mathspic in Perl

Apostolos Syropoulos
366, 28th October Str.
GR-671 00 Xanthi
Greece
email: asyropoulos@yahoo.com

R.W.D. Nickalls
Department of Anaesthesia
Nottingham City Hospital NHS Trust
Hucknall Road
Nottingham, NG5-1PB
United Kingdom
email: dicknickalls@compuserve.com

version 1.10 Feb 18, 2007

Introduction

mathspic is a graphics program which implements a simple programming notation, *mathspic*, suitable for the creation of diagrams or mathematical figures. mathspic's input is a LaTeX file containing mathspic plotting commands. mathspic's output is the equivalent LaTeX file containing PiCTeX plotting commands. Technically, therefore, mathspic is a preprocessor or `filter' for use with the PiCTeX drawing engine. mathspic was originally written in PowerBASIC 3.5, a DOS-based programming language. Since, many potential users are working in rather different programming environments, the authors thought of porting mathspic into another programming cross-platform language which would be widely available. The authors decided to rewrite mathspic in Perl since not only is Perl pretty stable, but it has extensive mathematical support.

Program Structure

Initially, we define a little package that is used to implement the loop command. Then, we must do is to check the possible command line arguments. Next, we process the input file. If the user has used the -b (see below), the program will `beep' if any errors are found during processing. We need some auxiliary subroutines in order to properly parse the input file and of course to handle the various commands. We also need a few global variables.

```
<*>=
#!/usr/bin/perl
#
#(c) Copyright 2005-2007
#                    Apostolos Syropoulos   &  R.W.D. Nickalls
#                    asyropoulos@yahoo.com     dicknickalls@compuserve.com
#
# 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.
#
<package DummyFH >
package main;
use Math::Trig;
<Define global variables>
<subroutine definitions>
<Check for command line arguments>
<process file>
print $alarm if $no_errors > 0;
__END__
```

The package DummyFH is used in the implementation of the loop command. It creates a dummy filehandle that is associated with an array of strings. Since we only read data from this dummy filehandle,

we implement the READLINE subroutine. When we read a line from this dummy filehandle, we actually requesting the next entry of the array (if any). That is why we use the package variable $index. When there are no more entries in the array, subroutine READLINE returns the value undef so to falsify loop that controls the consumption of input from this dummy filehandle.

```
<package DummyFH >= (<-U)
package DummyFH;
my $index = 0;
sub TIEHANDLE {
    my $class = shift;
    my $self= shift;
    bless $self, $class;
}
sub READLINE {
    my $self = shift;
    #shift @$self;
    if ($index > $#$self) {
      $index = 0;
      return undef;
    }
    else {
      return $self->[$index++];
    }
}
```

mathspic accepts at most four command-line switches, namely -b for enabling the beep, -s for automatic screen viewing of the output-file, -c for cleaning out all comment-lines, and -o with a following file-name for specifying the output file-name. mathspic requires the name of an existing input-file (the so-called mathspic-file) containing mathspiccommands. If no command-line arguments are supplied, we print a suitable usage message indicating the syntax. For each command-line argument we set a global variable. The default behavior is that the `bell' does not beep and comment-lines are not removed from the output-file.

```
<Check for command line arguments>= (<-U)
our $alarm="";
our $comments_on=1;
our $out_file="default";
our $argc=@ARGV;
if ($argc == 0 || $argc > 5 ){ # no command line arguments or more than 4
                               # arguments
  die "\nmathspic version $version_number\n" .
      "Usage: mathspic [-h] [-b] [-c] [-o <out file>] <in file>\n\n";
}
else {
  <Process command line arguments>
  print "This is mathspic version $version_number\n";
}
<Check if .m file exists>
```

In order to get the various command-line arguments we use a simple while loop that checks each element of the array @ARGV. We check for all the switches, and we get the name of the input-file.

```
<Process command line arguments>= (<-U)
   our $file = "";
   SWITCHES:
   while($_ = $ARGV[0]) {
     shift;
     if (/^-h$/) {
       die "\nThis is mathspic version $version_number\n" .
           "Type \"man mathspic\" for detailed help\n".
           "Usage:\tmathspic  [-h] [-b] [-c] [-o <out file>] <in file>\n" .
           "\twhere,\n" .
```

```
            "\t[-b]\tenables bell sound if error exists\n" .
            "\t[-c]\tdisables comments in ouput file\n" .
            "\t[-h]\tgives this help listing\n" .
            "\t[-o]\tcreates specified output file\n\n";
      }
      elsif (/^-b$/) {
        $alarm = chr(7);
      }
      elsif (/^-c$/) {
        $comments_on = 0;
      }
      elsif (/^-o$/) {
        die "No output file specified!\n" if !@ARGV;
        $out_file = $ARGV[0];
        shift;
      }
      elsif (/^-\w+/) {
        die "$_: Illegal command line switch!\n";
      }
      else {
        $file = $_;
      }
    }
    die "No input file specified!\n" if $file eq "";
```

In order to check whether the input-file exists, we simply use the `-e` operator. First we check to see if `$file` exits. If the input-file does exist then the variable `$file` contains the file name. In case the user has not specified an output file, the default output file name is the name of the input file with extension `.mt`. Finally, the program outputs all error messages to the screen and to a log file. The name of the log file consists of the contents of the variable `$file` and the extension `.mlg`.

```
<Check if .m file exists>= (<-U)
    our ($source_file, $log_file);
    if (! -e $file) {
       die "$file: no such file!\n" if (! (-e "$file.m"));
       $source_file = "$file.m";
    }
    else {
       $source_file = $file;
       $file = $1 if $file =~ /(\w[\w-\.]+)\.\w+/;
    }
    $out_file= "$file.mt" if $out_file eq "default";
    $log_file= "$file.mlg";
```

Now that we have all the command line arguments, we can start processing the input file. This is done by calling the subroutine `process_input`. Before that we must open all necessary files. Next, we print some `header' information to the output file and to the log file.

```
<process file>= (<-U)
  open(IN,"$source_file")||die "Can't open source file: $source_file\n";
  open(OUT,">$out_file")||die "Can't open output file: $out_file\n";
  open(LOG,">$log_file")||die "Can't open log file: $log_file\n";
  print_headers;
  process_input(IN,"");
```

In this section we define a few global variables. More specifically: the variable `$version_number` contains the current version number of the program, the variable `$commandLineArgs` contains the command line arguments. These two variables are used in the `print_headers` subroutine. The variable `$command` will contain the whole current input line. Hash `%PointTable` is used to store point names and related information. Hash `%VarTable` is used to store mathspic variable names and related information, while the associative array `%ConstTable` contains the names of constants. Note that the

values of both constants and variables are kept in %VarTable. The variable $no_errors is incremented whenever the program encounters an error in the input file. The variables $xunits, $yunits and $units are related to the paper command. In particular, the variable $units is used to parse the unit part of the unit part of the paper command. The variable $defaultsymbol is used to set the point shape. The constant PI holds the value of the mathematical constant pi. The constant R2D holds the transformation factor to transform radians to degrees. The constant D2R holds the transformation factor to transform degrees to radians, i.e., the value 1/R2D. The global variables $arrowLength, $arrowAngleB and $arrowAngleC are actually parameters that are used by the subroutines that draw arrows. Since $arrowLength is actually a length, variable $arrowLenghtUnits holds the units of measure in which this length is expressed. The hash table %DimOfPoint contains the side or the radius of a point whose plot-symbol is a square or a circle, respectively. In case the default point symbol is a circle or a square, variable $GlobalDimOfPoints is used to store the length of the radius or the length of the side of default point symbol, respectively. Variable $LineThickness holds the current line thickness (the default value is 0.4 pt).

```
<Define global variables>= (<-U)
our $version_number = "1.10 Feb 18, 2007";
our $commandLineArgs = join(" ", @ARGV);
our $command  = "";
our $curr_in_file = "";
our %PointTable = ();
our %VarTable = ();
our %ConstTable = ();
our $no_errors = 0;
our $xunits = "1pt";
our $yunits = "1pt";
our $units = "pt|pc|in|bp|cm|mm|dd|cc|sp";
our $defaultsymbol = "\$\\bullet\$";
our $defaultLFradius = 0;
use constant PI => atan2(1,1)*4;
use constant R2D => 180 / PI;
use constant D2R => PI / 180;
our $arrowLength = 2;
our $arrowLengthUnits = "mm";
our $arrowAngleB = 30;
our $arrowAngleC = 40;
our %DimOfPoint = ();
our $GlobalDimOfPoints = 0;
our @Macros = ();
our $LineThickness = 0.4;
```

In this section we define the various subroutines that are needed in order to process the input file.

Subroutine mpp is a mathspic preprocessor that allows the definition and use of macros with or without arguments. For the moment it is an experimental feature and it should be used with care.

Subroutine PrintErrorMessage is used to print error messages to the screen, to the output file and to the log file.

Subroutine PrintWarningMessage is used to print warning messages to the screen, to the output file and to the log file.

Subroutine PrintFatalError is used to print an error message to the screen and to abort execution, where the error is considered fatal and not recoverable.

Subroutine chk_lparen checks whether the next input character is a left parenthesis. Subroutine chk_rparen checks whether the next input character is a right parenthesis. Subroutine chk_comment checks whether a given command is followed by a trailing comment. In the same spirit, we define the subroutines chk_lcb, chk_rcb, chk_lsb, and chk_rsb which check for opening and closing curly

and square brackets respectively. The subroutine `chk_comma` checks whether the next token is a comma.

Subroutine `print_headers` is used to print a header to the output file, so a user knows that the file has been generated by `mathspic`.

Subroutine `get_point` is used to parse a point name and to check whether the point exists (i.e whether the point has been defined).

Subroutine `perpendicular` is used to compute the coordinates of the foot of perpendicular line from some point P to a line AB.

Subroutine `Length` is used to compute the distance between two points A and B.

Subroutine `triangleArea` computes the area of a triangle defined by three points.

Subroutine `PointOnLine` is used to compute the coordinates of a point on a line segment AB and a distance d units from A towards B.

Subroutine `circumCircleCenter` takes six arguments that are the coordinates of three points and computes the center of the circle that passes through the three points which define the triangle.

Subroutine `ComputeDist` is used to compute a numeric value that is specified by either a variable name, a pair of points, or just a number.

Subroutine `intersection4points` is used to compute the coordinates of the point of intersection of two lines specified by the four arguments (i.e. two arguments for each point).

Subroutine `IncircleCenter` is used to compute the center and the radius of a circle that touches internally the sides of a triangle, the coordinates of the three points which define the triangle being the arguments of the subroutine.

Subroutine `Angle` determines the opening in degrees of an angle defined by three points which are the arguments of this subroutine.

Subroutine `excircle` computes the center and the radius of a circle that externally touches a given side (4th and 5th arguments) of triangle (determined by the 1st, the 2nd and the 3rd argument).

Subroutine `DrawLineOrArrow` is used to parse the arguments of the commands `drawline`, `drawthickline`, `drawarrow`, `drawthickarrow` and `drawCurve`.

Subroutine `drawarrows` is used to draw one or more arrows between points.

Subroutine `drawlines` is used to draw one or more lines between points.

Subroutine `drawCurve` is used to draw a curve between an odd number of points.

Subroutine `drawpoints` is used to draw the point symbol of one or more points.

Subroutine `drawAngleArc` is used to draw an arc line within an angle.

Subroutine `drawAngleArrow` is used to draw an arc line with an arrow on the end, within an angle.

Subroutine `expr` and subroutines `term`, `factor` and `primitive` are used to parse an expression that follows a variable declaration.

Subroutine `memberOf` is used to determine whether a string is a member of a list of strings.

Subroutine `tand` computes the tangent of an angle, where the angle is expressed in degrees.

Subroutine `get_string` scans a string in order to extract a valid mathspic string.

Subroutine `is_tainted` checks whether a string contains data that may be proved harmful if used as arguments to a shell escape.

Subroutine `noOfDigits` has one argument which is a number and returns the number of decimal digits it has.

Subroutine `drawsquare` has one argument which is the radius of point and yields LaTeX code that draws a square.

Subroutine `X2sp` can be used to transform a length to sp units.

Subroutine `sp2X` can be used to transform a length expressed in sp units to any other acceptable unit.

Subroutine `setLineThickness` is used to determine the length of the linethickness in the current paper units.

Subroutine `process_input` parses the input file and any other file being included in the main file, and generates output.

Subroutine mpp is an implementation of a mathspic preprocessor that allows the definition of one-line macros with or without arguments. Macro definition has the following syntax:

"%def" macro_name "(" [parameters] ")" macro_code

where parameters is a list of comma separated strings (e.g., x,y,z). Once a macro is defined it can be used or it can be undefined. To undefine a macro one has to use the following command:

"%undef" [macro_name]

If the current input line starts with %def, then we assume that we have a macro definition. We parse each component of the macro definition and finally we store the macro name, the macro code and the macro parameters (if any) in an anonymous hash that eventually becomes part of an array. If we encounter any error, we simply skip to the next line after printing a suitable error message. Now, if the first tokens of an input line are %undef, we assume the user wants to delete a macro. In case these tokens are not followed by a macro name or the macro name has not been defined we simply go on. Otherwise, we delete the corresponding macro data from the global array @Macros that contains all the macro information. Macro expansion is more difficult and it will be described in detail in a separate document. At this point we would like to thank Joachim Schneider for a suggestion on improving macro expansion.

<subroutine mpp >= (<-U)

```
sub mpp {
  my $in_line;
  chomp($in_line = shift);
  my $LC = shift;
  my $out_line = $in_line;
  my $macro_name = "";
  my @macro_param = ();
  my $macro_code = "";
  if ($in_line =~ s/^%def\s*//) {
    if ($in_line =~ s/^(\w+)\s*//){
      $macro_name = $1;
    }
    else {
      PrintErrorMessage("No macro name has been found",$LC);
      return ""
    }
    if ($in_line =~ s/^\(\s*//) {
      # do nothing
    }
    else {
      PrintErrorMessage("No left parenthesis after macro name has been found",$LC);
      return "";
    }
    if ($in_line =~ s/^\)//) {
      # Macro has no parameters!
    }
    else {
      MACROS: while (1) {
        if ($in_line =~ s/^(\w+)\s*//) {
          push (@macro_param, $1);
        }
        else {
          PrintErrorMessage("No macro parameter name has been found",$LC);
          return "";
```

```
      }
      if ($in_line =~ s/^,\s*//) {
        next MACROS;
      }
      else {
        last MACROS;
      }
    }
    if ($in_line =~ s/^\)//) {
      # do nothing!
    }
    else {
      PrintErrorMessage("No closing parenthesis after macro parameters",$LC);
      return "";
    }
  }
  $in_line =~ s/([^%]+)(%.*)/$1/;
  $macro_code = $in_line;
  push ( @Macros , { 'macro_name' => $macro_name,
                     'macro_code' => $macro_code,
                     'macro_param' => \@macro_param });
  return $out_line;
}
elsif ($in_line =~ s/^%undef\s*//) {
  if ($in_line =~ s/^(\w+)//) {
    my $undef_macro = $1;
    for(my $i = $#Macros; $i >= 0; $i--) {
      if ($Macros[$i]->{'macro_name'} eq $undef_macro) {
       splice(@Macros,$i,1);
      }
    }
  }
  return $out_line;
}
elsif ($in_line =~ s/^\s*%//) {
  return $out_line;
}
else {
  my $comment = $2 if $in_line =~ s/([^%]+)(%.+)/$1/;
  EXPANSIONLOOP: while () {
    my $org_in_line = $in_line;
    for(my $i = $#Macros; $i >= 0; $i--) {
      my $macro_name = $Macros[$i]->{'macro_name'};
      if ($in_line =~ /&$macro_name\b/) {       ############################
        my $num_of_macro_args = @{$Macros[$i]->{'macro_param'}};
        if ( $num_of_macro_args > 0 ) {
        # Macro with parameters
          my $pattern = "&$macro_name\\(";
          foreach my $p ( 1..$num_of_macro_args ) {
            my $comma = ($p == $num_of_macro_args) ? "\\s*" : "\\s*,\\s*";
            $pattern .= "\\s*[^\\s\\)]+$comma";
          }
          $pattern .= "\\)";
          while($in_line =~ /&$macro_name\b/) {
            if ($in_line =~ /$pattern/) {
              my $before = $`;
              my $after = $';
              my $match = $&;
              my $new_code = $Macros[$i]->{'macro_code'};
              $match =~ s/^&$macro_name\(\s*//;
              $match =~ s/\)$//;
              foreach my $arg ( 0..($num_of_macro_args - 1) ) {
                my $old = $Macros[$i]->{'macro_param'}->[$arg];
                my $comma = ($arg == ($num_of_macro_args - 1)) ? "" : ",";
                $match =~ s/^\s*([^\s,]+)\s*$comma//;
                my $new = $1;
                # 'g': Parameter may occur several times
                # in $new_code.
                # '\b': Substitute only whole words
```

```
                # not x in xA
                $new_code =~ s/\b$old\b/$new/g;
              }
              $in_line = "$before$new_code$after";
            }
            else {
              PrintErrorMessage("Usage of macro &$macro_name does not " .
                                "match its definition", $LC);
              return "";
            }
          }
        }
        else {
          # Macro without parameters
          my $replacement = $Macros[$i]->{'macro_code'};
          # '\b': Substitute only whole words
          # not x in xA
          $in_line =~ s/&$macro_name\b/$replacement/g;
        }
      }
    }
    last EXPANSIONLOOP if ( $org_in_line eq $in_line );
  }
  return "$in_line$comment";
 }
}
```

Subroutine PrintErrorMessage has two parameters: the error message that will be printed on the screen, the log file and the output file, and the line number of the line containing the error was detected. The general form of the error message is the following:

```
line X: paper{units(
                    ,mm)xrange(0,20)yrange(0,30)axes(B)ticks(10,10)}

***Error: Error_Message
```

where X denotes the line number and Error_Message is the actual error message. Note, that we print the tokens processed so far and on the text line the unprocessed tokens, so that the user knows exactly where the error is. In the variable $A we store the processed tokens, while the variable $l holds the length of $A plus the length of the $error_line (that is the number of the input line where the error occurred) plus 7, i.e., 4 (the length of the word line) plus 2 (the two blank spaces) plus 1 (the symbol :). Finally, we increment the error counter (variable $no_errors). Note, that in case the user has specified the -c command line switch, we will not print any messages to the output file.

```
<subroutine PrintErrorMessage >= (<-U)
    sub PrintErrorMessage {
      my $errormessage = shift;
      my $error_line   = shift;
      my ($l,$A);
      $l = 1+length($command)-length;
      $A = substr($command,0,$l);
      $l += 7 +length($error_line);

      for my $fh (STDOUT, LOG) {
        print $fh "$curr_in_file", "Line $error_line: $A\n";
        print $fh " " x $l  ,$_,"***Error: $errormessage\n";
      }
      if ($comments_on) {  #print to output file file
        print OUT "%% *** $curr_in_file", "Line $error_line: $A\n";
```

```
        print OUT "%% *** "," " x $l  ,$_,"%% ... Error: $errormessage\n";
      }
      $no_errors++;
    }
```

Subroutine PrintWarningMessage behaves exactly like the subroutine PrintErrorMessage. The only difference is that the second subroutine prints only a warning message. A warning is issued when the system detects parameters that do nothing.

```
<subroutine PrintWarningMessage >= (<-U)
    sub PrintWarningMessage {
      my $warningMessage = shift;
      my $warning_line   = shift;
      my ($l,$A);
      $l = 1+length($command)-length;
      $A = substr($command,0,$l);
      $l += 7 +length($warning_line);

      for my $fh (STDOUT, LOG) {
        print $fh "$curr_in_file", "Line $warning_line: $A\n";
        print $fh " " x $l  ,$_,"***Warning: $warningMessage\n";
      }
      if ($comments_on) {  #print to output file file
        print OUT "%% *** $curr_in_file", "Line $warning_line: $A\n";
        print OUT "%% *** "," " x $l  ,$_,"%% ... Warning: $warningMessage\n";
      }
    }
```

The subroutine PrintFatalError behaves similarly to the subroutine PrintErrorMessage. It prints an error message to the screen and aborts execution.

```
<subroutine PrintFatalError >= (<-U)
    sub PrintFatalError {
      my $FatalMessage = shift;
      my $fatal_line   = shift;
      my ($l,$A);
      $l = 1+length($command)-length;
      $A = substr($command,0,$l);
      $l += 7 +length($fatal_line);

      die "$curr_in_file", "Line $fatal_line: $A\n" .
          (" " x $l) . $_ . "***Fatal Error: $FatalMessage\n";
    }
```

The subroutine chk_lparen accepts two arguments: the name of the token that should be immediately before the left parenthesis (variable $token), and the current line number (variable $lc). First we skip any leading white space and then check whether the next input character is a left parenthesis, then the subroutine skips any trailing white space; otherwise it prints an error message.

```
<subroutine chk_lparen >= (<-U)
sub chk_lparen {
  my $token = $_[0];
  my $lc    = $_[1];
  s/\s*//;
  if (/^[^\(]/) {
    PrintErrorMessage("Missing ( after $token",$lc);
  }
  else {
```

```
    s/^\(\s*//;
  }
}
```

The subroutine chk_rparen accepts two parameters: the name of the token that should be immediately after a right parenthesis (variable $token), and the current line number (variable $lc). Initially, we skip any leading white space and then we check whether the next input token is a right parenthesis. If it is not we issue a error message and return, otherwise we skip the parenthesis and any trailing white space.

```
<subroutine chk_rparen >= (<-U)
sub chk_rparen {
  my $token = $_[0];
  my $lc    = $_[1];
  s/\s*//;
  if (s/^\)//) {
    s/\s*//;
  }
  else {
    PrintErrorMessage("Missing ) after $token",$lc);
  }
}
```

