Random Crossword Generator with a Web User Interface

Yvette Graham
B.A. (Mod.) CSLL
Final Year Project, May 2003
Supervisor: Dr. Carl Vogel
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_________________________________________ May 2, 2003
Yvette Graham
Acknowledgements

Firsty I would like to thank my supervisor, Dr. Carl Vogel. Carl, you’ve been a great support to me this year. Everyone really appreciates the time and care you devote to all your students, we’ve been very lucky to have you, so thanks. Another person I would like to thank is Nicola McLelland, who seems to be always looking out for ways to help the CSLG class. Finally I would like to thank my mother, Eileen O’Brien, for all her help over the years. Your guidance and support have been a constant source of motivation for me. Thanks Mum.
“Pooh looked at his two paws. He knew that one of them was the right, and he knew that when you had decided which one of them was the right, then the other was the left, but he never could remember how to begin.” – A.A. Milne
Contents

1 Introduction 1
   1.1 What is is meant by Random Crossword Generator . . . . . . . . . . . . . 1
   1.2 Why did I decide to extend the Crossword Generator Project . . . . . . . 1
   1.3 Uses of the Crossword Generator . . . . . . . . . . . . . . . . . . . . . . 2
   1.4 What’s to come . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Generating Interesting Grids 4
   2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
   2.2 User Input . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
      2.2.1 The Difficulty Rating . . . . . . . . . . . . . . . . . . . . . . . . . . 4
   2.3 The Random Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   2.4 Symmetrical Grids . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      2.4.1 The Old Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      2.4.2 The New Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3 Selection of Clues and Answers 10
   3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   3.2 The Database Management System . . . . . . . . . . . . . . . . . . . . . . 10
      3.2.1 MySql . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
      3.2.2 Java and MySql . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   3.3 Dictionary Database . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
      3.3.1 Suitable Dictionaries . . . . . . . . . . . . . . . . . . . . . . . . . . 12
      3.3.2 Adding a new dictionary to the System . . . . . . . . . . . . . . . . 13
      3.3.3 Removing a Dictionary from the System . . . . . . . . . . . . . . . . 16
   3.4 English Dictionary Database . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      3.4.1 Design of Original Database . . . . . . . . . . . . . . . . . . . . . . 18
      3.4.2 Determining Redundancy of Data . . . . . . . . . . . . . . . . . . . 19
      3.4.3 Eliminating Redundant Data . . . . . . . . . . . . . . . . . . . . . . 19
      3.4.4 Advantages of Normalized Databases . . . . . . . . . . . . . . . . . 21
      3.4.5 The New English Dictionary Database . . . . . . . . . . . . . . . . . 22
      3.4.6 Populating the Database . . . . . . . . . . . . . . . . . . . . . . . . . 22
      3.4.7 Randomly Jumbling The Words . . . . . . . . . . . . . . . . . . . . . 24

4
## CONTENTS

3.5 Conclusion ............................................................... 25

4 User Interface ........................................................... 27
  4.1 Introduction .......................................................... 27
  4.2 The Java Applet ...................................................... 27
    4.2.1 Sockets and Server Sockets .................................. 27
  4.3 The Old Interface ................................................... 28
    4.3.1 The Clues ....................................................... 28
    4.3.2 The Grid ....................................................... 28
    4.3.3 Check Solutions Function ................................... 29
    4.3.4 The Code ....................................................... 29
  4.4 The New Interface ................................................... 29
    4.4.1 The Squares .................................................... 30
    4.4.2 The Solutions .................................................. 30
    4.4.3 The Crosswords .............................................. 31
    4.4.4 Use of Data Structures ....................................... 31
    4.4.5 Drawing the Grid ............................................. 31
    4.4.6 Numbering the Clues ......................................... 32
    4.4.7 Moving Around the Grid ..................................... 32
    4.4.8 Checking Validity of Letters input by the User .......... 33
  4.5 Conclusion ........................................................... 34

5 Prolog Solutions to Solving the Grid ................................. 39
  5.1 Introduction .......................................................... 39
  5.2 Crossword Compilation with Horn Clauses ......................... 39
    5.2.1 The Logic behind the Program ............................... 39
    5.2.2 The Prolog Program ......................................... 40
    5.2.3 Results ........................................................ 40
    5.2.4 The Search Space ............................................. 41
    5.2.5 Strategies for Reducing Search Space ...................... 42
    5.2.6 Evaluation ..................................................... 42
  5.3 Crossword Compiler-Compilation .................................. 44
    5.3.1 The Strategy .................................................. 44
    5.3.2 Evaluation ..................................................... 44
  5.4 The Inherited Prolog Solution .................................... 45
  5.5 The Old Prolog Code ................................................ 45
    5.5.1 How it worked ................................................ 45
    5.5.2 What was wrong with it ..................................... 46
  5.6 The New Approach ................................................... 47
    5.6.1 The Input ....................................................... 47
    5.6.2 Naive Backtracking ......................................... 47
    5.6.3 Avoiding Naive Backtracking ................................. 48
    5.6.4 Ordering the Word Slots ..................................... 48
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Flow Diagram to Illustrate how Crosswords are Generated</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>A Grid with Many Squares Interlinking Words</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>A Grid with Few Squares Interlinking Words</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Ratios of Black to White Squares</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>180 Degree Symmetrical Grid</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>90 Degree Symmetrical Grid</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>Half Grid with Randomly Placed Black Squares</td>
<td>8</td>
</tr>
<tr>
<td>2.7</td>
<td>Formula used to calculate the positions of the black squares in second half of grid</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>A Symmetrical Grid</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Example SQL statement used by Oracle.</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Example SQL statement used by MySql.</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>dictionary_data table</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of a table format that could be used by the system</td>
<td>13</td>
</tr>
<tr>
<td>3.5</td>
<td>SQL statements to change the name of columns</td>
<td>13</td>
</tr>
<tr>
<td>3.6</td>
<td>Example input file format</td>
<td>14</td>
</tr>
<tr>
<td>3.7</td>
<td>dictionary4</td>
<td>15</td>
</tr>
<tr>
<td>3.8</td>
<td>SQL statement to load a file into a table</td>
<td>15</td>
</tr>
<tr>
<td>3.9</td>
<td>dictionary_data table</td>
<td>16</td>
</tr>
<tr>
<td>3.10</td>
<td>SQL statement to remove Logic Dictionary entry from dictionary_data</td>
<td>17</td>
</tr>
<tr>
<td>3.11</td>
<td>SQL statement to retrieve the table_name of Logic Dictionary</td>
<td>17</td>
</tr>
<tr>
<td>3.12</td>
<td>SQL statement to delete table dictionary3</td>
<td>17</td>
</tr>
<tr>
<td>3.13</td>
<td>Example entries in the original database</td>
<td>18</td>
</tr>
<tr>
<td>3.14</td>
<td>The Boyce/Codd rule for Determining Redundancy of Data</td>
<td>19</td>
</tr>
<tr>
<td>3.15</td>
<td>Definitions of a Determinant and a Candidate Identifier</td>
<td>19</td>
</tr>
<tr>
<td>3.16</td>
<td>Conceptual Model Rules for Original Database</td>
<td>20</td>
</tr>
<tr>
<td>3.17</td>
<td>The word_data table</td>
<td>20</td>
</tr>
<tr>
<td>3.18</td>
<td>The definition_data table</td>
<td>21</td>
</tr>
<tr>
<td>3.19</td>
<td>Conceptual Model Rules for New Database. (see also fig. 3.15)</td>
<td>21</td>
</tr>
<tr>
<td>3.20</td>
<td>Excerpt from Webster’s English Dictionary file</td>
<td>23</td>
</tr>
<tr>
<td>3.21</td>
<td>Excerpt from file used to populate tables in Cleare(2002)</td>
<td>26</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.1 Old User Interface ................................................. 35
4.2 Example Code from Old Applet ................................. 36
4.3 Example of how code in fig. 4.2 should be written .......... 36
4.4 Example paint() method. ........................................... 37
4.5 The New User Interface ........................................... 38

5.1 Prolog Clause to find Solution .................................. 41
5.2 Berghel’s Explanation of the Potential Size of the Search Space .... 42
5.3 Berghel’s Method of Avoiding Naive Backtracking .............. 43
5.4 Example Grid ....................................................... 45
5.5 Original Encoding of the Lexicon ................................. 46
5.6 The Original Prolog Code used to solve the grid. ............... 52
5.7 Representing the lexicon in Prolog. .............................. 53
5.8 The Variable Names given to the Cells .......................... 53
5.9 Example Input for Grid shown in fig. 5.8. ...................... 53
5.10 Example To Illustrate Naive Backtracking ..................... 54
5.11 Evaluating the Words ............................................. 55

6.1 Predicate solve/3 ................................................... 59
Abstract

The aim of this project was to extend a project that involved the random generation of crossword puzzles. The program is written in Java and Prolog. It firstly generates a random symmetrical grid and then fills up the grid with appropriate words. The crosswords that are produced have a theme which the user decides upon and the size of the grid may also be determined by the user. Further control that the user has on the produced puzzle are its difficulty and the dictionary that is used as a source of clues and answers. The solution to the predefined grid is found using Prolog. The project including optimising the prolog code, by placing constraints on it to avoid naive backtracking. It was also necessary to interface Java and Prolog programs. This was done using the Jasper package. The crossword generator is web interfaced and therefore after a grid has been generated and a solution has been found, the empty crossword grid is displayed to the user, enabling them to attempt to solve the puzzle on-line.
Chapter 1

Introduction

1.1 What is meant by Random Crossword Generator

The Random Crossword Generator implemented in this project is a piece of software that can be used to create and solve random crossword puzzles. The parts of the puzzle that can be specified are its width, subject and difficulty rating, as well as the dictionary that will be used to supply its clues and answers. The crossword that is produced will contain a symmetrical pattern of black and white cells, the locations of which have been randomly chosen. The clues and answers of the crossword are thematically related to the subject, which is specified by the user. These clues and answers are randomly shuffled before a solution is found using them, which also adds to the randomness of the crossword puzzles that are produced.

1.2 Why did I decide to extend the Crossword Generator Project

The project I extended is described in Gibbons (2001). The first reason the project took my interest is because I enjoy doing crossword puzzles. Therefore the idea of creating crosswords crosswords as opposed to solving them interested me a great deal. When you think about it, if a crossword is difficult to solve, then how difficult must it have been to create it? Also if it was your job to create a different crossword every day, it could be very difficult to create a unique puzzle day after day. Another reason the project interested me was that for some people solving crosswords is very simple and they can do it very quickly. Once told the length of a word and maybe one or two of the letters it contains, they seem to be able to think of very many words that could fit. It is this behavior that was needed to be mimicked by a program in order to fill up a grid with words. It is often the case that when it comes to dealing with words and grammars, computers are much less able to cope with them than humans. Therefore the idea of trying to achieve something using a
computer that can be easily achieved by certain humans interested me.

1.3 Uses of the Crossword Generator

Crosswords have been around now for over a hundred years. Although mostly published in newspapers and magazines, crosswords are now popping up all over the internet due to their everlasting popularity. They seem to hold the interest of people of any age and must be the most popular form of puzzle around. Thus the main use for the crossword generator is simply for fun. People enjoy using it. In addition, giving the user the ability to customize the crossword generated improves the enjoyment value. Often a problem with crosswords is that they are too difficult or in some cases too easy to solve. By being able to specify the width and difficulty rating of the crossword the user can take control of how difficult the produced puzzle will be. Also being able to specify a theme that the crossword should be about should enable the user to test themselves on their knowledge of a certain subject.

1.4 What’s to come

The crossword generator is made up of various parts which are described in the various chapters of this dissertation. Each part is explained in detail in the appropriate chapter. It is also important however to give an account of how the crossword generator works. This is illustrated in figure 1.1. I will now describe the structure of the dissertation.

Chapter 2 describes the way in which the grid is generated. It gives a detailed account of how the user’s input contributes to the structure of the grid as well as how the grid produced is random. In Chapter 3 all of the database issues are discussed. This chapter describes a major part of the project, as a database is used throughout the project for different purposes. It also describes how the user can add a dictionary to the system.

Chapter 4 contains a description of how we’ve enabled Java to communicate with Prolog. It gives a detailed account of the Jasper package used that our Java code uses to consult a Prolog file to get the solution to the grid. In chapter 5 I describe the Java Applet used to interface the crossword generator. This chapter also contains information about some of the technologies used by the interface.

A review of some of the papers which have been written about crossword generation is given in chapter 6. This gives a background to the kind of study that has been done on the automation of crossword generation in the past. Chapter 7 then describes the method used for solving the grid. In this chapter I discuss the thinking behind the Prolog code I use to fill up the grid with words. Finally in chapter 8 I give an evaluation of the project and some conclusions I came to as well as some suggestions for further work that could be done on the project.
Figure 1.1: Flow Diagram to Illustrate how Crosswords are Generated
Chapter 2

Generating Interesting Grids

2.1 Introduction

Generating interesting grids is a vital part of the generator. The layout of the grid can influence the user’s interest in solving the puzzle. Since I want the puzzles to be enjoyed by many people, it was important to make the grids as appealing to the user as possible. The following chapter describes how the user has input into the layout of the grid, how the difficulty rating relates to the resulting grid layout, the use of the Java Random class in creating the grid and finally details on the symmetry of the grids produced by the generator.

2.2 User Input

As well as the subject of the crossword and the dictionary to be used to create the crossword, the user also specifies the width of the crossword to be generated and the difficulty level. Naturally the width of the crossword will affect the structure of the crossword grid that is generated. The difficulty rating also has an effect on the structure of the grid that will be produced.

2.2.1 The Difficulty Rating

When a crossword grid is being generated, its difficulty rating has been chosen by the user. There are four different levels of difficulty from which the user must select one. They are Easy, Not Too Easy, Hard and Very Hard. The difficulty rating is used to determine the ratio of black squares to white squares in the generated grid. Crosswords in general tend to be more difficult to solve when the words in the crossword rarely interlink each other. This is due to the fact that if you’ve solved a word that interlinks another word, before even attempting the second word you will already have one of its letters solved. Thus each square in the grid that occurs within two words reduces the difficulty of the crossword puzzle. But since the grid is randomly generated and the difficulty rating is chosen prior
to the grid being generated, it is not possible to rate the difficulty on the crossword exactly by the number of squares which interlink two words. However we are able to predetermine the ratio of black squares to white squares and by doing so can approximate the number of squares that interlink two words. This can be done based on the fact that the fewer black squares that the grid contains, the more squares it will contain that interlink two words (see fig. 2.1 & fig. 2.2).

![Figure 2.1: A Grid with Many Squares Interlinking Words](image1)

![Figure 2.2: A Grid with Few Squares Interlinking Words](image2)

Therefore the difficulty rating *Easy* will produce grids with the fewest black squares, while *Very Hard* will produce grids with more black squares. Thus I decided on ratios of black squares to white squares for the various difficulty ratings (see fig. 2.3).

An interesting point to note here is that the *more* interlinking squares a crossword contains, the easier it is for a human to solve. However the opposite is true for a program
CHAPTER 2. GENERATING INTERESTING GRIDS

<table>
<thead>
<tr>
<th>Difficulty Rating</th>
<th>% Black</th>
<th>% White</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy</td>
<td>30</td>
<td>70</td>
</tr>
<tr>
<td>Not Too Easy</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>Hard</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Very Hard</td>
<td>60</td>
<td>40</td>
</tr>
</tbody>
</table>

Figure 2.3: Ratios of Black to White Squares

trying to generate a crossword puzzle. The fewer interlinking squares the grid contains, the easier it is for a program to generate it.

2.3 The Random Class

The Java class java.util.Random is used to generate a stream of pseudo-random numbers (see Sun-Microsystems, 1995). The Random object is created using a seed. If no seed is specified, then one is gotten using the current time. To get the first of the random numbers, the seed is modified and then that number is used as the seed in generating the next pseudo-random number. The modification of the seed is using a linear congruential formula, which was defined by D.H. Lehmer and is described in Knuth (1997). The stream of pseudo-random numbers is in fact a sequence of numbers, but for most purposes the numbers generated are as random as they need to be.

2.4 Symmetrical Grids

Conventional crossword puzzles are usually based on a symmetrical Grid. They can either have 180 degree symmetry or 90 degree symmetry. In a 180 degree symmetrical grid, an identical grid is achieved by rotating the grid by 180 degrees. Figure 2.4 illustrates this more clearly. On the other hand if the grid is 90 degree symmetrical, then a rotation of 90 degrees, in any direction, will achieve the same grid. Figure 2.5 shows a 90 degree symmetrical grid. The crossword generators aim was to create crosswords similar to those published in the New York Times. These crosswords are 180 degree symmetrical and therefore we decided that the generator should also create crosswords that were 180 degree symmetrical as opposed to ones which were 90 degree symmetrical.

2.4.1 The Old Method

The old method of creating a symmetrical grid was similar to mine except that involved translating triangular shaped grids into square grids. This was a very complicated way of creating a symmetrical grid. I decided to redefine how symmetrical grids were created using a simpler method.
2.4.2 The New Method

The symmetry of the grid was achieved by dividing the grid into two halves. The number of black squares to be placed in the grid by this stage has been determined. Half of the total number black squares are then randomly placed in the first half of the grid (see fig. 2.6).

The locations of the black squares in the other half of the grid is then determined based on the positions of the squares in the first half, in such a way as to create a symmetrical pattern of black squares. How this is done is based on the fact that if we start in the upper left hand corner of the grid, and count in a certain number of squares, e.g. the fourth square, and decide that this square should be black, then the square in the other half of the grid that needs to be black in order for the grid to be symmetrical is the square that we get to when we count to four starting at the bottom right hand corner. Since we have numbered the squares 0 to 35, if we count in four squares from the top left we get to square 3 and if we count back four squares from the bottom right we get square 32. Thus simple formula can be used to work out the counterpart of any square in the first half of the grid (see fig. 2.7). Figure 2.8 shows a grid this formula was used to create.
2.5 Conclusion

In this chapter gave an account of what was involved in creating the grid layout of the generated crossword. It detailed how a different grid is created everytime a crossword is generated and how the locations of the black squares in the grid is randomly determined, while keeping the grid in 180 degree symmetry. I also gave an account of how the user may have input into the layout of the grid and how the difficulty rating relates to the ratio of balck to white squares in the grid.
counterpart = total number of squares - 1 - square number

Figure 2.7: Formula used to calculate the positions of the black squares in second half of grid.
Chapter 3

Selection of Clues and Answers

3.1 Introduction

In this chapter a detailed account is given of all the issues concerning databases. Since the project involved interaction with a database this was an important part. In the sections that follow I therefore define what a Database Management System is, then describe the specific DBMS used by this project, MySql. Following that I describe how Java can interact with MySql using a database driver. Next is a detailed description of the tables I store the data in and how more dictionaries can be added or removed from the system. Finally I give a description of the English dictionary database that I created from which words and definitions were selected which were subsequently used as answers and clues in the crossword puzzles.

