The Australian National University

Computer Science Technical Report

December 1993 Technical Report TR-CS-93-13

SHYSTER: The Program

James Popple

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Physical Sciences and Engineering

Joint Computer Science Technical Report Series

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Science Laboratory, Research School of Physical Sciences and Engineering, The Australian National University.

Direct correspondence regarding this series to:

The Secretary
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
AUSTRALIA

or send e-mail to

techreports@cs.anu.edu.au [internet]

Recent titles in this series:

TR-CS-93-12 B. B. Zhou and R. P. Brent, "Parallel Implementation of QRD Algorithms on the Fujitsu AP1000", November 1993.

TR-CS-93-08 B. Broom, "User-Mode Per-Process Name Spaces for the AP1000 File System", September 1993.

TR-CS-93-07 B. P. Molinari, "Does Computing Education need Mathemathics?", May 1993.

TR-CS-93-06 B. B. Zhou, Richard P. Brent, and A. Tridgell, "Efficient Implementation of Sorting Algorithms on Asynchronous MIMD Machines", May 1993.

TR-CS-93-05 B. B. Zhou and Richard P. Brent, "Parallel Computation of the Singular Value Decomposition on Tree Architectures", May 1993.

TR-CS-93-04 Richard P. Brent, "Fast Normal Random Number Generators for Vector Processors", March 1993.

TR-CS-93-03 J. M. Robson, "On Linear Programming, Graph Isomorphism and NP-Complete Problems", March 1993.

TR-CS-93-01 Andrew Tridgell and Richard P. Brent, "An Implementation of a General-Purpose Parallel Sorting Algorithm", February 1993.

SHYSTER: The Program

JAMES POPPLE

Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University

Technical Report TR-CS-93-13

December 1993
(revised April 1995)

This research was supported by an
Australian National University PhD Scholarship
funded by the Centre for Information Science Research

Contents

Introduction

Most legal expert systems attempt to implement complex models of legal reasoning. Yet the utility of a legal expert system lies not in the extent to which it simulates a lawyer's approach to a legal problem, but in the quality of its predictions and of its arguments. A complex model of legal reasoning is not necessary: a successful legal expert system can be based upon a simplified model of legal reasoning.

Some researchers have based their systems upon a jurisprudential approach to the law, yet lawyers are patently able to operate without any jurisprudential insight. A useful legal expert system should be capable of producing advice similar to that which one might get from a lawyer, so it should operate at the same pragmatic level of abstraction as does a lawyer—not at the more philosophical level of jurisprudence.

A legal expert system called SHYSTER has been developed to demonstrate that a useful legal expert system can be based upon a pragmatic approach to the law. SHYSTER has a simple representation structure which simplifies the problem of knowledge acquisition. Yet this structure is complex enough for SHYSTER to produce useful advice.

SHYSTER is a case-based legal expert system (although it has been designed so that it can be linked with a rule-based system to form a hybrid legal expert system). Its advice is based upon an examination of, and an argument about, the similarities and differences between cases. SHYSTER attempts to model the way in which lawyers argue with cases, but it does not attempt to model the way in which lawyers decide which cases to use in those arguments. Instead, it employs statistical techniques to quantify the similarity between cases. It decides which cases to use in argument, and what prediction it will make, on the basis of that similarity measure.

Module	*lines of code* `.h`	`.c`	*Total*
Shyster	64	347	411
Statutes	19	40	59
Cases	274	639	913
Tokenizer	56	385	441
Parser	4	947	951
Dumper	20	784	804
Checker	6	268	274
Scales	13	284	297
Adjuster	7	227	234
Consultant	19	260	279
Odometer	14	830	844
Reporter	12	1523	1535
Total	508	6534	7042

Figure 1: Lines of C code in SHYSTER, by module. Each module comprises two files: a definition (`.h`) file and an implementation (`.c`) file.

SHYSTER is of a general design: it provides advice in areas of case law that have been specified by a legal expert using a specification language, indicating the cases and attributes of importance in those areas. Four different, and disparate, areas of law have been specified for SHYSTER, and its operation has been tested in each of those legal domains.

Testing of SHYSTER in these four domains indicates that it is exceptionally good at predicting results, and fairly good at choosing cases with which to construct its arguments. SHYSTER demonstrates the viability of a pragmatic approach to legal expert system design.

* * *

SHYSTER is implemented using a dozen modules, written in ISO C.[1] (A breakdown, by module, of the number of lines of C code in SHYSTER is given in figure 1.) This report provides complete code listings of all twelve modules. This code, and the case law specifications used to test SHYSTER, are available on the worldwide web.[2] Full details of the design, implementation, operation and testing of SHYSTER are given elsewhere.[3]

`->`	structure/union pointer	$\rightarrow$
`*`	indirection	$*$
`*`	multiplication	$\times$
`==`	equal to	$\equiv$
`!=`	not equal to	$\neq$
`<=`	less than or equal to	$\leq$
`>=`	greater than or equal to	$\geq$
`!`	logical negation	$\neg$
`&&`	logical AND	$\wedge$
`||`	logical OR	$\vee$

Figure 2: Special symbols used in the listings of C code that appear in this report. The operators on the left are represented by the symbols on the right.

The SHYSTER module (§1) is the top-level module for the whole system. The STATUTES module (§2) is the top-level module for a rule-based system, presently unimplemented. The CASES module (§3) is the top-level module for the case-based system. The TOKENIZER and PARSER modules (§4 and §5) tokenize and parse a program written in SHYSTER's case law specification language. The DUMPER module (§6) displays the information that has been parsed. The CHECKER module (§7) checks for evidence of dependence between the attributes. The SCALES module (§8) determines the weight of each attribute. The ADJUSTER module (§9) allows the legal expert to adjust the weights of the attributes. The CONSULTANT module (§10) interrogates the user as to the attribute values in the instant case. The ODOMETER module (§11) determines the distances between the leading cases and the instant case, and the REPORTER module (§12) writes SHYSTER's legal opinion.

* * *

The format used for the display of C code in this report is based on that of the CWEB system.[4] Reserved words and preprocessor commands are set in boldface type. Identifiers are set in italics. String constants and character constants are set in a typewriter font, with "␣" representing a space. Some operators are represented by special symbols, as explained in figure 2.[5]

SHYSTER's external identifiers are made up of upper- and lower-case letters. Static identifiers consist only of lower-case letters. Identifiers in all upper-case are enumerated identifiers and other constants.

An index to all of the identifiers which appear in the code listings, and which are not reserved words or preprocessor commands, is at the end of this report.

* * *

Much of SHYSTER's output is in LaTeX format: i.e. it is suitable for processing by the LaTeX document processor.[6] This contributes to SHYSTER's portability, as LaTeX is widely available on many platforms. Using LaTeX simplifies the footnoting of text (`\footnote{...}`), allows some data to be displayed in a clear and economical tabular format (`\begin{tabular}...`), and ensures the aesthetic quality of the output.

[1]International standard ISO/IEC 9899: 1990; Australian standard AS 3955–1991. Kernighan and Ritchie 1988 describe ANSI C which is the same as ISO C.

[2]`http://cs.anu.edu.au/software/shyster`

[3]Popple 1993, 1996.

[4]Knuth and Levy 1994. CWEB is a version of Knuth's WEB system (1986a, 1986b) adapted to C. The CWEB system of structured documentation was not used in the development of SHYSTER, but CWEB's approach to "pretty-printing" C code has been adopted for this report. This report was prepared using the LaTeX document processor; LaTeX code was generated from SHYSTER's C code using a preprocessor constructed by the author.

[5]All of the special symbols in figure 2 are used by CWEB, except for $\times$: CWEB uses $*$ for both indirection and multiplication.

[6]Lamport 1986 describes LaTeX which is a set of macros for Knuth's TeX system (1984, 1986a). SHYSTER's LaTeX output is suitable for processing by LaTeX version 2.09 ⟨25 March 1992⟩ and TeX version 3.141. It can also be processed, in "compatibility mode," by LaTeX 2_ε which is described by Lamport 1994.

1

The SHYSTER module

shyster.h

/* This is the header file for the SHYSTER module. It is included by all twelve modules. */

/* version and copyright information */

#**define** *Shyster_Version* "SHYSTER␣version␣1.0"
#**define** *Copyright_Message* "Copyright␣James␣Popple␣1993"

/* a string which is written to *stderr* if SHYSTER is invoked without arguments */

```
#define usage_string \
   "usage:\t" \
   "shyster␣[␣-a␣]␣[␣-c␣filename␣]␣[␣-d␣filename␣]␣[␣-D␣filename␣]␣[␣-e␣]\n" \
   "\t\t[␣-h␣number␣number␣]␣[␣-i␣]␣[␣-l␣filename␣]␣[␣-p␣filename␣]\n" \
   "\t\t[␣-q␣]␣[␣-r␣filename␣]␣[␣-w␣filename␣]\n"
```

/* the versions of LaTeX and TeX for which SHYSTER has been developed */

#**define** *LaTeX_Version* "LaTeX␣version␣2.09␣<25␣March␣1992>"
#**define** *TeX_Version* "TeX␣version␣3.141"

/* maxima */

#**define** *Max_Filename_Length* 256
#**define** *Max_Error_Message_Length* 256
#**define** *Max_LaTeX_Line_Width* 64

/* other constants */

#**define** *Empty_String* ""
#**define** *Null_Character* '\0'
#**define** *Space_Character* '␣'
#**define** *Carriage_Return_Character* '\n'
#**define** *Top_Level* 0
#**define** *No_Hang* 0
#**define** *Hang* 1

```
/* functions whose returned values are always ignored */

#define fprintf (void) fprintf
#define free (void) free
#define gets (void) gets
#define sprintf (void) sprintf

/* simple types */

typedef unsigned int cardinal;
typedef float floating_point;
typedef char *string;
typedef FILE *file;

/* enumerated type */

typedef enum {
    FALSE,
    TRUE
} boolean;

/* external functions */

extern void
Indent(
        file stream,
        cardinal level);

extern void
Write(
        file stream,
        string write_string,
        const string suffix,
        const cardinal level,
        const boolean hanging_indent);

extern void
Write_Error_Message_And_Exit(
        file stream,
        const string module_name,
        string message);

extern void
Write_Warning_Message(
        file stream,
        const string module_name,
        string message,
        const cardinal level);

extern void
Write_LaTeX_Header(
        file stream,
        boolean inputable_latex);

extern void
Write_LaTeX_Trailer(
        file stream,
        boolean inputable_latex);
```

extern *boolean*
Is_Digit(
    int *ch*);

shyster.c

/* This is the implementation file for the SHYSTER module. */

```
#include ⟨stdio.h⟩
#include ⟨stdlib.h⟩
#include "shyster.h"
#include "cases.h"
#include "statutes.h"
```

extern void
Indent(
    file stream,
    cardinal level)

/* Writes the equivalent of $4 \times$ *level* spaces (1 tab = 8 spaces). */

```
{
    while (level > 1) {
        fprintf(stream, "\t");
        level -= 2;
    }
    if (level ≡ 1)
        fprintf(stream, "␣␣␣␣");
}
```

static void
write_line(
    file stream,
    *string *write_string*,
    const *cardinal level*,
    const *boolean hanging_indent*,
    cardinal count)

/* Writes a line of characters from **write_string*. Writes characters up to the last space in the next *count* characters, then breaks the line and indents the next line by $4 \times$ *level* spaces (plus an extra two spaces if *hanging_indent* is *TRUE*). Leaves **write_string* pointing to the character after the space at which the line was broken. */

```
{
    /* find the last space that will fit on the line */
    while ((*(*write_string + count) ≠ Space_Character) ∧ (count ≠ 0))
        count--;
```

```
    if (count ≡ 0) {

        /* there is no convenient place to break this line, so write as much as will fit, with a % character at the end of the line, and don't indent the next line */

        for (count = Max_LaTeX_Line_Width − level × 4 − 1; count ≠ 0; count−−)
            fprintf(stream, "%c", *((*write_string)++));
        fprintf(stream, "%%\n");
    } else {

        /* there is (at least) one space in this line, so write each of the characters up to the last space, break the line, and indent the next line */

        while (count ≠ 0) {
            fprintf(stream, "%c", *((*write_string)++));
            count−−;
        }
        (*write_string)++;
        fprintf(stream, "\n");
        Indent(stream, level);
        if (hanging_indent)
            fprintf(stream, "␣␣");
    }
}

extern void
Write(
        file stream,
        string write_string,
        const string suffix,
        const cardinal level,
        const boolean hanging_indent)

/* Writes write_string plus suffix. Breaks lines (at spaces) so that they are no longer than Max_LaTeX_Line_Width characters. Indents lines by 4 × level spaces. Indents lines after the first by a further two spaces, if hanging_indent is TRUE. (Assumes that suffix is less than Max_LaTeX_Line_Width − level × 4 characters long.) */

{
    cardinal count = 0,
        suffix_length = 0,
        line_length = Max_LaTeX_Line_Width − level × 4;
boolean hanged = FALSE;

if (write_string ≡ NULL)

    /* there is no string to write */

    fprintf(stream, "%s", Null_String);
```

```
else {
    /* there is a string to write */
    if (suffix ≠ Null_String)
        /* there is a suffix, so set suffix_length to the number of characters in the suffix, up
           to (but not including) the first carriage return (if there is one) */
        while ((*(suffix + suffix_length) ≠ Null_Character) ∧
               (*(suffix + suffix_length) ≠ Carriage_Return_Character))
            suffix_length++;
    Indent(stream, level);
    while (*(write_string + count) ≠ Null_Character) {
        if (count ≡ line_length) {
            /* there are more characters left in the string than will fit on this line, so write
               as much as will fit */
            write_line(stream, &write_string, level, hanging_indent, count);
            if (¬hanged ∧ hanging_indent) {
                line_length −= 2;
                hanged = TRUE;
            }
            count = 0;
        }
        count++;
    }
    /* the rest of the string will fit on a line */
    if (count + suffix_length > line_length) {
        /* ... but the rest of the string plus the suffix will be too long to fit on a line, so
           write as much as will fit */
        write_line(stream, &write_string, level, hanging_indent, count);
        /* the rest of the string plus the suffix will fit on the new line; set count to be the
           number of characters still to be written */
        count = 0;
        while (*(write_string + count) ≠ Null_Character)
            count++;
    }
    /* write the rest of the string */
    while (count ≠ 0) {
        fprintf(stream, "%c", *write_string++);
        count--;
    }
    /* write the suffix */
    fprintf(stream, "%s\n", suffix);
}
```

```
extern void
Write_Error_Message_And_Exit(
        file stream,
        const string module_name,
        string message)

/* Writes "ERROR (module_name): message." to stderr and to stream. Exits with a value of
   EXIT_FAILURE (defined in stdlib.h). */

{
    static char full_message[Max_Error_Message_Length];

    sprintf(full_message, "ERROR␣(%s):␣%s", module_name, message);

    Write(stderr, full_message, ".\n", Top_Level, Hang);

    /* write to stream only if stream ≠ stdout (the error message has already been written to
       stderr) */

    if ((stream ≠ NULL) ∧ (stream ≠ stdout))
        Write(stream, full_message, ".\n", Top_Level, Hang);

    exit(EXIT_FAILURE);
}

extern void
Write_Warning_Message(
        file stream,
        const string module_name,
        string message,
        const cardinal level)

/* Writes "WARNING (module_name): message." to stderr and to stream. */

{
    static char full_message[Max_Error_Message_Length];

    if (stream ≠ NULL) {

        sprintf(full_message, "WARNING␣(%s):␣%s", module_name, message);

        Write(stderr, full_message, ".\n", Top_Level, Hang);

        /* write to stream only if stream ≠ stdout (the warning message has already been
           written to stderr) */

        if (stream ≠ stdout)
            Write(stream, full_message, ".\n", level, Hang);
    }
}
```

extern void
Write_LaTeX_Header(
file stream,
boolean inputable_latex)

/* Writes LaTeX code to go at the start of a LaTeX document. Writes code that can be included in another LaTeX document (i.e. not stand-alone code), if *inputable_latex* is *TRUE*. */

{
fprintf(*stream*, "%% Produced by %s\n\n"
"%% %s\n\n", *Shyster_Version*, *Copyright_Message*);
if (¬*inputable_latex*)
fprintf(*stream*, "%% This is a stand-alone LaTeX file.\n");
else
fprintf(*stream*, "%% This is not a stand-alone LaTeX file.\n"
"%% Include it in a LaTeX document using the \\input command.\n");
fprintf(*stream*, "%% Use %s and %s.\n\n", *LaTeX_Version*, *TeX_Version*);
if (¬*inputable_latex*)
fprintf(*stream*, "\\documentstyle[12pt]{article}\n"
"\\oddsidemargin=-5.4mm\n"
"\\evensidemargin=-5.4mm\n"
"\\topmargin=-5.4mm\n"
"\\headheight=0mm\n"
"\\headsep=0mm\n"
"\\textheight=247mm\n"
"\\textwidth=170mm\n"
"\\footskip=15mm\n"
"\\pagestyle{plain}\n\n"
"\\begin{document}\n\n");
}

extern void
Write_LaTeX_Trailer(
file stream,
boolean inputable_latex)

/* Writes LaTeX code to go at the end of a LaTeX document. Writes code that can be included in another LaTeX document (i.e. not stand-alone code), if *inputable_latex* is *TRUE*. */

{
if (¬*inputable_latex*)
fprintf(*stream*, "\\end{document}\n");
}

extern *boolean*
Is_Digit(
int *ch*)

/* Returns *TRUE*, iff *ch* is a digit (0 ... 9). */

{
return ((*ch* ≥ *Zero_Character*) ∧ (*ch* ≤ *Nine_Character*));
}

```
static void
error_exit(
        const string message)
{
    Write_Error_Message_And_Exit(NULL, "Shyster", message);
}

static void
parse_arguments(
        int argc,
        string *argv,
        boolean *adjust,
        boolean *echo,
        cardinal *hypothetical_reports,
        cardinal *hypothetical_changes,
        boolean *inputable_latex,
        boolean *verbose,
        string *specification_filename,
        string *distances_filename,
        string *log_filename,
        string *probabilities_filename,
        string *report_filename,
        string *dump_filename,
        string *weights_filename)

/* Parses the UNIX command line arguments (argv[1] ... argv[argc - 1]) and stores the in-
    formation in the variables pointed to by the 13 other parameters. */
{
    string argument;
    char message[Max_Error_Message_Length];

    if (argc < 2) {

        /* no argument was provided, so write usage_string to stderr and exit with a value of
            EXIT_FAILURE (defined in stdlib.h) */

        fprintf(stderr, usage_string);
        exit(EXIT_FAILURE);
    }

    /* skip over the first argument (the name by which SHYSTER was invoked) */

    argc--;
    argv++;

    /* while there are still arguments to parse ... */

    while (argc > 0) {

        argument = *argv;

        if (*argument++ ≠ '-')

            /* this argument is not a switch */

            break;
```

```
switch (*argument++) {
    case 'a':
        /* -a: enable weight adjustment */
        *adjust = TRUE;
        break;
    case 'c':
        /* -c specification: read the case law specification from "specification.cls" */
        if (argc > 1) {
            argc--;
            argv++;
            *specification_filename = *argv;
        } else
            error_exit("must␣supply␣a␣filename␣with␣-c");
        break;
    case 'd':
        /* -d distances: write distances to "distances-area.tex" */
        if (argc > 1) {
            argc--;
            argv++;
            *distances_filename = *argv;
        } else
            error_exit("must␣supply␣a␣filename␣with␣-d");
        break;
    case 'D':
        /* -D dump: write dump to "dump.tex" */
        if (argc > 1) {
            argc--;
            argv++;
            *dump_filename = *argv;
        } else
            error_exit("must␣supply␣a␣filename␣with␣-D");
        break;
    case 'e':
        /* -e: enable echo mode */
        *echo = TRUE;
        break;
```

```
case 'h':

    /* -h r c: hypothesize, reporting on r hypotheticals per result with a limit of c
       changes */

    if (argc > 2) {
        argc--;
        argv++;
    while (**argv ≠ Null_Character) {
        if (¬Is_Digit(**argv))
            error_exit("argument to -h must be two numbers");
        *hypothetical_reports = (10 × *hypothetical_reports) +
                (cardinal) **argv − (cardinal) Zero_Character;
        (*argv)++;
    }
    argc--;
    argv++;
    *hypothetical_changes = 0;
    while (**argv ≠ Null_Character) {
        if (¬Is_Digit(**argv))
            error_exit("argument to -h must be two numbers");
        *hypothetical_changes = (10 × *hypothetical_changes) +
                (cardinal) **argv − (cardinal) Zero_Character;
        (*argv)++;
        }
    } else
        error_exit("must supply two numbers with -h");
    break;

case 'i':

    /* -i: write LaTeX code that can be included in another LaTeX document
       (i.e. not stand-alone code) */

    *inputable_latex = TRUE;
    break;

case 'l':

    /* -l log: write log to "log.log" */

    if (argc > 1) {
        argc--;
        argv++;
        *log_filename = *argv;
    } else
        error_exit("must supply a filename with -l");
    break;
```

```
case 'p':
    /* -p probabilities: write probabilities to "probabilities.tex" */
    if (argc > 1) {
        argc--;
        argv++;
        *probabilities_filename = *argv;
    } else
        error_exit("must supply a filename with -p");
    break;
case 'q':
    /* -q: enable quiet mode (don't summarize cases, etc.) */
    *verbose = FALSE;
    break;
case 'r':
    /* -r report: write report to "report-area.tex" */
    if (argc > 1) {
        argc--;
        argv++;
        *report_filename = *argv;
    } else
        error_exit("must supply a filename with -r");
    break;
case 'w':
    /* -w weights: write weights to "weights.tex" */
    if (argc > 1) {
        argc--;
        argv++;
        *weights_filename = *argv;
    } else
        error_exit("must supply a filename with -w");
    break;
default:
            sprintf(message, "unrecognized option -%s", argument - 1);
            error_exit(message);
            break;
        }
        argc--;
        argv++;
    }
}
```

```
extern int
main(
        int argc,
        string *argv)
/* Extracts the options and arguments from the UNIX command line, initializes the rule-based
    system and the case-based system, then invokes the rule-based system. */
{
char filename[Max_Filename_Length],
    message[Max_Error_Message_Length];
statute_law_specification statute_law;
case_law_specification case_law;
file log_stream;
boolean adjust = FALSE,
    echo = FALSE,
    inputable_latex = FALSE,
    verbose = TRUE;
cardinal hypothetical_reports = 0,
    hypothetical_changes;
string specification_filename = NULL,
    distances_filename = NULL,
    log_filename = NULL,
    probabilities_filename = NULL,
    report_filename = NULL,
    dump_filename = NULL,
    weights_filename = NULL;
/* extract the options and arguments from the UNIX command line */
parse_arguments(argc, argv, &adjust, &echo, &hypothetical_reports, &hypothetical_changes,
        &inputable_latex, &verbose, &specification_filename, &distances_filename,
        &log_filename, &probabilities_filename, &report_filename, &dump_filename,
        &weights_filename);
/* write version and copyright information to stdout */
fprintf(stdout, "%s\n\n%s\n\n", Shyster_Version, Copyright_Message);
if (log_filename ≡ NULL)
    /* no log filename was specified, so log information will be written to stdout */
    log_stream = stdout;
else {
    /* open the log file */
    sprintf(filename, "%s%s", log_filename, Log_File_Extension);
    if ((log_stream = fopen(filename, "w")) ≡ NULL) {
        sprintf(message, "can't␣open␣log␣file␣\"%s\"", filename);
        error_exit(message);
    }
    /* write version and copyright information to the log file */
    fprintf(log_stream, "%s\n\n"
            "%s\n\n", Shyster_Version, Copyright_Message);
}
```

```
/* initialize the rule-based system */
statute_law = Initialize_Statutes();
/* initialize the case-based system */
case_law = Initialize_Cases(log_stream, inputable_latex, verbose, specification_filename,
        dump_filename, probabilities_filename, weights_filename);
/* invoke the rule-based system */
Statute_Law(log_stream, statute_law, case_law, adjust, echo, inputable_latex, verbose,
        hypothetical_reports, hypothetical_changes, distances_filename, weights_filename,
        report_filename);
/* write "Finished." to the log file (if there is one) and to stdout */
if (log_filename ≠ NULL)
    fprintf(log_stream, "Finished.\n");
fprintf(stdout, "Finished.\n");
/* close the log file */
if (fclose(log_stream) ≡ EOF) {
    sprintf(message, "can't␣close␣log␣file␣\"%s\"", filename);
    error_exit(message);
}
/* everything worked, so exit with a value of EXIT_SUCCESS (defined in stdlib.h) */
return EXIT_SUCCESS;
}
```

2

The STATUTES module

statutes.h

/∗ This is the header file for the STATUTES module. It is also included by the SHYSTER module. ∗/

/∗ structure type ∗/

typedef struct {
    void ∗*dummy*;
} *statute_law_specification*;

/∗ external functions ∗/

extern *statute_law_specification*
Initialize_Statutes();

extern void
Statute_Law(
        file log_stream,
        statute_law_specification statute_law,
        case_law_specification case_law,
        boolean adjust,
        boolean echo,
        boolean inputable_latex,
        boolean verbose,
        cardinal hypothetical_reports,
        cardinal hypothetical_changes,
        string distances_filename,
        string weights_filename,
        string report_filename);

statutes.c

/* This is the implementation file for the STATUTES module. */

#include ⟨stdio.h⟩
#include "shyster.h"
#include "cases.h"
#include "statutes.h"

extern *statute_law_specification*
Initialize_Statutes()

/* Returns a pointer to a dummy structure. (If implemented, it would initialize the rule-based system, by reading a statute law specification, and return a pointer to SHYSTER's internal representation of that specification.) */

{
    statute_law_specification statute_law;

    statute_law.dummy = *NULL*;
    return *statute_law*;
}

extern void
Statute_Law(
        file log_stream,
        statute_law_specification statute_law,
        case_law_specification case_law,
        boolean adjust,
        boolean echo,
        boolean inputable_latex,
        boolean verbose,
        cardinal hypothetical_reports,
        cardinal hypothetical_changes,
        string distances_filename,
        string weights_filename,
        string report_filename)

/* Prompts the user for an identifier, then invokes the case-based system seeking advice in the area corresponding to that identifier. The result that the case-based system returns is written to *log_stream*. */

{
    char *area_identifier*[*Max_Identifier_Length*];
string result_identifier;

/* prompt the user for a case law area identifier */

fprintf(*stdout*, "Case␣law␣area␣identifier:␣");
gets(*area_identifier*);

fprintf(*log_stream*, "Case-based␣system␣called␣with␣area␣identifier␣\"%s\".\n\n",
        area_identifier);

```
/* invoke the case-based system */

result_identifier = Case_Law(log_stream, case_law, area_identifier, adjust, echo,
        inputable_latex, verbose, hypothetical_reports, hypothetical_changes, Top_Level,
        distances_filename, weights_filename, report_filename);

/* write the result identifier to the log file */

if (result_identifier ≠ NULL)
    fprintf(log_stream,
        "Case-based system returned result identifier \"%s\".\n\n",
        result_identifier);
}
```

3

The CASES module

cases.h

/* This is the header file for the CASES module. It is included by all twelve modules. */

/* maxima */

#**define** *Max_Identifier_Length* 16
#**define** *Max_Attribute_Options* 4

/* the number of characters first allocated for a string, and the number of extra characters allocated if that string has to be extended */

#**define** *String_Increment* 256

/* a "pseudo-infinite" weight for the calculation of weighted correlation coefficients */

#**define** *Very_Heavy_Indeed* 1000000.0

/* the threshold of likelihood, below which a given number of YES/YES pairs is considered unusually high or unusually low */

#**define** *Threshold* 0.05

/* arithmetic is precise to (log *Precision*) decimal places */

#**define** *Precision* 100.0

/* distance and weight comparisons are precise to (log *Distance_Precision*) decimal places (this is the threshold within which two cases are considered equidistant, or two weights are considered equal) */

#**define** *Distance_Precision* 100.0

/* the format for the display of floating point numbers (the number of decimal places should be (log *Distance_Precision*)) */

#**define** *Floating_Point_Format* `"%.2f"`

/* string constants for output to LaTeX files */

#**define** *Raise_Height* "0.6\\ht\\strutbox"
#**define** *Column_Separation* "\\tabcolsep"
#**define** *Matrix_Column_Separation* "0.4em"
#**define** *Heading* "\\subsection*"
#**define** *Subheading* "\\subsubsection*"
#**define** *Skip* "\\medskip\\noindent"
#**define** *Identifier_Font* "\\sf"
#**define** *Null_String* "{[{\\it␣null␣string\\/}]}"

/* special LaTeX symbols:

- • *Yes_Symbol*
- × *No_Symbol*
  Unknown_Symbol
- ■ *Functional_Dependence_Symbol*
- • *Stochastic_Dependence_Symbol*
- ⇒ *Specified_Direction_Symbol*
- $\overset{I}{\Rightarrow}$ *Ideal_Point_Direction_Symbol*
- $\overset{\mu}{\Rightarrow}$ *Centroid_Direction_Symbol*
- $\overset{\star}{\Rightarrow}$ *All_Directions_Symbol*
- ⇔ *External_Area_Symbol*
- ⇐ *External_Result_Symbol*
- ∨ *Disjunction_Symbol* */

#**define** *Yes_Symbol* "$\\bullet$"
#**define** *No_Symbol* "$\\times$"
#**define** *Unknown_Symbol* ""
#**define** *Functional_Dependence_Symbol* "\\rule[0.25ex]{0.35em}{0.35em}"
#**define** *Stochastic_Dependence_Symbol* "$\\bullet$"
#**define** *Specified_Direction_Symbol* "$\\Rightarrow$"
#**define** *Ideal_Point_Direction_Symbol* \
"$\\stackrel{\\scriptscriptstyle␣I~}{\\Rightarrow}$"
#**define** *Centroid_Direction_Symbol* \
"$\\stackrel{\\scriptscriptstyle\\mu~}{\\Rightarrow}$"
#**define** *All_Directions_Symbol* \
"$\\stackrel{\\scriptscriptstyle\\star~}{\\Rightarrow}$"
#**define** *External_Area_Symbol* "$\\Leftrightarrow$"
#**define** *External_Result_Symbol* "$\\Leftarrow$"
#**define** *Disjunction_Symbol* "$\\vee$"

/* file extensions */

#**define** *LaTeX_File_Extension* ".tex"
#**define** *Log_File_Extension* ".log"
#**define** *Specification_File_Extension* ".cls"

```
/* character constants */

#define Attribute_Vector_Begin_Character '('
#define Attribute_Vector_End_Character ')'
#define Little_A_Character 'a'
#define Little_Z_Character 'z'
#define Big_A_Character 'A'
#define Big_Z_Character 'Z'
#define Zero_Character '0'
#define Nine_Character '9'
#define Yes_Character 'Y'
#define No_Character 'N'
#define Unknown_Character 'U'
#define Help_Character 'H'
#define Quit_Character 'Q'

/* other constants */

#define Year_Digits 4
#define Yes_Value 1.0
#define No_Value 0.0

/* enumerated types */

typedef enum {
    NO,
    YES,
    UNKNOWN
} attribute_value_type;

typedef enum {
    NEARER,
    EQUIDISTANT,
    FURTHER
} relative_distance_type;

/* structure types */

typedef struct {
    boolean infinite;
    floating_point finite;
} weight_type;

typedef struct {
    cardinal infinite;
    floating_point finite;
} distance_subtype;

typedef struct {
    distance_subtype known;
    distance_subtype unknown;
} distance_type;

typedef struct {
    boolean meaningless;
    floating_point unweighted;
    floating_point weighted;
} correlation_type;
```

```
typedef struct {
    distance_type distance;
    cardinal number_of_known_differences,
        number_of_known_pairs;
    floating_point weighted_association_coefficient;
    correlation_type correlation_coefficient;
} metrics_type;

typedef struct vector_element {
    attribute_value_type attribute_value;
    struct vector_element *next;
} vector_element;

typedef struct matrix_element {
    attribute_value_type attribute_value;
    struct matrix_element *case_next,
      *attribute_next;
} matrix_element;

typedef struct centroid_element {
    boolean unknown;
    floating_point value;
    struct centroid_element *next;
} centroid_element;

typedef struct probability_element {
    boolean unknown,
        functional_dependence;
    floating_point probability_that_or_fewer,
        probability_that_or_more;
    struct probability_element *next;
} probability_element;

typedef struct identifier_list_element {
    string identifier;
    struct identifier_list_element *next;
} identifier_list_element;

typedef struct weight_list_element {
    weight_type weight;
    struct weight_list_element *next;
} weight_list_element;

typedef struct hypothetical_list_element {
    vector_element *hypothetical_head;
    distance_type nearest_neighbour_distance;
    struct hypothetical_list_element *next;
} hypothetical_list_element;
```

```
/* the structure type kase is so-named because the more obvious case is a reserved word */
typedef struct kase {
    cardinal number;
    string name;
    string short_name;
    string citation;
    cardinal year;
    string court_string;
    cardinal court_rank;
    matrix_element *matrix_head;
    string summary;
    boolean summarized;
    metrics_type metrics;
    struct kase *equidistant_known_next,
      *equidistant_unknown_next,
      *next;
} kase;

typedef struct result {
    string identifier,
        string;
    kase *case_head,
      *nearest_known_case,
      *nearest_unknown_case;
    relative_distance_type nearest_known_compared_with_unknown;
    distance_subtype specified_direction;
    struct result *equidistant_specified_direction_next;
    vector_element *ideal_point_head;
    metrics_type ideal_point_metrics;
    struct result *equidistant_ideal_point_next;
    distance_subtype ideal_point_direction;
    struct result *equidistant_ideal_point_direction_next;
    centroid_element *centroid_head;
    metrics_type centroid_metrics;
    struct result *equidistant_centroid_next;
    distance_subtype centroid_direction;
    struct result *equidistant_centroid_direction_next;
    hypothetical_list_element *hypothetical_list_head;
    struct result *equidistant_next,
      *next;
} result;

typedef struct direction_list_element {
    result *result;
    struct direction_list_element *next;
} direction_list_element;

typedef struct local_attribute_type {
    string question,
        help;
} local_attribute_type;
```

```
typedef struct external_attribute_type {
    string area_identifier;
    identifier_list_element *yes_identifier_head,
      *no_identifier_head,
      *unknown_identifier_head;
} external_attribute_type;

typedef struct attribute {
    cardinal number;
    boolean external_attribute;
    union {
        local_attribute_type local;
        external_attribute_type external;
    } details;
    string yes;
    direction_list_element *yes_direction_head;
    string no;
    direction_list_element *no_direction_head;
    string unknown;
    direction_list_element *unknown_direction_head;
    matrix_element *matrix_head;
    floating_point mean;
    weight_type weight;
    weight_list_element *weights_head;
    probability_element *probability_head;
    struct attribute *next;
} attribute;

typedef struct area {
    string identifier,
        opening,
        closing;
    result *result_head;
    cardinal number_of_results;
    attribute *attribute_head;
    cardinal number_of_attributes;
    boolean infinite_weight;
    boolean correlation_coefficients;
    result *nearest_result;
    result *nearest_ideal_point;
    result *nearest_centroid;
    result *strongest_specified_direction;
    result *strongest_ideal_point_direction;
    result *strongest_centroid_direction;
    struct area *next;
} area;

typedef struct court {
    string identifier,
        string;
    cardinal rank;
    struct court *next;
} court;
```

```
typedef struct {
    court *court_head;
    area *area_head;
} case_law_specification;

/* external functions */

extern boolean
Is_Zero(
        floating_point x);

extern boolean
Is_Equal(
        floating_point x,
        floating_point y,
        floating_point precision);

extern boolean
Is_Less(
        floating_point x,
        floating_point y,
        floating_point precision);

extern boolean
Is_Zero_Subdistance(
        distance_subtype x);

extern boolean
Is_Zero_Distance(
        distance_type x);

extern boolean
Attribute_Value(
        attribute_value_type attribute_value,
        floating_point *value);

extern attribute_value_type
Nearest_Attribute_Value(
        floating_point value);

extern void
Write_Floating_Point(
        file stream,
        floating_point number,
        string warning_string);

extern case_law_specification
Initialize_Cases(
        file log_stream,
        boolean inputable_latex,
        boolean verbose,
        string specification_filename,
        string dump_filename,
        string probabilities_filename,
        string weights_filename);
```

extern *string*
Case_Law(
file log_stream,
case_law_specification case_law,
string area_identifier,
boolean adjust,
boolean echo,
boolean inputable_latex,
boolean verbose,
cardinal hypothetical_reports,
cardinal hypothetical_changes,
cardinal level,
string distances_filename,
string weights_filename,
string report_filename);

cases.c

/* This is the implementation file for the CASES module. */

```
#include ⟨stdio.h⟩
#include ⟨stdlib.h⟩
#include ⟨string.h⟩
#include ⟨math.h⟩
#include "shyster.h"
#include "cases.h"
#include "tokenizer.h"
#include "parser.h"
#include "dumper.h"
#include "checker.h"
#include "scales.h"
#include "adjuster.h"
#include "consultant.h"
#include "odometer.h"
#include "reporter.h"
```

static void
error_exit(
file stream,
const *string message*)
{
Write_Error_Message_And_Exit(*stream*, "Cases", *message*);
}

