



**EPCC-SS-2000-05**

## **A Reading List Authoring Tool for Course Web Site Creators**

**Andrew Marshall**

### **Abstract**

A University course web site will invariably contain references to textbooks. The course page creator will generally have to type this information in. The work of checking library holdings for these texts also has to be done manually. This project tries to ease the process of creating online reading lists by allowing authors to search the library catalogue, select desired items and generate lists directly from the search results. The web-based list can then be used by students to provide immediate information about the whereabouts and loan status of each item. The application is web based using CGI scripts written in Perl. It uses the Z39.50 [1] to access the library database via a Perl module called ZetaPerl [2].



## Introduction

A University course web site will invariably contain references to textbooks, the course reading list, which students have to refer to in order to supplement their course. The course lecturer will generally have to type this information in, a process that is slow and prone to error. Even then the library may not have holdings, that is whether the library has a copy of the book or if it is on the shelf or out on loan, on some of the texts the lecturer is recommending. This situation is far from ideal. This project tries to ease the process by creating these on-line reading lists.

The project can be split into two main parts:

- An authoring front end that allows lecturers to access the library database and select the required books directly. The authoring process generates a complete html page (or optionally an html stub) containing information about the selected books - title, author etc. In essence, this first part produces the reading list.
- The second part works at the student or client end. When the student visits the course web page, they are presented with the reading list previously generated. Moreover beside each entry, there will also be an html link. This link, when activated, presents them with more detailed information on the textbook, publisher etc, as well as the library's current holding information.

The program interface is web based, using calls to CGI (Common Gateway Interface) scripts written in Perl. CGI is a standard method of connecting programs to web servers.

One of the primary aims of this project is to allow portability of the CGI script and hence not incorporate any vendor specific methods or information. To this end, the way the library database is accessed is done using the Z39.50 standard [1]. This is a fairly complicated standard to use. Most of the Z39.50 functionality used in the script has been done through a Perl module called ZetaPerl [2].

Many university libraries support at least the basics of this standard [3]. At the time of writing, we could not obtain Edinburgh University Library's holding information via Z39.50 so an alternative was sought. Edinburgh University uses a library system called Voyager, created by Endeavor [4], which provides a web interface to search the library and obtain holdings information. The program simply piggybacks onto the 'end' part of this to get the holdings information.

## Current system and program objectives

The simplest way for a course web site creator to enter texts into the page is to manually type them in. This has three main drawbacks:

- The lecturer may have no idea if the library owns the recommended texts, or even how many copies it has.

- It is potentially error prone – entering texts could result in mistakes being made, etc.
- When a student visits the web page, there may be no indication if the books are in stock in the library – or, in the case where the library holdings are distributed over several buildings, which library the books are held at.

Edinburgh University Library has a web-based method to query its library database. This system gives bibliographic information on the text (title, author etc) as well as current holding information. The lecturer could in principle use this to search for every text they are thinking of adding to the reading list, and thus provide a mechanism for creating the reading list in the first place. They could even ‘copy and paste’ the information returned from the library we searches into their own web site and thus save on typing. The student could also use the same on-line library catalogue and type in the book information from the reading list each time they want to find out the holding status of the books required for their course.

Providing an academic with an integrated environment in which they can search the library database, select the required texts and have a reading list created would however be much more advantageous. Allowing a student to easily query the holdings information for availability of the texts as well as location would be an added bonus.

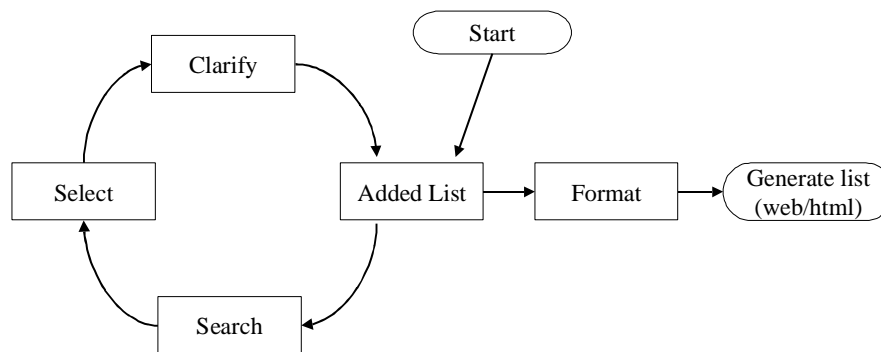
The requirements for the CGI application are thus:

- Provide a method of searching the library database to generate a web based reading list for an academic
- The interface must be fast - or it would be quicker for the user to just type in or cut and past the references
- The interface must be simple to use and robust
- The program should be easily maintainable
- The program should be portable

The fifth requirement implies the application must be vendor independent. In this instance, Z39.50 was used, as many libraries appear to support this standard. There are alternative emerging methods, such as one based on XML. It should be straightforward to replace Z39.50, used in the script that has been written, with an alternative mechanism to connect to a public library database. So if another standard does come to prominence the application can be modified to use the new interface instead.

## **Program Overview (usage)**

The general flow chart describing the reading list creator is shown below. The arrows indicate the normal flow of the program. It is often possible to go back a stage, but this is not shown on this diagram.