The subroutine chk_lcb behaves in a similar way to the subroutine chk_lparen.

```
<subroutine chk_lcb >= (<-U)
sub chk_lcb {
  my $token = $_[0];
  my $lc    = $_[1];
  s/\s*//;
  if ($_ !~ /^\{/) {
    PrintErrorMessage("Missing { after $token",$lc);
  }
  else {
    s/^{\s*//;
  }
}
```

Subroutine chk_rcb behaves in a similar way to the subroutine chk_rparen.

```
<subroutine chk_rcb >= (<-U)
sub chk_rcb {
  my $token = $_[0];
  my $lc    = $_[1];
  if ($_ !~ /^\s*\}/) {
    PrintErrorMessage("Missing } after $token",$lc);
  }
  else {
    s/^\s*}\s*//;
  }
}
```

Subroutine chk_lsb behaves in a similar way to the subroutine chk_lparen.

```
<subroutine chk_lsb >= (<-U)
sub chk_lsb {
   my $token = $_[0];
   my $lc    = $_[1];

   s/\s*//;
   if ($_ !~ /^\[/) {
     PrintErrorMessage("Missing [ after  $token",$lc);
   }
   else {
      s/^\[\s*//;
   }
}
```

Subroutine chk_rsb behaves in a similar way to the subroutine chk_rparen.

```
<subroutine chk_rsb >= (<-U)
sub chk_rsb {
   my $token = $_[0];
   my $lc    = $_[1];

   s/\s*//;
   if ($_ !~ /^\]/) {
     PrintErrorMessage("Missing ] after  $token",$lc);
   }
   else {
     s/^\]\s*//;
   }
}
```

The subroutine chk_comma checks whether the next token is a comma. If it is not then it prints an error message, otherwise it consumes the comma and any white space that follows the comma.

```
<subroutine chk_comma >= (<-U)
sub chk_comma {
  my $lc  = $_[0];

  s/\s*//;
  if (/^[^,]/) {
    PrintErrorMessage("Did not find expected comma",$lc);
  }
  else {
    s/^,\s*//;
  }
}
```

The subroutine chk_comment has only one parameter which is the current line number. It checks whether the next input character is a comment character and in this case it does nothing!. Otherwise, if there is some trailing text it simply prints a warning to the screen.

```
<subroutine chk_comment >= (<-U)
sub chk_comment {
   my $lc = $_[0];

   s/\s*//;
   if (/^%/) {
     # do nothing!
   }
   elsif (/^[^%]/) {
```

```
    PrintWarningMessage("Trailing text is ignored",$lc);
   }
}
```

The subroutine print_headers prints a header to the output file, as well as a header to the LOG file. The header contains information regarding the version of the program, a copyright notice, the command line, date and time information, and the names of the various files processed/generated.

```
<subroutine print_headers >= (<-U)
  sub print_headers
  {
     my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime;
     $year+=1900;
     $mon+=1;
     $now_string = "$year/" . ($mon>9 ? "$mon/" : "0$mon/") .
                   ($mday>9 ? "$mday    " : "0$mday    ") .
                   ($hour>9 ? "$hour:" : "0$hour:") .
                   ($min>9 ? "$min:" : "0$min:") .
                   ($sec>9 ? "$sec" : "0$sec");
     print OUT "%* ---------------------------------------------\n";
     print OUT "%* mathspic (Perl version $version_number)\n";
     print OUT "%* A filter program for use with PiCTeX\n";
     print OUT "%* Copyright (c) 2005 A Syropoulos & RWD Nickalls \n";
     print OUT "%* Command line: $0 $commandLineArgs\n";
     print OUT "%* Input filename : $source_file\n";
     print OUT "%* Output filename: $out_file\n";
     print OUT "%* Date & time: $now_string\n";
     print OUT "%* ---------------------------------------------\n";
     #
     print LOG "----\n";
     print LOG "$now_string\n";
     print LOG "mathspic (Perl version $version_number)\n";
     print LOG "Copyright (c) 2005 A Syropoulos & RWD Nickalls \n";
     print LOG "Input file  = $source_file\n";
     print LOG "Output file = $out_file\n";
     print LOG "Log file    = $log_file\n";
     print LOG "----\n";
  }
```

The subroutine get_point parses an individual point name. If the next token is also a point name then it returns the point name (but only if the only if the point name exists in the PointTable). In all other cases it returns the string _undef_ to indicate that something is wrong.

```
<subroutine get_point >= (<-U)
   sub get_point {

      my ($lc) = $_[0];
      my ($PointName);

      if (s/^([^\W\d_]\d{0,3})\s*//i) { #point name
        $PointName = $1;
        if (!exists($PointTable{lc($PointName)})) {
          PrintErrorMessage("Undefined point $PointName",$lc);
          return "_undef_";
        }
        else {
          return lc($PointName);
        }
      }
      else {
```

```
        PrintErrorMessage("Point name expected",$lc);
        return "_undef_";
      }
  }
```

The subroutine perpendicular has 6 parameters that correspond to the coordinates of some point P and to the coordinates of two points A and B that define a line. The subroutine returns a pair of numbers that correspond to the coordinates of a point that lies at the foot of the perpendicular to the line AB that passes through point P. The slope of line AB is m_1 and so its equation is $y=m_1x+c_1$. Similarly, the slope of the line PF is $m_2=-1/m_1$ and its equation is $y=m_2x+c_2$. Since the line AB passes through A, then $c_1=y_A-m_1x_A$. Similarly, as P is on line PF, then $c_2=y_P-m_2x_P$. Now point F is on both lines, therefore $y_F=m_2x_F+c_2$ and $y_F=m_1x_F+c_1$. Solving these equations for x_F and y_F gives:

$$x_F=(c_2-c_1)/(m_1-m_2)$$
$$y_F=(m_1c_2-m_2c_1)/\ (m_1-m_2)$$

<subroutine perpendicular >= (<-U)
```
sub perpendicular {
    my ($xP, $yP, $xA, $yA, $xB, $yB) = @_;
    my ($xF, $yF, $deltax, $deltay, $m1, $m2, $c1, $c2, $factor);

    $deltax = $xA - $xB;
    return ($xA, $yP) if abs($deltax) < 0.0000001;
    $deltay = $yA - $yB;
    return ($xP, $yA) if abs($deltay) < 0.0000001;
    $m1 = $deltay / $deltax;
    eval { $m2 = (-1) / $m1;};
    PrintFatalError("Division by zero",$lc) if $@;
    $c1 = $yA - $m1 * $xA;
    $c2 = $yP - $m2 * $xP;
    eval { $factor = 1 / ($m1 - $m2)};
    PrintFatalError("Division by zero",$lc) if $@;
    return (($c2 - $c1) * $factor, ($m1 * $c2 - $m2 * $c1) * $factor);
}
```

The subroutine Length computes the distance between two points A and B. Notice, that the name of the subroutine starts with a capital L, just to avoid conflict with the predefined Perl function. The subroutine requires four parameters which are the coordinates of the two points.

<subroutine Length >= (<-U)
```
sub Length {
    my ($xA, $yA, $xB, $yB)=@_;
    return sqrt(($xB - $xA)**2 + ($yB - $yA)**2);
}
```

The subroutine triangleArea computes the area of a triangle by using Heron's formula, i.e., given a triangle ABC, we first compute s=(AB+BC+CA)/2 and then the area of the triangle is equal to the square root of s times (s-AB) times (s-BC) times (s-BA), where AB, BC, and CA are the lengths of the three sides of the triangle. The subroutine accepts 6 parameters, which correspond to the coordinates of three points that define the triangle.

<subroutine triangleArea >= (<-U)
```
  sub triangleArea {
      my ($xA, $yA, $xB, $yB, $xC, $yC)=@_;
      my ($lenAB, $lenBC, $lenCA, $s);

      $lenAB = Length($xA,$yA,$xB,$yB);
      $lenBC = Length($xB,$yB,$xC,$yC);
      $lenCA = Length($xC,$yC,$xA,$yA);
      $s = ($lenAB + $lenBC + $lenCA) / 2;
      return sqrt($s * ($s - $lenAB)*($s - $lenBC)*($s - $lenCA));
  }
```

The subroutine poinOnLine accepts five arguments: the coordinates of two points and the decimal number which corresponds to the distance from the first point towards the second one. The way we compute the coordinates of the point is fairly simple.

<subroutine pointOnLine >= (<-U)
```
  sub pointOnLine {
      my ($xA, $yA, $xB, $yB, $dist)=@_;
      my ($deltax, $deltay, $xPol, $yPol);

      $deltax = $xB - $xA;
      $deltay = $yB - $yA;
      $xPol = $xA + ($dist * $deltax / &Length($xA,$yA,$xB,$yB));
      $yPol = $yA + ($dist * $deltay / &Length($xA,$yA,$xB,$yB));
      return ($xPol, $yPol);
  }
```

As we have mentioned above the subroutine circumCircleCenter takes six arguments that correspond to the coordinates of three points that define a triangle. The subroutine computes the coordinates of the center of a circle that passes through these three points, and the radius of the circle. We now describe how the subroutine computes the center of the circle and its radius. Let the triangle points be t1, t2 and t3. We use the two pairs of points to define two sides, i.e., t1t2 and t2t3. For each side we locate the midpoints and get the their coordinates. We check whether either of these two lines is either vertical or horizontal. If this is true, we know that one of the coordinates of the center of the circumcircle is the same as that of the midpoints of the horizontal or vertical line. Next, we determine the slopes of the lines t1t2 and t2t3. We now determine the slope of lines at right-angles to these lines. We solve the resulting equations and obtain the center of the circumcircle. Now we get the radius, and then we are done.

<subroutine circumCircleCenter >= (<-U)

```
  sub circumCircleCenter {
     my ($xA, $yA, $xB, $yB, $xC, $yC, $lc)=@_;
     my ($deltay12, $deltax12, $xs12, $ys12);
     my ($deltay23, $deltax23, $xs23, $ys23);
     my ($xcc, $ycc);
     my ($m23, $mr23, $c23, $m12, $mr12, $c12);
     my ($sideA, $sideB, $sideC, $a, $radius);

     if (abs(triangleArea($xA, $yA, $xB, $yB, $xC, $yC)) < 0.0000001)
     {
        PrintErrorMessage("Area of triangle is zero!",$lc);
        return (0,0,0);
     }
```

```
$deltay12 = $yB - $yA;
$deltax12 = $xB - $xA;
$xs12 = $xA + $deltax12 / 2;
$ys12 = $yA + $deltay12 / 2;
#
$deltay23 = $yC - $yB;
$deltax23 = $xC - $xB;
$xs23 = $xB + $deltax23 / 2;
$ys23 = $yB + $deltay23 / 2;
#
CCXYLINE:{
if (abs($deltay12) < 0.0000001)
{
   $xcc = $xs12;
   if (abs($deltax23) < 0.0000001)
   {
      $ycc = $ys23;
      last CCXYLINE;
   }
   else
   {
      $m23 = $deltay23 / $deltax23;
      $mr23 = -1 / $m23;
      $c23 = $ys23 - $mr23 * $xs23;
      $ycc = $mr23 * $xs12 + $c23;
      last CCXYLINE;
   }
}
if (abs($deltax12) < 0.0000001)
{
   $ycc = $ys12;
   if (abs($deltay23) < 0.0000001)
   {
      $xcc = $xs23;
      last CCXYLINE;
   }
   else
   {
      $m23 = $deltay23 / $deltax23;
      $mr23 = -1 / $m23;
      $c23 = $ys23 - $mr23 * $xs23;
      $xcc = ($ys12 - $c23) / $mr23;
      last CCXYLINE;
   }
}
if (abs($deltay23) < 0.0000001)
{
   $xcc = $xs23;
   if (abs($deltax12) < 0.0000001)
   {
      $ycc = $ys12;
      last CCXYLINE;
   }
   else
   {
      $m12 = $deltay12 / $deltax12;
      $mr12 = -1 / $m12;
      $c12 = $ys12 - $mr12 * $xs12;
      $ycc = $mr12 * $xcc + $c12;
      last CCXYLINE;
   }
}
if (abs($deltax23) < 0.0000001)
{
   $ycc = $ys23;
   if (abs($deltay12) < 0.0000001)
   {
      $xcc = $xs12;
      last CCXYLINE;
```

```
        }
        else
        {
           $m12 = $deltay12 / $deltax12;
           $mr12 = -1 / $m12;
           $c12 = $ys12 - $mr12 * $xs12;
           $xcc = ($ycc - $c12) / $mr12;
           last CCXYLINE;
        }
     }
     $m12 = $deltay12 / $deltax12;
     $mr12 = -1 / $m12;
     $c12 = $ys12 - $mr12 * $xs12;
     #-----
     $m23 = $deltay23 / $deltax23;
     $mr23 = -1 / $m23;
     $c23 = $ys23 - $mr23 * $xs23;
     $xcc = ($c23 - $c12) / ($mr12 - $mr23);
     $ycc = ($c23 * $mr12 - $c12 * $mr23) / ($mr12 - $mr23);
     }
     #
     $sideA = &Length($xA,$yA,$xB,$yB);
     $sideB = &Length($xB,$yB,$xC,$yC);
     $sideC = &Length($xC,$yC,$xA,$yA);
     $a = triangleArea($xA, $yA, $xB, $yB, $xC, $yC);
     $radius = ($sideA * $sideB * $sideC) / (4 * $a);
     #
     return ($xcc, $ycc, $radius);
  }
```

The subroutine ComputeDist is used to compute a distance that is specified by either a float number, a pair of points, or a variable name. In case we have a pair of identifiers, we check whether the first one is a point. If it isn't a point we assume we have a variable followed by a keyword. Otherwise, i.e., if it is a point name, we check whether the second identifier is also a point name. If it is, we simply return the distance between them, otherwise we issue an error message. If we have only a single identifier, we check whether it is a variable that has already been defined, and if so we return its value. Since, this subroutine is heavily used, it actually returns a pair of numbers: the first one being the computed distance and the second one being an error indicator. If the value of this indicator is 0, then there is no error. If its value is 1, then there is an error. Moreover, in case there is an error the distance is assumed to be equal to zero.

```
<subroutine ComputeDist >= (<-U)
  sub ComputeDist {
     my ($lc) = $_[0];
     my ($v1, $v2);

     if (s/^((\+|-)?\d+(\.\d+)?([eE](\+|-)?\d+)?)//) #is it a number?
     {
         return ($1, 1);
     }
     elsif (/^[^\W\d_]\d{0,3}[^\W\d_]\d{0,3}/) #it is a pair of IDs?
     {
        s/^([^\W\d_]\d{0,3})//i;
        $v1 = $1;
        if (!exists($PointTable{lc($v1)})) {
           if (exists($VarTable{lc($v1)})) {
              return ($VarTable{lc($v1)}, 1);
           }
           PrintErrorMessage("Point $v1 has not been defined", $lc);
```

```
            s/^\s*[^\W\d_]\d{0,3}//i;
            return (0,0);
         }
         $v1 = lc($v1);
         s/^\s*([^\W\d_]\d{0,3})//i;
         $v2 = $1;
         if (!exists($PointTable{lc($v2)}))
         {
            PrintErrorMessage("Point $v2 has not been defined", $lc);
            return (0,0);
         }
         $v2 = lc($v2);
         my ($x1,$y1,$pSV1,$pS1) = unpack("d3A*",$PointTable{$v1});
         my ($x2,$y2,$pSV2,$pS2) = unpack("d3A*",$PointTable{$v2});
         return (Length($x1,$y1,$x2,$y2), 1);
      }
      elsif (s/^([^\W\d_]\d{0,3})//i) # it is a single id
      {
        $v1 = $1;
        if (!exists($VarTable{lc($v1)})) #it isn't a variable
        {
          PrintErrorMessage("Variable $v1 has not been defined", $lc);
          return (0,0);
        }
        return ($VarTable{lc($v1)}, 1);
      }
      else
      {
         PrintErrorMessage("Unexpected token", $lc);
         return (0,0);
      }
   }
```

The subroutine intersection4points has 8 parameters that correspond to the coordinates of four points that uniquely determine two lines, and computes the the point of intersection of these two lines.

```
<subroutine intersection4points >= (<-U)
   sub intersection4points {
     my ($x1, $y1, $x2, $y2, $x3, $y3, $x4, $y4) = @_;
     my ($deltay12, $deltax12, $deltay34, $deltax34);
     my ($xcc, $ycc, $m34, $c34, $m12, $c12);

     $deltay12 = $y2 - $y1;
     $deltax12 = $x2 - $x1;
     #
     $deltay34 = $y4 - $y3;
     $deltax34 = $x4 - $x3;
     I4PXYLINE:{
        if (abs($deltay12) < 0.0000001)
        {
           $ycc = $y1;
           if (abs($deltax34) < 0.0000001)
           {
              $xcc = $x3;
              last I4PXYLINE;
           }
           else
           {
              $m34 = $deltay34 / $deltax34;
              $c34 = $y3 - $m34 * $x3;
              $xcc = ($ycc - $c34) / $m34;
              last I4PXYLINE;
           }
        }
        if (abs($deltax12) < 0.0000001)
        {
```

```
            $xcc = $x1;
            if (abs($deltay34) < 0.0000001)
            {
               $ycc = $y3;
               last I4PXYLINE;
            }
            else
            {
               $m34 = $deltay34 / $deltax34;
               $c34 = $y3 - $m34 * $x3;
               $ycc = $m34 * $xcc + $c34;
               last I4PXYLINE;
            }
         }
         if (abs($deltay34) < 0.0000001)
         {
            $ycc = $y3;
            if (abs($deltax12) < 0.0000001)
            {
               $xcc = $x1;
               last I4PXYLINE;
            }
            else
            {
               $m12 = $deltay12 / $deltax12;
               $c12 = $y1 - $m12 * $x1;
               $xcc = ($ycc - $c12) / $m12;
               last I4PXYLINE;
            }
         }
         if (abs($deltax34) < 0.0000001)
         {
            $xcc = $x3;
            if (abs($deltay12) < 0.0000001)
            {
               $ycc = $y1;
               last I4PXYLINE;
            }
            else
            {
               $m12 = $deltay12 / $deltax12;
               $c12 = $y1 - $m12 * $x1;
               $ycc = $m12 * $xcc + $c12;
               last I4PXYLINE;
            }
         }
         $m12 = $deltay12 / $deltax12;
         $c12 = $y1 - $m12 * $x1;
         $m34 = $deltay34 / $deltax34;
         $c34 = $y3 - $m34 * $x3;
         $xcc = ($c34 - $c12) / ($m12 - $m34);
         $ycc = ($c34 * $m12 - $c12 * $m34) / ($m12 - $m34);
      }
      return ($xcc, $ycc);
   }
```

The subroutine IncircleCenter computes the center and the radius of the circle that is inside a triangle and touches the sides of the triangle. The subroutine has six arguments that correspond to the coordinates of three points that uniquely determine the triangle. Here are the details:

- Let the triangle points be A, B, C and sides a, b, c, where side B is opposite angle B, etc.
- Use angles A and B only.

- Let the bisector of angle A meet side a in point A1, and let the distance of A1 from B be designated BA1
- Using the sine rule, one gets: BA1/c = a/(b+c), that is BA1 = c * a/(b+c).
- Now do the same for side b, and determine equivalent point B1. CB1/a = b/(b+c), that is CB1 = a * b/(b+c).
- We can now find the intersection of the line from point A to point A1, and the line from point B to point B1. We have four points, so we use the mathspic internal intersection4points subroutine to return the coordinates of the intersection X_i, Y_i.
- Now get the radius: R=(area of triangle)/(a+b+c)/2
- Finally, return the radius and the coordinates of the center.

```
<subroutine IncircleCenter >= (<-U)
  sub IncircleCenter {
     my ($Ax, $Ay, $Bx, $By, $Cx, $Cy) = @_;
     my ($sideA, $sideB, $sideC);
     my ($ba1, $xA1, $yA1, $cb1, $ac1, $xB1, $yB1, $xC1, $yC1, $a, $s, $r);

     #determine the lengths of the sides
     $sideA = Length($Bx, $By, $Cx, $Cy);
     $sideB = Length($Cx, $Cy, $Ax, $Ay);
     $sideC = Length($Ax, $Ay, $Bx, $By);
     #
     $ba1 = ($sideC * $sideA) / ($sideB + $sideC);
     ($xA1, $yA1) = pointOnLine($Bx, $By, $Cx, $Cy, $ba1);
     $cb1 = ($sideA * $sideB) / ($sideC + $sideA);
     ($xB1, $yB1) = pointOnLine($Cx, $Cy, $Ax, $Ay, $cb1);
     $ac1 = ($sideB * $sideC) / ($sideA + $sideB);
     ($xC1, $yC1) = pointOnLine($Ax, $Ay, $Bx, $By, $ac1);
     ($xcenter, $ycenter) = &intersection4points($Ax, $Ay, $xA1, $yA1,
                                                 $Bx, $By, $xB1, $yB1);
     # get radius
     $a = &triangleArea($Ax, $Ay, $Bx, $By, $Cx, $Cy);
     $s = ($sideA + $sideB +$sideC) / 2;
     $r = $a / $s;
     return ($xcenter, $ycenter, $r);
  }
```

The subroutine Angle takes six arguments which correspond to the coordinates of three points that define an angle. The subroutine computes the opening of the angle in degrees. In case there is an error it returns the number -500. ****EXPLAIN THE ALGORITHM****

```
<subroutine Angle >= (<-U)
  sub Angle {
    my ($Ax, $Ay, $Bx, $By, $Cx, $Cy) = @_;
    my ($RAx, $RAy, $RBx, $RBy, $RCx, $RCy, $deltax, $deltay);
    my ($lineBA, $lineBC, $lineAC, $k, $kk, $angle);
    my ($T, $cosT, $sinT) = (0.3, cos(0.3), sin(0.3));

    $RAx = $Ax * $cosT + $Ay * $sinT;
    $RAy = -$Ax * $sinT + $Ay * $cosT;
    $RBx = $Bx * $cosT + $By * $sinT;
    $RBy = -$Bx * $sinT + $By * $cosT;
    $RCx = $Cx * $cosT + $Cy * $sinT;
    $RCy = -$Cx * $sinT + $Cy * $cosT;
    $deltax = $RBx - $RAx;
    $deltay = $RBy - $RAy;
    $lineBA = sqrt($deltax*$deltax + $deltay*$deltay);
    if ($lineBA < 0.0000001)
    {
       return -500;
```

```
    }
    $deltax = $RBx - $RCx;
    $deltay = $RBy - $RCy;
    $lineBC = sqrt($deltax*$deltax + $deltay*$deltay);
    if ($lineBC < 0.0000001)
    {
       return -500;
    }
    $deltax = $RAx - $RCx;
    $deltay = $RAy - $RCy;
    $lineAC = sqrt($deltax*$deltax + $deltay*$deltay);
    if ($lineAC < 0.0000001)
    {
       return -500;
    }
    $k = ($lineBA*$lineBA + $lineBC*$lineBC - $lineAC*$lineAC ) /
         (2 * $lineBA * $lineBC);
    $k = -1 if $k < -0.99999;
    $k = 1 if $k > 0.99999;
    $kk = $k * $k;
    if (($kk * $kk) == 1)
    {
       $angle = PI if $k == -1;
       $angle = 0 if $k == 1;
    }
    else
    {
       $angle = (PI / 2) - atan2($k / sqrt(1 - $kk),1);
    }
    return $angle * 180 / PI;
  }
```

The subroutine excircle computes the center and the radius of a circle that externally touches a given side (4th and 5th arguments) of triangle (determined by the 1rst, the 2nd and 3rd argument). Here are the details:

- Let the triangle points be A, B, C, and the given side be BC.
- Now calculate the radius of Excircle = (triangle area)/(s - side length), where s = (a+b+c)/2
- Calculate the distance from the angle (A) (opposite the given side BC) to the excircle center = radius/sin(A/2)
- Now determine the the Excircle center by locating it on the angle bisector (i.e., same line that the IncircleCenter is on), but at distance d further away from angle A. So, we now have the Incircle center (I), determine deltaX and deltaY from I to A, calculate the distance AI, and then extend the line from I by distance d to Excenter Xc, Yc.