static void
warning(
file stream,
const *string message*,
cardinal level)
{
Write_Warning_Message(*stream*, "Cases", *message*, *level*);
}

```
extern boolean
Is_Zero(
        floating_point x)

/* Returns TRUE, iff x = 0 precise to (log Precision) decimal places. */

{
    return floor((double) x × Precision + 0.5) ≡ 0.0;
}

extern boolean
Is_Equal(
        floating_point x,
        floating_point y,
        floating_point precision)

/* Returns TRUE, iff x = y precise to (log precision) decimal places. */

{
    return floor((double) x × precision + 0.5) ≡ floor((double) y × precision + 0.5);
}

extern boolean
Is_Less(
        floating_point x,
        floating_point y,
        floating_point precision)

/* Returns TRUE, iff x < y precise to (log precision) decimal places. */

{
    return floor((double) x × precision + 0.5) < floor((double) y × precision + 0.5);
}

extern boolean
Is_Zero_Subdistance(
        distance_subtype x)

/* Returns TRUE, iff both of x's infinite and finite components are zero. */

{
    return (x.infinite ≡ 0) ∧ Is_Zero(x.finite);
}

extern boolean
Is_Zero_Distance(
        distance_type x)

/* Returns TRUE, iff both of x's known and unknown components are zero. */

{
    return Is_Zero_Subdistance(x.known) ∧ Is_Zero_Subdistance(x.unknown);
}
```

```
extern boolean
Attribute_Value(
        attribute_value_type attribute_value,
        floating_point *value)

/* Sets *value to 1, if attribute_value is YES; to 0, if attribute_value is NO. Returns TRUE, iff
    attribute_value is known. */

{
    switch (attribute_value) {
        case YES:
            *value = Yes_Value;
            return TRUE;
        case NO:
            *value = No_Value;
            return TRUE;
        default:
            return FALSE;
    }
}

extern attribute_value_type
Nearest_Attribute_Value(
        floating_point value)

/* Returns the nearest attribute value to value (0 ≤ value < 0.5: NO; 0.5 ≤ value ≤ 1: YES) */

{
    if (value < 0.5)
        return NO;
    else
        return YES;
}

extern void
Write_Floating_Point(
        file stream,
        floating_point number,
        string warning_string)

/* Writes number, precise to (log Precision) decimal places. If warning_string is not empty, it
    is written after number. */

{
    if (Is_Less(number, 0.0, Precision))
        fprintf(stream, "$-$");

    fprintf(stream, Floating_Point_Format,
            floor(fabs((double) number) × Precision + 0.5) / Precision);

    if (strcmp(warning_string, Empty_String))
        fprintf(stream, "\\rlap{\\makebox[\\tabcolsep]{%s}}", warning_string);
}
```

```
extern case_law_specification
Initialize_Cases(
        file log_stream,
        boolean inputable_latex,
        boolean verbose,
        string specification_filename,
        string dump_filename,
        string probabilities_filename,
        string weights_filename)

/* Calls the TOKENIZER and PARSER to read the case law specification in specification_filename
   and build an internal representation of that specification. Invokes the DUMPER to dump
   that internal representation to dump_filename. Uses the CHECKER to check for attribute de-
   pendence, and to write the probabilities to probabilities_filename. Calls the SCALES module
   to assign weights to all the attributes, and to write those weights to weights_filename. Re-
   turns a pointer to SHYSTER's internal representation of the specification with all attributes
   weighted. */

{
    file specification_stream = NULL,
        dump_stream = NULL,
        probabilities_stream = NULL,
        weights_stream = NULL;
    char filename[Max_Filename_Length];
    char message[Max_Error_Message_Length];
    case_law_specification case_law;
    area *area_pointer;
    result *result_pointer;

    if (specification_filename ≡ NULL)

        /* there is no specification filename */

        error_exit(log_stream, "no case law specification file specified");

    /* open the case law specification file */

    sprintf(filename, "%s%s", specification_filename, Specification_File_Extension);
    if ((specification_stream = fopen(filename, "r")) ≡ NULL) {
        sprintf(message, "can't open case law specification file \"%s\"", filename);
        error_exit(log_stream, message);
    }
    fprintf(log_stream,
            "Reading case law specification from \"%s\" ...\n\n", filename);

    case_law = Parse_Specification(specification_stream, log_stream);

    if (case_law.area_head ≡ NULL)

        /* no areas were specified in the specification file */

        error_exit(log_stream, "no case law specified");

    /* close the case law specification file */

    if (fclose(specification_stream) ≡ EOF) {
        sprintf(message, "can't close case law specification file \"%s\"", filename);
        error_exit(log_stream, message);
    }
```

```
fprintf(log_stream, "Case law specification is valid.\n\n");

/* check that every result in every area has either a case or an ideal point */

for (area_pointer = case_law.area_head; area_pointer ≠ NULL;
        area_pointer = area_pointer→next)
    for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
            result_pointer = result_pointer→next)
        if (result_pointer→case_head ≡ NULL)
            if (result_pointer→ideal_point_head ≡ NULL) {
                sprintf(message, "%s result in %s area has neither cases"
                        " nor an ideal point", result_pointer→identifier,
                        area_pointer→identifier);
                warning(log_stream, message, Top_Level);
            } else {
                sprintf(message, "%s result in %s area has no cases",
                        result_pointer→identifier, area_pointer→identifier);
                warning(log_stream, message, Top_Level);
            }

if (dump_filename ≠ NULL) {

    /* a dump filename was specified, so open the dump file */

    sprintf(filename, "%s%s", dump_filename, LaTeX_File_Extension);
    if ((dump_stream = fopen(filename, "w")) ≡ NULL) {
        sprintf(message, "can't open dump file \"%s\"", filename);
        error_exit(log_stream, message);
    }
    fprintf(log_stream, "Writing dump to \"%s\".\n\n", filename);

    Dump_Specification(dump_stream, log_stream, case_law, inputable_latex, verbose);

    /* close the dump file */

    if (fclose(dump_stream) ≡ EOF) {
        sprintf(message, "can't close dump file \"%s\"", filename);
        error_exit(log_stream, message);
    }
}
if (probabilities_filename ≠ NULL) {

    /* a probabilities filename was specified, so open the probabilities file */

    sprintf(filename, "%s%s", probabilities_filename, LaTeX_File_Extension);
    if ((probabilities_stream = fopen(filename, "w")) ≡ NULL) {
        sprintf(message, "can't open probabilities file \"%s\"", filename);
        error_exit(log_stream, message);
    }
    fprintf(log_stream, "Writing probabilities to \"%s\".\n\n",
            filename);
}
Check_for_Attribute_Dependence(probabilities_stream, log_stream, case_law, inputable_latex);
```

```
if (probabilities_filename ≠ NULL)
    /* a probabilities filename was specified, so close the probabilities file */
    if (fclose(probabilities_stream) ≡ EOF) {
        sprintf(message, "can't␣close␣probabilities␣file␣\"%s\"", filename);
        error_exit(log_stream, message);
    }
if (weights_filename ≠ NULL) {
    /* a weights filename was specified, so open the weights file */
    sprintf(filename, "%s%s", weights_filename, LaTeX_File_Extension);
    if ((weights_stream = fopen(filename, "w")) ≡ NULL) {
        sprintf(message, "can't␣open␣weights␣file␣\"%s\"", filename);
        error_exit(log_stream, message);
    }
    fprintf(log_stream, "Writing␣weights␣to␣\"%s\".\n\n", filename);
}
Weight_Attributes(weights_stream, log_stream, case_law, inputable_latex);
if (weights_filename ≠ NULL)
    /* a weights filename was specified, so close the weights file */
    if (fclose(weights_stream) ≡ EOF) {
        sprintf(message, "can't␣close␣weights␣file␣\"%s\"", filename);
        error_exit(log_stream, message);
    }
    return case_law;
}

static void
write_facts(
        file log_stream,
        vector_element *vector_pointer)

/* Writes the fact vector pointed to by vector_pointer. Uses Y, N and U characters to represent
   YES, NO and UNKNOWN, respectively. */

{
    fprintf(log_stream, "(");
    while (vector_pointer ≠ NULL) {
        switch (vector_pointer→attribute_value) {
            case YES:
                fprintf(log_stream, "Y");
                break;
            case NO:
                fprintf(log_stream, "N");
                break;
            case UNKNOWN:
                fprintf(log_stream, "U");
                break;
        }
```

```
        vector_pointer = vector_pointer→next;
    }
    fprintf(log_stream, ")");
}

static vector_element *
copy_facts(
        file log_stream,
        vector_element *vector_pointer)

/* Returns a pointer to a copy of the fact vector pointed to by vector_pointer. */

{
    vector_element *temp_pointer;

    /* allocate memory for this vector element */

    if ((temp_pointer =
                    (vector_element *) malloc(sizeof(vector_element))) ≡ NULL)
        error_exit(log_stream, "malloc␣failed␣during␣fact␣copying");

    temp_pointer→attribute_value = vector_pointer→attribute_value;
    if (vector_pointer→next ≡ NULL)
        temp_pointer→next = NULL;
    else
        temp_pointer→next = copy_facts(log_stream, vector_pointer→next);
    return temp_pointer;
}

static void
remove_facts(vector_element *vector_pointer)

/* Frees the memory taken up by the fact vector pointed to by vector_pointer. */


{
    if (vector_pointer ≠ NULL) {
        remove_facts(vector_pointer→next);
        free(vector_pointer);
    }
}

static void
mark_differences(
        file log_stream,
        vector_element *vector_pointer_X,
        vector_element *vector_pointer_Y)

/* Marks, with carets, the differences between the two fact vectors pointed to by vector_pointer_X and vector_pointer_Y (one of which has already been written on the previous line of log_stream). */
```

```
{
    if (vector_pointer_X ≠ NULL) {
        if (vector_pointer_X→attribute_value ≠ vector_pointer_Y→attribute_value)
            fprintf(log_stream, "^");
        else
            fprintf(log_stream, "␣");
        mark_differences(log_stream, vector_pointer_X→next, vector_pointer_Y→next);
    }
}

static void
instantiate(
        file log_stream,
        file distances_stream,
        file report_stream,
        case_law_specification case_law,
        area *area_pointer,
        vector_element *instantiated_head,
        vector_element *facts_head,
        result *nearest_result,
        boolean verbose,
        cardinal *instantiation_number,
        cardinal *different_results,
        cardinal level)

/* Instantiates the unknown attribute values in the fact vector pointed to by instantiated_head
    to create instantiations of the instant case in which all the attribute values are known.
    Treats each instantiation as if it were a new instant case, and invokes the ODOMETER and
    the REPORTER to recalculate the distances and argue with the instantiation. */

{
    vector_element *vector_pointer;
    static char message[Max_Error_Message_Length];
    boolean all_known = TRUE;
    cardinal temp_cardinal = *instantiation_number / 10;

    /* make a copy of the fact vector pointed to by instantiated_head */

    instantiated_head = copy_facts(log_stream, instantiated_head);

    for (vector_pointer = instantiated_head; (vector_pointer ≠ NULL) ∧
            (all_known = vector_pointer→attribute_value ≠ UNKNOWN);
            vector_pointer = vector_pointer→next);

    if (all_known ∧ (*instantiation_number ≠ 0)) {
        /* all of the attribute values in the instantiation are known, and the instantiation is
            not the instant case itself, so treat it as if it were a new instant case */

        Indent(log_stream, level);
        fprintf(log_stream, "Instantiation␣%u␣is␣", *instantiation_number);
        write_facts(log_stream, instantiated_head);
        fprintf(log_stream, ".\n");

        Indent(log_stream, level + 5);
```

```
while (temp_cardinal ≠ 0) {
    fprintf(log_stream, "␣");
    temp_cardinal = temp_cardinal / 10;
}

mark_differences(log_stream, instantiated_head, facts_head);
fprintf(log_stream, "\n");

Calculate_Distances(distances_stream, log_stream, area_pointer, case_law,
    instantiated_head, FALSE, *instantiation_number, level + 1);

if (area_pointer→nearest_result ≡ nearest_result) {

    /* the same result was reached as was reached in the instant case, so write a report
       to a NULL report file (nothing will be added to the report, although information
       will still be added to the log file) */

    Write_Report(NULL, log_stream, area_pointer, instantiated_head, facts_head,
        verbose, FALSE, FALSE, *instantiation_number, level + 1);

} else {

    /* a different result was reached, so write a full report on the instantiation */

    Write_Report(report_stream, log_stream, area_pointer, instantiated_head, facts_head,
        verbose, FALSE, FALSE, *instantiation_number, level + 1);

    sprintf(message, "Instantiation␣%u␣in␣%s␣area␣has␣a␣different␣result␣"
        "to␣that␣of␣the␣uninstantiated␣instant␣case",
        *instantiation_number, area_pointer→identifier);
    warning(log_stream, message, level);

    (*different_results)++;
  }

  (*instantiation_number)++;
}
if (*instantiation_number ≡ 0)
  *instantiation_number = 1;

if (¬all_known) {

  /* vector_pointer points to the first UNKNOWN attribute value in the instantiation, so
     set it to YES and instantiate the whole fact vector then set it to NO and instantiate
     the whole fact vector again */

  vector_pointer→attribute_value = YES;
  instantiate(log_stream, distances_stream, report_stream, case_law, area_pointer,
      instantiated_head, facts_head, nearest_result, verbose, instantiation_number,
      different_results, level);
  vector_pointer→attribute_value = NO;
  instantiate(log_stream, distances_stream, report_stream, case_law, area_pointer,
      instantiated_head, facts_head, nearest_result, verbose, instantiation_number,
      different_results, level);
  }

  /* free the memory taken up by the instantiation */

  remove_facts(instantiated_head);
}
```

```
static void
add_to_hypothetical_list(
        file log_stream,
        vector_element *new_head,
        distance_type new_distance,
        hypothetical_list_element **hypothetical_list_pointer,
        cardinal count,
        cardinal hypothetical_reports)

/* Inserts the hypothetical pointed to by new_head into the list of hypotheticals pointed to by
   *hypothetical_list_pointer. The list is kept sorted: the nearer a hypothetical is to the instant
   case, the closer it is to the head of the list (the new hypothetical is new_distance from the
   instant case). *hypothetical_list_pointer points to hypothetical number count in the list.
   The total number of hypotheticals in the list is not allowed to exceed hypothetical_reports.
   Removes the last hypothetical in the list if inserting the new hypothetical makes the list
   too long. */

{
    hypothetical_list_element *temp_pointer;

    if (count > hypothetical_reports)

        /* *hypothetical_list_pointer points past the end of the list, so return without inserting
           anything into the list */

        return;

    if ((*hypothetical_list_pointer ≡ NULL) ∨ (Relative_Distance(new_distance,
                        (*hypothetical_list_pointer)→nearest_neighbour_distance) ≡
                NEARER)) {

        /* the new hypothetical belongs at the head of the list, so allocate memory for it, make
           a copy of it, and put that copy at the head of the list */

        if ((temp_pointer =
                        (hypothetical_list_element *) malloc(sizeof(hypothetical_list_element))) ≡
                NULL)
            error_exit(log_stream, "malloc␣failed␣during␣hypothetical␣handling");

        temp_pointer→hypothetical_head = copy_facts(log_stream, new_head);
        temp_pointer→nearest_neighbour_distance = new_distance;
        temp_pointer→next = *hypothetical_list_pointer;
        *hypothetical_list_pointer = temp_pointer;

        /* skip through to the end of the list */

        while ((temp_pointer→next ≠ NULL) ∧ (count < hypothetical_reports)) {
            temp_pointer = temp_pointer→next;
            count++;
        }
```

```
    if (temp_pointer→next ≠ NULL) {
        /* there's now one more hypothetical in the list than will be required when reports
           are written, so remove the last hypothetical from the list and free the memory it
           takes up */
        remove_facts(temp_pointer→next→hypothetical_head);
        free(temp_pointer→next);
        temp_pointer→next = NULL;
        }
    } else
        /* the hypothetical does not go at the head of the list, so check whether it belongs
           somewhere in the rest of the list */
        add_to_hypothetical_list(log_stream, new_head, new_distance,
                &(*hypothetical_list_pointer)→next, count + 1, hypothetical_reports);
}

static void
hypothesize(
        file log_stream,
        case_law_specification case_law,
        area *area_pointer,
        attribute *attribute_pointer,
        vector_element *hypothetical_head,
        vector_element *facts_head,
        result *instant_result,
        distance_type instant_distance,
        cardinal *hypothetical_number,
        cardinal hypothetical_reports,
        cardinal hypothetical_changes,
        cardinal level)

/* Makes hypothetical variations to the fact vector pointed to by hypothetical_head, allowing
   no more than hypothetical_changes differences (in the known attribute values) between the
   hypothetical and the instant case (the fact vector of which is pointed to by facts_head).
   Treats each hypothetical as if it were a new instant case, and invokes the ODOMETER to
   recalculate the distances.

   Builds (for each result) a list of those hypotheticals which are eligible to be reported on.
   A hypothetical is considered eligible to be reported on if its nearest result is different
   to that of the instant case (pointed to by instant_result), or if it has the same nearest
   result but its nearest neighbour is nearer the instant case than is that of the instant case
   (which is instant_distance from the instant case). Lists only the nearest hypothetical_reports
   hypotheticals for each result. */
{
    vector_element *vector_pointer = facts_head,
        *hypothetical_pointer,
        *new_head;
    cardinal count;

    if (attribute_pointer ≡ NULL)
        return;
```

```
/* count the known differences between the hypothetical and the instant case */
count = 0;
for (hypothetical_pointer = hypothetical_head; hypothetical_pointer ≠ NULL;
        hypothetical_pointer = hypothetical_pointer→next) {
    if ((vector_pointer→attribute_value ≠ UNKNOWN) ∧
            (hypothetical_pointer→attribute_value ≠ vector_pointer→attribute_value))
        count++;
    vector_pointer = vector_pointer→next;
}
if ((hypothetical_changes ≠ 0) ∧ (count ≡ hypothetical_changes))
    /* the maximum number of changes has already been made */
    return;
new_head = copy_facts(log_stream, hypothetical_head);
/* find the attribute to change in the new hypothetical */
count = 1;
for (hypothetical_pointer = new_head; count < attribute_pointer→number;
        hypothetical_pointer = hypothetical_pointer→next)
    count++;
if (hypothetical_pointer→attribute_value ≡ UNKNOWN) {
    /* the value of the attribute to change in the new hypothetical is UNKNOWN, so ignore
       it—UNKNOWN values have already been instantiated by instantiate()—and continue
       hypothesizing using the next attribute */
    hypothesize(log_stream, case_law, area_pointer, attribute_pointer→next,
            new_head, facts_head, instant_result, instant_distance,
            hypothetical_number, hypothetical_reports, hypothetical_changes, level);
} else {
    (*hypothetical_number)++;
    /* change the (known) value of the attribute in the new hypothetical */
    if (hypothetical_pointer→attribute_value ≡ YES)
        hypothetical_pointer→attribute_value = NO;
    else
        hypothetical_pointer→attribute_value = YES;
    /* calculate distances without writing anything to the log file or the distances file */
    Calculate_Distances(NULL, NULL, area_pointer, case_law, new_head, TRUE, 0, level + 1);
    if (area_pointer→nearest_result ≡ instant_result) {
        /* the hypothetical has the same result as that of the instant case, so only add it
           to the hypothetical list if its nearest neighbour is nearer than was that of the
           instant case */
        if (instant_result→nearest_known_compared_with_unknown ≠ FURTHER) {
            /* the nearest known neighbour is the nearest neighbour (although there may
               be an equidistant case with an unknown distance) */
```

```
            if (Relative_Distance(instant_result→nearest_known_case→metrics.distance,
                        instant_distance) ≡ NEARER)
                add_to_hypothetical_list(log_stream, new_head,
                        instant_result→nearest_known_case→metrics.distance,
                        &instant_result→hypothetical_list_head, 1, hypothetical_reports);
        } else {
            /* the nearest unknown neighbour is the nearest neighbour */
            if (Relative_Distance(instant_result→nearest_unknown_case→metrics.distance,
                        instant_distance) ≡ NEARER)
                add_to_hypothetical_list(log_stream, new_head,
                        instant_result→nearest_unknown_case→metrics.distance,
                        &instant_result→hypothetical_list_head, 1, hypothetical_reports);
            }
        } else {
            /* the hypothetical has a different result to that of the instant case */
            if (area_pointer→nearest_result→nearest_known_compared_with_unknown ≠
                    FURTHER)
                /* the nearest known neighbour is the nearest neighbour (although there may
                    be an equidistant case with an unknown distance) */
                add_to_hypothetical_list(log_stream, new_head,
                        area_pointer→nearest_result→nearest_known_case→metrics.distance,
                        &area_pointer→nearest_result→hypothetical_list_head, 1,
                        hypothetical_reports);
            else
                /* the nearest unknown neighbour is the nearest neighbour */
                add_to_hypothetical_list(log_stream, new_head,
                        area_pointer→nearest_result→nearest_unknown_case→metrics.distance,
                        &area_pointer→nearest_result→hypothetical_list_head, 1,
                        hypothetical_reports);
        }
        /* hypothesize, using the next attribute, with the unchanged hypothetical */
        hypothesize(log_stream, case_law, area_pointer, attribute_pointer→next,
                hypothetical_head, facts_head, instant_result, instant_distance,
                hypothetical_number, hypothetical_reports, hypothetical_changes, level);
        /* ... and with the new hypothetical */
        hypothesize(log_stream, case_law, area_pointer, attribute_pointer→next,
                new_head, facts_head, instant_result, instant_distance,
                hypothetical_number, hypothetical_reports, hypothetical_changes, level);
        }
        /* free the memory taken up by the hypothetical */
        remove_facts(new_head);
    }
```

```
static void
write_hypotheticals(
        file log_stream,
        file distances_stream,
        file report_stream,
        case_law_specification case_law,
        area *area_pointer,
        vector_element *facts_head,
        boolean verbose,
        boolean same_result,
        hypothetical_list_element *hypothetical_list_pointer,
        cardinal *hypothetical_number,
        cardinal level)

/* Writes reports on the hypotheticals in the list, the head of which is pointed to by
   hypothetical_list_pointer. */

{
    cardinal temp_cardinal;

    /* while there are still hypotheticals in the list ... */

    while (hypothetical_list_pointer ≠ NULL) {

        /* treat the hypothetical as if it were the instant case */

        (*hypothetical_number)++;
        Indent(log_stream, level);
        fprintf(log_stream, "Hypothetical␣%u␣is␣", *hypothetical_number);
        write_facts(log_stream, hypothetical_list_pointer→hypothetical_head);
        fprintf(log_stream, ".\n");

        Indent(log_stream, level + 4);
        fprintf(log_stream, "␣␣␣");
        temp_cardinal = *hypothetical_number / 10;
        while (temp_cardinal ≠ 0) {
            fprintf(log_stream, "␣");
            temp_cardinal = temp_cardinal / 10;
        }

        mark_differences(log_stream, hypothetical_list_pointer→hypothetical_head, facts_head);
        fprintf(log_stream, "\n");

        /* calculate the distances again (when they were calculated in hypothesize(), nothing
           was written to the log file or the distances file) and write a report on this hypothetical
           (which also writes to the log file) */

        Calculate_Distances(distances_stream, log_stream, area_pointer, case_law,
                hypothetical_list_pointer→hypothetical_head, TRUE, *hypothetical_number,
                level + 1);

        Write_Report(report_stream, log_stream, area_pointer,
                hypothetical_list_pointer→hypothetical_head, facts_head, verbose, TRUE,
                same_result, *hypothetical_number, level + 1);

        hypothetical_list_pointer = hypothetical_list_pointer→next;
    }
}
```

```
extern string
Case_Law(
        file log_stream,
        case_law_specification case_law,
        string area_identifier,
        boolean adjust,
        boolean echo,
        boolean inputable_latex,
        boolean verbose,
        cardinal hypothetical_reports,
        cardinal hypothetical_changes,
        cardinal level,
        string distances_filename,
        string weights_filename,
        string report_filename)
```

/* Determines the "likely result" of the instant case in the *area_identifier* area of the *case_law* specification, and constructs an argument supporting that conclusion.

Calls the ADJUSTER, if *adjust* is *TRUE*. Invokes the CONSULTANT to interrogate the user as to the attribute values in the instant case. (The CONSULTANT recursively invokes *Case_Law*(), if required, to resolve open textured—external—attributes.) Calls the ODOMETER to calculate the distances between the instant case and the leading cases, and to determine the nearest neighbours and results. Invokes the REPORTER to write a report about the instant case.

Returns the identifier of the "likely result." */

```
{
    file distances_stream = NULL,
        report_stream = NULL;
    static char filename[Max_Filename_Length];
    static char message[Max_Error_Message_Length];
    vector_element *facts_head;
    area *area_pointer;
    result *result_pointer,
       *instant_result = NULL;
    kase *case_pointer;
    distance_type instant_distance;
    cardinal instantiation_number = 0,
        hypothetical_number = 0,
        different_results = 0,
        hypothetical_count = 0;

    /* find the area with an identifier matching area_identifier */
    for (area_pointer = case_law.area_head;
            (area_pointer ≠ NULL) ∧ strcmp(area_pointer→identifier, area_identifier);
            area_pointer = area_pointer→next);

    if (area_pointer ≡ NULL) {

        /* area_identifier does not match the identifier of any area */

        sprintf(message, "%s area not found", area_identifier);
        error_exit(log_stream, message);
    }
```

```
Indent(log_stream, level);
fprintf(log_stream, "Area is %s.\n\n", area_identifier);
if (distances_filename ≠ NULL) {
    /* a distances filename was specified, so open a distances file for this area */
    sprintf(filename, "%s-%s%s", distances_filename,
            area_identifier, LaTeX_File_Extension);
    if ((distances_stream = fopen(filename, "w")) ≡ NULL) {
        sprintf(message, "can't open distances file \"%s\"", filename);
        error_exit(log_stream, message);
    }
    Indent(log_stream, level);
    fprintf(log_stream, "Writing distances to \"%s\".\n\n", filename);
}
if (report_filename ≠ NULL) {
    /* a report filename was specified, so open a report file for this area */
    sprintf(filename, "%s-%s%s", report_filename,
            area_identifier, LaTeX_File_Extension);
    if ((report_stream = fopen(filename, "w")) ≡ NULL) {
        sprintf(message, "can't open report file \"%s\"", filename);
        error_exit(log_stream, message);
    }
    Indent(log_stream, level);
    fprintf(log_stream, "Writing report to \"%s\".\n\n", filename);
}
if (adjust)
    Adjust_Attributes(log_stream, area_pointer, weights_filename, level, inputable_latex);
/* interrogate the user as to the facts in the instant case */
if ((facts_head = Get_Facts(log_stream, case_law, area_pointer, adjust, echo, inputable_latex,
                            verbose, hypothetical_reports, hypothetical_changes, level,
                            distances_filename, weights_filename, report_filename)) ≠ NULL) {
    /* the user has entered some facts */
    if (distances_stream ≠ NULL) {
        /* a distances file is open for this area, so write its header */
        fprintf(distances_stream, "%% Distances file\n\n");
        Write_LaTeX_Header(distances_stream, inputable_latex);
    }
    if (report_stream ≠ NULL) {
        /* a report file is open for this area, so write its header */
        fprintf(report_stream, "%% Report file\n\n");
        Write_LaTeX_Header(report_stream, inputable_latex);
}
Indent(log_stream, level);
fprintf(log_stream, "Fact vector is ");
write_facts(log_stream, facts_head);
fprintf(log_stream, ".\n\n");
```

```
Calculate_Distances(distances_stream, log_stream, area_pointer,
        case_law, facts_head, FALSE, 0, level + 1);

/* mark all cases as unsummarized */

for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
        result_pointer = result_pointer→next)
    for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
            case_pointer = case_pointer→next)
        case_pointer→summarized = FALSE;

Write_Report(report_stream, log_stream, area_pointer, facts_head,
        NULL, verbose, FALSE, FALSE, 0, level + 1);

instant_result = area_pointer→nearest_result;

if (instant_result→nearest_known_compared_with_unknown ≠ FURTHER)

    /* the nearest known neighbour is the nearest neighbour (although there may be
       an equidistant case with an unknown distance) */

    instant_distance = instant_result→nearest_known_case→metrics.distance;

else

    /* the nearest unknown neighbour is the nearest neighbour */

    instant_distance = instant_result→nearest_unknown_case→metrics.distance;

/* instantiate the unknown attribute values in the instant case */

instantiate(log_stream, distances_stream, report_stream, case_law, area_pointer,
        facts_head, facts_head, instant_result, verbose, &instantiation_number,
        &different_results, level);

instantiation_number--;

if (hypothetical_reports ≠ 0) {

    /* the user requested hypothesizing */

    hypothesize(log_stream, case_law, area_pointer, area_pointer→attribute_head,
            facts_head, facts_head, instant_result, instant_distance,
            &hypothetical_number, hypothetical_reports, hypothetical_changes, level);

    /* write the hypotheticals for this result */

    write_hypotheticals(log_stream, distances_stream, report_stream,
            case_law, area_pointer, facts_head, verbose, TRUE,
            instant_result→hypothetical_list_head, &hypothetical_count, level);

    /* write the hypotheticals for all other results */

    for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
            result_pointer = result_pointer→next)
        if (result_pointer ≠ instant_result)
            write_hypotheticals(log_stream, distances_stream, report_stream,
                    case_law, area_pointer, facts_head, verbose, FALSE,
                    result_pointer→hypothetical_list_head, &hypothetical_count, level);
}
```

```
/* write details of the instantiations to the log file */

if (instantiation_number ≡ 0)
    sprintf(message, "No instantiations");
else if (different_results ≡ 0) {
    if (instantiation_number ≡ 2)
        sprintf(message, "Both");
    else
        sprintf(message, "All %u", instantiation_number);
    sprintf(message, "%s instantiations have the same nearest result "
            "as does the instant case", message);
} else
    sprintf(message, "%u of the %u instantiations %s a nearest result "
            "different to that of the instant case", different_results,
            instantiation_number, different_results ≡ 1 ? "has" : "have");
Write(log_stream, message, ".\n", level, Hang);

/* write details of the hypothesizing to the log file */

Indent(log_stream, level);
if (hypothetical_reports ≡ 0)
    fprintf(log_stream, "No hypotheticals.\n\n");
else {
    fprintf(log_stream, "Reported on %u hypothetical%s of %u",
            hypothetical_count, hypothetical_count ≡ 1 ? "" : "s", hypothetical_number);
    if (hypothetical_changes ≠ 0)
        fprintf(log_stream, " (limit of %u change%s)", hypothetical_changes,
                hypothetical_changes ≡ 1 ? "" : "s");
        fprintf(log_stream, ".\n\n");
    }

    if (distances_stream ≠ NULL)

        /* a distances file is open for this area, so write its trailer */

        Write_LaTeX_Trailer(distances_stream, inputable_latex);

    if (report_stream ≠ NULL)

        /* a report file is open for this area, so write its trailer */

        Write_LaTeX_Trailer(report_stream, inputable_latex);
}
if (distances_filename ≠ NULL) {
    /* a distances filename was specified, so close the distances file */

    sprintf(filename, "%s-%s%s", distances_filename,
            area_identifier, LaTeX_File_Extension);
    if (fclose(distances_stream) ≡ EOF) {
        sprintf(message, "can't close distances file \"%s\"", filename);
        error_exit(log_stream, message);
    }
}
```

```
if (report_filename ≠ NULL) {
    /* a report filename was specified, so close the report file */
    sprintf(filename, "%s-%s%s", report_filename,
            area_identifier, LaTeX_File_Extension);
    if (fclose(report_stream) ≡ EOF) {
        sprintf(message, "can't close report file \"%s\"", filename);
        error_exit(log_stream, message);
        }
    }
    /* return the identifier of the "likely result" */
    if (instant_result ≡ NULL)
        return NULL;
    else
        return instant_result→identifier;
}
```

4

The TOKENIZER module

tokenizer.h

```
/* This is the header file for the TOKENIZER module. It is also included by the CASES and
   PARSER modules. */

/* string and character constants */

#define Quoted_LaTeX_Characters "$&%"
#define Comment_Character '%'
#define Quote_Character '"'
#define Equals_Character '='
#define Hyphen_Character '-'
#define Tab_Character '\t'
#define Vertical_Tab_Character '\v'
#define Form_Feed_Character '\f'
#define Backslash_Character '\\'

/* enumerated types */

typedef enum {
    TK_KEYWORD,
    TK_IDENTIFIER,
    TK_STRING,
    TK_YEAR,
    TK_ATTRIBUTE_VECTOR,
    TK_EQUALS,
    TK_EOF
} token_type;
```

```
typedef enum {
    KW_AREA,
    KW_ATTRIBUTE,
    KW_CASE,
    KW_CITATION,
    KW_CLOSING,
    KW_COURT,
    KW_EXTERNAL,
    KW_FACTS,
    KW_HELP,
    KW_HIERARCHY,
    KW_IDEAL,
    KW_NO,
    KW_OPENING,
    KW_QUESTION,
    KW_RESULT,
    KW_RESULTS,
    KW_SUMMARY,
    KW_UNKNOWN,
    KW_YEAR,
    KW_YES
} keyword_type;

/* structure type */

typedef struct {
    cardinal line_number,
        column_number;
    token_type token;
    union {
        keyword_type keyword;
        string identifier,
            string;
        cardinal year;
        matrix_element *matrix_head;
    } details;
} token_details;

/* external function */

extern token_details
Get_Token(
        file in_stream,
        file log_stream);
```

tokenizer.c

```
/* This is the implementation file for the TOKENIZER module. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "shyster.h"
#include "cases.h"
#include "tokenizer.h"
```

```
static void
error_exit(
        file stream,
        string message,
        token_details *token)
{
    char full_message[Max_Error_Message_Length];

    sprintf(full_message, "%s [%u,%u]", message, token→line_number,
            token→column_number);
    Write_Error_Message_And_Exit(stream, "Tokenizer", full_message);
}


static void
warning(
        file stream,
        const string message,
        const token_details *token)
{
    char full_message[Max_Error_Message_Length];

    sprintf(full_message, "%s [%u,%u]", message, token→line_number,
            token→column_number);
    Write_Warning_Message(stream, "Tokenizer", full_message, Top_Level);
}


static int
get_char(
        file in_stream,
        cardinal *line_number,
        cardinal *column_number,
        boolean *eof)

/* Returns the next character from in_stream. Adjusts *line_number and *column_number
   appropriately. Sets *eof to TRUE, if the end of in_stream has been encountered (i.e. the
   character returned is EOF). */

{
    int ch;

    if (¬(*eof = (ch = getc(in_stream)) ≡ EOF))
        if (ch ≡ Carriage_Return_Character) {
            (*line_number)++;
            *column_number = 0;
        } else
            (*column_number)++;
    return ch;
}
```

```
static void
unget_char(
        file in_stream,
        file log_stream,
        int ch,
        token_details *token,
        cardinal *line_number,
        cardinal *column_number)

/* Pushes ch back onto in_stream. Adjusts *line_number and *column_number appro-
    riately. */

{
    char message[Max_Error_Message_Length];

    if (ch ≡ Carriage_Return_Character)
        (*line_number)--;
    else
        (*column_number)--;
    if (ch ≠ EOF)
        if (ungetc((int) ch, in_stream) ≡ EOF) {
            sprintf(message, "ungetc failed with character '%c'", ch);
            error_exit(log_stream, message, token);
        }
}

static boolean
is_whitespace(
        int ch)

/* Returns TRUE, iff ch is a whitespace character (a space, a tab, a vertical tab, a carriage
    return, or a form feed). */

{
    return ((ch ≡ Space_Character) ∨ (ch ≡ Tab_Character) ∨
            (ch ≡ Vertical_Tab_Character) ∨ (ch ≡ Carriage_Return_Character) ∨
            (ch ≡ Form_Feed_Character));
}

static boolean
is_alpha(
        int ch)

/* Returns TRUE, iff ch is an alphabetic character (A ... Z, a ... z). */

{
    return (((ch ≥ Big_A_Character) ∧ (ch ≤ Big_Z_Character)) ∨
            ((ch ≥ Little_A_Character) ∧ (ch ≤ Little_Z_Character)));
}
```

```
static void
get_keyword_or_ident(
        file in_stream,
        file log_stream,
        int ch,
        token_details *token,
        cardinal *line_number,
        cardinal *column_number,
        boolean *eof)

/* Gets an identifier, which may be a keyword (the first character of the identifier—ch—
   has just been read). Changes the structure pointed to by token: sets token→token to
   TK_KEYWORD or TK_IDENTIFIER, and sets token→details appropriately.

   EBNF: identifier = letter { letter | digit | "-" }.                              */

{
   cardinal length = 1;
   string identifier;
   char message[Max_Error_Message_Length];

   /* allocate memory for the identifier */

   if ((identifier = (string) malloc((Max_Identifier_Length + 1) × sizeof(char))) ≡ NULL)
      error_exit(log_stream, "malloc␣failed␣during␣keyword/identifier␣handling",
            token);

   /* put up to Max_Identifier_Length characters into the identifier */

   identifier[0] = ch;
   ch = get_char(in_stream, line_number, column_number, eof);
   while ((length < Max_Identifier_Length) ∧
         (is_alpha(ch) ∨ Is_Digit(ch) ∨ (ch ≡ Hyphen_Character))) {
      identifier[length++] = ch;
      ch = get_char(in_stream, line_number, column_number, eof);
   }
   identifier[length] = Null_Character;

   if (is_alpha(ch) ∨ Is_Digit(ch) ∨ (ch ≡ Hyphen_Character)) {

      /* there is more of the identifier, so warn the user and skip over the rest of it */

      sprintf(message, "identifier␣truncated␣to␣\"%s\"", identifier);
      warning(log_stream, message, token);
      while (is_alpha(ch) ∨ Is_Digit(ch) ∨ (ch ≡ Hyphen_Character))
         ch = get_char(in_stream, line_number, column_number, eof);

   } else

      /* reallocate (just enough) memory for the identifier */

      if ((identifier = (string) realloc((void *) identifier, length × sizeof(char))) ≡
            NULL)
         error_exit(log_stream, "realloc␣failed␣during␣keyword/identifier␣handling",
               token);

   /* push the first character after the identifier back onto in_stream */

   unget_char(in_stream, log_stream, ch, token, line_number, column_number);
```

```
/* check whether the identifier is a keyword */

if (¬strcmp(identifier, "AREA")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_AREA;
} else if (¬strcmp(identifier, "ATTRIBUTE")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_ATTRIBUTE;
} else if (¬strcmp(identifier, "CASE")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_CASE;
} else if (¬strcmp(identifier, "CITATION")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_CITATION;
} else if (¬strcmp(identifier, "CLOSING")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_CLOSING;
} else if (¬strcmp(identifier, "COURT")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_COURT;
} else if (¬strcmp(identifier, "EXTERNAL")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_EXTERNAL;
} else if (¬strcmp(identifier, "FACTS")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_FACTS;
} else if (¬strcmp(identifier, "HELP")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_HELP;
} else if (¬strcmp(identifier, "HIERARCHY")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_HIERARCHY;
} else if (¬strcmp(identifier, "IDEAL")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_IDEAL;
} else if (¬strcmp(identifier, "NO")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_NO;
} else if (¬strcmp(identifier, "OPENING")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_OPENING;
} else if (¬strcmp(identifier, "QUESTION")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_QUESTION;
} else if (¬strcmp(identifier, "RESULT")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_RESULT;
} else if (¬strcmp(identifier, "RESULTS")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_RESULTS;
```

```
} else if (¬strcmp(identifier, "SUMMARY")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_SUMMARY;
} else if (¬strcmp(identifier, "UNKNOWN")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_UNKNOWN;
} else if (¬strcmp(identifier, "YEAR")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_YEAR;
} else if (¬strcmp(identifier, "YES")) {
    token→token = TK_KEYWORD;
    token→details.keyword = KW_YES;
} else {

    /* the identifier is not a keyword */

    token→token = TK_IDENTIFIER;
    token→details.identifier = identifier;
}
}

static void
get_string(
        file in_stream,
        file log_stream,
        token_details *token,
        cardinal *line_number,
        cardinal *column_number,
        boolean *eof)

/* Gets a string (a " character has just been read). Treats a pair of consecutive " characters as
   a single " character. Treats consecutive whitespace characters as a single space character.
   Sets token→details appropriately (token→token has already been set to TK_STRING).

   EBNF: string = """ character {character} """.                                    */

{
    int ch,
        next_ch;
    string temp_string;
    cardinal allocated_length,
        actual_length;

    allocated_length = String_Increment;
    actual_length = 0;

    /* allocate memory for the string */

    if ((temp_string = (string) malloc(allocated_length × sizeof(char))) ≡ NULL)
        error_exit(log_stream, "malloc␣failed␣during␣string␣handling", token);

    /* get the first character of the string */

    ch = get_char(in_stream, line_number, column_number, eof);
```

```
for (;;) {
    if (ch ≡ EOF)
        error_exit(log_stream, "end of file in string", token);
    if (strchr(Quoted_LaTeX_Characters, ch) ≠ NULL) {
        /* the character is one of those in Quoted_LaTeX_Characters (i.e. it is $, & or %);
           it has a special meaning in LaTeX and needs to be prefixed in the string by a
           \ character */
        temp_string[actual_length++] = Backslash_Character;
        if (actual_length ≡ allocated_length)
            /* the string is too long for temp_string, so reallocate some more memory */
            if ((temp_string = (string) realloc((void *) temp_string,
                                    (allocated_length += String_Increment) ×
                                    sizeof(char))) ≡ NULL)
                error_exit(log_stream, "realloc failed during string handling",
                    token);
    }
    if (ch ≡ Quote_Character)
        /* the character is a " character */
        if ((next_ch = get_char(in_stream, line_number, column_number, eof)) ≠
                Quote_Character) {
            /* the next character is not a " character so this is the end of the string; push
               the first character after the string back onto in_stream */
            unget_char(in_stream, log_stream, next_ch, token, line_number, column_number);
            if (actual_length ≡ 0)
                error_exit(log_stream, "empty string", token);
            else {
                temp_string[actual_length++] = Null_Character;
                if (actual_length < allocated_length)
                    /* reallocate (just enough) memory for the string */
                    if ((temp_string = (string) realloc((void *) temp_string,
                                            actual_length × sizeof(char))) ≡ NULL)
                        error_exit(log_stream, "realloc failed during string handling",
                            token);
            }
            token→details.string = temp_string;
            return;
        }
    if (is_whitespace(ch)) {
        /* skip to the next non-whitespace character */
        for (ch = get_char(in_stream, line_number, column_number, eof);
                is_whitespace(ch);
                ch = get_char(in_stream, line_number, column_number, eof));
```

```
    if (ch ≡ EOF)
        error_exit(log_stream, "end of file in string", token);
    /* put a single space character in the string for all of the whitespace */
    temp_string[actual_length++] = Space_Character;
    /* push the non-whitespace character back onto in_stream */
    unget_char(in_stream, log_stream, ch, token, line_number, column_number);
  } else
    temp_string[actual_length++] = ch;
  if (actual_length ≡ allocated_length)
    /* the string is too long for temp_string, so reallocate some more memory */
    if ((temp_string = (string) realloc((void *) temp_string,
                                        (allocated_length += String_Increment) ×
                                        sizeof(char))) ≡ NULL)
      error_exit(log_stream, "realloc failed during string handling", token);
  /* get the next character */
  ch = get_char(in_stream, line_number, column_number, eof);
 }
}

static void
get_year(
    file in_stream,
    file log_stream,
    int ch,
    token_details *token,
    cardinal *line_number,
    cardinal *column_number,
    boolean *eof)

/* Gets a year (the first digit of the year—ch—has just been read). Sets token→details appro-
   priately (token→token has already been set to TK_YEAR).
   EBNF: year = digit [digit] [digit] [digit].                                    */
{
  cardinal digits = 1,
    year = (cardinal) ch − (cardinal) Zero_Character;
  for (ch = get_char(in_stream, line_number, column_number, eof);
       (Is_Digit(ch) ∧ (digits < Year_Digits));
       ch = get_char(in_stream, line_number, column_number, eof)) {
    year = (10 × year) + (cardinal) ch − (cardinal) Zero_Character;
    digits++;
  }
  if (Is_Digit(ch))
    error_exit(log_stream, "year has too many digits", token);
  unget_char(in_stream, log_stream, ch, token, line_number, column_number);
  token→details.year = year;
}
```

```
static void
get_attribute_vector(
        file in_stream,
        file log_stream,
        token_details *token,
        cardinal *line_number,
        cardinal *column_number,
        boolean *eof)
```

/* Gets an attribute vector (a left parenthesis character has just been read). Sets *token→details* appropriately (*token→token* has already been set to *TK_YEAR*).