3.2 The Database Management System

This section introduces the concept of a database management system, and describes both the DBMS used by the crossword generator, MySql and the driver used to connect Java to MySql (see MySQL-AB, 1995).

Database Management Systems are used to administer databases. Thus a Database Management System is used both to create and maintain a database. It is also used when either a user or an external system interacts with the database. In other words accessing a database is done through its relevant DBMS. There are many types of DBMSs. For example Oracle, MS-Access, MySql, DB2 and SQLServer are all popular DBMSs. Nearly all DBMSs allow the user (or another system) to access its databases using Structured Query Language (SQL). SQL is a standard language, but can vary in slight ways from one DBMS to another. For example Oracle requires the user to specify the full name of a column when selecting across two tables, whereas when using MySql entering the full name of a column is only necessary when there is ambiguity between the column names. For example, given two tables one called book_data and the other called author_data. To select the name of a book from the book_data table and the name of its author from the
author_data table in Oracle you need the SQL statement shown in figure 3.1. When using
MySQL however the statement shown in figure 3.2 can be used to achieve the same thing,
because none of the columns have the same names. This is a handy feature of MySQL.

```sql
SELECT book_data.book_name, author_data.author_name FROM book_data, author_data
    WHERE book_data.author = author_data.author_id;
```

Figure 3.1: Example SQL statement used by Oracle.

```sql
SELECT book_name, author_name FROM book_data, author_data
    WHERE author = author_id;
```

Figure 3.2: Example SQL statement used by MySQL.

### 3.2.1 MySQL

MySQL was chosen as the database management system, as it had been previously used
for the project when I took it on and there seemed to be no problems with using it.
MySQL was originally chosen as the DBMS because it is available to use in the Computer
Science Department and its driver is available to download for free from the Internet. The
database that was set up for this project is called *dictionary* and is located on a machine
called *macneill* (See Appendix).

### 3.2.2 Java and MySQL

For a program to access data contained in a database it is necessary to use a driver. Drivers
used to connect Java to a database are called JDBC drivers. The DBMS vendor (in this
case MySQL) provides a free JDBC driver. This is available free from MySQL-AB (1995).
Once this is downloaded and if necessary unzipped, its location must be added to the
class-paths set on the machine that the server is running on (e.g. sun40). Setting up the
driver allows the Java code to interact with MySQL. It can now for example create tables,
access data, delete data etc. in any MySQL database that it has permission to access, using
SQL.
3.3 Dictionary Database

As the system can use a number of different dictionaries to create crosswords, it was necessary to store certain information about each of the available dictionaries. I decided to use a database to store this information, as it’s a structured and secure way of storing data. Therefore I created a table to store the data about the relevant dictionaries and called it dictionary_data (stored in dictionary_database on macneill). As mentioned above each dictionary that the system uses is in the form of a database. The table, dictionary_data stores the name of the dictionary e.g. "Websters English Dictionary", the location of the database in which the dictionary is stored e.g. on macneill, the name of the table containing the words and definitions e.g. dictionary3, and the location of a suitable driver e.g. the location of a MySql JDBC driver. This information is sufficient to connect to the DBMS managing the database in question and to retrieve words and definitions from the appropriate table. Figure 3.3 illustrates more clearly the structure of the table containing the dictionary data.

<table>
<thead>
<tr>
<th>dict_id</th>
<th>name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Webster’s English Dictionary</td>
<td>jdbc:mysql://macneill ...</td>
</tr>
<tr>
<td>2</td>
<td>French Dictionary</td>
<td>jdbc:mysql://macneill ...</td>
</tr>
<tr>
<td>3</td>
<td>German Dictionary</td>
<td>jdbc:oracle:thin:@ela ...</td>
</tr>
<tr>
<td>4</td>
<td>Hungarian Dictionary</td>
<td>jdbc:microsoft:sqlserver: ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>driver</th>
<th>table_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.gjt.mm.mysql.Driver</td>
<td>definition_data</td>
</tr>
<tr>
<td>org.gjt.mm.mysql.Driver</td>
<td>french_dict_data</td>
</tr>
<tr>
<td>oracle.jdbc.driver.OracleDriver</td>
<td>german_table</td>
</tr>
</tbody>
</table>
| com.microsoft.jdbc.sqlserver.SQLServerDriver | dict_hung_table |}

Figure 3.3: dictionary_data table

3.3.1 Suitable Dictionaries

The system can use a dictionary stored in any type of relational database management system e.g. MySql, Oracle etc. The database can also contain any number of tables. The system will use the correct table to select words and definitions from. The table containing the words and definitions must have one column called word containing the head-words (which will be used as answers to the clues) and one column called definition containing the definitions (used as clues). For example the table format shown in figure 3.4 could be used by the system.

However if a dictionary is available to use, whose columns containing the words and definitions are not called words and definitions the dictionary can still be used. All that needs to be done is to change the appropriate column names of the table to word and
CHAPTER 3. SELECTION OF CLUES AND ANSWERS

<table>
<thead>
<tr>
<th>wordID</th>
<th>word</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cat</td>
<td>a feline mammal used to catch mice</td>
</tr>
<tr>
<td>2</td>
<td>dog</td>
<td>a canine mammal associated with the wolf</td>
</tr>
<tr>
<td>3</td>
<td>line</td>
<td>the shortest distance between two points</td>
</tr>
<tr>
<td>4</td>
<td>oasis</td>
<td>a fertile place in a desert</td>
</tr>
</tbody>
</table>

Figure 3.4: Example of a table format that could be used by the system.

definition. Figure 3.5 shows the two SQL statements that could be used to change the names of two columns of table french_dict called french_word and english_word to word and definition.

```
ALTER TABLE french_dict CHANGE french_word word varchar(50);
ALTER TABLE french_dict CHANGE english_word definition blob;
```

Figure 3.5: SQL statements to change the name of columns.

3.3.2 Adding a new dictionary to the System

As a useful feature of the crossword generator I decided to include the function of being able to add a dictionary to the system, so that crosswords can be generated using the dictionary. This feature has many advantages. It may simply be used to generate crosswords which are more suited to the individual user or it could be used as a teaching tool for almost any level of education. A teacher can easily provide the system with the exact set of vocabulary that is relevant to the lesson, and a crossword may be generated from this set of words and definitions. It could also very easily be used for foreign language learning. In this case the head-words would be words of the foreign language in question and the definitions would be the mother tongue translation of these head-words. This would produce very interesting crossword puzzles.

When a new dictionary is added to the system it is permanently added, so that the new dictionary is available to the user when he or she revisits the crossword generator web-site (see also 3.3.3). The user may add this new dictionary in the form of either a text file or a database. To add a new dictionary the user simply clicks on Other in the choice of dictionaries on the web page. A pop up box appears asking whether the new dictionary is in the form of a file or a database. If the new dictionary is in the form of a file, the user
enters the word *file* in the provided input box. But if the new dictionary is in the form of a database, the user enters the word *database*.

**Adding a Dictionary in the Form of a File**

After the user has input that the new dictionary is in the form of a file, the system then prompts him or her to enter the location of the file. When the location has been entered and the file has been found, the system creates a new database table to store the words and definitions contained in the file. The program then reads through the file line by line, inserting the data into the database as it goes. The file must be in the format shown in figure 3.6. The first line of the file should contain a word to be entered and the second line should contain the definition of this word. The third line should contain the next word to be entered and the fourth line its definition and so on. Figure 3.6 shows an example of a section of a file that could be added as a new dictionary to the system. Punctuation can be added to the definition if desired, but no punctuation should be contained in the line containing the words. More restrictions on the format of the file are that each word and definition is contained on a single line, the line following that of a word contains its definition, the first line of the file contains a word and there are no blank or nonsense lines in the file.

```
Gaggle
Goose Gathering.
Stingers
Wasps’ weapons.
Human
Bipedal Primate.
Runt
Littlest litter-mate.
Neigh
Horse sound.
```

Figure 3.6: Example input file format.

The tables storing the new dictionaries are also stored in the database called *dictionary* located on *macneill*. Each table containing a new dictionary is given the name *dictionary* followed by a number. The number it should be given is the minimum number that has not yet been used to number a table in the database. For example if the database already contains a table named *dictionary3*, when a new dictionary is created, its table will be called *dictionary4*. So before a new dictionary table is created the names of the others are selected from the database. The names are then compared with each other in order to find the one with the largest number. One is then simply added to this number in order to get
the number of the new dictionary table. New tables are created in the form illustrated in figure 3.7.

<table>
<thead>
<tr>
<th>word</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaggle</td>
<td>Goose Gathering.</td>
</tr>
<tr>
<td>Stingers</td>
<td>Wasps’ weapons.</td>
</tr>
<tr>
<td>Human</td>
<td>Bipedal Primate.</td>
</tr>
<tr>
<td>Runt</td>
<td>Littlest litter-mate.</td>
</tr>
<tr>
<td>Neigh</td>
<td>Horse sound.</td>
</tr>
</tbody>
</table>

Figure 3.7: dictionary4

It is worth mentioning that there is a simpler way to insert the data from a file into the tables of the database. The SQL statement shown in figure 3.8 can be used to load a file into a table very quickly (see MySQL-AB, 1995). Some versions of MySql do not allow a program to use this statement, but allow the statement to be used when accessing the DBMS directly. The statement loads the file directly into the columns of the new table which has just been created. The file that is loaded must also be in a specific format i.e. the desired contents of each column must be separated in the file by a new line. Thus when the file is being prepared it is necessary to press enter after each word and each definition has been typed. This is a very quick way of loading a large file into a table. It was not used here though because the version of MySql that is available to use in the Computer Science Department is version 3.23.49-log. This version of MySql does not allow a program to use this statement. Access is not allowed for the following security reason; if a program can load a file in this way from the client’s machine, this would give the DBMS access to files located on the clients machine that it normally wouldn’t have access to (see MySQL-AB, 1995). Because this version of MySql did not allow this SQL statement to be used by a program I did not use it, but if a different DBMS that supported this statement was used it would have been a much quicker way of loading the file into the table.

```
LOAD DATA LOCAL INFILE <address of file> INTO TABLE <table name> FIELDS TERMINATED BY '\n';
```

Figure 3.8: SQL statement to load a file into a table.

After the file has been loaded into the appropriate table, the details of the new dictionary are added to the table containing the information about all the dictionaries available to the system i.e. dictionary.data (see fig. 3.9).
Adding a Dictionary in the Form of a Database

When a dictionary in the form of a database is added to the system, the user must enter the name of the new dictionary, its address, the location of a suitable driver to enable Java to connect to its DBMS and the name of the table containing the words and definitions which will be used to create the crossword. The location of a driver must be specified so that the program can connect to any kind of DBMS e.g. Oracle, DB2 etc. Instead of copying the information to a new table, as done above when a dictionary file is added to the system, the system simply records the information that it needs to connect to this database and adds the dictionary to the choice of dictionaries available to the user i.e. by adding it to the dictionary_data table (see fig. 3.9). Then each time the new dictionary is selected by the user to create a crossword from, the system simply connects to this database to retrieve the words and definitions.

3.3.3 Removing a Dictionary from the System

As some of the dictionaries have taken a good amount of time to create, I decided that the user should not have access to deleting dictionaries from the system. Therefore a dictionary may not be deleted from the web page, but only by directly connecting to a Computer Science MySql DBMS stored on macneill e.g. from one of the sun machines.
So after connecting to macneill (See Appendix), the relevant entry must be deleted from
the table containing the information about all of the dictionaries available to the system
(dictionary_data). Figure 3.10 shows the SQL statement used to remove the entry for Logic
Dictionary from the table dictionary_data.

```
DELETE FROM dictionary_data WHERE name = "Logic Dictionary";
```

Figure 3.10: SQL statement to remove Logic Dictionary entry from dictionary_data.

If the dictionary we are deleting from the system was one that was created from a text
file, and therefore stored in a table in the database, this table should also be deleted from
the database. To do this, firstly the name of the table containing the dictionary must be
gotten from the dictionary_data table. The SQL statement in figure 3.11 will achieve
this.

```
SELECT table_name FROM dictionary_data WHERE name = "Logic Dictionary";
```

Figure 3.11: SQL statement to retrieve the table_name of Logic Dictionary.

Then the SQL statement shown in figure 3.12 could be used to delete the table. For
this example I shall assume that the name of that table containing the Logic Dictionary is
called dictionary3.

```
DROP TABLE dictionary3;
```

Figure 3.12: SQL statement to delete table dictionary3.

### 3.4 English Dictionary Database

When the project was taken on, an English dictionary database was already available
for the system to use. This dictionary was created using Webster’s Unabridged English
Dictionary, which is available in the form of a html tagged file from Project Gutenberg
and is free to download. Project Gutenberg is a project set up to make information, books and other material available to the general public in forms that computers, programs and people can easily use (see Hart, 1971). The version of the dictionary that was downloaded was (Webster, 1996). I decided after much consideration not to use the old dictionary database, but to create my own from scratch. The sections that follow describe how I came to the decision that I should create a new English dictionary.

### 3.4.1 Design of Original Database

The original dictionary database consisted of one table with the fields word_id, word, definition, length and special_char. The length of each word had been worked out using a Java program, which selected each word from the database, got its length and then updated the value of the length field of the word. The special_char value is whether or not the word contained any characters other than letters. The special_char values were also achieved by a program selecting the words and examining them to see if they contained any spaces or hyphens etc. Figure 3.13 shows an example of some of the entries that the table contained.

<table>
<thead>
<tr>
<th>word_id</th>
<th>word</th>
<th>definition</th>
<th>length</th>
<th>special_char</th>
</tr>
</thead>
<tbody>
<tr>
<td>99992</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>99993</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>99994</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>99995</td>
<td>Linch</td>
<td>A ledge; a right-angled protection</td>
<td>5</td>
<td>NULL</td>
</tr>
<tr>
<td>99996</td>
<td>Line</td>
<td>Flax; linen</td>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>99997</td>
<td>Line</td>
<td>The longer and finer fiber of flax.</td>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>99998</td>
<td>Line</td>
<td>To cover the inner surface of.</td>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>99999</td>
<td>Line</td>
<td>To put something in the inside of; to fill.</td>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>100000</td>
<td>Lineage</td>
<td>Progeny; race.</td>
<td>7</td>
<td>NULL</td>
</tr>
<tr>
<td>100001</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>100002</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>100003</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Figure 3.13: Example entries in the original database.

A dictionary database will naturally have more than one definition for many of the words it contains. For this reason the original database contained duplicated data, for example in the example above, the fact that the word line is of length 4 and contains no special characters is stored 4 times. Having duplicated data in a database is not always a bad thing, and is in some circumstances necessary, but if the duplication of data is unnecessary, the duplicated data is redundant. In the original table some of the data which is duplicated is redundant, and therefore should be excluded from the database.
3.4.2 Determining Redundancy of Data

The Boyce/Codd rule (see Wade, 2001) for determining redundancy can be used to illustrate that there is redundant data being stored in the old database (see fig. 3.14).

"Every determinant must be a candidate identifier."

Figure 3.14: The Boyce/Codd rule for Determining Redundancy of Data.

To make sense of this rule it is necessary first to define what a determinant is, and also what a candidate identifier is (see fig. 3.15).

Field A is a determinant of field B if each possible value of field A has exactly one possible value for field B.

A candidate identifier is a field that uniquely identifies a row in a table.

Figure 3.15: Definitions of a Determinant and a Candidate Identifier.

The Conceptual Model Rules of a database state which fields of the database are determinants of other fields (Wade, 2001). It is therefore necessary to have a list of such rules before we can be sure that some of the information in the database is redundant (see fig. 3.16).

For this database to comply to the Boyce/Codd rule, both word_id and word must be candidate identifiers. But we can see from the table above that word is not a candidate identifier, as when it has the value line it references four different rows in the table, and therefore does not uniquely identify a row in the table. Since the field word is a determinant and not a candidate identifier the database does not comply with the Boyce/Codd rule. Therefore redundant data is being stored in the database.

3.4.3 Eliminating Redundant Data

To deal with this redundancy I created a new table in which the non identifying determinant (word) is a candidate identifier. Therefore when designing the new database structure, I decided to store the necessary data in two tables as opposed to one single table. By doing this the length of each word and whether or not each word has a special character will only be stored only once, instead of being stored ten times if the word has ten definitions. The
• **word_id** is a determinant of **word**

• **word_id** is a determinant of **definition**

• **word_id** is a determinant of **length**

• **word_id** is a determinant of **special_char**

• **word** is a determinant of **length**

• **word** is a determinant of **special_char**

Figure 3.16: Conceptual Model Rules for Original Database. see also 3.15

Now by examining the conceptual model rules for the new dictionary database (see fig. 3.19), it can be determined whether there is any redundant information stored in the table.

The new database complies to the **Boyce/Codd rule** for determining redundancy (see fig. 3.14), as there are no determinants in our database which aren’t also candidate identifiers. **word** is a determinant of **length** and **special_char** and also a candidate identifier, as it uniquely determines a row in the word_data table. **definition_id** is a determinant of **definition** and is a candidate identifier, as it uniquely identifies a row in definition_data. Therefore it no longer contains any redundant data and is therefore in normalized form.
CHAPTER 3. SELECTION OF CLUES AND ANSWERS

<table>
<thead>
<tr>
<th>definitionID</th>
<th>word</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>155551</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>155552</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>155553</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>155554</td>
<td>Linch</td>
<td>A ledge; a right-angled protection</td>
</tr>
<tr>
<td>155555</td>
<td>Line</td>
<td>Flax; linen</td>
</tr>
<tr>
<td>155556</td>
<td>Line</td>
<td>The longer and finer fiber of flax.</td>
</tr>
<tr>
<td>155557</td>
<td>Line</td>
<td>To cover the inner surface of.</td>
</tr>
<tr>
<td>155558</td>
<td>Line</td>
<td>To put something in the inside of;</td>
</tr>
<tr>
<td>155559</td>
<td>Lineage</td>
<td>Progeny; race.</td>
</tr>
<tr>
<td>155560</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>155561</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>155562</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Figure 3.18: The definition_data table.