<subroutine excircle >= (**<-U**)

```
  sub excircle {
    my ($A, $B, $C, $D, $E) = @_;
    my ($Ax,$Ay,$Bx,$By,$Dx,$Dy,$Ex,$Ey,$ASVA,$ASA);
    ($Ax,$Ay,$ASVA,$ASA)=unpack("d3A*",$PointTable{$A});
    ($Bx,$By,$ASVA,$ASA)=unpack("d3A*",$PointTable{$B});
    ($Cx,$Cy,$ASVA,$ASA)=unpack("d3A*",$PointTable{$C});
    ($Dx,$Dy,$ASVA,$ASA)=unpack("d3A*",$PointTable{$D});
    ($Ex,$Ey,$ASVA,$ASA)=unpack("d3A*",$PointTable{$E});
    my ($sideA, $sideB, $sideC, $s, $R, $theAdeg, $d);
    my ($Xmypoint, $Ymypoint, $deltax, $deltay, $mylength, $xc, $yc);

    $sideA = &Length($Bx, $By, $Cx, $Cy);
    $sideB = &Length($Cx, $Cy, $Ax, $Ay);
    $sideC = &Length($Ax, $Ay, $Bx, $By);
```

```
    $s = ($sideA + $sideB + $sideC) / 2;
    $R = triangleArea($Ax, $Ay, $Bx, $By, $Cx, $Cy) /
         ($s - &Length($Dx, $Dy, $Ex, $Ey));
    if (($D eq $A && $E eq $B) || ($D eq $B && $E eq $A))
    {
      $theAdeg = &Angle($Bx, $By, $Cx, $Cy, $Ax, $Ay);
      $Xmypoint = $Cx;
      $Ymypoint = $Cy;
    }
    elsif (($D eq $B && $E eq $C) || ($D eq $C && $E eq $B))
    {
      $theAdeg = &Angle($Cx, $Cy, $Ax, $Ay, $Bx, $By);
      $Xmypoint = $Ax;
      $Ymypoint = $Ay;
    }
    elsif (($D eq $C && $E eq $A) || ($D eq $A && $E eq $C))
    {
      $theAdeg = &Angle($Ax, $Ay, $Bx, $By, $Cx, $Cy);
      $Xmypoint = $Bx;
      $Ymypoint = $By;
    }
    else
    {
       return (0,0,0);
    }
    $d = $R  / sin($theAdeg * PI / 180 / 2);
    my ($xIn, $yIn, $rin) = &IncircleCenter($Ax, $Ay, $Bx, $By, $Cx, $Cy);
    $deltax = $xIn - $Xmypoint;
    $deltay = $yIn - $Ymypoint;
    $mylength = sqrt($deltax*$deltax + $deltay*$deltay);
    $xc = $Xmypoint + $d * $deltax / $mylength;
    $yc = $Ymypoint + $d * $deltay / $mylength;
    return ($xc, $yc, $R);
  }
```

The DrawLineOrArrow subroutine is used to parse the arguments of the commands drawline, drawthickline, drawarrow, drawthickarrow and drawCurve. In general, these commands have as arguments a list of points separated by commas that are used to draw a set of lines. The list of points is enclosed in parentheses. Here we give only the syntax of the drawline comma, as the syntax of the other commands is identical:

```
   drawline ::= "drawline"  "(" Points { "," Points } ")"
   Points ::= Point { separator Point}
   separator ::= blank | empty
```

In the following code we scan a list of points (possibly separated by blanks) and we stop when we encounter either a comma or some other character. In case we have found a comma, we check whether we have a drawline command and if this is the case we plot the list of points. We continue with the next list of points, until there are no more points. The inner while-loop is used to control the consumption of point tokens and the external to reset the array PP which holds the point names.

```
<subroutine DrawLineOrArrow >= (<-U)
   sub DrawLineOrArrow {
     my $draw_Line = shift;
     my $lc = shift;
     my $lineLength = -1;
     my $stacklen = 0;
     my @PP = ();
#     if ($draw_Line != 2) {
```

```
#       s/\s*//;
#       if (s/^\[\s*//) { # optional length specifier
#         $lineLength = expr($lc);
#         if ($lineLength <= 0) {
#           PrintErrorMessage("length must greater than zero",$lc);
#           $lineLength = -1;
#         }
#         chk_rsb("optional part",$lc);
#       }
#     }
      chk_lparen("$cmd",$lc);
      DRAWLINES:while(1) {
        @PP = () ;
        while(1) {
          if (s/^([^\W\d_]\d{0,3})\s*//i) { #point name
            $P = $1;
            if (!exists($PointTable{lc($P)})) {
              PrintErrorMessage("Undefined point $P",$lc);
            }
            else {
              push (@PP,$P);
            }
          }
          else {
            $stacklen = @PP;
            if ($draw_Line != 2) {
              if ($stacklen <= 1) {
                PrintErrorMessage("Wrong number of points",$lc);
              }
              else {
                push(@PP,$lc);
                if ($draw_Line == 0) {
                  drawarrows(@PP);
                }
                elsif ($draw_Line == 1) {
                  drawlines(@PP);
                }
              }
            }
            if (s/^,\s*// and $draw_Line != 2) {
              next DRAWLINES;
            }
            else {
              last DRAWLINES;
            }
          }
        }
      }
      if ($draw_Line == 2) {
        $stacklen =  @PP;
        if ($stacklen < 2) {
          PrintErrorMessage("Wrong number of points",$lc);
        }
        elsif ($stacklen % 2 == 0) {
          PrintErrorMessage("Number of points must be odd",$lc);
        }
        else {
          drawCurve(@PP);
        }
      }
      chk_rparen("arguments of $cmd",$lc);
      chk_comment($lc);
    }
```

The subroutine drawarrows is used to draw one or more lines. The subroutine accepts as argument an array which contains the names of the points which define the lines, plus the current program line

number. Each arrow is printed using the following code:

\arrow < ArrowLength mm> [beta , gamma] from x1 y1 to x2 y2

where beta is equal to tan($arrowAngleB * d2r /2) and gamma is equal to 2*tan($arrowAngleC * d2r / 2).

```
<subroutine drawarrows >= (<-U)
  sub drawarrows {
    my ($NoArgs);
    $NoArgs = @_;
    my ($lc) = $_[$NoArgs-1]; #line number is the last argument
    my ($NumberOfPoints, $p, $q, $r12, $d12);
    my ($px,$py,$pSV,$pS, $qx,$qy,$qSV,$qS);

    $NumberOfPoints = $NoArgs - 1;
    LOOP: for(my $i=0; $i < $NumberOfPoints - 1; $i++)
    {
       $p = $_[$i];
       $q = $_[$i+1];
       ($px,$py,$pSV,$pS) = unpack("d3A*",$PointTable{lc($p)});
       ($qx,$qy,$qSV,$qS) = unpack("d3A*",$PointTable{lc($q)});
       $pSV = $defaultLFradius if $pSV == 0;
       $qSV = $defaultLFradius if $qSV == 0;
       $r12 = $pSV + $qSV;
       $d12 = Length($px,$py,$qx,$qy);
       if ($d12 <= $r12)
       {
          if($d12 == 0)
          {
             PrintErrorMessage("points $p and $q are the same", $lc);
             next LOOP;
          }
          PrintWarningMessage("arrow $p$q not drawn: points too close or ".
                              "radii too big", $lc);
          next LOOP;
       }
       ($px, $py) = pointOnLine($px, $py, $qx, $qy, $pSV) if $pSV > 0;
       ($qx, $qy) = pointOnLine($qx, $qy, $px, $py, $qSV) if $qSV > 0;
      my ($beta, $gamma);
       $beta  = tan($arrowAngleB * D2R / 2);
       $gamma = 2 * tan($arrowAngleC * D2R / 2);
       printf OUT "\\arrow <%.5f%s> [%.5f,%.5f] from %.5f %.5f to %.5f %.5f\n",
              $arrowLength, $arrowLengthUnits, $beta, $gamma, $px, $py, $qx, $qy;
    }
  }
```

The subroutine drawlines is used to draw one or more lines. The subroutine accepts as argument an array which contains the names of the points which define the lines, plus the current program line number. If there are only two points (i.e., only one line), then we output the following PiCTeX code:

\plot x1 y1 x2 y2 / %% pointname1 pointname2

If there are more than two points, then we need to write the PiCTeX code in pairs with two points on each line (just to keep things simple) as follows:

\plot x1 y1 x2 y2 / %% pointname1 pointname2 \plot x2 y2 x3 y3 / %% pointname2 pointname3 \plot x3 y3 x4 y4 / %% pointname3 pointname4

An important part of the subroutine is devoted to checking whether either or both of the pairs of points are associated with a line-free zone, and if so, then we must take care not to draw the line inside the line-free zone. If a point does have a line-free zone, then we use the pointOnLine subroutine to determine the point on the line which is just on the line-free boundary, and draw the line to the that point instead of to the exact point-location.

```
<subroutine drawlines >= (<-U)
  sub drawlines {
    my ($NoArgs);
    $NoArgs = @_;
    my ($lc) = $_[$NoArgs-1]; #line number is the last argument
    my ($NumberOfPoints, $p, $q, $r12, $d12);
    my ($px,$py,$pSV,$pS, $qx,$qy,$qSV,$qS);

    $NumberOfPoints = $NoArgs - 1;
    LOOP: for(my $i=0; $i < $NumberOfPoints - 1; $i++)
    {
       $p = $_[$i];
       $q = $_[$i+1];
       ($px,$py,$pSV,$pS) = unpack("d3A*",$PointTable{lc($p)});
       ($qx,$qy,$qSV,$qS) = unpack("d3A*",$PointTable{lc($q)});
       $pSV = $defaultLFradius if $pSV == 0;
       $qSV = $defaultLFradius if $qSV == 0;
       $r12 = $pSV + $qSV;
       $d12 = Length($px,$py,$qx,$qy);
       if ($d12 <= $r12)
       {
          if($d12 == 0)
          {
             PrintErrorMessage("points $p and $q are the same", $lc);
             next LOOP;
          }
          PrintWarningMessage("line $p$q not drawn: points too close or ".
                              "radii too big", $lc);
          next LOOP;
       }
       ($px, $py) = pointOnLine($px, $py, $qx, $qy, $pSV) if $pSV > 0;
       ($qx, $qy) = pointOnLine($qx, $qy, $px, $py, $qSV) if $qSV > 0;
       if ($px == $qx || $py == $qy)
       {
          printf OUT "\\putrule from %.5f %.5f to %.5f %.5f %%%% %s%s\n",
                     $px,$py,$qx,$qy,$p,$q;
       }
       else
       {
          printf OUT "\\plot %.5f %.5f\t%.5f %.5f / %%%% %s%s\n",
                     $px, $py,$qx,$qy,$p,$q;
       }
    }
  }
```

The subroutine drawCurve is used to draw a curve that passes through an odd number of points. The subroutine has as argument an array which contains the names of the points which define the lines plus the current program line number. The subroutine emits code that has the following general form:

```
\setquadratic
\plot
  X1 Y1
  X2 Y2
  X3 Y3
\setlinear
```

```
<subroutine drawCurve >= (<-U)
  sub drawCurve {
    my ($NoArgs);
    $NoArgs = @_;
    my ($lc) = $_[$NoArgs-1]; #line number is the last argument
    my ($NumberOfPoints, $p);

    $NumberOfPoints = $NoArgs - 1;
```

```
    print OUT "\\setquadratic\n\\plot\n";
    for(my $i=0; $i <= $NumberOfPoints; $i++)
    {
       $p = $_[$i];
       my ($px,$py,$pSV,$pS) = unpack("d3A*",$PointTable{lc($p)});
       printf OUT "\t%0.5f  %0.5f", $px, $py;
       print OUT (($i == $NumberOfPoints) ? " / %$p\n" : " %$p\n");
    }
    print OUT "\\setlinear\n";
  }
```

The subroutine drawpoints is used to draw one or more points. The subroutine has as arguments a list of points. For each point we produce code that has the following general form:

\put {SYMBOL} at Px PY

where SYMBOL is either the default plot symbol, i.e., $\bullet$, whatever the user has set with the PointSymbol command, or the plot symbol specified in the definition of the point.

```
<subroutine drawpoints >= (<-U)
  sub drawpoints {
    my ($NumberOfPoints,$p);
    $NumberOfPoints = @_;
    my ($px,$py,$pSV,$pS);

    for($i=0; $i < $NumberOfPoints; $i++)
    {
       $p = $_[$i];
       ($px,$py,$pSV,$pS) = unpack("d3A*",$PointTable{lc($p)});
       if ($pS eq "" and $defaultsymbol =~ /circle|square/) {
         $pS = $defaultsymbol;
       }
       POINTSWITCH: {
         if ($pS eq "") # no plot symbol specified
         {
           printf OUT "\\put {%s} at %.5f %.5f %%%% %s\n",
                      $defaultsymbol, $px, $py, $p;
           last POINTSWITCH;
         }
         if ($pS eq "circle") # plot symbol is a circle
         {
           my $radius = (defined($DimOfPoint{lc($p)})) ? $DimOfPoint{lc($p)} :
                        $GlobalDimOfPoints;
           if ($radius > 0) # draw a circle using the current units
           {
              if ($radius == 1.5) # use \bigcirc
              {
                printf OUT "\\put {\$\\bigcirc\$} at %.5f %.5f  %%%% %s\n",
                           $px, $py, $p;
              }
              else
              {
                printf OUT "\\circulararc 360 degrees from %.5f %.5f center at %.5f
%.5f %%%% %s\n",
                           $px+$radius, $py, $px, $py, $p;
              }
           }
           else #use \circ symbol
           {
             printf OUT "\\put {\$\\circ\$} at %.5f %.5f %%%% %s\n",
                        $px,$py,$p;
           }
           last POINTSWITCH;
         }
         if ($pS eq "square")
         {
```

```
          my $side = (defined($DimOfPoint{lc($p)})) ? $DimOfPoint{lc($p)} :
                       $GlobalDimOfPoints;
          printf OUT "\\put {%s} at %.5f %.5f %%%% %s\n",
                       drawsquare($side), $px, $py, $p;
          last POINTSWITCH;
        }
        printf OUT "\\put {%s} at %.5f %.5f %%%% %s\n", $pS,$px,$py,$p;
      }
    }
  }
```

The subroutine drawAngleArc gets six arguments which correspond to three points defining an angle (variables $P1, $P2 and $P3), the radius, the internal/external specification and the direction specification (clockwise or anticlockwise). Depending on the values of these arguments, the subroutine returns the corresponding PiCTeX code, the general format of which is

```
    \circulararc Angle degrees from x y center at x2 y2
```

where Angle is the angle that the three points P1 P2 P3 define (computed by subroutine Angle), and x and y are the coordinates of a point residing on line P2P1 at distance equal to a $radius from point $P2; and x2, y2 are the coordinates of the center of the circle about which the arc is drawn, i.e., point $P2.

```
<subroutine drawAngleArc >= (<-U)
sub drawAngleArc {
  my ($P1, $P2, $P3, $radius, $inout, $direction) = @_;
  my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$P1});
  my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$P2});
  my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$P3});

  my $internalAngle = Angle($x1, $y1, $x2, $y2, $x3, $y3);
  my $externalAngle = 360 - $internalAngle;
  my ($x, $y) = pointOnLine($x2, $y2, $x1, $y1, $radius);
  my $code = "";
  if ($inout eq "internal" and $direction eq "clockwise" ) {
     $code = sprintf "\\circulararc %.5f degrees from %.5f %.5f center at %.5f
%.5f\n",
              -1 * $internalAngle, $x, $y, $x2, $y2;
  }
  elsif ($inout eq "internal" and $direction eq "anticlockwise" ) {
     $code = sprintf "\\circulararc %.5f degrees from %.5f %.5f center at %.5f
%.5f\n",
              $internalAngle, $x, $y, $x2, $y2;
  }
  elsif ($inout eq "external" and $direction eq "clockwise" ) {
     $code = sprintf "\\circulararc %.5f degrees from %.5f %.5f center at %.5f
%.5f\n",
              -1 * $externalAngle, $x, $y, $x2, $y2;
  }
  elsif ($inout eq "external" and $direction eq "anticlockwise" ) {
     $code = sprintf "\\circulararc %.5f degrees from %.5f %.5f center at %.5f
%.5f\n",
              $externalAngle, $x, $y, $x2, $y2;
  }
  return $code;
}
```

The subroutine drawAngleArrow gets six arguments which correspond to three points defining an angle (variables $P1, $P2 and $P3), the radius, the internal/external specification and the direction

specification. The subroutine mainly draws the arrowhead, and calls the subroutine drawAngleArc to draw the arc part of the arrow.

```
<subroutine drawAngleArrow >= (<-U)
sub drawAngleArrow {
   my ($P1, $P2, $P3, $radius, $inout, $direction) = @_;
   my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$P1});
   my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$P2});
   my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$P3});

   my $code = drawAngleArc($P1, $P2, $P3, $radius, $inout, $direction);

   my ($xqp, $yqp) = pointOnLine($x2, $y2, $x1, $y1, $radius);
   my ($deltax, $deltay) = ($x1 - $x2, $y1 - $y2);
   my $AL;

   if ($xunits =~ /mm/) {
     $AL = 1;
   }
   elsif ($xunits =~ /cm/) {
     $AL = 0.1;
   }
   elsif ($xunits =~ /pt/) {
     $AL = 2.845;
   }
   elsif ($xunits =~ /bp/) {
     $AL = 2.835;
   }
   elsif ($xunits =~ /pc/) {
     $AL = 0.2371;
   }
   elsif ($xunits =~ /in/) {
     $AL = 0.03937;
   }
   elsif ($xunits =~ /dd/) {
     $AL = 2.659;
   }
   elsif ($xunits =~ /cc/) {
     $AL = 0.2216;
   }
   elsif ($xunits =~ /sp/) {
     $AL = 186467.98;
   }
   my $halfAL = $AL / 2;
   my $d = sqrt($radius * $radius - $halfAL * $halfAL);
   my $alpha = atan2($d / $halfAL, 1) * R2D;
   my $beta = 2 * (90 - $alpha);
   my $thetaqr;
   if (abs($deltay) < 0.00001) {
     if ($deltax > 0 ) {$thetaqr = 0 }
     elsif ($deltax < 0) {$thetaqr = -180}
   }
   else {
     if (abs($deltax) < 0.00001) {
        $thetaqr = 90;
     }
     else {
       $thetaqr = atan2($deltay / $deltax, 1) * R2D;
     }
   }
   my ($xqr, $yqr) = pointOnLine($x2, $y2, $x3, $y3, $radius);
   $deltax = $x3 - $x2;
   $deltay = $y3 - $y2;
   $alpha = atan2(sqrt($radius * $radius - $halfAL * $halfAL) / $halfAL, 1) /
            D2R;
   $beta = 2 * (90 - $alpha);
   LINE2 : {
     if (abs($deltax) < 0.00001) {
```

```
      if ($deltay > 0) { $thetaqr = 90 }
      elsif ($deltay < 0) { $thetaqr = - 90 }
      last LINE2;
    }
    else {
      $thetaqr = atan2($deltay / $deltax, 1) * R2D;
    }
    if (abs($deltay) < 0.00001) {
      if ($deltax > 0)    { $thetaqr = 0 }
      elsif ($deltax < 0) { $thetaqr = -180 }
      last LINE2;
    }
    else {
      $thetaqr = atan2($deltay / $deltax, 1) * R2D;
    }
    if ($deltax < 0 and $deltay > 0) { $thetaqr += 180 }
    elsif ($deltax < 0 and $deltay < 0) { $thetaqr += 180 }
    elsif ($deltax > 0 and $deltay < 0) { $thetaqr += 360 }
  }
  my $xqrleft  = $x2 + $radius * cos(($thetaqr + $beta) * D2R);
  my $yqrleft  = $y2 + $radius * sin(($thetaqr + $beta) * D2R);
  my $xqrright = $x2 + $radius * cos(($thetaqr - $beta) * D2R);
  my $yqrright = $y2 + $radius * sin(($thetaqr - $beta) * D2R);
  if ($inout eq "internal" and $direction eq "clockwise") {
    $code .= sprintf "\\arrow <1.5mm> [0.5, 1] from %.5f %.5f to %.5f %.5f\n",
              $xqrleft, $yqrleft, $xqr, $yqr;
  }
  elsif ($inout eq "internal" and $direction eq "anticlockwise") {
    $code .= sprintf "\\arrow <1.5mm> [0.5, 1] from %.5f %.5f to %.5f %.5f\n",
              $xqrright, $yqrright, $xqr, $yqr;
  }
  elsif ($inout eq "external" and $direction eq "clockwise") {
    $code .= sprintf "\\arrow <1.5mm> [0.5, 1] from %.5f %.5f to %.5f %.5f\n",
              $xqrleft, $yqrleft, $xqr, $yqr;
  }
  elsif ($inout eq "external" and $direction eq "anticlockwise") {
    $code .= sprintf "\\arrow <1.5mm> [0.5, 1] from %.5f %.5f to %.5f %.5f\n",
              $xqrright, $yqrright, $xqr, $yqr;
  }
  return $code;
}
```