EBNF: attribute-vector = "(" attribute-value { attribute-value } ")".
attribute-value = "Y" | "N" | "U". */

```
{
    int ch;
    matrix_element *matrix_head,
      *matrix_pointer;
    boolean empty = TRUE;
    char message[Max_Error_Message_Length];

    /* allocate memory for this matrix element (the first in the list) */

    if ((matrix_head = (matrix_element *) malloc(sizeof(matrix_element))) ≡ NULL)
        error_exit(log_stream, "malloc␣failed␣during␣attribute␣vector␣handling",
            token);

matrix_pointer = matrix_head;

/* for every character that is not a right parenthesis ... */

for (ch = get_char(in_stream, line_number, column_number, eof);
        ch ≠ Attribute_Vector_End_Character;
        ch = get_char(in_stream, line_number, column_number, eof)) {

    if (¬empty) {

        /* allocate memory for this matrix element */

        if ((matrix_pointer→case_next =
                        (matrix_element *) malloc(sizeof(matrix_element))) ≡ NULL)
            error_exit(log_stream, "malloc␣failed␣during␣attribute␣vector␣handling",
                token);
        matrix_pointer = matrix_pointer→case_next;
    }
    switch (ch) {
        case Yes_Character:
            matrix_pointer→attribute_value = YES;
            break;
        case No_Character:
            matrix_pointer→attribute_value = NO;
            break;
```

```
        case Unknown_Character:
            matrix_pointer→attribute_value = UNKNOWN;
            break;
        default:
            sprintf(message, "invalid␣attribute␣value␣‘%c’", ch);
            error_exit(log_stream, message, token);
            break;
    }
    empty = FALSE;
    matrix_pointer→case_next = NULL;
    matrix_pointer→attribute_next = NULL;
  }
  if (empty)
    error_exit(log_stream, "empty␣attribute␣vector", token);
  token→details.matrix_head = matrix_head;
}

static void
skip_to_end_of_line(
    file in_stream,
    cardinal *line_number,
    cardinal *column_number,
    boolean *eof)
/* Skips over characters until the end of the line, or the end of the file, is reached. */
{
  int ch;

  for (;;) {
    ch = get_char(in_stream, line_number, column_number, eof);
    if ((ch ≡ EOF) ∨ (ch ≡ Carriage_Return_Character))
      return;
  }
}

extern token_details
Get_Token(
    file in_stream,
    file log_stream)
/* Returns details of the next token from in_stream. */
{
  token_details token;
  int ch;
  static cardinal line_number = 1,
    column_number = 0;
  static boolean eof = FALSE;
  char message[Max_Error_Message_Length];
```

```
for (;;) {
    if (eof) {
        token.token = TK_EOF;
        return token;
    }
    /* skip to the next non-whitespace character */
    for (ch = get_char(in_stream, &line_number, &column_number, &eof);
            is_whitespace(ch);
            ch = get_char(in_stream, &line_number, &column_number, &eof));
    token.line_number = line_number;
    token.column_number = column_number;
    if (is_alpha(ch)) {
        get_keyword_or_ident(in_stream, log_stream, ch, &token,
                &line_number, &column_number, &eof);
        return token;
    } else if (ch ≡ Quote_Character) {
        token.token = TK_STRING;
        get_string(in_stream, log_stream, &token, &line_number, &column_number, &eof);
        return token;
    } else if (Is_Digit(ch)) {
        token.token = TK_YEAR;
        get_year(in_stream, log_stream, ch, &token, &line_number, &column_number, &eof);
        return token;
    } else if (ch ≡ Attribute_Vector_Begin_Character) {
        token.token = TK_ATTRIBUTE_VECTOR;
        get_attribute_vector(in_stream, log_stream, &token,
                &line_number, &column_number, &eof);
        return token;
    } else if (ch ≡ Equals_Character) {
        token.token = TK_EQUALS;
        return token;
    } else if (ch ≡ EOF) {
        token.token = TK_EOF;
        return token;
    } else if (ch ≡ Comment_Character)
        skip_to_end_of_line(in_stream, &line_number, &column_number, &eof);
    else {
        sprintf(message, "invalid␣character␣'%c'", ch);
        error_exit(log_stream, message, &token);
    }
}
```

5

The PARSER module

parser.h

/* This is the header file for the PARSER module. It is also included by the CASES module. */

/* external function */

extern *case_law_specification*
Parse_Specification(
    file in_stream,
    file log_stream);

parser.c

/* This is the implementation file for the PARSER module. */

#include ⟨stdio.h⟩
#include ⟨stdlib.h⟩
#include ⟨string.h⟩
#include "shyster.h"
#include "cases.h"
#include "parser.h"
#include "tokenizer.h"

static void
error_exit(
    file stream,
    string message,
    *token_details *token*)

```
{
    char full_message[Max_Error_Message_Length];

    if (token ≠ NULL) {
        sprintf(full_message, "%s␣[%u,%u]", message, token→line_number,
                token→column_number);
        Write_Error_Message_And_Exit(stream, "Parser", full_message);
    } else
        Write_Error_Message_And_Exit(stream, "Parser", message);
}

static void
warning(
        file stream,
        const string message)
{
    Write_Warning_Message(stream, "Parser", message, Top_Level);
}

static void
parse_court_pair(
        file in_stream,
        file log_stream,
        court *court_pointer,
        token_details *token,
        cardinal *count,
        cardinal *rank)

/* Parses a court identifier/string pair, and puts details in the court pointed to by court_pointer.
   A court identifier has just been read, and its details are pointed to by token. *count is the
   number of courts already parsed. *rank is the rank of this court.

   EBNF: hierarchy-block = court-identifier string
                           { [ "=" ] court-identifier string }.
         court-identifier = identifier.                          */

{
    court_pointer→identifier = token→details.identifier;

    /* get the next token (it should be a string) */
    *token = Get_Token(in_stream, log_stream);

    if (token→token ≠ TK_STRING)
        error_exit(log_stream, "string␣expected␣in␣hierarchy␣block␣after␣identifier",
                token);

    court_pointer→string = token→details.string;

    court_pointer→rank = (*rank)++;
```

```
    /* get the next token (it should be an =, another court identifier, or the keyword AREA) */
    *token = Get_Token(in_stream, log_stream);
  }
  (*count)++;
  court_pointer->next = NULL;
}

static court *
parse_hierarchy(
        file in_stream,
        file log_stream,
        token_details *token,
        cardinal *count)

/* Parses a hierarchy, and returns a pointer to a list of courts. The keyword HIERARCHY has
    just been read, and the details of the token that followed it are pointed to by token. Sets
    *count to the number of courts in the hierarchy.

    EBNF: hierarchy          = hierarchy-header hierarchy-block.
          hierarchy-header   = "HIERARCHY".
          hierarchy-block    = court-identifier string
                               { [ "=" ] court-identifier string }.
          court-identifier   = identifier.                          */

{
  cardinal rank = 1;
  court *court_head = NULL,
    *court_pointer = NULL,
    *temp_court_pointer;
  char message[Max_Error_Message_Length];

  while (token->token == TK_IDENTIFIER) {

    if (court_head == NULL) {

      /* allocate memory for this court (the first in the list) */

      if ((court_head = (court *) malloc(sizeof(court))) == NULL)
        error_exit(log_stream, "malloc failed during hierarchy handling",
                token);

      court_pointer = court_head;
```

```
} else {
    /* go to the end of the list of courts, checking that this court has not already been
       specified */
    for (temp_court_pointer = court_head; temp_court_pointer ≠ NULL;
            temp_court_pointer = temp_court_pointer→next)
        if (¬strcmp(token→details.identifier, temp_court_pointer→identifier)) {
            sprintf(message, "%s court already specified",
                    token→details.identifier);
            error_exit(log_stream, message, token);
        }
    /* allocate memory for this court */
    if ((court_pointer→next = (court *) malloc(sizeof(court))) ≡ NULL)
        error_exit(log_stream, "malloc failed during hierarchy handling",
                token);
    court_pointer = court_pointer→next;
  }
  parse_court_pair(in_stream, log_stream, court_pointer, token, count, &rank);
 }
 return court_head;
}

static void
parse_result_pair(
        file in_stream,
        file log_stream,
        result *result_pointer,
        token_details *token,
        cardinal *count)

/* Parses a result identifier/string pair, and puts details in the result pointed to by
   result_pointer. A result identifier has just been read, and its details are pointed to by
   token. *count is the number of results already parsed in this area.
   EBNF: results-block     = result-identifier string
                             result-identifier string
                             { result-identifier string }.
         result-identifier = identifier.                          */
{
    result_pointer→identifier = token→details.identifier;
    /* get the next token (it should be a string) */
    *token = Get_Token(in_stream, log_stream);
    if (token→token ≠ TK_STRING)
        error_exit(log_stream, "string expected in results block after identifier",
                token);
    result_pointer→string = token→details.string;
```

```
    result_pointer→case_head = NULL;
    result_pointer→ideal_point_head = NULL;
    result_pointer→centroid_head = NULL;
    result_pointer→hypothetical_list_head = NULL;
    (*count)++;
    /* get the next token (it should be a result identifier, or the keyword ATTRIBUTE) */
    *token = Get_Token(in_stream, log_stream);
    result_pointer→next = NULL;
}

static result *
parse_results(
        file in_stream,
        file log_stream,
        token_details *token,
        cardinal *count)

/* Parses results, and returns a pointer to a list of results. The keyword RESULTS has just been
   read, and the details of the token that followed it are pointed to by token. Sets *count to
   the number of results in the area.
   EBNF: results           = results-header results-block.
         results-header    = "RESULTS".
         results-block     = result-identifier string
                             result-identifier string
                             { result-identifier string }.
         result-identifier = identifier.                          */
{
    result *result_head = NULL,
      *result_pointer = NULL,
      *temp_result_pointer;
    char message[Max_Error_Message_Length];
    while (token→token ≡ TK_IDENTIFIER) {
        if (result_head ≡ NULL) {
            /* allocate memory for this result (the first in the list) */
            if ((result_head = (result *) malloc(sizeof(result))) ≡ NULL)
                error_exit(log_stream, "malloc failed during result handling", token);
            result_pointer = result_head;
        } else {
/* go to the end of the list of results, checking that this result has not already been
   specified */
            for (temp_result_pointer = result_head; temp_result_pointer ≠ NULL;
                 temp_result_pointer = temp_result_pointer→next)
                if (¬strcmp(token→details.identifier, temp_result_pointer→identifier)) {
                    sprintf(message, "%s result already specified",
                            token→details.identifier);
                    error_exit(log_stream, message, token);
                }
```

```
    /* allocate memory for this result */

    if ((result_pointer→next = (result *) malloc(sizeof(result))) ≡ NULL)
        error_exit(log_stream, "malloc␣failed␣during␣result␣handling", token);

    result_pointer = result_pointer→next;
}
parse_result_pair(in_stream, log_stream, result_pointer, token, count);
    }
    return result_head;
}

static void
add_to_direction_list(
        file log_stream,
        direction_list_element **list_head,
        result *result_pointer,
        token_details *token)

/* Adds result_pointer to the list of directions pointed to by *list_head. */

{
    direction_list_element *list_pointer,
      *last_list_pointer;
    char message[Max_Error_Message_Length];

if (*list_head ≡ NULL) {

    /* allocate memory for this direction (the first in the list) */

    if ((*list_head =
                (direction_list_element *) malloc(sizeof(direction_list_element))) ≡
            NULL)
        error_exit(log_stream, "malloc␣failed␣during␣result␣list␣handling", token);

    list_pointer = *list_head;

} else {

    /* go to the end of the list of directions, checking that this result has not already been
       specified for this attribute value */

    for (list_pointer = *list_head; list_pointer ≠ NULL;
            list_pointer = list_pointer→next) {
        if (list_pointer→result ≡ result_pointer) {
            sprintf(message,
                    "%s␣result␣already␣specified␣for␣this␣attribute␣value",
                    token→details.identifier);
            error_exit(log_stream, message, token);
        }
        last_list_pointer = list_pointer;
    }
```

```
    /* allocate memory for this direction */

    if ((last_list_pointer→next =
                    (direction_list_element *) malloc(sizeof(direction_list_element))) ≡
            NULL)
        error_exit(log_stream, "malloc failed during result list handling", token);

    list_pointer = last_list_pointer→next;
}
    list_pointer→result = result_pointer;

    list_pointer→next = NULL;
}

static void
add_to_identifier_list(
        file log_stream,
        identifier_list_element **list_head,
        string identifier,
        token_details *token)

/* Adds identifier to the list of identifiers pointed to by *list_head. */

{
    identifier_list_element *list_pointer,
      *last_list_pointer;
    char message[Max_Error_Message_Length];

    if (*list_head ≡ NULL) {

        /* allocate memory for this identifier list element (the first in the list) */

        if ((*list_head = (identifier_list_element *) malloc(sizeof(identifier_list_element))) ≡
                NULL)
            error_exit(log_stream, "malloc failed during identifier list handling",
                    token);

        list_pointer = *list_head;

    } else {

        /* go to the end of the list of identifiers, checking that this identifier has not already
           been specified for this attribute value */

        for (list_pointer = *list_head; list_pointer ≠ NULL;
                list_pointer = list_pointer→next) {
            if (¬strcmp(list_pointer→identifier, identifier)) {
                sprintf(message,
                        "%s identifier already specified for this attribute value",
                        token→details.identifier);
                error_exit(log_stream, message, token);
            }
            last_list_pointer = list_pointer;
        }
```

```
/* allocate memory for this identifier list element */

if ((last_list_pointer→next =
                (identifier_list_element *) malloc(sizeof(identifier_list_element))) ≡
        NULL)
    error_exit(log_stream, "malloc␣failed␣during␣identifier␣list␣handling",
            token);

list_pointer = last_list_pointer→next;
}
identifier_list_pointer→identifier = identifier;

list_pointer→next = NULL;
}

static attribute *
parse_attributes(
        file in_stream,
        file log_stream,
        result *result_head,
        token_details *token,
        string area_identifier,
        cardinal *count)

/* Parses attributes, and returns a pointer to a list of attributes. The keyword ATTRIBUTE
   has just been read, and the details of the token that followed it are pointed to by token.
   *result_head is the head of the list of results for this area. Sets *count to the number of
   attributes in the area.
   EBNF: attribute          = attribute-header attribute-block.
         attribute-header   = "ATTRIBUTE".
         attribute-block    = local-attribute | external-attribute.
         local-attribute    = "QUESTION" string
                              [ "YES" string { result-identifier } ]
                              [ "NO" string { result-identifier } ]
                              [ "UNKNOWN" string { result-identifier } ]
                              [ "HELP" string ].
         external-attribute = "AREA" area-identifier
                              [ "YES" string { result-identifier }
                                [ "EXTERNAL" result-identifier { result-identifier } ] ]
                              [ "NO" string { result-identifier }
                                [ "EXTERNAL" result-identifier { result-identifier } ] ]
                              [ "UNKNOWN" string { result-identifier }
                                [ "EXTERNAL" result-identifier { result-identifier } ] ].   */

{
attribute *attribute_pointer;
result *result_pointer = result_head;
boolean found = FALSE;
char message[Max_Error_Message_Length];

/* allocate memory for this attribute */

if ((attribute_pointer = (attribute *) malloc(sizeof(attribute))) ≡ NULL)
    error_exit(log_stream, "malloc␣failed␣during␣attribute␣handling", token);
```

```
if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_QUESTION))
    attribute_pointer→external_attribute = FALSE;
else if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_AREA))
    attribute_pointer→external_attribute = TRUE;
else
    error_exit(log_stream, "keyword QUESTION or AREA expected in attribute",
            token);

/* get the next token (if the attribute is local, it should be a string; if the attribute is
    external, it should be an area identifier) */

*token = Get_Token(in_stream, log_stream);

if (attribute_pointer→external_attribute) {

    if (token→token ≠ TK_IDENTIFIER)
        error_exit(log_stream,
                "identifier expected in attribute after keyword AREA", token);

    if (¬strcmp(token→details.string, area_identifier)) {
        sprintf(message, "Recursive external attribute in %s area",
                area_identifier);
        error_exit(log_stream, message, token);
    }
    attribute_pointer→details.external.area_identifier = token→details.string;

} else {

    if (token→token ≠ TK_STRING)
        error_exit(log_stream,
                "string expected in attribute after keyword QUESTION", token);

    attribute_pointer→details.local.question = token→details.string;
}

/* get the next token (it should be the keyword YES, the keyword NO, or the keyword
    UNKNOWN) */

*token = Get_Token(in_stream, log_stream);

if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_YES)) {

    /* get the next token (it should be a string) */

    *token = Get_Token(in_stream, log_stream);

    if (token→token ≠ TK_STRING)
        error_exit(log_stream, "string expected in attribute after keyword YES",
                token);

    attribute_pointer→yes = token→details.string;

    /* get the next token (it should be a result identifier, the keyword NO, the keyword
        UNKNOWN, the keyword HELP, the keyword ATTRIBUTE, or the keyword CASE; if the
        attribute is external the token could also be the keyword EXTERNAL) */

    *token = Get_Token(in_stream, log_stream);

    attribute_pointer→yes_direction_head = NULL;
```

```
/* while there are result identifiers to parse ... */
while (token→token ≡ TK_IDENTIFIER) {
    /* find the result matching the result identifier, and add that result to the list of
       directions for YES for this attribute */
    do {
        if (result_pointer ≡ NULL) {
            sprintf(message, "%s␣result␣not␣found",
                    token→details.identifier);
            error_exit(log_stream, message, token);
        }
        found = ¬strcmp(token→details.identifier, result_pointer→identifier);
        if (found)
            add_to_direction_list(log_stream, &attribute_pointer→yes_direction_head,
                    result_pointer, token);
        else
            result_pointer = result_pointer→next;
    } while (¬found);

    result_pointer = result_head;

    /* get the next token (it should be a result identifier, the keyword NO, the keyword
       UNKNOWN, the keyword HELP, the keyword ATTRIBUTE, or the keyword CASE; if
       the attribute is external the token could also be the keyword EXTERNAL) */
    *token = Get_Token(in_stream, log_stream);
}

if (attribute_pointer→external_attribute) {

    attribute_pointer→details.external.yes_identifier_head = NULL;

    if ((token→token ≡ TK_KEYWORD) ∧
            (token→details.keyword ≡ KW_EXTERNAL)) {

        /* get the next token (it should be a result identifier) */

        *token = Get_Token(in_stream, log_stream);

        while (token→token ≡ TK_IDENTIFIER) {

            /* add the result identifier to the list of external result identifiers for YES for
               this attribute */

            add_to_identifier_list(log_stream,
                &attribute_pointer→details.external.yes_identifier_head,
                token→details.identifier, token);

            /* get the next token (it should be a result identifier, the keyword NO, the
               keyword UNKNOWN, the keyword HELP, the keyword ATTRIBUTE, or the
               keyword CASE) */

            *token = Get_Token(in_stream, log_stream);
        }
```

```
            if (attribute_pointer→details.external.yes_identifier_head ≡ NULL)
                error_exit(log_stream,
                        "identifier␣expected␣in␣attribute␣after␣keyword␣EXTERNAL",
                        token);
        }
    }
} else
    attribute_pointer→yes = NULL;
if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_NO)) {
    /* get the next token (it should be a string) */
    *token = Get_Token(in_stream, log_stream);
    if (token→token ≠ TK_STRING)
        error_exit(log_stream, "string␣expected␣in␣attribute␣after␣keyword␣NO",
                token);
    attribute_pointer→no = token→details.string;
    /* get the next token (it should be a result identifier, the keyword UNKNOWN, the keyword
       HELP, the keyword ATTRIBUTE, or the keyword CASE; if the attribute is external the
       token could also be the keyword EXTERNAL) */
    *token = Get_Token(in_stream, log_stream);
    attribute_pointer→no_direction_head = NULL;
    /* while there are result identifiers to parse ... */
    while (token→token ≡ TK_IDENTIFIER) {
        /* find the result matching the result identifier, and add that result to the list of
           directions for NO for this attribute */
        do {
            if (result_pointer ≡ NULL) {
                sprintf(message, "%s␣result␣not␣found",
                        token→details.identifier);
                error_exit(log_stream, message, token);
            }
            found = ¬strcmp(token→details.identifier, result_pointer→identifier);
            if (found)
                add_to_direction_list(log_stream, &attribute_pointer→no_direction_head,
                        result_pointer, token);
            else
                result_pointer = result_pointer→next;
        } while (¬found);
        result_pointer = result_head;
        /* get the next token (it should be a result identifier, the keyword UNKNOWN, the
           keyword HELP, the keyword ATTRIBUTE, or the keyword CASE; if the attribute is
           external the token could also be the keyword EXTERNAL) */
        *token = Get_Token(in_stream, log_stream);
    }
```

```
if (attribute_pointer→external_attribute) {
    attribute_pointer→details.external.no_identifier_head = NULL;
    if ((token→token ≡ TK_KEYWORD) ∧
            (token→details.keyword ≡ KW_EXTERNAL)) {
        /* get the next token (it should be a result identifier) */
        *token = Get_Token(in_stream, log_stream);
        while (token→token ≡ TK_IDENTIFIER) {
            /* add the result identifier to the list of external result identifiers for NO for
               this attribute */
            add_to_identifier_list(log_stream,
                    &attribute_pointer→details.external.no_identifier_head,
                    token→details.identifier, token);
            /* get the next token (it should be a result identifier, the keyword UNKNOWN,
               the keyword HELP, the keyword ATTRIBUTE, or the keyword CASE) */
            *token = Get_Token(in_stream, log_stream);
        }
        if (attribute_pointer→details.external.no_identifier_head ≡ NULL)
            error_exit(log_stream,
                    "identifier␣expected␣in␣attribute␣after␣keyword␣EXTERNAL",
                    token);
        }
    }
} else
    attribute_pointer→no = NULL;
if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_UNKNOWN)) {
    /* get the next token (it should be a string) */
    *token = Get_Token(in_stream, log_stream);
    if (token→token ≠ TK_STRING)
        error_exit(log_stream,
                "string␣expected␣in␣attribute␣after␣keyword␣UNKNOWN", token);
    attribute_pointer→unknown = token→details.string;
    /* get the next token (it should be a result identifier, the keyword HELP, the keyword
       ATTRIBUTE, or the keyword CASE; if the attribute is external the token could also be
       the keyword EXTERNAL) */
    *token = Get_Token(in_stream, log_stream);
    attribute_pointer→unknown_direction_head = NULL;
```

```
/* while there are result identifiers to parse ... */
while (token→token ≡ TK_IDENTIFIER) {
    /* find the result matching the result identifier, and add that result to the list of
       directions for UNKNOWN for this attribute */
    do {
        if (result_pointer ≡ NULL) {
            sprintf(message, "%s result not found",
                    token→details.identifier);
            error_exit(log_stream, message, token);
        }
        found = ¬strcmp(token→details.identifier, result_pointer→identifier);
        if (found)
            add_to_direction_list(log_stream, &attribute_pointer→unknown_direction_head,
                    result_pointer, token);
        else
            result_pointer = result_pointer→next;
    } while (¬found);
    result_pointer = result_head;
    /* get the next token (it should be a result identifier, the keyword HELP, the keyword
       ATTRIBUTE, or the keyword CASE; if the attribute is external the token could also
       be the keyword EXTERNAL) */
    *token = Get_Token(in_stream, log_stream);
}
if (attribute_pointer→external_attribute) {
    attribute_pointer→details.external.unknown_identifier_head = NULL;
    if ((token→token ≡ TK_KEYWORD) ∧
            (token→details.keyword ≡ KW_EXTERNAL)) {
        /* get the next token (it should be a result identifier) */
        *token = Get_Token(in_stream, log_stream);
        while (token→token ≡ TK_IDENTIFIER) {
            /* add the result identifier to the list of external result identifiers for
               UNKNOWN for this attribute */
            add_to_identifier_list(log_stream,
                    &attribute_pointer→details.external.unknown_identifier_head,
                    token→details.identifier, token);
            /* get the next token (it should be a result identifier, the keyword HELP, the
               keyword ATTRIBUTE, or the keyword CASE) */
            *token = Get_Token(in_stream, log_stream);
        }
        if (attribute_pointer→details.external.unknown_identifier_head ≡ NULL)
            error_exit(log_stream,
                    "identifier expected in attribute after keyword EXTERNAL",
                    token);
    }
}
```

```
} else
    attribute_pointer→unknown = NULL;

if ((attribute_pointer→yes ≡ NULL) ∧ (attribute_pointer→no ≡ NULL) ∧
        (attribute_pointer→unknown ≡ NULL))
    error_exit(log_stream, "keyword␣YES,␣NO␣or␣UNKNOWN␣expected␣in␣attribute",
            token);

if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_HELP)) {

    if (attribute_pointer→external_attribute)
        error_exit(log_stream, "keyword␣HELP␣not␣allowed␣in␣external␣attribute",
                token);

    /* get the next token (it should be a string) */

    *token = Get_Token(in_stream, log_stream);

    if (token→token ≠ TK_STRING)
        error_exit(log_stream, "string␣expected␣in␣attribute␣after␣keyword␣HELP",
                token);

    attribute_pointer→details.local.help = token→details.string;

    /* get the next token (it should be the keyword ATTRIBUTE, or the keyword CASE) */

    *token = Get_Token(in_stream, log_stream);

} else if (¬attribute_pointer→external_attribute)
    attribute_pointer→details.local.help = NULL;

attribute_pointer→number = ++(*count);
attribute_pointer→matrix_head = NULL;
attribute_pointer→weights_head = NULL;
attribute_pointer→probability_head = NULL;

if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_ATTRIBUTE)) {

    /* get the next token (it should be the keyword QUESTION, or the keyword AREA) */

    *token = Get_Token(in_stream, log_stream);

    /* parse the next attribute */

        attribute_pointer→next = parse_attributes(in_stream, log_stream, result_head, token,
                area_identifier, count);

    } else
        attribute_pointer→next = NULL;

    return attribute_pointer;
}
```

```
static boolean
is_more_important(
        kase *x,
        kase *y)

/* Returns TRUE, iff case x is more important than case y: i.e. case x was a decision of a
   more important court, or of an equally important court at a later date (if neither case x
   nor case y has a court then the more recent of the two is the more important). */
{
    if (x→court_string ≡ NULL)
        if (y→court_string ≡ NULL)

            /* neither case x nor case y has a court, so return TRUE if case x is a more recent
               decision than case y */

            return x→year > y→year;

        else

            /* case y has a court and case x doesn't, so case y is assumed to be more important
               than case x */

            return FALSE;

    else if (y→court_string ≡ NULL)

        /* case x has a court and case y doesn't, so case x is assumed to be more important
           than case y */

        return TRUE;

    else {

        /* both case x and case y have a court */

        if (x→court_rank < y→court_rank)

            /* case x was a decision of a more important court then was case y */

            return TRUE;

        else if (x→court_rank ≡ y→court_rank)

            /* case x and case y were decisions of an equally important court, so return TRUE
               if case x is a more recent decision than case y */

            return x→year > y→year;

        else

            /* case x was a decision of a less important court than was case y */

            return FALSE;
    }
}
```

```
static void
rank_cases(
        kase **case_head)

/* Reorders the list of cases pointed to by *case_head so that the cases are listed in order of
   their importance (more important cases first). */

{
    boolean changed;
    kase *case_pointer,
      *previous_case_pointer,
      *next_case_pointer,
      *temp_case_pointer;

    do {
        changed = FALSE;
        previous_case_pointer = NULL;
        case_pointer = *case_head;

        /* while there are still cases in the list ... */

        while (case_pointer ≠ NULL) {

            next_case_pointer = case_pointer→next;

            if (next_case_pointer ≠ NULL) {

                if (is_more_important(next_case_pointer, case_pointer)) {

/* swap this case and the next */

if (previous_case_pointer ≡ NULL)

    /* case_pointer points to the first case in the list */

    *case_head = next_case_pointer;

else {

    /* case_pointer points to a case which is not the first in the list */

    previous_case_pointer→next = next_case_pointer;
}

case_pointer→next = next_case_pointer→next;
next_case_pointer→next = case_pointer;
temp_case_pointer = case_pointer;
case_pointer = next_case_pointer;
next_case_pointer = temp_case_pointer;
                    changed = TRUE;
                }
            }
            previous_case_pointer = case_pointer;
            case_pointer = next_case_pointer;
        }
    } while (changed);
}
```

```
static void
number_cases(
        result *result_head)
/* Assigns a number to each case for each result in the list pointed to by result_head. */
{
    result *result_pointer;
    kase *case_pointer;
    cardinal count = 1;
    /* for every result ... */
    for (result_pointer = result_head; result_pointer ≠ NULL; result_pointer =
            result_pointer→next)
        /* for every case ... */
        for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
                case_pointer = case_pointer→next)
            case_pointer→number = count++;
}

static void
cross_link(
        result *result_head,
        attribute *attribute_head)
/* Links attribute values by attribute (they are already linked by case). */
{
    result *result_pointer;
    kase *case_pointer;
    attribute *attribute_pointer;
    matrix_element *case_matrix_pointer,
      *attribute_matrix_pointer;
    /* for every result ... */
    for (result_pointer = result_head; result_pointer ≠ NULL; result_pointer =
            result_pointer→next)
        /* for every case ... */
        for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
                case_pointer = case_pointer→next) {
            attribute_pointer = attribute_head;
            case_matrix_pointer = case_pointer→matrix_head;
            if (attribute_pointer→matrix_head ≡ NULL)
                /* this is the first attribute value for this (or any other) attribute, so each
                   attribute value for this case becomes the head of the appropriate attribute's
                   list */
            while ((attribute_pointer ≠ NULL) ∧ (case_matrix_pointer ≠ NULL)) {
                attribute_pointer→matrix_head = case_matrix_pointer;
                case_matrix_pointer = case_matrix_pointer→case_next;
                attribute_pointer = attribute_pointer→next;
            }
```

```
        else

            /* this is not the first attribute value for this attribute, so add each attribute
               value for this case to the end of the appropriate attribute's list */

            while (attribute_pointer ≠ NULL) {

                for (attribute_matrix_pointer = attribute_pointer→matrix_head;
                        attribute_matrix_pointer→attribute_next ≠ NULL;
                        attribute_matrix_pointer =
                        attribute_matrix_pointer→attribute_next);

                    attribute_matrix_pointer→attribute_next = case_matrix_pointer;
                    case_matrix_pointer = case_matrix_pointer→case_next;
                    attribute_pointer = attribute_pointer→next;
                }
        }
}

static void
check_for_identical_cases(
        file log_stream,
        area *area_pointer)

/* Checks every case in the area_pointer area against every other case in that area and warns
   if two cases are identical, or identical but for UNKNOWN values. */

{
    result *result_pointer_X,
      *result_pointer_Y;
    kase *case_pointer_X,
      *case_pointer_Y;
    matrix_element *matrix_pointer_X,
      *matrix_pointer_Y;
    boolean identical,
       possibly_identical;
    char message[Max_Error_Message_Length];

    /* for every result ... */

    for (result_pointer_X = area_pointer→result_head; result_pointer_X ≠ NULL;
        result_pointer_X = result_pointer_X→next)

/* for every case X ... */

for (case_pointer_X = result_pointer_X→case_head; case_pointer_X ≠ NULL;
        case_pointer_X = case_pointer_X→next) {

    case_pointer_Y = case_pointer_X;
    result_pointer_Y = result_pointer_X;
```

```
/* for every case Y (i.e. every case after case X) ... */
while (case_pointer_Y ≠ NULL) {
    if (case_pointer_X ≠ case_pointer_Y) {
        /* X and Y are not the same case */
        identical = TRUE;
        possibly_identical = TRUE;
        matrix_pointer_X = case_pointer_X→matrix_head;
        matrix_pointer_Y = case_pointer_Y→matrix_head;
        while (possibly_identical ∧
                (matrix_pointer_X ≠ NULL) ∧ (matrix_pointer_Y ≠ NULL)) {
            /* look for differences between case X and case Y */
            if (matrix_pointer_X→attribute_value ≠
                    matrix_pointer_Y→attribute_value) {
                identical = FALSE;
                if ((matrix_pointer_X→attribute_value ≠ UNKNOWN) ∧
                        (matrix_pointer_Y→attribute_value ≠ UNKNOWN))
                    possibly_identical = FALSE;
            }
            matrix_pointer_X = matrix_pointer_X→case_next;
            matrix_pointer_Y = matrix_pointer_Y→case_next;
        }
        if (identical) {
            sprintf(message,
                    "C%u␣and␣C%u␣in␣%s␣area␣have␣"
                    "identical␣attribute␣vectors",
                    case_pointer_X→number, case_pointer_Y→number,
                    area_pointer→identifier);
            if (result_pointer_X ≠ result_pointer_Y)
                sprintf(message, "%s␣and␣different␣results", message);
            warning(log_stream, message);
        } else if (possibly_identical) {
            sprintf(message,
                    "C%u␣and␣C%u␣in␣%s␣area␣have␣"
                    "identical␣attribute␣values␣(except␣for␣unknowns)",
                    case_pointer_X→number, case_pointer_Y→number,
                    area_pointer→identifier);
            if (result_pointer_X ≠ result_pointer_Y)
                sprintf(message, "%s␣and␣different␣results", message);
            warning(log_stream, message);
        }
    }
    case_pointer_Y = case_pointer_Y→next;
```

```
        while ((case_pointer_Y ≡ NULL) ∧ (result_pointer_Y ≠ NULL)) {
            /* the next case Y is not of this result, so move to the next result */
            result_pointer_Y = result_pointer_Y→next;
            if (result_pointer_Y ≠ NULL)
                case_pointer_Y = result_pointer_Y→case_head;
        }
    }
}
```

static void
parse_case(
    file in_stream,
    file log_stream,
    *court *court_pointer*,
    *area *area_pointer*,
    *token_details *token*,
    *cardinal *count*)