- **word** is a determinant of **length**
- **word** is a determinant of **special_char**
- **definition_id** is a determinant of **definition**

Figure 3.19: Conceptual Model Rules for New Database. (see also fig. 3.15)

### 3.4.4 Advantages of Normalized Databases

One of the advantages of having the tables of a database in normalized form is that there is no redundant data stored in normalized tables and therefore this keeps the quantity of data to a minimum. This may not seem like a huge advantage but we have approximately 150,000 words in our dictionary database. Some of the words have over 30 different definitions, so by normalizing the table I have reduced the amount of data stored significantly. This will improve the speed of certain parts of the program. For example if the program has to select words of length 4 from the word_data table, it will only have to look at one row for each word, where it could have been looking at up to 40 rows for each word. I considered this is an important factor, as keeping the execution time of the program to a minimum increases the enjoyment the user gets out of the system.

Having the tables of a database in normalized form also makes manipulating the data in the tables much easier. For example, when I’m inserting data about a definition I don’t have to insert the data about the word i.e. its length etc, or if I decided to delete a definition from the database, the details about the head-word won’t be deleted along with it.
3.4.5 The New English Dictionary Database

Since the old database was not in normalized form, I needed to create new tables and to populate them. One way I could have populated the tables would have been to use the data already contained in the old database to populate the new one. However a newer version of the dictionary source had been released since the project had been started. I thought using the newer version would be advantageous to the system, since the newer version of the dictionary seemed to contain many more definitions for each word, than the old dictionary database contained. It is important for the system to have a very large source of words and definitions to choose from, so that it can retrieve enough words that are related to the subject entered by the user. Therefore I used the more up to date version of the dictionary to populate the tables of the database, (Webster, 1999).

3.4.6 Populating the Database

The database needed to be populated with the words and definitions contained in the Project Gutenberg html file (see figure 3.20). I decided to write one Java program to populate the tables of the database. This program reads in the file using a Java FileReader and then buffers the contents of the file using Java’s BufferedReader. In this way it scans through each line of the file, and uses the html tags to recognize the beginning and end of the head-words and definitions. When a head-word is found, it is immediately inserted into the word\_data table along with the necessary information concerning it. The pieces of text that follow a head-word that are labelled as definitions are then inserted as the definitions this head-word. The advantage to entering the data in this way as oposed to using the load data infile statement (see 3.3.2) is that you have full control of what data you are inserting, and the method used to find the head-words and its definitions is very simple. There are however some disadvantages to writing a program to insert the data. Firstly there was a restriction on the size of the file that Java’s FileReader can read in at once. It seemed to only be able to cope with about 5000 lines of text. This meant that the file containing the dictionary had to be split into about 30 files. Also running the program took a long time. This was not a huge disadvantage however, because the program simply needed to be started and could be left running until it had finished. Running the program took about 20 hours in total.

It is worth mentioning another way of populating the tables that I considered. Other projects that use a dictionary seemed to all use the SQL statement shown in figure 3.8 to populate the tables of their databases. As I discussed above this statement may not be used by a program with MySql version 3.23.49, but it may be used when accessing the DBMS directly. This statement loads a text file very quickly into a table. It simply works by the user specifying what each column is terminated by, and then blindly loads the data in the text file into the table, one column after the next. A major advantage to using this command to populate a database is the amount of time it takes. For example for populating the database used for Cleare (2002) it took only 6.1 seconds to populate one of a table which contained thousands of entries. But there are also disadvantages to using the LOAD
CHAPTER 3. SELECTION OF CLUES AND ANSWERS

<entry><hw>Roar"er</hw> (?) <pos><i>n.</i></pos> <sn><b>1.</b></sn>
<def>One who, or that which, roars.</def> Specifically:
<sd><i>(a)</i></sd>
<def>A riotous fellow; a roaring boy.</def></p>
<p><blockquote>A lady to turn <i>roarer</i>, and break glasses. <i>Massinger.</i></blockquote></p>
<p><sd><i>(b)</i></sd> <i>(Far.)</i>
<def>A horse subject to roaring. See <u>Roaring</u>, 2.</def></p>
<p><sn><b>2.</b></sn> <i>(Zo&ouml;l.)</i>
<def>The barn owl.</def> [Prov. Eng.]</p>
</entry>

<entry><hw>Roar"ing</hw>, <pos><i>n.</i></pos> <sn><b>1.</b></sn>
<def>A loud, deep, prolonged sound, as of a large beast, or of a person in distress, anger, mirth, etc., or of a noisy congregation.</def></p>
<p><sn><b>2.</b></sn> <i>(Far.)</i>
<def>An affection of the windpipe of a horse, causing a loud, peculiar noise in breathing under exertion; the making of the noise so caused. See <u>Roar</u>, <pos><i>v. i.</i></pos>, 5.</def></p>
</entry>

<entry><hw>Roar"ing*ly</hw>, <pos><i>adv.</i></pos>
<def>In a roaring manner.</def></p>
</entry>

Figure 3.20: Excerpt from Webster's English Dictionary file.

DATA LOCAL INFILE statement. Firstly the file to be loaded must be prepared so that the appropriate delimiter (e.g. carriage return or comma) occurs between each column. Secondly if each entry requires a primary key or has a column that references an entry in another table, this complicates things a lot. All database tables require a primary key, to uniquely identify each row. Thus in the text file that is loaded into the table, the values of the primary key field must also be included. Also each value of the foreign key columns must be written into the file in the appropriate place. Figure 3.21 shows an example of the text files used in Cleare(2002) that were loaded into the database using LOAD DATA LOCAL INFILE. Thus using this SQL statement requires a lot of preparation on text files before the tables can be populated, which could be time consuming. Also if there is a small mistake anywhere in the text files you prepare (which is quite likely), a whole section of the
database entries could contain inappropriate data and checking that this hasn’t happened would also be very time consuming.

### 3.4.7 Randomly Jumbling The Words

Once the words had been retrieved from the database it was necessary that they be jumbled. Jumbling the words is important for a number of reasons. Since Prolog’s search mechanism is top down, the order of the words in the code will influence their likelihood of each word ending up in the solution. Although the words are evaluated and sorted by the prolog code, it often happens that a number of the words will be given the same value. The words are evaluated on how likely they are to cause a problem at later stages in filling up the grid. So if a number of words are equally as likely to cause a problem in solving the grid, then they are given the same value, and the word that occurs first in the database will always be used before the others. Therefore shuffling them is important so that the words that are given the same value have an even chance of making it into the solution see also 5.6.4.

However the words that were retrieved from the database were stored in an array. Many methods of shuffling arrays are problematic due to the amount of time they take to execute. A usual method of shuffling the items in an array would be to create a random number, select an element of the array based on this random number, insert this element into another array, and then reduce the size of the first array by removing the selected element. But removing a single element from an array in Java involves copying each element in the array that occurs before the element we are trying to remove to another array and then copying every element after the removed element to this array, skipping the element that we want to remove. This is ok to use when you are dealing with small arrays, however the arrays that we are concerned with are mostly very large. For example when ‘e’ is entered as the subject to the crossword, over 40,000 words will be selected from the database and will need to be jumbled. Therefore the usual method used to sort arrays was not suitable.

I came up with the following way of jumbling the words that would decrease the amount of time it took significantly. The array of words is converted into a Java List. Java Lists are like arrays in that they can be used to store objects. Lists can be sorted very quickly using the sort method of class Collections. The sort() method sorts the list of objects into ascending order, according to the natural order of its elements. To sort a List of elements, all elements in the List must be mutually comparable (that is, e1.compareTo(e2) must be possible) i.e. if the object is made up of Person objects, then class Person must contain a boolean method called compareTo(Person other), that takes another Person object as its only parameter and compares the current Person object with the other Person object. Therefore I created a class called WordData, which would simply function to store an individual word and to have a method to compare the current WordData object with another WordData object. In this way when a List of WordData objects were put into a List, they could then be sorted. But our aim here was not to sort the words but to jumble them randomly. Therefore I redefined the method used to compare two objects. Instead of them being sorted alphabetically, the would now be sorted based on the values of two randomly generated numbers. Each WordData object was assigned a random number and whether
the first WordData object should appear before the other in a sorted List was determined upon if the value of its random number was less than the other objects random number. The sorting algorithm used is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed \( n \log(n) \) performance, where \( n \) is the length of the list, and can approach linear performance on nearly sorted lists. This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the \( n^2 \log(n) \) performance that would result from attempting to sort a linked list in place (Sun-Microsystems, 1995).

3.5 Conclusion

This chapter described the database issues involved in the project. In it a detailed account of the DBMS used by the project is given as well as a description of what a DBMS is and why a driver is needed for a program to connect to a DBMS. In this chapter a detailed description of the table used in the project to store information about the locations of the various dictionaries available to the system was also given. Finally I gave a description of the English dictionary database that I created from which words and definitions were selected. From working on the parts of the project that involved databases I feel I have learnt a great deal about database design as well as SQL, the language used to query the database.
CHAPTER 3. SELECTION OF CLUES AND ANSWERS

26

Figure 3.21: Excerpt from file used to populate tables in Cleare(2002)

Gur

Mis

donkey.

Abair2

: ABAR. ~

Abairt

f.

(ge. ~e, pl. ~i)

Abairteach

41

= ABARTHA.

A’balta

43.

A’baltacht

f.

(gs. ~a)

Abar

m.

(gi. & npl. ~air, gpL ~)

Morass.

2.
Chapter 4

User Interface

4.1 Introduction

When I took on the project at the beginning of the year, it already had a web based interface. It had been decided that the crossword generator should be set up on a server and be accessible via the Internet, thus enabling a wide audience to access the program without having to download software or having to configure their machines in any way (see Gibbons, 2001). A Java applet was chosen as the web interface, as the majority of the project at that stage had been written in Java. Although I decided not to use the interface that had been written for the crossword generator, I decided to stick with an applet for the same reasons an applet had been previously chosen (see figure 4.3). In the following sections I describe Java Applets, the old user interface and the new user interface.

4.2 The Java Applet

A Java applet is a program that is written in Java and can be included in a html page. When a html page containing an applet is visited, the applet code is downloaded to the machine the page is being viewed on and is run using a Java Virtual Machine. Much of the information in the following section was gotten from Deitel (1997).

4.2.1 Sockets and Server Sockets

Sockets are used to transmit information between two hosts. The host that initiates the connection is called the client and the host that responds to the connection is called the server. A server socket is used to set up a socket between the server and a client. It binds itself to a particular port on the local machine and listens for incoming connection attempts. If an attempt to connect is detected, the server socket accepts it, resulting in a socket being set up between the client and the server.

Only one socket can listen to a particular port at a time. However servers usually need to be able to handle more than one connection at once, therefore server programs need to
be multi-threaded. The server sockets listening on a port accept the connections and the processing of connections is then passed off to a separate thread. If there is more than one incoming connection at a time, then they are stored in a queue.

4.3 The Old Interface

In order to illustrate that creating a new user interface from scratch was indeed necessary I will point out the faults I saw with the old interface. The only reason to do this is to show my motivation for creating a new interface. I do not wish in any way to disregard any of the work that was put into this project before I took it on as it served as a useful starting point from which to advance.

The applet that was initially used to interface the project was automatically generated by Visual Cafe. Visual Cafe for Java is an integrated development environment, which can be used to create Java programs. It allows you to create a user interface using drag-and-drop, and writes the code for you. Writing an applet to display a crossword is quite complicated, and this along with the amount of work that was involved in the project before I took it on were perhaps the reasons that an application was used to generate the code. The code that was generated produced an applet that didn’t look very much like a crossword and it was not very user friendly either. Figure 4.1 shows a screen shot of the old user interface.

4.3.1 The Clues

As you can see from the screen shot above the clues of the crossword were not number in the traditional way that crossword clues are usually numbered. This resulted in an un-intuitive labeling of the clues. This was achieved by the rows and columns of the crossword grid being numbered and then each clue being given a direction, a row and a column. The row and column referred to the the starting cell of the clue in question. Although this part of the interface didn’t make it impossible for the user to solve a crossword using it, it in general got a bad reaction from those who attempted to use it.

4.3.2 The Grid

Another feature of the interface which made the crossword generator difficult to use was the grid itself. The grid was made up of a number of text fields A square-shaped text field was used to represent each open cell in the grid. Java Text Fields are designed for taking in text input strings from the user. An example of how texts fields are normally used is for inputting the subject of the crossword. But because the text fields used to create the grid had to be square shaped and only allow one character to be typed within them, the result was that when a letter was typed in one of the fields, that it was not completely visible. This resulted in the user having to remember to some extent what the answer to the clues were, instead of them being clearly displayed. The same problem occurred with the text
field used to take in the subject. The dimensions of this input field were also set to small values, resulting in a poor display of typed input.

4.3.3 Check Solutions Function

Another unappealing feature of the interface was the check solutions function. When the appropriate button was pressed, this function would be carried out, resulting in the incorrect letters that had been entered in the grid being simply erased. I considered this an insufficient way of relaying to the user that an incorrect letter had been entered, due to the fact that the letters that had been entered were barely visible in the first place, and also the user may not want to simply erase every letter they’ve entered, they may wish to look over the incorrect words before they are erased from existence.

4.3.4 The Code

The last point I wish to make about the old crossword applet is that the code that was used to produce it was very ugly and unreadable code. A number of things that could have been done in a couple of lines in an array were instead carried out by pages and pages of code. This was due to the code being automatically generated. Figure 4.2 shows an example of some code that was contained in the CrossWordAppletF class, which creates 110 text field objects.

The following code achieves exactly the same effect as the code used in the old applet, but is done in a couple of lines. This is how such code is normally written. An array of Text Fields with 110 elements is created and a loop is used to initialize each of these objects:

The fact that the code was written in such a manner and that the classes used to create the grid etc. were in my opinion not appropriate, I decided to scrap the old interface and start a new one from scratch.

4.4 The New Interface

As I mentioned above displaying a crossword puzzle in an applet is quite complicated, but displaying such a puzzle can be made much simpler by taking full advantage of Java. Due to the fact that crossword puzzles in general are all structured in a very similar way, it is easy to make many generalities about them. For example all crosswords (that we wish to represent anyway) have a width, and contain a width*width number of squares etc. When such generalities are easily found about the objects that we are trying to represent, Java can be very useful for defining structures to represent these objects. Therefore Java was very appropriate to use to represent the different parts that make up a crossword. Once the necessary structures are defined, it makes displaying and interacting with a crossword much simpler. Therefore in order to represent a crossword grid using Java I decided that first I would need some data structures or classes to represent the various components that
make up a crossword puzzle. There were three different data structures that seemed would be useful to have; the Square structure, to represent the individual cells or squares of the grid, the Solution structure, to represent the word slots and the information needed about each word slot, and the Crossword structure, used to store all the information about a crossword.

4.4.1 The Squares

First I will describe the data structure I designed to represent the squares or cells of the grid. Since it was necessary for each square of the grid to be displayed, each square structure should be able to hold whether it is open or closed (black or white). Also due to the fact that each white square has a solution letter and that the user may enter a value into any white square, the squares should also be able to store a character which they contain, further allowing these values to be changed by the user. The final thing that a square needs to store is its own location in the grid. This was achieved by each square having row and column.

4.4.2 The Solutions

It was advantageous to store all the information needed about each word slot (see figure 5.9). Each Solution needed to store the following information about each word slot in the crossword:

- its clue
- its direction of the word slot (across / down)
- its number (of it’s clue)
- its first square (represented by a Square object)\(^1\)
- its answer (i.e. the word that will fill up the slot)
- its length (how many squares of the grid it traverses)

Here we can see one of the many advantages to object-oriented programming. Since we have already defined what information is needed to represent a Square in the grid, we can simply without a second thought, include a Square object as part of the information needed about a Solution (first square of Solution). This in turn will hold information that will be needed by the Solution, for example the first square object of a Solution will hold the position of the Solution in the grid. This means that when we are defining what a Solution contains, because we’ve already defined what information we need about each square, by including this square object in the solution, we don’t need to worry about any of the information about the first square of the solution when defining what a solution is. This is a big advantage of object-oriented programming.

\(^1\)The location of the first square of word slot is needed in order to number the clues.
4.4.3 The Crosswords

Finally a structure was needed to store all the information needed about the crossword itself. The following information needed to be stored:

- the width of the crossword
- the number of word slots it contained
- a number of Squares\(^2\)
- a number of Solutions\(^3\)

The most important of these pieces of information are the Squares and the Solutions. Because we previously defined what information a square holds and what information a solution holds, it is very easy to define a crossword as a width*width number of squares and a number of Solutions.

4.4.4 Use of Data Structures

The applet uses two Crossword objects as described above. Both of these crossword objects represent the grid in question. The difference between them is that one represents the crossword in a solved state (solution_puzzle), and the other represents the current state of the crossword on the screen (user_puzzle). The solution crossword never changes. The user_puzzle starts its life with the contents of its squares set to empty, and whenever the user enters a letter in one of its squares on the screen this letter is added in the appropriate position to the user_puzzle. Likewise if a letter is deleted by the user, then the appropriate letter shall be deleted from the user_puzzle. The applet also uses a Solution object to represent the current solution that the user is attempting to solve. Finally the applet has a current row and column which represent the current position on the grid (a Square object could also have been used for this, but would have caused unnecessary complication).

4.4.5 Drawing the Grid

The drawing of the grid is done in the applets paint method. Each time an applet is loaded, its paint method gets called to draw what is desired to appear on the screen. It is best to draw everything you will want to draw at any stage in the applet in the paint method. This is important because the applet gets repainted repeatedly. For example if the user simply clicks on the applet it gets repainted, or if a pop-up box appears and then disappears the applet gets repainted again. Therefore it is necessary to draw anything that we want to appear on the screen for a significant length of time in the paint method. However we do not want everything to be drawn every time the paint method is carried out, therefore the

\(^2\)Objects of class Square described above
\(^3\)Objects of class Solution described above
things we wish to draw are contained in if statements and whether it is drawn is decided each time paint is carried out by the value of the contents of the if condition. For example the paint method of my applet is first called before a crossword has been generated, but no crossword will be drawn until the value of the boolean variable createButtonPushed has been set to true, which is done after the crossword has been created (see fig. 4.4).

This may not seem too important, but it took me a while to realize that you come across countless problems with drawing things outside of the paint method. Although I looked at a lot of sources about applets none of them mentioned this, so I considered it worthwhile mentioning.

More importantly though is what the paint method draws. It firstly draws an empty grid with the squares colored appropriately. Then it fills in the letters that the user currently has entered in the grid (they will be all clear when the crossword has just been created). It does this simply by drawing the letters that are being stored in the user_puzzle Crossword object. Each time a key is pressed, this causes the contents of the square in the user puzzle to change that is located at the current row and column in the puzzle, and the applet is repainted using the repaint() method. This happens very quickly, so that it appears as if pressing a key automatically types the appropriate character into the highlighted square.