The subroutine expr is used to parse an expression. We are using a recursive descent parser to parse and evaluate an expression. The general syntax of an expression is as follows:

```
expr      ::= term { addop term }
addop     ::= "+" | "-"
term      ::= factor { mulop factor }
mulop     ::= "*" | "/" | "rem"
factor    ::= primitive [ ** factor ]
primitive ::= [ "+" | "-"] primitive   | number               | variable
|
              pair-of-points           | "(" expr ")"         |
              "sin (" expr ")"         | "cos (" expr ")"     | "area (" ThreePoints
")" |
              "tan (" expr ")"         | "exp (" expr ")"     | "int" "(" expr ")"
|
              "log (" expr ")"         | "atan (" expr ")"    | "sgn" "(" expr ")"
|
              "sqrt (" expr ")"        | "acos (" expr ")"    | "asin (" expr ")"
|
              "atan (" expr ")"        | "_pi_"               | "_e_"
|
              "xcoord (" point ")"     | "ycoord (" point ")" | "angle "("
ThreePoints ")"|
              "angledeg" "(" ThreePoints ")" | "direction" "(" TwoPoints ")" |
```

```
"directiondeg" "(" TwoPoints ")"  | "_linethickness_"
```

Note that _pi_ and _e_ can be used to access the value of the constants Pi and e.

```
<subroutine expr >= (<-U)
sub expr {
  my $lc = $_[0];
  my($left,$op,$right);

  $left = term($lc);
  while ($op = addop()) {
    $right = term($lc);
    if ($op eq '+')
      { $left += $right }
    else
      { $left -= $right }
  }
  return $left;
}

sub addop {
  s/^([+-])// && $1;
}

sub term {
  my $lc = $_[0];
  my ($left, $op, $right);
  $left = factor($lc);
  while ($op = mulop()) {
    $right = factor($lc);
    if ($op eq '*')
      { $left *= $right }
    elsif ($op =~ /rem/i) {
      eval {$left %= $right};
      PrintFatalError("Division by zero", $lc) if $@;
    }
    else {
      eval {$left /= $right};
      PrintFatalError("Division by zero", $lc) if $@;
    }
  }
  return $left;
}

sub mulop {
  (s#^([*/])## || s/^(rem)//i) && lc($1);
}

sub factor {
  my $lc = $_[0];
  my ($left);

  $left = primitive($lc);
  if (s/^\*\*//) {
    $left **= factor($lc);
  }
  return $left;
}

sub primitive {
  my $lc = $_[0];
  my $val;
  s/\s*//;
  if (s/^\(//) {  #is it an expr in parentheses
    $val = expr($lc);
    s/^\)// || PrintErrorMessage("Missing right parenthesis", $lc);
  }
```

```
elsif (s/^-//) { # is it a negated primitive
   $val = - primitive();
}
elsif (s/^\+//) { # is it a positive primitive
   $val = primitive();
}
elsif (s/^angledeg//i) {
   chk_lparen("angledeg",$lc);
   my $point_1 = get_point($lc);
   my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_1});
   my $point_2 = get_point($lc);
   my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point_2});
   my $point_3 = get_point($lc);
   my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point_3});
   my $d12 = Length($x1, $y1, $x2, $y2);
   my $d23 = Length($x2, $y2, $x3, $y3);
   my $d31 = Length($x3, $y3, $x1, $y1);
   if ( $d12 == 0 ) {
     PrintErrorMessage("points `$point_1' and `$point_2' are the same", $lc);
     $val = 0;
   }
   elsif ( $d23 == 0 ) {
     PrintErrorMessage("points `$point_2' and `$point_3' are the same", $lc);
     $val = 0;
   }
   elsif ( $d31 == 0 ) {
     PrintErrorMessage("points `$point_1' and `$point_3' are the same", $lc);
     $val = 0;
   }
   else {
     $val = Angle($x1, $y1, $x2, $y2, $x3, $y3);
     $val = 0 if $val == -500;
   }
   chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^angle//i) {
   chk_lparen("angle".$lc);
   my $point_1 = get_point($lc);
   my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_1});
   my $point_2 = get_point($lc);
   my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point_2});
   my $point_3 = get_point($lc);
   my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point_3});
   my $d12 = Length($x1, $y1, $x2, $y2);
   my $d23 = Length($x2, $y2, $x3, $y3);
   my $d31 = Length($x3, $y3, $x1, $y1);
   if ( $d12 == 0 ) {
     PrintErrorMessage("points `$point_1' and `$point_2' are the same", $lc);
     $val = 0;
   }
   elsif ( $d23 == 0 ) {
     PrintErrorMessage("points `$point_2' and `$point_3' are the same", $lc);
     $val = 0;
   }
   elsif ( $d31 == 0 ) {
     PrintErrorMessage("points `$point_1' and `$point_3' are the same", $lc);
     $val = 0;
   }
   else {
     $val =  Angle($x1, $y1, $x2, $y2, $x3, $y3);
     if ($val == -500) {
       $val = 0;
     }
     else {
       $val = D2R * $val;
     }
   }
   chk_rparen("Missing right parenthesis", $lc);
}
```

```
elsif (s/^area//i) {
   chk_lparen("angledeg",$lc);
   my $point_1 = get_point($lc);
   my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_1});
   my $point_2 = get_point($lc);
   my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point_2});
   my $point_3 = get_point($lc);
   my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point_3});
   $val = triangleArea($x1, $y1, $x2, $y2, $x3, $y3);
   chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^asin//i) {
   chk_lparen("asin");
   $val = expr();
   PrintFatalError("Can't take asin of $val", $lc) if $val < -1 || $val > 1;
   $val = asin($val);
   chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^acos//i) {
   chk_lparen("acos");
   $val = expr();
   PrintFatalError("Can't take acos of $val", $lc) if $val < -1 || $val > 1 ;
   $val = acos($val);
   chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^atan//i) {
   chk_lparen("atan");
   $val = expr();
   $val = atan($val);
   chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^cos//i) {
   chk_lparen("cos");
   $val = expr();
   $val = cos($val);
   chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^directiondeg//i) {
   chk_lparen("directiondeg",$lc);
   my $point_1 = get_point($lc);
   my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_1});
   my $point_2 = get_point($lc);
   my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point_2});
   my $x3 = $x1+1;
   if ( ($y2 - $y1) >= 0) {
     $val = Angle($x3, $y1, $x1, $y1, $x2, $y2);
     $val = 0 if $val == -500;
   }
   else {
     $val = 360 - Angle($x3, $y1, $x1, $y1, $x2, $y2);
     $val = 0 if $val == -500;
   }
   chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^direction//i) {
   chk_lparen("direction",$lc);
   my $point_1 = get_point($lc);
   my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_1});
   my $point_2 = get_point($lc);
   my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point_2});
   my $x3 = $x1+1;
   if ( ($y2 - $y1) >= 0) {
     $val = Angle($x3, $y1, $x1, $y1, $x2, $y2);
     $val = 0 if $val == -500;
     $val = D2R * $val;
   }
   else {
     $val = 360 - Angle($x3, $y1, $x1, $y1, $x2, $y2);
     $val = 0 if $val == -500;
```

```
      $val = D2R * $val;
    }
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^exp//i) {
    chk_lparen("exp");
    $val = expr();
    $val = exp($val);
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^int//i) {
    chk_lparen("int");
    $val = expr();
    $val = int($val);
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^log//i) {
    chk_lparen("log");
    $val = expr();
    PrintFatalError("Can't take log of $val", $lc) if $val <= 0;
    $val = log($val);
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^sin//i) {
    chk_lparen("sin");
    $val = expr();
    $val = sin($val);
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^sgn//i) {
    chk_lparen("sgn");
    $val = expr();
    if ($val > 0) {
      $val = 1;
    }
    elsif ($val == 0) {
      $val = 0;
    }
    else {
      $val = -1;
    }
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^sqrt//i) {
    chk_lparen("sqrt");
    $val = expr();
    $val = sqrt($val);
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^tan//i) {
    chk_lparen("tan");
    $val = expr();
    $val = sin($val)/cos($val);
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^xcoord//i) {
    chk_lparen("xcoord");
    my $point_name = get_point;
    my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_name});
    $val = $x1;
    chk_rparen("Missing right parenthesis", $lc);
}
elsif (s/^ycoord//i) {
    chk_lparen("ycoord");
    my $point_name = get_point;
    my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_name});
    $val = $y1;
    chk_rparen("Missing right parenthesis", $lc);
}
```

```
  elsif (s/^_pi_//i) {
     $val = PI;
  }
  elsif (s/^_e_//i) {
     $val = 2.71828182845905;
  }
  elsif (s/^_linethickness_//i) {
     $val = $LineThickness / $xunits;
  }
  else  {
    my $err_code;
    ($val,$err_code) = ComputeDist($lc);
  }
  s/\s*//;
  return $val;
}
```

The subroutine memberOf is used to check whether a string is part of a list of strings. We assume that the first argument is the string in question. We compare each list element against the string in question and if we find it we stop and return the value 1 (denoting truth). Otherwise, we simply return the value 0 (denoting false).

```
<subroutine memberOf >= (<-U)
sub memberOf {
   my $elem = shift(@_);

   my $found = 0;
   foreach $item (@_){
       if ($item eq $elem){
          $found = 1;
          last;
       }
   }
   return $found;
}
```

The subroutine tand computes the tangent of an angle. The angle is supposed to be in degrees. We simply transform it into radians and then compute the actual result.

```
<subroutine tand >= (<-U)
sub tand {
  my $d = $_[0];
  $d = $d * PI / 180;
  return sin($d)/cos($d);
}
```

The subroutine get_string is used to extract a leading valid mathspic string from the input line. A string must start with a quotation mark, i.e., ", and must end with the same symbol. A string may contain quotation marks which must be escaped with a backslash, i.e., \. Initially, we remove all leading white space. If the next character of the string is not a quotation mark we print an error message and stop. Otherwise, we split the string into an array of characters and store the characters up to the next quotation mark to the array @cmd. In case the next character is a backslash and we aren't at the end of the input string and the next character is a quotation mark, we have an escape sequence. This means that we store these two characters in the @cmd array and skip to characters after the quotation mark. Otherwise,

we simply store the character in the @cmd array and skip to the next character. This process is repeated until either we consume all the characters of the string or until we find a sole quotation mark. Since we are not sure what has forced the loop to exit, we check whether there are still characters in the input string and we check whether this is a quotation mark. If these tests fail we have a string without a closing quotation mark. In all cases we return a triplet consisting of a number denoting success (1) or failure (0) and what we have consumed from the input string, and what is left from the input string.

```
<subroutine get_string >= (<-U)
sub get_string {
  my $string = shift;
  my $lc = shift;

  $string =~ s/^\s+//;
  if ($string !~ s/^\"//) {
    PrintErrorMessage("No starting \" found",$lc);
    return (1,$string,$string);
  }
  my @ch = split //,$string;
  my @cmd;
  while (@ch and $ch[0] ne "\"") {
    if ($ch[0] eq "\\" and (defined $ch[1]) and $ch[1] eq "\"") {
       shift @ch;
       push @cmd, $ch[0];
       shift @ch;
    }
    else {
      push @cmd, $ch[0];
      shift @ch;
    }
  }
  if (! defined $ch[0]) {
     PrintErrorMessage("No closing \" found",$lc);
     return (1,join("",@cmd), join("",@ch))
  }
  else {
    shift @ch;
    return (0, join("",@cmd), join("",@ch))
  }
}
```

The definition as well as an explanation of the functionality of the following subroutine can be found in "Programming Perl", 3rd edition.

```
<subroutine is_tainted >= (<-U)
sub is_tainted {
  my $arg = shift;
  my $nada = substr($arg,0,0);
  local $@;
  eval { eval "# $nada"};
  return length($@) != 0;
}
```

The subroutine noOfDigits has one argument which is a number and returns the number of decimal digits it has. If the number matches the regular expression ^\d+(?!\.) (a series of digits *not* followed by a period), then the number of decimal digits is zero. If the number matches the regular expression ^\d+\.(\d+)?, then number of decimal digits equals length($1). Naturally, it maybe zero!

```
<subroutine noOfDigits >= (<-U)
sub noOfDigits {
  my $num = $_[0];

  if ($num =~ /^[\+-]?\d+(?!\.)/) {
    return 0;
  }
  elsif ($num =~ /^[\+-]\d+\.(\d+)?/) {
    return length($1);
  }
}
```

Subroutine drawsquare is use by the drawpoints routine to plot a point whose point symbol is a square. The subroutine has one argument, which is equal to the radius of the point. From this argument it computes the side of the square.

```
<subroutine drawsquare >= (<-U)
sub drawsquare {
   my $s = $_[0];
   #$s *= sqrt(2);
   $s = sprintf "%.5f", $s;
   my $code = "\\setlength{\\unitlength}{$xunits}%\n";
   $code .= "\\begin{picture}($s,$s)\\put(0,0)" .
            "{\\framebox($s,$s){}}\\end{picture}";
   return $code;
}
```

Subroutine X2sp has two arguments: a number and a length unit. It returns the length expresssed in sp units.

```
<subroutine X2sp >= (<-U)
sub X2sp {
   my $LT = shift;
   my $units = shift;

   if ($units eq "pc") {
     return $LT * 786432;
   }
   elsif ($units eq "pt") {
     return $LT * 65536;
   }
   elsif ($units eq "in") {
     return $LT * 4736286.72;
   }
   elsif ($units eq "bp") {
     return $LT * 65781.76;
   }
   elsif ($units eq "cm") {
     return $LT * 1864679.811023622;
   }
   elsif ($units eq "mm") {
     return $LT * 186467.981102362;
   }
   elsif ($units eq "dd") {
     return $LT * 70124.086430424;
   }
   elsif ($units eq "cc") {
     return $LT * 841489.037165082;
   }
   elsif ($units eq "sp") {
     return $LT;
   }
}
```

Subroutine sp2X has two arguments: a number that denotes a length in sp units and a length unit. It returns the length expresssed in units that are specified by the second argument.

```
<subroutine sp2X >= (<-U)
sub sp2X {
   my $LT = shift;
   my $units = shift;

   if ($units eq "pc") {
     return $LT / 786432;
   }
   elsif ($units eq "pt") {
     return $LT / 65536;
   }
   elsif ($units eq "in") {
     return $LT / 4736286.72;
   }
   elsif ($units eq "bp") {
     return $LT / 65781.76;
   }
   elsif ($units eq "cm") {
     return $LT / 1864679.811023622;
   }
   elsif ($units eq "mm") {
     return $LT / 186467.981102362;
   }
   elsif ($units eq "dd") {
     return $LT / 70124.086430424;
   }
   elsif ($units eq "cc") {
     return $LT / 841489.037165082;
   }
   elsif ($units eq "sp") {
     return $LT;
   }
}
```

Subroutine setLineThickness takes two arguments: the value of the variable $xunits and a string denoting the linethickness. It returns the linthickness expressed in the units of the $xunits.

```
<subroutine setLineThickness >= (<-U)
sub setLineThickness {
  my $Xunits = shift;
  my $LT = shift;
  $Xunits =~ s/^((\+|-)?\d+(\.\d+)?([eE](\+|-)?\d+)?)//;
  my $xlength = "$1";
  $Xunits =~ s/^\s*($units)//;
  my $x_in_units = $1;
  $LT =~ s/^((\+|-)?\d+(\.\d+)?([eE](\+|-)?\d+)?)//;
  my $LTlength = "$1";
  $LT =~ s/^\s*($units)//;
  my $LT_in_units = $1;
  $LTlength = X2sp($LTlength,$LT_in_units);
  $LTlength = sp2X($LTlength,$x_in_units);
  return $LTlength;
}
```

The subroutine process_input accepts one argument which is a file handle that corresponds to the file that the subroutine is supposed to process. The processing cycle is fairly simple: we input one line at

the time, remove any leading space characters and the trailing new line character, and then start the actual processing. The variable $INFILE contains the name of the input file and the variable $lc is the local line counter. The commands beginSkip and endSkip can be used to ignore blocks of code and so we need to process them here. The variable $no_output is used as a switch to toggle from process mode to no-precess mode. If the first token is beginSkip, we set the variable $no_output to 1, print a comment to the output file and continue with the next input line. If the first token is endSkip, we check whether we are in a no-process mode. If this is the case, we revert to process mode; otherwise we print an error message. Finally, depending on whether we are in process or no-process mode we process the input text or simply printed commented out to the output file. Note, that we don't allow nested comment blocks, as this makes really no sense!

```
<subroutine process_input >= (<-U)
  sub process_input {
    my ($INFILE,$currInFile) = @_;
    my $lc = 0;
    my $no_output = 0;
    $curr_in_file = $currInFile;
    LINE: while(<$INFILE>) {
      $lc++;
      chomp($command = $_);
      s/^\s+//;
      if (/^beginSkip\s*/i) {
        $no_output = 1;
        print OUT "%%$_" if $comments_on;
        next LINE;
      }
      elsif (/^endSkip\s*/i) {
        if ($no_output == 0) {
          PrintErrorMessage("endSkip without beginSkip",$lc);
        }
        else {
          $no_output = 0;
        }
        print OUT "%%$_" if $comments_on and !$no_output;
        next LINE;
     }
     elsif ($no_output == 1) {
       next LINE;
     }
     else {
        if (/^[^\\]/) {
          my $out_line  = mpp($command,$lc) unless /^\\/;  #call macro pre-processor
          $_ = "$out_line\n";
        }
       <process input line>
     }
   }
 }
```

Each command line starts with a particular *token* and depending on which one we have we perform different actions. If the first character is % we have a comment line, and depending on the value of the variable $comments_on we either output the comment on the output file (default action) or just ignore it and continue with the next input line. In case the first token is the name of a valid command we process the command and output the corresponding code. Otherwise, we print an error message to the screen and to the log file and continue with the next input line. Note that the input language is case-

insensitive and so one is free to write a command name using any combination of upper and lower case letters, e.g., the tokens lAtEx, LaTeX, and latex are considered exactly the same. The valid *MathsPIC* commands are the following (don't pay attention to the case!):

- Commands drawAngleArc and drawAngleArrow are used to draw an arc and an arrow, respectively. Since, their user interface is identical, we process them as if they were identical commands.
- Command drawcircle is used to draw a circle with a specified radius.
- Command drawCircumCircle is used to draw the circumcircle of triangle specified by three points.
- Command drawexcircle is used to draw the excircle of triangle relative to a given side of the triangle.
- Command drawincircle is used to draw the incircle of triangle.
- Command drawincurve is used to draw a curve that passes through a number of points.
- Command drawline is used to draw either a line (not necessarily a straight one) or a number of lines from a list or lists of points. The lines are specified as pairs of points that can be separated by blank spaces.
- Command drawthickline is used to draw either a thick line (not necessarily a straight one) or a number of lines from a list or lists of points. The lines are specified as pairs of points that can be separated by blank spaces.
- Command drawPerpendicular draws a perpendicular line from point A to line BC.
- Command drawpoint is used to draw one, two or more points. The point names can be separated by blanks.
- Command drawRightAngle draws an angle, specified by three points, of a size specified by a side length.
- Command drawsquare draws a square, centered at the coordinates of the first arguments, which is assumed to be a point, with side equal to the second argument.
- Command inputfile* is used to verbatim include a file into the output file.
- Command inputfile is used to include a *MathsPIC* program file into the main file.
- Command linethickness should be used to set the thickness of lines.
- The paper command sets the paper scale, size, axes, etc. The most general format of the command follows:

  ```
  paper{units(mm), xrange(0,120), yrange(0,100),axes(LRTB)}
  ```

  Note, that one may opt not to write the commas between the different parts of command.
- Command point* allocates *new* co-ordinates and optionally a T_EX point-name, to an existing point-name. Command point allocates co-ordinates and, optionally a T_EX point character, to a *new* point-name. Since, both commands have identical syntax, we handle them together.
- Command PointSymbol is used to set or reset the default point symbol, i.e., when one plots a point this is the symbol that will appear on the final DVI/PostScript file.
- In the original DOS version of mathspic the command setPointNumber was used to set the length of the arrays that keep the various point related information. Since, in Perl arrays are dynamic objects and one can push as many objects as he/she wants,

the command is implemented as an no-op. For reasons of compatibility, we only check the syntax of the command.
- Commands showAngle and showArea can be used to get the angle or the area determined by three points. In addition, the command showLenght can be used to get the length between two points. These three commands produce a comment to the output file.
- The system command provides a shell escape.
- The text command is used to put a symbol/text at a particular point location.
- Command var is used to store a numeric value into a comma separated list of variables.
- Command const is used to store a numeric value into a comma separated list of variables, whose value cannot be altered.
- If a line starts with a backslash, \, then we copy verbatim this line to the output file. In case the second character is a space character, then we simply output a copy of the line without the leading backslash.