/* Parses a case, and adds it to the list of cases for the appropriate result in the area pointed to by *area_pointer*. The keyword `CASE` has just been read, and the details of the token that followed it are pointed to by *token*. Increments **count* (the number of cases already parsed in this area).

```
EBNF: case         = case-header case-block.
      case-header  = "CASE" string [string].
      case-block   = "CITATION" string
                     "YEAR" year
                     ["COURT" court-identifier]
                     "FACTS" attribute-vector
                     "RESULT" result-identifier
                     ["SUMMARY" string].                                    */
```

```
{
    result *result_pointer = area_pointer→result_head;
    kase *case_pointer,
      *temp_case_pointer;
    attribute *attribute_pointer;
    matrix_element *matrix_pointer;
    boolean found = FALSE;
    char message[Max_Error_Message_Length];

    if (token→token ≠ TK_STRING)
        error_exit(log_stream, "string␣expected␣in␣case␣after␣keyword␣CASE", token);
    /* allocate memory for this case */
    if ((case_pointer = (kase *) malloc(sizeof(kase))) ≡ NULL)
        error_exit(log_stream, "malloc␣failed␣during␣case␣handling", token);
    case_pointer→name = token→details.string;
    /* get the next token (it should be a string or the keyword CITATION) */
    *token = Get_Token(in_stream, log_stream);
```

```
if (token→token ≡ TK_STRING) {
    case_pointer→short_name = token→details.string;
    /* get the next token (it should be the keyword CITATION) */
    *token = Get_Token(in_stream, log_stream);
} else
    case_pointer→short_name = case_pointer→name;
if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_CITATION))
    error_exit(log_stream, "keyword␣CITATION␣expected␣in␣case", token);
/* get the next token (it should be a string) */
*token = Get_Token(in_stream, log_stream);
if (token→token ≠ TK_STRING)
    error_exit(log_stream, "string␣expected␣in␣case␣after␣keyword␣CITATION",
        token);
case_pointer→citation = token→details.string;
/* get the next token (it should be the keyword YEAR) */
*token = Get_Token(in_stream, log_stream);
if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_YEAR))
    error_exit(log_stream, "keyword␣YEAR␣expected␣in␣case", token);
/* get the next token (it should be a year) */
*token = Get_Token(in_stream, log_stream);
if (token→token ≠ TK_YEAR)
    error_exit(log_stream, "year␣expected␣in␣case␣after␣keyword␣YEAR", token);
case_pointer→year = token→details.year;
/* get the next token (it should be the keyword COURT, or the keyword FACTS) */
*token = Get_Token(in_stream, log_stream);
if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_COURT)) {
    /* get the next token (it should be a court identifier) */
    *token = Get_Token(in_stream, log_stream);
    if (token→token ≠ TK_IDENTIFIER)
        error_exit(log_stream, "identifier␣expected␣in␣case␣after␣keyword␣COURT",
            token);
    /* find the court identifier in the list of courts, and link the case to that court */
    do {
        if (court_pointer ≡ NULL) {
            sprintf(message, "%s␣court␣not␣found", token→details.identifier);
            error_exit(log_stream, message, token);
        }
```

```
        found = ¬strcmp(token→details.identifier, court_pointer→identifier);
        if (found) {
            case_pointer→court_string = court_pointer→string;

            case_pointer→court_rank = court_pointer→rank;
        } else
            court_pointer = court_pointer→next;
    } while (¬found);
    /* get the next token (it should be the keyword FACTS) */

    *token = Get_Token(in_stream, log_stream);

} else

    /* no court specified */

    case_pointer→court_string = NULL;

if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_FACTS))
    error_exit(log_stream, "keyword␣FACTS␣expected␣in␣case", token);

/* get the next token (it should be an attribute vector) */

*token = Get_Token(in_stream, log_stream);

if (token→token ≠ TK_ATTRIBUTE_VECTOR)
    error_exit(log_stream,
            "attribute␣vector␣expected␣in␣case␣after␣keyword␣FACTS", token);

case_pointer→matrix_head = token→details.matrix_head;

/* check that there are as many values in the attribute vector as there are attributes */

matrix_pointer = case_pointer→matrix_head;
for (attribute_pointer = area_pointer→attribute_head;
        (attribute_pointer ≠ NULL) ∧ (matrix_pointer ≠ NULL);
        attribute_pointer = attribute_pointer→next)
    matrix_pointer = matrix_pointer→case_next;
if (attribute_pointer ≠ NULL)
    error_exit(log_stream, "too␣few␣values␣in␣attribute␣vector", token);
if (matrix_pointer ≠ NULL)
    error_exit(log_stream, "too␣many␣values␣in␣attribute␣vector", token);

/* get the next token (it should be the keyword RESULT) */

*token = Get_Token(in_stream, log_stream);

if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_RESULT))
    error_exit(log_stream, "keyword␣RESULT␣expected␣in␣case", token);

/* get the next token (it should be a result identifier) */

*token = Get_Token(in_stream, log_stream);

if (token→token ≠ TK_IDENTIFIER)
    error_exit(log_stream, "identifier␣expected␣in␣case␣after␣keyword␣RESULT",
            token);
```

```
/* find the result identifier in the list of results, and add the case to the list of cases for
   that result */

do {
    if (result_pointer ≡ NULL) {
        sprintf(message, "%s result not found", token→details.identifier);
        error_exit(log_stream, message, token);
    }
    found = ¬strcmp(token→details.identifier, result_pointer→identifier);
    if (found) {
        if (result_pointer→case_head ≡ NULL)
            result_pointer→case_head = case_pointer;
        else {
            for (temp_case_pointer = result_pointer→case_head;
                    temp_case_pointer→next ≠ NULL;
                    temp_case_pointer = temp_case_pointer→next);
            temp_case_pointer→next = case_pointer;
        }
    } else
        result_pointer = result_pointer→next;
} while (¬found);

(*count)++;

case_pointer→summary = NULL;

/* get the next token (it should be the keyword SUMMARY, the keyword CASE, the keyword
   IDEAL, the keyword AREA, or the end of the file) */

*token = Get_Token(in_stream, log_stream);

if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_SUMMARY)) {

    /* get the next token (it should be a string) */

    *token = Get_Token(in_stream, log_stream);

    if (token→token ≠ TK_STRING)
        error_exit(log_stream, "string expected in case after keyword SUMMARY",
                token);

    case_pointer→summary = token→details.string;

        /* get the next token (it should be the keyword CASE, the keyword IDEAL, the keyword
           AREA, or the end of the file) */

        *token = Get_Token(in_stream, log_stream);
    }
    case_pointer→next = NULL;
}
```

```
static vector_element *
vector_from_matrix(
        file log_stream,
        matrix_element *matrix_pointer,
        token_details *token)

/* Returns a pointer to a new list of vector elements whose values correspond to those in the
   list of matrix elements pointed to by matrix_pointer (linked by case). Frees the memory
   taken up by the list of matrix elements. */

{
    vector_element *vector_head,
      *vector_pointer;
    matrix_element *next_matrix_pointer;

    /* allocate memory for the first vector element in the list */

    if ((vector_head = (vector_element *) malloc(sizeof(vector_element))) ≡ NULL)
        error_exit(log_stream, "malloc␣failed␣during␣matrix/vector␣conversion", token);

    vector_pointer = vector_head;

    /* while there are still matrix elements in the list ... */

    while (matrix_pointer ≠ NULL) {

        /* copy the attribute value into the vector element */

        vector_pointer→attribute_value = matrix_pointer→attribute_value;
        next_matrix_pointer = matrix_pointer→case_next;

        /* free the memory taken up by the matrix element */

        free(matrix_pointer);
        if (next_matrix_pointer ≡ NULL)
            vector_pointer→next = NULL;
        else {

            /* allocate memory for the next vector element */

            if ((vector_pointer→next = (vector_element *) malloc(sizeof(vector_element))) ≡
                    NULL)
                error_exit(log_stream, "malloc␣failed␣during␣matrix/vector␣conversion",
                        token);

            vector_pointer = vector_pointer→next;
        }
        matrix_pointer = next_matrix_pointer;
    }
    return vector_head;
}
```

```
static void
parse_ideal_point(
        file in_stream,
        file log_stream,
        result *result_pointer,
        attribute *attribute_pointer,
        token_details *token,
        cardinal *count)

/* Parses an ideal point, and makes it the ideal point for the appropriate result in the list of
   results pointed to by result_pointer. The keyword IDEAL has just been read, and the details
   of the token that followed it are pointed to by token. *attribute_pointer is the head of the
   list of attributes for this area. Increments *count (the number of ideal points already parsed
   in this area).

   EBNF: ideal-point         = ideal-point-header ideal-point-block.
         ideal-point-header  = "IDEAL".
         ideal-point-block   = "FACTS" attribute-vector
                               "RESULT" result-identifier.                              */

{
    vector_element *temp_vector_head;
    matrix_element *matrix_pointer;
    boolean found = FALSE;
    char message[Max_Error_Message_Length];

    if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_FACTS))
        error_exit(log_stream, "keyword␣FACTS␣expected␣in␣ideal␣point", token);

    /* get the next token (it should be an attribute vector) */

    *token = Get_Token(in_stream, log_stream);

    if (token→token ≠ TK_ATTRIBUTE_VECTOR)
        error_exit(log_stream,
                "attribute␣vector␣expected␣in␣ideal␣point␣"
                "after␣keyword␣FACTS", token);

    /* check that there are as many values in the attribute vector as there are attributes */

    matrix_pointer = token→details.matrix_head;
    for (;
            (attribute_pointer ≠ NULL) ∧ (matrix_pointer ≠ NULL);
            attribute_pointer = attribute_pointer→next)
        matrix_pointer = matrix_pointer→case_next;
    if (attribute_pointer ≠ NULL)
        error_exit(log_stream, "too␣few␣values␣in␣attribute␣vector", token);
    if (matrix_pointer ≠ NULL)
        error_exit(log_stream, "too␣many␣values␣in␣attribute␣vector", token);
```

```
    /* convert the attribute vector from a list of matrix elements into a list of vector
       elements */

    temp_vector_head = vector_from_matrix(log_stream, token→details.matrix_head, token);

    /* get the next token (it should be the keyword RESULT) */

    *token = Get_Token(in_stream, log_stream);

    if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_RESULT))
        error_exit(log_stream, "keyword␣RESULT␣expected␣in␣ideal␣point", token);

    /* get the next token (it should be a result identifier) */

    *token = Get_Token(in_stream, log_stream);

    if (token→token ≠ TK_IDENTIFIER)
        error_exit(log_stream,
                "identifier␣expected␣in␣ideal␣point␣after␣keyword␣RESULT", token);

    /* find the result identifier in the list of results, and link that result to the ideal point */

    do {
        if (result_pointer ≡ NULL) {
            sprintf(message, "%s␣result␣not␣found", token→details.identifier);
            error_exit(log_stream, message, token);
        }
        found = ¬strcmp(token→details.identifier, result_pointer→identifier);
        if (found) {
            if (result_pointer→ideal_point_head ≡ NULL)
                result_pointer→ideal_point_head = temp_vector_head;
            else {
                sprintf(message,
                        "ideal␣point␣for␣%s␣result␣already␣specified",
                        token→details.identifier);
                error_exit(log_stream, message, token);
            }
        } else
            result_pointer = result_pointer→next;
    } while (¬found);

    (*count)++;

    /* get the next token (it should be the keyword IDEAL, the keyword AREA, or the end of
       the file) */

    *token = Get_Token(in_stream, log_stream);
}
```

```
static void
parse_area_block(
        file in_stream,
        file log_stream,
        area *area_pointer,
        token_details *token,
        court *court_head)

/* Parses an area block, and puts details in the area pointed to by area_pointer. The keyword
   AREA and an area identifier have just been read; the identifier's details are pointed to by
   token.
   EBNF: area            = area-header area-block.
         area-header     = "AREA" area-identifier.
         area-block      = [opening] [closing]
                           results
                           attribute {attribute}
                           case {case}
                           {ideal-point}.
         area-identifier = identifier.
         opening         = "OPENING" string.
         closing         = "CLOSING" string.                                          */

{
    cardinal number_of_cases = 0,
        number_of_ideal_points = 0;
    result *result_pointer;

    area_pointer→identifier = token→details.identifier;

    Indent(log_stream, 1);
    fprintf(log_stream, "%s area:\n\n", area_pointer→identifier);

    /* get the next token (it should be the keyword OPENING, the keyword CLOSING, or the
       keyword RESULTS) */

    *token = Get_Token(in_stream, log_stream);

    if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_OPENING)) {

        /* get the next token (it should be a string) */

        *token = Get_Token(in_stream, log_stream);

        if (token→token ≠ TK_STRING)
            error_exit(log_stream, "string expected after keyword OPENING", token);

        area_pointer→opening = token→details.string;

        /* get the next token (it should be the keyword CLOSING, or the keyword RESULTS) */

        *token = Get_Token(in_stream, log_stream);
    } else
        area_pointer→opening = NULL;
```

```
if ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_CLOSING)) {
    /* get the next token (it should be a string) */
    *token = Get_Token(in_stream, log_stream);
    if (token→token ≠ TK_STRING)
        error_exit(log_stream, "string expected after keyword CLOSING", token);
    area_pointer→closing = token→details.string;
    /* get the next token (it should be the keyword RESULTS) */
    *token = Get_Token(in_stream, log_stream);
} else
    area_pointer→closing = NULL;
if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_RESULTS))
    error_exit(log_stream, "keyword RESULTS expected in results header", token);
/* get the next token (it should be a result identifier) */
*token = Get_Token(in_stream, log_stream);
area_pointer→number_of_results = 0;
area_pointer→result_head = parse_results(in_stream, log_stream, token,
        &area_pointer→number_of_results);
switch (area_pointer→number_of_results) {
    case 0:
        error_exit(log_stream,
                "no results (at least two are required)", NULL);
        break;
    case 1:
        error_exit(log_stream,
                "only one result (at least two are required)", NULL);
        break;
    default:
        Indent(log_stream, 2);
        fprintf(log_stream,
                "%u results\n", area_pointer→number_of_results);
        break;
}
if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_ATTRIBUTE))
    error_exit(log_stream, "keyword ATTRIBUTE expected in attribute header",
            token);
/* get the next token (it should be the keyword QUESTION, or the keyword AREA) */
*token = Get_Token(in_stream, log_stream);
area_pointer→number_of_attributes = 0;
area_pointer→attribute_head = parse_attributes(in_stream, log_stream,
        area_pointer→result_head, token, area_pointer→identifier,
        &area_pointer→number_of_attributes);
```

```
Indent(log_stream, 2);
fprintf(log_stream, "%u attribute%s\n", area_pointer→number_of_attributes,
        area_pointer→number_of_attributes ≡ 1 ? Empty_String : "s");

if ((token→token ≠ TK_KEYWORD) ∨ (token→details.keyword ≠ KW_CASE))
    error_exit(log_stream, "keyword CASE expected in case header", token);

while ((token→token ≡ TK_KEYWORD) ∧ (token→details.keyword ≡ KW_CASE)) {

    /* get the next token (it should be a string) */

    *token = Get_Token(in_stream, log_stream);

    parse_case(in_stream, log_stream, court_head, area_pointer, token, &number_of_cases);
}
Indent(log_stream, 2);
fprintf(log_stream, "%u case%s\n", number_of_cases,
        number_of_cases ≡ 1 ? Empty_String : "s");

for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
        result_pointer = result_pointer→next)
    rank_cases(&result_pointer→case_head);

number_cases(area_pointer→result_head);

cross_link(area_pointer→result_head, area_pointer→attribute_head);

while ((token→token ≡ TK_KEYWORD) ∧
        (token→details.keyword ≡ KW_IDEAL)) {

    /* get the next token (it should be the keyword FACTS) */

    *token = Get_Token(in_stream, log_stream);

    parse_ideal_point(in_stream, log_stream, area_pointer→result_head,
            area_pointer→attribute_head, token, &number_of_ideal_points);
}

if (number_of_ideal_points ≠ 0) {
    Indent(log_stream, 2);
    fprintf(log_stream, "%u ideal point%s\n", number_of_ideal_points,
            number_of_ideal_points ≡ 1 ? Empty_String : "s");
    }
    fprintf(log_stream, "\n");
    area_pointer→correlation_coefficients = FALSE;
    area_pointer→next = NULL;

    check_for_identical_cases(log_stream, area_pointer);
}
```

```
static area *
parse_areas(
        file in_stream,
        file log_stream,
        court *court_head)
/* Parses areas, and returns a pointer to a list of areas. The keyword AREA has just been
   read. */
{
  area *area_head = NULL,
    *area_pointer = NULL,
    *temp_area_pointer;
  token_details token;
  char message[Max_Error_Message_Length];
  do {
    /* get the next token (it should be an area identifier) */
    token = Get_Token(in_stream, log_stream);
    if (token.token ≠ TK_IDENTIFIER)
      error_exit(log_stream,
            "identifier␣expected␣in␣area␣header␣after␣keyword␣AREA", &token);
    if (area_head ≡ NULL) {
      /* allocate memory for this area (the first in the list) */
      if ((area_head = (area *) malloc(sizeof(area))) ≡ NULL)
        error_exit(log_stream, "malloc␣failed␣during␣area␣handling", &token);
      area_pointer = area_head;
    } else {
      /* go to the end of the list of areas, checking that an area with this identifier has
         not already been specified */
      for (temp_area_pointer = area_head; temp_area_pointer ≠ NULL;
            temp_area_pointer = temp_area_pointer→next)
        if (¬strcmp(token.details.identifier, temp_area_pointer→identifier)) {
          sprintf(message, "%s␣area␣already␣specified",
                token.details.identifier);
          error_exit(log_stream, message, &token);
        }
      /* allocate memory for this area */
      if ((area_pointer→next = (area *) malloc(sizeof(area))) ≡ NULL)
        error_exit(log_stream, "malloc␣failed␣during␣area␣handling", &token);
      area_pointer = area_pointer→next;
    }
    parse_area_block(in_stream, log_stream, area_pointer, &token, court_head);
  } while ((token.token ≡ TK_KEYWORD) ∧ (token.details.keyword ≡ KW_AREA));
  if (token.token ≠ TK_EOF)
    error_exit(log_stream, "end␣of␣file␣expected", &token);
  return area_head;
}
```

```
extern case_law_specification
Parse_Specification(
        file in_stream,
        file log_stream)

/* Parses the specification file in_stream, and returns a case law specification.
   EBNF: specification = [hierarchy]
                         area {area}.                                          */
{
    case_law_specification case_law;
    token_details token;
    cardinal number_of_courts = 0;

    /* get the first token */

    token = Get_Token(in_stream, log_stream);

    if ((token.token ≡ TK_KEYWORD) ∧ (token.details.keyword ≡ KW_HIERARCHY)) {

        /* get the next token (it should be a court identifier) */

        token = Get_Token(in_stream, log_stream);

        case_law.court_head = parse_hierarchy(in_stream, log_stream, &token,
                &number_of_courts);

        if (case_law.court_head ≡ NULL)
            error_exit(log_stream,
                    "identifier␣expected␣in␣hierarchy␣block␣"
                    "after␣keyword␣HIERARCHY", &token);
        else {
            Indent(log_stream, 1);
            fprintf(log_stream, "%u␣court%s␣in␣the␣hierarchy.\n\n",
                    number_of_courts, number_of_courts ≡ 1 ? Empty_String : "s");
        }

    } else
        case_law.court_head = NULL;

    if ((token.token ≠ TK_KEYWORD) ∨ (token.details.keyword ≠ KW_AREA))
        error_exit(log_stream, "keyword␣AREA␣expected␣in␣area␣header", &token);

    case_law.area_head = parse_areas(in_stream, log_stream, case_law.court_head);

    return case_law;
}
```

6

The DUMPER module

dumper.h

/* This is the header file for the DUMPER module. It is also included by the CASES, ODOMETER and REPORTER modules. */

/* external functions */

extern void
Write_Matrix(
file stream,
*area *area_pointer*,
*vector_element *facts_head*,
*court *court_head*,
boolean hypothetical,
cardinal number);

extern void
Write_Year_and_Court(
file stream,
*kase *case_pointer*,
cardinal level);

extern void
Dump_Specification(
file dump_stream,
file log_stream,
case_law_specification case_law,
boolean inputable_latex,
boolean verbose);