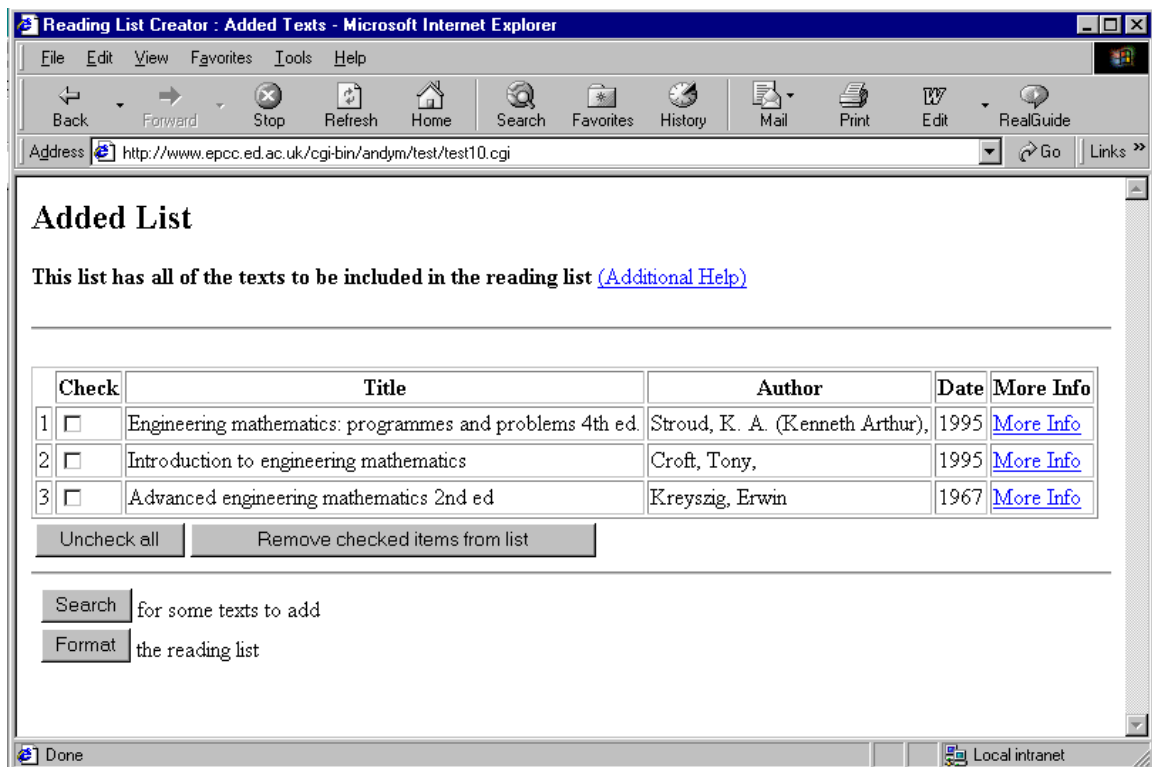


**Figure 1: schematic representation of the program flow.**

Each of these components will now be described below.

### Added List:

Contains a summary of all the texts selected for the reading list, see Figure 2.

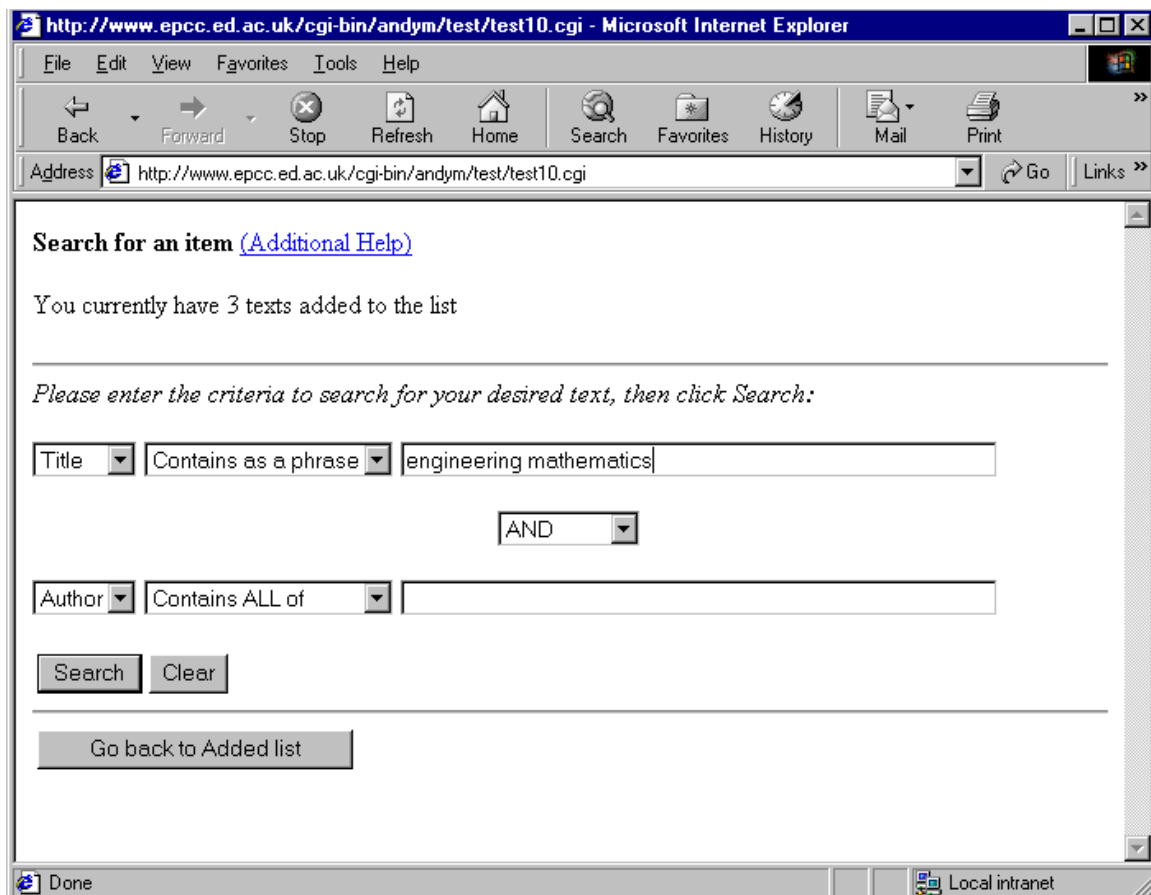


**Figure 2: the added list page. The reading list author selects books returned from the library search.**

This screen could be regarded as the 'base'. Most of the program revolves around getting the required texts into this screen. From the added list, you can move on and search for more texts to add - as many times as required. It is also possible to remove records at this point.

## Search:

When you select books from the library, you are not presented with every text the library owns. You set criteria specifying which books you want to be presented with. For example, you may require adding the book 'Illustrated card games for children'. On the 'search' screen, shown on the diagram below, you might select/enter: 'title contains "card games for children"', which may retrieve a small handful of records from which you select the required texts. There is more information on searching criteria usage at the end of this chapter.



