprint("...");

    # we now grab a copy of the dataline, and we shall use this
    # in the ErrorMessage SUB to indicate which command
    # an error is in.
    $rotationcommand = $dataline;    ## used in output message
    gprint("...command=$rotationcommand");

    # need to check that a second argument exists (else --> ErrorMessage).
    # ---should be either 'random',
    # ---or a macroname for a rotation sequence,
    # ---or the first element of a rotation sequence.

    if ( $data[1] eq "" ) {    # no second argument
        gprint(".*missing second argument");
        ErrorMessage("QUITTING PERL PROGRAM --- missing second argument:");
        ErrorMessage("--- ? bad rotation macro-name");
        quitprogram();
    }

    ##-----random-----
    ## if second argument = random,
    ## THEN we also need to check if third argument is an integer;

```

```

## if so -->random sub.
## if the 3rd argument is NOT an integer then reject line & get next input line
if ( $data[1] eq 'random' ) {
  if ( $data[2] =~ /\D/ ) {
    ## if true then cannot be a number (D matches for word elements)
    ErrorMessage("[ $data[2]] is not an integer");
    ## we reject the command (as bad syntax) and get next line
    next LINE;
  }
  else {
    ## the next argument is an integer (n), so we do n random rotations
    random( $data[2] );
    next LINE;
  }
} ## end of IF
#-----

else {
  # -----rotation sequence-----
  # the line must be a rotation sequence line, so send the sequence
  # to the rotation sub;

  # Note that a copy of the rotation command is already held in the
  # variable rotationcommand (see above). It is used in the
  # ErrorMessage SUB.

  #-----

# Process and remove any infoblocks if they exist
# infoblocks are text delimited by <...>
# The SUB cutinfoblock returns the name of the new revised string = newdataline,
# and also the contents of the infoblock = $SequenceInfo
# RubikRotation command uses infoblocks to carry keywords, eg INVERSE or inverse
# If several infoblocks exist, then we collect the contents into
# variable SequenceInfo, and separate them with a colon;

  infoblockcolon($dataline);
  $dataline = $newdataline;

```

```

$newdataline = "";          ## reset the variable

gprint("...dataline = $dataline");

## now pass the string to cutinfoblock
local @seq = ();

while ( ( index $dataline, '<' ) != -1 ) {

    cutinfoblock($dataline);

    # best to use the whole word <inverse> to avoid errors
    if ( $SequenceInfo =~ m/(inverse|INVERSE)/ ) {
        $directionflag = $inverse;
        print " FLAG set to = $inverse\n";
    }

    # append each infoblock to an array
    push @seq, $SequenceInfo;
    $dataline = $newdataline;
}

# finally, we join the seqInfo array into a string so we can print it
$SequenceInfo = join( "; ", @seq );

#-----
## there are now no more infoblocks, so we now look for repeat-blocks.

## reformulate any repeat blocks (,) --> {}; if they exist
## this is to allow us to process any repeat blocks as separate elements
# this sub returns the name of the new revised string.
while ( ( index $dataline, '(' ) != -1 ) {
    fixrepeatelement($dataline);
    $dataline = $newdataline;
    $newdataline = "";          ## reset the variable
}

## rename remaining dataline string as SequenceShortBrace

```

```

## since if there are any repeat blocks, they are now reformulated with braces
$SequenceShortBrace = $dataline;

## clean leading and trailing whitespace
$SequenceShortBrace = cleanstring($SequenceShortBrace);

##-----
## form a new array from $SequenceShortBrace (since we have changed the format
## slightly; ie some commands may have been reformulated as semicolons).
@data = split( /,/ , $SequenceShortBrace );

## need to remove keyword <rotation> (= first element in the array)
## removing it late like this is convenient for error checking purposes,
## as then the keyword 'rotation' is on the string
shift(@data);

## now need to recreate the string from the array @data for use later
## (as rotation keyword has been removed)

$SequenceShortBrace = join( ",", @data );

#-----create SequenceShort, so we can output it later-----

# since the 'rotating' keyword has been removed, we can
# replace any braces around repeat strings (if exist) and
# rename it as SequenceShort which we will output at the end.
if ( ( index $SequenceShortBrace, '{' ) != -1 ) {
    print " repairing braces--> ()\n";
    ## swap: BBook p 138--139
    $SequenceShortBrace =~ tr/{/(/;    # swap , --> ;
    $SequenceShortBrace =~ tr/\/)/;    # swap ( --> {
    $SequenceShortBrace =~ tr/;/,/;    # swap ) --> }
}

#rename to SequenceShort
$SequenceShort = $SequenceShortBrace;

print " SequenceShort = $SequenceShort\n";

```

```

##-----

## now we continue processing the array "data"

my $n = 0;    ##total no of array elements in "data"
$n = ( $#data + 1 );
print " processing rotation arguments: = @data (n= $n)\n";

# setup a loop counter (for use in the rotation sub)
# this is used to identify the first element (rcode)
# and used to grab [name] --> SequenceName.
our $jrcode = 0;

## -----check for state of direction flag-----
## FLAG defined in line 224.
## FLAG is set in line 400 on detecting <..> delimiters = infoblock
## if flag set (by inverse keyword) then reverse the sequence array
if ( $directionflag eq $inverse ) {

    # FLAG is set, so we need to inverse the array
    gprint("...directionFLAG set; reversing array...");

    # but before reversing, look at the first array element
    # to see if it is a square bracket element = NAME element
    # so check the first char to see if it is [
    if ( substr( $data[0], 0, 1 ) eq '[' ) {
        $SequenceName = $data[0];
        print " SequenceName (inv) = $SequenceName \n";
    }

    @data = reverse @data;
    print " processing rotation arguments: = @data (n= $n)\n";
}

# send each rotation element to the sub rotation()
print " CALLing SUB rotation\n";

foreach $element (@data) {

```

```

        ## clean leading and trailing white space
        $element = cleanstring($element);
        ## send element to rotation SUB
        rotation($element);
    }

    }    # end of else
}
;      # end of IF (rotation keyword)

#-----
## place any new keywords for processing here
##-----

};    ## end of while

## we have now finished reading in all the lines from the source file,
## and processing all the rotations etc,
## so we now just write the new cube state
## to the output file = TeX_OUT_FILE (so LaTeX can read it)
## plus any ErrorMessages
## -- all these are handled by the quitprogram sub

quitprogram();

##=====end of main=====

```

2.1 rotation

```

sub rotation {

    print " SUB rotation\n";

    ## here we process the array @data (from main) consisting of all
    ## the rotation commands associated with
    ## a single RubikRotation command -- the 'rotation' key word has already been removed
    ## so we start here with [name] if it exists.

```

```

##-----
## variables used in SUBs  rotation() and rubikmod()
## need to be defined outside the SUBs

$modnumber = -1;    #multiple associated with the char, eg D2 etc
$rotcode   = "";
$rotnumber = 0;

#-----
my @repeatcode = ();

my $m          = -1;
my $originalrcode = "";
my $j;          ## used with m below
my $numberofchars;  ## length of a string
my $nfrontchars;

##-----

## grab the rotation code passed to this sub from MAIN
my $rcode = $_[0];

## now we start a big loop processing each cs-element (= rcode),
## and collecting these elements into two cs-strings
## ($Sequence --> original string output as SHORT string (has codes like R2,L3 etc),
##  and $rotationseqNEW --> output as LONG string -- all short codes expanded)

## first, clean leading and trailing white space (eg between,  R ,)
$rcode = cleanstring($rcode);

## grab a copy of the element (char) for use if  m Mod4=0
$originalrcode = $rcode;

## increment the loop counter (initialised in MAIN)
$jrcode = $jrcode + 1;    ## increment rotation element (char) counter

## -----

```

```

## We look at the first character of each element in the sequence
## if an element has a leading [ then it is a label (not a rotation)
## should really be matched, but this is not checked for at present.
## If this is the case, then jump to next element in the array

## BUT if trailing comma is missing, then next rotation is included as part of the label
## so need to trap this and test: is first AND last char a sq bracket?

if ( ( substr( $rcode, 0, 1 ) =~ /\[/ )
    and ( substr( $rcode, -1 ) ) =~ /\]/ )
{

    gprint("...$rcode is a label OK");

    if ( $directionflag eq $inverse ) {

        # do nothing
    }

    else {
        ## if this 'label' is also the FIRST element, then label = name
        if ( $jrcode == 1 ) { $SequenceName = $rcode }
    }
    ;    # end of IF

    ## now get next rotation element
    next;
}
;    ## end of if

##-----

## the rcode must therefore be either a rotation code or a repeat-block.

##-----
## we have already replaced any repeat (,) with {};
## so we now check for elements with leading { and then expand them.

```

```

## Note that if there is NO comma before the {} of a {repeat block}, then
## the true repeat block will not be recognised by the
## usual test -- since the test is looking for a leading { etc.
## However, in this event, the string being handled (not a true element)
## will be processed as if it were a rotation, and an
## error will be thrown, so it will get picked up OK.

if ( substr( $rcode, 0, 1 ) =~ /\{/ ) {
    print " repeat block found          = $rcode \n";

    ## since we now want to send each rotation element in the repeat block to
    ## the rotation sub, we need to replace any ; with commas
    ## therefore translate ; --> , but retain the {}
    $rcode =~ tr/;/,/;

    print " repeat block reformulated = $rcode \n";

    #-----log file message-----
    ## log file: we want to show the repeat string in the users original form
    ## so we translate it back to the user's orig form {,} --> (,)
    $origrcode = $rcode;
    $origrcode =~ tr/\{/(/;
    $origrcode =~ tr/\}/)/;
    gprint("...Expanding: $origrcode ...");

    #-----

    #-----
    ## expand the code in the repeat block
    print " CALLing SUB: repeat($rcode)\n";

    repeat($rcode);    # this expands the repeated elements in the block

    ## this sub returns the expanded form as $insert
    $expanded_repeatcode = $insert;

    print " expanded_repeatcode = $expanded_repeatcode\n\n";

```

```

#-----

#-----
# process each new element in the expanded_repeatcode --> rotation
# make expanded_repeatcode into an array, and send each element on

@repeatcode = split( /,/ , $expanded_repeatcode );

## -----check for direction flag-----
## if flag set then reverse the array
if ( $directionflag eq $inverse ) { @repeatcode = reverse @repeatcode }

# send each element to rotation SUB for processing
print " CALLing SUB rotation\n";
foreach $E (@repeatcode) {
    print " sending repeat element $E to rotation SUB\n";
    rotation($E);
}

# when this foreach is finished, then get next rotation element from
# the original @data array (see foreach.. near end of MAIN)
next;

}
;    ## end of if

##=====

## if an element has got this far, it must be a single rotation code
## (maybe with a trailing digit), so it needs processing as a rotation
## and appending the code to what will become the SequenceLONG string.