Empty lines are always ignored.

<process input line>= (<-U)

```
if (/^\s*%/)
{
   print OUT "$_" if $comments_on;
}
elsif (s/^\s*(beginloop(?=\W))//i) {
  s/\s+//;
  my $times = expr($lc);
  print OUT "%% BEGINLOOP $times\n" if $comments_on;
  my @C = ();
  REPEATCOMMS: while (<$INFILE>) {
    if (/^\s*endloop/i) {
      last REPEATCOMMS;
    }
    else {
      push @C, $_;
    }
  }
  if (! /^\s*endloop/i) {
    PrintFatalError("unexpected end of file",$lc);
  }
  else {
    s/^\s*endloop//i;
    for(my $i=1; $i<=$times; $i++) {
      tie *DUMMY, 'DummyFH', \@C;
      process_input(DUMMY, $currInFile);
      untie *DUMMY;
    }
    print OUT "%% ENDLOOP\n" if $comments_on;
  }
}
elsif (s/^\s*(ArrowShape(?=\W))//i)
{
    my $cmd = $1;
    print OUT "%% $cmd$_" if $comments_on;
   <process ArrowShape command>
}
elsif (s/^\s*(const(?=\W))//i)
{
   print OUT "%% $1$_" if $comments_on;
   <process const command>
}
elsif (s/^\s*(dasharray(?=\W))//i)
{
```

```
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process dasharray command>
}
elsif (s/^\s*(drawAngleArc(?=\W))//i or s/^\s*(drawAngleArrow(?=\W))//i )
{
   my $cmd = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawAngleArcOrArrow command>
}
elsif (s/^\s*(drawArrow(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   DrawLineOrArrow(0,$lc);
}
elsif (s/^\s*(drawcircle(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawcircle command>
}
elsif (s/^\s*(drawcurve(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   DrawLineOrArrow(2,$lc);
}
elsif (s/^\s*(drawcircumcircle(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawcircumcircle command>
}
elsif (s/^\s*(drawexcircle(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawexcircle command>
}
elsif (s/^\s*(drawincircle(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawincircle command>
}
elsif (s/^\s*(drawline(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   DrawLineOrArrow(1,$lc);
}
elsif (s/^\s*(drawthickarrow(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   print OUT "\\setplotsymbol  ({\\usefont{OT1}{cmr}{m}{n}\\large .})%\n";
   print OUT "{\\setbox1=\\hbox{\\usefont{OT1}{cmr}{m}{n}\\large .}%\n";
   print OUT " \\global\\linethickness=0.31\\wd1}%\n";
   DrawLineOrArrow(0,$lc);
   print OUT "\\setlength{\\linethickness}{0.4pt}%\n";
   print OUT "\\setplotsymbol  ({\\usefont{OT1}{cmr}{m}{n}\\tiny .})%\n";
}
elsif (s/^\s*(drawthickline(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   print OUT "\\setplotsymbol  ({\\usefont{OT1}{cmr}{m}{n}\\large .})%\n";
   print OUT "{\\setbox1=\\hbox{\\usefont{OT1}{cmr}{m}{n}\\large .}%\n";
```

```
   print OUT " \\global\\linethickness=0.31\\wd1}%\n";
   DrawLineOrArrow(1,$lc);
   print OUT "\\setlength{\\linethickness}{0.4pt}%\n";
   print OUT "\\setplotsymbol  ({\\usefont{OT1}{cmr}{m}{n}\\tiny .})%\n";
}
elsif (s/^\s*(drawperpendicular(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawPerpendicular command>
}
elsif (s/^\s*(drawpoint(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawpoint command>
}
elsif (s/^\s*(drawRightAngle(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawRightAngle command>
}
elsif (s/^\s*(drawsquare(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process drawsquare command>
}
elsif (s/^\s*inputfile\*//i)
{
   <process inputfile* command>
}
elsif (s/^\s*(inputfile(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process inputfile command>
}
elsif (s/^\s*(linethickness(?=\W))//i)
{
   my $cmd = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process linethickness command>
}
elsif (s/^\s*(paper(?=\W))//i)
{
   my ($cmd) = $1;
   print OUT "%% $cmd$_" if $comments_on;
   <process paper command>
}
elsif (s/^\s*(PointSymbol(?=\W))//i)
{
    my $cmd = $1;
    print OUT "%% $cmd$_" if $comments_on;
   <process PointSymbol command>
}
elsif (s/^\s*point(?=\W)//i)
{
   my ($Point_Line);
   chomp($Point_Line=$_);
   <process point/point* commands>
}
elsif (/^\s*setPointNumber(?=\W)/i)
{
   PrintWarningMessage("Command setPointNumber is ignored",$lc);
   next LINE;
}
elsif (s/^\s*(showAngle(?=\W))//i)
```

```
{
   <process showAngle command>
}
elsif (s/^\s*(showArea(?=\W))//i)
{
   <process showArea command>
}
elsif (s/^\s*(showLength(?=\W))//i)
{
   <process showLength command>
}
elsif (/^\s*showPoints(?=\W)/i)
{
   print OUT "%%-------------------------------------------------\n";
   print OUT "%%           L I S T  O F  P O I N T S            \n";
   print OUT "%%-------------------------------------------------\n";
   foreach my $p (keys(%PointTable)) {
     my ($x, $y, $pSV, $pS) = unpack("d3A*",$PointTable{$p});
     printf OUT "%%%%\t%s\t= ( %.5f, %.5f ), LF-radius = %.5f, symbol = %s\n",
            $p, $x, $y, $pSV, $pS;
   }
   print OUT "%%-------------------------------------------------\n";
   print OUT "%%      E N D  O F  L I S T  O F  P O I N T S      \n";
   print OUT "%%-------------------------------------------------\n";
   next LINE;
}
elsif (/^\s*showVariables(?=\W)/i)
{
   print OUT "%%-------------------------------------------------\n";
   print OUT "%%       L I S T  O F  V A R I A B L E S          \n";
   print OUT "%%-------------------------------------------------\n";
   foreach my $var (keys(%VarTable)) {
     print OUT "%%\t", $var, "\t=\t", $VarTable{$var}, "\n";
   }
   print OUT "%%-------------------------------------------------\n";
   print OUT "%%   E N D  O F  L I S T  O F  V A R I A B L E S   \n";
   print OUT "%%-------------------------------------------------\n";
   next LINE;
}
elsif (s/^\s*(system(?=\W))//i)
{
   print OUT "%% $1$_" if $comments_on;
   <process system command>
}
elsif (s/^\s*(text(?=\W))//i)
{
   print OUT "%% $1$_" if $comments_on;
   <process text command>
}
elsif (s/^\s*(var(?=\W))//i)
{
   print OUT "%% $1$_" if $comments_on;
   <process var command>
}
elsif (/^\s*\\(.+)/)
{
   my $line = $1;
   if ($line =~ /^\s+(.+)/)
   {
      print OUT " $line\n";
   }
   else
   {
     print OUT "\\$line\n";
   }
   next LINE;
}
elsif (0==length) #empty line
{
```

```
    next LINE;
  }
  else {
    PrintErrorMessage("command not recognized",$lc);
    next LINE;
  }
```

Command dasharray takes an arbitrary number of arguments that are used to specify a dash pattern. Its general syntax follows:

$$\text{"dasharray" "(" } d_1 \text{ "," } g_1 \text{ "," } d_2 \text{ "," } g_2 \text{ "," ... ")"}$$

where d_i denotes the length of a dash and g_i denotes the length of gap between two consecutive dashes. Each d_i and g_i is a length (i.e., a number accompanied by a length of unit). Since we do not a priori know the number of arguments, we push them onto a stack and then we produce a command of the form

$$\backslash\text{setdashpattern} < d_1, g_1, d_2, g_2, \ldots >$$

<process dasharray command>= (<-U)

```
chk_lparen($cmd,$lc);
my @DashArray = ();
my $dash = "";
my $dashpattern = "";
PATTERN: while (1) {
  $dash = sprintf("%.5f", expr($lc));
  if (s/^\s*($units)//i) {
     push (@DashArray, "$dash$1");
  }
  else {
    PrintErrorMessage("Did not found unit after expression", $lc);
  }
  s/\s*//;
  if (/^[^,]/) {
    last PATTERN;
  }
  else {
    s/^,\s*//;
  }
}
print OUT "\\setdashpattern <";
while (@DashArray) {
  $dashpattern .= shift @DashArray;
  $dashpattern .= ",";
}
$dashpattern =~ s/,$//;
print OUT $dashpattern, ">\n";
chk_rparen("arguments of $cmd",$lc);
chk_comment($lc);
```

The command drawAngleArc draws an arc in the specified angle, a distance *radius* from the angle. The angle is either *internal* (<= 180 degrees) or *external* (>180 degrees). The direction of the arc is either *clockwise* or *anticlockwise*. The command drawAngleArrow draws an arrow just like the command drawAngleArc draws an arc. The syntax of these commands is as follows:

```
cmds      ::= ( "drawAngleArc" | "drawAngleArrow" ) args
args      ::= "{" angle comma radius comma internal comma clockwise "}"
angle     ::= "angle" "(" three-points ")"
radius    ::= "radius" "(" distance ")"
distance  ::= expression
internal  ::= "internal" | "external"
```

```
clockwise ::= "clockwise" | "anticlockwise"
comma     ::= "," | empty
```

We first collect all relevant information by parsing the args and then call the either the subroutine drawAngleArc or the subroutine drawAngleArrow to produce the actual code which is then printed into the output file. In order to be able to distinguish which command we are dealing with we simply use the variable $cmd. We now start parsing the input line. We first check whether there is a left curly bracket. Next, we parse the angle, the distance, the internal and the clockwise parts of the command. Finally, we check for right curly bracket and a trailing comment. Depending on the value of the variable $cmd we call either the subroutine drawAngleArc or the subroutine drawAngleArrow. These subroutines return the code that will be finally output to the output file.

<process drawAngleArcOrArrow command>= (<-U)

```
chk_lcb($cmd,$lc);
<process angle part of command>
s/^,\s*// or s/\s*//; #parse optional comma
<process radius part of command>
s/^,\s*// or s/\s*//; #parse optional comma
my $inout = "";
if (s/^(internal(?=\W))//i or s/^(external(?=\W))//i) {
  $inout = $1;
}
else {
  PrintErrorMessage("Did not find expected 'internal' specifier", $lc);
  next LINE;
}
s/^,\s*// or s/\s*//; #parse optional comma
my $direction = "";
if (s/^(clockwise(?=\W))//i or s/^(anticlockwise(?=\W))//i) {
  $direction = $1;
}
else {
  PrintErrorMessage("Did not find expected 'direction' specifier", $lc);
  next LINE;
}
chk_rcb("arguments of $cmd",$lc);
chk_comment($lc);
my $code;
if (lc($cmd) eq "drawanglearc") {
  $code = drawAngleArc($P1, $P2, $P3, $radius, $inout, $direction);
}
else {
  $code = drawAngleArrow($P1, $P2, $P3, $radius, $inout, $direction);
}
print OUT $code if $code ne "";
```

We first check whether the first token is the word angle. In case it isn't, this yields an unrecoverable error. In case the expected word is there, we check for a left parenthesis. Next, we parse the three points that must follow. For this purpose we use the user-defined subroutine get_point. Now we check that the angle has a reasonable value, i.e., if it is less than -400 or equal to zero, the value yields an unrecoverable error. We finish by checking whether there is a right parenthesis.

<process angle part of command>= (<-U)

```
my ($P1, $P2, $P3);
if (s/^angle(?=\W)//i) {
```

```
      chk_lparen("token angle of command $cmd",$lc);
      $P1 = get_point($lc);
      next LINE if $P1 eq "_undef_";
      $P2 = get_point($lc);
      next LINE if $P2 eq "_undef_";
      $P3 = get_point($lc);
      next LINE if $P3 eq "_undef_";
      my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$P1});
      my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$P2});
      my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$P3});
      my $Angle = Angle($x1, $y1, $x2, $y2, $x3, $y3);
      if ($Angle <= 0) {
        if ($Angle == 0) {
          PrintErrorMessage("Angle is equal to zero",$lc);
          next LINE;
        }
        elsif ($Angle < -400) {
          PrintErrorMessage("Something is wrong with the points",$lc);
          next LINE;
        }
      }
      chk_rparen("angle part of command $cmd",$lc);
    }
    else {
      PrintErrorMessage("Did not find expected angle part",$lc);
      next LINE;
    }
```

In this section we parse the radius part of the drawAngleArc or the drawAngleArrow command. We first check whether the next token is the word radius. If it is not, then we continue with the next line.

<process radius part of command>= **(<-U)**

```
    my $radius;
    if (s/^radius(?=\W)//i) {
      chk_lparen("token radius of command $cmd",$lc);
      $radius = expr($lc);
      chk_rparen("radius part of command $cmd",$lc);
    }
    else {
      PrintErrorMessage("Did not found expected angle part",$lc);
      next LINE;
    }
```

Command drawcircle accepts two arguments--a point name that is used to specify the center of the circle and the radius of the circle. The radius is simply an expression, whose value must be greater than zero. Otherwise, we print an error message and continue with the next input line. The general syntax of the command is as follows:

```
"drawcircle" "(" point-name "," rad ")"
```

The code we emit for a point with coordinates x and y and for radius equal to R is:

```
\circulararc 360 degrees from X y center at x y
```

where X = x+R.

Initially, we check whether there is an opening left parenthesis. Next, we get the point name by using the subroutine get_point which

issues an error message if the point hasn't been defined. In this case we stop processing the command, as there is absolutely no reason to do otherwise. Next, we parse the comma and then the radius by using the subroutine ComputeDist. If there is no problem, we emit the code and finally we check for a closing right parenthesis and for possible garbage that may follow the command.

```
<process drawcircle command>= (<-U)
      chk_lparen("drawcircle",$lc);
      my $Point = get_point($lc);
      next LINE if $Point eq "_undef_";
      chk_comma($lc);
      my $R = expr($lc);
      if ($R <= 0) {
         PrintErrorMessage("Radius must be greater than zero",$lc);
         next LINE;
      }
      my ($x,$y,$pSV,$pS)=unpack("d3A*",$PointTable{lc($Point)});
      printf OUT "\\circulararc 360 degrees from %.5f %.5f center at %.5f %.5f\n",
                  $x+$R, $y, $x, $y;
      chk_rparen("arguments of $cmd",$lc);
      chk_comment($lc);
```

Command drawcircumcircle is used to draw the circumcircle of triangle specified by three points which are the arguments of the command. We start by parsing the opening left parenthesis. Next, we get the three points that define the triangle. We are now able to compute the center and the radius of the circumcircle by calling the subroutine circumCircleCenter. If the triangle area is equal to zero, then this subroutine will return the array (0,0,0) to indicate this fact. We now have all necessary information to draw the circumcircle. We use the following code to do the job:

```
   \circulararc 360 degrees from X y center x y
```

where x and y are the coordinates of the center, R its radius and X=x+R. What is left is to check whether there is a closing right parenthesis and any trailing garbage.

```
<process drawcircumcircle command>= (<-U)
      chk_lparen("drawcircumcircle",$lc);
      my $point1 = get_point($lc);
      next LINE if $point1 eq "_undef_";
      my $point2 = get_point($lc);
      next LINE if $point2 eq "_undef_";
      my $point3 = get_point($lc);
      next LINE if $point3 eq "_undef_";
      my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point1});
      my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point2});
      my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point3});
      my ($xc, $yc,$r) = circumCircleCenter($x1,$y1,$x2,$y2,$x3,$y3,$lc);
      next LINE if $xc == 0 and  $yc == 0 and $r == 0;
      print OUT "%% circumcircle center = ($xc,$yc), radius = $r\n" if $comments_on;
      printf OUT "\\circulararc 360 degrees from %.5f %.5f center at %.5f %.5f\n",
                  $xc+$r, $yc, $xc, $yc;
      chk_rparen("arguments of $cmd",$lc);
      chk_comment($lc);
```

The syntax of the drawexcircle command is as follows:

```
      drawexcircle ::= "drawexcircle" "(" ThreePoints "," TwoPoints ")"
```

```
                    [ modifier ]
    modifier ::= "[" expr "]"
```

The modifier is an expression that is used to modify the radius of the excicle. We start by checking whether there is a left parenthesis. Then we get names of the three points. In case any of the points is not defined we issue an error message and continue with the next input line. Next, we check whether there is a comma that separates the three points defining the triangle from the two points defining a side of the triangle (variables $point1, $point2, and $point3). Moreover, we must ensure that the area of the area defined by these points is not equal to zero. If it is we issue an error message and we continue with the next input line. Now, we are ready to get the two point names that define the side of the triangle (variables $point3 and $point5). At this point we must make sure that these points are different points and that they are members of the list of points that define the triangle. We make this check by calling the subroutine memberOf. Next, we check whether there is a closing right parenthesis. We now compute the center and the radius of the excircle by calling the subroutine excircle. The coordinates of the center are stored in the variables $xc and $yc, while the radius is stored in the variable $r. If the next non-blank input character is a left square bracket, then we know the user has specified the optional part. We use the subroutine expr to get the value of the optional part. The value of the optional part is stored in the variable $R. At this point we check whether the sum of the radius plus the optional part is equal to zero and if it is we continue with the next input line. Next, we check for a closing right square bracket. We are now ready to emit the source code. The first thing we must check is that the radius is not too big for PiCTeX, i.e., not greater than 500/2.845. Then we print some informative text to the output file and of course the actual code. We use the following code to do the job:

```
\circulararc 360 degrees from (xc+R) yc center xc yc
```

The last thing we check is whether there is some trailing garbage.

```
<process drawexcircle command>= (<-U)
    chk_lparen("drawexcircle",$lc);
    my $point1 = get_point($lc);
    next LINE if $point1 eq "_undef_";
    my $point2 = get_point($lc);
    next LINE if $point2 eq "_undef_";
    my $point3 = get_point($lc);
    next LINE if $point3 eq "_undef_";
    my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point1});
    my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point2});
    my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point3});
    if (triangleArea($x1, $y1, $x2, $y2, $x3, $y3) < 0.0001) {
      PrintErrorMessage("Area of triangle is zero!",$lc);
      next LINE;
    }
    chk_comma($lc);
    my $point4 = get_point($lc);
    if (!memberOf($point4, $point1, $point2, $point3)) {
      PrintErrorMessage("Current point isn't a side point",$lc);
      next LINE;
    }
    next LINE if $point4 eq "_undef_";
    my $point5 = get_point($lc);
    next LINE if $point5 eq "_undef_";
```

```
  if (!memberOf($point5, $point1, $point2, $point3)) {
    PrintErrorMessage("Current point isn't a side point",$lc);
    next LINE;
  }
  if ($point4 eq $point5) {
    PrintErrorMessage("Side points are identical",$lc);
    next LINE;
  }
  chk_rparen("arguments of $cmd",$lc);
  my ($xc, $yc, $r) = excircle($point1, $point2, $point3,
                               $point4, $point5);
  my $R=$r;
  if (s/^\s*\[\s*//) {
     $R += expr($lc);
     if ($R < 0.0001) {
       PrintErrorMessage("Radius has become equal to zero!",$lc);
       next LINE;
     }
     chk_rsb($lc);
  }
  if ($R > (500 / 2.845)) {
    PrintErrorMessage("Radius is greater than 175mm!",$lc);
    next LINE;
  }
  print OUT "%% excircle center = ($xc,$yc) radius = $R\n" if $comments_on;
  printf OUT "\\circulararc 360 degrees from %.5f %.5f center at %.5f %.5f\n",
              $xc+$R, $yc, $xc, $yc;
  chk_comment($lc);
```

The syntax of the drawincircle command is as follows:

```
drawincircle ::= "drawincircle" "(" ThreePoints ")" [ modifier]
modifier     ::= "[" expr "]"
```

where ThreePoints correspond to the points defining the triangle and modifier is an optional modification factor. The first thing we do is to check whether there is an opening left parenthesis. Then we get the names of the three points that define the triangle (variables $point1, $point2, and $point3). Next, we make sure that the area of the triangle defined by these three points is not equal to zero. If it is, then we issue an error message and continue with the next input line. Now, we compute the center and the radius of the incircle (variables $xc, $yc, and $r). If the next non-blank input character is a left square bracket, then we now the user has specified the optional part. We use subroutine expr to get the value of the optional part. The value of the optional part is stored in the variable $R. At this point we check whether the sum of the radius plus the optional part is equal to zero and if it is we continue with the next input line. Next, we check for a closing right square bracket. We are now ready to emit the source code. The first thing we must check is that the radius is not too big for PiCTeX, i.e., not greater than 500/2.845. Then we print some informative text to the output file and of course the actual code. We use the following code to do the job:

```
\circulararc 360 degrees from (xc+R) yc center xc yc
```

The last thing we check is whether there is some trailing garbage.