The screenshot shows a Microsoft Internet Explorer window with the address bar displaying `http://www.epcc.ed.ac.uk/cgi-bin/andym/test/test10.cgi`. The page content includes a search form with the following elements:

- A header section: "Search for an item ([Additional Help](#))".
- A status message: "You currently have 3 texts added to the list".
- A prompt: "Please enter the criteria to search for your desired text, then click Search:".
- A search criteria input area with two rows:
  - Row 1: A dropdown menu set to "Title", a dropdown menu set to "Contains as a phrase", and a text input field containing "engineering mathematics".
  - Row 2: A dropdown menu set to "Author", a dropdown menu set to "Contains ALL of", and an empty text input field.
- A connector dropdown menu set to "AND" between the two rows.
- Buttons for "Search" and "Clear" below the input fields.
- A button labeled "Go back to Added list" at the bottom of the form.

The browser's status bar at the bottom shows "Done" and "Local intranet".

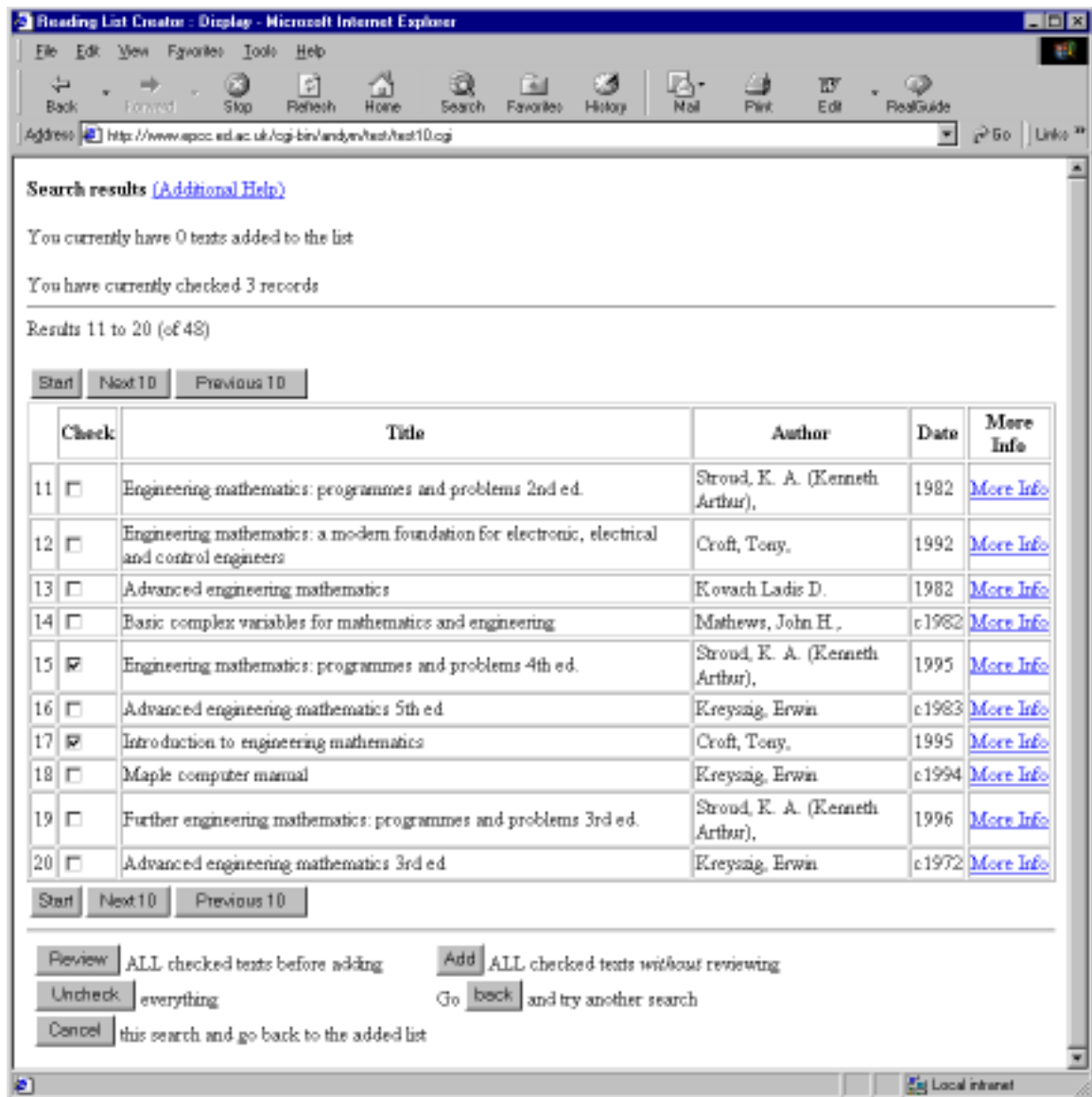
**Figure 3: the search page.**

If the search returns no results *or over 100* entries, then the search screen will be displayed again, indicating that there are either too many texts or none at all. The arbitrary 100-text display maximum is implemented to maintain speed within the program, the more texts returned, the slower the 'select and clarify' screens will become.

For more information on search criteria usage, see the section entitled: *Search criteria usage* at the end of this chapter.

**Select:**

The results of the previous search will be displayed here. Ten records will be displayed at a time allowing you to move to the next ten, previous ten or back to record one.