4.4.6 Numbering the Clues

The clues of the crossword are numbered when a crossword object is created. It is quite simply done using the method isFirstSquare() that can be carried out on any Square object. As the name suggests, when this method is called on a Square object, it returns true if it is the initial Square of a Solution, and false if it is not. So to number the Squares of the grid, I simply loop through each square of the grid, giving it a number if it is a first square of a solution, and skipping it if it is not. As I number each square, I also number the appropriate clue(s).

4.4.7 Moving Around the Grid

Once the crossword has been created, the current row and column are set to the row and column of the first white square in the grid starting at the upper left hand corner and moving right. Since this square is highlighted, the user knows that if he or she enters presses a key, that the letter will be entered in the highlighted square. The user may move around the grid using the arrow keys. When an arrow key is pressed, the value of the current row or column may change. For example if the user presses the up arrow key and the cell in the grid directly above the current square is white, then one will be taken away from the current row and the applet will be repainted, thus displaying the effect of moving up to the previous row. The same method is used for moving down, left and right.
4.4.8 Checking Validity of Letters input by the User

From being an avid user of online crosswords I knew that an important feature of the interface would be the function of being able to check if the answers you’d gotten were indeed correct. I personally have found this feature to be very useful, especially when attempting to solve difficult crossword puzzles. From looking at other interfaces to crosswords, I realized that there are a few different approaches to how the user is able to check whether or not the answers he or she has entered are correct. The different methods that I came across were the following:

- Highlight the incorrect letters (Irish Times Online Crossword)
- Reveal the solution (The Chambers Online Crossword)

Due to the way I had designed the structures used to represent the crossword, it was very easy to include both of these functions.

Highlighting Incorrect Letters

The user can highlight the incorrect letters they have entered by clicking on the button labeled Check Answers. The identity of the incorrect letters is achieved by comparing the user_puzzle to the solution_puzzle. The letters in the user_puzzle are drawn one by one, but the color that each letter is drawn in is determined by whether or not the same square in the solution_puzzle contains the same letter. If the letter is the same it’s drawn in black, if they differ the letter is drawn in red.

Revealing the Solution

To reveal the solution is also quite simple. The user has the choice of either revealing the current solution (highlighted squares) or to reveal all of the answers in the puzzle in one go. A Solution is revealed by that Solution in the user_puzzle being given the values of the Solution in the solution_puzzle, and the grid being repainted. If the user chooses to reveal all the Solutions, then all of the squares of the user_puzzle that are not the same as those of the solution_puzzle are given assigned the squares of the solution_puzzle, and the user_puzzle is repainted.

What if all the answers are correct?

If all the answers that the user has entered are correct and the user presses the Check Solution button, then user is informed that all the answers are correct. If the user has filled in all of the answers and they are all correct then they are congratulated on their achievement and informed of how many incorrect letters they entered during their game. The number of incorrect letters is incremented for every incorrect letter entered every time that the check answers function is used. It is also incremented if a word is revealed. For each blank or incorrect letter in the word to be revealed, one is added to the number of
incorrect letters entered. But this number will never be incremented when a correct letter had been entered in a square, even if reveal word or reveal all is used.

4.5 Conclusion

In this section I described all of the user interface issues concerning the project. This part of the project was very enjoyable to do, because it involved being able to create a visual representation of the crosswords that were created. Having the old applet to start with helped here as it gave a point to start from and also let me display the crosswords before I had created the new applet, so that I didn’t have to attempt this straight off. The resulting user interface is a user-friendly piece of software that can be used to solve the created crossword puzzles with great ease and enjoyment.
Figure 4.1: Old User Interface
import java.awt.*;

TextField[] s = new TextField[110];

for(int i=0; i<s.length; i++){
    s[i] = new TextField();
}

Figure 4.2: Example Code from Old Applet

Figure 4.3: Example of how code in fig. 4.2 should be written
public void paint(Graphics g){
    this.setBackground(PURPLE);
    if(createButtonPushed == true){
        drawGrid();
    }
}

Figure 4.4: Example paint() method.
15 ACROSS: A small West Indian tree (Trophis Americana) of the Mulberry family, whose leaves and twigs are used as fodder for cattle. (6)
Chapter 5

Prolog Solutions to Solving the Grid

5.1 Introduction

In the following chapter I discuss a variety of different approaches to solving crosswords. Firstly I describe the approach used by Berghel (1987), which uses a succession of horn clauses to solve the crossword. Next I discuss the crossword compiler compiler described in (Berghel & YI, 1989), which can be used to create code which can be compiled and used to get solutions to the puzzle. I then describe the approach used in (Gibbons, 2001) i.e. the prolog part of this project when I took it on. Finally I give a detailed account of the prolog code I wrote that is currently used by the project to find solutions to the crossword grid.

5.2 Crossword Compilation with Horn Clauses

Berghel (1987) describes a method of crossword compilation that uses horn clauses. The aim of Berghel (1987) was to create a way of constructing one or more solution sets to a predefined crossword grid. The problem was first analyzed and expressed in first-order predicate logic. This analysis was then used to construct a crossword compiler using Prolog.

5.2.1 The Logic behind the Program

Firstly Berghel broke down the problem of solving the two-dimensional puzzle into the idea of two interlinking words. Supposing the words were of length $m$ and $n$, respectively, and that the intercepting cell is cell $i$ in the first word, and cell $j$ in the second word, then the problem is as follows: do two words exist in the dictionary which share the same character in position $i$ in one word and position $j$ in the other word?

Berghel the generalizes the problem for the whole puzzle. Assume there are $m$ across words and $n$ down words. Each word is named using a the name predicate, so that every word can be represented in the following form:

- name(slot.k,[X^1,X^2, ...,X^l])
This naming of cells has the effect of giving cells $X^1$ to $X^i$ the name slot$_k$. Also for each word slots that intersects slot$_k$ a new existentially quantified variable $Z^j$ will be substituted at the intersecting cell. Thus the problem of solving the crossword has been converted from a two-dimensional puzzle into a linear list of predicative expressions.

### 5.2.2 The Prolog Program

The program is written in Prolog, because it has a built-in theorem prover which operates on sets of Horn clauses. The program determines whether a solution to the crossword exists by investigating if the expressions I’ve described above are consistent with a set of lexical axioms. The lexical axioms are in the form of predicative expressions whose arguments will be character constants:

- $\text{word}(c^1, c^2, ..., c^n)$.

Besides the lexicon, the program consists of a rule called $\text{solution}$ which takes a list as its only argument. This list contains variables which represent the various word slots in the grid. The body of the rule then consists of pairs of predicates, one for each word slot. As you can see from figure /refhornclauses the purpose of the first part of each of these pairs of predicates is to retrieve a word form the lexicon that matches to the length of the word slot and possibly to any of its interlinking words which have been previously instantiated. The second predicate is used then to give the appropriate name to the word slot, so that it may be returned in the correct position in the list argument of the rule $\text{solution}$.

Each cell is identified by its row and column coordinates in the grid. For example $C2_1$ is the cell that lies in the second row and the first column. One of the main reasons this rule will find a valid solution is because of the fact that each occurrence of a given variable is bound by the same quantifier and therefore multiple occurrences of the same variable within this rule, will be instantiated to the same character. This ensures that the shared cells in the grid will be instantiated to the same character.

### 5.2.3 Results

If Berghel’s program is given an adequate dictionary and enough time, it can be used to generate all possible solutions to any given crossword grid. The program was tested using a $12 \times 12$ grid which had appeared in a paper written by Smith and Steen. The program was terminated after 87 hours and had found 1286 different solutions for the grid. The tests were run on a micro-computer and when tested on this $12 \times 12$ grid generated a solution on average approximately every four minutes (see Berghel, 1987). When considering these results it must be noted that there was no uniqueness check done on the words entered into the grid, so many of the solutions that were found would have contained multiple occurrences of the same words (see 5.6.4). Since no results are mentioned in the paper of how many of the crosswords that were returned did not contain duplications of words we can not really be sure how successful it was in creating valid crosswords. For example
if every fifth solution that was returned did not contain duplications of the same word, then the search time would be 20 minutes as opposed to four minutes, but we can only speculate.

5.2.4 The Search Space

One of the major problems with solving a crossword is the time it takes to do so. The reason it takes such a great deal of time is due to the size of the search space. Berghel explains the possible enormity of the search space of the program as follows:
Suppose that one uses a simple generate and test approach to compile a full \( n \times n \) crossword with complete orthogonal interlocking. If \( k \) is the size of the alphabet from which the words are drawn, this strategy would create \( k^{n^2} \) possible solutions, each one of which would require \( 2^n \) tests. If we restrict ourselves to the Roman alphabet, a simple \( 4 \times 4 \) puzzle would have \( 10^{22} \) solutions.

Figure 5.2: Berghel’s Explanation of the Potential Size of the Search Space

### 5.2.5 Strategies for Reducing Search Space

Before Berghel (1987) thought up this way of solving crosswords, there seemed to be two other approaches that you could take to solving crosswords. One was the letter by letter approach and the other was the whole word approach. Both of the methods had their own advantages. The advantage of the whole word approach was that each letter that is inserted would be guaranteed to be part of a word in at least one direction. But the disadvantage of this method was that word slots that occurred parallel in the grid could result in impossible combinations of letters occurring in word slots of the other direction. But the letter by letter approach had the advantage of ensuring that illegal letter combinations in both directions were avoided, but at the same time there is no guarantee that the letters entered would ever turn into valid words. Berghel’s approach is a combination of the letter by letter approach and the whole word approach. He says that it is whole word, because whole words are inserted at once, as opposed to individual letters, and that it is letter by letter because the insertion of a word is constrained to whether the substrings they impose on the intersecting word slots are legitimate.

### 5.2.6 Evaluation

Berghel’s method of crossword compilation is a very straightforward approach that takes advantage of Prolog’s built-in theorem prover. One problem with Berghel’s crossword compiler was that many of the solutions it produced contained multiple occurrences of the same words. He suggests that duplications of words could be avoided by collecting the chosen words in a set, and then only considering a word as a candidate insertion if it is in the dictionary but not in that set. Although Berghel suggests this as a strategy, he has not included it in his implementation at all. Therefore when considering Berghel’s results, it is important to remember the fact that many of the crosswords his program generates will be invalid crossword puzzles, since they will contain multiple occurrences of the same word.

Berghel also suggests a method of avoiding naive backtracking. He suggests that immediately after a word has been inserted into a particular word slot, that there should be a check done on each of the word slots which interlink the word slot in question. This should check that by inserting the word in that word slot that no problems will be caused like for example the letter \( x \) being inserted in cell \( C4_1 \) if there are no three letter words beginning
with x in the dictionary. He says that by including these checks, that backtracking will start at an earlier point, which will reduce the search times. The cells of the grid that have not yet been given a letter are left as underscores. To illustrate more clearly how he means to do this, figure 5.3 shows an example of how the first part of the code would look for the same grid as coded above.

```prolog
solution([SLOT_1d, SLOT_2d, SLOT_3d, SLOT_4d, SLOT_1a, SLOT_14a, SLOT_17a, SLOT_20a, SLOT_23a]):-
    word(C1_1, C2_1, C3_1, C4_1, C5_1),
    word(C1_1, _, _, _),
    word(C2_1, _, _, _),
    word(C3_1, _, _, _),
    word(C4_1, _, _),
    word(C5_1, _, _, _, _, _),
    name(SLOT_1d, [C1_1, C2_1, C3_1, C4_1, C5_1]),
    ...  
```

Figure 5.3: Berghel’s Method of Avoiding Naive Backtracking

This method does indeed cause the algorithm to disallow words that are going to cause a problem at later stages in the grid. There is one disadvantage to it though and that is that writing out a solution rule like this for one crossword is very time consuming, as when including these check word predicates, each cell has to be checked to see whether it would have been instantiated or not yet in order to know whether or not should be left anonymous. This could be avoided by writing a program to do figure out what the solution rule should contain, but it would also be a complicated algorithm.

Berghel also comments on the importance of the order in which the word slots of the grid are assigned words. He says that the order of the word slots can be arranged to increase efficiency. He suggests that the length words of lowest frequency in the dictionary should be filled in first. He also suggests that the areas of high interlink in the puzzle should also be filled in first. Although Berghel suggests that the order in which the word slots are given words is important, he does not give any specific way for a program to work out what the best order is. He does this in his paper Crossword Compiler Compilation, which was written three years later.
5.3 Crossword Compiler-Compilation

Berghel and Yi (1989) describes an automated way of creating crossword compilers, like the one described in the previous section. As I mentioned above, simply writing out a rule like the one in figure 5.3 is very time consuming. If we also wish to work out the order that the word slots should be assigned words based on some sort of algorithm, this would complicate things even further and be even more time consuming. Thus Berghel and Yi (1989) saw the need for a crossword compiler compiler, which takes in the size of the host matrix and its internal design as its only input. This compiler compiler then creates a rule such as the one in the last section. The produce rule can then be compiled and used to generate solutions to a crossword. The crossword compiler compiler involves the same method used above for avoiding naive backtracking, by checking for invalid combinations of letters in the interlinking word slots. In addition to this the compiler also decides the best order to fill in the word slots.

5.3.1 The Strategy

The order in which the word slots are filled in will determine at what point impossible combinations will be ruled out. The point where impossible combinations is discovered should be as early as possible to avoid naive backtracking. As an example of how the order is important, see the grid shown in figure 5.4. The grid could be filled in as follows: 1D, 2D, 3D, 1A, 2A, 3A. Filling in the word slots like this would result in no fail point occurring until the fourth word were inserted. Alternatively if the grid was filled in in the order: 1D, 1A, 2D, 2A, 3D, 3A, the first fail point would be at the insertion of the second word. The second method of filling in the words is obviously better as it forces backtracking at an earlier point. However Berghel and C. Yi dismiss the idea of working out the best path through the grid based upon which word slots interlink each other, because they say that the algorithm is not straight-forward enough when the open-cells do not fall within a rectangle. They then opt for a completely different strategy of deciding which word slots to fill in first. This strategy is based on the fact that in a normal dictionary, the frequency of words of certain lengths seems to be much less than the frequency of other word lengths. It is suggested that for this reason the length words of least frequency in the lexicon, should be filled in first. Thus each word slot is given a value based on the frequency of words of that length in the lexicon, and the best route through the grid is worked out by adding up the values of the word slots occurring down a given path. By this they mean that if you start with a certain word slot then each of its possible routes is evaluated to an N degree, where N is the depth to which the values of the linking word slots are added. The user can define the level of evaluation, i.e. the value of N.

5.3.2 Evaluation

In the paper it stated that there had been very little previous work on automatic creation of crossword compilers. As I mentioned above writing out a rule to solve a crossword grid
is time consuming especially when we wish to constrain the order in which the word slots of the grid are filled in. Therefore having a crossword compiler is very useful, so that writing such rules can be completely avoided. The structure of the grid can simply be given as input and this compiler will create the code that can be used to get solutions to the grid. Also having the compiler decide which order to fill in the word slots of the grid is an advantage. However I do not agree with their strategy of filling in the word slots in an order based on the word length frequencies in the lexicon. I think their first idea about filling the words in an order which forces backtracking at the earliest time possible would have been a better solution. Figuring out the best order in this way is difficult to implement, but an attempt at this would in my opinion probably be more effective than going by the frequencies of the word lengths. In my experience filling in word slots of less frequent length at a late stage did not cause a problem.

5.4 The Inherited Prolog Solution

5.5 The Old Prolog Code

5.5.1 How it worked

The method used by the original Prolog code was very similar to the one described in Berghel (1987). It used a rule called solve which was made up of a series of horn clauses which were used to retrieve words from a Prolog database of words and then assigned a word slot name to each of the chosen words. The words were encoded using the ASCII values of the characters they were made up of (see figure 5.5).

Each cell of the grid was represented by a uniquely named variable. This ensured that
each cell of the grid would only be allocated one character, thus validly interlinking two words. The rule used to get the solutions to the grid consisted of a series of the horn clauses use to retrieve the words. These horn clauses represented the word slots in the grid. Each word slot is made up of the variables representing each of the cells it occupies in the grid. Each word slot is also given a a uniquely named variable. The predicate name/2 is used to assign these variable names to the lists of cells that are contained in the appropriate word slot, which by this stage have been instantiated to the ASCII values of characters. Finally the predicate unique/2 is used to check that all of the words that have been assigned to the word slots are unique. The complete Prolog rule used to solve the grid is shown in figure 5.6. The code shown here differs from Berghel’s slightly (see Berghel, (1987)) because the calls to the name/2 predicate are done after all the word slots have been assigned a word. This is an improvement on Berghel’s method because backtracking is forced at an earlier stage.

### 5.5.2 What was wrong with it

The main thing that was wrong with the prolog code was that it didn’t return a solution to the grid within a reasonable time for crossword grids that were any larger than 4 x 4. This was partly due to the predicate unique/2 which checked that there were no multiple occurrences of the same word in the grid. This method was called once after all of the word slots had been given a word. This resulted in the crossword finding many solutions to the grid that contained duplicated words, before it would eventually find a crossword whose words were unique. Something that added to this problem was that there were many multiple occurrences of the same word in the lexicon. If a word had 5 definitions in the dictionary database, then there would be five occurrences of it in the Prolog database of words.

Another thing to note is that even without this uniqueness check the code did not find a solution in a very efficient way. Although by using Berghel’s method of horn clauses reduces the search space (see fig. 5.2) to some degree, it needs to be reduced further so that it could return a solution within a reasonable amount of time.
5.6 The New Approach

The code I use to generate the solutions to the crosswords is in some ways similar to the crossword compiler compiler described in Berghel and Yi (1989). The only input needed is the structure of the grid and a lexicon of words. However it differs to the crossword compiler compiler in that it simply outputs the first found solution to the grid, instead of outputting code that must be first compiled and then queried to return a solution to the grid. Like Berghel and C. Yi’s crossword compiler compiler, my crossword solver decides the order in which the word slots shall be filled. In addition to this though I have placed a number of constraints on the code, so that backtracking is caused at the most sensible points, avoiding naive backtracking. Another interesting aspect of the solution generation is that before a word slot is assigned a word, the code will figure out which of the possible words available to fill into the word slot will be least likely to cause a problem at further stages in filling up the grid. The combination of the constraints I put in place on the code resulted in the code filling up the crossword in the most sensible way it could.

5.6.1 The Input

The crossword solver takes simply the lexicon of words and the structure of the grid as input. Each word is represented once by a Prolog fact (see figure 5.7).

The structure of the grid is also input as a Prolog fact. Its only argument is a list whose elements represent each of the word slots in the grid. Each word slot is given a unique number to identify it. Each cell in the grid has been allocated a single unique variable, so that each cell shall be instantiated to a single character in each solution of the grid. Each word slot is then represented by a two element long list. The first element is the number used to identify the word slot and the second element is another list, whose elements are the cells which the word slot occupies in the grid. This is illustrated more clearly in figures 5.8 and 5.9.