##-----

## CALL the sub rubikmod to process the rotation element,
## and to return the front code (= $rotcode), orig no = $rotnumber,
## and mod4 value (= $modnumber).

```

```

rubikmod($rcode);

## update rcode <--- rotcode   (returned by the SUB rubikmod() )
## collect  $m <--- modnumber (returned by the SUB rubikmod() )
$rcode = $rotcode;
$m      = $modnumber;

## we collect all the new versions of rcode into a cs-string = $SequenceLong
## which will finally be output as the LONG string

#-----
# check with directionflag
if ( $directionflag eq $inverse ) { $rcode = inverse($rcode) }

#-----

##-----
if ( $m == 0 ) {
    ## do NOT implement the rotation code, and do NOT append to SequenceLong
    ## print the /original/ rcode (eg R4, or D0 etc)
    gprint(
"..*rotation , $originalrcode, ERROR ($rotnumber = 0 mod 4) not implemented"
    );
    ErrorMessage(
        "$originalrcode, -- ($rotnumber = 0 mod 4) not implemented");
    next;
}

if ( $m == 1 ) {
    if ( $rotnumber >= 5 ) {
        gprint("...Expanding $originalrcode ($rotnumber = $m mod 4) ...");
    }
    $SequenceLong = $SequenceLong . $rcode . ",";
}
else {
    # m = 2 or 3
    if ( $rotnumber >= 5 ) {
        gprint("...Expanding $originalrcode ($rotnumber = $m mod 4) ...");
    }
}

```

```

    }
    else { gprint("...Expanding $originalrcode ...") }

    for ( $j = 1 ; $j <= $m ; $j++ )
    {
        ## append rcode m times to sequenceLONG
        $SequenceLong = $SequenceLong . $rcode . ",";
    }
}
;      ## end of else

##-----

## if single trailing digit present,
##     then we implement the rotation command m times.
## if more than one trailing digit
##     then the error is trapped at the end (as frontstring will not be recognised
##     ie will not be in the following list, and hence will be trapped as an error, eg R3)

if ( $rcode eq "L" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation L,  OK (= Lp3)");
        &rrL;
    }
}
elseif ( $rcode eq "Lp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Lp,  OK");
        &rrLp;
    }
}
elseif ( $rcode eq "Lw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Lw,  OK (= Lp3 + Srp)");
        &rrLw;
    }
}
elseif ( $rcode eq "Lwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {

```

```

        gprint("...rotation Lwp, OK (= Lp + Sr)");
        &rrLwp;
    }
}
elseif ( $rcode eq "Ls" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ls, OK (= L + Rp)");
        &rrLs;
    }
}
elseif ( $rcode eq "Lsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Lsp, OK (= Lp + R)");
        &rrLsp;
    }
}
elseif ( $rcode eq "La" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation La, OK (= L + R)");
        &rrLa;
    }
}
elseif ( $rcode eq "Lap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Lap, OK (= Lp + Rp)");
        &rrLap;
    }
}
####
elseif ( $rcode eq "R" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) { gprint("...rotation R, OK"); &rrR }
}
elseif ( $rcode eq "Rp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rp, OK (= R3)");
        &rrRp;
    }
}
}

```

```

elseif ( $rcode eq "Rw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rw, OK (= R + Sr)");
        &rrRw;
    }
}
elseif ( $rcode eq "Rwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rwp, OK (= Rp + Srp)");
        &rrRwp;
    }
}
elseif ( $rcode eq "Rs" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rs, OK (= R + Lp)");
        &rrRs;
    }
}
elseif ( $rcode eq "Rsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rsp, OK (= Rp + L)");
        &rrRsp;
    }
}
elseif ( $rcode eq "Ra" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ra, OK (= R + L)");
        &rrRa;
    }
}
elseif ( $rcode eq "Rap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rap, OK (= Rp + Lp)");
        &rrRap;
    }
}
####
elseif ( $rcode eq "U" ) {

```

```

        for ( $j = 1 ; $j <= $m ; $j++ ) { gprint("...rotation U,   OK"); &rrU }
    }
    elseif ( $rcode eq "Up" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Up,   OK (= U3)");
            &rrUp;
        }
    }
    elseif ( $rcode eq "Uw" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Uw,   OK (= U + Su)");
            &rrUw;
        }
    }
    elseif ( $rcode eq "Uwp" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Uwp, OK (= Up + Sup)");
            &rrUwp;
        }
    }
    elseif ( $rcode eq "Us" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Us,   OK (= U + Dp)");
            &rrUs;
        }
    }
    elseif ( $rcode eq "Usp" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Usp, OK (= Up + D)");
            &rrUsp;
        }
    }
    elseif ( $rcode eq "Ua" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Ua,   OK (= U + D)");
            &rrUa;
        }
    }
}

```

```

elseif ( $rcode eq "Uap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Uap, OK (= Up + Dp)");
        &rrUap;
    }
}
####
elseif ( $rcode eq "D" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation D, OK (= Dp3)");
        &rrD;
    }
}
elseif ( $rcode eq "Dp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Dp, OK ");
        &rrDp;
    }
}
elseif ( $rcode eq "Dw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Dw, OK (= Dp3 + Sup)");
        &rrDw;
    }
}
elseif ( $rcode eq "Dwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Dwp, OK (= Dp + Su)");
        &rrDwp;
    }
}
elseif ( $rcode eq "Ds" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ds, OK (= D + Up)");
        &rrDs;
    }
}
elseif ( $rcode eq "Dsp" ) {

```

```

        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Dsp, OK (= Dp + U)");
            &rrDsp;
        }
    }
    elseif ( $rcode eq "Da" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Da, OK (= D + U)");
            &rrDa;
        }
    }
    elseif ( $rcode eq "Dap" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Dap, OK (= Dp + Up)");
            &rrDap;
        }
    }
    #####
    elseif ( $rcode eq "F" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) { gprint("...rotation F, OK"); &rrF }
    }
    elseif ( $rcode eq "Fp" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Fp, OK (= F3)");
            &rrFp;
        }
    }
    elseif ( $rcode eq "Fw" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Fw, OK (= F + Sf)");
            &rrFw;
        }
    }
    elseif ( $rcode eq "Fwp" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Fwp, OK (= Fp + Sfp)");
            &rrFwp;
        }
    }

```

```

}
elseif ( $rcode eq "Fs" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fs,  OK (= F + Bp)");
        &rrFs;
    }
}
elseif ( $rcode eq "Fsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fsp, OK (= Fp + B)");
        &rrFsp;
    }
}
elseif ( $rcode eq "Fa" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fa,  OK (= F + B)");
        &rrFa;
    }
}
elseif ( $rcode eq "Fap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fap, OK (= Fp + Bp)");
        &rrFap;
    }
}
####
elseif ( $rcode eq "B" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation B,   OK (= Fp3)");
        &rrB;
    }
}
elseif ( $rcode eq "Bp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bp,  OK");
        &rrBp;
    }
}

```

```

elseif ( $rcode eq "Bw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bw, OK (= Fp3 + Sfp)");
        &rrBw;
    }
}
elseif ( $rcode eq "Bwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bwp, OK (= Fp + Sf)");
        &rrBwp;
    }
}
elseif ( $rcode eq "Bs" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bs, OK (= B + Fp)");
        &rrBs;
    }
}
elseif ( $rcode eq "Bsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bsp, OK (= Bp + F)");
        &rrBsp;
    }
}
elseif ( $rcode eq "Ba" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ba, OK (= B + F)");
        &rrBa;
    }
}
elseif ( $rcode eq "Bap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bap, OK (= Bp + Fp)");
        &rrBap;
    }
}

#### -----

```

```

#### inner-slice (= middle slice)
## need to include MES (middle slice) notation
elseif ( $rcode eq "M" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation M,   OK (= Sl) ");
        &rrSl;
    }
}
elseif ( $rcode eq "Mp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Mp,  OK (= Sr) ");
        &rrSr;
    }
}
elseif ( $rcode eq "E" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation E,   OK (= Sd) ");
        &rrSd;
    }
}
elseif ( $rcode eq "Ep" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ep,  OK (= Su) ");
        &rrSu;
    }
}
elseif ( $rcode eq "S" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation S,   OK (= Sf) ");
        &rrSf;
    }
}
elseif ( $rcode eq "Sp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sp,  OK (= Sb) ");
        &rrSb;
    }
}

```

```

}

#### middle slice rotations (Singmaster)
elseif ( $rcode eq "Su" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Su, OK ");
        &rrSu;
    }
}
elseif ( $rcode eq "Sup" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sup, OK (= Su3)");
        &rrSup;
    }
}
elseif ( $rcode eq "Sd" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sd, OK (= Sup)");
        &rrSd;
    }
}
elseif ( $rcode eq "Sdp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sdp, OK (= Su)");
        &rrSdp;
    }
}
elseif ( $rcode eq "Sl" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sl, OK (= Srp)");
        &rrSl;
    }
}
elseif ( $rcode eq "Slp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Slp, OK (= Sr)");
        &rrSlp;
    }
}

```

```

}
elseif ( $rcode eq "Sr" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sr, OK");
        &rrSr;
    }
}
elseif ( $rcode eq "Srp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Srp, OK (= Sr3)");
        &rrSrp;
    }
}
elseif ( $rcode eq "Sf" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sf, OK");
        &rrSf;
    }
}
elseif ( $rcode eq "Sfp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sfp, OK (= Sf3)");
        &rrSfp;
    }
}
elseif ( $rcode eq "Sb" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sb, OK (= Sfp)");
        &rrSb;
    }
}
elseif ( $rcode eq "Sbp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sbp, OK (= Sf)");
        &rrSbp;
    }
}
}

```

```

## need to include Jaap Puzzles website for middle slice notation (Lm, Lmp)
## also include Randelshofer website middle slice notation (ML,MLp..)

elseif ($rcode eq "ML"
        or $rcode eq "MRp"
        or $rcode eq "Lm"
        or $rcode eq "Rmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Lm = M = Sl) ");
        &rrSl;
    }
}

elseif ($rcode eq "MR"
        or $rcode eq "MLp"
        or $rcode eq "Rm"
        or $rcode eq "Lmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Rm = Mp = Sr) ");
        &rrSr;
    }
}

elseif ($rcode eq "MU"
        or $rcode eq "MDp"
        or $rcode eq "Um"
        or $rcode eq "Dmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Um = Ep = Su) ");
        &rrSu;
    }
}

elseif ($rcode eq "MD"
        or $rcode eq "MUp"

```

```

        or $rcode eq "Dm"
        or $rcode eq "Ump" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Dm = E = Sd) ");
        &rrSd;
    }
}

elseif ($rcode eq "MF"
        or $rcode eq "MBp"
        or $rcode eq "Fm"
        or $rcode eq "Bmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Fm = S = Sf) ");
        &rrSf;
    }
}

elseif ($rcode eq "MB"
        or $rcode eq "MFp"
        or $rcode eq "Bm"
        or $rcode eq "Fmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Bm = Sp = Sb) ");
        &rrSb;
    }
}

##-----

#### double outer slice (wide) notation
#### need to include Randelshofer TL, Tlp double outer slice notation
#### (equiv to the w wide notation)
elseif ( $rcode eq "TL" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TL,  OK (= Lw = Lp3 + Srp)");
    }
}

```

```

        &rrLw;
    }
}
elseif ( $rcode eq "TLp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TLp, OK (= Lwp = Lp + Sr)");
        &rrLwp;
    }
}
elseif ( $rcode eq "TR" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TR, OK (= Rw = R + Sr)");
        &rrRw;
    }
}
elseif ( $rcode eq "TRp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TRp, OK (= Rwp = Rp + Srp)");
        &rrRwp;
    }
}
elseif ( $rcode eq "TU" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TU, OK (= Uw = U + Su)");
        &rrUw;
    }
}
elseif ( $rcode eq "TUp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TUp, OK (= Uwp = Up + Sup)");
        &rrUwp;
    }
}
elseif ( $rcode eq "TD" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TD, OK (= Dw = Dp3 + Sup)");
        &rrDw;
    }
}

```

```

}
elseif ( $rcode eq "TDp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TDp, OK (= Dwp = Dp + Su)");
        &rrDwp;
    }
}
elseif ( $rcode eq "TF" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TF, OK (= Fw = F + Sf)");
        &rrFw;
    }
}
elseif ( $rcode eq "TFp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TFp, OK (= Fwp = Fp + Sfp)");
        &rrFwp;
    }
}
elseif ( $rcode eq "TB" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TB, OK (= Bw = Fp3 + Sfp)");
        &rrBw;
    }
}
elseif ( $rcode eq "TBp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TBp, OK (= Bwp = Fp + Sf)");
        &rrBwp;
    }
}

## -----
## opposite slice notation of Randelshofer (SR, SRp) (= standard Rs, Rsp)
## opposite outer slices rotated in SAME direction as the FACE

elseif ( $rcode eq "SL" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {

```