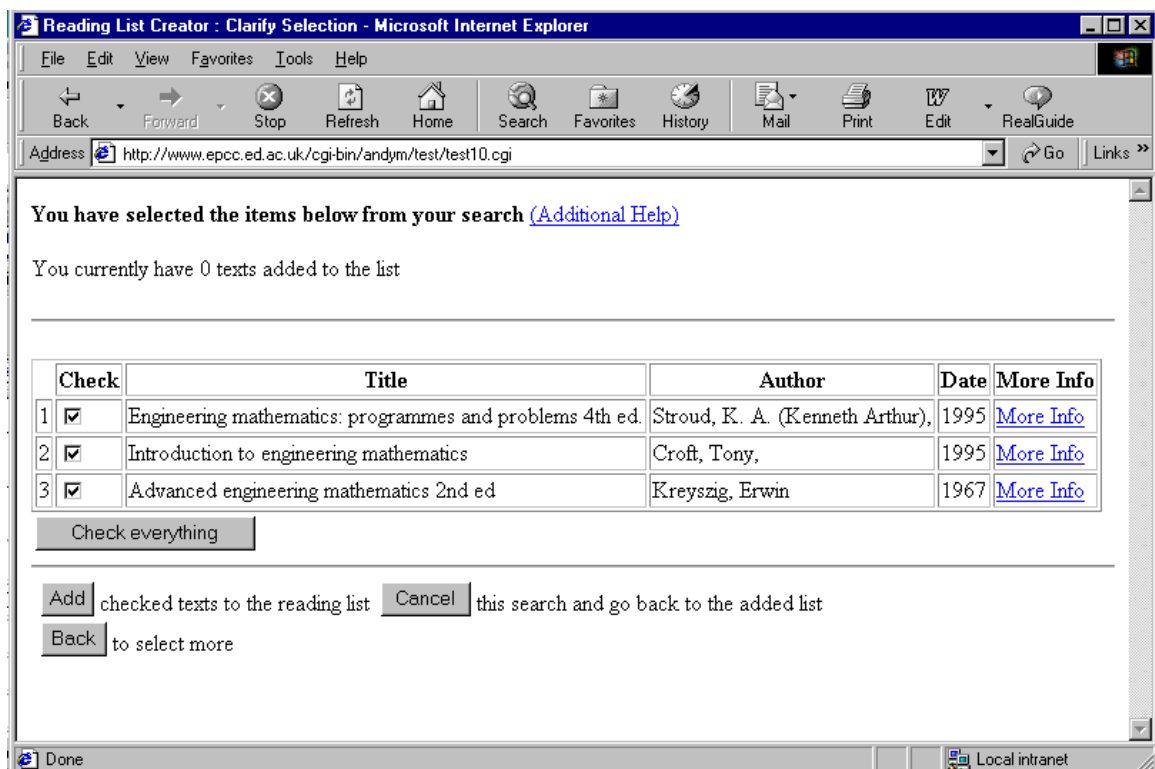
**Figure 4: the select page. Texts 15 and 17 have been selected above.**

Each text has, to its left, a check box. A text is 'selected' when this is clicked and the box is filled in grey or with a cross or tick (browser specific). The text can be 'unchecked' by clicking on the box again. From this screen there are two main options:

- The first is to add all the texts, which have been checked, directly to the ‘added list’. Pressing on the ‘Add’ button will bring you to the ‘added list’ with all the new texts included.
- The second option provides a buffer between the select screen and the ‘added list’. Clicking the ‘clarify’ button will present the user with each selected text. This allows the user to determine quickly if they have missed a text out or added an incorrect item. Ultimately, this allows you to return to the ‘select’ screen without having to redo the search.

### Clarify:

Each text that was checked on the previous search is now presented again. From here you can add all the checked texts to the added list. It is basically a buffer between the ‘selection and added list’.



**Figure 5: the clarify screen allows the reading author to ensure that all the correct texts have been selected.**

### Format:

The format screen serves two purposes:

- allows reordering of selected texts and
- allows annotations to be made to the selected texts.

Up to this point, the order of the added list is simply the order in which the user has added the texts. The 'format' screen, shown below, allows reordering by two methods.

**Reading List Creator : Order and Annotate - Microsoft Internet Explorer**

File Edit View Favorites Tools Help

Back Forward Stop Refresh Home Search Favorites History Mail Print

Address <http://www.epcc.ed.ac.uk/cgi-bin/andym/test/test10.cgi> Go Links

You have selected the 3 items below for your reading list ([Additional Help](#))

Header 1 : 3rd Year Mathematics Reading List

Header 2 : Course F300

#	Title	Author	Date
1	Engineering mathematics: programmes and problems 4th ed.	Stroud, K. A. (Kenneth Arthur),	1995
2	Introduction to engineering mathematics	Croft, Tony,	1995
3	Advanced engineering mathematics 2nd ed	Kreyszig, Erwin	1967

Pre Text :

Post Text :

Ending :

Sort by Author Rearrange the list by the new numbers

Back to the added list Next stage

Done Local intranet

**Figure 6: the format screen.**

The first, and simplest, is sorting alphabetically by author. The second gives more control over the final ordering. To the left of each text is a drop down menu that



initially indicates its position in the list. The first text is assigned number 1; the second is number 2 etc. To move, say, the fifth text to the third position, simply change the fifth text's number to 3. It is possible to reorder several texts in one go. If the program does not interpret the wishes of the reading list author exactly, then simply 'reorder the reordered list' again.

The second use of the format screen is to add annotations. There are two types of annotation supported:

- annotations specific to a single text (for example, 'just read chapters 5 and 9') and
- general reading list 'customisations' (for example, 'Reading list for the autumn term').

There are five specific reading list annotations:

1. Two headers, which are displayed in larger/darker text than the rest of the final page. These are displayed above the 'table' of texts and are expected to contain such things as 'Reading list for course x'.
2. One 'pre text/subheader' field. This is displayed within the 'texts' table but before any actual book.
3. One 'post text' field, which is displayed within the 'texts' table but after any book.
4. One 'end/footer' field, which is displayed after the 'texts' table.

All annotations are stored throughout the program so the user can go back to the 'added list' to search for more texts for example and still keep annotations they have previously added.