5.6.2 Naive Backtracking

Naive backtracking was one of the main reasons that the program took so long to find a solution. I shall explain how this was a problem using the example code shown in figure 5.10. In the example below the words 'treated', 'start', 'donor', 'hem' and 'an' are filled into word slots 1, 2, 3, 4 and 5. When word slot 6 is attempted to be filled, it fails because there are no words in the given lexicon of length 7 that begin with 'sd'. When it fails on word slot 6, the program then redoes word slot 5 with another word of length 5 and again tries to fill up word slot 6 but fails again. Because filling up word slot 5 has no effect on word slot 6, redoing word slot 5 and trying word slot 6 again is pointless. Nevertheless this is how the program works and word slot 5 is filled up with every possible word of length 2 in the lexicon. When it fails for the last time on word slot 5, i.e. when it’s tried every word of length two, it goes back and redoes word slot 4. When it goes back and redo word
slot 4, it does the same thing again. Word slot 4 has no effect on word slot 6, so it redoes word slot 4, then word slot 5, then fails again on 6. It then goes back and redoes 5 again, then tries 6 again and fails, and retries every possible word of length two again in 5 before it goes back and redoes 4. It repeats this the number of times there are words of length 3 in the lexicon, because it will redo word slot 4 that number of times, each time redoing word slot 5 as many times as it can. This illustrates, I hope, how naive backtracking was causing the program to take so much time.

5.6.3 Avoiding Naive Backtracking

There were three main ways in which I got the program to avoid naive backtracking. I will discuss in the following sections how the two methods work.

5.6.4 Ordering the Word Slots

The order in which the word slots are filled is important. As an example take the grid structure described in figures 5.9 and 5.8. Consider if the grid’s word slots were filled in in the following order \([0,2,3,6,1,4,5]\). If the words are filled in in this order the first point of failure will be when word 1 is assigned a word. It is important that if a word has been entered into a particular word slot, and this word will eventually cause a problem in the grid, that the code will fail as soon after this insertion as possible. Therefore if we insert the words in an order which delays the first possible point of failure until the fifth inserted word slot, then this will slow down finding the solution significantly. Thus an algorithm needed to be defined, which would figure out the order in which the word slots should be filled up so that the number of early points of failure was maximized. Berghel and YI (1989) had considered this too difficult to implement on white cells that were not in a rectangle, so they opted for an ordering of word slots based on the frequency of words in the lexicon. As I mentioned previously, I found that ordering the word slots so as to maximize early points of failure was more successful. Therefore I decided on a simple algorithm to do this. Firstly the program finds a starting word slot in the grid. It bases where it should begin on the length of the word slots. It chooses the longest word slot in the grid, since it is likely to be the word slot that interlinks most other word slots e.g. word slot 1 above. The next word slots it will fill are those that interlink this word slot e.g. word slots 3, 4, 5 and 6 above. The next word slots to be filled will be those that interlink the second word slot in the sequence so far i.e the word slots that interlink word slot 3. Added to the end of the sequence then are all the word slots that interlink word slot 4, then 5 etc. The sequence of word slots is traversed adding all of the interlinking word slots to the end of the sequence as we go until all of the word slots of the grid are in the list. Thus the order of the grid in figures 5.8 and 5.9 would be: \([1,3,4,5,6,0,2]\). If the word slots are filled in in this order the first point of failure will be when the second word slot is assigned a word. Also it is worth mentioning that if there are islands of word slots in the grid that do not interlink each other, the algorithm will start each new island as a new crossword and start with the longest word slot, then its linking word slots etc. This algorithm will ensure that at least
one point of failure will occur on the filling up of a word slot, if the word slot interlinks any other word slots. If it doesn’t interlink any other word slots when it is filled up it is of little importance, because it will not cause any problems in the grid.

Evaluating the Words

Firstly I thought if the program could avoid filling up word slots with words that would cause a problem later on in the grid, this would decrease the amount of backtracking it would have to do. To do this, before each word slot is given a word, a check is done on whether the word will cause a problem when it comes to assigning words to its interlinking word slots. I extended this idea further. I decided as well as excluding words that will cause a problem further on to evaluate each of the possible words that could be entered into each word slot based on how likely the words are to cause a problem later on in the grid, then I insert the word into the word slot that is least likely to cause a problem. Therefore before a word slot is given a word, all of the possible words are put into a list. Each word is given such a value, and the words that will definitely cause an impossible situation are removed from the list as well as words that are already in the solution. The value that a word is given is based on the how unlikely it is to cause a problem. The higher the value, the less likely the word is to be problematic. The values of the word slots are calculated just before a word slot is given its word, because previous word slots that affect the current word slot may have already been given words. If some of the cells of a word slots have already been given a character, this is taken into account when evaluating it. To illustrate how the value of a word in relation to a particular word slot is evaluated I will use the example grid shown in figures 5.8 and 5.9. Say for example that word slot 0 had already been filled in with 'cat' and word slot 2 had also been filled in by 'dog', and all the other words have not yet been instantiated. Then the value of the word ‘atlas’ for word slot 1 is calculated by pretending that word slot 1 has been instantiated to 'atlas' and then adding up the number of words that fit into word slot 1’s interlinking word slots. Word slots 3, 4, 5 and 6 interlink word slot 1. The value that is given to ‘atlas’ for word slot 1 is the value of the lowest scoring interlinking word slot. Using this example again, if there were 20 words of length 3 in the dictionary that began with ‘a’ then word slot 3 is given the value 20. If there are 2 words of length three in the dictionary, that begin with the letter ‘a’ and end in the letter ‘t’ then word slot 4 is given the value 2. For word slot 5, if there are are say 3 words of length 3 in the lexicon that begin with ‘a’ and end with ‘o’, then word slot 5 is given the value 3. Finally if there are 7 words of length 3 in the dictionary that end in the letter ‘s’, then word slot 6 gets the value 7. Then the word ‘atlas’ shall be given the value 2 for word slot 1, since this was the lowest score of the interlinking words. In this way the words that will enable the possible number of words to be as high as possible, that can be inserted into the interlinking word slots. Ensuring that the best words are inserted will lead to the puzzle being solved more quickly.

Figure 5.11 shows the Prolog code used to evaluate the words. This predicate takes in a list of words to be evaluated as its first argument, e.g. [[g,a,p],[c,a,t],[b,a,t],[t,a,r],[f,a,d]], the wordslot that we are trying to fill as its second argument, e.g [S1,a,S3] (S2 having been
already instantiated by another word slot, a list of the word slots that link this word slot as its third argument e.g. \([0, [S1, S4]], [1, [a, S5, S8]], \), and its final argument is a variable used to return a list of pairs, each pair containing a word and its value. Because I don’t want any variables to actually be instantiated I copy the word slot and its linking word slots to new variables which can then be instantiated if necessary without impacting on the solution to the puzzle. Then the first word is evaluated using the predicate eval/5. eval/5 takes the word we want to evaluate as its first argument e.g. \([g, a, p]\), the copied word slot is given as its second argument e.g. \([C1, a, C3]\), the copied linking word slots are given as the third argument e.g. \([[C1, C4], [a, S5, S8]]\), the fourth argument is the highest value of the linking word slots so far. This is set to 1000000, when eval/5 is first called, so that when the first linking word slot is evaluated its value will be the lowest so far. The next argument is a variable which shall be instantiated to the value of the word \((g, a, p)\) for that word slot \((C1, a, C3)\). eval/5 firstly instantiates the variables of the word slot to the characters of the word we are evaluating e.g. \(C1 = 'g', \ C3 = 'p'\). Then it is checked if instantiating these variables causes a problem within the grid. This is checked by seeing if there is at least one word in the database that can be assigned to the linking word slots, when \(C1\) and \(C3\) are instantiated to ‘g’ and ‘p’. Next it gets all of the words that can be assigned to the first linking word slot of \([C1, a, C3]\), which is in this case the word slot \([C1, C4]\). Since \(C1\) has been assigned ‘g’, it gets all the words of length 2 in the prolog database that begin with ‘g’. The possible words are returned as a list e.g. \([[go], [g, a]]\). The length of this list is then gotten using length/2. Then if the length of this list is less than the value that had been passed in (at the beginning this is 1000000), then this is the new value, since we only care about the value of the linking word slot that is the lowest. The rest of the linking word slots are evaluated in the same way and the lowest value is retained each time. eval/5 therefore returns the value of the lowest scoring linking word slot. If this value is 0, then the word we are evaluating e.g. \([g, a, p]\) will not be contained in the list of evaluated words returned to the user. Also if the word causes a problem later on and therefore fails when pass2/1 is called in eval, then it is not included in the returned list either.

**Backtracking**

In order to optimize the search times for a solution to the grid, I needed to constrain the prolog code in such a way as to cause it to backtrack in the most sensible way it could. As mentioned above, when the program fills up a certain number of word slots it usually reaches a word slot that it cannot fill and then begins to backtrack. For convenience sake I shall call this failing word slot the *problem word slot*. If the word slot preceding the problem word slot does not affect the problem word slot, this results in the program carrying out an exhaustive and moreover pointless search as it fills in a word slot that does not affect the problem word slot. The problem worsens then for every word slot that does not affect the problem word slot that precedes the preceding word slot. For example if the word slot directly preceding the problem word slot has 50 possible words, and the word slot before that has 40 possible words, and neither of them affect the problem word slot, the program will attempt the problem word slot failing 2000 times, failing each time before it redeses
the word slot before this which hopefully affects the problem word slot and by filling this in with a new word, hopefully the program can move on further. To avoid the program pointless filling in numerous word slots with hundreds of words I decided to record the problem word and then before redoing a word slot check that the word slot we are redoing affects the problem word slot, and only bother to redo it if it does indeed affect the problem word slot. However what we wanted to record depended on the word slot failing, and this caused a problem. When Prolog fails, it undoes everything it’s done so far, so how can we carry out an action based on whether or not something has failed? We need to record that a word slot is the problem word slot only if we fail to get a word to fill it up with. The way I got around this is by recording that before each word slot is attempted to be filled it is recorded that its a problem. Then once we have filled the word slot with a word, the fact that it was a problem is erased. This was a very simple solution to s difficult problem. By this means the problem word is recorded and when the program backtracks, it only will redo words which actually have an effect on the problem word slot, and therefore redoing those word slots that do effect it might result on a solution to the grid being found. This method of avoiding backtracking has improved the speed times of the search significantly.

In addition to the problem word slot being recorded as being a problem, in some occasions other word slots will also be recorded as problem word slots. If we are filling in word slot 1 in the example grid in 5.8 and fail, sometimes it will fail because although there is a word that will fill up word slot 1, when this is filled in there will be no words to fill in word slot 3. Therefore word slot 1 would fail. In this circumstance both word slot 3 and word slot 1 are recorded as problem word slots and therefore on backtracking, any words that affect either of these will be redone. The problem words are erased when a word slot affecting one of the problem words is redone. This causes the prolog code to backtrack sensibly, avoiding exhaustive and pointless inserting of words.

5.7 Conclusion

In this chapter I gave an account of all the methods used to find the solutions to a crossword grid. Although this part of the project was in my opinion conceptually the most difficult part, it proved to be successful. Putting constraints on the way the grid was filled up with words greatly improved the search times.
CHAPTER 5. PROLOG SOLUTIONS TO SOLVING THE GRID

solve([SLOT_a21, SLOT_a26, SLOT_a31, SLOT_a36, SLOT_a41, SLOT_a46, SLOT_d22, SLOT_d27, SLOT_d42, SLOT_d47, SLOT_d05, SLOT_d65, SLOT_a55, SLOT_a65, SLOT_a05, SLOT_a15, SLOT_d37, SLOT_d57, SLOT_d17]):-

word(Space5, Space12, Space19, Space26, Space33, Space40, Space47),
word(Space1, Space2, Space3, Space4, Space5),
word(Space8, Space9, Space10, Space11, Space12),
word(Space4, Space11),
word(Space2, Space9),
word(Space1, Space8, Space15, Space22, Space29, Space36, Space43),
word(Space3, Space10, Space17, Space24, Space31, Space38, Space45),
word(Space26, Space27),
word(Space33, Space34),
word(Space19, Space20),
word(Space20, Space27, Space34),
word(Space36, Space37, Space38, Space39, Space40),
word(Space43, Space44, Space45, Space46, Space47),
word(Space39, Space46),
word(Space37, Space44),
word(Space14, Space15),
word(Space14, Space21, Space28),
word(Space28, Space29),
word(Space21, Space22),

name(SLOT_d57,[Space5, Space12, Space19, Space26, Space33, Space40, Space47]),
name(SLOT_a15,[Space8, Space9, Space10, Space11, Space12]),
name(SLOT_d37,[Space3, Space10, Space17, Space24, Space31, Space38, Space45]),
name(SLOT_d17,[Space1, Space8, Space15, Space22, Space29, Space36, Space43]),
name(SLOT_d05,[Space14, Space21, Space28]),
name(SLOT_d65,[Space20, Space27, Space34]),
name(SLOT_a55,[Space36, Space37, Space38, Space39, Space40]),
name(SLOT_a65,[Space43, Space44, Space45, Space46, Space47]),
name(SLOT_a05,[Space1, Space2, Space3, Space4, Space5]),
name(SLOT_a21,[Space14, Space15]),
name(SLOT_a26,[Space19, Space20]),
name(SLOT_a31,[Space21, Space22]),
name(SLOT_a36,[Space26, Space27]),
name(SLOT_a41,[Space28, Space29]),
name(SLOT_a46,[Space33, Space34]),
name(SLOT_d22,[Space2, Space9]),
name(SLOT_d27,[Space37, Space44]),
name(SLOT_d42,[Space4, Space11]),
name(SLOT_d47,[Space39, Space46]),

unique([SLOT_a21, SLOT_a26, SLOT_a31, SLOT_a36, SLOT_a41, SLOT_a46, SLOT_d22, SLOT_d27, SLOT_d42, SLOT_d47, SLOT_d05, SLOT_d65, SLOT_a55, SLOT_a65, SLOT_a05, SLOT_a15, SLOT_d37, SLOT_d57, SLOT_d17],[SLOT_a21, SLOT_a26, SLOT_a31, SLOT_a36, SLOT_a41, SLOT_a46, SLOT_d22, SLOT_d27, SLOT_d42, SLOT_d47, SLOT_d05, SLOT_d65, SLOT_a55, SLOT_a65, SLOT_a05, SLOT_a15, SLOT_d37, SLOT_d57, SLOT_d17]).
word([c,a,t]).
word([j,a,m]).
word([p,i,g]).
word([t,o,a,d]).
word([f,i,r,e]).
word([f,r,e,e]).
word([c,r,a,z,y]).
word([g,r,e,a,t]).
word([c,r,i,m,e]).

Figure 5.7: Representing the lexicon in Prolog.

Figure 5.8: The Variable Names given to the Cells

crossword([0,[S0,S1,S2]],
[1,[S10,S11,S12,S13,S14]],
[2,[S22,S23,S24]],
[3,[S10,S15,S20]],
[4,[S1,S6,S11]],
[5,[S13,S18,S23]],
[6,[S4,S9,S14]])

Figure 5.9: Example Input for Grid shown in fig. 5.8.
solve([SLOT_a21, SLOT_a26, SLOT_a31, SLOT_a36, SLOT_a41, SLOT_a46, SLOT_d22, SLOT_d27, SLOT_d42, SLOT_d47, SLOT_d50, SLOT_d56, SLOT_a56, SLOT_a65, SLOT_a55, SLOT_a65, SLOT_a05, SLOT_a15, SLOT_d37, SLOT_d57, SLOT_d17]) :-

  word(t, r, e, a, t, e, d),
  word(s, t, a, r, t),
  word(d, o, n, o, r),
  word(h, e, m),
  word(a, n),
  word(s, d, Space15, Space22, Space29, Space36, Space43),
  word(a, n, Space17, Space24, Space31, Space38, Space45),
  word(r, o),
  word(t, o),
  word(a, e),
  word(t, Space34),
  word(e, h),
  word(Space36, Space37, Space38, Space39, e),
  word(Space43, Space44, Space45, Space46, d),
  word(Space39, Space46),
  word(Space14, Space15),
  word(Space14, Space21, Space28),
  word(Space28, Space29),
  word(Space21, Space22),

...
evaluateAll([],_,_,[]).
evaluateAll([Word1|RWords],Spaces,Linking,Rest):-
copy_term((Spaces,Linking),(SpacesVars,LinkingVars)),
eval(Word1,SpacesVars,LinkingVars,1000000,0),
evaluateAll(RWords,Spaces,Linking,Rest).
evaluateAll([Word1|RWords],Spaces,Linking,[[Word1,Value]|Rest]]):-
copy_term((Spaces,Linking),(SpacesVars,LinkingVars)),
eval(Word1,SpacesVars,LinkingVars,1000000,Value),
evaluateAll(RWords,Spaces,Linking,Rest).
% if there's a word that causes a problem don't add it to return list
evaluateAll([Word1|RWords],Spaces,Linking,Rest):-
evaluateAll(RWords,Spaces,Linking,Rest).

% If the word has no linking words then give all the words the same value
eval(_,_,[],Min,Min).
eval(WordSpaces,WordSpaces,[[Num,Link1]|Rest],Minin,Minout):-
pass2([[Num,Link1]|Rest]),!,
setof(Link1,word(Link1),All),
length(All,Len),
((Len =< Minin, MinNew = Len); MinNew = Minin),
eval(WordSpaces,WordSpaces,Rest,MinNew,Minout).

Figure 5.11: Evaluating the Words
Chapter 6

Communication between Java and Prolog

6.1 Introduction

The random crossword generator involves the programming languages Java and Prolog. Java was used to generate a random grid and to give a user-friendly interface to the crossword generator, whereas Prolog was used to fill up the randomly generated grid with words, so that a valid crossword puzzle was generated. Since the generator uses these two languages, it was necessary to interface Java with Prolog. In the following sections I will describe how the prolog file used to solve the grid was generated and how to interface Java with Prolog, for further information (see SICS-AB, 2001).

6.2 Writing the Prolog File

Since the structure of the grid that is created is different each time a crossword is created, and a different database of words is used to fill up the grid, a prolog file must be written each time a crossword is generated. This is achieved using Java’s class File, FileWriter and BufferedWriter. A new file called crossword.pl is created, which overwrites the old crossword.pl file. The file contains the following lines of prolog code to import the necessary predicates contained in the file functions.pl:

- :- use_module(functions).

Also written in this file is the prolog database of words, that may be used to fill up the grid and the crossword fact used to represent the layout of the grid.