```

        gprint("...rotation $rcode, OK (= Ls = L + Rp)");
        &rrLs;
    }
}
elseif ( $rcode eq "SLp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= Lsp = Lp + R)");
        &rrLsp;
    }
}

elseif ( $rcode eq "SR" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= Rs = R + Lp)");
        &rrRs;
    }
}

elseif ( $rcode eq "SRp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= Rsp = Rp + L)");
        &rrRsp;
    }
}

elseif ( $rcode eq "SU" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= Us = U + Dp)");
        &rrUs;
    }
}

elseif ( $rcode eq "SUP" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= Usp = Up + D)");
        &rrUsp;
    }
}

elseif ( $rcode eq "SD" ) {

```

```

        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation $rcode, OK (= Ds = D + Up)");
            &rrDs;
        }
    }
    elseif ( $rcode eq "SDp" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation $rcode, OK (= Dsp = Dp + U)");
            &rrDsp;
        }
    }

    elseif ( $rcode eq "SF" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation $rcode, OK (= Fs = F + Bp)");
            &rrFs;
        }
    }
    elseif ( $rcode eq "SFp" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation $rcode, OK (= Fsp = Fp + B)");
            &rrFsp;
        }
    }

    elseif ( $rcode eq "SB" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation $rcode, OK (= Bs = B + Fp)");
            &rrBs;
        }
    }
    elseif ( $rcode eq "SBp" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation $rcode, OK (= Bsp = Bp + F)");
            &rrBsp;
        }
    }
}

```

```

## -----

## whole cube rotations
## need to include x,y,z (upper and lowercase) and also u,d,l,r,f,b (lowercase only) equivalents
elseif ( $rcode eq "X" or $rcode eq "x" or $rcode eq "r" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= x = R + Sr + Lp)");
        &rrR;
        &rrSr;
        &rrLp;
    }
}
elseif ( $rcode eq "Xp" or $rcode eq "xp" or $rcode eq "l" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= xp = Rp + Srp + L)");
        &rrRp;
        &rrSrp;
        &rrL;
    }
}
elseif ( $rcode eq "Y" or $rcode eq "y" or $rcode eq "u" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= y = U + Su + Dp)");
        &rrU;
        &rrSu;
        &rrDp;
    }
}
elseif ( $rcode eq "Yp" or $rcode eq "yp" or $rcode eq "d" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= yp = Up + Sup + D)");
        &rrUp;
        &rrSup;
        &rrD;
    }
}
elseif ( $rcode eq "Z" or $rcode eq "z" or $rcode eq "f" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {

```

```

        gprint("...rotation $rcode, OK (= z = F + Sf + Bp)");
        &rrF;
        &rrSf;
        &rrBp;
    }
}
elseif ( $rcode eq "Zp" or $rcode eq "zp" or $rcode eq "b" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= zp = Fp + Sfp + B)");
        &rrFp;
        &rrSfp;
        &rrB;
    }
}

## more whole cube notation
## need to include Jaap website whole cube Lc notation
## also include Randelshofer C notation (CL, CLp.)

elseif ($rcode eq "CL"
        or $rcode eq "CRp"
        or $rcode eq "Lc"
        or $rcode eq "Rcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Lc = xp = Rp + Srp + L)");
        &rrRp;
        &rrSrp;
        &rrL;
    }
}

elseif ($rcode eq "CR"
        or $rcode eq "CLp"
        or $rcode eq "Rc"
        or $rcode eq "Lcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {

```

```

        gprint("...rotation $rcode OK (= Rc = x = R + Sr + Lp)");
        &rrR;
        &rrSr;
        &rrLp;
    }
}

elseif ($rcode eq "CU"
        or $rcode eq "CDp"
        or $rcode eq "Uc"
        or $rcode eq "Dcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Uc = y = U + Su + Dp)");
        &rrU;
        &rrSu;
        &rrDp;
    }
}

elseif ($rcode eq "CD"
        or $rcode eq "CUp"
        or $rcode eq "Dc"
        or $rcode eq "Ucp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Dc = yp = Up + Sup + D)");
        &rrUp;
        &rrSup;
        &rrD;
    }
}

elseif ($rcode eq "CF"
        or $rcode eq "CBp"
        or $rcode eq "Fc"
        or $rcode eq "Bcp" )
{

```

```

        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation $rcode OK (= Fc = z = F + Sf + Bp)");
            &rrF;
            &rrSf;
            &rrBp;
        }
    }

elseif ($rcode eq "CB"
        or $rcode eq "CFp"
        or $rcode eq "Bc"
        or $rcode eq "Fcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Bc = zp = Fp + Sfp + B)");
        &rrFp;
        &rrSfp;
        &rrB;
    }
}

## -----

## check empty string --> missing rotation
elseif ( $rcode eq "" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint(".*rotation , $rcode, ERROR ? typo or missing rotation");
        ErrorMessage(", $rcode, -- ? typo or missing rotation");
    }
}

## finally -----
else {
    ## to fall this far then the rotation (char) must be undefined
    ## but before we can send these rotation code strings out in ErrorMessages
    ## we need to check that they are in the original format.
    ## ie., do not have any {;} chars etc. If they do, then we need to
    ## translate them back, ie {;} --> (,) etc

```

```

## we use 'originalcode' in the ErrorMessage because the user needs to be
## shown the 'bad' code as it was originally input by the RubikRotation{} command.

## check for code with { ; } and restore to normal syntax
if ( $rcode =~ m/(\{|;\}|\}\{|\}\{)/ ) {
    $rcode          = restorebrackets($rcode);
    $originalrcode = restorebrackets($originalrcode);
}

if ( $rcode =~ m/(\(|\)|\[[\]])/ ) {
    gprint(
"..*rotation $rcode ERROR -- code not known ? missing comma or nested brackets"
    );
    ErrorMessage(
"$originalrcode -- code not known ? missing comma or nested brackets"
    );
}
else {
    gprint(
"..*rotation $rcode ERROR -- code not known ? typo or missing comma"
    );
    ErrorMessage(
"$originalrcode -- code not known ? typo or missing comma");
}

# missing comma after random is a common error
if ( $originalrcode =~ m/random/i ) {
    ErrorMessage("$originalrcode -- ? missing comma after random");
}

#-----
next;
}
;    #end of else

next;
}    # end of sub

```

```
#=====
```

2.2 random

```
sub random {

    print " SUB random\n";

    ## scramble randomly using n rotations
    ## example command = RubikRotation{random,74}
    ## if no n given (second argument = ""), then use default n=50
    ## if second argument is some string (not integer) then --> ERROR
    ##
    ## assign numbers to the minimal set of rotations to be used using a hash array list
    ## (perl 5 book page 68)
    ## ? maybe we should only use the 18 rotations mentioned in Rokicki 2013 paper?
    ## but here I have included all the S ones too.

    my @rrlist = (
        "U", "Up", "Um", "Ump", "D", "Dp", "Dm", "Dmp",
        "L", "Lp", "Lm", "Lmp", "R", "Rp", "Rm", "Rmp",
        "F", "Fp", "Fm", "Fmp", "B", "Bp", "Bm", "Bmp"
    );

    my $rrlistnumber = $#rrlist;
    print " rrlistnumber = $rrlistnumber\n";

    # these are numbered 0--$rrlistnumber,

    ## let default no of random rotations for scrambling = 50
    my $defaultn = 50;
    my $maxn      = 200;

    ## grab the integer passed from the random() command in main
    my $s = $_[0];

    if ( $s >= $maxn ) {
```

```

    $s = $maxn;
    gprint(".*WARNING: maximum n = 200");
    ErrorMessage("random: max n =200 (n=200 was used)");
}
elseif ( $s == 0 ) {
    $s = $defaultn;
    gprint(".*WARNING: integer = 0 or missing: using default value 50");
    ErrorMessage("warning: integer n missing or invalid (n=50 was used)");
}

my @rr;    ## array to hold all the random rotations
print " randomising the available rotations\n";

## set the seed for the randomisation (BlackBook p 235)
srand;

## now select s numbers at random (with replacement) from range 0--listnumber+1
## Since we are using int(rand x), and using nos from 0--lastindex number,
## then max rand vaue = (lastindexnumber -1).99999, the integer of which
## = (lastindexnumber -1). Therefore we need to use the range 0--(lastindexnumber+1)
## in order to randomise all possibilities on our list.

my $j;

for ( $j = 1 ; $j <= $s ; $j = $j + 1 ) {
    my $p = int( rand( $rrlistnumber + 1 ) );
    print "Rotation = $p, $rrlist[$p] \n";
    ## push rotation code $rrlist[$p] on to END of array @rr
    push( @rr, $rrlist[$p] );
}

## we assume the user is starting from a solved cube (ie use the state given by user)
gprint("...scrambling Rubik cube using $s random rotations");

## now send the array off to the rotation sub

my $E;
```

```

        foreach $E (@rr) { rotation($E) }

}    ##end of sub

#=====

```

2.3 writestate

```

sub writestate {

    print " SUB writestate\n";

    ## this  writes the final state to the TeX_OUT_FILE (= rubikstateNEW.dat) will be read by latex.

    print( TeX_OUT_FILE "%\% ...output datafile=$out_file\n" );
    print( TeX_OUT_FILE
        "%\% ...PERL script=rubikrotation.pl version $version\n" );
    print( TeX_OUT_FILE
        "\\typeout{...writing new Rubik state to file $out_file}\%\n" );
    print( TeX_OUT_FILE
        "\\RubikFaceUp\\{$Ult[0]\\}\\{$Umt[0]\\}\\{$Urt[0]\\}\\{$Ulm[0]\\}\\{$Umm[0]\\}\\{$Urm[0]\\}\\{$Ulb[0]\\}\\{$Umb[0]\\}\\{$Urb[0]\\}\%\n"
    );
    print( TeX_OUT_FILE
        "\\RubikFaceDown\\{$Dlt[0]\\}\\{$Dmt[0]\\}\\{$Drt[0]\\}\\{$Dlm[0]\\}\\{$Dmm[0]\\}\\{$Drm[0]\\}\\{$Dlb[0]\\}\\{$Dmb[0]\\}\\{$Drb[0]\\}\%\n"
    );
    print( TeX_OUT_FILE
        "\\RubikFaceLeft\\{$Llt[0]\\}\\{$Lmt[0]\\}\\{$Lrt[0]\\}\\{$Llm[0]\\}\\{$Lmm[0]\\}\\{$Lrm[0]\\}\\{$Llb[0]\\}\\{$Lmb[0]\\}\\{$Lrb[0]\\}\%\n"
    );
    print( TeX_OUT_FILE
        "\\RubikFaceRight\\{$Rlt[0]\\}\\{$Rmt[0]\\}\\{$Rrt[0]\\}\\{$Rlm[0]\\}\\{$Rmm[0]\\}\\{$Rrm[0]\\}\\{$Rlb[0]\\}\\{$Rmb[0]\\}\\{$Rrb[0]\\}\%\n"
    );
    print( TeX_OUT_FILE
        "\\RubikFaceFront\\{$Flt[0]\\}\\{$Fmt[0]\\}\\{$Frt[0]\\}\\{$Flm[0]\\}\\{$Fmm[0]\\}\\{$Frm[0]\\}\\{$Flb[0]\\}\\{$Fmb[0]\\}\\{$Frb[0]\\}\%\n"
    );
    print( TeX_OUT_FILE
        "\\RubikFaceBack\\{$Blt[0]\\}\\{$Bmt[0]\\}\\{$Brt[0]\\}\\{$Blm[0]\\}\\{$Bmm[0]\\}\\{$Brm[0]\\}\\{$Blb[0]\\}\\{$Bmb[0]\\}\\{$Brb[0]\\}\%\n"
    );
}

```

```

##-----RWDN 2016---create four  new holder commands for separate strings-----

##  these four names are defined in the rubikrotation.sty file so they can be renewed etc
## SequenceInfo
## SequenceName
## SequenceShort
## SequenceLong

## ----RWDN  25 Sept 2016 -----
## now remove the first and last chars of [name] to output just  NAME without [ and ]

## initialise some variables we shall need
    $numberofcharsinstring = 0;
    $nmiddlecharsinstring  = 0;

##-----SequenceName-----

## the SequenceName currently includes the [...]
## need to remove the [] before sending it to LaTeX,
## so need to detect when NAME string itself is empty, eg []
## so create a variable:
    $SequenceNameNew = "";

    $numberofcharsinstring = length $SequenceName;

##  NEED to create error message if [] and  empty string etc

if ( $numberofcharsinstring <= 2 ) { $SequenceNameNew = $SequenceName }
else {

    $nmiddlecharsinstring = ( $numberofcharsinstring - 2 );
    ## reassign the string without first and last chars
    ### format of substr = (origstring, start possn, no of chars to use)
    $SequenceNameNew = substr( $SequenceName, 1, $nmiddlecharsinstring );
}

print( TeX_OUT_FILE

```

```

        "\\renewcommand\\SequenceName\\{$SequenceNameNew\\}\\%\\n" );

print( TeX_OUT_FILE "\\typeout{...SequenceName = $SequenceNameNew}\\%\\n" );