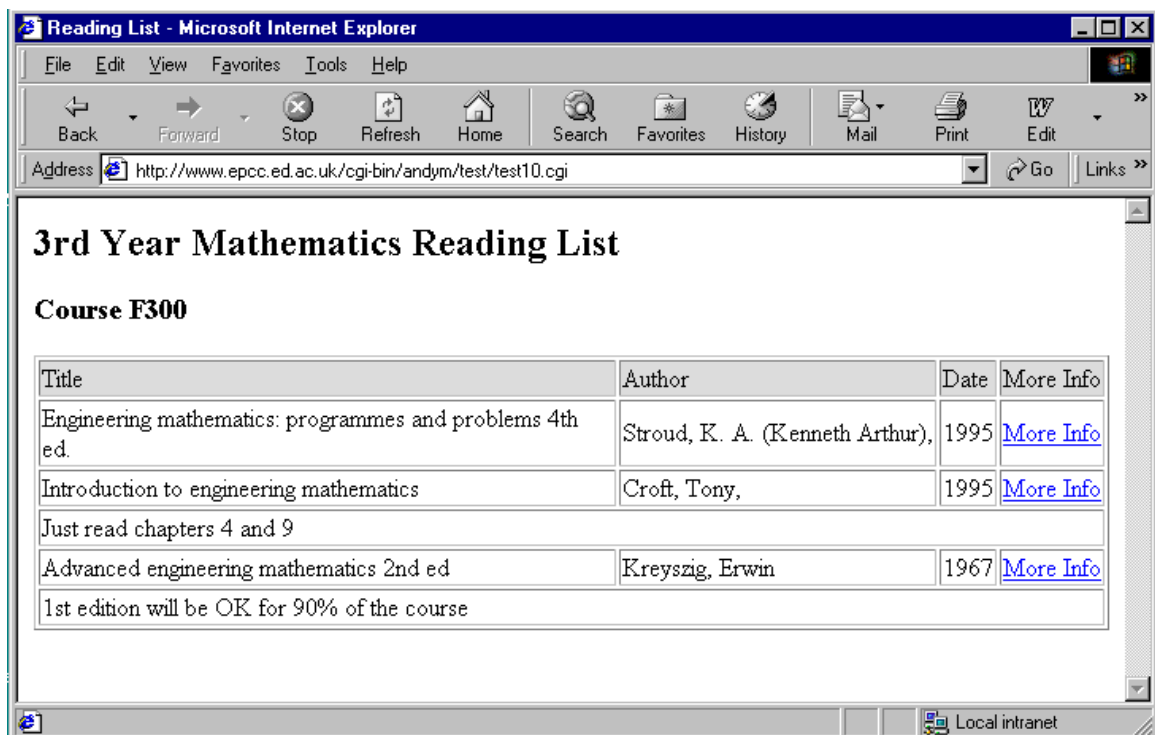
### **Generate List:**

This is the final part of the application. All of the texts selected in the added list must somehow be added to a web page to produce the final reading list page. There are two options:

*Generate web page automatically.* With this option, the program takes all of the text information as well as annotations and then generates a formatted new web page containing this. This page can then be displayed as a whole, independent page on a browser. At this point, the user should save the html page to an appropriate location and create a link from the relevant course web page to the reading list. See Figure 7.

*Generate html.* This will display a page to the creator showing blocks of raw html. Each block contains all the information about each text. These can then be copied and pasted directly into the source of a course web page. The difference here is that no html table is produced and each text stands by itself. This option gives ultimate

control to the site creator allowing them to manipulate the text display at will. This should really only be used by the experienced user. See Figure 8 on page 11.



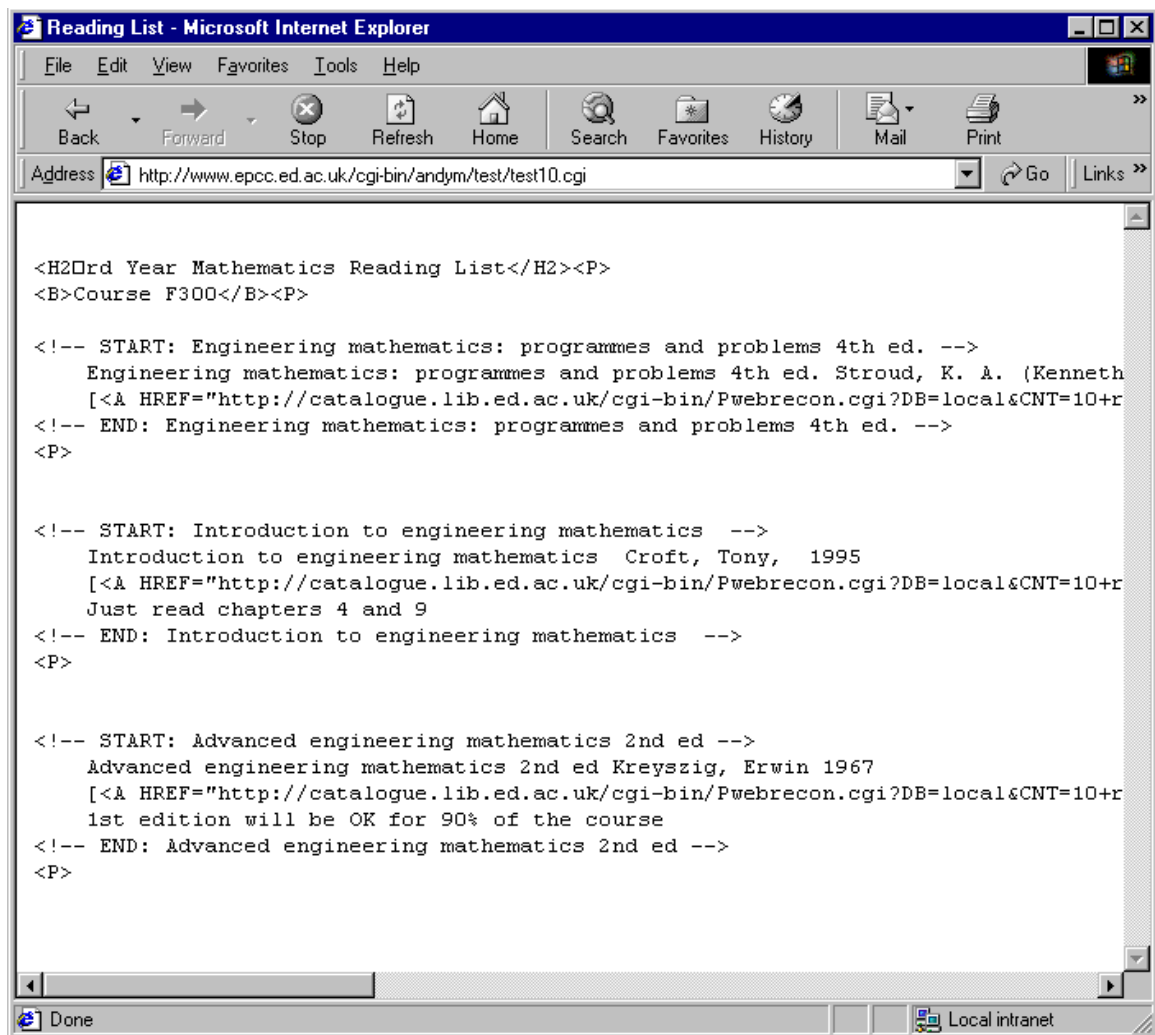
**Figure 7: an automatically generated stand alone reading list page.**

A slight alternative to this might be to generate the web page automatically and then use the html source code of this new page and embed them into other pre-existing html pages.