6.3 Jasper

Jasper is a bi-directional interface that is used to connect Java and Prolog programs. Since it is bi-directional it can enable either a Prolog program to use a Java program, or
it can be used by a Java program to call a Prolog program. In the case of the crossword
generator, I have used Java to call Prolog. The Java-side of the Jasper interface consists of
a Java package called java.se.sics.jasper, which contains classes that represent the SICStus
emulator. These classes provide ways to create prolog terms in Java and to ask queries,
requesting one or more solutions. The main Java classes of the package needed are:

- class SICStus
- class SPPredicate
- class SPTerm
- class SPQuery

### 6.4 Using Jasper to call Prolog from Java

Calling a prolog program from within a Java program involves a number of steps.

1. **Initializing the SICStus Emulator**
   
   Initializing the SICStus emulator in a Java class is achieved by creating a SICStus
   object. The constructor method of Class SICStus is used to create a new SICStus
   object:

   ```java
   public SICStus(String bootpath);
   ```

   The bootpath is the url of the files used to start up SICStus e.g. `/usr/local/depot/SICStusProlog-
   3.8.6/lib/sicstus-3.8.6/bin`. This method should only ever be called once within a
   program. If a SICStus object is needed in different parts of a program, or the object
   is used in a part of the program that will be carried out more than once, then in
   these cases instead of using the constructor to create a SICStus object, a method is
   used to retrieve the one SICStus object that has already been created:

   ```java
   public SICStus getInitializedSICStus();
   ```

2. **Loading a Prolog File**
   
   The prolog file containing the predicates we want to query must then be loaded into
   the SICStus object. This is achieved using the following method:

   ```java
   public void load(String filename);
   ```

   In the case of the crossword generator the filename specified in this method is `cross-
   word.pl` (see 6.2). This is the prolog file that we wish to query.
3. Creating a Predicate

It is necessary to create a predicate, which represents the predicate in the prolog file that we wish to query. This is achieved by calling the SPPredicate constructor. To create a predicate, we must specify the SICStus object we are using (see 1. above), as well as the name of the predicate in the prolog file we have loaded e.g. `solve`, how many arguments it takes e.g. `3` and lastly the name of the module to use when calling prolog. You can choose not to specify a module by including a set of empty quotation marks in place of the module name e.g. ""

- public SPPredicate(SICStus sictsusObj, String predicateName, int predicateArity, String module);

4. Creating Terms

It is also necessary to create terms to represent the arguments of the predicate. The terms that we are interested in creating were prolog variables and a prolog integer. A prolog variable is created as follows:

- SPTerm var = new SPTerm(sp).putVariable();

A prolog integer is created and given a value in one step. The following creates a prolog integer and gives it the value of 1000:

- SPTerm num = new SPTerm(sp,1000);

These terms can then be used in a query. For example if we wish to query the predicate `solve/3` (see Appendix)

- ?- solve(Solution,1000,Success).

5. Querying a Predicate

In order to query a predicate a query must be opened on the SISStus object we are using. When opening a query on a predicate, the predicate object in question must be specified as well as an array of terms which will be the arguments of the predicate. This is done by the following method of class SICStus:

- public SPQuery openQuery(SPPredicate pred, SPTerm[] termArray);

Once the query has been opened, it is used to query the predicate that was specified. The following method of class SPQuery gets the next solution to the query that has been opened:

- public void nextSolution();
This method is only called once in the crossword generator, as we are only interested in the first possible solution to the grid, but it may be called repeatedly on a query to acquire all possible solutions, if that was required. Once we have finished with the query it should then be closed, so that it can be garbage collected:

- public void close();

If left open the query could slow down further queries significantly.

### 6.5 Limiting the Time Allowed

The function solve/3 is used to get the solution to the puzzle. solve/3 uses the prolog predicate time_out/3 to put a limit on the amount of time given to finding a solution to the crossword. Figure 6.1 shows the predicate solve/3 which uses time_out/3.

```
solve(?Solution,+Timeout,?Success):-
    time_out(solve(Solution),Timeout,Success).
```

Figure 6.1: Predicate solve/3

The time limit is specified in milliseconds and is given as the second argument to time_out/3. If solve/1 succeeds within the specified number of milliseconds, the function succeeds, instantiating Solution to the solution to the grid and Success to ‘success’. If the the specified number of milliseconds passes and the function solve/1 has not yet succeeded or failed, then the solve/3 succeeds, but leaves Solution un-instantiated and instantiates Success to ‘timeout’. However if solve/1 fails before the specified number of milliseconds has elapsed then solve/3 simply fails.

### 6.6 Conclusion

In this chapter I described the technology involved in interfacing Java and Prolog and described how this was applied to the problem at hand i.e. interfacing the Java code of this project to Prolog code generated by the Java code. Interfacing the two languages proved not too difficult and enabled taking advantage of the advantages of both languages.
Chapter 7

Conclusion

7.1 Evaluation

When the project was taken on a number of goals were set out that should be achieved. They included;

- Creating a user friendly interface.
- Optimizing the search times of the Prolog code.
- Enable it to make foreign language/ multi-lingual crosswords.
- Redesigning the English dictionary database.

The first of these goals, to create a user friendly interface was achieved. The interface I created is indeed very user friendly. The crossword grid itself is displayed in the traditional style that crosswords are displayed in, so that the user is not distracted or confused by a complicated system of labeling the clues. Also, only the current clue is displayed at one time, so that the user does not have to scroll through lots of text to find each answer. In addition the user may simply move around the squares of the grid using the arrow keys or by clicking with the mouse. Therefore I think the interface I created is well designed and functions very well also, enabling the user to experience the full enjoyment of completing their crossword puzzle.

A major part of the work done on this project was on the way the Prolog code solved the grid. In my opinion the optimization of the code has been successful. When I took on the project, the largest grid the prolog would find a solution to in any reasonable time was width 4, and it can now solve grids of width 8+. If I had more time though I would have liked to test some of the constraints I put on the code individually. For example I would have liked to test the amount of time it took without the evaluation of the words to see how much of an effect this has on the search times.

The third of these aims was to enable the crossword generator to create crosswords in different languages. When it came about to implementing this, it seemed that instead of
creating a new dictionary in a foreign language and supplying the user with a choice of
two dictionaries, it was decided that it would be better to give the user the opportunity to
create their own dictionary and be able to add this dictionary permanently to the system.
In order to test that this function worked I prepared a small dictionary file which contained
about 2000 English words and their German translations. This was then loaded using the
add dictionary function. When this dictionary is selected it created crosswords whose
clues are in German and answers are the English translations of the clues. To test adding
a dictionary in the form of a database I also created a small dictionary made up of words,
and quotes of famous authors containing the words. This was also successfully added using
the add dictionary function. When this dictionary is selected, the crosswords that are
produced are quite difficult and a great knowledge of English literature would be required
to complete one of these.

The final aim was to redesign the database. This was achieved by examining the old
database and eliminating redundant data. I ended up deciding to create a new English
dictionary database from scratch as opposed to using the old database. This was a success
also as the current English dictionary database contains many more definitions than the
previous one. It had been a problem that more often than not, not enough words in
the dictionary were to do with the subject input by the user. This helped as the more
definitions there were the more words it would find for the subject.

Besides these aims it is probably worth mentioning that most of the Java code that
was inherited was also rewritten. In part the purpose of this was to make the code object
oriented. Other parts were rewritten because the algorithms used were unnecessarily very
complex. Rewriting the code was time consuming, but should result in the code being
much more readable if someone were to extend the project further.

### 7.2 Skills Learned

From working on this project over the last year I feel that I have really learned a great deal.
Since the project involved using databases for different reasons, I learned about designing
databases as well as structured query language. I also learned of the different ways there
are to populate databases. In addition to this I discovered some practical knowledge about
using databases. For example I learned that for a program to query a database it is quite
time consuming. Therefore it is better to construct a large SQL statement where possible
to select all the information needed in one go, than to select the data in multiple SQL
statements, and then sort through the retrieved data using Java. I also gained a great deal
of knowledge about how to program in Prolog. I heard someone say once (I think Vinny
Cahill) that the only way to learn to program is to write programs. This has proven to
be very true, as I never felt competent when programming in Prolog before this year. I
have also learned about some of the technical issues to do with servers, clients and applets,
which I'm sure will be useful in the future. Other bits of technology I learned along the
way were how the two programming languages, Java and Prolog can communicate through
Jasper.
7.3 Future Work

There are some things that I would have liked to improved on given enough time. One of them is relation of the subject input by the user to the clues and answers. As it stands the answers and clues may be used to create a puzzle if the word or its definition contains the sequence of characters entered by the user as the subject. This means that if the user enters the word "cat", the clues and words that are selected must contain 'c', 'a', 't'. Therefore clues that would be counted as related to the subject cat might just contain a word that contains the sequence of letters 'c', 'a', 't' and not the actual word "cat". It could have easily been changed that it does in fact just use words and definitions that contain the word as opposed to the sequence of letters, but when this was done there would not be enough words and definitions to create a crossword. One way of changing this would have been perhaps to use wordnet. wordnet is a database of words. It would be useful because related words are linked to each other. In the future perhaps wordnet could be used to get the words related to the subject. If it was used in this manor instead of the English dictionary, then the words would actually be related to the subject and possibly it would get enough words to create crosswords with.

A further modification that could be done, would be to rate the words in the dictionary on there difficulty, so that only words of a certain difficulty rating would be used in a puzzle of that difficulty rating. It was suggested that the difficulty ratings of the words could be relative to the number of definitions that the word had i.e. the more definitions a word has, the more ambiguous it is and therefore the more difficult it is to solve.

Yet another interesting idea of how the crossword be extended would be make a polyglot option available to the user. The clues and definitions could then be selected from all of the dictionaries available at that point in time. This would result in very interesting crosswords as there is currently a German dictionary available to it.

Once a crossword has been created, the applet can be used by another user to create another crossword. However a disadvantage of the system is that it cannot create more than one crossword at a time. Enabling the generator to do this would definitely be an advantage as the crossword is web-interfaced and therefore should be able to be used simultaneously by many users. This would not be too difficult. The reason multiple crossword cannot be created simultaneously is because the generator uses one single prolog file to get the solution to the crossword. A solution to this would be to simply to name each file it creates uniquely and then destroy the file when the crossword has been solved.

Another way in which the prolog could be improved would be to improve the search times so that larger grids could be created. One way of doing this would be to use Constraint Logic Programming over Finite Domains. A finite domain is a subset of integers, but only small integers and unbound variables are allowed in finite domain constraints. As constraints are added to a variable its domain gets increasingly smaller, and if the domain becomes empty due to a constraint being added, then the accumulated constraints cannot be satisfied and this results in failure of the current computation branch. CLP could be used by the system by giving each of the variables that represents each cell in the grid a finite domain. The letters of the alphabet could be represented in our prolog database of
words as the integers 1 to 26. Then the finite domain of each cell could be worked out relative to the database of words and there interlinking word slots. Take for example the following word slots; word slot [S1,S2,S3] across and its interlinking word slot [S1,S4,S7,S10] down. If there are only two words in the database of length 3, and one begins with ‘c’ and the other begins with ‘h’ then the finite domain of S1 would be the set 3,8. Then if there are no words of length 4 in the database beginning with ‘c’, a further constraint is placed on S1 reducing its finite domain to 8 (‘h’). If such constraints were placed on the domains of the variables representing the cells of the grid prior to the grid being solved, this would really improve the amount of time it takes to find a solution. Therefore I think Constraint Logic Programming would be particularly useful for finding solutions to crosswords.

7.4 Conclusions

Overall I think the project was a success. I feel that I’ve learned a great deal from it and personally am happy with what I’ve achieved. The project has been interfaced very well, as the applet is extremely easy to use and visually appealing. The efficiency of the crossword generator has also been improved greatly. This was achieved through constraining the way in which the crossword grid was filled up with words. This was a major part of the project and I feel from doing this I have really learned a lot and have not only increased the efficiency but increased the possible width of the crosswords from width 4 to width 10. Creating the English Dictionary database was worth-while piece of work, since it not only is useful for this project, but could very easily be used as part of another project that was in need of an English dictionary. Working on the database also taught me a great deal about database design and how to interface a program with a DBMS, which I’m sure will be very useful in the future. Another worthwhile achievement of the project was adding the feature of being able to add a user-supplied dictionary permanently to the system. This feature could be seen as a stand-alone piece of software that can be used by anyone to load a text file into a database. The main advantage of this feature is that because the dictionary can be in the form of a text file the user need know nothing about databases, DBMSs or structured query language to use it. Overall I feel I have gained a great deal of knowledge from doing this project and I hope it can be carried on with the same success.
Bibliography


Appendix A

The Static Prolog Code
:- dynamic(problem/1).

% solve(?Solution,+Timeout,+Success)  
solve(Solution,T,S) :- time_out(solve(Solution),T,S).

% If the word space has no linking spaces return empty list  
getLinking(Num,Spaces,[],[],[],WSF):-  
crossword(C),  
fillUp(C,WSF),  
member([Num,Spaces],C).

% If the word space only has one linking space don’t use recursive call  
getLinking(Num,Spaces,[NH|[]],[LH],[[NH,LH]],WSF):-  
crossword(C),  
fillUp(C,WSF),  
member([Num,Spaces],C),  
member([NH,LH],C).

getLinking(Num,Spaces,[NH|NT],[LH|LT],[[NH,LH]|PairsT],WSF):-  
crossword(C),  
fillUp(C,WSF),  
member([Num,Spaces],C),  
member([NH,LH],C),  
getSpaces1(NT,LT,PairsT,C).

getSpaces1([Num],[Linking],[[Num,Linking],[],],C):-  
member([Num,Linking],C).

getSpaces1([NH|NT],[LH|LT],[[NH,LH]|PairsT],C):-  
member([NH,LH],C),  
getSpaces1(NT,LT,PairsT,C).

getSpaces2(NumX,NumY,SpacesX,SpacesY):-  
crossword(C),  
member([NumX,SpacesX],C),  
member([NumY,SpacesY],C).

fillUp(C,[]).  
fillUp(C,[[Num,Spaces]|WSFT]):-  
member([Num,Spaces],C),  
fillUp(C,WSFT).

% if we run out of words to fit a word this word becomes the problem word  
fit(Num,[],_,_,_):-  
clearAll,  
record(problem(Num)),!, fail.

fit(Num,[[Word1,_]|_],WSF_Old,[[Num,Word1]|WSF_Old]):-  
not(pointless(Num)).

% If fitting this word has no effect on problem word and its linking words
APPENDIX A. THE STATIC PROLOG CODE

% stop trying to fit this word
fit(Num, [_|Rest], WSF_Old, WSF_New):-
    not(pointless(Num)),
    fit(Num, Rest, WSF_Old, WSF_New).

% solving this word space is pointless if it does not directly affect one % of the problem words
pointless(Num):-
    setof(Problems, problem(Problems), All), !,
    not(affectsAny(Num, All)).

affectsAny(Num, [H|_]):-
    affects(Num, H).
affectsAny(Num, [_|T]):-
    affectsAny(Num, T).

% affects1 returns the list of numbers of words that are affected by NumX % not including NumX
% affects2 returns the list of numbers of words that are affected by NumX % including NumX

% If we’ve reached the end
affects1(_, Start, End, In, In):-
    Start is End +1.

% We don’t need to know that a number affects itself
affects1(NumX, NumX, End, In, Affected):-
    NewS is NumX + 1,

% If word affects current word then record it and keep going
affects1(NumX, Start, End, In, Affected):-
    affects(NumX, Start),
    NewS is Start + 1,
    affects1(NumX, NewS, End, [Start|In], Affected).

% If word doesn’t affect current word then don’t record it but keep going
affects1(NumX, Start, End, In, Affected):-
    NewS is Start + 1,
    %NewS =< End,

% If we’ve reached the end
affects2(_, Start, End, In, In):-
    Start is End +1.

% If word affects current word then record it and keep going
affects2(NumX, Start, End, In, Affected):-
    affects(NumX, Start),
    NewS is Start + 1,
% NewS <= End,
affects2(NumX, NewS, End, [Start|In], Affected).

% If word doesn’t affect current word then don’t record it but keep going
affects2(NumX, Start, End, In, Affected):-
    NewS is Start +1,
    % NewS <= End,

% instantiating the letters of word numbered NumX effects word number NumY
affects2(NumX, NumY):-
    getSpaces2(NumX, NumY, SpacesX, SpacesY),
    copy_term((SpacesX, SpacesY), (Xvars, YVars)),
    word(Xvars), !,
    not(equals(SpacesY, YVars)).

% instantiating the letters of word numbered NumX effects word number NumY
affects(NumX, NumY):-
    getSpaces2(NumX, NumY, SpacesX, SpacesY),
    copy_term((SpacesX, SpacesY), (Xvars, YVars)),
    word(Xvars), !,
    not(equals(SpacesY, YVars)).

equals([], []).
equals(Word1, Word2):-
    var(Word1),
    var(Word2).

equals([H1|T1], [H2|T2]) :-
    var(H1),
    var(H2),
    equals(T1, T2).

equals([H1|T1], [H2|T2]) :-
    nonvar(H1),
    nonvar(H2),
    H1 = H2,
    equals(T1, T2).

not(X) :- X, !, fail.
not(_).

member(H, [H|_]).
member(H, [_|T]) :-
    member(H, T).

pass1([], _).
% If the linking word has already been solved then don’t check it
pass1([([HNum, HSpaces]|T], WSF) :-
    record(problem(HNum)),
    member([HNum, _], WSF),
    clear(HNum), !,
    pass1(T, WSF).

% Check that there is at least one word that can fill the linking spaces
% (word cannot already be part of solution or be the word that we are
% checking can go into solution)
pass1([[HNum,HSpaces]|T],WSF) :-
    record(problem(HNum)),
    word(HSpaces),
    not(member([_,HSpaces],WSF)),
    clear(HNum),!,
    pass1(T,WSF).

pass2([]).
pass2([[HNum,HSpaces]|T]) :-
    not(noword(HSpaces)),
    pass2(T).

noword(Word):-
    not(word(Word)).

name2(Word,LetterList) :-
    name1(LetterList,IntList),
    name(Word,IntList).

name1([],[]).
name1([Atom|List],[Int|IList]) :-
    name(Atom,[Int]),
    name1(List,IList).

% enter the current word slot, its linking slots and should return
% the possible words sorted the most likely word first
getWords(Num,Sorted,WSF,NumWords):-
    record(problem(Num)),
    affects1(Num,0,NumWords,[],LinkingNums),
    getLinking(Num,Spaces,LinkingNums,Linking,LPairs,WSF),
    var(Sorted),!,
    setof(Spaces,word(Spaces),All),
    trim(All,Spaces,LPairs,WSF,Trimmed),!,
    Trimmed \= [],
    clear(Num),
    clearAll,
    evaluateAll(Trimmed,Spaces,LPairs,WordsNValues),
    heapSort(WordsNValues,Sorted),!.