dumper.c

```
/* This is the implementation file for the DUMPER module. */

#include ⟨stdio.h⟩
#include "shyster.h"
#include "cases.h"
#include "dumper.h"
#include "odometer.h"

static void
warning(
        file stream,
        const string message)
{
    Write_Warning_Message(stream, "Dumper", message, Top_Level);
}

static void
write_hierarchy_table(
        file dump_stream,
        court *court_pointer)

/* Writes a table of courts, with their ranks */

{
    boolean same_rank = FALSE;

    fprintf(dump_stream, "%s{Hierarchy}\n\n", Heading);
    Indent(dump_stream, 1);
    fprintf(dump_stream, "\\begin{small}\n");
    Indent(dump_stream, 2);
    fprintf(dump_stream, "\\begin{trivlist}\\item[]\n");
    Indent(dump_stream, 3);
    fprintf(dump_stream, "\\begin{tabular}{|r|l|}\\hline\n");
    Indent(dump_stream, 4);
    fprintf(dump_stream,
            "\\multicolumn{1}{|c|}{$c$}&"
            "\\multicolumn{1}{c|}{\\it␣Court\\/}\\\\\\hline\\hline");

/* while there are still courts to list ... */

    while (court_pointer ≠ NULL) {
        fprintf(dump_stream, "\n");
        Indent(dump_stream, 4);
        if (¬same_rank)
            fprintf(dump_stream, "%u", court_pointer→rank);
        fprintf(dump_stream, "&%s\\\\", court_pointer→string);
        if (court_pointer→next ≠ NULL)
            same_rank = (court_pointer→rank ≡ court_pointer→next→rank);
        court_pointer = court_pointer→next;
    }
```

```
    fprintf(dump_stream, "\\hline\n");
    Indent(dump_stream, 3);
    fprintf(dump_stream, "\\end{tabular}\n");
    Indent(dump_stream, 2);
    fprintf(dump_stream, "\\end{trivlist}\n");
    Indent(dump_stream, 1);
    fprintf(dump_stream, "\\end{small}\n\n");
}

static void
write_distance(
        file stream,
        distance_subtype distance,
        boolean centre,
        boolean rule_at_right)

/* Writes distance as a cell in a table of distances, centred (if centre is TRUE) and with a
   vertical rule at the right (if rule_at_right and centre are both TRUE). */

{
    if (distance.infinite ≠ 0) {

        if (Is_Zero(distance.finite)) {

            /* the distance has an infinite component, but no finite component */

            if (distance.infinite ≡ 1)
                if (centre)
                    fprintf(stream, "\\multicolumn{1}{c%s}{$\\infty$}",
                            rule_at_right ? "|" : "");
                else
                    fprintf(stream, "$\\infty$");
            else if (centre)
                fprintf(stream, "\\multicolumn{1}{c%s}{$%u\\infty$}",
                        rule_at_right ? "|" : "", distance.infinite);
            else
                fprintf(stream, "$%u\\infty$", distance.infinite);

        } else {

            /* the distance has both a finite component and an infinite component */

            if (distance.infinite ≡ 1) {
                fprintf(stream, "$\\infty$+");
                Write_Floating_Point(stream, distance.finite, Empty_String);
            } else {
                fprintf(stream, "$%u\\infty$+", distance.infinite);
                Write_Floating_Point(stream, distance.finite, Empty_String);
            }
        }
```

```
    } else {
        if (Is_Zero(distance.finite)) {
            /* the distance has neither a finite component nor an infinite component */
            if (centre)
                fprintf(stream, "\\multicolumn{1}{c%s}{--}", rule_at_right ? "|" : "");
            else
                fprintf(stream, "--");
        } else
            /* the distance has a finite component, but no infinite component */
            Write_Floating_Point(stream, distance.finite, Empty_String);
    }
}

static void
write_metrics(
        file stream,
        metrics_type metrics,
        boolean weighted_association_coefficient,
        boolean correlation_coefficients)

/* Writes, as cells in a table of distances, each of the metrics in metrics: the known distance d_K,
   the unknown distance d_U, the unweighted distance Δ, the association coefficient S, the
   weighted association coefficient S' (if weighted_association_coefficient is TRUE), the cor-
   relation coefficient r (if correlation_coefficients is TRUE), and the weighted correlation
   coefficient r' (if correlation_coefficients is TRUE). */

{
    write_distance(stream, metrics.distance.known, TRUE, FALSE);
    fprintf(stream, "&");
    write_distance(stream, metrics.distance.unknown, TRUE, TRUE);
    fprintf(stream, "&");

    if (metrics.number_of_known_pairs ≡ 0) {
    /* there are no known pairs, so write zeroes ("–") for Δ, S and S', and nothing for r
       and r' */

    fprintf(stream, "--&--&");
    if (weighted_association_coefficient)
        fprintf(stream, "--&");
    if (correlation_coefficients)
        fprintf(stream, "&&");
```

```
} else {
    if (metrics.number_of_known_differences ≡ 0) {
        /* there are no known differences, so write zeroes ("–") for Δ, S and S′ */
        fprintf(stream, "--&--&");
        if (weighted_association_coefficient)
            fprintf(stream, "--&");
    } else {
        /* there are known differences, so write values for Δ, S and S′ */
        fprintf(stream, "%u&", metrics.number_of_known_differences);
        Write_Floating_Point(stream,
                (floating_point) metrics.number_of_known_differences /
                metrics.number_of_known_pairs, Empty_String);
        fprintf(stream, "&");
        if (weighted_association_coefficient) {
            Write_Floating_Point(stream,
                    metrics.weighted_association_coefficient, Empty_String);
            fprintf(stream, "&");
        }
    }
    if (correlation_coefficients)
        if (metrics.correlation_coefficient.meaningless)
            /* either this case (or ideal point or centroid) or the instant case has all attribute
               values equal: the correlation coefficients r and r′ are meaningless */
            fprintf(stream, "&&");
        else {
            /* write values for r and r′ */
            Write_Floating_Point(stream,
                    metrics.correlation_coefficient.unweighted, Empty_String);
            fprintf(stream, "&");
            Write_Floating_Point(stream,
                    metrics.correlation_coefficient.weighted, Empty_String);
            fprintf(stream, "&");
        }
    }
}
```

```
static boolean
all_three_equal(
        distance_subtype x,
        distance_subtype y,
        distance_subtype z)

/* Returns TRUE, iff distances x, y and z are equal. */

{
    return ((x.infinite ≡ y.infinite) ∧
            Is_Equal(x.finite, y.finite, Distance_Precision) ∧
            (x.infinite ≡ z.infinite) ∧
            Is_Equal(x.finite, z.finite, Distance_Precision));
}

static void
write_result(
        file stream,
        cardinal count,
        cardinal first_result_row,
        boolean write_directions,
        result *result_pointer)

/* Writes the identifier, and the strength of every non-zero direction (if write_directions is
   TRUE), for the result pointed to by result_pointer, as cells in a table of distances. Writes
   the identifier, then the directions, in consecutive rows of the "Result" column, starting with
   row first_result_row so that the information is centred vertically. */

{
    static boolean specified_to_be_written,
        ideal_point_to_be_written,
        centroid_to_be_written,
        all_to_be_written;

    if (count ≡ first_result_row) {

        /* the result identifier should be written in this row */

        fprintf(stream, "{%s %s}", Identifier_Font, result_pointer→identifier);

        if (write_directions) {

            /* determine which directions will be written (on subsequent invocations of this
               function) */

            specified_to_be_written =
                    ¬Is_Zero_Subdistance(result_pointer→specified_direction);
            ideal_point_to_be_written =
                    ¬Is_Zero_Subdistance(result_pointer→ideal_point_direction);
            centroid_to_be_written =
                    ¬Is_Zero_Subdistance(result_pointer→centroid_direction);
            all_to_be_written = (specified_to_be_written ∧
                    ideal_point_to_be_written ∧ centroid_to_be_written ∧
                    all_three_equal(result_pointer→specified_direction,
                            result_pointer→ideal_point_direction,
                            result_pointer→centroid_direction));
}
```

```
} else if (write_directions ∧ (count > first_result_row)) {

    /* write the strength of the next non-zero direction: a ⇒ symbol is used for specified
       direction; a ⇒^I symbol for ideal point direction; a ⇒^μ symbol for centroid direction;
       a ⇒^* symbol if all three directions are of the same strength */

    if (all_to_be_written) {
        fprintf(stream, "%s\\,", All_Directions_Symbol);
        write_distance(stream, result_pointer→specified_direction, FALSE, FALSE);
        all_to_be_written = FALSE;
        specified_to_be_written = FALSE;
        ideal_point_to_be_written = FALSE;
        centroid_to_be_written = FALSE;
    } else if (specified_to_be_written) {
        fprintf(stream, "%s\\,", Specified_Direction_Symbol);
        write_distance(stream, result_pointer→specified_direction, FALSE, FALSE);
        specified_to_be_written = FALSE;
    } else if (ideal_point_to_be_written) {
        fprintf(stream, "%s\\,", Ideal_Point_Direction_Symbol);
        write_distance(stream, result_pointer→ideal_point_direction, FALSE, FALSE);
        ideal_point_to_be_written = FALSE;
    } else if (centroid_to_be_written) {
        fprintf(stream, "%s\\,", Centroid_Direction_Symbol);
        write_distance(stream, result_pointer→centroid_direction, FALSE, FALSE);
        centroid_to_be_written = FALSE;
    }
}

extern void
Write_Matrix(
        file stream,
        area *area_pointer,
        vector_element *facts_head,
        court *court_head,
        boolean hypothetical,
        cardinal number)

/* Writes a matrix of attribute values and (if facts_head is TRUE) metric information, for the
   instant case, the cases in the area pointed to by area_pointer, the ideal points in that area,
   and each result's centroid (if they have been calculated). If number is not zero then the
   instant case is actually a hypothetical (if hypothetical is TRUE) or an instantiation, and
   number is its number. */

{
    result *result_pointer;
    kase *case_pointer;
    matrix_element *matrix_pointer;
    vector_element *vector_pointer;
    centroid_element *centroid_pointer;
    cardinal count,
        first_result_row;
```

```
Indent(stream, 1);
fprintf(stream, "\\begin{small}\n");
Indent(stream, 2);
fprintf(stream, "\\begin{tabular}{*{2}{|c}");

if (area_pointer→number_of_attributes > 1)
    fprintf(stream, "*{%u}{@{\\hspace{%s}}c}|",
            area_pointer→number_of_attributes − 1, Matrix_Column_Separation);

if (court_head ≠ NULL)

    /* there will be a column for the rank of each case's court */

    fprintf(stream, "r|");

if (facts_head ≠ NULL) {

    /* there will be columns for metric information */

    fprintf(stream, "r@{\\hspace{%s}}r|r|", Column_Separation);

    if (area_pointer→infinite_weight)

        /* at least one of the attribute's weights is infinite, so the values obtained for S′ are
           meaningless: there will be no S′ column */

        fprintf(stream, "c|");
    else

        /* there will be an S′ column */

        fprintf(stream, "c@{\\hspace{%s}}c|", Column_Separation);

    if (area_pointer→correlation_coefficients)

        /* the instant case does not have all attribute values the same, and not every case,
           ideal point and centroid has all attribute values the same: there will be columns
           for the correlation coefficients r and r′ (if either of these conditions does not
           hold, all the values of r and r′ are meaningless) */

        fprintf(stream, "r@{\\hspace{%s}}r|", Column_Separation);
}
fprintf(stream, "c|}\\hline\n");

/* write the column headings */

Indent(stream, 3);
fprintf(stream, "&\\multicolumn{%u}{|c|}{\\it␣Attributes\\/}&",
        area_pointer→number_of_attributes);

if (court_head ≠ NULL)
    fprintf(stream, "&");
```

```
if (facts_head ≠ NULL) {
    fprintf(stream, "&&&&");
    if (¬area_pointer→infinite_weight)
        fprintf(stream, "&");
    if (area_pointer→correlation_coefficients)
        fprintf(stream, "&&");
}
fprintf(stream, "\\\\\n");
Indent(stream, 3);

fprintf(stream, "\\smash{\\raisebox{%s}{\\it Case\\/}}&", Raise_Height);

for (count = 1; count ≤ area_pointer→number_of_attributes; count++)
    fprintf(stream, "$A_{%u}$&", count);

if (court_head ≠ NULL)
    fprintf(stream, "\\multicolumn{1}{c|}{\\smash{\\raisebox{%s}{$c$}}}&",
        Raise_Height);

if (facts_head ≠ NULL) {
    fprintf(stream,
        "\\multicolumn{1}{c}{\\smash{\\raisebox{%s}{$d_{\\rm K}$}}}&"
        "\\multicolumn{1}{c|}{\\smash{\\raisebox{%s}{$d_{\\rm U}$}}}&"
        "\\multicolumn{1}{c|}{\\smash{\\raisebox{%s}{$\\Delta$}}}&"
        "\\smash{\\raisebox{%s}{$S$}}&",
        Raise_Height, Raise_Height, Raise_Height, Raise_Height);

    if (¬area_pointer→infinite_weight)
        fprintf(stream, "\\smash{\\raisebox{%s}{$S'$}}&",
            Raise_Height);

    if (area_pointer→correlation_coefficients)
        fprintf(stream, "\\multicolumn{1}{c}{\\smash{\\raisebox{%s}{$r$}}}&"
            "\\multicolumn{1}{c|}{\\smash{\\raisebox{%s}{$r'$}}}&",
            Raise_Height, Raise_Height);
}
fprintf(stream, "\\smash{\\raisebox{%s}{\\it Result\\/}}\\\\\\hline",
    Raise_Height);

if (facts_head ≠ NULL) {

    /* write details of the instant case */
    fprintf(stream, "\\hline\n");
    Indent(stream, 3);

    if (number ≡ 0)

        /* the instant case is the uninstantiated and unhypothesized instant case */

        fprintf(stream, "$C_{\\rm Instant}$&");
```

```
else if (hypothetical)
    /* the instant case is hypothetical number */
    fprintf(stream, "$C_{\\mbox{\\scriptsize␣Hypo-%u}}$&", number);
else
    /* the instant case is instantiation number */
    fprintf(stream, "$C_{\\mbox{\\scriptsize␣Inst-%u}}$&", number);
/* write the attribute values for the instant case: a • symbol for YES; a × symbol for
    NO; a blank space for UNKNOWN */
for (vector_pointer = facts_head; vector_pointer ≠ NULL;
        vector_pointer = vector_pointer→next)
    fprintf(stream, "%s&",
            vector_pointer→attribute_value ≡ YES ? Yes_Symbol :
            vector_pointer→attribute_value ≡ NO ? No_Symbol :
            Unknown_Symbol);
/* leave an appropriate number of columns empty */
fprintf(stream, "\\multicolumn{%u}{c|}{}\\\\\\\\hline",
        court_head ≡ NULL ?
            area_pointer→infinite_weight ?
            area_pointer→correlation_coefficients ? 7 : 5 :
            area_pointer→correlation_coefficients ? 8 : 6 :
            area_pointer→infinite_weight ?
            area_pointer→correlation_coefficients ? 8 : 6 :
            area_pointer→correlation_coefficients ? 9 : 7);
}
/* for every result ... */
for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
        result_pointer = result_pointer→next) {
    /* determine the first row in which information should appear, for this result, in the
        "Result" column so that the information is centred vertically */
    first_result_row = 0;
    for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
            case_pointer = case_pointer→next)
        first_result_row++;
    if (result_pointer→ideal_point_head ≠ NULL)
        first_result_row++;
    if (result_pointer→centroid_head ≠ NULL)
        first_result_row++;
    if (first_result_row ≠ 0) {
        if ((facts_head ≠ NULL) ∧ (first_result_row > 1) ∧
                ¬Is_Zero_Subdistance(result_pointer→specified_direction)) {
            first_result_row--;
```

```
if (¬all_three_equal(result_pointer→specified_direction,
                     result_pointer→ideal_point_direction,
                     result_pointer→centroid_direction)) {
    if ((first_result_row > 1) ∧
            ¬Is_Zero_Subdistance(result_pointer→ideal_point_direction))
        first_result_row--;
    if ((first_result_row > 1) ∧
            ¬Is_Zero_Subdistance(result_pointer→centroid_direction))
        first_result_row--;
    }
}
first_result_row = (first_result_row + 1) / 2;

count = 1;
fprintf(stream, "\\hline");

/* for every case with this result ... */

for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
        case_pointer = case_pointer→next) {
    fprintf(stream, "\n");
    Indent(stream, 3);
    fprintf(stream, "$C_{%u}$&", case_pointer→number);

    /* write the attribute values for this case */

    for (matrix_pointer = case_pointer→matrix_head; matrix_pointer ≠ NULL;
            matrix_pointer = matrix_pointer→case_next)
        fprintf(stream, "%s&",
                matrix_pointer→attribute_value ≡ YES ? Yes_Symbol :
                matrix_pointer→attribute_value ≡ NO ? No_Symbol :
                Unknown_Symbol);

    if (court_head ≠ NULL)

        /* write the rank of the case's court */

        if ((case_pointer→court_string ≠ NULL) ∧
                (case_pointer→court_rank ≠ 0))
            fprintf(stream, "%u&", case_pointer→court_rank);
        else
            fprintf(stream, "\\footnotesize?&");
        if (facts_head ≠ NULL)
            write_metrics(stream, case_pointer→metrics,
                    ¬area_pointer→infinite_weight, area_pointer→correlation_coefficients);

        write_result(stream, count++, first_result_row, facts_head ≠ NULL,
                result_pointer);
        fprintf(stream, "\\\\");
}
```

```
/* write a line (an appropriate number of columns wide) under the cases for this
   result */
if ((result_pointer→case_head ≠ NULL) ∧
        ((result_pointer→ideal_point_head ≠ NULL) ∨
                (result_pointer→centroid_head ≠ NULL)))
    fprintf(stream, "\\cline{2-%u}", facts_head ≡ NULL ?
            court_head ≡ NULL ?
            area_pointer→number_of_attributes + 1 :
            area_pointer→number_of_attributes + 2 :
            court_head ≡ NULL ?
            area_pointer→infinite_weight ?
            area_pointer→correlation_coefficients ?
            area_pointer→number_of_attributes + 7 :
            area_pointer→number_of_attributes + 5 :
            area_pointer→correlation_coefficients ?
            area_pointer→number_of_attributes + 8 :
            area_pointer→number_of_attributes + 6 :
            area_pointer→infinite_weight ?
            area_pointer→correlation_coefficients ?
            area_pointer→number_of_attributes + 8 :
            area_pointer→number_of_attributes + 6 :
            area_pointer→correlation_coefficients ?
            area_pointer→number_of_attributes + 9 :
            area_pointer→number_of_attributes + 7);
if (result_pointer→ideal_point_head ≠ NULL) {
    /* this result has an ideal point, so write its details */
    fprintf(stream, "\n");
    Indent(stream, 3);
    fprintf(stream, "$I_{\\mbox{\\scriptsize%s␣%s}}$&",
            Identifier_Font, result_pointer→identifier);
    /* write the attribute values for this ideal point */
    for (vector_pointer = result_pointer→ideal_point_head;
            vector_pointer ≠ NULL; vector_pointer = vector_pointer→next)
        fprintf(stream, "%s&", (vector_pointer→attribute_value ≡ YES ?
                Yes_Symbol : (vector_pointer→attribute_value ≡ NO ?
                        No_Symbol : Unknown_Symbol)));
    if (court_head ≠ NULL)
        fprintf(stream, "&");
    if (facts_head ≠ NULL)
        write_metrics(stream, result_pointer→ideal_point_metrics,
                ¬area_pointer→infinite_weight, area_pointer→correlation_coefficients);
    write_result(stream, count++, first_result_row, facts_head ≠ NULL,
            result_pointer);
    fprintf(stream, "\\\\");
}
```

```
if (result_pointer→centroid_head ≠ NULL) {

    /* this result has a centroid, so write its details */

    fprintf(stream, "\n");
    Indent(stream, 3);
    fprintf(stream, "$\\mu_{\\mbox{\\scriptsize%s %s}}$&",
            Identifier_Font, result_pointer→identifier);

    /* write the attribute values for this centroid */

    for (centroid_pointer = result_pointer→centroid_head;
         centroid_pointer ≠ NULL; centroid_pointer = centroid_pointer→next)
        fprintf(stream, "%s&",
                centroid_pointer→unknown ? Unknown_Symbol :
                Nearest_Attribute_Value(centroid_pointer→value) ≡ YES ?
                Yes_Symbol : No_Symbol);

    if (court_head ≠ NULL)
        fprintf(stream, "&");

    if (facts_head ≠ NULL)
        write_metrics(stream, result_pointer→centroid_metrics,
                ¬area_pointer→infinite_weight, area_pointer→correlation_coefficients);
                    write_result(stream, count++, first_result_row, facts_head ≠ NULL,
                            result_pointer);
                    fprintf(stream, "\\\\");
                }
                fprintf(stream, "\\hline");
            }
        }
        fprintf(stream, "\n");
        Indent(stream, 2);
        fprintf(stream, "\\end{tabular}\n");
        Indent(stream, 1);
        fprintf(stream, "\\end{small}\n\n");
}

static void
write_opening_and_closing(
        file dump_stream,
        area *area_pointer,
        boolean verbose)

/* Writes the opening and closing strings for the area pointed to by area_pointer. Writes each
   string in full only if verbose is TRUE. */
```

```
{
    if (area_pointer→opening ≠ NULL) {
        fprintf(dump_stream, "%s{Opening}\n\n", Subheading);
        Indent(dump_stream, 1);
        fprintf(dump_stream, "\\begin{list}{}{\\leftmargin=0mm}\\item[]\n");
        if (verbose)
            Write(dump_stream, area_pointer→opening, Empty_String, 2, Hang);
        else
            Write(dump_stream, "[Opening.]", Empty_String, 2, Hang);
        Indent(dump_stream, 1);
        fprintf(dump_stream, "\\end{list}\n\n");
    }
    if (area_pointer→closing ≠ NULL) {
        fprintf(dump_stream, "%s{Closing}\n\n", Subheading);
        Indent(dump_stream, 1);
        fprintf(dump_stream, "\\begin{list}{}{\\leftmargin=0mm}\\item[]\n");
        if (verbose)
            Write(dump_stream, area_pointer→closing, Empty_String, 2, Hang);
        else
            Write(dump_stream, "[Closing.]", Empty_String, 2, Hang);
        Indent(dump_stream, 1);
        fprintf(dump_stream, "\\end{list}\n\n");
    }
}
```

```
static void
write_result_list(
        file dump_stream,
        result *result_pointer)

/* Writes details of each result in the list of results pointed to by result_pointer. */

{
    fprintf(dump_stream, "%s{Results}\n\n", Subheading);
    Indent(dump_stream, 1);

    fprintf(dump_stream, "\\begin{description}\n\n");

    /* while there are still results ... */

    while (result_pointer ≠ NULL) {
        Indent(dump_stream, 2);
        fprintf(dump_stream, "\\item[\\rm{%s␣%s}:]\n",
                Identifier_Font, result_pointer→identifier);
        Write(dump_stream, result_pointer→string, ".\n", 3, No_Hang);
        result_pointer = result_pointer→next;
    }
    Indent(dump_stream, 1);
    fprintf(dump_stream, "\\end{description}\n\n");
}
```

```
static void
write_attribute_list(
        file dump_stream,
        attribute *attribute_pointer)

/* Writes details of each attribute in the list of attributes pointed to by attribute_pointer. */

{
    direction_list_element *direction_list_pointer;
    identifier_list_element *identifier_list_pointer;

    fprintf(dump_stream, "%s{Attributes}\n\n", Subheading);
    Indent(dump_stream, 1);

    fprintf(dump_stream, "\\begin{description}\n\n");

    /* while there are still attributes ... */

    while (attribute_pointer ≠ NULL) {

        Indent(dump_stream, 2);
        fprintf(dump_stream, "\\item[\\rm$A_{%u}$:]\n", attribute_pointer→number);

        if (attribute_pointer→external_attribute) {

            /* the attribute is external, so indicate the area to which it is linked using a ⇔
               symbol */

            Indent(dump_stream, 3);
            fprintf(dump_stream, "%s␣{%s␣%s}␣area\n", External_Area_Symbol,
                    Identifier_Font, attribute_pointer→details.external.area_identifier);

        } else

            /* the attribute is local, so write the attribute's question */

            Write(dump_stream, attribute_pointer→details.local.question, "?", 3, No_Hang);

    fprintf(dump_stream, "\n");

    Indent(dump_stream, 3);
    fprintf(dump_stream, "\\begin{description}\n\n");

    if (attribute_pointer→yes ≠ NULL) {

        /* the attribute has a YES string, so write it */

        Indent(dump_stream, 4);
        fprintf(dump_stream, "\\item[\\sc␣yes:]\n");
        Write(dump_stream, attribute_pointer→yes, ".\n", 5, No_Hang);
```

```
if (attribute_pointer→external_attribute) {

    /* the attribute is external, so indicate the association of results from the external area with YES values of this attribute using a ⇐ symbol (list the result identifiers, separated by a ∨ symbol) */

    identifier_list_pointer = attribute_pointer→details.external.yes_identifier_head;
    while (identifier_list_pointer ≠ NULL) {
        if (identifier_list_pointer ≡
                attribute_pointer→details.external.yes_identifier_head) {

            /* this is the first identifier to write */

            Indent(dump_stream, 5);
            fprintf(dump_stream, "%s␣{%s␣%s}", External_Result_Symbol,
                    Identifier_Font, identifier_list_pointer→identifier);
        } else {
            fprintf(dump_stream, "~%s\n", Disjunction_Symbol);
            Indent(dump_stream, 5);
            fprintf(dump_stream, "{%s␣%s}", Identifier_Font,
                    identifier_list_pointer→identifier);
        }
        identifier_list_pointer = identifier_list_pointer→next;
    }
    if (attribute_pointer→details.external.yes_identifier_head ≠ NULL)
        fprintf(dump_stream, "\n\n");
}

/* indicate the specified direction for YES values of this attribute using a ⇒ symbol (list the result identifiers, separated by a ∨ symbol) */

direction_list_pointer = attribute_pointer→yes_direction_head;
while (direction_list_pointer ≠ NULL) {
    if (direction_list_pointer ≡ attribute_pointer→yes_direction_head) {

        /* this is the first identifier to write */

        Indent(dump_stream, 5);
        fprintf(dump_stream, "%s␣{%s␣%s}", Specified_Direction_Symbol,
                Identifier_Font, direction_list_pointer→result→identifier);
    } else {
        fprintf(dump_stream, "~%s\n", Disjunction_Symbol);
        Indent(dump_stream, 5);
        fprintf(dump_stream, "{%s␣%s}", Identifier_Font,
                direction_list_pointer→result→identifier);
        }
        direction_list_pointer = direction_list_pointer→next;
    }
    if (attribute_pointer→yes_direction_head ≠ NULL)
        fprintf(dump_stream, "\n\n");
}
```

```
if (attribute_pointer→no ≠ NULL) {
    /* the attribute has a NO string, so write it */
    Indent(dump_stream, 4);
    fprintf(dump_stream, "\\item[\\sc no:]\n");
    Write(dump_stream, attribute_pointer→no, ".\n", 5, No_Hang);
    if (attribute_pointer→external_attribute) {
        /* the attribute is external, so indicate the association of results from the ex-
        ternal area with NO values of this attribute using a ⇐ symbol (list the result
        identifiers, separated by a ∨ symbol) */
        identifier_list_pointer = attribute_pointer→details.external.no_identifier_head;
        while (identifier_list_pointer ≠ NULL) {
            if (identifier_list_pointer ≡
                    attribute_pointer→details.external.no_identifier_head) {
                /* this is the first identifier to write */
                Indent(dump_stream, 5);
                fprintf(dump_stream, "%s {%s %s}", External_Result_Symbol,
                        Identifier_Font, identifier_list_pointer→identifier);
            } else {
                fprintf(dump_stream, "~%s\n", Disjunction_Symbol);
                Indent(dump_stream, 5);
                fprintf(dump_stream, "{%s %s}", Identifier_Font,
                        identifier_list_pointer→identifier);
            }
            identifier_list_pointer = identifier_list_pointer→next;
        }
        if (attribute_pointer→details.external.no_identifier_head ≠ NULL)
            fprintf(dump_stream, "\n\n");
    }
    /* indicate the specified direction for NO values of this attribute using a ⇒ symbol
    (list the result identifiers, separated by a ∨ symbol) */
    direction_list_pointer = attribute_pointer→no_direction_head;
    while (direction_list_pointer ≠ NULL) {
        if (direction_list_pointer ≡ attribute_pointer→no_direction_head) {
            /* this is the first identifier to write */
            Indent(dump_stream, 5);
            fprintf(dump_stream, "%s {%s %s}", Specified_Direction_Symbol,
                    Identifier_Font, direction_list_pointer→result→identifier);
        } else {
            fprintf(dump_stream, "~%s\n", Disjunction_Symbol);
            Indent(dump_stream, 5);
            fprintf(dump_stream, "{%s %s}", Identifier_Font,
                    direction_list_pointer→result→identifier);
        }
        direction_list_pointer = direction_list_pointer→next;
    }
    if (attribute_pointer→no_direction_head ≠ NULL)
        fprintf(dump_stream, "\n\n");
}
```

```
if (attribute_pointer→unknown ≠ NULL) {
    /* the attribute has an UNKNOWN string, so write it */
    Indent(dump_stream, 4);
    fprintf(dump_stream, "\\item[\\sc␣unknown:]\n");
    Write(dump_stream, attribute_pointer→unknown, ".\n", 5, No_Hang);
    if (attribute_pointer→external_attribute) {
        /* the attribute is external, so indicate the association of results from the external area with UNKNOWN values of this attribute using a ⇐ symbol (list the result identifiers, separated by a ∨ symbol) */
        identifier_list_pointer =
                attribute_pointer→details.external.unknown_identifier_head;
        while (identifier_list_pointer ≠ NULL) {
            if (identifier_list_pointer ≡
                    attribute_pointer→details.external.unknown_identifier_head) {
                /* this is the first identifier to write */
                Indent(dump_stream, 5);
                fprintf(dump_stream, "%s␣{%s␣%s}", External_Result_Symbol,
                        Identifier_Font, identifier_list_pointer→identifier);
            } else {
                fprintf(dump_stream, "~%s\n", Disjunction_Symbol);
                Indent(dump_stream, 5);
                fprintf(dump_stream, "{%s␣%s}", Identifier_Font,
                        identifier_list_pointer→identifier);
            }
            identifier_list_pointer = identifier_list_pointer→next;
        }
        if (attribute_pointer→details.external.unknown_identifier_head ≠ NULL)
            fprintf(dump_stream, "\n\n");
    }
    /* indicate the specified direction for UNKNOWN values of this attribute using a ⇒ symbol (list the result identifiers, separated by a ∨ symbol) */
    direction_list_pointer = attribute_pointer→unknown_direction_head;
    while (direction_list_pointer ≠ NULL) {
        if (direction_list_pointer ≡ attribute_pointer→unknown_direction_head) {
            /* this is the first identifier to write */
            Indent(dump_stream, 5);
            fprintf(dump_stream, "%s␣{%s␣%s}", Specified_Direction_Symbol,
                    Identifier_Font, direction_list_pointer→result→identifier);
        } else {
            fprintf(dump_stream, "~%s\n", Disjunction_Symbol);
            Indent(dump_stream, 5);
            fprintf(dump_stream, "{%s␣%s}", Identifier_Font,
                    direction_list_pointer→result→identifier);
        }
        direction_list_pointer = direction_list_pointer→next;
    }
```

```
    if (attribute_pointer→unknown_direction_head ≠ NULL)
        fprintf(dump_stream, "\n\n");
}
Indent(dump_stream, 3);
fprintf(dump_stream, "\\end{description}\n\n");

if (¬attribute_pointer→external_attribute ∧
        (attribute_pointer→details.local.help ≠ NULL))
            /* the attribute has a help string, so write it */

            Write(dump_stream, attribute_pointer→details.local.help, "\n", 3, No_Hang);

        attribute_pointer = attribute_pointer→next;
    }
    Indent(dump_stream, 1);
    fprintf(dump_stream, "\\end{description}\n\n");
}

extern void
Write_Year_and_Court(
        file stream,
        kase *case_pointer,
        cardinal level)

/* Describes the case pointed to by case_pointer as "a year decision of court". */
{
    cardinal hundreds = case_pointer→year / 100;

    Indent(stream, level);
    if ((hundreds ≡ 8) ∨ (hundreds ≡ 11) ∨ (hundreds ≡ 18))
        fprintf(stream, "an");
    else
        fprintf(stream, "a");
    fprintf(stream, " %u decision", case_pointer→year);

    if (case_pointer→court_string ≠ NULL) {
        fprintf(stream, " of\n");
        Indent(stream, level);
        fprintf(stream, "  %s", case_pointer→court_string);
    }
}

static void
write_case_list(
        file dump_stream,
        file log_stream,
        area *area_pointer,
        boolean verbose)

/* Writes details of each case in the area pointed to by area_pointer. Summarizes each case in
    full only if verbose is TRUE. */
```

```
    if (attribute_pointer→unknown ≠ NULL) {
        options[count] = Unknown_Character;
        count += sizeof(char);
    }
    if (attribute_pointer→details.local.help ≠ NULL) {
        options[count] = Help_Character;
        count += sizeof(char);
    }
    options[count] = Null_Character;
    count += sizeof(char);

    /* prompt the user for one of the valid attribute values */

    do {
        if ((option = Get_Option(attribute_pointer→details.local.question, options)) ≡
                Quit_Character) {
            Indent(log_stream, level);
            fprintf(log_stream, "Quitting␣consultation.\n\n");
            return FALSE;
        }
        if (option ≡ Help_Character)
            fprintf(stdout, "%s\n", attribute_pointer→details.local.help);
    } while (option ≡ Help_Character);

    /* set this attribute's value to that chosen */

    switch (option) {
        case Yes_Character:
            vector_pointer→attribute_value = YES;
            if (echo)
                fprintf(stdout, "Yes:␣%s.\n", attribute_pointer→yes);
            break;
        case No_Character:
            vector_pointer→attribute_value = NO;
            if (echo)
                fprintf(stdout, "No:␣%s.\n", attribute_pointer→no);
            break;
        case Unknown_Character:
            vector_pointer→attribute_value = UNKNOWN;
            if (echo)
                fprintf(stdout, "Unknown:␣%s.\n", attribute_pointer→unknown);
            break;
    }
    return TRUE;
}
```

```
if (ch ≠ Carriage_Return_Character) {

    /* skip over the rest of the line of input */

    for (temp_ch = getc(stdin);
            (temp_ch ≠ Carriage_Return_Character) ∧ (temp_ch ≠ EOF);
            temp_ch = getc(stdin));

    if (temp_ch ≡ EOF)
        return Quit_Character;
    }
    if (strchr(options, ch) ≠ NULL)

        /* the user has entered a valid option */

        return ch;

    /* the user has entered an invalid option */

    fprintf(stdout, "Please enter '%c'", options[0]);
    for (count = 1; options[count] ≠ Null_Character; count += sizeof(char))
        fprintf(stdout, ", '%c'", options[count]);
    fprintf(stdout, " or '%c'.\n%s? ", Quit_Character, question);
  }
}

static boolean
get_local_fact(
        file log_stream,
        attribute *attribute_pointer,
        vector_element *vector_pointer,
        boolean echo,
        cardinal level)

/* Interrogates the user as to the value, in the instant case, of the local attribute pointed to by
   attribute_pointer. Puts the attribute value in the vector element pointed to by vector_pointer.
   Returns FALSE, if the user chooses to quit. */

{
    char options[Max_Attribute_Options + 1],
        option;
    cardinal count = 0;

    /* determine which values are valid for this attribute */

    if (attribute_pointer→yes ≠ NULL) {
        options[count] = Yes_Character;
        count += sizeof(char);
    }
    if (attribute_pointer→no ≠ NULL) {
        options[count] = No_Character;
        count += sizeof(char);
    }
```

static *boolean*
get_external_fact(
file log_stream,
case_law_specification case_law,
boolean adjust,
boolean echo,
boolean inputable_latex,
boolean verbose,
cardinal hypothetical_reports,
cardinal hypothetical_changes,
cardinal level,
string distances_filename,
string weights_filename,
string report_filename,
*attribute *attribute_pointer*,
cardinal attribute_number,
*vector_element *vector_pointer*)

/* Resolves the value, in the instant case, of the external attribute pointed to by *attribute_pointer* (attribute *attribute_number*) by reference to the relevant area. Puts the attribute value in the vector element pointed to by *vector_pointer*. Returns *FALSE*, if the user chooses to quit. */

{
string nearest_result_identifier;
boolean found = *FALSE*;
*identifier_list_element *identifier_list_pointer*;
char *message*[*Max_Error_Message_Length*];

Indent(*log_stream*, *level*);
fprintf(*log_stream*, "A%u is external.\n\n", *attribute_number*);

/* determine the identifier of the "likely result" of the instant case in the relevant area (*attribute_pointer*→*details.external.area_identifier*) */

nearest_result_identifier = *Case_Law*(*log_stream*, *case_law*,
attribute_pointer→*details.external.area_identifier*, *adjust*, *echo*,
inputable_latex, *verbose*, *hypothetical_reports*, *hypothetical_changes*,
level + 1, *distances_filename*, *weights_filename*, *report_filename*);

if (*nearest_result_identifier* ≡ *NULL*)

/* quit */

return *FALSE*;

/* search for the result identifier in the list of external result identifiers for YES for this attribute */

identifier_list_pointer = *attribute_pointer*→*details.external.yes_identifier_head*;
vector_pointer→*attribute_value* = *YES*;
while ((*identifier_list_pointer* ≠ *NULL*) ∧ ¬*found*) {
found = ¬*strcmp*(*identifier_list_pointer*→*identifier*, *nearest_result_identifier*);
if (¬*found*)
identifier_list_pointer = *identifier_list_pointer*→*next*;
}

```
if (¬found) {

    /* search for the result identifier in the list of external result identifiers for NO for this
       attribute */

    identifier_list_pointer = attribute_pointer→details.external.no_identifier_head;
    vector_pointer→attribute_value = NO;
    while ((identifier_list_pointer ≠ NULL) ∧ ¬found) {
        found = ¬strcmp(identifier_list_pointer→identifier, nearest_result_identifier);
        if (¬found)
            identifier_list_pointer = identifier_list_pointer→next;
    }
    if (¬found) {

        /* search for the result identifier in the list of external result identifiers for UNKNOWN
           for this attribute */

        identifier_list_pointer = attribute_pointer→details.external.unknown_identifier_head;
        vector_pointer→attribute_value = UNKNOWN;
        while ((identifier_list_pointer ≠ NULL) ∧ ¬found) {
            found = ¬strcmp(identifier_list_pointer→identifier, nearest_result_identifier);
            if (¬found)
                identifier_list_pointer = identifier_list_pointer→next;
        }
        if (¬found) {
            sprintf(message, "Unexpected␣external␣result␣identifier␣\"%s\"",
                    nearest_result_identifier);
            error_exit(log_stream, message);
        }
    }
}

/* write details of the attribute value to the log file */

Indent(log_stream, level);
fprintf(log_stream, "Value␣of␣A%u␣is␣", attribute_number);
switch (vector_pointer→attribute_value) {
    case YES:
        fprintf(log_stream, "YES");
        break;
    case NO:
        fprintf(log_stream, "NO");
        break;
    case UNKNOWN:
        fprintf(log_stream, "UNKNOWN");
        break;
    }
    fprintf(log_stream, ".\n\n");

    return TRUE;
}
```

```
static vector_element *
get_fact(
        file log_stream,
        case_law_specification case_law,
        boolean adjust,
        boolean echo,
        boolean inputable_latex,
        boolean verbose,
        cardinal hypothetical_reports,
        cardinal hypothetical_changes,
        cardinal level,
        string distances_filename,
        string weights_filename,
        string report_filename,
        attribute *attribute_pointer,
        cardinal attribute_number)
```

/* Determines the value, in the instant case, of the attribute pointed to by *attribute_pointer* (attribute *attribute_number*). Interrogates the user, if the attribute is local; resolves the value by reference to the relevant area, if the attribute is external. Returns a pointer to a single vector element containing the attribute value; or *NULL*, if the user chooses to quit. */

```
{
    vector_element *vector_pointer;
    /* allocate memory for this vector element */
    if ((vector_pointer = (vector_element *) malloc(sizeof(vector_element))) ≡ NULL)
        error_exit(log_stream, "malloc failed during fact vector handling");
    if (attribute_pointer→external_attribute) {
        /* the attribute is external */
        if (¬get_external_fact(log_stream, case_law, adjust, echo, inputable_latex, verbose,
                hypothetical_reports, hypothetical_changes, level,
                distances_filename, weights_filename, report_filename,
                attribute_pointer, attribute_number, vector_pointer)) {
            /* quit */
            free(vector_pointer);
            return NULL;
        }
    } else {
        /* the attribute is local */
        if (¬get_local_fact(log_stream, attribute_pointer, vector_pointer, echo, level)) {
            /* quit */
            free(vector_pointer);
            return NULL;
        }
    }
    vector_pointer→next = NULL;
    return vector_pointer;
}
```

```
extern vector_element *
Get_Facts(
        file log_stream,
        case_law_specification case_law,
        area *area_pointer,
        boolean adjust,
        boolean echo,
        boolean inputable_latex,
        boolean verbose,
        cardinal hypothetical_reports,
        cardinal hypothetical_changes,
        cardinal level,
        string distances_filename,
        string weights_filename,
        string report_filename)
```

/* Interrogates the user as to the values of local attributes, in the instant case, in the area pointed to by *area_pointer*. Resolves the value of external attributes by reference to the relevant area, recursively invoking *Case_Law*(). Prompts the user by writing to *stdout*; reads the user's response from *stdin*. Returns a pointer to a fact vector containing the facts of the instant case; or *NULL*, if the user chooses to quit. */

```
{
    attribute *attribute_pointer;
    vector_element *facts_head = NULL,
      *vector_pointer = NULL;
    cardinal count = 1;
    /* for every attribute ... */
    for (attribute_pointer = area_pointer→attribute_head; attribute_pointer ≠ NULL;
            attribute_pointer = attribute_pointer→next)
        if (facts_head ≡ NULL) {
            /* this is the first attribute */
            if ((facts_head = get_fact(log_stream, case_law, adjust, echo, inputable_latex,
                                       verbose, hypothetical_reports, hypothetical_changes, level,
                                       distances_filename, weights_filename, report_filename,
                                       attribute_pointer, 1)) ≡ NULL)
                /* quit */
                return NULL;
            vector_pointer = facts_head;
```

```
} else {
    /* this is not the first attribute */
    count++;
    if ((vector_pointer→next =
                get_fact(log_stream, case_law, adjust, echo, inputable_latex,
                    verbose, hypothetical_reports, hypothetical_changes, level,
                    distances_filename, weights_filename, report_filename,
                    attribute_pointer, count)) ≡ NULL)
            /* quit */
            return NULL;
        vector_pointer = vector_pointer→next;
    }
    return facts_head;
}
```

11

The ODOMETER module

odometer.h

/* This is the header file for the ODOMETER module. It is also included by the CASES, DUMPER and REPORTER modules. */

/* external functions */

extern *relative_distance_type*
Relative_Distance(
    distance_type x,
    distance_type y);

extern void
Calculate_Distances(
    file distances_stream,
    file log_stream,
    *area *area_pointer*,
    case_law_specification case_law,
    *vector_element *facts_head*,
    boolean hypothetical,
    cardinal number,
    cardinal level);

odometer.c

/* This is the implementation file for the ODOMETER module. */

```
#include ⟨stdio.h⟩
#include ⟨math.h⟩
#include "shyster.h"
#include "cases.h"
#include "odometer.h"
#include "dumper.h"
#include "scales.h"
```

```
static void
error_exit(
        file stream,
        const string message)
{
    Write_Error_Message_And_Exit(stream, "Odometer", message);
}
```

```
static void
warning(
        file stream,
        const string message,
        cardinal level)
{
    Write_Warning_Message(stream, "Odometer", message, level);
}
```

```
static void
zero_subdistance(
        distance_subtype *subdistance_pointer)

/* Sets the subdistance pointed to by subdistance_pointer to zero. */

{
    subdistance_pointer→infinite = 0;
    subdistance_pointer→finite = 0.0;
}
```

```
static void
zero_distance(
        distance_type *distance_pointer)

/* Sets the distance pointed to by distance_pointer to zero. */

{
    zero_subdistance(&distance_pointer→known);
    zero_subdistance(&distance_pointer→unknown);
}
```

```
static void
zero_correlation(
        correlation_type *correlation_pointer)
/* Sets the correlation coefficient pointed to by correlation_pointer to zero. */
{
    correlation_pointer→meaningless = FALSE;
    correlation_pointer→unweighted = 0.0;
    correlation_pointer→weighted = 0.0;
}

static void
zero_metrics(
        metrics_type *metrics_pointer)
/* Sets the metrics pointed to by metrics_pointer to zero. */
{
    zero_distance(&metrics_pointer→distance);
    metrics_pointer→number_of_known_differences = 0;
    metrics_pointer→number_of_known_pairs = 0;
    metrics_pointer→weighted_association_coefficient = 0.0;
    zero_correlation(&metrics_pointer→correlation_coefficient);
}

static relative_distance_type
relative_subdistance(
        distance_subtype x,
        distance_subtype y)
/* Returns NEARER, if the subdistance x is less than the subdistance y; returns FURTHER,
    if x is greater than y; returns EQUIDISTANT, otherwise. */
{
    if ((x.infinite ≡ y.infinite) ∧ Is_Equal(x.finite, y.finite, Distance_Precision))
        return EQUIDISTANT;
    else if ((x.infinite < y.infinite) ∨
            ((x.infinite ≡ y.infinite) ∧ Is_Less(x.finite, y.finite, Distance_Precision)))
        return NEARER;
    else
        return FURTHER;
}
```

```
extern relative_distance_type
Relative_Distance(
        distance_type x,
        distance_type y)

/* Returns NEARER, if the distance x is less than the distance y; returns FURTHER, if x is
   greater than y; returns EQUIDISTANT, otherwise. */
{
    if ((x.known.infinite + x.unknown.infinite ≡
                    y.known.infinite + y.unknown.infinite) ∧
            Is_Equal(x.known.finite + x.unknown.finite,
                    y.known.finite + y.unknown.finite, Distance_Precision))
        return EQUIDISTANT;
    else if ((x.known.infinite + x.unknown.infinite <
                    y.known.infinite + y.unknown.infinite) ∨
            ((x.known.infinite + x.unknown.infinite ≡
                    y.known.infinite + y.unknown.infinite) ∧
                Is_Less(x.known.finite + x.unknown.finite,
                    y.known.finite + y.unknown.finite, Distance_Precision)))
        return NEARER;
    else
        return FURTHER;
}

static void
add_weight(
        file log_stream,
        attribute_value_type x,
        attribute_value_type y,
        weight_type weight,
        metrics_type *metrics_pointer,
        cardinal level)

/* Adds weight to the known distance in the metrics pointed to by metrics_pointer, if the
   attribute values x and y are known and different. Adds weight to the unknown distance
   in the metrics pointed to by metrics_pointer, if either of the attribute values x or y is
   UNKNOWN. */
{
    if (¬weight.infinite ∧ Is_Zero(weight.finite) ∧ ((x ≠ UNKNOWN) ∨ (y ≠ UNKNOWN)))
        warning(log_stream, "known␣attribute␣value␣for␣weightless␣attribute", level);
    else if ((x ≡ UNKNOWN) ∨ (y ≡ UNKNOWN))
        if (weight.infinite)
            metrics_pointer→distance.unknown.infinite++;
        else
            metrics_pointer→distance.unknown.finite += weight.finite;
```

```
else {
    if (x ≠ y) {
        if (weight.infinite)
            metrics_pointer→distance.known.infinite++;
        else
            metrics_pointer→distance.known.finite += weight.finite;
    }
}

static void
sum_pair(
        attribute_value_type x,
        attribute_value_type y,
        weight_type weight,
        metrics_type *sum_pointer_X,
        correlation_type *sum_pointer_Y)

/* Adds the attribute values (weighted and unweighted) of both x and y to various of the
   metrics of sum_pointer_X and sum_pointer_Y, if both x and y are known. (These metrics
   are used here as temporary storage; the final sums are later used to calculate association
   and correlation coefficients.) */

{
    floating_point temp;

    if ((x ≠ UNKNOWN) ∧ (y ≠ UNKNOWN)) {

        sum_pointer_X→number_of_known_pairs++;
        if (x ≠ y)
            sum_pointer_X→number_of_known_differences++;

        sum_pointer_X→weighted_association_coefficient += weight.finite;

        if (weight.infinite)

            /* one of the attributes is infinitely weighted, so use a "pseudo-infinite" weight for
               the calculation of weighted correlation coefficients */

            weight.finite = Very_Heavy_Indeed;

        (void) Attribute_Value(x, &temp);
            sum_pointer_X→correlation_coefficient.unweighted += temp;
            sum_pointer_X→correlation_coefficient.weighted += temp × weight.finite;

            (void) Attribute_Value(y, &temp);
            sum_pointer_Y→unweighted += temp;
            sum_pointer_Y→weighted += temp × weight.finite;
        }
}
```

static void
correlate_pair(
attribute_value_type x,
attribute_value_type y,
floating_point mean_X,
floating_point mean_Y,
*floating_point *numerator*,
*floating_point *left_denominator*,
*floating_point *right_denominator*,
weight_type weight)

/* Updates **numerator*, **left_denominator* and **right_denominator* appropriately, if both x and y are known: **left_denominator* is incremented by the square of the weighted value of x less *mean_X*; **right_denominator* is incremented by the square of the weighted value of y less *mean_Y*; **numerator* is incremented by the product of the weighted value of x less *mean_X* and the weighted value of y less *mean_Y*. These variables are later used in the calculation of the weighted correlation coefficient r' which is defined as

$$r' = \frac{\sum_{i=1}^{n} \left(A_{ij} \times w_i - \bar{A}'_j\right)\left(A_{ik} \times w_i - \bar{A}'_k\right)}{\sqrt{\sum_{i=1}^{n} \left(A_{ij} \times w_i - \bar{A}'_j\right)^2 \sum_{i=1}^{n} \left(A_{ik} \times w_i - \bar{A}'_k\right)^2}}$$

where A_{ij} is the value of the ith attribute for the jth case, w_i is the weight of the ith attribute, and $\bar{A}'_j$ is the weighted mean of all attribute values for the jth case. (**numerator*, **left_denominator* and **right_denominator* are so-named because they form the numerator, and the left and right side of the (square of the) denominator, in the above formula.) */

{
floating_point temp_X,
temp_Y;

if $((x \neq UNKNOWN) \wedge (y \neq UNKNOWN))$ {

if (*weight.infinite*)

/* one of the attributes is infinitely weighted, so use a "pseudo-infinite" weight for the calculation of weighted correlation coefficients */

weight.finite = *Very_Heavy_Indeed*;