Each text has three html parts to it (this is only really relevant for the second option, see Figure 8, or users who want to customise an automatically generated reading list page):

1. The text information, i.e. the title, author and date, which is displayed to the student.
2. Holdings information link, the link that the student follows to obtain more information about the book.
3. The annotations.

The *More Info* hyperlink displayed in Figure 7 when activated will query the holdings information of the corresponding book with the library. This can also be done if one uses the html stub output version. Unfortunately, as has already been mentioned, this part of the project could not be implemented using Z39.50 as this information could not be obtained from Edinburgh University Library. If you look carefully at Figure 8 you will see an explicit call to *Pwbrecon.cgi* which is Endeavor's Voyager system. Sadly this is vendor specific.



**Figure 8: stubs of html are presented to the reading list creator. These can be cut and pasted into individual separate pages.**

### Search criteria usage

The text information is stored in a database. Each part of the textbook, such as the title or the author, is effectively stored in a separate part, called a field. So, the name of the author is stored in that text's author field. When you search for some texts, you need to specify the words you are searching for (e.g. 'introduction to') and what field your searching in, e.g. the title field.

The application allows two options for how the 'words' are searched for in the field. You can search as a phrase so the field will contain the words you type in, in the order that they are typed in. For example, entering the criteria in the title field 'catcher in the' will return the book 'the catcher in the rye'.

The alternative to this is searching for every word in any order. So, specifying this in the title field, with the text 'rye catcher' would also return the 'the catcher in the rye'. This method of searching is recommended for searching in the author field. So, for this example, you would search in the author field for 'salinger j d' or 'j d salinger' or just 'salinger'. This method is recommended because of inconsistencies in the storage of author names.

## Getting the Data from the Library

### Z39.50

Z39.50 is a standard for information retrieval, defined by the American Library of Congress [1]. It is formally known as *ANSI/NISO Z39.50-1995 - Information Retrieval (Z39.50): Application Service Definition and Protocol Specification*.

Z39.50 follows the client/server model of computing. In Z39.50 the client is known as the "Origin" and is that part of the local system, which requests information about, in this case, texts in the library. The Z39.50 server part is known as the "Target". It interfaces with the database in the remote system and responds to messages received from the Origin system, such as displaying a certain text.

### ZetaPerl

The Z39.50 standard defines exactly how the client and server communicate. Software modules have been written to allow the programmer to use Z39.50 without having to understand this exact communication protocol.

ZetaPerl is a Z39.50 module that makes it easy for Perl to talk using Z39.50. It provides, through an *object-oriented* interface, all the basic operations offered by Z39.50 protocol to its client.

The main flow of ZetaPerl (for our use) is shown below and, naturally, follows that of Z39.50:

Connect -> Initialise -> Search -> Present

*Connect*: the connect function has two parameters: the host name and the port number.

These tell ZetaPerl who to set up a socket<sup>1</sup> with – in effect this is the basis for the internet. In Edinburgh University Library's case the host is 'catalogue.lib.ed.ac.uk' and the port number is '7090'. Other libraries will have their own host and port numbers through which their public catalogues may be accessed.

This routine essentially returns 'true' if the connection was successful and 'false' if it was not (with a reason).

---

<sup>1</sup> A socket allows two processes to communicate.

*Initialise:* within the context of Z39.50, initialise is a brokering process between the client and server determining how the communication will take place, e.g. to decide which version of Z39.50 they will both support, what functions (such as search) they both support etc.

For a programmer using ZetaPerl, the only two parameters that are of any concern, are the username and password. These are required to access some databases. Edinburgh University Library does not impose this restriction for the public information we require

Again, ZetaPerl indicates whether the action succeeded or failed by the return code.

*Search:* Ultimately this function searches the library database for specified texts. Zetaperl uses four parameters:

1. Database: which database to do the search on. Z39.50 allows searches on multiple databases. Edinburgh University Library's database name is *Voyager*.
2. ResultSet: when the search is performed, the results are stored at the server side, i.e. the library. By naming each set of results, it allows one to effectively have multiple searches 'stored' at once.
3. AttributeSet/Profile: this is a request to the server specifying the type of information the client requires. For our purposes, we need bibliographic information. This includes such things as the text's title and author.
4. Query: the general format of this, in reference to our use, is: 1=4 "videotape", see below for an explanation of this.

In the query example, the initial '1' indicates the 'use' attribute. There are six attributes and this one basically means 'search in'. The '4' indicates the title field (see appendix 3 of the Z39.50-1995 standard [1] for a complete list of fields). What is being searched for is enclosed in quotation marks. So in this example, we are 'searching in the title field for the term videotape'.

Z39.50 allows complex queries to be formulated by using Reverse Polish Notation. ZetaPerl hides this part through its interface and allows query's to be formed in a more natural fashion. Here is an example (note Z39.50 is case insensitive and ignores punctuation, etc completely):

1=4 "dictionary of biology" @AND@ 1=1003 "stockley"

This simply means, search for all texts with 'dictionary of biology' in the title field and 'stockley' in the author field.

For Perl use, this query would be stored in a scalar variable thus:

```
$query = "1=4 \"dictionary of biology\" \@AND\@ 1=1003 \"stockley\"";
```

Four operators are defined in the standard: AND, OR, AND-NOT and Prox. Prox is short for 'proximity' and allows very complicated searches, enabling the user to specify how close the words physically are from each other. This is not used in the application. An example of AND is used above. If AND were replaced with OR, then a search would be done returning all records with 'dictionary of biology' in the title, or 'stockley' in the author as well as both. ANDNOT would return any text including the said title but not written by 'stockley'.

More complicated searches can be separated in parenthesis, thus:

(1=4 "computers" @AND@ 1=4 "people") @ANDNOT@ 1=1003 "norton"

The search function again returns a value indicating if the request was successful or not. Importantly, it also returns a value indicating how many records were found on this search.

*Present:* This final function asks the server to give the bibliographic information about the texts from the result set. ZetaPerl requires 5 parameters:

1. Resultset: the resultset used in the search.
2. Howmany: how many records to return
3. Start: which record number to start returning from (starting at 1)
4. Format: allows the client to request how much information is passed back. For example, 'full' gives all the information that the server has. This is what we use.
5. Syntax: the way in which the information is returned. Edinburgh University Library only supports the USMARC syntax (explained below), but other options are available.