#-----

#-----SequenceInfo-----
## we need to preserve any {} structures in the info string,
## so change { } --> [ ] since
## otherwise they will disappear or cause error when printed in LaTeX

$SequenceInfo =~ tr/{/[/;    ## swap { --> [
$SequenceInfo =~ tr/}/\]/;    ## swap } --> ]

print( TeX_OUT_FILE "\\renewcommand\\SequenceInfo\\{$SequenceInfo\\}\\%\\n" );

print( TeX_OUT_FILE "\\typeout{...SequenceInfo = $SequenceInfo}\\%\\n" );

#-----

##-----SequenceShort-----
## generated in MAIN
## SequenceShort = original argument of \RubikRotation{} /without/ any infoblocks
## therefore it may contain square brackets

print( TeX_OUT_FILE "\\renewcommand\\SequenceShort\\{$SequenceShort\\}\\%\\n" );

print( TeX_OUT_FILE "\\typeout{...SequenceShort = $SequenceShort}\\%\\n" );
##-----

## now prepare the new LONG rotation sequence for output =(LONG sequence + NO NAME)
## BUT before outputting the string, we need to remove the terminal comma

$numberOfcharsinstring = length $SequenceLong;
$nfrontcharsinstring = $numberOfcharsinstring - 1;
## reassign the string except the terminal comma
$SequenceLong = substr( $SequenceLong, 0, $nfrontcharsinstring );

```

```

#-----
print( TeX_OUT_FILE "\\renewcommand\\SequenceLong\\{$SequenceLong\\}\\%\\n" );

print( TeX_OUT_FILE "\\typeout{...SequenceLong = $SequenceLong}\\%\\n" );

##-----

## now include any error messages generated
## (these are all in an array waiting to be printed out)

if ( $erroralert eq "YES" ) {
    ## write errors to a separate file (just for errors---we append the errors to end of file)
    ## the error file (rubikstateERRORS.dat) was created by the TeX file
    my $ne;    #number of errors
    $ne = $#error
        ;    ## number of errors= largest index num since we started at zero

    ## do not attach error to a <checkstate> command, since we really want
    ## to see the checkstate errors (in the ERROR file) printed AFTER the 'rotation' command.
    if ( $rotationcommand eq "checkstate" ) { }
    else { print( ERROR_OUT_FILE "*ERR cmd= $rotationcommand\\n" ) }

    ## last index number or array = $#arrayname (Black book p 62)
    my $k;

    for ( $k = 0 ; $k <= $ne ; $k = $k + 1 ) {

        ## restore correct brackets etc before outputting to Latex
        my $errorstring = $error[$k];
        $errorstring = restorebrackets($errorstring);

        print( TeX_OUT_FILE "\\typeout{$errorstring}\\%\\n" );
        print( ERROR_OUT_FILE "$errorstring\\n" );

    };    # end of for
}
;    # end of IF
print " Perl output file written OK\\n";

```

```
}    #end of sub
```

```
#=====
```

2.4 ErrorMessage

This subroutine places error messages which are generated into an array, the elements of which will be written (later) to the file rubikstateERRORS.dat.

```
sub ErrorMessage {
```

```
    ## writes the argument as a standard error message to out file
```

```
    my $errormess = $_[0];    ## parameter passed to sub
```

```
    ## restore correct brackets etc before outputting to Latex
    $errormess = restorebrackets($errormess);
```

```
    $erroralert = "YES";      ## set error alert flag (for use in out message)
    $error[$errornumber] = "*ERR    $errormess";
    $errornumber++;          ## increment number
```

```
}
```

```
#=====
```

2.5 gprint

```
sub gprint {
```

```
    ## prints argument (comments) to screen and also to TeX_OUT_FILE.
    ## The typeout commands will find its way into the log file when read by latex
    ## Important to include trailing % for messages written to the TeX_OUT_FILE
    ##    to stop extra <spaces> being seen by TeX.
```

```

    my $gmess = $_[0];
    print "$gmess\n";
    print( TeX_OUT_FILE "\\typeout{$gmess}\\%\n" );
}

```

```

#=====

```

2.6 checkstate

```

sub checkstate {

    print " SUB checkstate\n";

    ## only a simple check -- to see if wrong no of colours being used etc
    ## uses the cubie colours as used by rubikcube package= ROYGBWX

    gprint("...checking state of cube");

    my @cubies = (
        $Ult[0], $Umt[0], $Urt[0], $Ulm[0], $Umm[0], $Urm[0], $Ulb[0],
        $Umb[0], $Urb[0], $Dlt[0], $Dmt[0], $Drt[0], $Dlm[0], $Dmm[0],
        $Drm[0], $Dlb[0], $Dmb[0], $Drb[0], $Llt[0], $Lmt[0], $Lrt[0],
        $Llm[0], $Lmm[0], $Lrm[0], $Llb[0], $Lmb[0], $Lrb[0], $Rlt[0],
        $Rmt[0], $Rrt[0], $Rlm[0], $Rmm[0], $Rrm[0], $Rlb[0], $Rmb[0],
        $Rrb[0], $Flt[0], $Fmt[0], $Frt[0], $Flm[0], $Fmm[0], $Frm[0],
        $Flb[0], $Fmb[0], $Frb[0], $Blt[0], $Bmt[0], $Brt[0], $Blm[0],
        $Bmm[0], $Brm[0], $Blb[0], $Bmb[0], $Brb[0]
    );

    my $R = 0, my $O = 0, my $Y = 0, my $G = 0, my $B = 0, my $W = 0, my $X = 0;

    my $cubiecolour = "";

    foreach $cubiecolour (@cubies) {
        if ( $cubiecolour eq R ) { $R = $R + 1 }
        elsif ( $cubiecolour eq O ) { $O = $O + 1 }
        elsif ( $cubiecolour eq Y ) { $Y = $Y + 1 }
    }
}

```

```

        elsif ( $cubiecolour eq G ) { $G = $G + 1 }
        elsif ( $cubiecolour eq B ) { $B = $B + 1 }
        elsif ( $cubiecolour eq W ) { $W = $W + 1 }
        elsif ( $cubiecolour eq X ) { $X = $X + 1 }
        else {
            gprint("...*cubie-colour counting ERROR");
        }
    }

    my $cubiesum = 0;
    $cubiesum = $R + $O + $Y + $G + $B + $W + $X;
    gprint(
"...cubiesum = $cubiesum (Red=$R, Or=$O, Ye=$Y, Gr=$G, Bl=$B, Wh=$W, X=$X)"
    );

    # only generate ErrorMessage if n>9 (as may be using a Grey cube)
    if ( $cubiesum != 54 ) {
        ErrorMessage("cubiesum not = 54");
        gprint("...*cubiesum not = 54");
    }

    if ( $R > 9 ) {
        ErrorMessage("red cubies > 9 (=$R)");
        gprint("...*red cubies > 9 (=$R)");
    }

    if ( $O > 9 ) {
        ErrorMessage("orange cubies > 9 (=$O)");
        gprint("...*orange cubies > 9 (=$O)");
    }

    if ( $Y > 9 ) {
        ErrorMessage("yellow cubies > 9 (=$Y)");
        gprint("...*yellow cubies > 9 (=$Y)");
    }

    if ( $G > 9 ) {
        ErrorMessage("green cubies > 9 (=$G)");
    }

```

```

        gprint("...*green cubies > 9 (=$G)");
    }

    if ( $B > 9 ) {
        ErrorMessage("blue cubies > 9 (=$B)");
        gprint("...*blue cubies > 9 (=$B)");
    }

    if ( $W > 9 ) {
        ErrorMessage("white cubies > 9 (=$W)");
        gprint("...*white cubies > 9 (=$W)");
    }

    if ( $X == 54 ) {
        ErrorMessage("no colours allocated (X=54)");
        gprint("...*no colours allocated (X=54)");
    }

    print " done\n\n";
}

```

2.7 rr-overview of rotation subs

```

## The following 9 subs (90 degree rotation transformations) are used
##   to generate all the rotations used in the 'rotation sub' (see above).
## Each of these is a permutation for both colours and numbers of the cubie facelets.
## The following 9 subroutines are named as follows:
##   (about X-axis) rrR, rrSr, rrLp
##   (about Y-axis) rrU, rrSu, rrDp
##   (about Z-axis) rrF, rrSf, rrBp
##   see the rubikcube package documentation for full details regarding rotation notation and commands.
##
## METHOD & NOTATION
## Each sub (below) starts by making an array[0] for the cubie colour
##   and an array[1] for the cubie number.
## Each of the face rotations (rrR, rrLp, rrU, rrDp, rrF, rrBp) is involved with

```

```

## two pairs of connected but different permutations/transformations as follows:
## (a) one pair for the 12 'Side' cubies (arrays = @Xs0 (for Side colours), @Xs1 (for Side numbers)), and
## (b) one pair for the 9 'Face' cubies (arrays = @Xf0 (for Face colours), @Xf1 (for Face numbers)).
## Each of the center slice rotations (rrSr, rrSu, rrSf) is involved with just one pair of
## permutations for the 12 Side cubies (arrays = @Xs0 (for Side colours), @Xs1 (for Side numbers)).
## We document only the Side and Face of the first sub (rrR) in detail, since the other subs are
## of similar form.
##=====

```

2.8 rrR

See subsections **rr—overview** (above) for details of notation.

```

sub rrR {

## the RIGHT (slice + face) transform
## R = RIGHT, s = side; 0=colour, 1= number
## make the clockwise rotation permutation
## In this permutation the Front-right-bottom (Frb) (side)facelet rotates to
## the new position of Up-right-bottom (Urb) (side)facelet.
##-----SIDE-----
## 12 side cubie facelets in arrays @Rs0 (colours) and @Rs1 (numbers)
## these are the initial positions

    @Rs0 = (
        $Frb[0], $Frm[0], $Frt[0], $Urb[0], $Urm[0], $Urt[0],
        $Blt[0], $Blm[0], $Blb[0], $Drb[0], $Drm[0], $Drt[0]
    );

    @Rs1 = (
        $Frb[1], $Frm[1], $Frt[1], $Urb[1], $Urm[1], $Urt[1],
        $Blt[1], $Blm[1], $Blb[1], $Drb[1], $Drm[1], $Drt[1]
    );

## now we reallocate the initial array elements to the new
## post (90 degree clockwise) rotation position.

```

```

## Cube is viewed from FRONT.
## Positions of side facelets of Right slice are numbered 0-11 in clockwise direction,
##   (as seen from Right face) starting with Up-right-bottom facelet.
## First line example:
## variable $Urb[0] (Upface-right-bottom colour) <-- colour of first element in @Rs0 (=Frb[0])
## variable $Urb[1] (Upface-right-bottom number) <-- number of first element in @Rs1 (=Frb[1])

$Urb[0] = $Rs0[0];
$Urb[1] = $Rs1[0];
$Urm[0] = $Rs0[1];
$Urm[1] = $Rs1[1];
$Urt[0] = $Rs0[2];
$Urt[1] = $Rs1[2];

$Blt[0] = $Rs0[3];
$Blt[1] = $Rs1[3];
$Blm[0] = $Rs0[4];
$Blm[1] = $Rs1[4];
$Blb[0] = $Rs0[5];
$Blb[1] = $Rs1[5];

$Drb[0] = $Rs0[6];
$Drb[1] = $Rs1[6];
$Drm[0] = $Rs0[7];
$Drm[1] = $Rs1[7];
$Drt[0] = $Rs0[8];
$Drt[1] = $Rs1[8];

$Frb[0] = $Rs0[9];
$Frb[1] = $Rs1[9];
$Frm[0] = $Rs0[10];
$Frm[1] = $Rs1[10];
$Frt[0] = $Rs0[11];
$Frt[1] = $Rs1[11];

##-----Right FACE-----
## RIGHT FACE (9 cubies in each array)
## (numbered in rows: 1,2,3/4,5,6/7,8,9 from top left(1) to bottom right(9))

```

```

## R=Right, f = face; 0=colour, 1= number
## do the Rface (90 degree) rotation transform
## here the Right-left-bottom (Rlb) facelet rotates to the possn of Right-left-top (Rlt)
## we start with two arrays (one for colours @Rf0, one for numbers @Rf1) with 9 elements each.