% trims off the words that will cause a problem
% records the number of the linking words that cause the problem
% these will be cleared again if the list of possible words
% returned is not empty
trim([],[],[],[],[]).
trim([H1|T1],Spaces,LPairs,WSF,[H1|T2]):-
    not(member([],H1,WSF)),
    copy_term((Spaces,LPairs),(SVars,LVars)),
    SVars = H1,
    % check that the linking spaces can be filled with at least one word
% that's not in WSF or is not the word we are filling the spaces with
pass1(LVars,[[_|H1]|WSF]),
trim(T1,Spaces,Linking,WSF,T2).
trim([_|T1],Spaces,Linking,WSF,T2):-
trim(T1,Spaces,Linking,WSF,T2).

evaluateAll([],_,_,[]).
evaluateAll([Word1|RWords],Spaces,Linking,Rest):-
copy_term((Spaces,Linking),(SpacesVars,LinkingVars)),
eval(Word1,SpacesVars,LinkingVars,1000000,0),
evaluateAll(RWords,Spaces,Linking,Rest).
evaluateAll([Word1|RWords],Spaces,Linking, [[Word1,Value]|Rest]]):-
copy_term((Spaces,Linking),(SpacesVars,LinkingVars)),
eval(Word1,SpacesVars,LinkingVars,1000000,Value),
evaluateAll(RWords,Spaces,Linking,Rest).
% if there's a word that causes a problem don't add it to return list
evaluateAll([Word1|RWords],Spaces,Linking,Rest):-
evaluateAll(RWords,Spaces,Linking,Rest).

% If the word has no linking words then give all the words the same value
eval(_,_,[],Min,Min).
eval(WordSpaces,WordSpaces,[[Num,Link1]|Rest],Minin,Minout):-
pass2([[Num,Link1]|Rest]),!,
setof(Link1,word(Link1),All),
length(All,Len),
((Len <= Minin, MinNew = Len); MinNew = Minin),
eval(WordSpaces,WordSpaces,Rest,MinNew,Minout).

% term t(Root, LeftPQ, RightPQ)
% insert(+Item, +PQueue, ?NewPQ)
insertP(Pair, [], t(Pair,[],[])). % base case

% insert to Right then switch
insertP([H1,T1], t([H2,T2],Left,Right), t(NewR,NewL,Left)) :-
(T1>T2, !, NewR=[H1,T1], insertP([H2,T2],Right,NewL));
(NewR=[H2,T2],insertP([H1,T1], Right, NewL)).
removeP(t(Root, [], []), Root, []). % base case
removeP(t(Root, Left, Right), Root, t(NewRo,NewLe,NewRi)):-
removeP(Left,Node,NewRi),
insertP(Node,Right,Temp),
removeP(Temp,NewRo,NewLe).

% listToPq(+List, ?PQueue)
list_to_pq([],[]).
list_to_pq([H|Rest],PQ):-
list_to_pq(Rest, PQQ),
insertP(H, PQQ, PQ).

% pqToList(+PQueue, ?List)
pq_to_list([], []).

pq_to_list(t(Root, PQ1, PQ2), [Root | L]) :-
   removeP(t(Root, PQ1, PQ2), Root, PQQ),
   pq_to_list(PQQ, L).

% heapsort
heapSort(List, SortedList) :-
   list_to_pq(List, PQueue),
   pq_to_list(PQueue, SortedList).

insert(Item, [], t(Item, [], [])).

% insert to Right then switch
insert(Item, t(Root, Left, Right), t(NewR, NewL, Left)) :-
   (Item > Root, !, NewR = Item, insert(Root, Right, NewL));
   (NewR = Root, insert(Item, Right, NewL)).

clear(Num) :- retract(problem(Num)).

clearAll :- not(problem(_)).
clearAll :-
   retract(problem(Num)),
   clearAll.

% if its already recorded that that number is a problem don’t record it again
record(X) :- X, !.
record(X) :- asserta(X).

append([], List, List).
append([H|T], List, [H|New]) :-
   append(T, List, New).

solve(Solution) :-
   getNumWords(NumWords),
   getOrder(Order), !,
   solveInOrder(Order, NumWords, [], Solution).

getNumWords(NumWords) :-
   crossword(C),
   length(C, NumWords).

g.getLongestWord([H|T], Longest) :-
   crossword(C),
   getLongest(C, [H|T], H, Longest).
APPENDIX A. THE STATIC PROLOG CODE

getLongest([],_,LSF,LSF).
getLongest([[Num,Word]|T],NumList,LSF,NewLSF):-
    member(Num,NumList),
    length(Word,Len1),
    crossword(C),
    member([LSF,LWord],C),
    length(LWord,Len2),
    ((Len1>Len2, 
    getLongest(T,NumList,Num,NewLSF));
    getLongest(T,NumList,LSF,NewLSF)).
getLongest([[Num,Word]|T],NumList,LSF,NewLSF):-
    getLongest(T,NumList,LSF,NewLSF).

getOrder(Order):-
    getAllNums(NumList),
    getLongestWord(NumList,L),
    getNumWords(N),
    getOrder([L],N,[L],Order).
getOrder([],NumWords,OSF,OSF):-
    length(OSF,NumWords).
getOrder([],N,OSF,Order):-
    getAllNums(All),
    trim1(All,OSF,NumList),
    getLongestWord(NumList,L),
    append(OSF,[L],NewOSF),
    getOrder([L],N,NewOSF,Order).
getOrder([HOrdered|TOrdered],NumWords,OSF,NewOrdered2):-
    affects1(HOrdered,0,NumWords,[],Affected),
    trim1(Affected,OSF,NewAffected),
    append(OSF,NewAffected,NewestOSF),
    append(TOrdered,NewAffected,Rest),
    getOrder(Rest,NumWords,NewestOSF,NewOrdered2).

getAllNums(Nums):-
    crossword(C),
    getAll(C,Nums).
ggetAll([[Num,Word]],[Num]).
ggetAll([[Num,Word]|T],[Num|NT]):-
    getAll(T,NT).

% a function to remove all items in List1 that also are in List2
trim1([],_,[]).
trim1([H1|T1],List2,Trimmed):-
    member(H1,List2),
    trim1(T1,List2,Trimmed).
trim1([H1|T1],List2,[H1|Trimmed]):-
trim1(T1, List2, Trimmed).

% solveInOrder(+Order,+NumWords,[],?WordsSoFar)
solveInOrder([],_,WSF,WSF).
solveInOrder([H|T], NumWords, WSF, WSF2):-
    getWords(H, Words, WSF, NumWords),
    fit(H, Words, WSF, WSF1),
    solveInOrder(T, NumWords, WSF1, WSF2).
Appendix B

Example Generation Parameters

![Image of example generation parameters]

- Width: 8
- Difficulty: Easy
- Subject: "café"
- Dictionary: Websters Unabridged English Dictionary
- Create A Crossword!
Appendix C

Example Resulting Dynamic Code
% subject = cat
:- use_module(library(timeout)).
:- use_module(functions).
:- dynamic(problem/1).

crossword([[0,[S0,S1,S2,S3,S4]],
            [1,[S6,S7]],
            [2,[S11,S12,S13]],
            [3,[S16,S17,S18,S19,S20,S21]],
            [4,[S30,S31]],
            [5,[S32,S33]],
            [6,[S42,S43,S44,S45,S46,S47]],
            [7,[S50,S51,S52]],
            [8,[S56,S57]],
            [9,[S59,S60,S61,S62,S63]],
            [10,[S16,S24,S32,S40,S48,S56]],
            [11,[S18,S26]],
            [12,[S42,S50]],
            [13,[S3,S11,S19]],
            [14,[S43,S51,S59]],
            [15,[S4,S12,S20]],
            [16,[S44,S52,S60]],
            [17,[S13,S21]],
            [18,[S37,S45]],
            [19,[S7,S15,S23,S31,S39,S47]])).