Present, again, returns if the request was successful or not. It also returns the 'number of records returned' – this should tie up with the number of records requested but if there are only 10 texts and 100 are requested, then the number returned will be different. It also returns all records requested. In Perl, this is in the form of a list with one record per element.

### **Bibliographical Information**

Records are returned following the syntax requested by the 'search' function. To obtain the title etc, information, BIB1 must be selected for the 'profile' parameter. The format in which the data is returned is specified in the 'syntax' parameter within 'present'. Edinburgh Libraries supported format is USMARC. Z39.50 is an American standard and hence most libraries support USMARC, although UKMARC is also defined.

The differences between MARC formats are slight and they all follow the same sort of rules. The MARC format starts with some header information specifying such things as the record length. Each line then contains a 3 digit numerical tag indicating what that

line represents and then that line's actual data. Here is an example of the format of a returned record:

```

1.2.840.10003.5.10&00663nam 2200241 4500
001 681470
005 19990902141954.0
008 990114s1995 enk 0011 eng|
015 $a b8712932
020 $a 0333620224
035 $a 0333620224
035 $9 LCN10929773
049 00 $a E
050 4 $a TA330
100 1 $a Stroud, K. A. $q (Kenneth Arthur), $d 1908-
245 10 $a Engineering mathematics:$b programmes and problems/$c K.A. Stroud.
250 $a 4th ed.
260 $a Basingstoke : $b Macmillan, $c 1995.
300 $a xxiii,1032p ; $c 24 cm.
500 $a Previous ed.: 1987.
650 00 $a Engineering mathematics $x Programmed instruction.
989 00 $a 51
990 00 $a 62

```

The line beginning with code 245 represents the book title. The following number 10 gives a bit more information – in this example it tells us that this is all of the title and that there are no characters that come before the title, such as *a* or *the*, which should be ignored for sorting purposes.

The main differences in the MARC formats are, simply, what each numerical tag represents, 450 might be publishers name in one format and the number of pages in another. The USMARC (also called the MARC or MARC 21 format) defines tags up to 889 and after this number (890+) the library can define its own tags. For instance 989 and 990 are used in the above example. If any of these 'internal' tags are used within the program, then it becomes more vendor specific. Thankfully, we only need such information as author and publisher, which are defined in the standard. More information about MARC formats can be found at [5].

## Holdings Information

Z39.50 defines a method of retrieving the current holding information. When requesting a search, instead of specifying 'bibliographic information', you simply specify 'holdings information'.

At the time of writing, we could not obtain the holdings information from Edinburgh University Library using Z39.50. Other libraries, such as Dundee University and Aberdeen University, return the holdings information as an extension to the original MARC format, when bibliographic information is requested. To allow the program to display this information, an alternative method had to be sought.

As mentioned in the 'Current System And Program Objectives' chapter, the library currently has a web based search facility. This uses CGI calls to a script called PWEBRECON.CGI. It is possible to use this script by calling it with the required GET

request parameters on the URL. The method used to get the holdings information is implemented by performing a search on the exact title and author and the returned page contains all information about the text.

It is also possible to specify a date value. If this is done, then only those texts printed at date x will be displayed. This is not desirable as the library may carry a reprint, which would not be returned but is still relevant to the student. Endeavor does not seem to support 'greater than operators', so you can't request all dates after 1990 for example.

A workaround is to parse the returned web page and only display the relevant texts.

The general URL format to use on pwebrecon is as follows (all in one line though):

```
http://catalogue.lib.ed.ac.uk/cgi-bin/Pwebrecon.cgi?  
DB=local  
&CNT=25+records+per+page  
&CMD=QUERY
```

The first part tells the browser where pwebrecon.cgi is; DB specifies using the local database; CNT is the number of records to display at one time and CMD tells the database what to do, in our case search. QUERY specifies what to search for.

An example of the format of 'query' is shown below (note that the whole URL string must be formatted accordingly to be able to be passed through a URL, i.e. spaces are replaced by +s, etc.). Perl's *CGI:escape*, yet another Perl module which in this case facilitates the writing of CGI scripts, handles this, see [6].):

```
QUERY = tkey+physics+AND+nkey+tipler
```

This, in English, is read as – search for texts containing 'physics' in the title field (tkey) as well as 'tipler' in the author field (nkey). For more details on pwebrecon usage see [7].

## Program specifics

### Form Parameters

The user interacts with the program via web based HTML forms. HTML forms allow the user to enter information, such as the search query, and then submit the form to a CGI script giving the script all of the form's information. The script then acts on this and normally displays another web page, perhaps containing another form.

If, for example, a user were at the 'added list' part with all of the texts they wish to add to the reading list displayed in front of them, if they moved on to the next form, this information would be lost. Hence, a method of 'remembering' the entries (keeping state) must be implemented. There are many ways of doing this, cookies or creating files at the server side etc are all possible. The method chosen was to save all of this information



within the form as hidden entries. These are form entries that are passed to the CGI script but not displayed to the user. This is probably the simplest way, but it also means that any information that must be ‘remembered’ has to be passed when the form is submitted and created – increasing transmission time.

Every text that has been added to the reading list, is stored in each form as a hidden parameter, see below. The parameter name is ‘addedx’, where x is the text number (starting from zero). There is also the parameter ‘no\_of\_added’, which stores the number of added texts.

```
no_of_added = "2"
added0 = "1.2.840.10003.5.10%2600663nam%20%202200241....."
added1 = "1.2.840.10003.5.10%2600965nam%20%202200289....."
```

The value of the parameter of ‘addedx’ contains all of the information returned by the ‘present’ function on that text. A Perl module, called CGI.pm [6], is used to encode the text’s information into a format that can be used accordingly within the HTTP protocol. This is the reason for characters such as %20 appearing where a space would be in the above example. The two functions used from CGI.pm are ‘escape’ to encode the data and ‘unescape’ to decode the data.

## Program Internals

Each time the program script is called from a form, it uses the hidden form parameters to fill a Perl list, called @added, with each text. This list is a global variable and the texts are in their unescaped form.

The same process is used to store all of the records retrieved from the search. These are stored in the global list, @all, and are passed with the parameters ‘recx’.