<process drawincircle command>= **(<-U)**

```
  chk_lparen("drawincircle",$lc);
  my $point1 = get_point($lc);
  next LINE if $point1 eq "_undef_";
```

```
my $point2 = get_point($lc);
next LINE if $point2 eq "_undef_";
my $point3 = get_point($lc);
next LINE if $point3 eq "_undef_";
my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point1});
my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point2});
my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point3});
if (triangleArea($x1, $y1, $x2, $y2, $x3, $y3) < 0.0001) {
  PrintErrorMessage("Area of triangle is zero!",$lc);
  next LINE;
}
my ($xc, $yc, $r) = IncircleCenter($x1,$y1,$x2,$y2,$x3,$y3);
my $R=$r;
if (s/^\s*\[\s*//) {
   $R += expr($lc);
   if ($R < 0.0001) {
     PrintErrorMessage("Radius has become equal to zero!",$lc);
     next LINE;
   }
   chk_rsb($lc);
}
if ($R > (500 / 2.845)) {
  PrintErrorMessage("Radius is greater than 175mm!",$lc);
  next LINE;
}
print OUT "%% incircle center = ($xc,$yc) radius = $R\n" if $comments_on;
printf OUT "\\circulararc 360 degrees from %.5f %.5f center at %.5f %.5f\n",
             $xc+$R, $yc, $xc, $yc;
chk_rparen("arguments of $cmd",$lc);
chk_comment($lc);
```

The command drawPerpendicular command draws a line from point A to line BC, such that it is perpendicular to line BC. The general syntax of the command is as follows:

```
drawPenpedicular ::= "drawPenpedicular" "(" Point "," TwoPoints ")"
```

The first thing we do is to parse the left parenthesis. Then we parse the name of the first point, namely A. If this point is undefined we print an error message and continue with the next line. Next, we parse the expected leading comma and the names of the other two points. Certainly, in case either of these two points has not been defined, we simply print an error message and continue with the next input line. Finally, we check for a closing right parenthesis and a possible trailing comment. Now we are ready to compute the coordinates of the foot of the perpendicular line. We do so my calling subroutine perpendicular. Certainly, before we do this we have to get the coordinates of the points that we have parsed. Finally, we output the PiCTeX code:

```
\plot x1 y1   xF xY /
```

where x1 and y1 are coordinates of the point A and xF and yF the coordinates of the foot.

<process drawPerpendicular command>= (<-U)

```
chk_lparen($cmd,$lc);
my $A = get_point($lc);
next LINE if $A eq "_undef_";
chk_comma($lc);
my $B = get_point($lc);
next LINE if $A eq "_undef_";
```

```
      s/\s*//; #ignore white space
      my $C = get_point($lc);
      next LINE if $A eq "_undef_";
      chk_rparen("arguments of $cmd",$lc);
      chk_comment($lc);
      #
      #start actual computation
      #
      my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$A});
      my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$B});
      my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$C});
      my ($xF, $yF) = perpendicular($x1, $y1, $x2, $y2, $x3, $y3);
      printf OUT "\\plot  %.5f %.5f    %.5f %.5f  /\n",
             $x1, $y1, $xF, $yF;
```

The drawpoint command has a number of points as arguments and produces PiCTeX code that draws a plot symbol at the coordinates of each point. The syntax of the command is as follows:

```
drawpoint ::= "drawpoint" "(" Point { separator Point } ")"
```

The while loop is used to consume all points that are between an opening left parenthesis and a closing right parenthesis. All points are pushed on the local array PP. When we have parsed the lists of points, we call the subroutine drawpoints to emit the actual PiCTeX code. Finally, we check whether there is a closing parenthesis parenthesis, and whether there is some trailing text that makes no sense. In case there are no points between the parentheses, then we issue an appropriate error message and we continue with the next input line.

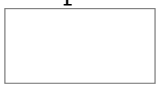
```
<process drawpoint command>= (<-U)
      my ($stacklen);
      chk_lparen("$cmd",$lc);
      if (/^\)/) {
        PrintErrorMessage("There are no point to draw",$lc);
        next LINE;
      }
      my(@PP);
      DRAWPOINTS:while(1) {
        if (s/^([^\W\d_]\d{0,3})//i) { #point name
          $P = $1;
          if (!exists($PointTable{lc($P)})) {
            PrintErrorMessage("Undefined point $P",$lc);
            next DRAWPOINTS;
          }
          else {
            push (@PP,$P);
            s/\s*//;
          }
        }
        else {
          last DRAWPOINTS;
        }
      }
      drawpoints(@PP);
      chk_rparen("arguments of $cmd",$lc);
      chk_comment($lc);
```

The syntax of the drawRightAngle command is as follows:

```
    drawRightAngle "(" ThreePoints  "," dist ")"
    dist ::= expr | TwoPoints
```

Before we proceed with the actual computation we parse the left parenthesis, the three points, the comma, the dist, and the right parenthesis. In case we have neither three points nor a dist we print an error message and continue with the next input line, i.e., these errors are irrecoverable. The names of the three points are stored in variables $point1, $point2, and $point3. The value of the distance is stored in the variable $dist. Let's now explain the semantics of this command.

Our aim is to draw lines S_1-S, S_2-S (S_1 and S_2 are at distance d from B). All the relevant points are depicted in the following figure:



Some notes are in order:

1. BS bisects angle ABC, and meets AC in Q, so start by determining point Q, then determine S, and then S_1 and S_2, and then draw S_1-S and S_2-S.
2. Distance AQ is given by AC/(1+tan(BCA))
3. The coordinates of Q are computed using the subroutine pointOnLine.
4. Now we compute the coordinates of S on line BQ.
5. We compute the coordinates of S_1 and S_2 by using The subroutine pointOnLine.

In order to implement the above steps we first compute the length of the line AB. Note that A is $point1, etc. Next we compute the angle BAC. Now we compute the distance AQ (variable $line1). The coordinates of point Q are stored in variables $xQ and $yQ. The coordinates of point S are stored in variables $xS and $yS. Now we have to determine the coordinates of points S_1 and S_2. These coordinates are stored in variables $xS1, $yS1 and $xS2, $yS2, respectively. Finally, we emit the PiCTeX target code.

<process drawRightAngle command>= (<-U)

```
chk_lparen("drawRightAngle",lc);
my $point1 = get_point($lc);
next LINE if $point1 eq "_undef_";
my $point2 = get_point($lc);
next LINE if $point2 eq "_undef_";
my $point3 = get_point($lc);
next LINE if $point3 eq "_undef_";
my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point1});
my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point2});
my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point3});
chk_comma($lc);
my $dist = expr($lc);
chk_rparen("arguments of $cmd",$lc);
chk_comment($lc);
#
#actual computation
#
my $lengthAC = Length($x1, $y1, $x3, $y3);
my $angleBAC = Angle($x2, $y2, $x1, $y1, $x3, $y3);
my $line1 = $lengthAC / (1 + tand($angleBAC));
#
# determine coordinates of point Q
#
```

```
    my ($xQ, $yQ) = pointOnLine($x1, $y1, $x3, $y3, $line1);
    #
    # determine coordinates of point S
    #
    my $deltax = $xQ - $x2;
    my $deltay = $yQ - $y2;
    my $lengthBQ = sqrt($deltax * $deltax + $deltay * $deltay);
    my $xS = $x2 + ($dist * sqrt(2) * $deltax / $lengthBQ);
    my $yS = $y2 + ($dist * sqrt(2) * $deltay / $lengthBQ);
    #
    # determine coordinates of points S1 and S2
    #
    my ($xS1, $yS1) = pointOnLine($x2, $y2, $x3, $y3, $dist);
    ($xS2, $yS2) = ($xS, $yS);
    #
    # emit PiCTeX code
    #
    printf OUT "\\plot  %.5f %.5f    %.5f %.5f   /\n",
           $xS1, $yS1, $xS2, $yS2;
    ($xS1, $yS1) = pointOnLine($x2, $y2, $x1, $y1, $dist);
    printf OUT "\\plot  %.5f %.5f    %.5f %.5f   /\n",
           $xS1, $yS1, $xS2, $yS2;
```

The command drawsquare has two arguments: a point, which specifies the coordinates of the point where the square will be placed, and a number, which specifies the length of the side of the square. The syntax of the command is as follows:

"drawSquare" "(" Point "," expression ")"

Note that RWDN has suggested to alter the value of the $side variable (see the line with RWDN comment).

```
<process drawsquare command>= (<-U)
   chk_lparen("drawSquare",$lc);
   my $p = get_point($lc);
   chk_comma($lc);
   my $side = expr($lc);
   $side = $side - (1.1 * $LineThickness/$xunits); #Suggested by RWDN
   my ($x,$y,$pSV,$pS) = unpack("d3A*",$PointTable{$p});
   printf OUT "\\put {%s} at %.5f %.5f %%drawsquare\n", drawsquare($side), $x, $y;
   chk_rparen("arguments of $cmd",$lc);
   chk_comment($lc);
```

The argument of the inputfile* command is a file name that is always enclosed in parentheses:

```
 starred-input-file ::= "inputfile*" "(" file-name ")"
 file-name ::= (alpha | period) { alpha | period }
 alpha ::= letter | digit | "_" | "-"
```

Note, that the input file is assumed to contain TeX code. We first check to see if there is a left parenthesis. Then we consume the file name. We check if the file exists and then we copy verbatim the input file to the output file. Next, we check for the closing parenthesis. Now, if there is a trailing comment we copy it to the output file depending on the value of the variable $comments_on, else if there is some other text we simply ignore it and issue a warning message.

```
<process inputfile* command>= (<-U)
     chk_lparen("inputfile*",$lc);
     my $row_in = "";
     if (s/^((\w|-|\.)+)//) {
       $row_in = $1;
```

```
}
else {
  PrintErrorMessage("No input file name found",$lc);
  next LINE;
}
if (!(-e $row_in)) {
  PrintErrorMessage("File $row_in does not exist",$lc);
  next LINE;
}
open(ROW, "$row_in")|| die "Can't open file $row_in\n";
while (defined($in_line=<ROW>)) { print OUT $in_line; }
print OUT "%% ... end of input file <$row_in>\n";
close ROW;
chk_rparen("input file name",$lc);
chk_comment($lc);
```

The inputfile command has at most two arguments, second being optional: a file name enclosed in curly brackets and the number of times this file should be included in square brackets:

```
inputfile ::= "inputfile" "(" file-name ")" [ Times ]
Times ::= "[" expr "]"
```

Note that the input file is assumed to contain mathspic commands. In addition, if the expression is equal to a decimal number, it is truncated. As in the case of the inputfile* command we parse the left parenthesis, the file name, the right parenthesis and the optional argument if it exists. In order to process the commands contained in the input file, we call The subroutine process_input.

<process inputfile command>= (<-U)

```
chk_lparen("inputfile",$lc);
my $comm_in = "";
if (s/^((\w|-|\.)+)//) {
  $comm_in = $1;
}
else {
  PrintErrorMessage("No input file name found",$lc);
  next LINE;
}
if (!(-e $comm_in)) {
  PrintErrorMessage("File $comm_in does not exist",$lc);
  next LINE;
}
chk_rparen("input file name",$lc);
my $input_times = 1; #default value
if (s/^\[//) {
  $input_times = expr($lc);
  chk_rsb("optional argument",$lc);
}
print OUT "%% ... start of file <$comm_in> loop [$input_times]\n";
for (my $i=0; $i<int($input_times); $i++) {
  open(COMM,"$comm_in") or die "Can't open file $comm_in\n";
  print OUT "%%% Iteration number: ",$i+1,"\n";
  my $old_file_name = $curr_in_file;
  process_input(COMM,"File $comm_in, ");
  $curr_in_file = $old_file_name;
  close COMM;
}
print OUT "%% ... end of file <$comm_in> loop [$input_times]\n";
chk_comment($lc);
```

The linethickness command should be used to set the thickness of lines. The command has one argument, which is a length or the word default. The default line thickness is 0.4 pt.

```
<process linethickness command>= (<-U)
     chk_lparen("linethickness", $lc);
     if (s/^default//i) {
       print OUT "\\linethickness=0.4pt\\Linethickness{0.4pt}%%\n";
       print OUT "\\setplotsymbol  ({\\usefont{OT1}{cmr}{m}{n}\\tiny .})%\n";
       $LineThickness = setLineThickness($xunits,"0.4pt");
     }
     else {
       my $length = expr($lc);
       if (s/^\s*($units)//i) {
         my $units = $1;
         printf OUT "\\linethickness=%.5f%s\\Linethickness{%.5f%s}%%\n",
                $length, $units, $length, $units;
         $LineThickness = setLineThickness($xunits,"$length$units");
         my $mag;
         if ($units eq "pc") {
           $mag = $length * 12;
         }
         elsif ($units eq "in") {
           $mag = $length * 72.27;
         }
         elsif ($units eq "bp") {
           $mag = $length * 1.00375;
         }
         elsif ($units eq "cm") {
           $mag = $length * 28.45275;
         }
         elsif ($units eq "mm") {
           $mag = $length * 2.845275;
         }
         elsif ($units eq "dd") {
           $mag = $length * 1.07001;
         }
         elsif ($units eq "cc") {
           $mag = $length * 0.08917;
         }
         elsif ($units eq "sp") {
           $mag = $length * 0.000015259;
         }
         elsif ($units eq "pt") {
           $mag = $length;
         }
         $mag = 10 * $mag / 1.00278219;
         printf OUT "\\font\\CM=cmr10 at %.5fpt%%\n", $mag;
         print OUT "\\setplotsymbol  ({\\CM .})%\n";
       }
       else {
         PrintErrorMessage("Did not found expect units part",$lc);
       }
     }
     chk_rparen("linethickness", $lc);
     chk_comment($lc);
```

We first output the input line as a comment into the output file. Now, after the paper token we look for an opening brace. Then we process the units part of the command, if the token units is present. Note that the units part is optional. Next we process the xrange and the yrange part of the command, which are also optional parts of the command. We are now ready to process the axis part. Note, that the user is allowed to alternatively specify this part with the word axes.

The variable $axis is supposed to hold the various data relate to the axis part. The last thing we check is the ticks part. In case the user has not specified this part we assume that both ticks are equal to zero. If everything is according to the language syntax, we expect a closing right curly bracket. Now, that we have all relevant information we can output the rest of the code, as some parts of it have already been output during parsing. The last thing we do is to check whether there is any trailing comment.

```
<process paper command>= (<-U)
     chk_lcb("paper", $lc);
     if (s/^units(?=\W)//i)
     {
        <process unit part>
        $nounits = 0;
     }
     else
     {
        $nounits = 1;
     }
     s/^,\s*// or s/\s*//;
     if (s/^xrange//i)
     {
        <process xrange part>
        $noxrange = 0;
     }
     else
     {
        $noxrange = 1;
     }
     s/^,\s*// or s/\s*//;
     if (s/^yrange//i)
     {
        <process yrange part>
        $noyrange = 0;
     }
     else
     {
        $noyrange = 1;
     }
     <generate plot area related commands>
     s/^,\s*// or s/\s*//;
     $axis = "";
     if (s/^ax[ei]s(?=\W)//i)
     {
        <process axis part>
     }
     $axis = uc($axis);
     s/^,\s*// or s/\s*//;
     if (s/^ticks(?=\W)//i)
     {
        <process ticks part>
     }
     else
     {
        $xticks = $yticks = 0;
     }
     chk_rcb("paper", $lc);
     <generate the rest of the code for the paper command>
     chk_comment($lc);
```

We first check whether there is a left parenthesis. Next, we check whether there is decimal number or a variable name. In case there isn't one we assume it is the number 1. Now, we get the units. If there is no valid unit, we issue an error and the x-unit is set to its

default value. In case, there is a trailing comma, we assume the user wants also to specify the y-unit and we process this part just like we did with the x-unit part. Finally, we output the corresponding PiCTeX command. In case there is no y-unit we assume it is equal to the x-unit.

<process unit part>= (<-U)

```
chk_lparen("units",$lc);
if(s/^\)//)
{
  PrintWarningMessage("Missing value in \"units\"--default is 1pt",
                      $lc);
  $xunits = "1pt";
}
else {
  $xunits = expr($lc);
  s/\s*//;
  if (s/^($units)//i) {
    $xunits .= "$1";
    $LineThickness = setLineThickness($xunits,"0.4pt");
  }
  elsif(s/^(\w)+//i) {
    PrintErrorMessage("$1 is not a valid mathspic unit",$lc);
    $xunits = "1pt";
  }
  else {
    PrintErrorMessage("No x-units found",$lc);
    $xunits = "1pt";
  }
  s/\s*//; #ignore white space
  if (s/^,//) {  # there is a comma so expect an y-units
    s/\s*//; #ignore white space
    $yunits = expr($lc);
    s/\s*//; #ignore white space
    if (s/^($units)//i) {
      $yunits .= "$1";
    }
    elsif(s/^(\w)+//i) {
      PrintErrorMessage("$1 is not a valid mathspic unit",$lc);
      $yunits = "1pt";
    }
    else {
      PrintErrorMessage("No y-units found",$lc);
      $yunits = $xunits;
    }
  }
  else {
    $yunits = $xunits;
  }
  chk_rparen("units",$lc);
}
```

The xrange token must be followed by a left parenthesis, so we check whether the next token is a left parenthesis. We store in the variables $xlow and $xhigh the values of the range. The range is specified as pair of decimal numbers/variable/pair of points, separated by a comma. We use the subroutine ComputeDist to get the value of the lower end and the upper end of the range. The last thing we check is whether the lower end is less than the upper end. If this isn't the case we issue an error message and we skip into the next input line.

<process xrange part>= (<-U)

```
chk_lparen("xrange",$lc);
my $ec;
($xlow,$ec) = ComputeDist($lc);
next LINE if $ec == 0;
chk_comma($lc);
($xhigh,$ec) = ComputeDist($lc);
next LINE if $ec == 0;
if ($xlow >= $xhigh)
{
   PrintErrorMessage("xlow >= xhigh in xrange",$lc);
   next LINE;
}
chk_rparen("$xhigh",$lc);
```

The yrange token must be followed by a left parenthesis, so we check whether the next token is a left parenthesis. We store in the variables $ylow and $yhigh the values of the range. The range is specified as pair of decimal numbers/variable/pair of points, separated by a comma. We use the subroutine ComputeDist to get the value of the lower end and the upper end of the range. The last thing we check is whether the lower end is less than the upper end. If this isn't the case we issue an error message and we skip into the next input line.

<process yrange part>= (<-U)

```
chk_lparen("yrange",$lc);
my $ec;
($ylow,$ec) = ComputeDist($lc);
next LINE if $ec == 0;
chk_comma($lc);
($yhigh,$ec) = ComputeDist($lc);
next LINE if $ec == 0;
if ($ylow >= $yhigh)
{
   PrintErrorMessage("ylow >= yhigh in yrange",$lc);
   next LINE;
}
chk_rparen("$yhigh",$lc);
```

The showAngle command has three arguments that correspond to three distinct points and emits a comment of the form:

```
%% angle(ABC) = 45
```

Note that the computed angle is expressed in degrees.

<process showAngle command>= (<-U)

```
chk_lparen("showangle",$lc);
my $point_1 = get_point($lc);
my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_1});
my $point_2 = get_point($lc);
my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point_2});
my $point_3 = get_point($lc);
my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point_3});
my $angle = Angle($x1, $y1, $x2, $y2, $x3, $y3);
$angle = 0 if $angle == -500;
printf OUT "%%%% angle(%s%s%s) = %.5f deg ( %.5f rad)\n", $point_1,
       $point_2, $point_3, $angle, $angle*D2R;
chk_rparen("Missing right parenthesis", $lc);
```

The showArea command has three arguments that correspond to three

distinct points and emits a comment of the form:

%% area(ABC) = 45

Note that the computed angle is expressed in degrees.

<process showArea command>= (<-U)

```
chk_lparen("showarea",$lc);
my $point_1 = get_point($lc);
my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_1});
my $point_2 = get_point($lc);
my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point_2});
my $point_3 = get_point($lc);
my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$point_3});
print OUT "%% area($point_1$point_2$point_3) = ",
      triangleArea($x1, $y1, $x2, $y2, $x3, $y3), "\n";
chk_rparen("Missing right parenthesis", $lc);
```

The showLength command has two arguments that correspond to two distinct points and emits a comment of the form:

%% length(AB) = 45

Note that the computed angle is expressed in degrees.

<process showLength command>= (<-U)

```
chk_lparen("showlength",$lc);
my $point_1 = get_point($lc);
my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$point_1});
my $point_2 = get_point($lc);
my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$point_2});
print OUT "%% length($point_1$point_2) = ",
      Length($x1, $y1, $x2, $y2), "\n";
chk_rparen("Missing right parenthesis", $lc);
```

If the user hasn't specified units then we use the previous values to set the coordinate system. If the user hasn't specified either the xunits part or the yunits, then we don't emit code. In case he/she has specified both parts we generate the command that sets the plot area.

<generate plot area related commands>= (<-U)

```
if (!$nounits)
{
   printf OUT "\\setcoordinatesystem units <%s,%s>\n",
              $xunits,$yunits;
}
if(!$noxrange && !$noyrange)
{
   printf OUT "\\setplotarea x from %.5f to %.5f, y from %.5f to %.5f\n",
              $xlow, $xhigh, $ylow, $yhigh;

}
```

We first check to see whether there is an opening left parenthesis. Next we get the various options the user may have entered. The valid options are the letters L, R, T, B, X, and Y. These letters may be followed by an optional star * with space characters between the letter and the star. We use a loop, that stops when a right parenthesis is found, to go through all possible arguments and append each argument in the string $axis. Note one can have blank space between different arguments. The last thing we do is to check for the closing right parenthesis.