(**void**) *Attribute_Value*(x, &*temp_X*);
(**void**) *Attribute_Value*(y, &*temp_Y*);
**numerator* += (*temp_X* × *weight.finite* − *mean_X*) ×
(*temp_Y* × *weight.finite* − *mean_Y*);
**left_denominator* += (*temp_X* × *weight.finite* − *mean_X*) ×
(*temp_X* × *weight.finite* − *mean_X*);
**right_denominator* += (*temp_Y* × *weight.finite* − *mean_Y*) ×
(*temp_Y* × *weight.finite* − *mean_Y*);
}
}

```
static void
calculate_case_means(
        matrix_element *matrix_pointer,
        vector_element *vector_pointer,
        attribute *attribute_pointer,
        metrics_type *metrics_pointer,
        correlation_type *correlation_pointer)

/* Calculates the mean attribute values for a leading case and the instant case (their attribute
   values are pointed to by matrix_pointer and vector_pointer, respectively) and stores them as
   the correlation coefficients in *metrics_pointer and *correlation_pointer, respectively. (These
   correlation coefficients are used here as temporary storage; their values are later used to
   calculate the actual correlation coefficients.) *attribute_pointer is the head of the list of
   attributes for this area. */

{
    while (matrix_pointer ≠ NULL) {
        sum_pair(matrix_pointer→attribute_value, vector_pointer→attribute_value,
                attribute_pointer→weight, metrics_pointer, correlation_pointer);
        matrix_pointer = matrix_pointer→case_next;
        vector_pointer = vector_pointer→next;
        attribute_pointer = attribute_pointer→next;
    }
    if (metrics_pointer→number_of_known_pairs ≠ 0) {
        metrics_pointer→correlation_coefficient.unweighted /=
                metrics_pointer→number_of_known_pairs;
        metrics_pointer→correlation_coefficient.weighted /=
                metrics_pointer→number_of_known_pairs;
        correlation_pointer→unweighted /=
                metrics_pointer→number_of_known_pairs;
        correlation_pointer→weighted /=
                metrics_pointer→number_of_known_pairs;
    }
}

static void
calculate_case_metrics(
        file log_stream,
        matrix_element *matrix_pointer,
        vector_element *vector_pointer,
        attribute *attribute_pointer,
        metrics_type *metrics_pointer,
        boolean *correlation_coefficients,
        cardinal level)

/* Calculates the metrics for a leading case and the instant case (their attribute values
   are pointed to by matrix_pointer and vector_pointer, respectively) and stores them in
   *metrics_pointer. *attribute_pointer is the head of the list of attributes for this area.
   *correlation_coefficients is set to TRUE, if the correlation coefficients are meaningful
   (i.e. neither of the two cases has all attribute values equal). */
```

```
{
    correlation_type vector_means,
        numerator,
        left_denominator,
        right_denominator;
    weight_type unit_weight = { FALSE, 1.0 };

    zero_correlation(&vector_means);
    zero_correlation(&numerator);
    zero_correlation(&left_denominator);
    zero_correlation(&right_denominator);

    calculate_case_means(matrix_pointer, vector_pointer, attribute_pointer,
            metrics_pointer, &vector_means);

    while (attribute_pointer ≠ NULL) {

        add_weight(log_stream, matrix_pointer→attribute_value,
                vector_pointer→attribute_value,
                attribute_pointer→weight, metrics_pointer, level);

        correlate_pair(matrix_pointer→attribute_value, vector_pointer→attribute_value,
                metrics_pointer→correlation_coefficient.unweighted, vector_means.unweighted,
                &numerator.unweighted, &left_denominator.unweighted,
                &right_denominator.unweighted, unit_weight);

        correlate_pair(matrix_pointer→attribute_value, vector_pointer→attribute_value,
                metrics_pointer→correlation_coefficient.weighted, vector_means.weighted,
                &numerator.weighted, &left_denominator.weighted,
                &right_denominator.weighted, attribute_pointer→weight);

        matrix_pointer = matrix_pointer→case_next;
        vector_pointer = vector_pointer→next;
        attribute_pointer = attribute_pointer→next;
    }
    metrics_pointer→weighted_association_coefficient =
            metrics_pointer→distance.known.finite /
            metrics_pointer→weighted_association_coefficient;

    if (Is_Zero(left_denominator.unweighted × right_denominator.unweighted))

        /* either this case or the instant case has all attribute values equal: the correlation
            coefficients are meaningless */

        metrics_pointer→correlation_coefficient.meaningless = TRUE;

    else {
            metrics_pointer→correlation_coefficient.unweighted = numerator.unweighted /
                    (floating_point) sqrt((double)
                    (left_denominator.unweighted × right_denominator.unweighted));
            metrics_pointer→correlation_coefficient.weighted = numerator.weighted /
                    (floating_point) sqrt((double)
                    (left_denominator.weighted × right_denominator.weighted));
            *correlation_coefficients = TRUE;
        }
}
```

```
static void
calculate_ideal_point_means(
        vector_element *vector_pointer_X,
        vector_element *vector_pointer_Y,
        attribute *attribute_pointer,
        metrics_type *metrics_pointer,
        correlation_type *correlation_pointer)

/* Calculates the mean attribute values for an ideal point and the instant case (their attribute
   values are pointed to by vector_pointer_X and vector_pointer_Y, respectively) and stores
   them as the correlation coefficients in *metrics_pointer and *correlation_pointer, respect-
   ively. (These correlation coefficients are used here as temporary storage; their values are
   later used to calculate the actual correlation coefficients.) *attribute_pointer is the head of
   the list of attributes for this area. */

{
    while (vector_pointer_X ≠ NULL) {
        sum_pair(vector_pointer_X→attribute_value, vector_pointer_Y→attribute_value,
                attribute_pointer→weight, metrics_pointer, correlation_pointer);
        vector_pointer_X = vector_pointer_X→next;
        vector_pointer_Y = vector_pointer_Y→next;
        attribute_pointer = attribute_pointer→next;
    }
    if (metrics_pointer→number_of_known_pairs ≠ 0) {
        metrics_pointer→correlation_coefficient.unweighted /=
                metrics_pointer→number_of_known_pairs;
        metrics_pointer→correlation_coefficient.weighted /=
                metrics_pointer→number_of_known_pairs;
        correlation_pointer→unweighted /=
                metrics_pointer→number_of_known_pairs;
        correlation_pointer→weighted /=
                metrics_pointer→number_of_known_pairs;
    }
}

static void
calculate_ideal_point_metrics(
        file log_stream,
        vector_element *vector_pointer_X,
        vector_element *vector_pointer_Y,
        attribute *attribute_pointer,
        metrics_type *metrics_pointer,
        boolean *correlation_coefficients,
        cardinal level)

/* Calculates the metrics for an ideal point and the instant case (their attribute values
   are pointed to by vector_pointer_X and vector_pointer_Y, respectively) and stores them
   in *metrics_pointer. *attribute_pointer is the head of the list of attributes for this area.
   *correlation_coefficients is set to TRUE, if the correlation coefficients are meaningful
   (i.e. neither the ideal point nor the instant case has all attribute values equal). */
```

```
{
    correlation_type vector_means,
        numerator,
        left_denominator,
        right_denominator;
    weight_type unit_weight = { FALSE, 1.0 };

    zero_correlation(&vector_means);
    zero_correlation(&numerator);
    zero_correlation(&left_denominator);
    zero_correlation(&right_denominator);

    calculate_ideal_point_means(vector_pointer_X, vector_pointer_Y, attribute_pointer,
            metrics_pointer, &vector_means);

    while (attribute_pointer ≠ NULL) {
        add_weight(log_stream, vector_pointer_X→attribute_value,
                vector_pointer_Y→attribute_value,
                attribute_pointer→weight, metrics_pointer, level);

        correlate_pair(vector_pointer_X→attribute_value, vector_pointer_Y→attribute_value,
                metrics_pointer→correlation_coefficient.unweighted, vector_means.unweighted,
                &numerator.unweighted, &left_denominator.unweighted,
                &right_denominator.unweighted, unit_weight);

        correlate_pair(vector_pointer_X→attribute_value, vector_pointer_Y→attribute_value,
                metrics_pointer→correlation_coefficient.weighted, vector_means.weighted,
                &numerator.weighted, &left_denominator.weighted,
                &right_denominator.weighted, attribute_pointer→weight);

        vector_pointer_X = vector_pointer_X→next;
        vector_pointer_Y = vector_pointer_Y→next;
        attribute_pointer = attribute_pointer→next;
    }
    metrics_pointer→weighted_association_coefficient =
            metrics_pointer→distance.known.finite /
            metrics_pointer→weighted_association_coefficient;

    if (Is_Zero(left_denominator.unweighted × right_denominator.unweighted))

        /* either this ideal point or the instant case has all attribute values equal: the correla-
            tion coefficients are meaningless */

        metrics_pointer→correlation_coefficient.meaningless = TRUE;
    else {
        metrics_pointer→correlation_coefficient.unweighted = numerator.unweighted /
                (floating_point) sqrt((double)
                (left_denominator.unweighted × right_denominator.unweighted));
            metrics_pointer→correlation_coefficient.weighted = numerator.weighted /
                (floating_point) sqrt((double)
                (left_denominator.weighted × right_denominator.weighted));
            *correlation_coefficients = TRUE;
        }
    }
```

```
static void
calculate_centroid_means(
        centroid_element *centroid_pointer,
        vector_element *vector_pointer,
        attribute *attribute_pointer,
        metrics_type *metrics_pointer,
        correlation_type *correlation_pointer)
/* Calculates the mean attribute values for a centroid and the instant case (their attribute
    values are pointed to by centroid_pointer and vector_pointer, respectively) and stores them as
    the correlation coefficients in *metrics_pointer and *correlation_pointer, respectively. (These
    correlation coefficients are used here as temporary storage; their values are later used to
    calculate the actual correlation coefficients.) *attribute_pointer is the head of the list of
    attributes for this area. */
{
    floating_point temp;
    weight_type weight;
    while (centroid_pointer ≠ NULL) {
        if ((¬centroid_pointer→unknown) ∧ (vector_pointer→attribute_value ≠ UNKNOWN)) {
            metrics_pointer→number_of_known_pairs++;
            if (Nearest_Attribute_Value(centroid_pointer→value) ≠
                    vector_pointer→attribute_value)
                metrics_pointer→number_of_known_differences++;
            weight = attribute_pointer→weight;
            metrics_pointer→weighted_association_coefficient += weight.finite;
            /* use the actual centroid value, not the nearest attribute value (as is done for
                leading cases and ideal points) */
            metrics_pointer→correlation_coefficient.unweighted += centroid_pointer→value;
            metrics_pointer→correlation_coefficient.weighted +=
                    centroid_pointer→value × weight.finite;
            (void) Attribute_Value(vector_pointer→attribute_value, &temp);
            correlation_pointer→unweighted += temp;
            correlation_pointer→weighted += temp × weight.finite;
        }
        centroid_pointer = centroid_pointer→next;
        vector_pointer = vector_pointer→next;
        attribute_pointer = attribute_pointer→next;
    }
    if (metrics_pointer→number_of_known_pairs ≠ 0) {
        metrics_pointer→correlation_coefficient.unweighted /=
                metrics_pointer→number_of_known_pairs;
        metrics_pointer→correlation_coefficient.weighted /=
                metrics_pointer→number_of_known_pairs;
        correlation_pointer→unweighted /=
                metrics_pointer→number_of_known_pairs;
        correlation_pointer→weighted /=
                metrics_pointer→number_of_known_pairs;
    }
}
```

```
static void
calculate_centroid_metrics(
        file log_stream,
        centroid_element *centroid_pointer,
        vector_element *vector_pointer,
        attribute *attribute_pointer,
        metrics_type *metrics_pointer,
        boolean *correlation_coefficients,
        cardinal level)

/* Calculates the metrics for a centroid and the instant case (their attribute values are pointed
   to by centroid_pointer and vector_pointer, respectively) and stores them in *metrics_pointer.
   *attribute_pointer is the head of the list of attributes for this area. *correlation_coefficients
   is set to TRUE, if the correlation coefficients are meaningful (i.e. neither the centroid nor
   the instant case has all attribute values equal). */

{
    correlation_type vector_means,
        numerator,
        left_denominator,
        right_denominator;
    floating_point temp,
        weight;

    zero_correlation(&vector_means);
    zero_correlation(&numerator);
    zero_correlation(&left_denominator);
    zero_correlation(&right_denominator);

    calculate_centroid_means(centroid_pointer, vector_pointer, attribute_pointer,
            metrics_pointer, &vector_means);

    while (attribute_pointer ≠ NULL) {

        if (centroid_pointer→unknown)
            add_weight(log_stream, UNKNOWN, vector_pointer→attribute_value,
                    attribute_pointer→weight, metrics_pointer, level);
        else {
            add_weight(log_stream, Nearest_Attribute_Value(centroid_pointer→value),
                    vector_pointer→attribute_value, attribute_pointer→weight,
                    metrics_pointer, level);

            if (vector_pointer→attribute_value ≠ UNKNOWN) {

if (attribute_pointer→weight.infinite)

    /* one of the attributes is infinitely weighted, so use a "pseudo-infinite"
       weight for the calculation of weighted correlation coefficients */

    weight = Very_Heavy_Indeed;
else
    weight = attribute_pointer→weight.finite;

(void) Attribute_Value(vector_pointer→attribute_value, &temp);
```

```
/* use the actual centroid value, not the nearest attribute value (as is done for
   leading cases and ideal points) */
numerator.unweighted += (centroid_pointer→value −
        metrics_pointer→correlation_coefficient.unweighted) ×
        (temp − vector_means.unweighted);
left_denominator.unweighted += (centroid_pointer→value −
        metrics_pointer→correlation_coefficient.unweighted) ×
        (centroid_pointer→value −
        metrics_pointer→correlation_coefficient.unweighted);
right_denominator.unweighted += (temp − vector_means.unweighted) ×
        (temp − vector_means.unweighted);

numerator.weighted += (centroid_pointer→value × weight −
        metrics_pointer→correlation_coefficient.weighted) ×
        (temp × weight − vector_means.weighted);
left_denominator.weighted += (centroid_pointer→value × weight −
        metrics_pointer→correlation_coefficient.weighted) ×
        (centroid_pointer→value × weight −
        metrics_pointer→correlation_coefficient.weighted);
right_denominator.weighted += (temp × weight − vector_means.weighted) ×
        (temp × weight − vector_means.weighted);
    }
}
centroid_pointer = centroid_pointer→next;
vector_pointer = vector_pointer→next;
    attribute_pointer = attribute_pointer→next;
}
metrics_pointer→weighted_association_coefficient =
        metrics_pointer→distance.known.finite /
        metrics_pointer→weighted_association_coefficient;

if (Is_Zero(left_denominator.unweighted × right_denominator.unweighted))

    /* either this centroid or the instant case has all attribute values equal: the correlation
       coefficients are meaningless */

    metrics_pointer→correlation_coefficient.meaningless = TRUE;
else {
    metrics_pointer→correlation_coefficient.unweighted = numerator.unweighted /
            (floating_point) sqrt((double)
            (left_denominator.unweighted × right_denominator.unweighted));
    metrics_pointer→correlation_coefficient.weighted = numerator.weighted /
            (floating_point) sqrt((double)
            (left_denominator.weighted × right_denominator.weighted));
        *correlation_coefficients = TRUE;
    }
}

static weight_list_element *
result_weight(
        weight_list_element *weights_pointer,
        result *result_pointer,
        result *target_result_pointer)
```

```
/* Returns the result weight (from the list of result weights pointed to by weights_pointer)
   which corresponds to the result pointed to by target_result_pointer. (*result_pointer is the
   head of the list of results for this area.) */

{
    while (result_pointer ≠ target_result_pointer) {
        weights_pointer = weights_pointer→next;
        result_pointer = result_pointer→next;
    }
    return weights_pointer;
}

static void
calculate_specified_directions(
        attribute *attribute_pointer,
        vector_element *vector_pointer,
        result *result_head)

/* For every attribute, if the attribute value in the instant case is directed towards a result,
   adds the weight of the attribute to the specified direction for that result. *attribute_pointer
   is the head of the list of attributes for this area. The attribute values of the instant case
   are pointed to by vector_pointer. */

{
    direction_list_element *direction_pointer;
    weight_list_element *weights_pointer;

    while (attribute_pointer ≠ NULL) {
        switch (vector_pointer→attribute_value) {
        case YES:
            direction_pointer = attribute_pointer→yes_direction_head;
            break;
        case NO:
            direction_pointer = attribute_pointer→no_direction_head;
            break;
        case UNKNOWN:
            direction_pointer = attribute_pointer→unknown_direction_head;
            break;
        }
        while (direction_pointer ≠ NULL) {
            weights_pointer = result_weight(attribute_pointer→weights_head,
                    result_head, direction_pointer→result);
            if (weights_pointer→weight.infinite)
                direction_pointer→result→specified_direction.infinite++;
            else
                direction_pointer→result→specified_direction.finite +=
                    weights_pointer→weight.finite;
            direction_pointer = direction_pointer→next;
        }
        vector_pointer = vector_pointer→next;
        attribute_pointer = attribute_pointer→next;
    }
}
```

```
static void
calculate_other_directions(
        area *area_pointer,
        vector_element *vector_pointer)

/* For every attribute, if only one ideal point has an attribute value matching that of the instant
   case, adds the weight of the attribute to the ideal point direction for that ideal point's result.
   Similarly, for every attribute, if only one centroid has an attribute value matching that of
   the instant case, adds the weight of the attribute to the centroid direction for that centroid's
   result. The attribute values of the instant case are pointed to by vector_pointer.

   UNKNOWN values are ignored when counting matches. This differs from the calculation of
   specified directions (in calculate_specified_directions()) because an UNKNOWN value in an
   ideal point could mean "don't know," while in a centroid it just indicates an absence of
   values; by contrast, an UNKNOWN specified direction means "an UNKNOWN value for this
   attribute suggests this result." */

{
    result *result_pointer,
      *ideal_point_matching_result,
      *centroid_matching_result;
    attribute *attribute_pointer = area_pointer→attribute_head;
    vector_element *ideal_point_pointer;
    centroid_element *centroid_pointer;
    cardinal count,
      ideal_point_matches_count,
      centroid_matches_count;
    weight_list_element *weights_pointer;

    while (vector_pointer ≠ NULL) {

        if (vector_pointer→attribute_value ≠ UNKNOWN) {

            /* count the number of ideal points and centroids with the same value for this
               attribute as has the instant case */

            ideal_point_matches_count = 0;
            centroid_matches_count = 0;
            result_pointer = area_pointer→result_head;
            while (result_pointer ≠ NULL) {
                ideal_point_pointer = result_pointer→ideal_point_head;
                centroid_pointer = result_pointer→centroid_head;
                for (count = 1; count < attribute_pointer→number; count++) {
                    if (ideal_point_pointer ≠ NULL)
                        ideal_point_pointer = ideal_point_pointer→next;
                    if (centroid_pointer ≠ NULL)
                        centroid_pointer = centroid_pointer→next;
                }
```

```
if ((ideal_point_pointer ≠ NULL) ∧
        (ideal_point_pointer→attribute_value ≡
                vector_pointer→attribute_value)) {
    ideal_point_matches_count++;
    ideal_point_matching_result = result_pointer;
}
if ((centroid_pointer ≠ NULL) ∧
        ((centroid_pointer→unknown ∧
                (vector_pointer→attribute_value ≡ UNKNOWN)) ∨
            (Nearest_Attribute_Value(centroid_pointer→value) ≡
                vector_pointer→attribute_value))) {
        centroid_matches_count++;
        centroid_matching_result = result_pointer;
    }
    result_pointer = result_pointer→next;
}

if (ideal_point_matches_count ≡ 1) {

    /* add the weight of the attribute to the ideal point direction for the matching
       ideal point's result */

    weights_pointer = result_weight(attribute_pointer→weights_head,
            area_pointer→result_head, ideal_point_matching_result);

    if (weights_pointer→weight.infinite)
        ideal_point_matching_result→ideal_point_direction.infinite++;
    else
        ideal_point_matching_result→ideal_point_direction.finite +=
                weights_pointer→weight.finite;
}
if (centroid_matches_count ≡ 1) {

    /* add the weight of the attribute to the centroid direction for the matching
       centroid's result */

    weights_pointer = result_weight(attribute_pointer→weights_head,
            area_pointer→result_head, centroid_matching_result);

    if (weights_pointer→weight.infinite)
        centroid_matching_result→centroid_direction.infinite++;
    else
        centroid_matching_result→centroid_direction.finite +=
                weights_pointer→weight.finite;
            }
        }
        attribute_pointer = attribute_pointer→next;
        vector_pointer = vector_pointer→next;
    }
}
```

static void
find_nearest_and_strongest(*area* ∗*area_pointer*)

/∗ Finds the nearest result, nearest ideal point, nearest centroid, and strongest directions (specified, ideal point, and centroid) in the area pointed to by *area_pointer*, and adjusts various pointers in ∗*area_pointer* to point to them. ∗/

```
{
    result *result_pointer,
    *equidistant_pointer,
    *nearest_result = NULL;
relative_distance_type relative_distance;
kase *nearest_neighbour;

area_pointer→nearest_result = NULL;
area_pointer→nearest_ideal_point = NULL;
area_pointer→nearest_centroid = NULL;
area_pointer→strongest_specified_direction = NULL;
area_pointer→strongest_ideal_point_direction = NULL;
area_pointer→strongest_centroid_direction = NULL;

/* for every result ... */

for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
        result_pointer = result_pointer→next) {

result_pointer→equidistant_next = NULL;

if (result_pointer→nearest_known_compared_with_unknown ≡ FURTHER)
    nearest_neighbour = result_pointer→nearest_unknown_case;
else
    nearest_neighbour = result_pointer→nearest_known_case;

if (nearest_neighbour ≠ NULL)

    /* this result has a nearest neighbour (i.e. it has at least one case) */

    if (nearest_result ≡ NULL)

        /* no nearest result has been found yet, so this result is the nearest so far */

        nearest_result = result_pointer;

    else {

    /* a nearest result has previously been found, so compare the nearest neighbour
        for this result with the nearest neighbour for that nearest result */

if (nearest_result→nearest_known_compared_with_unknown ≡ FURTHER)
    relative_distance = Relative_Distance(nearest_neighbour→metrics.distance,
            nearest_result→nearest_unknown_case→metrics.distance);
else
    relative_distance = Relative_Distance(nearest_neighbour→metrics.distance,
            nearest_result→nearest_known_case→metrics.distance);
```

```
    if (relative_distance ≡ EQUIDISTANT) {

        /* the two cases are equidistant from the instant case, so add this result to
           the end of the list of equidistant results */

        for (equidistant_pointer = nearest_result;
                equidistant_pointer→equidistant_next ≠ NULL;
                equidistant_pointer = equidistant_pointer→equidistant_next);
        equidistant_pointer→equidistant_next = result_pointer;
    } else if (relative_distance ≡ NEARER)

        /* the nearest neighbour for this result is nearer to the instant case than the
           previous nearest neighbour, so this result becomes the nearest result */

        nearest_result = result_pointer;
    }

/* check whether this result has the nearest ideal point */

result_pointer→equidistant_ideal_point_next = NULL;
if (result_pointer→ideal_point_head ≠ NULL) {
    if (area_pointer→nearest_ideal_point ≡ NULL)
        area_pointer→nearest_ideal_point = result_pointer;
    else if ((relative_distance =
                    Relative_Distance(result_pointer→ideal_point_metrics.distance,
                        area_pointer→nearest_ideal_point→
                        ideal_point_metrics.distance)) ≡ EQUIDISTANT) {
        for (equidistant_pointer = area_pointer→nearest_ideal_point;
                equidistant_pointer→equidistant_ideal_point_next ≠ NULL;
                equidistant_pointer = equidistant_pointer→equidistant_ideal_point_next);
        equidistant_pointer→equidistant_ideal_point_next = result_pointer;
    } else if (relative_distance ≡ NEARER)
        area_pointer→nearest_ideal_point = result_pointer;
}

/* check whether this result has the nearest centroid */

result_pointer→equidistant_centroid_next = NULL;
if (result_pointer→centroid_head ≠ NULL) {
    if (area_pointer→nearest_centroid ≡ NULL)
        area_pointer→nearest_centroid = result_pointer;
    else if ((relative_distance =
                    Relative_Distance(result_pointer→centroid_metrics.distance,
                        area_pointer→nearest_centroid→
                        centroid_metrics.distance)) ≡ EQUIDISTANT) {
        for (equidistant_pointer = area_pointer→nearest_centroid;
                equidistant_pointer→equidistant_centroid_next ≠ NULL;
                equidistant_pointer = equidistant_pointer→equidistant_centroid_next);
        equidistant_pointer→equidistant_centroid_next = result_pointer;
    } else if (relative_distance ≡ NEARER)
        area_pointer→nearest_centroid = result_pointer;
}
```

```
/* check whether this result has the strongest specified direction */

result_pointer→equidistant_specified_direction_next = NULL;
if (area_pointer→strongest_specified_direction ≡ NULL)
    area_pointer→strongest_specified_direction = result_pointer;
else if ((relative_distance = relative_subdistance(result_pointer→specified_direction,
                                area_pointer→strongest_specified_direction→
                            specified_direction)) ≡ EQUIDISTANT) {
    for (equidistant_pointer = area_pointer→strongest_specified_direction;
            equidistant_pointer→equidistant_specified_direction_next ≠ NULL;
            equidistant_pointer =
            equidistant_pointer→equidistant_specified_direction_next);
    equidistant_pointer→equidistant_specified_direction_next = result_pointer;
} else if (relative_distance ≡ FURTHER)
    area_pointer→strongest_specified_direction = result_pointer;

/* check whether this result has the strongest ideal point direction */

result_pointer→equidistant_ideal_point_direction_next = NULL;
if (area_pointer→strongest_ideal_point_direction ≡ NULL)
    area_pointer→strongest_ideal_point_direction = result_pointer;
else if ((relative_distance = relative_subdistance(result_pointer→ideal_point_direction,
                                area_pointer→strongest_ideal_point_direction→
                            ideal_point_direction)) ≡ EQUIDISTANT) {
    for (equidistant_pointer = area_pointer→strongest_ideal_point_direction;
            equidistant_pointer→equidistant_ideal_point_direction_next ≠ NULL;
            equidistant_pointer =
            equidistant_pointer→equidistant_ideal_point_direction_next);
    equidistant_pointer→equidistant_ideal_point_direction_next = result_pointer;
} else if (relative_distance ≡ FURTHER)
    area_pointer→strongest_ideal_point_direction = result_pointer;

/* check whether this result has the strongest centroid direction */

result_pointer→equidistant_centroid_direction_next = NULL;
if (area_pointer→strongest_centroid_direction ≡ NULL)
    area_pointer→strongest_centroid_direction = result_pointer;
else if ((relative_distance = relative_subdistance(result_pointer→centroid_direction,
                                area_pointer→strongest_centroid_direction→
                            centroid_direction)) ≡ EQUIDISTANT) {
    for (equidistant_pointer = area_pointer→strongest_centroid_direction;
            equidistant_pointer→equidistant_centroid_direction_next ≠ NULL;
            equidistant_pointer =
            equidistant_pointer→equidistant_centroid_direction_next);
        equidistant_pointer→equidistant_centroid_direction_next = result_pointer;
    } else if (relative_distance ≡ FURTHER)
        area_pointer→strongest_centroid_direction = result_pointer;
}
area_pointer→nearest_result = nearest_result;
```

```
/* ensure that none of the strongest directions is so small as to be effectively zero */
if (Is_Zero_Subdistance(area_pointer→strongest_specified_direction→
                specified_direction))
    area_pointer→strongest_specified_direction = NULL;
if (Is_Zero_Subdistance(area_pointer→strongest_ideal_point_direction→
                ideal_point_direction))
    area_pointer→strongest_ideal_point_direction = NULL;
    if (Is_Zero_Subdistance(area_pointer→strongest_centroid_direction→
                    centroid_direction))
        area_pointer→strongest_centroid_direction = NULL;
}

static void
resolve_equidistant_results(
        file log_stream,
        area *area_pointer,
        cardinal level)

/* Chooses between two or more nearest results (and sets area_pointer→nearest_result appro-
    priately) by reference to the rank of the courts involved in the nearest neighbours, and the
    recentness of those cases. Issues a warning as to how the equidistance has been resolved. */

{
    result *result_pointer,
      *highest_ranking_result,
      *most_recent_result;
cardinal highest_ranking_court = 0,
    most_recent_year = 0;
kase *nearest_neighbour;
boolean one_highest_ranking_court = FALSE;
boolean one_most_recent_year = FALSE;

/* for each equidistant result ... */

for (result_pointer = area_pointer→nearest_result; result_pointer ≠ NULL;
        result_pointer = result_pointer→equidistant_next) {

    /* find the nearest neighbour for this result */

    if (result_pointer→nearest_known_compared_with_unknown ≡ FURTHER)
        nearest_neighbour = result_pointer→nearest_unknown_case;
    else
        nearest_neighbour = result_pointer→nearest_known_case;

    if ((nearest_neighbour→court_string ≠ NULL) ∧
        (nearest_neighbour→court_rank ≠ 0)) {

    /* this case has a court, with a rank */

    if ((highest_ranking_court ≡ 0) ∨
            (nearest_neighbour→court_rank < highest_ranking_court)) {

        /* this is the first court for any result, or this court is more important than the
            highest ranking court yet found */
```

```
        highest_ranking_result = result_pointer;
        highest_ranking_court = nearest_neighbour→court_rank;
        one_highest_ranking_court = TRUE;

        /* this must also be the most recent case for this rank so far */

        most_recent_result = result_pointer;
        most_recent_year = nearest_neighbour→year;
        one_most_recent_year = TRUE;
    } else if (nearest_neighbour→court_rank ≡ highest_ranking_court) {
        /* there are two or more equidistant cases with courts of this rank */

        one_highest_ranking_court = FALSE;

        if (nearest_neighbour→year > most_recent_year) {

            /* this is the most recent case for this rank so far */

            most_recent_result = result_pointer;
            most_recent_year = nearest_neighbour→year;
            one_most_recent_year = TRUE;

        } else if (nearest_neighbour→year ≡ most_recent_year)

            /* this case is exactly as old as the most recent case yet found for this
               rank */

            one_most_recent_year = FALSE;
    }
} else {

    /* this case has no court, or a court with no rank */

    if ((most_recent_year ≡ 0) ∨
            (nearest_neighbour→year > most_recent_year)) {

        /* this is the first case for any result, or this case is more recent than the most
           recent yet found */

        most_recent_result = result_pointer;
        most_recent_year = nearest_neighbour→year;
        one_most_recent_year = TRUE;

    } else if (nearest_neighbour→year ≡ most_recent_year)

        /* this decision is exactly as old as the most recent decision yet found */

        one_most_recent_year = FALSE;
    }
}
if (one_highest_ranking_court) {

    /* there was only one highest ranking court, so choose that case's result */

    area_pointer→nearest_result = highest_ranking_result;
    warning(log_stream,
            "equidistant␣results;␣nearest␣result␣chosen␣"
            "on␣the␣basis␣of␣rank", level);
    return;
}
```

```
    if (one_most_recent_year) {
        /* there was only one case decided as recently as this, so choose that case's result */
        area_pointer→nearest_result = most_recent_result;
        warning(log_stream,
                "equidistant results; nearest result chosen "
                "on the basis of recentness", level);
        return;
    }
    error_exit(log_stream, "can't choose between equidistant results");
}

extern void
Calculate_Distances(
        file distances_stream,
        file log_stream,
        area *area_pointer,
        case_law_specification case_law,
        vector_element *facts_head,
        boolean hypothetical,
        cardinal number,
        cardinal level)
/* Calculates the distances between the instant case and every leading case, ideal point
   and centroid in the area pointed to by area_pointer, and writes a table of distances to
   distances_stream (if it is not NULL). If number is not zero then the instant case is actually
   a hypothetical (if hypothetical is TRUE) or an instantiation, and number is its number. */
{
    result *result_pointer;
    kase *case_pointer,
      *equidistant_pointer;
    relative_distance_type relative_distance;

    /* for every result ... */

    for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
            result_pointer = result_pointer→next) {

        /* initialize result details */

        result_pointer→nearest_known_case = NULL;
        result_pointer→nearest_unknown_case = NULL;
        zero_metrics(&result_pointer→ideal_point_metrics);
        zero_metrics(&result_pointer→centroid_metrics);
        zero_subdistance(&result_pointer→specified_direction);
        zero_subdistance(&result_pointer→ideal_point_direction);
        zero_subdistance(&result_pointer→centroid_direction);
```

```
/* for every case with this result ... */

for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
        case_pointer = case_pointer→next) {

    /* initialize case details */

    zero_metrics(&case_pointer→metrics);
    case_pointer→equidistant_known_next = NULL;
    case_pointer→equidistant_unknown_next = NULL;

    calculate_case_metrics(log_stream, case_pointer→matrix_head,
            facts_head, area_pointer→attribute_head, &case_pointer→metrics,
            &area_pointer→correlation_coefficients, level);

    if (¬Is_Zero_Subdistance(case_pointer→metrics.distance.unknown)) {

        /* the case has unknown distance, so check to see whether it is the nearest
           unknown neighbour */

        if (result_pointer→nearest_unknown_case ≡ NULL)
            result_pointer→nearest_unknown_case = case_pointer;
        else if ((relative_distance = Relative_Distance(case_pointer→metrics.distance,
                                result_pointer→nearest_unknown_case→
                            metrics.distance)) ≡ EQUIDISTANT) {
            for (equidistant_pointer = result_pointer→nearest_unknown_case;
                    equidistant_pointer→equidistant_unknown_next ≠ NULL;
                    equidistant_pointer
                    = equidistant_pointer→equidistant_unknown_next);
                equidistant_pointer→equidistant_unknown_next = case_pointer;
            } else if (relative_distance ≡ NEARER)
                result_pointer→nearest_unknown_case = case_pointer;

    } else {

        /* the case has no unknown distance, so check to see whether it is the nearest
           known neighbour */

        if (result_pointer→nearest_known_case ≡ NULL)
            result_pointer→nearest_known_case = case_pointer;
        else if ((relative_distance = Relative_Distance(case_pointer→metrics.distance,
                                result_pointer→nearest_known_case→
                            metrics.distance)) ≡ EQUIDISTANT) {
            for (equidistant_pointer = result_pointer→nearest_known_case;
                    equidistant_pointer→equidistant_known_next ≠ NULL;
                    equidistant_pointer = equidistant_pointer→equidistant_known_next);
                equidistant_pointer→equidistant_known_next = case_pointer;
            } else if (relative_distance ≡ NEARER)
                result_pointer→nearest_known_case = case_pointer;
        }
    }
```

```
    /* note which of the nearest known neighbour and the nearest unknown neighbour is
       the nearest neighbour */
    if ((result_pointer→nearest_known_case ≡ NULL) ∧
            (result_pointer→nearest_unknown_case ≡ NULL))
        result_pointer→nearest_known_compared_with_unknown = EQUIDISTANT;
    else if (result_pointer→nearest_known_case ≡ NULL)
        result_pointer→nearest_known_compared_with_unknown = FURTHER;
    else if (result_pointer→nearest_unknown_case ≡ NULL)
        result_pointer→nearest_known_compared_with_unknown = NEARER;
    else
        result_pointer→nearest_known_compared_with_unknown =
                Relative_Distance(result_pointer→nearest_known_case→metrics.distance,
                result_pointer→nearest_unknown_case→metrics.distance);
    if (result_pointer→ideal_point_head ≠ NULL) {
        calculate_ideal_point_metrics(log_stream, result_pointer→ideal_point_head,
                facts_head, area_pointer→attribute_head,
                &result_pointer→ideal_point_metrics,
                &area_pointer→correlation_coefficients, level);
    }
    calculate_centroid_metrics(log_stream, result_pointer→centroid_head,
            facts_head, area_pointer→attribute_head, &result_pointer→centroid_metrics,
            &area_pointer→correlation_coefficients, level);
}
calculate_specified_directions(area_pointer→attribute_head, facts_head,
        area_pointer→result_head);
calculate_other_directions(area_pointer, facts_head);
find_nearest_and_strongest(area_pointer);
if (area_pointer→nearest_result→equidistant_next ≠ NULL)
    /* there are two or more equidistant results, so choose one of them */
    resolve_equidistant_results(log_stream, area_pointer, level);
if (distances_stream ≠ NULL) {
    if (number ≡ 0)
        /* the instant case is the uninstantiated and unhypothesized instant case */
        fprintf(distances_stream, "%s{%s area}\n\n"
                "%s{Instant case}\n\n", Heading, area_pointer→identifier, Subheading);
    else if (hypothetical)
        /* the instant case is hypothetical number */
        fprintf(distances_stream, "%s{Hypothetical %u}\n\n", Subheading, number);
    else
        /* the instant case is instantiation number */
        fprintf(distances_stream, "%s{Instantiation %u}\n\n", Subheading, number);
    Write_Matrix(distances_stream, area_pointer, facts_head,
            case_law.court_head, hypothetical, number);
  }
}
```

12

The Reporter module

reporter.h

/* This is the header file for the Reporter module. It is also included by the Cases module. */

/* external function */

extern void
Write_Report(
file report_stream,
file log_stream,
*area *area_pointer*,
*vector_element *facts_head*,
*vector_element *original_facts*,
boolean verbose,
boolean hypothetical,
boolean same_result,
cardinal number,
cardinal level);

reporter.c

/* This is the implementation file for the Reporter module. */

```
#include ⟨stdio.h⟩
#include "shyster.h"
#include "cases.h"
#include "reporter.h"
#include "dumper.h"
#include "odometer.h"
```

```
static void
warning(
        file stream,
        const string message,
        cardinal level)
{
    Write_Warning_Message(stream, "Reporter", message, level);
}
```

```
static void
list_facts(
        file report_stream,
        vector_element *vector_pointer,
        attribute *attribute_pointer,
        cardinal count)
```

/* Lists the facts pointed to by *vector_pointer* by writing the appropriate string (YES, NO, or UNKNOWN) for each attribute. **attribute_pointer* is the head of the list of attributes for this area. *count* is the number of attributes. */

```
{
    /* while there are still facts to list ... */
    while ((attribute_pointer ≠ NULL) ∧ (count ≠ 0)) {
        if (count ≡ 1)
            Write(report_stream, vector_pointer→attribute_value ≡ YES ?
                    attribute_pointer→yes : vector_pointer→attribute_value ≡ NO ?
                    attribute_pointer→no : attribute_pointer→unknown, ".", 1, Hang);
        else if (count ≡ 2)
            Write(report_stream, vector_pointer→attribute_value ≡ YES ?
                    attribute_pointer→yes : vector_pointer→attribute_value ≡ NO ?
                    attribute_pointer→no : attribute_pointer→unknown, "; and", 1, Hang);
        else
            Write(report_stream, vector_pointer→attribute_value ≡ YES ?
                    attribute_pointer→yes : vector_pointer→attribute_value ≡ NO ?
                    attribute_pointer→no : attribute_pointer→unknown, ";", 1, Hang);
        vector_pointer = vector_pointer→next;
        attribute_pointer = attribute_pointer→next;
        count--;
    }
}
```

```
static void
list_equidistant_cases_known_first(
        file report_stream,
        kase *known_case_pointer,
        kase *unknown_case_pointer,
        boolean short_names,
        boolean and)
```

/* Lists the case pointed to by *known_case_pointer* (and any known equidistant cases), then the case pointed to by *unknown_case_pointer* (and any unknown equidistant cases). Writes short case names, if *short_names* is *TRUE*. Writes "and" between the last two cases in the list, if *and* is *TRUE*; writes "or", otherwise. */

```
{
    /* while there are still known cases to list ... */
    while (known_case_pointer ≠ NULL) {
        /* write the case name with appropriate trailing characters */
        fprintf(report_stream, "{\\it␣%s", short_names ?
                known_case_pointer→short_name : known_case_pointer→name);
        if ((known_case_pointer→equidistant_known_next ≡ NULL) ∧
            (unknown_case_pointer ≡ NULL))
            /* this is the last case to list */
            fprintf(report_stream, "\\/}");
        else if (((known_case_pointer→equidistant_known_next ≠ NULL) ∧
                    (known_case_pointer→equidistant_known_next→
                        equidistant_known_next ≡ NULL) ∧
                    (unknown_case_pointer ≡ NULL)) ∨
                ((known_case_pointer→equidistant_known_next ≡ NULL) ∧
                    (unknown_case_pointer ≠ NULL) ∧
                    (unknown_case_pointer→equidistant_unknown_next ≡ NULL)))
            /* this is the penultimate case to list */
            fprintf(report_stream, "\\/}␣%s\n", and ? "and" : "or");
        else
            fprintf(report_stream, "},\n");
        known_case_pointer = known_case_pointer→equidistant_known_next;
    }
    /* while there are still unknown cases to list ... */
    while (unknown_case_pointer ≠ NULL) {
        /* write the case name with appropriate trailing characters */
        fprintf(report_stream, "{\\it␣%s", short_names ?
                unknown_case_pointer→short_name : unknown_case_pointer→name);
        if (unknown_case_pointer→equidistant_unknown_next ≡ NULL)
            /* this is the last case to list */
            fprintf(report_stream, "\\/}");
        else if (unknown_case_pointer→equidistant_unknown_next→
                equidistant_unknown_next ≡ NULL)
            /* this is the penultimate case to list */
            fprintf(report_stream, "\\/}␣%s\n", and ? "and" : "or");
        else
            fprintf(report_stream, "},\n");
        unknown_case_pointer = unknown_case_pointer→equidistant_unknown_next;
    }
}
```

```
static void
list_equidistant_cases_unknown_first(
        file report_stream,
        kase *known_case_pointer,
        kase *unknown_case_pointer,
        boolean short_names,
        boolean and)
```