The annotations are passed in a similar fashion. Each text’s annotation is passed as the hidden parameter ‘ann\_textx’ and each ‘general’ annotation is stored as ‘ann\_head1’, ‘ann\_posttext’ etc. If the user has not entered an annotation for ‘text4’ for example, then there will be no ‘ann\_text4’ parameter. When the script is called, all of these annotations are stored in a global associative array. The keys are the same as the form’s parameter but without the ‘ann\_’ prefix.

Every time the script is called, it uses the hidden parameter ‘from’ to decide which form it was called from and acts accordingly. Each form has this parameter with the value set to the form’s name, for example search or format.

## Future Recommendations

As with all programs, there is room for improvement. Below is a list of the implementer’s ideas as to how the script could be improved:

- Get holdings information by using Z39.50 thus allowing it to be displayed at creation time and thus become more vendor independent, i.e. not using Endvours pwebrecon.cgi.
- If pwebrecon continues to be used, then perhaps parse and reformat the web page.
- Use frames to store record information rather than passing it through each form, or even use cookies to achieve the same results. The user could turn off both of these however.
- Make the 'more information' link use more of the MARC format.

There may be scope for some of the current functionality to be implemented using JavaScript rather than CGI. This could then be done at the client end and not necessitate a further communication with the server hosting the script.

### **Keeping state**

Probably the most useful future implementation would be a feature to allow the reading list to be updated rather than completely recreated. For this to happen, the program needs to know which texts are already in use. Three methods will be briefly outlined here:

*Server side database.* The texts would not be displayed within the course page but obtained by following a link from that page, which runs a script to access the database and dynamically generate the list. There are security issues, such as ensuring that a lecturer can't edit the wrong list, and students certainly can not edit a reading list. There are also storage issues.

*Client side database.* This could (as above could also) consist of a single file per reading list. There are issues with the application writing to the course sites computer system.

*Texts stored within the web page,* and a method of retrieving these texts at a later date directly from the records stored in an html page. An author has the freedom to format the texts as they wish, but then there is a high risk that they may not be able to read the reading list data back in to the CGI script, because of the increased difficulties in parsing this data back into the system. It would also be difficult for the script to update the course reading list.

It was decided to use the third method, although time did not allow this the list to be retrieved at a later date. It is thought that this would be implemented by storing, as HTML comments, all required information about each text within the web page. These would then be retrieved and the reading list generated.

If the creator had used the 'complete web page' option when generating the page, rather than using raw html, then the new page could simply overwrite the old one, and the format would be the same.

Preliminary design used the following format for storing all general annotations:

```

<!--##READING LIST:VER 1:start BLOCK A##-->
<!--##These blocks must be kept in tact to keep state##-->
<!--head1:(escaped first header)-->
<!--head2:(escaped second header)-->
<!--pretext:(escaped pretext)-->
<!--posttext:(escaped posttext)-->
<!--ending:(escaped ending)-->
<!--##end BLOCK A##-->

```

And for the textual format:

```

<!--##READING LIST:VER 1:start BLOCK 0##-->
<!--##These blocks must be kept in tact to keep state##-->
<!--title:(escaped text)-->
<!--author:(escaped author)-->
<!--date:(escaped date)-->
<!--annotation:(escaped annotation)-->
<!--##end BLOCK 0##-->

```

Some code has been written to parse this information from an html page but additional supporting code will have to be written to make this effective. It is thought that to make this feature useful, because there is an issue that the creator could accidentally change the data, there should be error checking within each block. This could take the form of a code indicating how many texts are expected, which have annotations etc.

This data would need to be formatted into the MARC format when read in, to allow the final holdings information code to work. Alternatively, all of the texts information as returned by the present function, could be stored in the page.

The decision to store this 'keeping state' information within the HTML page was done at a time when the annotations were not a feature of the program. Storing the texts within the page allows the creator to add such things as 'read chapters 4 and 7' to each text, which was a large part of the decision. Now that this feature is implemented within the format screen, the author slightly favours using a single file for each reading list. The file would be stored at the client end, wherever the creator wishes it (option two). This decision is based only if the problem of the program writing safely to the client sides computer system can be solved.

This system would not allow much alteration of the list but it is thought that this would not be a big issue. Also, it would not allow the texts to be within the course web page, so that even for a single book, the student would have to follow a link.

Perhaps a completely different idea is needed.

## Conclusion

The current system used for course web site creators to add a reading list or list of texts to refer to, is either by typing them in manually or copying then pasting from the library's current web based search engine. The first option is slow and error prone and the second

is also slow and requires more computer skills but has the main advantage of the lecturer knowing if the library carries that book or not

The project set out to create an integrated environment where the site creator could search the library database for texts, select the required ones and have the reading list created for them. When the student visits the course site, they can also obtain current holdings information about the text.

From the user side, the program must be simple to use as well as fast, else it will be easier to just type in the texts.

From the programming side, it must be easily maintainable and vendor independent. To this end, the method adopted to access the library database is by using the Z39.50 standard, which is supported by many libraries. We were unable to use Z39.50 to obtain holdings information from Edinburgh University Library's database, so their current web based system was used for this.

As the program stands, it is in a useable form, creating the reading list from start to finish.

## References

1. <http://lcweb.loc.gov/z3950/agency/markup/markup.html> (Z39.50 standard).
2. <http://zeta.tlcp.i.finsiel.it/z3950/zetaperl/> (ZetaPerl).
3. [http://wp283.lib.strath.ac.uk/Z39\\_50list.html](http://wp283.lib.strath.ac.uk/Z39_50list.html) (A list of Z39 supporting Universities).
4. <http://www.endinfosys.com/> (Endeavors web page).
5. <http://lcweb.loc.gov/marc/bibliographic/ecbdhome.html> (MARC format).
6. [http://stein.cshl.org/WWW/software/CGI/cgi\\_docs.html](http://stein.cshl.org/WWW/software/CGI/cgi_docs.html) (CGI escape).
7. <http://madcat.library.wisc.edu/help/cannedsearches.htm> (PWEbrecon) .

## Acknowledgements

Finally I would like to thank Wilma and John from Sellic library for giving a student the chance to make a real program. I'd like to thank Gordon for his outsiders' base, giving fresh ideas and thoughts on everything while not allowing me to get away with anything. Everyone at EPCC, especially Catherine for organising the SSP program, but yet still making it fun. Everyone within the SSP for making sitting in the Xterm room for hours not seem like hours. And most importantly Mario, who not only helped *so* much with the program but also became a good friend.