<process axis part>= (<-U)

```
        chk_lparen("axis",$lc);
        while(/^[^\)]/)
        {
            if (s/^([lrtbxy]{1}\*?)//i)
            {
               $axis .= $1;
            }
            elsif (s/^([^lrtbxy])//i)
            {
               PrintErrorMessage("Non-valid character \"$1\" in axis()",$lc);
            }
            s/\s*//;
        }
        chk_rparen("axis(arguments",$lc);
```

As usual we start by skipping white space. Next we check whether there is an opening left parenthesis. Now, we expect two numbers/variables/pair of point representing the ticks increment value. These ticks increment values must be separated by a comma (and possibly some white space around them). We use the subroutine ComputeDist to get the value of the ticks increment value and we assign to the variables $xticks and $yticks the value of x-ticks and y-ticks increment value. In case there is a problem we issue an error message and continue with the next line. The last thing we check is whether there is a closing right parenthesis.

<process ticks part>= (<-U)

```
        chk_lparen("ticks",$lc);
        my $ec;
        ($xticks,$ec) = ComputeDist($lc);
        next LINE if $ec == 0;
        chk_comma($lc);
        ($yticks,$ec) = ComputeDist($lc);
        next LINE if $ec == 0;
        chk_rparen("ticks(arguments",$lc);
```

We actually emit code if the user has specified either the X or Y option in the axis part. If the user has specified the Y* or the X* option in the axis part, we just emit the commands \axis left shiftedto x=0 or \axis bottom shiftedto y=0 respectively and exit. If the use has specified ticks, then, depending on the options he had supplied with the axis part, we emit code that implements the user's wishes. **** HERE WE MUST EXPLAIN THE MEANING OF THE CODE EMITTED!!! *****

<generate the rest of the code for the paper command>= (<-U)

```
YBRANCH: {
   if (index($axis, "Y")>-1)
   {
      if (index($axis, "Y*")>-1)
      {
         print OUT "\\axis left shiftedto x=0 / \n";
         last YBRANCH;
      }
      if ($yticks > 0)
      {
         if (index($axis, "T")>-1 && index($axis, "B")==-1)
         {
            print OUT "\\axis left shiftedto x=0 ticks numbered from ";
            print OUT "$ylow to -$yticks by $yticks\n      from $yticks to ";
```

```
         print OUT $yhigh-$yticks," by $yticks /\n";
      }
      elsif (index($axis, "T")==-1 && index($axis, "B")>-1)
      {
         print OUT "\\axis left shiftedto x=0 ticks numbered from ";
         print OUT $ylow+$yticks," to -$yticks by $yticks\n      from ";
         print OUT "$yticks to $yhigh by $yticks /\n";
      }
      elsif (index($axis, "T")>-1 && index($axis, "B")>-1)
      {
         print OUT "\\axis left shiftedto x=0 ticks numbered from ";
         print OUT $ylow+$yticks," to -$yticks by $yticks\n      from ";
         print OUT "$yticks to ",$yhigh-$yticks," by $yticks /\n";
      }
      else
      {
         print OUT "\\axis left shiftedto x=0 ticks numbered from ";
         print OUT "$ylow to -$yticks by $yticks\n      from ";
         print OUT "$yticks to $yhigh by $yticks /\n";
      }
   }
   else
   {
      print OUT "\\axis left shiftedto x=0 /\n";
   }
}
}
XBRANCH: { if (index($axis, "X")>-1)
{
   if (index($axis, "X*")>-1)
   {
      print OUT "\\axis bottom shiftedto y=0 /\n";
      last XBRANCH;
   }
   if ($xticks > 0)
   {
      if (index($axis, "L")>-1 && index($axis, "R")==1)
      {
         print OUT "\\axis bottom shiftedto y=0 ticks numbered from ";
         print OUT $xlow + $xticks," to -$xticks by $xticks\n      from";
         print OUT " $xticks to $xhigh by $xticks /\n";
      }
      elsif (index($axis, "L")==-1 && index($axis, "R")>-1)
      {
         print OUT "\\axis bottom shiftedto y=0 ticks numbered from ";
         print OUT "$xlow to -$xticks by $xticks\n      from ";
         print OUT "$xticks to ",$xhigh-$xticks," by $xticks /\n";
      }
      elsif (index($axis, "L")>-1 && index($axis, "R")>-1)
      {
         print OUT "\\axis bottom shiftedto y=0 ticks numbered from ";
         print OUT $xlow + $xticks," to -$xticks by $xticks\n      from ";
         print OUT "$xticks to ",$xhigh - $xticks," by $xticks /\n";
      }
      else
      {
         print OUT "\\axis bottom shiftedto y=0 ticks numbered from ";
         print OUT "$xlow to -$xticks by  $xticks\n      from ";
         print OUT "$xticks to $xhigh by $xticks /\n";
      }
   }
   else
   {
      print OUT "\\axis bottom shiftedto y=0 /\n";
   }
} }
LBRANCH: {if (index($axis, "L")>-1)
{
   if (index($axis, "L")>-1)
```

```
   {
      if (index($axis, "L*")>-1)
      {
         print OUT "\\axis left /\n";
         last LBRANCH;
      }
      if ($yticks > 0)
      {
         print OUT "\\axis left ticks numbered from ";
         print OUT "$ylow to $yhigh by $yticks /\n";
      }
      else
      {
         print OUT "\\axis left /\n";
      }
   }
} }
RBRANCH: { if (index($axis, "R")>-1)
{
   if (index($axis, "R*")>-1)
   {
      print OUT "\\axis right /\n";
      last RBRANCH;
   }
   if ($yticks > 0)
   {
      print OUT "\\axis right ticks numbered from $ylow to $yhigh by ";
      print OUT "$yticks /\n";
   }
   else
   {
      print OUT "\\axis right /\n";
   }
} }
TBRANCH: { if (index($axis, "T")>-1)
{
   if (index($axis, "T*")>-1)
   {
      print OUT "\\axis top /\n";
      last TBRANCH;
   }
   if ($xticks > 0)
   {
      print OUT "\\axis top ticks numbered from $xlow to $xhigh by ";
      print OUT "$xticks /\n";
   }
   else
   {
      print OUT "\\axis top /\n";
   }
} }
BBRANCH: { if (index($axis, "B")>-1)
{
   if (index($axis, "B*")>-1)
   {
      print OUT "\\axis bottom /\n";
      last BBRANCH;
   }
   if ($xticks > 0)
   {
      print OUT "\\axis bottom ticks numbered from $xlow to $xhigh by ";
      print OUT "$xticks /\n";
   }
   else
   {
      print OUT "\\axis bottom /\n";
   }
} }
```

The syntax of the point commands follows:

```
point[*](PointName){Coordinates}[PointSymbol]
```

where PointName is valid point name, Coordinates is either a pair of numbers denoting the coordinates of the point or an expression by means of which the system computes the coordinates of the point, and the PointSymbol is a valid T_EX command denoting a point symbol. A valid point name consists of a letter and at most two trailing digits. That is, the names a11, b2 and c are valid names while qw and s123 are not. The first thing we do is to set the point shape to the default symbol (this has been initialized in the main program). Next, we check whether we have a pointcommand or a point* simply by inspecting the very next token. Note that there must be no blank spaces between the token point and the star symbol. Next, we get the point name: remember that the point name is surrounded by parentheses. In case we don't find a valid point name we issue an error message and continue with the next line of input. Suppose the point name was a valid one. If we have a point* command we must ensure that the this particular point name has been defined. If we have a point command we must ensure that this particular point name has not been defined. Point names are stored in the hash %PointTable. We are now ready to process the coordinates part and the optional plot symbol part.

```
<process point/point* commands>= (<-U)
     my ($pointStar, $PointName, $origPN);
     $pointStar = 0; # default value: he have a point command
     $pointStar = 1 if s/^\*//;
     chk_lparen("point" . (($pointStar)?"*":""),$lc);
     if (s/^([^\W\d_](?![^\W\d_])\d{0,3})//i) {
     #
     # Note: the regular expression (foo)(?!bar) means that we are
     # looking a foo not followed by a bar. Moreover, the regular
     # expression [^\W\d_] means that we are looking for letter.
     #
       $origPN = $1;
       $PointName = lc($1);
     }
     else {
       PrintErrorMessage("Invalid point name",$lc);
       next LINE;
     }
     #if ($pointStar and !exists($PointTable{$PointName})) {
     #  PrintWarningMessage("Point $origPN has not been defined",$lc);
     #}
     if (!$pointStar and exists($PointTable{$PointName})) {
        PrintWarningMessage("Point $origPN has been used already",$lc);
     }
     chk_rparen("point" . (($pointStar)?"*":""). "($origPN",$lc);
     chk_lcb("point" . (($pointStar)?"*":""). "($origPN)",$lc);
     my ($Px, $Py);
     <process coordinates>
     chk_rcb("coordinates part",$lc);
     my $sv = $defaultsymbol;
     my $sh = $defaultLFradius;
     my $side_or_radius = undef;
     if (s/^\[\s*//) { # the user has opted to specify the optional part
        <process optional point shape part>
        chk_rsb("optional part",$lc);
     }
     # to avoid truncation problems introduced by the pack function, we
```

```
    # round each number up to five decimal digits
    $Px = sprintf("%.5f", $Px);
    $Py = sprintf("%.5f", $Py);
    print OUT "%% point$Point_Line \t$origPN = ($Px, $Py)\n" if $comments_on;
    chk_comment($lc);
    $PointTable{$PointName} = pack("d3A*",$Px,$Py,$sh,$sv);
    if (defined($side_or_radius)) {
      $DimOfPoint{$PointName} = $side_or_radius;
    }
```

In this section we parse the Coordinates part of the point command. The complete syntax of the Coordinates part follows:

```
Coordinates ::= Variable                                          |
                Distance "," Distance                             |
                "midpoint" "(" Point-Name Point-Name ")"          |
                "pointOnLine" "(" Two-Points "," Distance ")"     |
                "intersection" "(" Two-Points "," Two-Points ")"  |
                "perpendicular" "(" Point-Name "," Two-Points ")" |
                "circumCircleCenter" "(" Three-Points ")          |
                "incircleCenter" "(" Three-Points ")"             |
                "excircleCenter" "(" Three-Points "," Two-Points ")" |
                 Point-Name [ "," Modifier ]

Modifier ::= "shift" "(" Distance "," Distance ")"                |
             "polar" "(" Distance, Distance [ "deg" | "rad" ] ")" |
             "rotate" "(" Point-Name, Distance [ "deg" | "rad" ] ")" |
             "vector" "(" Two-Points ")"

Distance ::= expression

Two-Points ::= Point-Name Point-Name

Three-Points ::= Point-Name Two-Points
```

We now briefly explain the functionality of each option:

- midpoint(AB): the midpoint between points A and B
- pointOnLine(AB,d): point at distance d from A towards B
- intersection(AB,CD): intersection of lines defined by AB and CD
- perpendicular(A,BC): point of the foot of the perpendicular from A to line BC
- circumCircleCenter(ABC): center of circumcircle of triangle ABC
- incircleCenter(ABC):center of incircle of triangle ABC
- excircleCenter(ABC,BC): center of excircle of triangle ABC, touching side BC
- A, shift(x,y): Point displaced from A by x and y along the X and Y axes
- A, polar(r,d): Point displaced from A by distance r in direction d
- A, rotate(B,d): Rotate A about B by d

We now explain how the following piece of code operates. In case the first token is a number, we assume that the coordinates are specified by a number and another number, a variable or a pair of points. So, we check whether there is a comma and use the subroutine ComputeDist to get the second coordinate. In case the next token is one of the words perpendicular, intersection, midpoint, pointonline, circumcircleCenter, IncircleCenter, or ExcircleCenter we consume the corresponding token and process the corresponding case. In case the first two tokens are two identifiers, then we assume that we have a

pair of numbers. We compute their distance, check whether there is a leading comma and compute the y-coordinate by calling subroutine ComputeDist. In case the next token is a single identifier, we store its name in the variable $PointA. If this identifier is a defined point name, we check whether the next token is a comma. In case it is, we check whether he token after the comma is either the token shift, polar, or rotate and process each case accordingly. If it is none of these tokens we issue an error message and continue with the next input line. Now, if the token after the identifier isn't a comma, we assume that the coordinates of the point will be identical to those of the point whose name has been stored in the variable $PointA. If the identifier is a variable name, we assume that the x-coordinate is the value of this variable. We check whether the next token is a comma, and compute the y-coordinate by calling the subroutine ComputeDist. The x-coordinate is stored in the variable $Px and the y-coordinate in the variable $Py.

```
<process coordinates>= (<-U)
       if (s/^perpendicular(?=\W)//i) {
         <process perpendicular case>
       }
       elsif (s/^intersection(?=\W)//i) {
         <process intersection case>
       }
       elsif (s/^midpoint(?=\W)//i) {
         <process midpoint case>
       }
       elsif (s/^pointonline(?=\W)//i) {
         <process pointonline case>
       }
       elsif (s/^circumcircleCenter(?=\W)//i) {
         <process circumcircleCenter case>
       }
       elsif (s/^IncircleCenter(?=\W)//i) {
         <process IncircleCenter case>
       }
       elsif (s/^ExcircleCenter(?=\W)//i) {
         <process ExcircleCenter case>
       }
       elsif (/^[^\W\d_]\d{0,3}\s*[^,\w]/) {
         m/^([^\W\d_]\d{0,3})\s*/i;
         if (exists($PointTable{lc($1)})) {
           my $Tcoord = get_point($lc);
           my ($x,$y,$pSV,$pS)=unpack("d3A*",$PointTable{$Tcoord});
           $Px = $x;
           $Py = $y;
         }
         else {
           $Px = expr();
           chk_comma($lc);
           $Py = expr();
         }
       }
       elsif (/[^\W\d_]\d{0,3}\s*,\s*shift|polar|rotate|vector/i) { #a point?
         s/^([^\W\d_]\d{0,3})//i;
         my $PointA = $1;
         if (exists($PointTable{lc($PointA)})) {
           s/\s*//;
           if (s/^,//) {
             s/\s*//;
             if (s/^shift(?=\W)//i) {
               <process shift case>
             }
             elsif (s/^polar(?=\W)//i) {
               <process polar case>
```

```
            }
            elsif (s/^rotate(?=\W)//i) {
              <process rotate case>
            }
            elsif (s/^vector(?=\W)//i) {
              <process vector case>
            }
            else {
              PrintErrorMessage("unexpected token",$lc);
              next LINE;
            }
          }
          else {
            my ($xA,$yA,$pSVA,$pSA)=unpack("d3A*",$PointTable{lc($PointA)});
            $Px = $xA;
            $Py = $yA;
          }
        }
        else {
          PrintErrorMessage("Undefined point $PointA",$lc);
          next LINE;
        }
      }
      else {
        $Px = expr();
        chk_comma($lc);
        $Py = expr();
      }
```

In the following piece of code we process the perpendicular case of the point specification. We first check whether there is an opening left parenthesis. Next, we get the first point name. In case there is no point name, we simply abandon the processing of this line and continue with the next one. Then we see whether there is a trailing comma. Omitting this token yields a non-fatal error. Then we get two more points. As before, if we can't find any of these points this yields a fatal-error. Note, that each time we check that the point names correspond to existing point names. Then, we call subroutine perpendicular to calculate the coordinates of the point.

```
<process perpendicular case>= (<-U)
        chk_lparen("perpendicular",$lc);
        my $FirstPoint = &get_point($lc);
        next LINE if $FirstPoint eq "_undef_";
        chk_comma($lc);
        my $SecondPoint = &get_point($lc);
        next LINE if $SecondPoint eq "_undef_";
        my $ThirdPoint = &get_point($lc);
        next LINE if $ThirdPoint eq "_undef_";
        chk_rparen("No closing parenthesis found",$lc);
        my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$FirstPoint});
        my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$SecondPoint});
        my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$ThirdPoint});
        ($Px, $Py) = perpendicular($x1,$y1,$x2,$y2,$x3,$y3);
```

In the following piece of code we process the intersection case of the point specification. We get the four point names and if there is no error we compute the intersection point by calling subroutine intersection.

```
<process intersection case>= (<-U)
        chk_lparen("intersection",$lc);
        my $FirstPoint = get_point($lc);
```

```
next LINE if $FirstPoint eq "_undef_";
my $SecondPoint = get_point($lc);
next LINE if $SecondPoint eq "_undef_";
chk_comma($lc);
my $ThirdPoint = get_point($lc);
next LINE if $ThirdPoint eq "_undef_";
my $ForthPoint = get_point($lc);
next LINE if $ForthPoint eq "_undef_";
chk_rparen("No closing parenthesis found",$lc);
my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$FirstPoint});
my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$SecondPoint});
my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$ThirdPoint});
my ($x4,$y4,$pSV4,$pS4)=unpack("d3A*",$PointTable{$ForthPoint});
($Px, $Py) = intersection4points($x1,$y1,$x2,$y2,$x3,$y3,$x4,$y4);
```

Given two points A and B, the midpoint option computes the coordinates of a third point that lies on the middle of the line segment defined by these two points. We get the the two points, and then we compute the coordinates of the midpoint by means of the simple formula:

$$m_x=x_1+(y_2 - y_1)/2$$
$$m_y=y_1+(x_2 - x_1)/2$$

```
<process midpoint case>= (<-U)
        chk_lparen("midpoint",$lc);
        my $FirstPoint = &get_point($lc);
        next LINE if $FirstPoint eq "_undef_";
        my $SecondPoint = &get_point($lc);
        next LINE if $SecondPoint eq "_undef_";
        chk_rparen("No closing parenthesis found",$lc);
        my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$FirstPoint});
        my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$SecondPoint});
        $Px = $x1 + ($x2 - $x1)/2;
        $Py = $y1 + ($y2 - $y1)/2;
```

Given two points A and B and length d, the PointOnLine option computes the coordinates of a point that lies d units in the direction from A towards B. We first get the coordinates of the two points that define the line and then we get the distance, which can be a number, a variable, or a pair of points.

```
<process pointonline case>= (<-U)
        chk_lparen("pointonline",$lc);
        my $FirstPoint = &get_point($lc);
        next LINE if $FirstPoint eq "_undef_";
        my $SecondPoint = &get_point($lc);
        next LINE if $SecondPoint eq "_undef_";
        chk_comma($lc);
        # now get the distance
        my $distance = expr($lc);
        chk_rparen("No closing parenthesis found",$lc);
        my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$FirstPoint});
        my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$SecondPoint});
        ($Px, $Py) = pointOnLine($x1,$y1,$x2,$y2,$distance);
```

The circumcircleCenter is used when one wants to compute the coordinates of the center of circle that passes through the three points of a triangle defined by the three arguments of the option. All we do is get the coordinates of the three points and then we call the subroutine circumCircleCenter to compute the center.

```
<process circumcircleCenter case>= (<-U)
          chk_lparen("circumCircleCenter",$lc);
          my $FirstPoint = &get_point($lc);
          next LINE if $FirstPoint eq "_undef_";
          my $SecondPoint = &get_point($lc);
          next LINE if $SecondPoint eq "_undef_";
          my $ThirdPoint = &get_point($lc);
          next LINE if $ThirdPoint eq "_undef_";
          chk_rparen("No closing parenthesis found",$lc);
          my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$FirstPoint});
          my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$SecondPoint});
          my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$ThirdPoint});
          ($Px, $Py,$r) = &circumCircleCenter($x1,$y1,$x2,$y2,$x3,$y3,$lc);
          next LINE if $Px == 0 and $Py == 0 and $r == 0;
```

The IncircleCenter option is to determine the coordinates of a point that is the center of circle that internally touches the sides of a triangle defined by three given points. The coordinates are computed by the subroutine IncircleCenter.

```
<process IncircleCenter case>= (<-U)
          chk_lparen("IncircleCenter",$lc);
          my $FirstPoint = &get_point($lc);
          next LINE if $FirstPoint eq "_undef_";
          my $SecondPoint = &get_point($lc);
          next LINE if $SecondPoint eq "_undef_";
          my $ThirdPoint = &get_point($lc);
          next LINE if $ThirdPoint eq "_undef_";
          chk_rparen("No closing parenthesis found",$lc);
          my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{$FirstPoint});
          my ($x2,$y2,$pSV2,$pS2)=unpack("d3A*",$PointTable{$SecondPoint});
          my ($x3,$y3,$pSV3,$pS3)=unpack("d3A*",$PointTable{$ThirdPoint});
          ($Px, $Py, $r) = IncircleCenter($x1,$y1,$x2,$y2,$x3,$y3);
```

The ExcircleCenter option is used to define the coordinates of point that is the center of an excircle of a triangle. We first check whether there is an opening left parenthesis. Next, we get the names of the three points that define the triangle. Then, we check whether there is a comma. Now we get the names of the two points that define one side of the triangle. We check whether the two points we get are of the set of the triangle points. If not we issue an error message and continue with the next input line. Then we make sure that these two points are not identical. We compute the actual coordinates by calling the subroutine excircle. Finally, we make sure there is a closing right parenthesis.

```
<process ExcircleCenter case>= (<-U)
          chk_lparen("ExcircleCenter",$lc);
          my $PointA = get_point($lc);
          next LINE if $PointA eq "_undef_";
          my $PointB = get_point($lc);
          next LINE if $PointB eq "_undef_";
          my $PointC = get_point($lc);
          next LINE if $PointC eq "_undef_";
          chk_comma($lc);
          my $PointD = &get_point($lc);
          next LINE if $PointD eq "_undef_";
          if (!memberOf($PointD, $PointA, $PointB, $PointC)) {
            PrintErrorMessage("Current point isn't a side point",$lc);
            next LINE;
          }
          my $PointE = get_point($lc);
          next LINE if $PointE eq "_undef_";
```

```
        if (!memberOf($PointE, $PointA, $PointB, $PointC)) {
          PrintErrorMessage("Current point isn't a side point",$lc);
          next LINE;
        }
        if ($PointD eq $PointE) {
          PrintErrorMessage("Side points are identical",$lc);
          next LINE;
        }
        ($Px, $Py, $r) = excircle($PointA, $PointB, $PointC,
                                  $PointD, $PointE);
        chk_rparen("after coordinates part",$lc);
```

The shift option allows us to define a point's coordinates relative to the coordinates of an existing point by using two shift parameters. Each parameter can be either a float, a variable name, or a pair of points.