  @Rf0 = (
    $Rlb[0], $Rlm[0], $Rlt[0], $Rmb[0], $Rmm[0],
    $Rmt[0], $Rrb[0], $Rrm[0], $Rrt[0]
  );
  @Rf1 = (
    $Rlb[1], $Rlm[1], $Rlt[1], $Rmb[1], $Rmm[1],
    $Rmt[1], $Rrb[1], $Rrm[1], $Rrt[1]
  );

## now we reallocate the array elements to the new
## post (90 degree clockwise) rotation facelet position.
## Right face is viewed from RIGHT.
## First line example:
## variable $Rlt[0] (=Right-left-top colour) <-- colour of first element in @Rf0 (=Rlb[0])
## variable $Rlt[1] (=Right-left-top number) <-- number of first element in @Rf1 (=Rlb[1])

  $Rlt[0] = $Rf0[0];
  $Rlt[1] = $Rf1[0];
  $Rmt[0] = $Rf0[1];
  $Rmt[1] = $Rf1[1];
  $Rrt[0] = $Rf0[2];
  $Rrt[1] = $Rf1[2];

  $Rlm[0] = $Rf0[3];
  $Rlm[1] = $Rf1[3];
  $Rmm[0] = $Rf0[4];
  $Rmm[1] = $Rf1[4];
  $Rrm[0] = $Rf0[5];
  $Rrm[1] = $Rf1[5];

  $Rlb[0] = $Rf0[6];
  $Rlb[1] = $Rf1[6];
  $Rmb[0] = $Rf0[7];

```

```

    $Rmb[1] = $Rf1[7];
    $Rrb[0] = $Rf0[8];
    $Rrb[1] = $Rf1[8];

}

#=====

```

2.9 rrSr

See subsections **rr-overview** and **rrR** (above) for details of notation.

```

sub rrSr {

## Sr = RIGHT middle SLICE rotation (only 12 side facelets)
## modified from rrR (change the U,D,F, r --> m and Back Bl-->Bm; Rs--> ?Srs)
## change only the slice
## s = side; 0=colour, 1= number
## make the post rotation permutation

@SRs0 = (
    $Fmb[0], $Fmm[0], $Fmt[0], $Umb[0], $Umm[0], $Umt[0],
    $Bmt[0], $Bmm[0], $Bmb[0], $Dmb[0], $Dmm[0], $Dmt[0]
);

@SRs1 = (
    $Fmb[1], $Fmm[1], $Fmt[1], $Umb[1], $Umm[1], $Umt[1],
    $Bmt[1], $Bmm[1], $Bmb[1], $Dmb[1], $Dmm[1], $Dmt[1]
);

$Umb[0] = $SRs0[0];
$Umb[1] = $SRs1[0];
$Umm[0] = $SRs0[1];
$Umm[1] = $SRs1[1];
$Umt[0] = $SRs0[2];
$Umt[1] = $SRs1[2];

$Bmt[0] = $SRs0[3];

```

```

$Bmt[1] = $SRs1[3];
$Bmm[0] = $SRs0[4];
$Bmm[1] = $SRs1[4];
$Bmb[0] = $SRs0[5];
$Bmb[1] = $SRs1[5];

```

```

$Dmb[0] = $SRs0[6];
$Dmb[1] = $SRs1[6];
$Dmm[0] = $SRs0[7];
$Dmm[1] = $SRs1[7];
$Dmt[0] = $SRs0[8];
$Dmt[1] = $SRs1[8];

```

```

$Fmb[0] = $SRs0[9];
$Fmb[1] = $SRs1[9];
$Fmm[0] = $SRs0[10];
$Fmm[1] = $SRs1[10];
$Fmt[0] = $SRs0[11];
$Fmt[1] = $SRs1[11];

```

```

}

```

```

#=====

```

2.10 rrLp

See subsections **rr-overview** and **rrR** (above) for details of notation.

```

sub rrLp {

```

```

## LEFT slice (side + face) anticlockwise rotation

```

```

## s = side; 0=colour, 1= number

```

```

##-----side-----

```

```

@LPs0 = (
    $Flb[0], $Flm[0], $Flt[0], $Ulb[0], $Ulm[0], $Ult[0],
    $Brb[0], $Brm[0], $Brb[0], $Dlb[0], $Dlm[0], $Dlt[0]
);

```

```
@LPs1 = (
    $Flb[1], $Flm[1], $Flt[1], $Ulb[1], $Ulm[1], $Ult[1],
    $Brt[1], $Brm[1], $Brb[1], $Dlb[1], $Dlm[1], $Dlt[1]
);
```

```
$Ulb[0] = $LPs0[0];
$Ulb[1] = $LPs1[0];
$Ulm[0] = $LPs0[1];
$Ulm[1] = $LPs1[1];
$Ult[0] = $LPs0[2];
$Ult[1] = $LPs1[2];
```

```
$Brt[0] = $LPs0[3];
$Brt[1] = $LPs1[3];
$Brm[0] = $LPs0[4];
$Brm[1] = $LPs1[4];
$Brb[0] = $LPs0[5];
$Brb[1] = $LPs1[5];
```

```
$Dlb[0] = $LPs0[6];
$Dlb[1] = $LPs1[6];
$Dlm[0] = $LPs0[7];
$Dlm[1] = $LPs1[7];
$Dlt[0] = $LPs0[8];
$Dlt[1] = $LPs1[8];
```

```
$Flb[0] = $LPs0[9];
$Flb[1] = $LPs1[9];
$Flm[0] = $LPs0[10];
$Flm[1] = $LPs1[10];
$Flt[0] = $LPs0[11];
$Flt[1] = $LPs1[11];
```

```
##-----Left FACE-----
```

```
## do the LEFT face transform (in rows: 1,2,3//4,5,6//7,8,9)
```

```
## f = face; 0=colour, 1= number
```

```
## NOTES: not same as for R
```

```

@LPf0 = (
    $Lrt[0], $Lrm[0], $Lrb[0], $Lmt[0], $Lmm[0],
    $Lmb[0], $Llt[0], $Llm[0], $Llb[0]
);
@LPf1 = (
    $Lrt[1], $Lrm[1], $Lrb[1], $Lmt[1], $Lmm[1],
    $Lmb[1], $Llt[1], $Llm[1], $Llb[1]
);

$Llt[0] = $LPf0[0];
$Llt[1] = $LPf1[0];
$Lmt[0] = $LPf0[1];
$Lmt[1] = $LPf1[1];
$Lrt[0] = $LPf0[2];
$Lrt[1] = $LPf1[2];

$Llm[0] = $LPf0[3];
$Llm[1] = $LPf1[3];
$Lmm[0] = $LPf0[4];
$Lmm[1] = $LPf1[4];
$Lrm[0] = $LPf0[5];
$Lrm[1] = $LPf1[5];

$Llb[0] = $LPf0[6];
$Llb[1] = $LPf1[6];
$Lmb[0] = $LPf0[7];
$Lmb[1] = $LPf1[7];
$Lrb[0] = $LPf0[8];
$Lrb[1] = $LPf1[8];

}

#=====

```

2.11 rrU

See subsections **rr—overview** and **rrR** (above) for details of notation.

```
sub rrU {

## UP slice (side + face)
## do the Uside transform
## s = side; 0=colour, 1= number
## -----SIDE-----

    @Us0 = (
        $Lrt[0], $Lmt[0], $Llt[0], $Brt[0], $Bmt[0], $Blt[0],
        $Rrt[0], $Rmt[0], $Rlt[0], $Frt[0], $Fmt[0], $Flt[0]
    );

    @Us1 = (
        $Lrt[1], $Lmt[1], $Llt[1], $Brt[1], $Bmt[1], $Blt[1],
        $Rrt[1], $Rmt[1], $Rlt[1], $Frt[1], $Fmt[1], $Flt[1]
    );

    $Brt[0] = $Us0[0];
    $Brt[1] = $Us1[0];
    $Bmt[0] = $Us0[1];
    $Bmt[1] = $Us1[1];
    $Blt[0] = $Us0[2];
    $Blt[1] = $Us1[2];

    $Rrt[0] = $Us0[3];
    $Rrt[1] = $Us1[3];
    $Rmt[0] = $Us0[4];
    $Rmt[1] = $Us1[4];
    $Rlt[0] = $Us0[5];
    $Rlt[1] = $Us1[5];

    $Frt[0] = $Us0[6];
    $Frt[1] = $Us1[6];
    $Fmt[0] = $Us0[7];
    $Fmt[1] = $Us1[7];
```

```

$Flt[0] = $Us0[8];
$Flt[1] = $Us1[8];

$Lrt[0] = $Us0[9];
$Lrt[1] = $Us1[9];
$Lmt[0] = $Us0[10];
$Lmt[1] = $Us1[10];
$Llt[0] = $Us0[11];
$Llt[1] = $Us1[11];

```

```

##-----Up FACE-----
## do the Rface transform (in rows: 1,2,3//4,5,6//7,8,9)
## f = face; 0=colour, 1= number

```

```

@Uf0 = (
    $Ulb[0], $Ulm[0], $Ult[0], $Umb[0], $Umm[0],
    $Umt[0], $Urb[0], $Urm[0], $Urt[0]
);
@Uf1 = (
    $Ulb[1], $Ulm[1], $Ult[1], $Umb[1], $Umm[1],
    $Umt[1], $Urb[1], $Urm[1], $Urt[1]
);

```

```

$Ult[0] = $Uf0[0];
$Ult[1] = $Uf1[0];
$Umt[0] = $Uf0[1];
$Umt[1] = $Uf1[1];
$Urt[0] = $Uf0[2];
$Urt[1] = $Uf1[2];

```

```

$Ulm[0] = $Uf0[3];
$Ulm[1] = $Uf1[3];
$Umm[0] = $Uf0[4];
$Umm[1] = $Uf1[4];
$Urm[0] = $Uf0[5];
$Urm[1] = $Uf1[5];

```

```

$Ulb[0] = $Uf0[6];

```

```

$Ulb[1] = $Uf1[6];
$Umb[0] = $Uf0[7];
$Umb[1] = $Uf1[7];
$Urb[0] = $Uf0[8];
$Urb[1] = $Uf1[8];

```

```

}

```

```

#=====

```

2.12 rrSu

See subsections **rr-overview** and **rrR** (above) for details of notation.

```

sub rrSu {

```

```

## middle slice rotation (side only 12 facelets)

```

```

## s = side; 0=colour, 1= number

```

```

## make the post rotation permutation

```

```

##-----SIDE-----

```

```

@SUs0 = (
    $Lrm[0], $Lmm[0], $Llm[0], $Brm[0], $Bmm[0], $Blm[0],
    $Rrm[0], $Rmm[0], $Rlm[0], $Frm[0], $Fmm[0], $Flm[0]
);

```

```

@SUs1 = (
    $Lrm[1], $Lmm[1], $Llm[1], $Brm[1], $Bmm[1], $Blm[1],
    $Rrm[1], $Rmm[1], $Rlm[1], $Frm[1], $Fmm[1], $Flm[1]
);

```

```

$Brm[0] = $SUs0[0];
$Brm[1] = $SUs1[0];
$Bmm[0] = $SUs0[1];
$Bmm[1] = $SUs1[1];
$Blm[0] = $SUs0[2];
$Blm[1] = $SUs1[2];

```

```

$Rrm[0] = $SUs0[3];
$Rrm[1] = $SUs1[3];
$Rmm[0] = $SUs0[4];
$Rmm[1] = $SUs1[4];
$Rlm[0] = $SUs0[5];
$Rlm[1] = $SUs1[5];

```

```

$Frm[0] = $SUs0[6];
$Frm[1] = $SUs1[6];
$Fmm[0] = $SUs0[7];
$Fmm[1] = $SUs1[7];
$Flm[0] = $SUs0[8];
$Flm[1] = $SUs1[8];

```

```

$Lrm[0] = $SUs0[9];
$Lrm[1] = $SUs1[9];
$Lmm[0] = $SUs0[10];
$Lmm[1] = $SUs1[10];
$Llm[0] = $SUs0[11];
$Llm[1] = $SUs1[11];

```

```

}

```

```

#=====

```

2.13 rrDp

See subsections **rr-overview** and **rrR** (above) for details of notation.