/* Lists the case pointed to by *unknown_case_pointer* (and any unknown equidistant cases), then the case pointed to by *known_case_pointer* (and any known equidistant cases). Writes short case names, if *short_names* is *TRUE*. Writes "and" between the last two cases in the list, if *and* is *TRUE*; writes "or", otherwise. */

```
{
    /* while there are still unknown cases to list ... */
    while (unknown_case_pointer ≠ NULL) {
        /* write the case name with appropriate trailing characters */
        fprintf(report_stream, "{\\it␣%s", short_names ?
                unknown_case_pointer→short_name : unknown_case_pointer→name);
        if ((unknown_case_pointer→equidistant_unknown_next ≡ NULL) ∧
                (known_case_pointer ≡ NULL))
            /* this is the last case to list */
            fprintf(report_stream, "\\/}");
        else if (((unknown_case_pointer→equidistant_unknown_next ≠ NULL) ∧
                        (unknown_case_pointer→equidistant_unknown_next→
                                equidistant_unknown_next ≡ NULL) ∧
                        (known_case_pointer ≡ NULL)) ∨
                ((unknown_case_pointer→equidistant_unknown_next ≡ NULL) ∧
                        (known_case_pointer ≠ NULL) ∧
                        (known_case_pointer→equidistant_known_next ≡ NULL)))
            /* this is the penultimate case to list */
            fprintf(report_stream, "\\/}␣%s\n", and ? "and" : "or");
        else
            fprintf(report_stream, "},\n");
        unknown_case_pointer = unknown_case_pointer→equidistant_unknown_next;
    }
    /* while there are still known cases to list ... */
    while (known_case_pointer ≠ NULL) {
        /* write the case name with appropriate trailing characters */
        fprintf(report_stream, "{\\it␣%s", short_names ?
                known_case_pointer→short_name : known_case_pointer→name);
        if (known_case_pointer→equidistant_known_next ≡ NULL)
            /* this is the last case to list */
            fprintf(report_stream, "\\/}");
```

```
    else if (known_case_pointer→equidistant_known_next→equidistant_known_next ≡ NULL)
        /* this is the penultimate case to list */
        fprintf(report_stream, "\\/}␣%s\n", and ? "and" : "or");
    else
        fprintf(report_stream, "},\n");
    known_case_pointer = known_case_pointer→equidistant_known_next;
    }
}

static void
list_equidistant_cases(
        file report_stream,
        kase *known_case_pointer,
        kase *unknown_case_pointer,
        boolean unknown_first,
        boolean short_names,
        boolean and)

/* Lists the case pointed to by known_case_pointer (and any known equidistant cases) and the
   case pointed to by unknown_case_pointer (and any unknown equidistant cases). Lists the
   unknown cases first, if unknown_first is TRUE. Writes short case names, if short_names is
   TRUE. Writes "and" between the last two cases in the list, if and is TRUE; writes "or",
   otherwise. */

{
    if (unknown_first)
        list_equidistant_cases_unknown_first(report_stream, known_case_pointer,
                unknown_case_pointer, short_names, and);
    else
        list_equidistant_cases_known_first(report_stream, known_case_pointer,
                unknown_case_pointer, short_names, and);
}

static void
state_opinion(
        file report_stream,
        result *result_pointer,
        kase *known_case_pointer,
        kase *unknown_case_pointer,
        boolean unknown_first)

/* States its opinion: that, following the cases pointed to by known_case_pointer and
   unknown_case_pointer, the result will be that which is pointed to by result_pointer. Lists
   the unknown cases first, if unknown_first is TRUE. */

{
    fprintf(report_stream, "---following␣\\frenchspacing\n");
    list_equidistant_cases(report_stream, known_case_pointer, unknown_case_pointer,
            unknown_first, FALSE, TRUE);
    fprintf(report_stream, "\\nonfrenchspacing---%%\n"
            "%s.\n\n", result_pointer→string);
}
```

```
static void
state_counter_opinion(
        file report_stream,
        result *result_pointer,
        kase *known_case_pointer,
        kase *unknown_case_pointer,
        boolean unknown_first)

/* States a counter opinion:  that, if the cases pointed to by known_case_pointer and
    unknown_case_pointer are followed, the result will be that which is pointed to by
    result_pointer. Lists the unknown cases first, if unknown_first is TRUE. */

{
    boolean plural = FALSE;

    fprintf(report_stream, "%s If \\frenchspacing\n", Skip);

    list_equidistant_cases(report_stream, known_case_pointer, unknown_case_pointer,
            unknown_first, FALSE, FALSE);

    if (known_case_pointer ≠ NULL) {
        if (unknown_case_pointer ≠ NULL)
            plural = TRUE;
        else if (known_case_pointer→equidistant_known_next ≠ NULL)
            plural = TRUE;
    } else if (unknown_case_pointer ≠ NULL)
        if (unknown_case_pointer→equidistant_unknown_next ≠ NULL)
            plural = TRUE;

    fprintf(report_stream, " \\nonfrenchspacing\n"
            "%s followed then %s.\n\n",
            plural ? "are" : "is", result_pointer→string);
}

static cardinal
number_of_similarities(
        matrix_element *matrix_pointer,
        vector_element *vector_pointer)

/* Returns the number of similarities in attribute value pairs between a leading case and the
    instant case (their attribute values are pointed to by matrix_pointer and vector_pointer,
    respectively), where both cases have known attribute values. */

{
    cardinal count = 0;

    /* while there are still attribute values to compare ... */

    while (matrix_pointer ≠ NULL) {

        if ((matrix_pointer→attribute_value ≠ UNKNOWN) ∧
                (matrix_pointer→attribute_value ≡ vector_pointer→attribute_value))

            /* the corresponding attribute values are identical and known */

            count++;
```

```
        matrix_pointer = matrix_pointer→case_next;
        vector_pointer = vector_pointer→next;
    }
    return count;
}

static void
list_similarities(
        file report_stream,
        matrix_element *matrix_pointer,
        vector_element *vector_pointer,
        attribute *attribute_pointer,
        cardinal count)

/* Lists the similarities between a leading case and the instant case (their attribute values are
    pointed to by matrix_pointer and vector_pointer, respectively), where both cases have known
    attribute values, by writing the appropriate string (YES or NO) for the similar attributes.
    *attribute_pointer is the head of the list of attributes for this area. count is the number of
    similarities. */

{
    /* while there are still attribute values to compare, and not all of the similarities have been
        listed ... */

    while ((attribute_pointer ≠ NULL) ∧ (count ≠ 0)) {

        if ((matrix_pointer→attribute_value ≠ UNKNOWN) ∧
                (matrix_pointer→attribute_value ≡ vector_pointer→attribute_value)) {

            /* the corresponding attribute values are identical and known, so write the relevant
                string with appropriate trailing characters */

            if (count ≡ 1)
                Write(report_stream, matrix_pointer→attribute_value ≡ YES ?
                        attribute_pointer→yes : attribute_pointer→no, ".\n", 1, Hang);
            else if (count ≡ 2)
                Write(report_stream, matrix_pointer→attribute_value ≡ YES ?
                        attribute_pointer→yes : attribute_pointer→no, "; and", 1, Hang);
            else
                Write(report_stream, matrix_pointer→attribute_value ≡ YES ?
                        attribute_pointer→yes : attribute_pointer→no, ";", 1, Hang);
            count--;
        }
        matrix_pointer = matrix_pointer→case_next;
        vector_pointer = vector_pointer→next;
        attribute_pointer = attribute_pointer→next;
    }
}
```

```
static cardinal
number_of_known_differences(
        matrix_element *matrix_pointer,
        vector_element *vector_pointer)
```

/* Returns the number of differences in attribute value pairs between a leading case and the instant case (their attribute values are pointed to by *matrix_pointer* and *vector_pointer*, respectively), where the leading case has a known attribute value. */

```
{
    cardinal count = 0;
    /* while there are still attribute values to compare ... */
    while (matrix_pointer ≠ NULL) {
        if ((matrix_pointer→attribute_value ≠ UNKNOWN) ∧
                (matrix_pointer→attribute_value ≠ vector_pointer→attribute_value))
            /* the corresponding attribute values are different, and the leading case's value is
               known, so increment the count */
            count++;
        matrix_pointer = matrix_pointer→case_next;
        vector_pointer = vector_pointer→next;
    }
    return count;
}
```

```
static void
list_known_differences(
        file report_stream,
        matrix_element *matrix_pointer,
        vector_element *vector_pointer,
        attribute *attribute_pointer,
        cardinal count)
```

/* Lists the differences between a leading case and the instant case (their attribute values are pointed to by *matrix_pointer* and *vector_pointer*, respectively), where the leading case has a known attribute value, by writing the appropriate string (YES or NO) for the different attributes. **attribute_pointer* is the head of the list of attributes for this area. *count* is the number of differences. */

```
{
    /* while there are still attribute values to compare, and not all of the differences have been
       listed ... */
    while ((attribute_pointer ≠ NULL) ∧ (count ≠ 0)) {
        if ((matrix_pointer→attribute_value ≠ UNKNOWN) ∧
                (matrix_pointer→attribute_value ≠ vector_pointer→attribute_value)) {
            /* the corresponding attribute values are different, and the leading case's value is
               known, so write the relevant string with appropriate trailing characters */
            if (count ≡ 1)
                Write(report_stream, matrix_pointer→attribute_value ≡ YES ?
                        attribute_pointer→yes : attribute_pointer→no, ". ", 1, Hang);
```

```
    else if (count ≡ 2)
        Write(report_stream, matrix_pointer→attribute_value ≡ YES ?
                attribute_pointer→yes : attribute_pointer→no, "; and", 1, Hang);
    else
        Write(report_stream, matrix_pointer→attribute_value ≡ YES ?
                attribute_pointer→yes : attribute_pointer→no, ";", 1, Hang);
    count--;
}
        matrix_pointer = matrix_pointer→case_next;
        vector_pointer = vector_pointer→next;
        attribute_pointer = attribute_pointer→next;
    }
}

static cardinal
number_of_unknowns(
        matrix_element *matrix_pointer)

/* Returns the number of UNKNOWNs in the leading case whose attribute values are pointed
   to by matrix_pointer. */

{
    cardinal count = 0;

    /* while there are still attribute values to check ... */

    while (matrix_pointer ≠ NULL) {
        if (matrix_pointer→attribute_value ≡ UNKNOWN)
            count++;
        matrix_pointer = matrix_pointer→case_next;
    }
    return count;
}

static void
list_unknowns(
        file report_stream,
        matrix_element *matrix_pointer,
        attribute *attribute_pointer,
        cardinal count)

/* Lists the UNKNOWN string for each unknown attribute in the leading case whose attribute
   values are pointed to by matrix_pointer. *attribute_pointer is the head of the list of attributes
   for this area. count is the number of UNKNOWNs. */

{
    /* while there are still attribute values to compare, and not all of the UNKNOWNs have
       been listed ... */

    while ((attribute_pointer ≠ NULL) ∧ (count ≠ 0)) {

        if (matrix_pointer→attribute_value ≡ UNKNOWN) {
            if (count ≡ 1)
                Write(report_stream, attribute_pointer→unknown, ".", 1, Hang);
```

```
        else if (count ≡ 2)
            Write(report_stream, attribute_pointer→unknown, "; and", 1, Hang);
        else
            Write(report_stream, attribute_pointer→unknown, ";", 1, Hang);
        count--;
    }
    matrix_pointer = matrix_pointer→case_next;
    attribute_pointer = attribute_pointer→next;
    }
}

static cardinal
number_of_differences(
        vector_element *vector_pointer_X,
        vector_element *vector_pointer_Y)

/* Returns the number of differences in attribute value pairs between two fact vectors. */

{
    cardinal count = 0;

    /* while there are still attribute values to compare ... */

    while (vector_pointer_X ≠ NULL) {
        if (vector_pointer_X→attribute_value ≠ vector_pointer_Y→attribute_value)
            count++;
        vector_pointer_X = vector_pointer_X→next;
        vector_pointer_Y = vector_pointer_Y→next;
    }
    return count;
}

static void
list_new_differences(
        file report_stream,
        vector_element *vector_pointer,
        vector_element *original_vector_pointer,
        attribute *attribute_pointer,
        cardinal count)

/* Lists the differences between an instantiation or hypothetical and the uninstantiated
    and unhypothesized instant case (their attribute values are pointed to by vector_pointer
    and original_vector_pointer, respectively), by writing the appropriate string (YES, NO or
    UNKNOWN) for the different attributes. *attribute_pointer is the head of the list of attrib-
    utes for this area. count is the number of differences. */

{
    /* while there are still attribute values to compare, and not all of the differences have been
        listed ... */

    while ((attribute_pointer ≠ NULL) ∧ (count ≠ 0)) {

        if (vector_pointer→attribute_value ≠ original_vector_pointer→attribute_value) {

            /* the corresponding attribute values are different, so write the relevant string with
                appropriate trailing characters */
```

```
    if (count ≡ 1)
        Write(report_stream, vector_pointer→attribute_value ≡ YES ?
                attribute_pointer→yes : vector_pointer→attribute_value ≡ NO ?
                attribute_pointer→no : attribute_pointer→unknown, ". ", 1, Hang);
    else if (count ≡ 2)
        Write(report_stream, vector_pointer→attribute_value ≡ YES ?
                attribute_pointer→yes : vector_pointer→attribute_value ≡ NO ?
                attribute_pointer→no : attribute_pointer→unknown, "; and", 1, Hang);
            else
                Write(report_stream, vector_pointer→attribute_value ≡ YES ?
                        attribute_pointer→yes : vector_pointer→attribute_value ≡ NO ?
                        attribute_pointer→no : attribute_pointer→unknown, ";", 1, Hang);
            count−−;
        }
        vector_pointer = vector_pointer→next;
        original_vector_pointer = original_vector_pointer→next;
        attribute_pointer = attribute_pointer→next;
    }
}

static void
summarize_case(
        file report_stream,
        kase *case_pointer,
        boolean written_linking_paragraph,
        boolean verbose)

/* Writes the name of the case pointed to by case_pointer with its citation in a footnote.
   Summarizes the case, if verbose is TRUE. If written_linking_paragraph is TRUE, a brief
   paragraph was just written linking the previous case with this case (the two cases are
   equidistant). */

{
    if (¬case_pointer→summarized) {

        /* the case has not been summarized yet */

        if (written_linking_paragraph)

            /* the case's citation has already been footnoted in the linking paragraph */

            fprintf(report_stream, "In␣\\frenchspacing\n"
                    "{\\it␣%s}\\nonfrenchspacing,\n",
                    case_pointer→short_name);

        else {

            fprintf(report_stream, "In␣\\frenchspacing\n"
                    "{\\it␣%s}\\nonfrenchspacing,%%\n"
                    "\\footnote{%s.}\n",
                    case_pointer→name, case_pointer→citation);
            Write_Year_and_Court(report_stream, case_pointer, 1);
            fprintf(report_stream, ",\n");
        }
```

```
if (verbose) {
    if (case_pointer→summary ≠ NULL) {
        Write(report_stream, case_pointer→summary, "\n", 1, Hang);
        case_pointer→summarized = TRUE;
    } else
        fprintf(report_stream, "\n");
} else
        Write(report_stream, "[summary].\n", Empty_String, 1, Hang);

    } else

        /* the case has already been summarized */

        fprintf(report_stream, "Details␣of␣\\frenchspacing\n"
                "{\\it␣%s\\/}␣\\nonfrenchspacing\n"
                "are␣summarized␣above.\n",
                case_pointer→short_name);
}


static void
list_similarities_and_differences(
        file report_stream,
        kase *case_pointer,
        attribute *attribute_head,
        vector_element *facts_head,
        string instant_case_type,
        boolean neighbour,
        boolean written_linking_paragraph,
        boolean unknown_list_to_follow,
        boolean verbose)

/* Cites the case pointed to by case_pointer, and lists the similarities between that case
   and the instant case (whose attribute values are pointed to by facts_head). Summar-
   izes the case, if verbose is TRUE. instant_case_type is either "instant", "instantiated" or
   "hypothetical". If neighbour is TRUE, this case is a nearest neighbour; otherwise, it is a
   nearest other. If written_linking_paragraph is TRUE, a brief paragraph was just written link-
   ing the previous case with this case (the two cases are equidistant). If unknown_list_to_follow
   is TRUE, an invocation of list_unknowns() will immediately follow this invocation of
   list_similarities_and_differences(). */

{
    cardinal count;

    if (¬written_linking_paragraph)

        /* a linking paragraph has not just been written */

        fprintf(report_stream, "%s␣", Skip);
```

```
if (case_pointer→summary ≠ NULL)
    /* this case has a summary, so write the case name (with its citation in a footnote) and
       the summary */
    summarize_case(report_stream, case_pointer, written_linking_paragraph, verbose);
if (Is_Zero_Subdistance(case_pointer→metrics.distance.known))
    /* there is no known distance between this case and the instant case */
    if (Is_Zero_Subdistance(case_pointer→metrics.distance.unknown)) {
        /* there is no unknown distance between this case and the instant case */
        fprintf(report_stream,
                "The␣%s␣case␣is␣on␣all␣fours␣with␣\\frenchspacing\n",
                instant_case_type);
        if (case_pointer→summary ≠ NULL)
            fprintf(report_stream,
                    "{\\it␣%s}\\null\\nonfrenchspacing.\n",
                    case_pointer→short_name);
        else
            /* the case has no summary, so its citation has not yet been footnoted */
            fprintf(report_stream,
                    "{\\it␣%s}\\null\\nonfrenchspacing.%%\n"
                    "\\footnote{%s.}\n",
                    case_pointer→name, case_pointer→citation);
    } else {
        /* there is some unknown distance between this case and the instant case */
        fprintf(report_stream, "The␣%s␣case␣{\\it␣may\\/}␣be␣"
                "on␣all␣fours␣with␣\\frenchspacing\n", instant_case_type);
        if (case_pointer→summary ≠ NULL)
            fprintf(report_stream, "{\\it␣%s\\/}\\nonfrenchspacing",
                    case_pointer→short_name);
        else
            /* the case has no summary, so its citation has not yet been footnoted */
            fprintf(report_stream, "{\\it␣%s\\/}\\nonfrenchspacing%%\n"
                    "\\footnote{%s.}",
                    case_pointer→name, case_pointer→citation);
        if (unknown_list_to_follow)
            /* a list of unknown differences follows immediately */
            fprintf(report_stream, "---but\n");
        else
            /* a statement that this case would have been followed (instead of the nearest
               neighbour) follows—and a list of unknown differences follows that */
            fprintf(report_stream, "␣and");
```

```
} else {
    /* there is some known and/or unknown distance between this case and the instant
       case */
    count = number_of_similarities(case_pointer→matrix_head, facts_head);
    if (count ≠ 0) {
        fprintf(report_stream, "There␣");
        if (neighbour)
            /* characterize the similarities as "extremely significant" (one), "very signific-
               ant" (two), or just "significant" (three or more) */
            switch (count) {
                case 1:
                    fprintf(report_stream,
                            "is␣one␣extremely␣significant␣similarity\n");
                    break;
                case 2:
                    fprintf(report_stream,
                            "are␣two␣very␣significant␣similarities\n");
                    break;
                default:
                    fprintf(report_stream,
                            "are␣several␣significant␣similarities\n");
                    break;
            }
        else
            switch (count) {
                case 1:
                    fprintf(report_stream,
                            "is␣one␣similarity\n");
                    break;
                case 2:
                    fprintf(report_stream,
                            "are␣two␣similarities\n");
                    break;
                default:
                    fprintf(report_stream,
                            "are␣several␣similarities\n");
                    break;
            }
        fprintf(report_stream, "between␣the␣%s␣case␣and␣\\frenchspacing\n",
                instant_case_type);
        if (case_pointer→summary ≠ NULL)
            fprintf(report_stream, "{\\it␣%s\\/}\\null\\nonfrenchspacing:\n",
                    case_pointer→short_name);
        else
            /* the case has no summary, so its citation has not yet been footnoted */
            fprintf(report_stream, "{\\it␣%s\\/}\\null\\nonfrenchspacing:%%\n"
                    "\\footnote{%s.}\n",
                    case_pointer→name, case_pointer→citation);
```

```
    /* list the similarities between this case and the instant case */

    list_similarities(report_stream, case_pointer→matrix_head, facts_head,
            attribute_head, count);
}
count = number_of_known_differences(case_pointer→matrix_head, facts_head);

if (neighbour)
    fprintf(report_stream,
            "However, the %s case is not on all fours "
            "with \\frenchspacing\n"
            "{\\it %s}\\null\\nonfrenchspacing.\n",
            instant_case_type, case_pointer→short_name);
else {

    if (count ≠ 0) {

        /* characterize the differences as "extremely significant" (one), "very significant" (two), or just "significant" (three or more) */

        fprintf(report_stream, "However, there ");
        switch (count) {
            case 1:
                fprintf(report_stream,
                        "is one extremely significant difference\n");
                break;
            case 2:
                fprintf(report_stream,
                        "are two very significant differences\n");
                break;
            default:
                fprintf(report_stream,
                        "are several significant differences\n");
                break;
        }
        fprintf(report_stream, "between the %s case and \\frenchspacing\n",
                instant_case_type);
        fprintf(report_stream,
                "{\\it %s}\\null\\nonfrenchspacing.\n",
                case_pointer→short_name);
    }
    else {
        fprintf(report_stream, "In that case\n");

        /* list the differences between this case and the instant case */

        list_known_differences(report_stream, case_pointer→matrix_head, facts_head,
                attribute_head, count);
    }
}
```

```
static boolean
has_less_known_distance(
        kase *case_pointer_X,
        kase *case_pointer_Y)

/* Returns TRUE, iff the case pointed to by case_pointer_X has less known distance than does
   that pointed to by case_pointer_Y. */

{
    return ((case_pointer_X ≠ NULL) ∧ (case_pointer_Y ≠ NULL) ∧
            ((case_pointer_X→metrics.distance.known.infinite <
                    case_pointer_Y→metrics.distance.known.infinite) ∨
                ((case_pointer_X→metrics.distance.known.infinite ≡
                        case_pointer_Y→metrics.distance.known.infinite) ∧
                    Is_Less(case_pointer_X→metrics.distance.known.finite,
                        case_pointer_Y→metrics.distance.known.finite,
                        Distance_Precision))));
}

static void
state_confidence(
        file report_stream,
        string short_name)

/* States its confidence that the case called short_name should still be followed. */

{
    fprintf(report_stream, "Nevertheless, I believe that \\frenchspacing\n"
            "{\\it %s\\/} \\nonfrenchspacing\n"
            "should be followed.\n\n", short_name);
}

static boolean
write_number_as_word(
        file report_stream,
        cardinal number)

/* Writes number: as a word, if number ≤ 10; as a number, otherwise. Returns TRUE, iff
   number ≠ 1: i.e. if the noun to follow should be plural. */

{
    switch (number) {
        case 1:
            fprintf(report_stream, "one");
            break;
        case 2:
            fprintf(report_stream, "two");
            break;
```

```
    case 3:
        fprintf(report_stream, "three");
        break;
    case 4:
        fprintf(report_stream, "four");
        break;
    case 5:
        fprintf(report_stream, "five");
        break;
    case 6:
        fprintf(report_stream, "six");
        break;
    case 7:
        fprintf(report_stream, "seven");
        break;
    case 8:
        fprintf(report_stream, "eight");
        break;
    case 9:
        fprintf(report_stream, "nine");
        break;
    case 10:
        fprintf(report_stream, "ten");
        break;
    default:
        fprintf(report_stream, "%u", number);
        break;
  }
  return (number ≠ 1);
}

static void
state_intransigence(
        file report_stream,
        kase *nearest_neighbour,
        kase *nearest_other)

/* Restates its opinion that the case pointed to by nearest_neighbour should be followed, and
   compares the relative importance of the courts that decided that case and the case pointed
   to by nearest_other. */

{
  if ((nearest_neighbour→court_string ≡ NULL) ∨ (nearest_other→court_string ≡ NULL))

    /* the nearest neighbour or the nearest other has no court, so make no comment about
       each case's relative importance */

    fprintf(report_stream, "\nConsequently, ␣");
```

else if (*nearest_neighbour*→*court_rank* < *nearest_other*→*court_rank*)

/∗ the nearest neighbour was decided by a more important court than was the nearest other ∗/

fprintf(*report_stream*, "Note%s␣that␣\\frenchspacing\n"
"{\\it␣%s\\/}␣\\nonfrenchspacing\n"
"is␣only␣a␣decision␣of\n"
"%s\n"
"and␣not␣as␣good␣authority␣as␣a␣case␣decided␣by\n"
"%s%%\n"
"---like␣\\frenchspacing\n"
"{\\it␣%s}\\null\\nonfrenchspacing.\n\n"
"Consequently,␣",
Is_Zero_Subdistance(*nearest_other*→*metrics.distance.known*) ∧
Is_Zero_Subdistance(*nearest_other*→*metrics.distance.unknown*) ?
",␣however,": "␣also",
nearest_other→*short_name*, *nearest_other*→*court_string*,
nearest_neighbour→*court_string*, *nearest_neighbour*→*short_name*);

else if (*nearest_neighbour*→*court_rank* > *nearest_other*→*court_rank*)

/∗ the nearest other was decided by a more important court than was the nearest neighbour ∗/

fprintf(*report_stream*, "\nDespite␣the␣fact␣that␣\\frenchspacing\n"
"{\\it␣%s\\/}␣\\nonfrenchspacing\n"
"is␣a␣decision␣of\n"
"%s\n"
"(and␣better␣authority␣than␣a␣case␣decided␣by\n"
"%s%%\n"
"---like␣\\frenchspacing\n"
"{\\it␣%s\\/}\\nonfrenchspacing),\n",
nearest_other→*short_name*, *nearest_other*→*court_string*,
nearest_neighbour→*court_string*, *nearest_neighbour*→*short_name*);

else if (*nearest_neighbour*→*court_string* ≡ *nearest_other*→*court_string*)

/∗ the nearest neighbour and the nearest other were decided by the same court ∗/

fprintf(*report_stream*, "\nDespite␣the␣fact␣that␣\\frenchspacing\n"
"{\\it␣%s\\/}␣\\nonfrenchspacing\n"
"and␣\\frenchspacing\n"
"{\\it␣%s\\/}␣\\nonfrenchspacing\n"
"are␣both␣decisions␣of\n"
"%s,\n", *nearest_other*→*short_name*,
nearest_neighbour→*short_name*, *nearest_other*→*court_string*);

else

/* the nearest neighbour and the nearest other were decided by different courts of the same rank */

fprintf(*report_stream*, "\nDespite␣the␣fact␣that␣\\frenchspacing\n"
"{\\it␣%s\\/}␣\\nonfrenchspacing\n"
"is␣a␣decision␣of\n"
"%s\n"
"(and␣as␣good␣authority␣as␣a␣case␣decided␣by\n"
"%s%%\n"
"---like␣\\frenchspacing\n"
"{\\it␣%s\\/}\\nonfrenchspacing),\n",
nearest_other→*short_name*, *nearest_other*→*court_string*,
nearest_neighbour→*court_string*, *nearest_neighbour*→*short_name*);

fprintf(*report_stream*, "there␣is␣nothing␣in␣\\frenchspacing\n"
"{\\it␣%s\\/}␣\\nonfrenchspacing\n"
"to␣warrant␣any␣change␣in␣my␣conclusion.\n\n",
nearest_other→*short_name*);
}

static *boolean*
write_linking_paragraph(
file report_stream,
kase ∗*previous_case_pointer*,
kase ∗*next_case_pointer*)

/* Writes a brief paragraph linking the previous case with the next case (the two cases are equidistant), if there is a next case. Puts the two cases into context (i.e. explains which is more important and why). Returns *TRUE*, if a paragraph is written, which means that the next paragraph should not include year and court information—information included in this linking paragraph. */

{
if ((*next_case_pointer* ≠ *NULL*) ∧
(*previous_case_pointer*→*court_string* ≠ *NULL*) ∧
(*next_case_pointer*→*court_string* ≠ *NULL*)) {

/* the previous case and the next case are equidistant and both have a court string, so write a linking paragraph */

fprintf(*report_stream*, "%s␣In␣%u,\n", *Skip*, *next_case_pointer*→*year*);

if (*previous_case_pointer*→*court_string* ≡ *next_case_pointer*→*court_string*) {

/* the previous case and the next case were decided by the same court */

if (*previous_case_pointer*→*year* ≡ *next_case_pointer*→*year*)