<process shift case>= (<-U U->)

```
        chk_lparen("shift",$lc);
        my $dist1 = expr($lc);
        chk_comma($lc);
        my $dist2 = expr($lc);
        my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{lc($PointA)});
        $Px = $x1 + $dist1;
        $Py = $y1 + $dist2;
        chk_rparen("shift part",$lc);
```

The polar option allows us to define a point's coordinates relative to the coordinates of an existing point using the polar coordinates of some other point. We first check whether there is a left parenthesis, Then we parse the various parts of the polar option. In case the user has specified the angle in degrees, we have to transform it into radians, as all trigonometric function expect their arguments to be radians. Next, we compute the coordinates of the point. We conclude by checking whether there is a closing parenthesis.

<process polar case>= (<-U U->)

```
        chk_lparen("polar",$lc);
        my ($R1, $Theta1);
        $R1 = expr($lc);
        chk_comma($lc);
        $Theta1 = expr($lc);
        my ($x1,$y1,$pSV1,$pS1)=unpack("d3A*",$PointTable{lc($PointA)});
        s/\s*//;
        if (s/^rad(?=\W)//i) {
           # do nothing!
        }
        elsif (s/^deg(?=\W)//i) {
          $Theta1 = $Theta1 * PI / 180;
        }
        else {
          #$Theta1 = $Theta1 * PI / 180;
        }
        $Px = $x1 + $R1 * cos($Theta1);
        $Py = $y1 + $R1 * sin($Theta1);
        chk_rparen("after polar part",$lc);
```

The rotate option allows us to define a point's coordinates by rotating an existing point, Q, about a third point, P, by a specified angle. The method to achieve this is to first get the coordinates of

points P and Q and then

1. translate origin to P
2. rotate about P
3. translate from P back to origin, etc

As in the case of the polar option, we check for an opening parenthesis. Next, we parse the point name and the angle. At this point we are able to compute the coordinates of the rotated point. We conclude by checking whether there is a closing parenthesis.

```
<process rotate case>= (<-U)
          chk_lparen("rotate",$lc);
          my $Q = lc($PointA);
          my $P = get_point($lc);
          next LINE if $P eq "_undef_";
          chk_comma($lc);
          my $Theta1 = expr($lc);
          my ($xP,$yP,$pSV1,$pS1)=unpack("d3A*",$PointTable{$P});
          my ($xQ,$yQ,$pSV2,$pS2)=unpack("d3A*",$PointTable{$Q});
          s/\s*//;
          if (s/^rad(?=\W)//i)
          {
             # do nothing!
          }
          elsif (s/^deg(?=\W)//i)
          {
              $Theta1 = $Theta1 * PI / 180;
          }
          else
          {
              $Theta1 = $Theta1 * PI / 180;
          }
          # shift origin to P
          $xQ -= $xP;
          $yQ -= $yP;
          # do the rotation
          $Px = $xQ * cos($Theta1) - $yQ * sin($Theta1);
          $Py = $xQ * sin($Theta1) + $yQ * cos($Theta1);
          # return origin back to original origin
          $Px += $xP;
          $Py += $yP;
          chk_rparen("after rotate part",$lc);
```

vector(PQ) is actually is a shorthand of shift(xQ-xP,yQ-yP). Thus, it is implemented by borrowing code from the shift modifier.

```
<process vector case>= (<-U)
          chk_lparen("vector",$lc);
          my ($x0,$y0,$pSV0,$pS0) = unpack("d3A*",$PointTable{lc($PointA)});
          my $P = get_point($lc);
          my $Q = get_point($lc);
          my ($x1,$y1,$pSV1,$pS1) = unpack("d3A*",$PointTable{$P});
          my ($x2,$y2,$pSV2,$pS2) = unpack("d3A*",$PointTable{$Q});
          $Px = $x0 + $x2 - $x1;
          $Py = $y0 + $y2 - $y1;
          chk_rparen("vector part",$lc);
```

When lines are drawn to a point, the line will (unless otherwise specified) extend to the point location. However, this can be prevented by allocating an optional circular line-free zone to a point

by specifying the radius (in square brackets) of the optional point shape part. Currently, in this part we are allowed to describe the point shape and the radius value. If only the radius is specified, e.g., [radius=5], then the line-free zone will be applied to the default point character, i.e., $\bullet$ or whatever it has been set to. Here is the syntax we employ:

```
Optional_point_shape_part ::= "[" [ symbol_part ] [","] [ radius_part ]"
symbol_part               ::= "symbol" "=" symbol
symbol                    ::= "circle" "(" expression ")" |
                              "square" "(" expression ")" |
                              LaTeX_Code
radius_part               ::= "radius" "=" expression
```

Note that it is possible to have right square bracket in the LaTeX_Code but it has to be escaped (i.e., \]).

```
<process optional point shape part>= (<-U)
          if (/^(symbol|radius|side)\s*/i) {
            my @previous_options = ();
            my $number_of_options = 1;
            my $symbol_set = 0;
            while (s/^(symbol|radius)\s*//i and $number_of_options <= 2) {
              my $option = lc($1);
              if (s/^=\s*//) {
                if (memberOf($option,@previous_options)) {
                  PrintErrorMessage("Option \"$option\" has been already defined",
$lc);
                  my $dummy = expr($lc);
                }
                elsif ($option eq "radius") {
                  $sh = expr($lc);
                  $sv = $defaultsymbol if ! $symbol_set;
                }
                elsif ($option eq "symbol") {
                  if (s/^circle\s*//i) {
                    $sv = "circle";
                    chk_lparen("after token circle",$lc);
                    $side_or_radius = expr($lc);
                    chk_rparen("expression",$lc);
                  }
                  elsif (s/^square\s*//i) {
                    $sv = "square";
                    chk_lparen("after token square",$lc);
                    $side_or_radius = expr($lc);
                    chk_rparen("expression",$lc);
                  }
                  elsif (s/^(((\\\]){1}|(\\,){1}|(\\\s){1}|[^\],\s])+)//) {
                    $sv = $1;
                    $sv =~ s/\\\]/\]/g;
                    $sv =~ s/\\,/,/g;
                    $sv =~ s/\\ / /g;
                    s/\s*//;
                  }
                  $symbol_set = 1;
                }
              }
              else {
                PrintErrorMessage("unexpected token", $lc);
                next LINE;
              }
              $number_of_options++;
              push (@previous_options, $option);
              s/^,\s*//;
            }
          }
```

```
  else {
    PrintErrorMessage("unexpected token", $lc);
    next LINE;
  }
```

The ArrowShape command has either one or three arguments. If the only argument of the command is the token default, then the parameters associated with the arrow shape resume their default values. Now, if there are three arguments, these are used to specify the shape of an arrow. The command actually sets the three global variables $arrowLength, $arrowAngleB and $arrowAngleC. Arguments whose value is equal to zero, do not affect the value of the corresponding global variables. To reset the values of the global variables one should use the commane with default as it only argument. The syntax of the command is as follows:

```
"ArrowShape" "(" expr [ units ] "," expr "," expr ")" or
             "ArrowShape" "(" "default" ")"
```

> Here units is any valid TeX unit (e.g., "mm", "cm", etc.). Note that if any of the three expressions is equal to zero, the default value is taken instead. As direct consequence, if the value of the first expression is zero, the units part is actually ignored.

<process ArrowShape command>= (<-U)

```
chk_lparen("$cmd",$lc);
if (s/^default//i) {
  $arrowLength = 2;
  $arrowLengthUnits = "mm";
  $arrowAngleB = 30;
  $arrowAngleC = 40;
}
else {
  my ($LocalArrowLength, $LocalArrowAngleB ,$LocalArrowAngleC) = (0,0,0);
  $LocalArrowLength = expr($lc);
  if (s/^\s*($units)//i) {
    $arrowLengthUnits = "$1";
  }
  else {
    $xunits =~ /(\d+(\.\d+)?)\s*($units)/;
    $LocalArrowLength *= $1;
    $arrowLengthUnits = "$3";
  }
  chk_comma($lc);
  $LocalArrowAngleB = expr($lc);
  chk_comma($lc);
  $LocalArrowAngleC = expr($lc);
  $arrowLength = ($LocalArrowLength == 0 ? 2  : $LocalArrowLength);
  $arrowLengthUnits = ($LocalArrowLength == 0 ? "mm" : $arrowLengthUnits);
  $arrowAngleB = ($LocalArrowAngleB == 0 ? 30 : $LocalArrowAngleB);
  $arrowAngleC = ($LocalArrowAngleC == 0 ? 40 : $LocalArrowAngleC);
}
chk_rparen("after $cmd arguments",$lc);
chk_comment("after $cmd command",$lc);
print OUT "%% arrowLength = $arrowLength$arrowLengthUnits, ",
          "arrowAngleB = $arrowAngleB ",
          "and arrowAngleC = $arrowAngleC\n" if $comments_on;
```

The PointSymbol command is used to set the point symbol and possibly its line-free radius. The point symbol can be either a LaTeX symbol or the word default which corresponds to the default point symbol, i.e., $\bullet$. The line-free radius can be an expression. Here is the complete syntax:

```
pointsymbol ::= "pointsymbol" ( symbol [ "," radius])
symbol      ::= "default" | circle | square | LaTeX_Code
circle      ::= "circle" "(" expression ")"
square      ::= "square" "(" expression ")"
radius      ::= expression
```

Note that the LaTeX_Code can contain the symbols \, and \) which are escape sequences for a comma and right parenthesis, respectively.

<process PointSymbol command>= **(<-U)**

```
chk_lparen("$cmd",$lc);
if (s/^default//i) #default point symbol
{
   $defaultsymbol = "\$\\bullet\$";
}
elsif (s/^(circle|square)//i) {
  $defaultsymbol = $1;
  chk_lparen($defaultsymbol, $lc);
  $GlobalDimOfPoints = expr($lc);
  chk_rparen("expression", $lc);
}
elsif (s/^(((\\,){1}|(\\\)){1}|(\\\s){1}|[^\),\s])+)//) #arbitrary LaTeX
point
{
   $defaultsymbol = $1;
   $defaultsymbol=~ s/\\\)/\)/g;
   $defaultsymbol=~ s/\\,/,/g;
   $defaultsymbol=~ s/\\ / /g;
}
else
{
   PrintErrorMessage("unrecognized point symbol",$lc);
}
if (s/\s*,\s*//) {
  $defaultLFradius = expr($lc);
}
chk_rparen("after $cmd arguments",$lc);
chk_comment("after $cmd command",$lc);
```

The system command provides a shell escape. However, we use a subroutine to check whether the argument of the command contains tainted data. If this is the case, then we simply ignore this command. The syntax of the command is as follows:

```
system-cmd ::= "system" "(" string ")"
```

where string is just a sequence of characters enclosed in quotation marks. We start by parsing a left parenthesis and then we get the command by calling the subroutine get_string. If there is an error we skip this command. Otherwise, we assign to the variable $_ what is left. Now we check if the variable $command contains any tainted data. If it doesn't, we execute the command, otherwise we print an error message and skip to the next input line. Next, we check for closing right parenthesis and a possible trailing comment.

<process system command>= **(<-U)**

```
chk_lparen("$cmd",$lc);
my ($error, $command, $rest) = get_string($_);
next LINE if $error == 1;
$_ = $rest;
if (! is_tainted($command)) {
```

```
      system($command);
   }
   else {
      PrintErrorMessage("String \"$command\" has tainted data", $lc);
      next LINE;
   }
   chk_rparen("after $cmd arguments",$lc);
   chk_comment("after $cmd command",$lc);
```

The text command is used to put a piece of text or a symbol on a particular point of the resulting graph. The syntax of the command is as follows:

```
text-comm ::= "text" "(" text ")" "{"coords"} "[" pos-code "]"
text ::= ascii string
coords ::= Coord "," Coord |
           Point-Name "," "shift" "(" Coord "," Coord ")" |
           Point-Name "," "polar" "(" Coord "," Coord  [angle-unit] ")"
Coord ::= decimal number | variable | pair-of-Point-Names
pair-of-Point-Names ::= Point-Name Point-Name
angle-unit ::= "deg" | "rad"
pos-code ::= lr-code [tb-code] | tb-code [lr-code]
lr-code ::= "l" | "r"
tb-code ::=  "t" | "b" | "B"
```

Initially, we parse the text. Since the text may contain parentheses we assume that the user enters pairs of matching parentheses. Note, that this is a flaw in the original design of the language, which may be remedied in future releases of the software. Then, we check the coords part. Next, if there is a left square bracket, we assume the user has specified the pos-code. We conclude by checking a possible trailing comment. The next thing we do is to generate the PiCTeX code. The two possible forms follow:

```
\put {TEXT} [POS] at Px Py
   \put {TEXT} at Px Py
```

<process text command>= (<-U)

```
chk_lparen("text",$lc);
my ($level,$text)=(1,"");
TEXTLOOP: while (1)
{
  $level++ if /^\(/;
  $level-- if /^\)/;
  s/^(.)//;
  last TEXTLOOP if $level==0;
  $text .= $1;
}
chk_lcb("text part",$lc);
my ($Px, $Py,$dummy,$pos);
$pos="";
s/\s*//;
<process coordinates part of text command>
chk_rcb("coordinates part of text command",$lc);
if (s/^\[//)
{
  s/\s*//;
  <process optional part of text command>
  s/\s*//;
  chk_rsb("optional part of text command",$lc);
}
chk_comment($lc);
if ($pos eq "")
{
```

```
            printf OUT "\\put {%s} at %f %f\n", $text, $Px, $Py;
         }
         else
         {
            printf OUT "\\put {%s} [%s] at %f %f\n", $text, $pos, $Px, $Py;
         }
```

In this section we define the code that handles the coordinates part of the text command. The code just implements the grammar given above. If the first token is a number, we assume this is the x-coordinate. If it is a variable, we assume its value is the x-coordinate. However, if it is a point name, we check whether the next token is another point name. In this case we compute the distance between the two points. In case we have a single point followed by a comma, we expect to have either a polar or a shift part, which we process the same we processed them in the point command.

<process coordinates part of text command>= (<-U)

```
  if (/^[^\W\d_]\d{0,3}\s*[^,\w]/) {
    my $Tcoord = get_point($lc);
    my ($x,$y,$pSV,$pS)=unpack("d3A*",$PointTable{$Tcoord});
    $Px = $x;
    $Py = $y;
  }
  elsif (/[^\W\d_]\d{0,3}\s*,\s*shift|polar/i) {
    s/^([^\W\d_]\d{0,3})//i;
    my $PointA = $1;
    if (exists($PointTable{lc($PointA)})) {
      s/\s*//;
      if (s/^,//) {
        s/\s*//;
        if (s/^shift(?=\W)//i) {
          <process shift case>
        }
        elsif (s/^polar(?=\W)//i) {
          <process polar case>
        }
      }
    }
    else {
      PrintErrorMessage("undefined point/var",$lc);
      next LINE;
    }
  }
  else {
    $Px = expr();
    chk_comma($lc);
    $Py = expr();
  }
```

In this section we process the optional part of the text command. The general rule is that we are allowed to have up to two options one from the characters l and r and one from the the characters B, b, and t. We first check whether the next character is letter, if it isn't we issue an error message and continue with the next input line. If it is a letter we check whether it belongs to one of the two groups and if it doesn't we issue an error message and continue with the next input line. If the next character belongs to first group, i.e., it is either l or r, we store this character into the variable $pos. Next, we check whether there is another letter. If it is a letter, we store it in the variable $np. Now we make sure that this character belongs to the

other group, i.e., it is either b, B, or t. In case it belongs to the other group, we append the value of $np to the string stored in the variable $pos. Otherwise we issue an error message and continue with the next input line. We work similarly for the other case. In order to check whether a character belongs to some group of characters, we use the user defined function memberOf.

<process optional part of text command>= (<-U)

```
if (s/^(\w{1})\s*//) {
  $pos .= $1;
  if (memberOf($pos, "l", "r")) {
    if (s/^(\w{1})\s*//) {
      my $np = $1;
      if (memberOf($np, "t", "b", "B")) {
        $pos .= $np;
      }
      else {
        if (memberOf($np, "l", "r")) {
           PrintErrorMessage("$np can't follow 'l' or 'r'", $lc);
        }
        else {
          PrintErrorMessage("$np is not a valid positioning option", $lc);
        }
        next LINE;
      }
    }
  }
  elsif (memberOf($pos, "t", "b", "B")) {
    if (s/^(\w{1})\s*//) {
      my $np = $1;
      if (memberOf($np, "l", "r")) {
        $pos .= $np;
      }
      else {
        if (memberOf($np, "t", "b", "B")) {
           PrintErrorMessage("$np can't follow 't', 'b', or 'B'", $lc);
        }
        else {
          PrintErrorMessage("$np is not a valid positioning option", $lc);
        }
        next LINE;
      }
    }
  }
  else {
    PrintErrorMessage("$pos is not a valid positioning option", $lc);
    next LINE;
  }
}
else {
  PrintErrorMessage("illegal token in optional part of text command",$lc);
  next LINE;
}
```

The const command is used to store values into a comma separated list of named constants. Constant names have the same format as point names, i.e., they start with a letter and are followed by up to two digits. The whole operation is performed by a do-while construct that checks that there is a valid constant name, a = sign, and an expression. The do-while construct terminates if the next token isn't a comma. Variable $Constname is used to store the initial variable name, while we store in variable $varname the lowercase version of the variable name. In addition, we make sure a constant is not redefined

(or else it wouldn't be a constant:-). The last thing we do is to check whether there is a trailing comment. In case there, we simply ignore itl; otherwise we print a warning message.

```
<process const command>= (<-U)
    do{
      s/\s*//;
      PrintErrorMessage("no identifier found after token const",$lc)
        if $_ !~ s/^([^\W\d_]\d{0,3})//i;
      my $Constname = $1;
      my $constname = lc($Constname);
      if (exists $ConstTable{$constname}) {
        PrintErrorMessage("Redefinition of constant $constname",$lc);
      }
      s/\s*//; #remove leading white space
      PrintErrorMessage("did not find expected = sign",$lc)
        if $_ !~ s/^[=]//i;
      my $val = expr($lc);
      $VarTable{$constname} = $val;
      $ConstTable{$constname} = 1;
      print OUT "%% $Constname = $val\n" if $comments_on;
    }while (s/^,//);
    chk_comment($lc);
    s/\s*//;
    if (/^[^%]/) {
      PrintWarningMessage("Trailing text is ignored",$lc);
    }
```

The var command is used to store values into a comma separated list of named variables. Variable names have the same format as point names, i.e., they start with a letter and are followed by up to two digits. The whole operation is performed by a do-while construct that checks that there is a valid variable name, a = sign, and an expression. The do-while construct terminates if the next token isn't a comma. The variable $Varname is used to store the initial variable name, while we store in the variable $varname the lowercase version of the variable name. The last thing we do is to check whether there is a trailing comment. In case there, we simply ignore itl; otherwise we print a warning message.

```
<process var command>= (<-U)
    do{
      s/\s*//;
      PrintErrorMessage("no identifier found after token var",$lc)
        if $_ !~ s/^([^\W\d_]\d{0,3})//i;
      my $Varname = $1;
      my $varname = lc($Varname);
      if (exists $ConstTable{$varname}) {
        PrintErrorMessage("Redefinition of constant $varname",$lc);
      }
      s/\s*//; #remove leading white space
      PrintErrorMessage("did not find expected = sign",$lc)
        if $_ !~ s/^[=]//i;
      my $val = expr($lc);
      $VarTable{$varname} = $val;
      print OUT "%% $Varname = $val\n" if $comments_on;
    }while (s/^,//);
    chk_comment($lc);
    s/\s*//;
    if (/^[^%]/) {
      PrintWarningMessage("Trailing text is ignored",$lc);
    }
```

- <*>: D1

- <subroutine chk_comment >: U1, D2
- <subroutine chk_lcb >: U1, D2
- <subroutine chk_lparen >: U1, D2
- <subroutine chk_lsb >: U1, D2
- <subroutine chk_rcb >: U1, D2
- <subroutine chk_rparen >: U1, D2
- <subroutine chk_rsb >: U1, D2
- <subroutine circumCircleCenter >: U1, D2
- <subroutine ComputeDist >: U1, D2
- <subroutine drawAngleArc >: U1, D2
- <subroutine drawAngleArrow >: U1, D2
- <subroutine drawarrows >: U1, D2
- <subroutine drawCurve >: U1, D2
- <subroutine DrawLineOrArrow >: U1, D2
- <subroutine drawlines >: U1, D2
- <subroutine drawpoints >: U1, D2
- <subroutine drawsquare >: U1, D2
- <subroutine excircle >: U1, D2
- <subroutine expr >: U1, D2
- <subroutine get_point >: U1, D2
- <subroutine get_string >: U1, D2
- <subroutine IncircleCenter >: U1, D2
- <subroutine intersection4points >: U1, D2
- <subroutine is_tainted >: U1, D2
- <subroutine Length >: U1, D2
- <subroutine memberOf >: U1, D2
- <subroutine mpp >: U1, D2
- <subroutine noOfDigits >: U1, D2
- <subroutine perpendicular >: U1, D2
- <subroutine pointOnLine >: U1, D2
- <subroutine print_headers >: U1, D2
- <subroutine PrintErrorMessage >: U1, D2
- <subroutine PrintFatalError >: U1, D2
- <subroutine PrintWarningMessage >: U1, D2
- <subroutine process_input >: U1, D2
- <subroutine setLineThickness >: U1, D2
- <subroutine sp2X >: U1, D2
- <subroutine tand >: U1, D2
- <subroutine triangleArea >: U1, D2
- <subroutine X2sp >: U1, D2
- <subroutine definitions>: U1, D2