```

sub rrDp {

```

```

## Down Face anticlockwise rotation (side and face)
## s = side; 0=colour, 1= number
## make the post rotation permutation
##-----SIDE-----

```

```

  @DPs0 = (
    $Lrb[0], $Lmb[0], $Llb[0], $Brb[0], $Bmb[0], $Blb[0],

```

```

    $Rrb[0], $Rmb[0], $Rlb[0], $Frb[0], $Fmb[0], $Flb[0]
);

@DPs1 = (
    $Lrb[1], $Lmb[1], $Llb[1], $Brb[1], $Bmb[1], $Blb[1],
    $Rrb[1], $Rmb[1], $Rlb[1], $Frb[1], $Fmb[1], $Flb[1]
);

$Brb[0] = $DPs0[0];
$Brb[1] = $DPs1[0];
$Bmb[0] = $DPs0[1];
$Bmb[1] = $DPs1[1];
$Blb[0] = $DPs0[2];
$Blb[1] = $DPs1[2];

$Rrb[0] = $DPs0[3];
$Rrb[1] = $DPs1[3];
$Rmb[0] = $DPs0[4];
$Rmb[1] = $DPs1[4];
$Rlb[0] = $DPs0[5];
$Rlb[1] = $DPs1[5];

$Frb[0] = $DPs0[6];
$Frb[1] = $DPs1[6];
$Fmb[0] = $DPs0[7];
$Fmb[1] = $DPs1[7];
$Flb[0] = $DPs0[8];
$Flb[1] = $DPs1[8];

$Lrb[0] = $DPs0[9];
$Lrb[1] = $DPs1[9];
$Lmb[0] = $DPs0[10];
$Lmb[1] = $DPs1[10];
$Llb[0] = $DPs0[11];
$Llb[1] = $DPs1[11];

##-----Down FACE-----
## f = face; 0=colour, 1= number

```

```

@DPf0 = (
    $Dlt[0], $Dlm[0], $Dlb[0], $Dmt[0], $Dmm[0],
    $Dmb[0], $Drt[0], $Drm[0], $Drb[0]
);
@DPf1 = (
    $Dlt[1], $Dlm[1], $Dlb[1], $Dmt[1], $Dmm[1],
    $Dmb[1], $Drt[1], $Drm[1], $Drb[1]
);

```

```

$Dlb[0] = $DPf0[0];
$Dlb[1] = $DPf1[0];
$Dmb[0] = $DPf0[1];
$Dmb[1] = $DPf1[1];
$Drb[0] = $DPf0[2];
$Drb[1] = $DPf1[2];

```

```

$Dlm[0] = $DPf0[3];
$Dlm[1] = $DPf1[3];
$Dmm[0] = $DPf0[4];
$Dmm[1] = $DPf1[4];
$Drm[0] = $DPf0[5];
$Drm[1] = $DPf1[5];

```

```

$Dlt[0] = $DPf0[6];
$Dlt[1] = $DPf1[6];
$Dmt[0] = $DPf0[7];
$Dmt[1] = $DPf1[7];
$Drt[0] = $DPf0[8];
$Drt[1] = $DPf1[8];

```

```

}

```

```

#=====

```

2.14 rrF

See subsections **rr—overview** and **rrR** (above) for details of notation.

```
sub rrF {

## do the Fside transform (side and face)
## s = side; 0=colour, 1= number
## -----SIDE-----

  @Fs0 = (
    $Lrb[0], $Lrm[0], $Lrt[0], $Ulb[0], $Umb[0], $Urb[0],
    $Rlt[0], $Rlm[0], $Rlb[0], $Drt[0], $Dmt[0], $Dlt[0]
  );

  @Fs1 = (
    $Lrb[1], $Lrm[1], $Lrt[1], $Ulb[1], $Umb[1], $Urb[1],
    $Rlt[1], $Rlm[1], $Rlb[1], $Drt[1], $Dmt[1], $Dlt[1]
  );

  $Ulb[0] = $Fs0[0];
  $Ulb[1] = $Fs1[0];
  $Umb[0] = $Fs0[1];
  $Umb[1] = $Fs1[1];
  $Urb[0] = $Fs0[2];
  $Urb[1] = $Fs1[2];

  $Rlt[0] = $Fs0[3];
  $Rlt[1] = $Fs1[3];
  $Rlm[0] = $Fs0[4];
  $Rlm[1] = $Fs1[4];
  $Rlb[0] = $Fs0[5];
  $Rlb[1] = $Fs1[5];

  $Drt[0] = $Fs0[6];
  $Drt[1] = $Fs1[6];
  $Dmt[0] = $Fs0[7];
  $Dmt[1] = $Fs1[7];
  $Dlt[0] = $Fs0[8];
```

```
$Dlt[1] = $Fs1[8];
```

```
$Lrb[0] = $Fs0[9];
```

```
$Lrb[1] = $Fs1[9];
```

```
$Lrm[0] = $Fs0[10];
```

```
$Lrm[1] = $Fs1[10];
```

```
$Lrt[0] = $Fs0[11];
```

```
$Lrt[1] = $Fs1[11];
```

```
## -----Front FACE-----
```

```
## f = face; 0=colour, 1= number
```

```
@Lf0 = (
```

```
    $Flb[0], $Flm[0], $Flt[0], $Fmb[0], $Fmm[0],
```

```
    $Fmt[0], $Frb[0], $Frm[0], $Frt[0]
```

```
);
```

```
@Lf1 = (
```

```
    $Flb[1], $Flm[1], $Flt[1], $Fmb[1], $Fmm[1],
```

```
    $Fmt[1], $Frb[1], $Frm[1], $Frt[1]
```

```
);
```

```
$Flt[0] = $Lf0[0];
```

```
$Flt[1] = $Lf1[0];
```

```
$Fmt[0] = $Lf0[1];
```

```
$Fmt[1] = $Lf1[1];
```

```
$Frt[0] = $Lf0[2];
```

```
$Frt[1] = $Lf1[2];
```

```
$Flm[0] = $Lf0[3];
```

```
$Flm[1] = $Lf1[3];
```

```
$Fmm[0] = $Lf0[4];
```

```
$Fmm[1] = $Lf1[4];
```

```
$Frm[0] = $Lf0[5];
```

```
$Frm[1] = $Lf1[5];
```

```
$Flb[0] = $Lf0[6];
```

```
$Flb[1] = $Lf1[6];
```

```
$Fmb[0] = $Lf0[7];
```

```

    $Fmb[1] = $Lf1[7];
    $Frb[0] = $Lf0[8];
    $Frb[1] = $Lf1[8];

}

#=====

```

2.15 rrSf

See subsections **rr-overview** and **rrR** (above) for details of notation.

```

sub rrSf {

## do the FRONT middle slice Fm transform (side only)
## s = side; 0=colour, 1= number
##-----SIDE-----

    @SFs0 = (
        $Lmb[0], $Lmm[0], $Lmt[0], $Ulm[0], $Umm[0], $Urm[0],
        $Rmt[0], $Rmm[0], $Rmb[0], $Drm[0], $Dmm[0], $Dlm[0]
    );

    @SFs1 = (
        $Lmb[1], $Lmm[1], $Lmt[1], $Ulm[1], $Umm[1], $Urm[1],
        $Rmt[1], $Rmm[1], $Rmb[1], $Drm[1], $Dmm[1], $Dlm[1]
    );

    $Ulm[0] = $SFs0[0];
    $Ulm[1] = $SFs1[0];
    $Umm[0] = $SFs0[1];
    $Umm[1] = $SFs1[1];
    $Urm[0] = $SFs0[2];
    $Urm[1] = $SFs1[2];

    $Rmt[0] = $SFs0[3];
    $Rmt[1] = $SFs1[3];
    $Rmm[0] = $SFs0[4];

```

```

$Rmm[1] = $SFs1[4];
$Rmb[0] = $SFs0[5];
$Rmb[1] = $SFs1[5];

$Drm[0] = $SFs0[6];
$Drm[1] = $SFs1[6];
$Dmm[0] = $SFs0[7];
$Dmm[1] = $SFs1[7];
$Dlm[0] = $SFs0[8];
$Dlm[1] = $SFs1[8];

$Lmb[0] = $SFs0[9];
$Lmb[1] = $SFs1[9];
$Lmm[0] = $SFs0[10];
$Lmm[1] = $SFs1[10];
$Lmt[0] = $SFs0[11];
$Lmt[1] = $SFs1[11];
}

```

23

```
#=====
```

2.16 rrBp

See subsections **rr-overview** and **rrR** (above) for details of notation.

```

sub rrBp {

## Back rotation anticlockwise (side + face)
## do the Bp side transform
## s = side; 0=colour, 1= number
## -----Side-----

@BPs0 = (
    $Llb[0], $Llm[0], $Llt[0], $Ult[0], $Umt[0], $Urt[0],
    $Rrt[0], $Rrm[0], $Rrb[0], $Drb[0], $Dmb[0], $Dlb[0]
);

@BPs1 = (

```

```

        $L1b[1], $L1m[1], $L1t[1], $U1t[1], $Umt[1], $Urt[1],
        $Rrt[1], $Rrm[1], $Rrb[1], $Drb[1], $Dmb[1], $D1b[1]
    );

    $U1t[0] = $BPs0[0];
    $U1t[1] = $BPs1[0];
    $Umt[0] = $BPs0[1];
    $Umt[1] = $BPs1[1];
    $Urt[0] = $BPs0[2];
    $Urt[1] = $BPs1[2];

    $Rrt[0] = $BPs0[3];
    $Rrt[1] = $BPs1[3];
    $Rrm[0] = $BPs0[4];
    $Rrm[1] = $BPs1[4];
    $Rrb[0] = $BPs0[5];
    $Rrb[1] = $BPs1[5];

    $Drb[0] = $BPs0[6];
    $Drb[1] = $BPs1[6];
    $Dmb[0] = $BPs0[7];
    $Dmb[1] = $BPs1[7];
    $D1b[0] = $BPs0[8];
    $D1b[1] = $BPs1[8];

    $L1b[0] = $BPs0[9];
    $L1b[1] = $BPs1[9];
    $L1m[0] = $BPs0[10];
    $L1m[1] = $BPs1[10];
    $L1t[0] = $BPs0[11];
    $L1t[1] = $BPs1[11];

    ##-----Back FACE-----
    ## do the B face transform (in rows: 1,2,3/4,5,6/7,8,9)
    ## f = face; 0=colour, 1= number

    @BPf0 = (
        $Brb[0], $Brm[0], $Brt[0], $Bmb[0], $Bmm[0],

```

```

        $Bmt[0], $B1b[0], $B1m[0], $B1t[0]
    );

    @BPf1 = (
        $Brb[1], $Brm[1], $Brt[1], $Bmb[1], $Bmm[1],
        $Bmt[1], $B1b[1], $B1m[1], $B1t[1]
    );

    $Brt[0] = $BPf0[0];
    $Brt[1] = $BPf1[0];
    $Bmt[0] = $BPf0[1];
    $Bmt[1] = $BPf1[1];
    $B1t[0] = $BPf0[2];
    $B1t[1] = $BPf1[2];

    $Brm[0] = $BPf0[3];
    $Brm[1] = $BPf1[3];
    $Bmm[0] = $BPf0[4];
    $Bmm[1] = $BPf1[4];
    $B1m[0] = $BPf0[5];
    $B1m[1] = $BPf1[5];

    $Brb[0] = $BPf0[6];
    $Brb[1] = $BPf1[6];
    $Bmb[0] = $BPf0[7];
    $Bmb[1] = $BPf1[7];
    $B1b[0] = $BPf0[8];
    $B1b[1] = $BPf1[8];
}

```

2.17 rr-overview of derivative subs

```

## Note that we have already defined (as rotation SUBs above) just 9 primary rotation
## transforms, namely:
## (x axis): rrR, rrSr, rrLp,
## (y axis): rrU, rrSu, rrDp,

```