word([m,y]).
word([i,s]).
word([i,d]).
word([e,x]).
word([i,c]).
word([i,n]).
word([u,n]).
word([a,m]).
word([e,e]).
word([r,e]).
word([t,o]).
word([o,f]).
word([g,o]).
word([s,o]).
word([o,n]).
word([m,o]).
word([b,e]).
word([h,e]).
word([b,i,g]).
word([p,a,w]).
word([m,e,w]).
word([s,p,y]).
word([r,a,t]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([c,o,g]).
word([r,o,d]).
word([o,a,k]).
word([l,u,z]).
word([m,a,p]).
word([o,i,l]).
word([u,s,e]).
word([p,u,n]).
word([s,l,y]).
word([s,e,t]).
word([c,u,e]).
word([n,i,p]).
word([c,o,y]).
word([p,u,r]).
word([p,o,p]).
word([r,u,t]).
word([p,o,t]).
word([b,o,s]).
word([h,o,g]).
word([p,a,n]).
word([f,i,x]).
word([f,o,g]).
word([h,i,p]).
word([g,i,n]).
word([n,e,t]).
word([d,y,e]).
word([t,h,y]).
word([r,u,n]).
word([h,e,m]).
word([t,i,c]).
word([f,o,r]).
word([l,a,y]).
word([b,e,g]).
word([g,e,e]).
word([f,l,y]).
word([i,s,m]).
word([v,e,x]).
word([o,u,s]).
word([g,a,s]).
word([o,s,e]).
word([f,o,x]).
word([n,a,b]).
word([t,i,e]).
word([l,i,n]).
word([o,n,e]).
word([o,r,f]).
word([t,e,d]).
word([v,o,w]).
word([l,e,x]).
word([r,u,m]).
word([w,a,s]).
word([o,l,d]).
word([c,r,y]).
word([l,o,o]).
word([h,a,w]).
word([f,i,t]).
word([p,a,r]).
word([y,a,k]).
word([m,a,y]).
word([i,s,o]).
word([d,r,y]).
word([d,o,g]).
word([s,e,e]).
word([w,e,b]).
word([s,o,w]).
word([w,a,x]).
word([o,m,o]).
word([y,o,u]).
word([c,a,t]).
word([s,u,e]).
word([t,e,a]).
word([a,l,e]).
word([y,e,w]).
word([t,a,g]).
word([1,i,e]).
word([a,r,t]).
word([s,a,y]).
word([i,n,e]).
word([h,o,p]).
word([b,y,e]).
word([b,o,w]).
word([a,r,e]).
word([f,e,w]).
word([m,a,d]).
word([g,i,b]).
word([c,u,p]).
word([c,u,b]).
word([g,a,d]).
word([1,1,a,m,a]).
word([s,a,l,v,e]).
word([f,i,t,c,h]).
word([f,r,a,m,e]).
word([c,h,a,s,e]).
word([r,a,a,s,h]).
word([j,a,n,u,s]).
word([s,o,o,t,h]).
word([1,a,b,o,r]).
word([r,a,1,1,y]).
word([j,u,s,s,i]).
word([b,r,e,e,d]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([d,r,i,n,k]).
word([r,e,a,c,h]).
word([u,l,t,r,a]).
word([f,o,r,k,y]).
word([a,p,p,l,y]).
word([p,l,a,c,e]).
word([m,o,r,a,l]).
word([p,a,p,e,r]).
word([s,a,l,v,o]).
word([t,e,n,s,e]).
word([s,1,u,s,h]).
word([t,r,o,l,l]).
word([s,e,n,n,a]).
word([c,o,u,c,h]).
word([t,i,g,h,t]).
word([t,o,u,c,h]).
word([b,l,u,s,h]).
word([f,u,d,g,e]).
word([p,u,r,g,e]).
word([s,w,e,l,l]).
word([f,l,u,o,r]).
word([h,a,t,c,h]).
word([s,a,u,c,e]).
word([c,h,a,t,i]).
word([1,a,t,c,h]).
word([p,e,n,n,y]).
word([f,o,r,c,e]).
word([s,p,a,w,l]).
word([a,c,t,o,r]).
word([g,r,o,s,s]).
word([s,c,r,i,p]).
word([s,a,l,p,a]).
word([c,r,a,v,e]).
word([i,n,f,i,x]).
word([f,i,e,l,d]).
word([s,c,o,u,r]).
word([r,e,b,u,s]).
word([m,a,t,c,h]).
word([g,u,i,s,e]).
word([r,i,n,s,e]).
word([t,o,w,e,r]).
word([c,a,t,e,r]).
word([s,h,a,r,p]).
word([c,h,a,j,a]).
word([f,e,r,r,l]).
word([p,y,l,o,n]).
word([a,g,a,t,e]).
word([j,u,l,u,s]).
word([1,y,r,i,e]).
word([e,n,t,e,r]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([g,u,i,d,e]).
word([d,r,u,n,k]).
word([m,u,c,u,s]).
word([t,r,e,a,t]).
word([t,h,u,r,l]).
word([p,o,w,e,r]).
word([f,r,o,n,d]).
word([b,o,o,l,y]).
word([f,e,r,r,o]).
word([b,o,o,z,y]).
word([k,r,a,o,l]).
word([m,e,r,c,y]).
word([a,n,o,d,e]).
word([t,a,k,e,r]).
word([f,e,l,i,s]).
word([b,o,n,n,y]).
word([m,y,r,i,a]).
word([r,e,c,t,o]).
word([r,a,m,u,s]).
word([t,a,s,t,e]).
word([s,l,i,d,e]).
word([e,x,o,d,e]).
word([t,a,r,s,o]).
word([p,r,e,s,s]).
word([s,h,a,l,l]).
word([s,q,u,i,b]).
word([t,i,n,g,e]).
word([g,l,e,a,n]).
word([s,m,o,o,r]).
word([p,r,o,t,o]).
word([m,i,d,g,e]).
word([c,h,e,c,k]).
word([m,i,a,u,l]).
word([f,i,b,e,r]).
word([p,a,u,x,i]).
word([h,i,t,c,h]).
word([s,p,l,a,y]).
word([o,l,i,v,e]).
word([b,r,a,k,e]).
word([s,w,e,a,t]).
word([b,h,a,n,g]).
word([p,o,i,n,d]).
word([c,a,t,s,o]).
word([c,e,n,s,e]).
word([a,b,u,s,s,e]).
word([f,l,o,a,t]).
word([m,a,u,v,e]).
word([b,l,o,o,m]).
word([h,o,v,e,n]).
word([f,l,i,n,g]).
word([b,l,o,w,n]).
word([c,a,t,e,l]).
word([g,o,e,t,y]).
word([s,t,y,l,e]).
word([s,t,u,d,y]).
word([s,n,a,r,l]).
word([f,a,u,l,t]).
word([a,s,k,e,w]).
word([b,r,a,n,d]).
word([t,a,l,l,y]).
word([m,o,u,s,e]).
word([c,r,a,n,k]).
word([m,i,n,o,r]).
word([p,a,t,c,h]).
word([f,r,i,t,h]).
word([v,o,i,c,e]).
word([h,o,o,o,v,e]).
word([w,e,a,v,e]).
word([c,l,a,s,s]).
word([d,r,o,w,n]).
word([g,r,a,c,e]).
word([g,a,y,a,l]).
word([r,a,n,c,h]).
word([t,o,o,t,h]).
word([s,t,e,l,l]).
word([a,n,i,o,n]).
word([s,t,o,c,k]).
word([t,a,b,b,y]).
word([m,o,n,e,y]).
word([t,i,r,m,a]).
word([k,e,t,c,h]).
word([i,n,d,e,x]).
word([j,u,r,a,t]).
word([t,e,a,c,h]).
word([b,a,r,t,h]).
word([p,e,a,r,l]).
word([s,t,a,n,d]).
word([t,w,i,s,t]).
word([s,h,a,d,e]).
word([w,r,a,w,l]).
word([s,h,o,t,s]).
word([c,a,r,v,e]).
word([s,k,i,l,l]).
word([h,e,a,d,y]).
word([s,w,e,a,r]).
word([o,f,f,e,r]).
word([d,a,i,l,y]).
word([c,l,o,s,h]).
word([c,a,t,c,h]).
word([r,i,g,o,r]).
word([d,e,v,o,n]).
word([c,o,b,i,a]).
word([b,o,y,a,u]).
word([b,e,r,y,l]).
word([w,r,i,t,e]).
word([g,a,r,t,h]).
word([p,i,c,r,a]).
word([c,h,o,k,e]).
word([t,a,c,h,e]).
word([t,r,o,p,e]).
word([g,a,r,t,h]).
word([t,a,c,h,e]).
word([t,r,o,p,e]).
word([s,c,o,r,e]).
word([c,l,o,c,k]).
word([a,v,o,i,d]).
word([t,o,k,e,n]).
word([b,l,i,n,d]).
word([r,u,c,h,e]).
word([s,w,o,o,p]).
word([t,i,p,s,y]).
word([m,a,n,u,l]).
word([s,n,a,r,e]).
word([c,r,o,s,s]).
word([f,l,o,s,s]).
word([p,u,s,s,y]).
word([f,e,r,a,l]).
word([p,o,u,n,d]).
word([m,o,d,u,s]).
word([t,o,p,a,z]).
word([p,o,i,n,t]).
word([c,a,t,e,s]).
word([n,o,o,s,e]).
word([a,g,i,s,t]).
word([g,y,p,s,y]).
word([o,v,u,l,e]).
word([h,e,a,v,e]).
word([h,o,u,s,e]).
word([c,r,o,o,n]).
word([v,e,l,u,m]).
word([p,a,u,s,e]).
word([c,h,a,f,f]).
word([s,p,i,r,e]).
word([s,p,i,l,l]).
word([s,t,a,i,n]).
word([c,l,o,s,e]).
word([s,i,r,u,p]).
word([c,l,i,c,k]).
word([g,r,a,s,s]).
word([b,u,r,k,e]).
word([a,u,g,u,r]).
word([w,a,t,r]).
word([s,t,y,l,o]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([b,e,n,e,t]).
word([d,e,u,t,o]).
word([r,o,w,e,n]).
word([j,o,i,n,t]).
word([d,r,i,e,r]).
word([p,a,i,n,t]).
word([b,r,e,a,m]).
word([t,a,b,l,e]).
word([d,r,o,v,e]).
word([s,t,a,r,t]).
word([m,a,n,g,e]).
word([h,o,v,e,l]).
word([r,a,d,i,o]).
word([b,o,o,b,y]).
word([g,r,a,z,e]).
word([b,r,e,v,e]).
word([s,o,u,n,d]).
word([w,a,s,t,e]).
word([c,a,r,e,t]).
word([v,a,g,u,e]).
word([g,r,i,p,e]).
word([b,l,e,n,d]).
word([t,r,a,w,l]).
word([n,o,m,a,d]).
word([t,i,l,d,e]).
word([r,a,n,g,e]).
word([b,l,a,n,k]).
word([s,c,a,l,a]).
word([f,a,g,o,t]).
word([l,a,r,v,a]).
word([l,o,b,b,y]).
word([c,u,r,s,e]).
word([s,o,b,e,r]).
word([s,h,e,l,l]).
word([l,a,p,s,e]).
word([g,u,l,e,s]).
word([l,e,a,r,n]).
word([r,o,u,n,d]).
word([t,o,p,i,c]).
word([d,o,l,c,e]).
word([h,y,d,r,o]).
word([n,a,s,t,y]).
word([c,i,v,e,t]).
word([l,a,s,s,o]).
word([s,e,n,s,e]).
word([r,a,v,e,l]).
word([c,r,e,s,t]).
word([c,l,e,a,r]).
word([t,r,a,i,n]).
word([s,t,r,a,w]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([l,i,b,e,l]).
word([p,r,i,n,t]).
word([b,u,i,l,d]).
word([a,n,g,l,e]).
word([s,t,r,e,w]).
word([v,e,n,o,m]).
word([f,l,e,s,h]).
word([a,g,a,m,i]).
word([o,r,g,a,n]).
word([t,h,u,j,a]).
word([s,m,a,c,k]).
word([s,t,i,v,e]).
word([m,o,d,a,l]).
word([s,p,i,t,e]).
word([g,a,n,j,a]).
word([v,a,l,u,e]).
word([s,t,a,l,k]).
word([d,r,i,f,t]).
word([t,r,u,n,k]).
word([o,x,f,l,y]).
word([c,a,u,s,e]).
word([f,i,s,h,y]).
word([l,o,g,i,c]).
word([s,t,a,l,l]).
word([v,e,r,s,e]).
word([t,i,b,i,o]).
word([s,c,a,t,h]).
word([p,l,e,a,d]).
word([s,h,a,d,d]).
word([d,r,o,m,e]).
word([r,i,g,h,t]).
word([t,h,y,r,o]).
word([p,i,l,o,t]).
word([f,l,o,c,k]).
word([k,n,e,l,l]).
word([h,e,d,g,e]).
word([g,r,a,s,p]).
word([p,e,w,e,e]).
word([f,l,y,e,r]).
word([s,m,i,l,e]).
word([h,o,r,s,e]).
word([e,r,u,c,a]).
word([z,e,b,u,b]).
word([p,l,a,i,n]).
word([r,a,i,s,e]).
word([s,h,i,f,t]).
word([s,t,u,f,f]).
word([l,i,g,h,t]).
word([b,r,e,a,k]).
word([[d,r,i,v,e]]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([c,a,t,t,y]).
word([t,h,r,o,w]).
word([p,l,u,m,m,b]).
word([q,u,o,l,l]).
word([l,a,b,e,l]).
word([s,t,a,l,e]).
word([d,u,c,a,t]).
word([s,n,e,e,r]).
word([h,a,s,t,y]).
word([s,c,a,l,e]).
word([g,r,e,a,t]).
word([c,a,n,o,n]).
word([d,u,o,m,o]).
word([s,a,v,o,r]).
word([c,o,l,z,a]).
word([u,s,a,g,e]).
word([d,u,m,m,y]).
word([v,o,l,t,a]).
word([a,m,e,n,t]).
word([f,o,r,g,e]).
word([i,d,i,o,t]).
word([s,h,o,v,e]).
word([s,p,r,a,y]).
word([b,r,o,a,d]).
word([c,l,a,s,p]).
word([a,v,o,w,r,y]).
word([c,o,r,y,z,a]).
word([a,c,c,e,n,t]).
word([a,b,l,a,u,t]).
word([c,l,a,u,s,e]).
word([a,l,b,i,t,e]).
word([a,v,o,c,a,t]).
word([d,u,r,h,a,m]).
word([c,a,t,n,i,p]).
word([c,o,b,w,e,b]).
word([c,a,t,t,l,e]).
word([f,o,r,k,e,d]).
word([c,e,r,a,t,e]).
word([f,r,a,x,i,n]).
word([g,i,g,g,l,e]).
word([a,r,r,e,s,t]).
word([f,o,m,e,n,t]).
word([e,n,t,r,a,p]).
word([f,i,l,t,e,r]).
word([a,u,g,u,r,y]).
word([c,a,t,s,u,p]).
word([g,r,e,a,s,e]).
word([a,n,s,w,e,r]).
word([e,n,t,a,i,l]).
word([b,a,s,h,a,w]).
word([c,i,p,p,u,s]).
word([a,p,p,e,a,l]).
word([a,n,g,o,r,a]).
word([e,a,t,a,g,e]).
word([f,i,n,i,n,g]).
word([d,r,e,d,g,e]).
word([f,e,r,r,e,t]).
word([c,o,l,l,a,r]).
word([e,a,t,a,g,e]).
word([d,r,o,v,e,r]).
word([e,n,s,i,g,n]).
word([f,o,s,t,e,r]).
word([c,e,r,i,t,e]).
word([b,a,r,b,e,l]).
word([g,r,e,a,s,y]).
word([a,d,v,i,s,e]).
word([c,o,r,o,n,a]).
word([b,e,c,a,r,d]).
word([e,s,c,h,a,r]).
word([c,h,a,r,g,e]).
word([e,x,p,e,n,d]).
word([b,a,g,n,e,t]).
word([c,o,r,r,a,l]).
word([e,n,s,i,g,n]).
word([f,o,s,t,e,r]).
word([c,e,r,i,t,e]).
word([b,a,r,b,e,l]).
word([g,r,e,a,s,y]).
word([a,d,v,i,s,e]).
word([c,o,p,u,i,a]).
word([a,u,r,o,x,a]).
word([e,d,i,t,o,r]).
word([a,v,e,n,g,e]).
word([b,i,c,k,e,r]).
word([d,o,m,i,n,o]).
word([a,n,g,l,e,r]).
word([a,n,g,l,n,a]).
word([b,l,a,z,o,n]).
word([d,a,t,i,v,e]).
word([a,c,t,i,v,e]).
word([d,e,f,i,v,e]).
word([b,e,h,a,l,f]).
word([b,e,l,i,a,l]).
word([c,r,e,d,i,t]).
word([d,a,y,n,e,t]).
word([f,o,r,g,e,r]).
word([e,e,l,p,o,t]).
word([l,e,a,d,e,r]).
word([a,r,t,e,r,y]).
word([b,o,n,d,a,r]).
word([c,o,w,p,o,x]).
word([f,o,u,s,s,a]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([e, x, c, e, s, s]).
word([b, i, r, d, e, r]).
word([b, e, c, l, a, p]).
word([c, o, u, p, o, n]).
word([d, e, g, r, e, s]).
word([g, r, i, p, p, e]).
word([b, o, n, i, t, o]).
word([d, e, e, s, i, s]).
word([d, a, i, n, t, y]).
word([g, u, i, t, a, r]).
word([i, o, l, i, t, e]).
word([d, e, s, i, g, n]).
word([b, a, w, d, r, y]).
word([d, e, c, a, n, e]).
word([f, a, b, r, i, c]).
word([d, e, h, o, r, n]).
word([h, e, m, m, e, l]).
word([l, a, r, k, e, r]).
word([a, l, p, a, c, a]).
word([a, n, n, a, l, s]).
word([g, u, i, l, t, y]).
word([d, e, n, o, t, e]).
word([i, n, v, o, k, e]).
word([c, r, a, d, l, e]).
word([c, o, w, b, o, y]).
word([c, r, i, s, i, s]).
word([d, a, c, a, p, o]).
word([f, i, r, i, n, g]).
word([i, t, a, l, i, c]).
word([a, p, p, e, a, r]).
word([a, b, d, e, s, t]).
word([c, r, o, u, p, y]).
word([i, m, p, l, e, x]).
word([h, a, m, m, e, r]).
word([f, o, d, d, e, r]).
word([d, e, t, e, n, t]).
word([e, n, r, o, l, l]).
word([c, a, l, i, c, o]).
word([i, n, c, u, l, k]).
word([c, a, n, a, r, d]).
word([l, e, s, s, o, n]).
word([e, n, a, m, e, l]).
word([b, r, e, a, c, h]).
word([l, i, t, a, n, y]).
word([f, o, r, a, g, e]).
word([g, a, s, i, f, y]).
word([c, a, r, i, n, a]).
word([k, i, d, d, l, e]).
word([b, r, y, o, n, y]).
word([b,e,t,r,a,p]).
word([f,e,l,i,n,e]).
word([c,a,r,b,o,n]).
word([i,n,d,i,c,e]).
word([b,u,c,k,l,i,e]).
word([e,t,y,m,o,n]).
word([b,u,t,t,o,n]).
word([l,e,t,t,e,r]).
word([c,a,t,k,i,n]).
word([k,i,t,t,e,n]).
word([e,x,p,o,r,t]).
word([b,u,c,k,e,t]).
word([h,a,f,f,l,e]).
word([d,i,s,p,e,l]).
word([d,i,g,e,e,t]).
word([f,i,g,u,r,e]).
word([g,i,b,c,a,t]).
word([b,r,a,n,c,h]).
word([c,a,t,g,u,t]).
word([h,e,l,m,e,t]).
word([c,a,t,h,a,y]).
word([l,i,t,t,e,r]).
word([d,e,r,i,v,e]).
word([j,u,n,k,e,t]).
word([e,n,m,e,s,h]).
word([f,u,c,a,t,e]).
word([d,i,v,e,r,t]).
word([i,n,f,e,s,t]).
word([g,e,n,t,x,y]).
word([g,e,n,d,e,r]).
word([b,e,s,n,o,w]).
word([b,e,l,i,e,f]).
word([l,e,v,a,n,t]).
word([d,e,v,i,s,e]).
word([d,o,m,i,f,y]).
word([f,r,u,g,a,l]).
word([d,e,v,o,t,e]).
word([c,a,t,e,n,a]).
word([i,n,f,o,r,m]).
word([c,l,o,u,d,y]).
word([c,h,e,b,e,c]).
word([h,u,m,i,t,e]).
word([c,a,t,i,o,n]).
word([b,e,t,r,a,y]).
word([d,o,d,d,e,d]).
word([g,l,e,a,m,y]).
word([f,e,e,d,e,r]).
word([c,e,n,s,o,r]).
word([d,i,v,i,n,e]).
word([b,o,w,n,e,t]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([p,h,l,e,u,m]).
word([m,a,n,a,g,e]).
word([s,e,e,s,a,w]).
word([s,m,u,d,g,e]).
word([s,c,a,t,h,e]).
word([s,c,h,o,o,l]).
word([m,a,n,t,r,a]).
word([s,i,m,o,o,m]).
word([s,y,n,d,i,c]).
word([s,t,r,i,n,g]).
word([m,e,t,h,o,d]).
word([r,e,l,l,i,s,h]).
word([t,a,m,p,o,n]).
word([t,a,x,i,n,e]).
word([p,e,c,o,o,r,a]).
word([m,a,n,n,e,r]).
word([p,a,p,i,s,t]).
word([t,a,t,t,l,e]).
word([r,e,a,d,e,r]).
word([t,a,u,r,u,s]).
word([s,n,a,t,c,h]).
word([s,p,a,d,i,x]).
word([m,a,k,i,n,g]).
word([s,t,o,w,c,e]).
word([s,t,r,i,k,e]).
word([m,a,r,k,e,t]).
word([s,p,a,r,s,e]).
word([l,u,x,a,t,e]).
word([s,e,a,p,i,e]).
word([q,u,i,n,s,y]).
word([s,q,u,e,a,l]).
word([s,e,r,v,a,l]).
word([p,a,l,m,e,r]).
word([s,e,r,i,a,l]).
word([p,r,e,a,c,h]).
word([p,o,a,c,h,y]).
word([m,a,n,g,e,r]).
word([p,r,a,y,e,r]).
word([m,e,r,c,a,t]).
word([s,o,e,v,e,r]).
word([m,e,d,i,u,m]).
word([s,t,r,i,f,e]).
word([m,a,t,a,c,o]).
word([m,e,n,s,e,s]).
word([r,a,c,k,e,t]).
word([s,t,e,r,n,o]).
word([s,t,i,f,l,e]).
word([r,e,m,o,v,e]).
word([s,u,b,t,l,y]).
word([s,t,r,o,n,g]).
word([s,t,r,i,p,e]).
word([m,e,n,a,c,e]).
word([s,p,h,e,n,o]).
word([s,t,r,o,k,e]).
word([p,s,y,c,h,e]).
word([p,u,r,g,e,r]).
word([p,r,e,f,i,x]).
word([m,u,f,f,i,s]).
word([t,e,i,l,s,e,r]).
word([m,o,t,i,o,n]).
word([t,i,c,k,e,t]).
word([t,h,r,i,v,e]).
word([p,u,l,q,u,e]).
word([m,u,c,a,t,e]).
word([m,o,u,s,e,r]).
word([t,i,b,c,a,t]).
word([o,b,l,a,t,e]).
word([r,e,d,u,c,e]).
word([t,i,p,c,a,t]).
word([t,i,s,s,u,e]).
word([m,u,l,l,e,t]).
word([p,h,y,s,i,c]).
word([r,e,p,e,a,l]).
word([t,e,g,m,e,n]).
word([p,a,t,r,o,n]).
word([t,e,n,d,e,r]).
word([t,i,d,b,i,t]).
word([t,h,e,c,i,a]).
word([t,h,i,n,l,y]).
word([t,e,m,p,l,e]).
word([o,c,t,a,v,o]).
word([p,u,t,a,g,e]).
word([m,u,d,d,l,e]).
word([t,i,t,a,n,o]).
word([r,e,t,u,r,n]).
word([r,e,d,t,o,p]).
word([t,i,t,l,e,r]).
word([m,u,s,t,e,r]).
word([t,r,i,s,t,e]).
word([t,r,a,v,e,l]).
word([r,e,c,a,l,l]).
word([t,o,m,c,a,t]).
word([r,e,v,i,e,w]).
word([t,r,a,p,a,n]).
word([r,e,p,o,s,e]).
word([r,e,d,a,c,t]).
word([u,m,l,a,u,t]).
word([u,n,b,a,r,k]).
word([v,a,c,a,t,e]).
APPENDIX C. EXAMPLE RESULTING DYNAMIC CODE

word([m,u,s,c,a,t]).
word([p,i,n,i,t,e]).
word([t,s,e,t,s,e]).
word([r,e,j,o,i,n]).
word([u,n,r,o,o,t]).
word([t,u,r,k,e,y]).
word([p,u,z,z,l,e]).
word([r,e,f,i,n,e]).
word([m,u,l,l,e,y]).
word([t,u,n,n,e,l]).
word([t,u,r,b,o,t]).
word([m,u,m,b,l,e]).
word([v,e,l,v,e,t]).
word([v,a,c,h,e,r]).
word([v,e,n,t,r,o]).
word([r,e,p,i,n,e]).
word([u,p,l,i,f,t]).
word([u,n,w,e,l,l]).
word([p,a,t,h,i,c]).
word([v,a,l,l,u,m]).
word([v,a,l,e,r,o]).
word([q,u,e,a,s,y]).
word([u,p,r,o,o,t]).
word([r,e,s,c,a,t]).
word([r,e,v,e,a,l]).
word([p,i,n,n,e,r]).
word([v,e,r,i,f,y]).
word([r,o,b,u,s,t]).
word([v,e,s,i,c,o]).
word([r,e,p,o,r,t]).
word([n,e,p,e,t,a]).
word([v,i,c,t,i,m]).
word([n,a,s,c,a,l]).
word([n,a,t,i,o,n]).
word([v,i,r,g,i,n]).
word([v,e,r,m,e,s]).
word([r,i,f,f,i,e]).
word([v,o,i,d,e,r]).
word([v,i,r,i,a,l]).
word([z,e,n,i,c,k]).
word([o,b,t,e,s,t]).
word([w,a,n,i,o,n]).
word([w,a,g,a,t,i]).
word([y,e,n,i,t,e]).
word([p,a,c,a,t,e]).
word([o,r,a,c,l,e]).
word([p,a,l,a,t,o]).
word([o,r,i,s,o,n]).
word([x,e,r,i,f,f]).
word([n,i,m,b,u,s]).
word([w,e,e,k,l,y]).
word([w,i,n,t,e,r]).
word([w,a,r,b,l,e]).
word([n,u,a,n,c,e]).
word([n,o,t,i,c,e]).
word([o,y,s,t,e,r]).
word([v,u,l,g,a,r]).
word([n,u,m,b,e,r]).
word([o,p,e,l,e,t]).
word([o,r,g,y,i,a]).
word([o,p,t,i,c,s]).
word([w,h,i,s,k,y]).
word([o,b,l,a,t,i]).
word([n,i,c,e,t,y]).
word([w,o,r,m,i,l]).
word([o,i,l,l,e,t]).
word([z,a,m,a,n,g]).
Appendix D

Example Screen Shot
Pre-Completion

1 ACROSS: To affirm or utter a solemn declaration, with an appeal to God for the truth of what is affirmed; to make a promise, threat, or resolve on oath; also, to affirm solemnly by some sacred object, or one regarded as sacred, as the Bible, the Koran, etc. (5)
Appendix E

Example Screen Shot
Post-Completion

18 ACROSS: To contort; to writhe; to complicate; to crook spirally; to convolve. (5)
Appendix F

Connecting to the Database
1. Secure shell to macneill.
   - csh> ssh macneill

2. Connect to MySql.
   - csh> mysql -u gibbonsa -p
     - password: persistence

3. Begin using the Dictionary Database
   - mysql> use dictionary;

4. View the Tables in the Database
   - mysql> show tables;
Appendix G

Setting the Classpaths
Before starting up the server the classpaths have to be set. Once you’ve secure shelled to the sun machine that the server runs from e.g. sun40. Type the following: This sets the environment variables of the sun machine you’ve secure shelled to. Each of the classpaths is separated in the statement by a colon. The ‘.’ makes all the files in the current directory visible. 

```
setenv CLASSPATH ./usr/local/depot/SICStusProlog-3.8.6/lib/sicstus-3.8.6/bin/jasper.jar:
/users/ug/grahamy/www/mysql-connector-java-2.0.14/mysql-connector-java-2.0.14-bin.jar
```

`/usr/local/depot/SICStusProlog-3.8.6/lib/sicstus-3.8.6/bin/jasper.jar` gives the location of the Jasper files needed to interface Java and Prolog. 

`/users/ug/grahamy/www/mysql-connector-java-2.0.14/mysql-connector-java-2.0.14-bin.jar` is the location of the files needed to connect to MySql.