/* the previous case and the next case were decided in the same year */

fprintf(*report_stream*, "the␣same␣year␣in␣which␣\\frenchspacing\n"
"{\\it␣%s\\/}␣\\nonfrenchspacing\n"
"was␣decided,\n",
previous_case_pointer→*short_name*);

```
    fprintf(report_stream, "%s\n"
            "also␣decided␣\\frenchspacing\n"
            "{\\it␣%s}\\null\\nonfrenchspacing.%%\n"
            "\\footnote{%s.}\n",
            next_case_pointer→court_string, next_case_pointer→name,
            next_case_pointer→citation);

    if (previous_case_pointer→year ≠ next_case_pointer→year) {
        /* the previous case and the next case were decided in a different year; the
           previous case must be more recent than the next case (otherwise the next
           case would be earlier in the list than the previous case) */

        fprintf(report_stream, "(Note,␣however,␣that␣\\frenchspacing\n"
                "{\\it␣%s\\/}␣\\nonfrenchspacing\n"
                "is␣", previous_case_pointer→short_name);

        if (write_number_as_word(report_stream,
                    previous_case_pointer→year − next_case_pointer→year))
            fprintf(report_stream, "␣years");
        else
            fprintf(report_stream, "␣year");

        fprintf(report_stream, "␣more␣recent␣than␣\\frenchspacing\n"
                "{\\it␣%s}\\null\\nonfrenchspacing.)\n",
                next_case_pointer→short_name);
    }
    } else if (previous_case_pointer→court_rank ≡ next_case_pointer→court_rank) {
        /* the previous case and the next case were decided by different courts of the same
           rank */

        if (previous_case_pointer→year ≡ next_case_pointer→year)

            /* the previous case and the next case were decided in the same year */

            fprintf(report_stream, "the␣same␣year␣in␣which␣\\frenchspacing\n"
                    "{\\it␣%s\\/}␣\\nonfrenchspacing\n"
                    "was␣decided␣by\n"
                    "%s,␣",
                    previous_case_pointer→short_name,
                    previous_case_pointer→court_string);

        fprintf(report_stream, "\\frenchspacing\n"
                "{\\it␣%s\\/}\\nonfrenchspacing%%\n"
                "\\footnote{%s.}\n"
                "was␣decided␣by\n"
                "%s.\n"
                "(A␣case␣decided␣by\n"
                "%s\n"
                "is␣as␣good␣authority␣as␣a␣case␣decided␣by\n"
                "%s", next_case_pointer→name, next_case_pointer→citation,
                next_case_pointer→court_string, next_case_pointer→court_string,
                previous_case_pointer→court_string);
```

```
if (previous_case_pointer→year ≡ next_case_pointer→year)
    /* the previous case and the next case were decided in the same year */
    fprintf(report_stream, ".)\n");
else {
    /* the previous case and the next case were decided in a different year; the
       previous case must be more recent than the next case (otherwise the next
       case would be earlier in the list than the previous case) */
    fprintf(report_stream, "%%\n"
            "---like␣\\frenchspacing\n"
            "{\\it␣%s\\/}\\nonfrenchspacing;\n"
            "note,␣however,␣that␣\\frenchspacing\n"
            "{\\it␣%s\\/}␣\\nonfrenchspacing\n"
            "is␣",
            previous_case_pointer→short_name,
            previous_case_pointer→short_name);
    if (write_number_as_word(report_stream,
                    previous_case_pointer→year − next_case_pointer→year))
        fprintf(report_stream, "␣years");
    else
        fprintf(report_stream, "␣year");
    fprintf(report_stream, "␣more␣recent␣than␣\\frenchspacing\n"
            "{\\it␣%s}\\null\\nonfrenchspacing.)\n",
            next_case_pointer→short_name);
    }
} else {
    /* the previous case and the next case were decided by different courts of different
       ranks; the previous case must be more important than the next case, otherwise
       the next case would be earlier in the list than the previous case) */
    if (previous_case_pointer→year ≡ next_case_pointer→year)
        /* the previous case and the next case were decided in the same year */
        fprintf(report_stream, "the␣same␣year␣in␣which␣\\frenchspacing\n"
                "{\\it␣%s\\/}␣\\nonfrenchspacing\n"
                "was␣decided␣by\n"
                "%s,␣",
                previous_case_pointer→short_name,
                previous_case_pointer→court_string);
    fprintf(report_stream, "\\frenchspacing\n"
            "{\\it␣%s\\/}\\nonfrenchspacing%%\n"
            "\\footnote{%s.}\n"
            "was␣decided␣by\n"
            "%s.\n"
            "(A␣case␣decided␣by\n"
            "%s\n"
            "is␣not␣as␣good␣authority␣as␣a␣case␣decided␣by\n"
            "%s", next_case_pointer→name, next_case_pointer→citation,
            next_case_pointer→court_string, next_case_pointer→court_string,
            previous_case_pointer→court_string);
```

```
if (previous_case_pointer→year ≡ next_case_pointer→year)
    /* the previous case and the next case were decided in the same year */
    fprintf(report_stream, ".)\n");
else
    fprintf(report_stream, "%%\n"
            "---like␣\\frenchspacing\n"
            "{\\it␣%s\\/}\\nonfrenchspacing;\n",
            previous_case_pointer→short_name);
if (previous_case_pointer→year > next_case_pointer→year) {
    /* the previous case is more recent than the next case */
    fprintf(report_stream, "furthermore␣\\frenchspacing\n"
            "{\\it␣%s\\/}␣\\nonfrenchspacing\n"
            "is␣", next_case_pointer→short_name);
    if (write_number_as_word(report_stream,
                    previous_case_pointer→year − next_case_pointer→year))
        fprintf(report_stream, "␣years");
    else
        fprintf(report_stream, "␣year");
    fprintf(report_stream, "␣older␣than␣\\frenchspacing\n"
            "{\\it␣%s}\\null\\nonfrenchspacing.)\n",
            previous_case_pointer→short_name);
} else if (previous_case_pointer→year < next_case_pointer→year) {
    /* the next case is more recent than the previous case */
    fprintf(report_stream, "though␣\\frenchspacing\n"
            "{\\it␣%s\\/}␣\\nonfrenchspacing\n"
            "is␣", next_case_pointer→short_name);
    if (write_number_as_word(report_stream,
                    next_case_pointer→year − previous_case_pointer→year))
        fprintf(report_stream, "␣years");
    else
        fprintf(report_stream, "␣year");
    fprintf(report_stream, "␣more␣recent␣than␣\\frenchspacing\n"
            "{\\it␣%s}\\null\\nonfrenchspacing.)\n",
            previous_case_pointer→short_name);
}
fprintf(report_stream, "\n");
return TRUE;
} else
    /* no linking paragraph has been written */
    return FALSE;
}
```

```
static void
handle_near_unknown(
        file report_stream,
        result *result_pointer,
        result *nearest_result_pointer,
        kase *nearest_neighbour,
        boolean nearest_neighbour_is_known,
        area *area_pointer,
        vector_element *facts_head,
        string instant_case_type,
        boolean neighbour,
        boolean verbose)

/* Argues that the nearest unknown case of the result pointed to by result_pointer would have
   been followed but for its unknown distance. */

{
    kase *case_pointer;
    boolean written_linking_paragraph = FALSE;

    if (result_pointer ≡ nearest_result_pointer) {

        fprintf(report_stream, "%s\\frenchspacing\n", Skip);

        list_equidistant_cases(report_stream, NULL, result_pointer→nearest_unknown_case,
                FALSE, FALSE, TRUE);

        fprintf(report_stream, "␣\\nonfrenchspacing\n"
                "%s␣in␣which␣%s.\n\n",
                result_pointer→nearest_unknown_case→equidistant_unknown_next ≡
                NULL ? "is␣another␣case" : "are␣other␣cases", result_pointer→string);

    }

    /* for each nearest unknown case ... */

    for (case_pointer = result_pointer→nearest_unknown_case; case_pointer ≠ NULL;
            case_pointer = case_pointer→equidistant_unknown_next) {

        list_similarities_and_differences(report_stream, case_pointer,
                area_pointer→attribute_head, facts_head, instant_case_type, neighbour,
                written_linking_paragraph, FALSE, verbose);

        fprintf(report_stream, "\n"
                "I␣would␣have␣suggested␣that␣\\frenchspacing\n"
                "{\\it␣%s\\/}␣\\nonfrenchspacing\n"
                "be␣followed␣(instead␣of␣\\frenchspacing\n",
                case_pointer→short_name);

    if (nearest_neighbour_is_known)
        list_equidistant_cases(report_stream, nearest_neighbour, NULL,
                FALSE, TRUE, TRUE);

    else
        list_equidistant_cases(report_stream, NULL, nearest_neighbour,
                FALSE, TRUE, TRUE);
```

```
    fprintf(report_stream, "\\nonfrenchspacing)\n"
            "except␣that\n");
    list_unknowns(report_stream, case_pointer→matrix_head,
            area_pointer→attribute_head, number_of_unknowns(case_pointer→matrix_head));
    if (¬neighbour)
        state_intransigence(report_stream, nearest_neighbour, case_pointer);
    written_linking_paragraph = write_linking_paragraph(report_stream, case_pointer,
                case_pointer→equidistant_unknown_next);
    }
}

static void
handle_nearest_known(
        file report_stream,
        result *result_pointer,
        result *nearest_result_pointer,
        kase *nearest_neighbour,
        area *area_pointer,
        vector_element *facts_head,
        string instant_case_type,
        boolean verbose)

/* Argues that the result should be same as that of the nearest known neighbour. Uses the
   nearest unknown neighbour too, if (but for its unknown distance) it would be the nearest
   neighbour. */

{
    kase *case_pointer;
    boolean written_linking_paragraph = FALSE;
    if (has_less_known_distance(result_pointer→nearest_unknown_case,
                result_pointer→nearest_known_case))
        /* if not for its unknown distance, the nearest unknown neighbour would be the nearest
           neighbour, so base the argument on the nearest known and the nearest unknown
           neighbours */
        state_opinion(report_stream, result_pointer,
                result_pointer→nearest_known_case,
                result_pointer→nearest_unknown_case, FALSE);
    else
        /* base the argument only on the nearest known neighbours */
        state_opinion(report_stream, result_pointer,
                result_pointer→nearest_known_case, NULL, FALSE);
    /* for each nearest known neighbour ... */
    for (case_pointer = result_pointer→nearest_known_case; case_pointer ≠ NULL;
            case_pointer = case_pointer→equidistant_known_next) {
        list_similarities_and_differences(report_stream, case_pointer,
                area_pointer→attribute_head, facts_head, instant_case_type, TRUE,
                written_linking_paragraph, FALSE, verbose);
        fprintf(report_stream, "\n");
```

```
    if (¬Is_Zero_Distance(case_pointer→metrics.distance))
        /* the instant case is not on all fours with the case */
        state_confidence(report_stream, case_pointer→short_name);
    written_linking_paragraph = write_linking_paragraph(report_stream, case_pointer,
            case_pointer→equidistant_known_next);
}

    if (has_less_known_distance(result_pointer→nearest_unknown_case,
                result_pointer→nearest_known_case)) {
        /* if not for its unknown distance, the nearest unknown neighbour would be the nearest
            neighbour */
        handle_near_unknown(report_stream, result_pointer, nearest_result_pointer,
                nearest_neighbour, TRUE, area_pointer,
                facts_head, instant_case_type, TRUE, verbose);
        fprintf(report_stream, "\n");
    }
}

static void
handle_nearest_unknown(
        file report_stream,
        result *result_pointer,
        area *area_pointer,
        vector_element *facts_head,
        string instant_case_type,
        boolean verbose)

/* Argues that the result should be same as that in the nearest unknown neighbour. Uses the
    nearest known neighbour too. */

{
    kase *case_pointer;
    boolean written_linking_paragraph = FALSE;
    cardinal count;
    /* base the argument on the nearest unknown and the nearest known neighbours */
    state_opinion(report_stream, result_pointer,
            result_pointer→nearest_known_case,
            result_pointer→nearest_unknown_case, TRUE);
    /* for every nearest unknown neighbour ... */
    for (case_pointer = result_pointer→nearest_unknown_case; case_pointer ≠ NULL;
            case_pointer = case_pointer→equidistant_unknown_next) {
        list_similarities_and_differences(report_stream, case_pointer,
                area_pointer→attribute_head, facts_head, instant_case_type, TRUE,
                written_linking_paragraph, TRUE, verbose);
        count = number_of_unknowns(case_pointer→matrix_head);
```

```
    if (count ≠ 0) {

        fprintf(report_stream, "Furthermore, ␣\n");

        list_unknowns(report_stream, case_pointer→matrix_head,
                area_pointer→attribute_head, count);
    }
    fprintf(report_stream, "\n");
        state_confidence(report_stream, case_pointer→short_name);

        written_linking_paragraph = write_linking_paragraph(report_stream, case_pointer,
                case_pointer→equidistant_unknown_next);
    }

    /* for every nearest known neighbour ... */

    written_linking_paragraph = FALSE;

    for (case_pointer = result_pointer→nearest_known_case; case_pointer ≠ NULL;
            case_pointer = case_pointer→equidistant_known_next) {

        list_similarities_and_differences(report_stream, case_pointer,
                area_pointer→attribute_head, facts_head, instant_case_type, TRUE,
                written_linking_paragraph, FALSE, verbose);

        fprintf(report_stream, "\n");

        if (¬Is_Zero_Distance(case_pointer→metrics.distance))
            state_confidence(report_stream, case_pointer→short_name);

        written_linking_paragraph = write_linking_paragraph(report_stream, case_pointer,
                case_pointer→equidistant_known_next);
    }
}


static void
handle_nearest_others(
        file report_stream,
        result *result_pointer,
        result *nearest_result_pointer,
        kase *nearest_other,
        boolean nearest_other_is_known,
        area *area_pointer,
        vector_element *facts_head,
        string instant_case_type,
        boolean verbose)

/* Make a counter argument that the result should be same as that of the nearest known other.
   Uses this result's nearest unknown other too, if (but for its unknown distance) it would be
   the nearest neighbour. */
```

```
{
    kase *case_pointer;
    boolean written_linking_paragraph = FALSE;
    cardinal count;

    if ((result_pointer→nearest_known_compared_with_unknown ≡ FURTHER) ∨
            has_less_known_distance(result_pointer→nearest_unknown_case,
                    nearest_other)) {
        /* the nearest unknown other is the nearest other or if not for its unknown distance,
           the nearest unknown other would be the nearest neighbour */

        state_counter_opinion(report_stream, result_pointer,
                result_pointer→nearest_known_case,
                result_pointer→nearest_unknown_case, TRUE);

        if (has_less_known_distance(result_pointer→nearest_unknown_case,
                    nearest_other)) {

            /* if not for its unknown distance, the nearest unknown other would be the nearest
               neighbour */

            handle_near_unknown(report_stream, result_pointer, nearest_result_pointer,
                    nearest_other, nearest_other_is_known, area_pointer,
                    facts_head, instant_case_type, FALSE, verbose);

        } else

            /* for every nearest unknown other ... */

            for (case_pointer = result_pointer→nearest_unknown_case;
                    case_pointer ≠ NULL;
                    case_pointer = case_pointer→equidistant_unknown_next) {

                list_similarities_and_differences(report_stream, case_pointer,
                        area_pointer→attribute_head, facts_head, instant_case_type, FALSE,
                        written_linking_paragraph, FALSE, verbose);

                count = number_of_unknowns(case_pointer→matrix_head);

                if (count ≠ 0) {

                    fprintf(report_stream, "Furthermore, \n");

                    list_unknowns(report_stream, case_pointer→matrix_head,
                            area_pointer→attribute_head, count);
                }
                state_intransigence(report_stream, nearest_other, case_pointer);

                written_linking_paragraph = write_linking_paragraph(report_stream, case_pointer,
                        case_pointer→equidistant_unknown_next);
            }

    } else

        /* the nearest known other is the nearest other, so base the counter-opinion only on
           the nearest known other */

        state_counter_opinion(report_stream, result_pointer,
                result_pointer→nearest_known_case, NULL, FALSE);
```

```
    written_linking_paragraph = FALSE;
    for (case_pointer = result_pointer→nearest_known_case; case_pointer ≠ NULL;
            case_pointer = case_pointer→equidistant_known_next) {
        /* for every nearest known other ... */
        list_similarities_and_differences(report_stream, case_pointer,
                area_pointer→attribute_head, facts_head, instant_case_type, FALSE,
                written_linking_paragraph, FALSE, verbose);
        state_intransigence(report_stream, nearest_other, case_pointer);
        written_linking_paragraph = write_linking_paragraph(report_stream, case_pointer,
                case_pointer→equidistant_known_next);
    }
}

static void
initialize_nearest_metrics(
        cardinal *minimum_known_differences,
        floating_point *minimum_association_coefficient,
        floating_point *minimum_weighted_association_coefficient,
        floating_point *max_correlation_coefficient,
        floating_point *max_weighted_correlation_coefficient,
        cardinal number_of_attributes)
/* Initializes the minimum and maximum metric variables. */
{
    *minimum_known_differences = number_of_attributes;
    *minimum_association_coefficient = 1.0;
    *minimum_weighted_association_coefficient = 1.0;
    *max_correlation_coefficient = -1.0;
    *max_weighted_correlation_coefficient = -1.0;
}

static void
find_nearest_metrics(
        metrics_type metrics,
        cardinal *minimum_known_differences,
        floating_point *minimum_association_coefficient,
        floating_point *minimum_weighted_association_coefficient,
        floating_point *max_correlation_coefficient,
        floating_point *max_weighted_correlation_coefficient,
        boolean weighted_association_coefficient)
/* Checks metrics against the various minimum and maximum values found so far, and changes
    each minimum/maximum if the relevant metric is less/more. */
{
    if (metrics.number_of_known_differences < *minimum_known_differences)
        *minimum_known_differences = metrics.number_of_known_differences;
    if (metrics.number_of_known_pairs ≡ 0) {
        *minimum_known_differences = 0;
        *minimum_association_coefficient = 0.0;
        *minimum_weighted_association_coefficient = 0.0;
```

```
} else {
    if (Is_Less((floating_point) metrics.number_of_known_differences /
                    metrics.number_of_known_pairs,
                    *minimum_association_coefficient, Precision))
        *minimum_association_coefficient =
                (floating_point) metrics.number_of_known_differences /
                metrics.number_of_known_pairs;
    if (weighted_association_coefficient)
        if (Is_Less(metrics.weighted_association_coefficient,
                        *minimum_weighted_association_coefficient, Precision))
            *minimum_weighted_association_coefficient =
                    metrics.weighted_association_coefficient;
    if (¬metrics.correlation_coefficient.meaningless) {
        if (Is_Less(*max_correlation_coefficient,
                        metrics.correlation_coefficient.unweighted, Precision))
            *max_correlation_coefficient =
                    metrics.correlation_coefficient.unweighted;
        if (Is_Less(*max_weighted_correlation_coefficient,
                        metrics.correlation_coefficient.weighted, Precision))
            *max_weighted_correlation_coefficient =
                    metrics.correlation_coefficient.weighted;
        }
    }
}

static boolean
matches_nearest_neighbour(
        kase *case_pointer,
        kase **nearest_known_neighbour_pointer,
        kase **nearest_unknown_neighbour_pointer)

/* Returns TRUE (and adjusts the appropriate pointer so as to point to the next equidistant
   case), if case_pointer points to a nearest neighbour (known or unknown). */

{
    if (case_pointer ≡ *nearest_known_neighbour_pointer) {
        *nearest_known_neighbour_pointer =
        (*nearest_known_neighbour_pointer)→equidistant_known_next;
        return TRUE;
    } else if (case_pointer ≡ *nearest_unknown_neighbour_pointer) {
        *nearest_unknown_neighbour_pointer =
                (*nearest_unknown_neighbour_pointer)→equidistant_unknown_next;
        return TRUE;
    } else
        return FALSE;
}
```

```
static void
log_case_number_and_name(
        file log_stream,
        cardinal case_number,
        string case_name)
{
    fprintf(log_stream, "C%u%s␣%s", case_number, case_number < 10 ? "␣" : "", case_name);
}
```

```
static void
log_case(
        file log_stream,
        kase *case_pointer,
        result *result_pointer,
        boolean additional,
        cardinal level,
        string *subheading)
```

/* Writes, to the log file, details of the case pointed to by *case_pointer*. Writes a character before the case name: "*", if *additional* is *TRUE* and *result_pointer* is not *NULL* (i.e. the case is suggested by an extra metric, but is not a neighbour, and has a different result to the nearest result); "+", if *additional* is *TRUE* (i.e. the case is suggested by an extra metric, but is not a neighbour); "–", otherwise (i.e. the case is not suggested by the metric, but is a nearest neighbour). Writes the result in parentheses after the case name, if *result_pointer* is not *NULL*. */

```
{
    if (*subheading ≠ NULL) {
        Indent(log_stream, level + 1);
        fprintf(log_stream, "%s:\n", *subheading);
        *subheading = NULL;
    }
    Indent(log_stream, level + 1);
    fprintf(log_stream, "␣␣%s␣", additional ? result_pointer ≠ NULL ? "*" : "+" : "-");
    log_case_number_and_name(log_stream, case_pointer→number, case_pointer→short_name);
    if (result_pointer ≠ NULL)
        fprintf(log_stream, "␣(%s)", result_pointer→identifier);
    fprintf(log_stream, "\n");
}
```

```
static void
log_case_if_necessary(
        file log_stream,
        result *result_pointer,
        result *nearest_result_pointer,
        kase *case_pointer,
        kase **nearest_known_case_pointer,
        kase **nearest_unknown_case_pointer,
        cardinal level,
        boolean neighbour,
        string *heading,
        string *subheading,
        boolean *different_result)
```

```
/* Writes, to the log file, details of those cases about which the known/unknown distance
   and the extra similarity measures disagree. Sets different_result to TRUE, if an alternative
   metric suggests, as a nearest neighbour, a case which is not a nearest neighbour and which
   has a result different from the nearest result. */

{
  if (neighbour) {

    /* the alternative metric suggests this as a nearest neighbour */

    if (¬matches_nearest_neighbour(case_pointer,
                nearest_known_case_pointer, nearest_unknown_case_pointer)) {

      /* it isn't a nearest neighbour, so log it as an additional case */

      if (*heading ≠ NULL) {
        Indent(log_stream, level);
        fprintf(log_stream, "%s:\n\n", *heading);
        *heading = NULL;
      }
      if (result_pointer ≠ nearest_result_pointer) {

        /* the result of this case is not the nearest result, so include the result in the
           log */

        log_case(log_stream, case_pointer, result_pointer, TRUE, level, subheading);
        if (different_result ≠ NULL)
          *different_result = TRUE;

      } else

        /* the result of this case is the nearest result, so don't include it in the log */

        log_case(log_stream, case_pointer, NULL, TRUE, level, subheading);
    }

  } else if (matches_nearest_neighbour(case_pointer,
                nearest_known_case_pointer, nearest_unknown_case_pointer)) {

    /* the alternative metric does not suggest this case as a nearest neighbour, but it is a
       nearest neighbour, so log it as a missing case */

    if (*heading ≠ NULL) {
      Indent(log_stream, level);
      fprintf(log_stream, "%s:\n\n", *heading);
      *heading = NULL;
    }

    /* the result of this case is the nearest result, so don't include it in the log */

    log_case(log_stream, case_pointer, NULL, FALSE, level, subheading);
  }
}
```

```
static void
implement_safeguards(
        file log_stream,
        area *area_pointer,
        string instant_case_type,
        cardinal level)

/* Implements the safeguards based on the extra similarity measures, and issues a warning in
    each of the following circumstances: the weighted association coefficients suggest that a case,
    with a different result than that of the nearest neighbour, ought to be the nearest neighbour;
    the weighted correlation coefficients suggest that a case, with a different result than that
    of the nearest neighbour, ought to be the nearest neighbour; an ideal point suggesting a
    different result is at least as near to the instant case as is the nearest neighbour; a centroid
    suggesting a different result is at least as near to the instant case as is the nearest neighbour;
    or the specified directions suggest a different result or results. */

{
    result *result_pointer,
      *equidistant_pointer;
    kase *case_pointer,
      *nearest_known_case_pointer,
      *nearest_unknown_case_pointer;
    cardinal minimum_known_differences;
    floating_point minimum_association_coefficient,
        minimum_weighted_association_coefficient,
        max_correlation_coefficient,
        max_weighted_correlation_coefficient;
    string heading = "Safeguards",
        subheading = NULL;
    char message[Max_Error_Message_Length];
    boolean weighted_different_result = FALSE,
        ideal_point_different_result = FALSE,
        centroid_different_result = FALSE,
        specified_direction_different_result = FALSE;

    initialize_nearest_metrics(&minimum_known_differences,
            &minimum_association_coefficient, &minimum_weighted_association_coefficient,
            &max_correlation_coefficient, &max_weighted_correlation_coefficient,
            area_pointer→number_of_attributes);

    for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
            result_pointer = result_pointer→next)
        for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
                case_pointer = case_pointer→next)
            find_nearest_metrics(case_pointer→metrics, &minimum_known_differences,
                    &minimum_association_coefficient,
                    &minimum_weighted_association_coefficient, &max_correlation_coefficient,
                    &max_weighted_correlation_coefficient, ¬area_pointer→infinite_weight);
```

```
    subheading = "Distance measures";

    nearest_known_case_pointer =
            area_pointer→nearest_result→nearest_known_case;
    nearest_unknown_case_pointer =
            area_pointer→nearest_result→nearest_unknown_case;

    for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
            result_pointer = result_pointer→next)
        for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
                case_pointer = case_pointer→next)
            log_case_if_necessary(log_stream, result_pointer, area_pointer→nearest_result,
                    case_pointer, &nearest_known_case_pointer,
                    &nearest_unknown_case_pointer, level,
                    case_pointer→metrics.number_of_known_differences ≡
                    minimum_known_differences,
                    &heading, &subheading, NULL);
    if (subheading ≡ NULL)
        fprintf(log_stream, "\n");

    subheading = "Association coefficients";

    nearest_known_case_pointer =
            area_pointer→nearest_result→nearest_known_case;
    nearest_unknown_case_pointer =
            area_pointer→nearest_result→nearest_unknown_case;

    for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
            result_pointer = result_pointer→next)
        for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
                case_pointer = case_pointer→next)
            log_case_if_necessary(log_stream, result_pointer, area_pointer→nearest_result,
                    case_pointer, &nearest_known_case_pointer,
                    &nearest_unknown_case_pointer, level,
                    Is_Equal((floating_point) case_pointer→
                            metrics.number_of_known_differences /
                            case_pointer→metrics.number_of_known_pairs,
                            minimum_association_coefficient, Precision),
                    &heading, &subheading, NULL);
    if (subheading ≡ NULL)
        fprintf(log_stream, "\n");
```

```
    if (¬area_pointer→infinite_weight) {
        /* none of the weights is infinite, so the values obtained for the weighted association
           coefficients are meaningful */
        subheading = "Weighted␣association␣coefficients";
        nearest_known_case_pointer =
                area_pointer→nearest_result→nearest_known_case;
        nearest_unknown_case_pointer =
                area_pointer→nearest_result→nearest_unknown_case;
        for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
                result_pointer = result_pointer→next)
            for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
                    case_pointer = case_pointer→next)
                log_case_if_necessary(log_stream, result_pointer, area_pointer→nearest_result,
                        case_pointer, &nearest_known_case_pointer,
                        &nearest_unknown_case_pointer, level,
                        Is_Equal(case_pointer→metrics.weighted_association_coefficient,
                                minimum_weighted_association_coefficient, Precision),
                        &heading, &subheading, &weighted_different_result);
        if (subheading ≡ NULL)
            fprintf(log_stream, "\n");
    }
    subheading = "Correlation␣coefficients";
    nearest_known_case_pointer =
            area_pointer→nearest_result→nearest_known_case;
    nearest_unknown_case_pointer =
            area_pointer→nearest_result→nearest_unknown_case;
    for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
            result_pointer = result_pointer→next)
        for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
                case_pointer = case_pointer→next)
            if (¬case_pointer→metrics.correlation_coefficient.meaningless)
                log_case_if_necessary(log_stream, result_pointer, area_pointer→nearest_result,
                        case_pointer, &nearest_known_case_pointer,
                        &nearest_unknown_case_pointer, level,
                        Is_Equal(max_correlation_coefficient,
                                case_pointer→metrics.correlation_coefficient.unweighted,
                                Precision),
                        &heading, &subheading, NULL);
    if (subheading ≡ NULL)
        fprintf(log_stream, "\n");
```

```
subheading = "Weighted␣correlation␣coefficients";

nearest_known_case_pointer =
        area_pointer→nearest_result→nearest_known_case;
nearest_unknown_case_pointer =
        area_pointer→nearest_result→nearest_unknown_case;

for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
        result_pointer = result_pointer→next)
    for (case_pointer = result_pointer→case_head; case_pointer ≠ NULL;
            case_pointer = case_pointer→next)
        if (¬case_pointer→metrics.correlation_coefficient.meaningless)
            log_case_if_necessary(log_stream, result_pointer, area_pointer→nearest_result,
                    case_pointer, &nearest_known_case_pointer,
                    &nearest_unknown_case_pointer, level,
                    Is_Equal(max_weighted_correlation_coefficient,
                            case_pointer→metrics.correlation_coefficient.weighted, Precision),
                    &heading, &subheading, &weighted_different_result);
if (subheading ≡ NULL)
    fprintf(log_stream, "\n");

subheading = "Ideal␣points";

for (equidistant_pointer = area_pointer→nearest_ideal_point; equidistant_pointer ≠ NULL;
        equidistant_pointer = equidistant_pointer→equidistant_ideal_point_next)
    if (equidistant_pointer ≠ area_pointer→nearest_result) {
    if (heading ≠ NULL) {
        Indent(log_stream, level);
        fprintf(log_stream, "%s:\n\n", heading);
        heading = NULL;
    }
    if (subheading ≠ NULL) {
        Indent(log_stream, level + 1);
        fprintf(log_stream, "%s:\n", subheading);
        subheading = NULL;
    }
    Indent(log_stream, level + 2);
    fprintf(log_stream, "%s\n", equidistant_pointer→identifier);
    if (area_pointer→nearest_result→nearest_known_case ≠ NULL)
        if (Relative_Distance(equidistant_pointer→ideal_point_metrics.distance,
                area_pointer→nearest_result→nearest_known_case→
                metrics.distance) ≠ FURTHER)
            ideal_point_different_result = TRUE;
        if (area_pointer→nearest_result→nearest_unknown_case ≠ NULL)
            if (Relative_Distance(equidistant_pointer→ideal_point_metrics.distance,
                    area_pointer→nearest_result→nearest_unknown_case→
                    metrics.distance) ≠ FURTHER)
                ideal_point_different_result = TRUE;
    }
if (subheading ≡ NULL)
    fprintf(log_stream, "\n");
```

```
subheading = "Centroids";
for (equidistant_pointer = area_pointer→nearest_centroid; equidistant_pointer ≠ NULL;
        equidistant_pointer = equidistant_pointer→equidistant_centroid_next)
    if (equidistant_pointer ≠ area_pointer→nearest_result) {
        if (heading ≠ NULL) {
            Indent(log_stream, level);
            fprintf(log_stream, "%s:\n\n", heading);
            heading = NULL;
        }
        if (subheading ≠ NULL) {
            Indent(log_stream, level + 1);
            fprintf(log_stream, "%s:\n", subheading);
            subheading = NULL;
        }
        Indent(log_stream, level + 2);
        fprintf(log_stream, "%s\n", equidistant_pointer→identifier);
        if (area_pointer→nearest_result→nearest_known_case ≠ NULL)
            if (Relative_Distance(equidistant_pointer→centroid_metrics.distance,
                    area_pointer→nearest_result→nearest_known_case→
                    metrics.distance) ≠ FURTHER)
                centroid_different_result = TRUE;
        if (area_pointer→nearest_result→nearest_unknown_case ≠ NULL)
            if (Relative_Distance(equidistant_pointer→centroid_metrics.distance,
                    area_pointer→nearest_result→nearest_unknown_case→
                    metrics.distance) ≠ FURTHER)
                centroid_different_result = TRUE;
    }
if (subheading ≡ NULL)
    fprintf(log_stream, "\n");
subheading = "Specified␣directions";
for (equidistant_pointer = area_pointer→strongest_specified_direction;
        equidistant_pointer ≠ NULL;
        equidistant_pointer = equidistant_pointer→equidistant_specified_direction_next)
    if (equidistant_pointer ≠ area_pointer→nearest_result) {
        if (heading ≠ NULL) {
            Indent(log_stream, level);
            fprintf(log_stream, "%s:\n\n", heading);
            heading = NULL;
        }
        if (subheading ≠ NULL) {
            Indent(log_stream, level + 1);
            fprintf(log_stream, "%s:\n", subheading);
            subheading = NULL;
        }
        Indent(log_stream, level + 2);
        fprintf(log_stream, "%s\n", equidistant_pointer→identifier);
        specified_direction_different_result = TRUE;
    }
if (subheading ≡ NULL)
    fprintf(log_stream, "\n");
```

```
    subheading = "Ideal point directions";

    for (equidistant_pointer = area_pointer→strongest_ideal_point_direction;
         equidistant_pointer ≠ NULL;
         equidistant_pointer = equidistant_pointer→equidistant_ideal_point_direction_next)
      if (equidistant_pointer ≠ area_pointer→nearest_result) {
        if (heading ≠ NULL) {
          Indent(log_stream, level);
          fprintf(log_stream, "%s:\n\n", heading);
          heading = NULL;
        }
        if (subheading ≠ NULL) {
          Indent(log_stream, level + 1);
          fprintf(log_stream, "%s:\n", subheading);
          subheading = NULL;
        }
        Indent(log_stream, level + 2);
        fprintf(log_stream, "%s\n", equidistant_pointer→identifier);
      }
    if (subheading ≡ NULL)
      fprintf(log_stream, "\n");

    subheading = "Centroid directions";

    for (equidistant_pointer = area_pointer→strongest_centroid_direction;
         equidistant_pointer ≠ NULL;
         equidistant_pointer = equidistant_pointer→equidistant_centroid_direction_next)
      if (equidistant_pointer ≠ area_pointer→nearest_result) {
        if (heading ≠ NULL) {
          Indent(log_stream, level);
          fprintf(log_stream, "%s:\n\n", heading);
          heading = NULL;
        }
        if (subheading ≠ NULL) {
          Indent(log_stream, level + 1);
          fprintf(log_stream, "%s:\n", subheading);
          subheading = NULL;
        }
        Indent(log_stream, level + 2);
        fprintf(log_stream, "%s\n", equidistant_pointer→identifier);
      }
    if (subheading ≡ NULL)
      fprintf(log_stream, "\n");
```

```
/* issue warnings if necessary */

if (weighted_different_result)
    warning(log_stream,
            "one␣or␣both␣of␣the␣weighted␣safeguard␣metrics␣suggest␣"
            "that␣a␣case␣(or␣cases)␣with␣a␣different␣result␣should␣"
            "be␣the␣nearest␣neighbour␣(or␣neighbours)", level);

if (ideal_point_different_result) {
    sprintf(message,
            "one␣or␣more␣ideal␣points␣with␣a␣different␣result␣are␣at␣"
            "least␣as␣near␣to␣the␣%s␣case␣as␣is␣the␣nearest␣neighbour",
            instant_case_type);
    warning(log_stream, message, level);
}
if (centroid_different_result) {
    sprintf(message,
            "one␣or␣more␣centroids␣with␣a␣different␣result␣are␣at␣"
            "least␣as␣near␣to␣the␣%s␣case␣as␣is␣the␣nearest␣neighbour",
            instant_case_type);
    warning(log_stream, message, level);
}
if (specified_direction_different_result)
    warning(log_stream,
            "the␣specified␣directions␣suggest␣a␣different␣result␣or␣results",
            level);
}

extern void
Write_Report(
        file report_stream,
        file log_stream,
        area *area_pointer,
        vector_element *facts_head,
        vector_element *original_facts,
        boolean verbose,
        boolean hypothetical,
        boolean same_result,
        cardinal number,
        cardinal level)

/* Writes SHYSTER's legal opinion about the facts pointed to by facts_head to report_stream
   (if it is not NULL). Cases are summarized in full, and opening and closing strings are
   written in full, if verbose is TRUE.

   If number is not zero then the instant case is actually a hypothetical (if hypothetical is
   TRUE) or an instantiation, and number is its number. If the instant case is an instantiation
   or a hypothetical, original_facts points to the facts of the uninstantiated and unhypothesized
   instant case. If the instant case is a hypothetical and same_result is TRUE, the hypothetical
   has the same result as the unhypothesized instant case. */
```

```
{
    result *result_pointer;
    kase *case_pointer,
        *nearest_neighbour;
    string instant_case_type;

    if (number ≡ 0) {
    /* the instant case is the uninstantiated and unhypothesized instant case */

    instant_case_type = "instant";

    if (report_stream ≠ NULL) {

        fprintf(report_stream, "%s{%s area}\n\n"
                "%s{Instant case}\n\n",
                Heading, area_pointer→identifier, Subheading);

        /* write the opening string */

        if (area_pointer→opening ≠ NULL) {
            if (verbose)
                Write(report_stream, area_pointer→opening, "\n", Top_Level, Hang);
            else
                Write(report_stream, "[Opening.]", "\n", Top_Level, Hang);
            fprintf(report_stream, "%s ", Skip);
            }

            /* write the facts of the instant case */

            fprintf(report_stream, "In the instant case,\n");
            list_facts(report_stream, facts_head, area_pointer→attribute_head,
                    area_pointer→number_of_attributes);

            fprintf(report_stream, "\n%s In my opinion", Skip);
        }
    } else if (¬hypothetical) {

        /* the instant case is instantiation number */

        instant_case_type = "instantiated";

        if (report_stream ≠ NULL) {

            fprintf(report_stream, "%s{Instantiation %u}\n\n", Subheading, number);
            fprintf(report_stream,
                    "It may be that the following is true of the instant case:\n");
            list_new_differences(report_stream, facts_head, original_facts,
                    area_pointer→attribute_head,
                    number_of_differences(facts_head, original_facts));

            fprintf(report_stream, "\n%s If that is so then in my opinion", Skip);
    }
```

```
} else {
    /* the instant case is hypothetical number */
    instant_case_type = "hypothetical";
    if (report_stream ≠ NULL) {
        fprintf(report_stream, "%s{Hypothetical␣%u}\n\n", Subheading, number);
        fprintf(report_stream, "Consider␣the␣instant␣case␣changed␣"
            "so␣that␣the␣following␣is␣true:\n");
list_new_differences(report_stream, facts_head, original_facts,
        area_pointer→attribute_head,
        number_of_differences(facts_head, original_facts));
    if (same_result)
        /* this hypothetical has the same result as does the instant case */
        fprintf(report_stream,
                "\n%s␣If␣that␣were␣so␣then␣I␣would␣be␣"
                "more␣strongly␣of␣the\n"
                "opinion␣that", Skip);
    else
        /* this hypothetical has a different result to that of the instant case */
        fprintf(report_stream,
                "\n%s␣If␣that␣were␣so␣then␣my␣opinion␣would␣be\n"
                "that", Skip);
    }
}
Indent(log_stream, level);
fprintf(log_stream, "Nearest␣neighbours:\n\n");
Indent(log_stream, level + 1);
fprintf(log_stream, "%s:\n", area_pointer→nearest_result→identifier);
if (area_pointer→nearest_result→nearest_known_compared_with_unknown ≠ FURTHER) {
    /* the nearest known neighbour is the nearest neighbour (although there may be an
       equidistant case with an unknown distance) */
    nearest_neighbour = area_pointer→nearest_result→nearest_known_case;
    /* for every nearest known case with this result ... */
    for (case_pointer = area_pointer→nearest_result→nearest_known_case;
            case_pointer ≠ NULL;
            case_pointer = case_pointer→equidistant_known_next) {
        /* log the case (a nearest known neighbour) */
        Indent(log_stream, level + 2);
        log_case_number_and_name(log_stream, case_pointer→number,
                case_pointer→short_name);
        if (Is_Zero_Subdistance(case_pointer→metrics.distance.known))
            fprintf(log_stream, "␣(identical)");
        fprintf(log_stream, "\n");
}
```

```
    if (has_less_known_distance(area_pointer→nearest_result→nearest_unknown_case,
                                area_pointer→nearest_result→nearest_known_case))

        /* if not for its unknown distance, the nearest unknown neighbour would be the
           nearest neighbour, so for every nearest unknown case with this result ... */

        for (case_pointer = area_pointer→nearest_result→nearest_unknown_case;
             case_pointer ≠ NULL;
             case_pointer = case_pointer→equidistant_unknown_next) {

            /* log the case (a nearest unknown neighbour) */

            Indent(log_stream, level + 2);
            log_case_number_and_name(log_stream, case_pointer→number,
                                     case_pointer→short_name);
            fprintf(log_stream, "\n");

    } else

        /* the nearest unknown neighbours should be ignored */

        area_pointer→nearest_result→nearest_unknown_case = NULL;

    fprintf(log_stream, "\n");

    if (report_stream ≠ NULL)
        handle_nearest_known(report_stream, area_pointer→nearest_result,
                             area_pointer→nearest_result, nearest_neighbour,
                             area_pointer, facts_head, instant_case_type, verbose);

} else {

    /* the nearest unknown neighbour is the nearest neighbour */

    nearest_neighbour = area_pointer→nearest_result→nearest_unknown_case;

    /* for every nearest unknown case with this result ... */

    for (case_pointer = area_pointer→nearest_result→nearest_unknown_case;
         case_pointer ≠ NULL;
         case_pointer = case_pointer→equidistant_unknown_next) {

        /* log the case (a nearest unknown neighbour) */

        Indent(log_stream, level + 2);
        log_case_number_and_name(log_stream, case_pointer→number,
                                 case_pointer→short_name);
        fprintf(log_stream, "\n");
    }
```

```
/* for every nearest known case with this result ... */
for (case_pointer = area_pointer→nearest_result→nearest_known_case;
        case_pointer ≠ NULL;
        case_pointer = case_pointer→equidistant_known_next) {
    /* log the case (a nearest known neighbour) */
    Indent(log_stream, level + 2);
    log_case_number_and_name(log_stream, case_pointer→number,
            case_pointer→short_name);
    fprintf(log_stream, "\n");
  }
  fprintf(log_stream, "\n");
  if (report_stream ≠ NULL)
    handle_nearest_unknown(report_stream, area_pointer→nearest_result,
            area_pointer, facts_head, instant_case_type, verbose);
}
Indent(log_stream, level);
fprintf(log_stream, "Nearest others:\n\n");
for (result_pointer = area_pointer→result_head; result_pointer ≠ NULL;
        result_pointer = result_pointer→next)
  /* for every result ... */
  if (result_pointer ≠ area_pointer→nearest_result) {
    /* this result is not the nearest result */
    Indent(log_stream, level + 1);
    fprintf(log_stream, "%s:\n", result_pointer→identifier);
    if ((result_pointer→nearest_known_case ≠ NULL) ∨
        (result_pointer→nearest_unknown_case ≠ NULL)) {
      /* this result has a nearest case (i.e. it has at least one case) */
      if ((result_pointer→nearest_known_compared_with_unknown ≡ FURTHER) ∨
          has_less_known_distance(result_pointer→nearest_unknown_case,
              nearest_neighbour))
        /* the nearest unknown other is the nearest other or, if not for its unknown
           distance, the nearest unknown other would be the nearest neighbour, so
           for every nearest unknown case with this result ... */
        for (case_pointer = result_pointer→nearest_unknown_case;
            case_pointer ≠ NULL;
            case_pointer = case_pointer→equidistant_unknown_next) {
          /* log the case (a nearest unknown other) */
          Indent(log_stream, level + 2);
          log_case_number_and_name(log_stream, case_pointer→number,
              case_pointer→short_name);
          fprintf(log_stream, "\n");
        }
```

```
    /* for every nearest known case with this result ... */
    for (case_pointer = result_pointer→nearest_known_case;
         case_pointer ≠ NULL;
         case_pointer = case_pointer→equidistant_known_next) {
      /* log the case (a nearest known other) */
      Indent(log_stream, level + 2);
      log_case_number_and_name(log_stream, case_pointer→number,
                               case_pointer→short_name);
      fprintf(log_stream, "\n");
    }
    if (report_stream ≠ NULL)
      handle_nearest_others(report_stream, result_pointer,
                            area_pointer→nearest_result, nearest_neighbour,
                            area_pointer→nearest_result→
                            nearest_known_compared_with_unknown ≠ FURTHER,
                            area_pointer, facts_head, instant_case_type, verbose);
  }
  fprintf(log_stream, "\n");
}
implement_safeguards(log_stream, area_pointer, instant_case_type, level);
/* log the nearest result */
Indent(log_stream, level − 1);
fprintf(log_stream, "Nearest␣result␣for␣");
if (number ≡ 0)
  /* this is the unhypothesized, uninstantiated instant case */
  fprintf(log_stream, "the␣instant␣case");
else if (¬hypothetical)
  /* this is an instantiation */
  fprintf(log_stream, "instantiation␣%u", number);
else
  /* this is a hypothetical */
  fprintf(log_stream, "hypothetical␣%u", number);
fprintf(log_stream, "␣is␣%s.\n\n", area_pointer→nearest_result→identifier);
if ((report_stream ≠ NULL) ∧ (number ≡ 0) ∧ (area_pointer→closing ≠ NULL)) {
  /* write the closing string */
  fprintf(report_stream, "%s\n", Skip);
  if (verbose)
    Write(report_stream, area_pointer→closing, "\n", Top_Level, Hang);
  else
    Write(report_stream, "[Closing.]", "\n", Top_Level, Hang);
 }
}
```

Bibliography

Kernighan, Brian W. and Ritchie, Dennis M. 1988, *The C Programming Language* (second edition), Prentice Hall, Englewood Cliffs, New Jersey. ISBN 0 13 110362 8.

Knuth, Donald E. 1984, *The TEXbook*, Addison-Wesley, Reading, Massachusetts. ISBN 0 201 13448 9.

Knuth, Donald E. 1986a, *TEX: The Program*, Computers and Typesetting, vol. B, Addison-Wesley, Reading, Massachusetts. ISBN 0 201 13437 3.

Knuth, Donald E. 1986b, *METAFONT: The Program*, Computers and Typesetting, vol. D, Addison-Wesley, Reading, Massachusetts. ISBN 0 201 13438 1.

Knuth, Donald E. and Levy, Silvio 1994, *The CWEB System of Structured Documentation*, Manual, Version 3.0.

Lamport, Leslie 1986, *LATEX: A Document Preparation System*, Addison-Wesley, Reading, Massachusetts. ISBN 0 201 15790 X.

Describes LaTeX 2.09.

Lamport, Leslie 1994, *LATEX: A Document Preparation System* (second edition), Addison-Wesley, Reading, Massachusetts. ISBN 0 201 52983 1.

Describes LaTeX 2_ε.

Popple, James 1993, *SHYSTER: A Pragmatic Legal Expert System*, PhD thesis, The Australian National University, Canberra, April. ISBN 0 7315 1827 6.

Popple, James 1996, *A Pragmatic Legal Expert System*, Applied Legal Philosophy Series, Dartmouth, Aldershot. ISBN 1 85521 739 2.

Index

All of the identifiers which appear in the code listings in this report, and which are not reserved words or preprocessor commands, are listed below.

Function names are followed by a pair of parentheses. The names of SHYSTER's defined types are marked with a star.

SHYSTER's external identifiers are made up of upper- and lower-case letters; its static identifiers consist only of lower-case letters. Identifiers in all upper-case are enumerated identifiers and other constants.

Page numbers set in boldface type refer to definitions of functions or constants, or to declarations of types or variables (including structure member declarations). Page numbers set in italics refer to function declarations.

Where an identifier is defined in an ISO C standard library, the name of that library appears in angle brackets.