```

## (z axis): rrF, rrSf, rrBp,
## and since all remaining possible rotations are simply combinations of these 9
## we now define all the other rotation subs in terms of these 9 primary rotations.
## Do NOT use multiples here: write each rotation separately
## Note that Sr, Su, Sf are middle slice rotations (= Rm, Um, Fm respectively).
##=====

```

2.18 rrRp

```
sub rrRp { &rrR; &rrR; &rrR };    # (=rrR3)
```

2.19 rrRw

```
sub rrRw { &rrR; &rrSr };          # (= rrR + rrSr)
```

2.20 rrRwp

```
sub rrRwp { &rrR; &rrR; &rrR; &rrSr; &rrSr; &rrSr };    # (= rrRp + rrSrp)
```

2.21 rrRs

```
sub rrRs { &rrR; &rrLp }
```

2.22 rrRsp

```
sub rrRsp { &rrRp; &rrL }
```

2.23 rrRa

```
sub rrRa { &rrR; &rrL }
```

2.24 rrRap

```
sub rrRap { &rrRp; &rrLp }
```

```
## -----
```

2.25 rrL

```
sub rrL { &rrLp; &rrLp; &rrLp };          # (= rrLp3)
```

2.26 rrLw

```
sub rrLw { &rrLp; &rrLp; &rrLp; &rrSrp };  # (=rrLp3 + rrSrp)
```

2.27 rrLwp

```
sub rrLwp { &rrLp; &rrSr }
```

2.28 rrLs

```
sub rrLs { &rrL; &rrRp }
```

2.29 rrLsp

```
sub rrLsp { &rrLp; &rrR }
```

2.30 rrLa

```
sub rrLa { &rrL; &rrR }
```

2.31 rrLap

```
sub rrLap { &rrLp; &rrRp }
```

```
## ----derivative subs from U ----
```

2.32 rrUp

```
sub rrUp { &rrU; &rrU; &rrU };          # (=rrU3)
```

2.33 rrUw

```
sub rrUw { &rrU; &rrSu };              #
```

2.34 rrUwp

```
sub rrUwp { &rrUp; &rrSup }
```

2.35 rrUs

```
sub rrUs { &rrU; &rrDp }
```

2.36 rrUsp

```
sub rrUsp { &rrUp; &rrD }
```

2.37 rrUa

```
sub rrUa { &rrU; &rrD }
```

2.38 rrUap

```
sub rrUap { &rrUp; &rrDp }
```

```
## -----
```

2.39 rrD

```
sub rrD { &rrDp; &rrDp; &rrDp }; # (= rrDp3)
```

2.40 rrDw

```
sub rrDw { &rrDp; &rrDp; &rrDp; &rrSup }; # (=rrDp3 + rrSup)
```

2.41 rrDwp

```
sub rrDwp { &rrDp; &rrSu }
```

2.42 rrDs

```
sub rrDs { &rrD; &rrUp }
```

2.43 rrDsp

```
sub rrDsp { &rrDp; &rrU }
```

2.44 rrDa

```
sub rrDa { &rrD; &rrU }
```

2.45 rrDap

```
sub rrDap { &rrDp; &rrUp }
```

```
## ----derivative subs from F ----
```

2.46 rrFw

```
sub rrFw { &rrF; &rrSf }; # (= rrF + rrSf)
```

2.47 rrFp

```
sub rrFp { &rrF; &rrF; &rrF }; # (=rrF3)
```

2.48 rrFwp

```
sub rrFwp { &rrF; &rrF; &rrF; &rrSf; &rrSf; &rrSf }; # (= rrF3 + rrSf3)
```

2.49 rrFs

```
sub rrFs { &rrF; &rrBp }
```

2.50 rrFsp

```
sub rrFsp { &rrFp; &rrB }
```

2.51 rrFa

```
sub rrFa { &rrF; &rrB }
```

2.52 rrFap

```
sub rrFap { &rrFp; &rrBp }
```

```
## -----
```

2.53 rrB

```
sub rrB { &rrBp; &rrBp; &rrBp }; # (= rrBp3)
```

2.54 rrBw

```
sub rrBw { &rrBp; &rrBp; &rrBp; &rrSfp }; # (=rrBp3 + rrSfp)
```

2.55 rrBwp

```
sub rrBwp { &rrBp; &rrSf }
```

2.56 rrBs

```
sub rrBs { &rrB; &rrFp }
```

2.57 rrBsp

```
sub rrBsp { &rrBp; &rrF }
```

2.58 rrBa

```
sub rrBa { &rrB; &rrF }
```

2.59 rrBap

```
sub rrBap { &rrBp; &rrFp }
```

```
## ----bring all the S versions together ----
```

2.60 rrSup

```
sub rrSup { &rrSu; &rrSu; &rrSu }; # (=rrSu3)
```

2.61 rrSd

```
sub rrSd { &rrSup }; # (=rrSup)
```

2.62 rrSdp

```
sub rrSdp { &rrSu }; # (=rrSu)
```

2.63 rrSl

```
sub rrSl { &rrSrp }; # (=rrSrp)
```

2.64 rrSlp

```
sub rrSlp { &rrSr }; # (=rrSr)
```

2.65 rrSrp

```
sub rrSrp { &rrSr; &rrSr; &rrSr }; # (=rrSr3)
```

2.66 rrSfp

```
sub rrSfp { &rrSf; &rrSf; &rrSf }; # (=rrSf3)
```

2.67 rrSb

```
sub rrSb { &rrSfp }; # (=rrSfp)
```

2.68 rrSbp

```
sub rrSbp { &rrSf }; # (=rrSf)
```

```
#=====
```

2.69 rubikmod

This subroutine processes a rotation code (e.g., Dp257) and splits it into the terminal numeric rotation number (\rightarrow \$rotnumber) and the (remaining) front part (\rightarrow \$rotcode). It then determines the value modulo-4 of the number \$rotnumber (\rightarrow \$modnumber). Finally, it returns the three values \$rotcode, \$rotnumber, \$modnumber.

For example, in this case, the rotation code Dp257 would be split into the rotation Rp \rightarrow \$rotcode, and the number 257 \rightarrow \$rotnumber. Since $257 \equiv 3 \pmod{4}$, then the value 3 \rightarrow \$modnumber.

```
sub rubikmod {

    ## for MODifying (MOD 4)
    print " SUB rubikmod\n";
    ## passing one RotationElement as a parameter, & return a modified one
    ## make local variables

    my $rot      = "";
    my $lencode = "";
    my $char     = "";
    my $m4       = -1;
    my $num      = -1;
    my $p        = 0;

    # grab the parameter string
    # $code = @_[0];
    $code = $_[0];    ## Perl says this is better

    $lencode = length $code;

    ## we want to split the code string into the front (Rubikcode) and terminal number
    ## so grab 1 char sequentially starting from the end of the string
    ## and identify the position of the first non-digit char we get to
    ## example: $lastchar = substr $code,-1,1 ;
    for ( $p = -1 ; $p > -$lencode - 1 ; $p = $p - 1 ) {
        $char = substr $code, $p, 1;
        if ( $char =~ /\d/ ) { }
        else {
            ## this char is the first non-digit from the end"
            ## its position = $p
        }
    }
}
```

8

```

        last;
    }
}

## now use the value of $p to split the code string
## into front part (= $rot) and back part (= $num)

## get $rot
$rot = substr $code, 0, ( $lencode + $p + 1 );

## get $num
$num = substr $code, $lencode + $p + 1, ( $lencode - ( length $rot ) );

##-----
## if no number at all (eg D) then this Rubikcode needs to be implemented just once
## so allocate its num to have value = 1

## if ($num == ""){$num=1}; ## BUT this gives an ErrorMessage when num="" etc
## so I have rewritten [if numlength --> 0 then..] then it works OK

$numlength = ( $lencode - ( length $rot ) );
if ( $numlength == 0 ) { $num = 1 }
##-----

## determine mod 4 of the value num
$m4 = $num % 4;

## now return the results
$rotcode = $rot;
$rotnumber = $num;    ## we return this so we can use it as a check
$modnumber = $m4;

return $rotcode, $rotnumber, $modnumber;

}    #end of sub

#=====

```

2.70 cleanstring

```
sub cleanstring {

    # to clean leading and trailing whitespace from a string

    my $line = "";

    $line = $_[0];    # copied from my RubikMOD()

    #clean leading & trailing whitespace

    $line =~ s/^\s+//;    ## clean leading whitespace
    $line =~ s/\s+$//;    ## clean trailing whitespace

    return $line;
}

#=====
```

2.71 cutinfoblock

```
sub cutinfoblock {

    ## remove each <infoblock> if any exists

    ## pass the whole dataline

    print " SUB cutinfoblock\n";

    my $dataline = $_[0];

    ## we know all brackets are balanced - as this has been checked already.

    print " dataline = $dataline\n";
    my $Langle      = 0;
    my $Rangle      = 0;
    my $angleblock  = "";
```

```

my $lenangleblock = 0;

## first see if there is a terminal infoblock
$Langle = index $dataline, '<';    ## <
$Rangle = index $dataline, '>';    ## <
$lenangleblock = $Rangle - $Langle + 1;

##-----
## angleblock is the whole block <...> including both angles
## check both angles exist
if ( ( $Langle != -1 ) and ( $Rangle != -1 ) ) {

    my $angleblock = substr( $dataline, $Langle, $lenangleblock );
    print " infoblock(s) present: first = $angleblock\n";

    my $lenangleblock = length $angleblock;
    my $lendataline    = length $dataline;

    # now need to remove the infoblock from $dataline
    # need to get front and back strings
    my $frontstring    = "";
    my $newfrontstring = "";
    my $backstring     = "";

    $lenbackstring = $lendataline - $Rangle - 1;
    $frontstring = substr( $dataline, 0, $Langle );    # string before Langle
    $backstring = substr( $dataline, $Rangle + 1 );    # string beyond Rangle

    print " Langle possn = $Langle\n";
    print " Rangle possn = $Rangle\n";
    print " lenangleblock (diff + 1) = $lenangleblock\n";
    print " lendataline = $lendataline\n";
    print " lenbackstring = $lenbackstring\n";
    print " frontstring = $frontstring\n";

    #remove the terminal comma from front string
    $newfrontstring = substr( $frontstring, 0, $Langle - 1 );
    print " new frontstring = $newfrontstring\n";

```

```

    print " backstring = $backstring\n";

    ##-----
    # remove angleblock from dataline (join front and back strings)
    $newdataline = $newfrontstring . $backstring;

    $SequenceInfo = substr( $angleblock, 1, $lenangleblock - 2 );

    print " new dataline = $newdataline\n";
    print " SequenceInfo = $SequenceInfo\n";
    print " newdataline = $newdataline\n";
    print " done\n\n";

    return $SequenceInfo, $newdataline;
}

else {
    # no infoblock, so need to make newdataline same as orig dataline
    $newdataline = $dataline;
    print " no <infoblock> to remove.\n\n";
    return $newdataline;
}

#-----

} ## end of sub

#=====

```

2.72 fixrepeatelement

```

sub fixrepeatelement {

    print " SUB fixrepeatelement\n";

    print " reformatting any repeat elements...\n";

```

```
## this sub replaces ,-->; and (--> { and ) --> } for the repeat element
## and inserts it back into the original rotation sequence, where it now
## appears as a separate rotation element.
```

```
my $repeatstring      = "";
my $lenrepeatstring = "";
my $newrepeatstring = "";
my $frontstring       = "";
my $backstring        = "";
my $p                 = 0;
my $q                 = 0;
my $len               = 0;
my $k1                = 0;
my $k2                = 0;
```

```
## pass the whole dataline without the keyword
my $dataline = $_[0];    # copied from my RubikMOD()
```

```
$p = index $dataline, '(';
$q = index $dataline, ')';
```

```
print " p = $p, q = $q\n";
```

```
$lenrepeatstring = $q - $p + 1;
$repeatstring = substr( $dataline, $p, $lenrepeatstring );
```

```
print " first repeat string = $repeatstring\n";
print " length of repeat string = $lenrepeatstring\n";
```

```
## translate the chars
$repeatstring =~ tr/,/,/;/;    ## swap , --> ; Black book page 138--139
$repeatstring =~ tr/\(/,\{/;    ## swap ( --> {
$repeatstring =~ tr/\)/,\}/;    ## swap ) --> }
```

```
$newrepeatstring = $repeatstring;
```

```
print "...new repeat string = $newrepeatstring\n";
```

```

#-----
$k1          = $p;                #start of cut
$k2          = $p + $lenrepeatstring;  #end of cut
$frontstring = substr( $dataline, 0, $k1 );
$backstring  = substr( $dataline, $k2 );
print " frontstring = $frontstring\n";
print " backstring = $backstring\n";

# add insert
$newdataline = $frontstring . $newrepeatstring . $backstring;
print " new dataline = $newdataline\n";

print " done\n\n";

}    # end of sub

#=====

```

∞

2.73 repeat

```

sub repeat {

    # to expand the repeating elements

    print " SUB repeat\n";

    ## pass the whole repeatstring = {...}n
    my $repeatstring = $_[0];

    # the string ={code}n
    # get the code sequence and the terminal digit

    my $p          = 0;
    my $q          = 0;
    my $repeatnumber = 0;
    my $repeatcode  = "";
    my $lenrepeatcode = 0;

```

```

$p = index $repeatstring, '{';
$q = index $repeatstring, '}';
$lenrepeatcode = $q - $p - 1;

$repeatcode = substr( $repeatstring, 1, $lenrepeatcode );
print " repeatcode = $repeatcode\n";    ## correct

$lenrepeatstring = length $repeatstring;

print " lenrepeatstring = $lenrepeatstring\n";
print " lenrepeatcode = $lenrepeatcode\n";
print " p = $p\n";
print " q = $q\n";

if ( $lenrepeatcode == ( $lenrepeatstring - 2 ) ) {
    print " there is no trailing number --> 1\n";
    $repeatnumber = 1;
    print " set repeatnumber = $repeatnumber\n";
}
else {
    $repeatnumber = substr( $repeatstring, $q + 1 );    # correct
    print " repeatnumber = $repeatnumber\n";

    ## need to check that repeatnumber is a valid integer
    if ( $repeatnumber =~ /\D/ ) {

        # not a valid number

        ## renormalise brackets etc before outputting to LaTeX
        $repeatnumber = restorebrackets($repeatnumber);

        gprint(".*repeat-no. ERROR: $repeatnumber not numeric");
        ErrorMessage(
"repeat-no. $repeatnumber not numeric ?missing comma or nested ()"
        );
    }
}

```

```

    }
    ;      #end of else

    ## make n copies of repeatcode
    ## we need commas only between elements (not at end)

    $insert = "";          ## $insert = global
    $insert = $repeatcode;
    for ( $t = 1 ; $t < $repeatnumber ; $t = $t + 1 ) {
        $insert = $insert . "," . $repeatcode;
    }

    print " insert = $insert\n";
    print " done\n\n";

}      # end sub

#=====

```

90

2.74 quitprogram

```

sub quitprogram {

    ## exiting the program cleanly
    print " closing down:  writing state...\n";
    writestate();      ## write to the output files
    close;              ## close all files
    exit;

}

#=====

```

2.75 showarray

```

sub showarray {

```

```

        # show the array as a string

        my @newarray = @_;
        my $arraystring = join( ",", @newarray );
        print "  the array  = *$arraystring*\n\n";
    }

#=====

```

2.76 cleanarray

```

sub cleanarray {

    # cleans array elements of leading and trailing whitespace

    my @cleanset = ();
    my @line     = @_;
    my $E;

    foreach $E (@line) {
        $E =~ s/^\s+//;    ## clean leading whitespace
        $E =~ s/\s+$//;    ## clean trailing whitespace
        push @cleanset, $E;
    }

    return @cleanset;
}

#=====

```

2.77 restorebrackets

```

sub restorebrackets {

    my $line = $_[0];

```

```

    ## translate the chars
    $line =~ tr/;/,/;      ## swap , --> ; Black book page 138--139
    $line =~ tr/\{ /\(/;   ## swap ( --> {
    $line =~ tr/\} /\)/;   ## swap ) --> }

    return $line;
}

#=====

```

2.78 infoblockcolon

```

sub infoblockcolon {

    print "...SUB InfoblockColon\n";

    ## pass the whole dataline without the keyword
    my $line = $_[0];    # copied from my RubikMOD()

    if ( ( index $line, '<' ) == -1 ) {

        # no infoblock, so need to make newdataline same as orig dataline
        print " no <infoblock> found.\n\n";
        $newdataline = $dataline;
        return $newdataline;
    }
    else {
        print " infoblock(s) present\n";

        print " start-string = $line\n";

        # look at each char
        my $j = 0;
        my $char = "";
    }
}

```

```

my $lenstring = 0;
$lenstring = length $line;

# set initial state of inout-flag
my $inoutflag = "outside";

#-----
for ( $j = 0 ; $j <= $lenstring ; $j = $j + 1 ) {
    $char = substr( $line, $j, 1 );

    if ( ( $char eq ',' ) and ( $inoutflag eq 'inside' ) ) {

        # replace the char with ;
        substr( $line, $j, 1, ";" );
        print " colon-string  = $line\n";
    }

    ## need these at end of the loop
    if ( $char eq '<' ) { $inoutflag = "inside" }
    if ( $char eq '>' ) { $inoutflag = "outside" }

};    # end of for

#-----

# -- repeat for [ ] brackets-----
$inoutflag = "outside";

for ( $j = 0 ; $j <= $lenstring ; $j = $j + 1 ) {
    $char = substr( $line, $j, 1 );

    if ( ( $char eq '[' ) and ( $inoutflag eq 'inside' ) ) {

        # replace the char with ;
        substr( $line, $j, 1, ";" );
        print " colon-string  = $line\n";
    }
}

```

```

        ## need these at end of the loop
        if ( $char eq '[' ) { $inoutflag = "inside" }
        if ( $char eq ']' ) { $inoutflag = "outside" }

    };    # end of for

    #-----
    ## make an array from the string so we can manipulate the elements
    our @linedata = ();
    @linedata = split( /\s/, $line );

    #-----clean the array-----
    my $E;
    my @cleandata = ();

    foreach $E (@linedata) {
        $E =~ s/^\s+//;    ## clean leading whitespace
        $E =~ s/\s+$//;    ## clean trailing whitespace
        push @cleandata, $E;
    }

    print " colon-array    = @cleandata\n";

    #=====

# Because <infoblocks> can be located inside curved brackets
# as for example, (\sixspot)2, [\sixspot macro contains an infoblock].
# Consequently, we need to remove
# the <..> blocks as parts of a string, not as elements in an array.
# --otherwise, removing the terminal infoblock associated with \sixspot
# will result in also removing the right-hand curved bracket --> error.
# So we return the data as a string, and then send it to sub cutinfoblock later.

    $newdataline = join( ",", @cleandata );
    print "...done\n\n";

    return $newdataline;

```

```

    } ## end of else
} ## end of sub

```

```

#=====

```

2.79 RemoveAllSpaces

```

sub RemoveAllSpaces {

    # remove all spaces in a string
    # Black book page 143

    my $string = $_[0];

    $string =~ s/\s//g;    # OK

    return $string;
}

```

95

```

#=====

```

2.80 CheckSyntax

This subroutine is called at an early stage to check the syntax of each input string. After first removing all spaces from the string it then checks for any unbalanced brackets, and also for illegal pairings of characters. If either (a) any bracket is unbalanced, or (b) there are any illegal pairings of characters then the program issues appropriate error messages and then terminates cleanly.

The syntax restrictions are as follows:

- (1) all functional elements must be comma separated.
- (2) square bracket environment: no commas and no nested brackets
- (3) angle bracket environment: no restrictions.
- (4) curved bracket environment: no nested brackets, and may end with an optional digit (repeat number).

```

sub CheckSyntax {

```

```

## checks that all ( ) { } <> are matched (if any exist)
## checks for other syntax problems, eg missing commas
## if any brackets are not balanced, then we SET an errorflag, and terminate the program

print " SUB CheckSyntax\n";

my $dataline = $_[0];

# first clean out all spaces so we can look for specific strings
$dataline = RemoveAllSpaces($dataline);

print " dataline = $dataline\n";

## count brackets; Angle, Square, Curved
my ( $nleftA, $nrightA ) = 0;
my ( $nleftS, $nrightS ) = 0;
my ( $nleftC, $nrightC ) = 0;
my ( $leftsum, $rightsum ) = 0;

## Blackbook p 139 - counting chars in a string
$nleftA = ( $dataline =~ tr/</</ );
$nrightA = ( $dataline =~ tr/>/>/ );

$nleftS = ( $dataline =~ tr/[/[/[/ );
$nrightS = ( $dataline =~ tr/]/]/]/ );

$nleftC = ( $dataline =~ tr/(/(/ );
$nrightC = ( $dataline =~ tr/)/)/ );

print " left and right <> = $nleftA, $nrightA\n";
print " left and right [] = $nleftS, $nrightS\n";
print " left and right ( ) = $nleftC, $nrightC\n";

$leftsum = $nleftA + $nleftS + $nleftC;
$rightsum = $nrightA + $nrightS + $nrightC;

print " leftsum, rightsum = $leftsum, $rightsum\n";

```

```

# define the current rotation command before any ErrorMessage are issued
# $rotationcommand=$dataline; ## used in writestate sub

my $errorflag = "";

if ( $leftsum != $rightsum ) {

    if ( $nleftS != $nrightS ) {
        gprint(
"...*brackets ERROR [ ] Left [$nleftS not equal to Right $nrightS]"
        );
        ErrorMessage(
            "brackets [ ]: Left [$nleftS not equal to Right $nrightS]");
        $errorflag = "SET";
    }

    if ( $nleftC != $nrightC ) {
        gprint(
"...*brackets ERROR ( ) Left ($nleftC not equal to Right $nrightC)"
        );
        ErrorMessage(
            "brackets ( ): Left ($nleftC not equal to Right $nrightC)");
        $errorflag = "SET";
    }

    if ( $nleftA != $nrightA ) {
        gprint(
"...*brackets ERROR < > Left <$nleftA not equal to Right $nrightA>"
        );
        ErrorMessage(
            "brackets < >: Left <$nleftA not equal to Right $nrightA>");
        $errorflag = "SET";
    }

}

#-----

```



```

        ErrorMessage("$charpair -- syntax error: missing comma");
        $errorflag = "SET";
        next;
    }

    # trap nested curved brackets
    if ( ( $char2 eq "(" ) and ( $curvedflag eq "inside" ) ) {
        ## nested curved brackets
        gprint(".*syntax error: $charpair -- nested ((..))");
        ErrorMessage(
            "$charpair -- syntax error: nested ((..)) not allowed");
        $errorflag = "SET";
    }

    # trap nested square brackets
    if ( ( $char2 eq "[" ) and ( $squareflag eq "inside" ) ) {
        ## nested square brackets
        gprint(".*syntax error: $charpair -- nested [[..]]");
        ErrorMessage(
            "$charpair -- syntax error: nested [[..]] not allowed");
        $errorflag = "SET";
    }

    if ( $squareflag eq "inside" ) {
        if ( $char1 eq "," ) {
            gprint(
                ".*syntax error: $charpair -- comma not allowed in [ ]"
            );
            ErrorMessage(
                "$charpair -- syntax error: comma not allowed in [ ]");
            $errorflag = "SET";
            next;
        }
    }
}
;    # end of if

## detect end of string
if ( $j == $lenstring - 1 ) { last }

```

```

        }
        ;          # end of if

};    # end of for
      #-----

if ( $errorflag eq "SET" ) {

    ## closing down
    gprint(".*Quitting Perl program -- syntax error");
    ErrorMessage("QUITTING PERL PROGRAM -- syntax error");

    print " closing down -- writing state..... OK\n";

    quitprogram();

}
else {
    print " syntax OK; brackets balanced OK\n";
    print " done\n\n";
}

} ## end sub

#=====

```

2.81 inverse

```

sub inverse {

    my $E = $_[0];

    my $lastchar  = substr( $E, -1, 1 );
    my $frontchars = substr( $E, 0,  -1 );    # correct

    if ( $lastchar eq "2" ) { $newE = $E }

```

```
elseif ( $lastchar eq "p" ) { $newE = $frontchars }  
  
else { $newE = $E . "p" }  
  
return $newE;  
}
```

```
##=====
```

—— END ——