A WINDOWS GUI FOR INTENSIONAL WEB AUTHORING

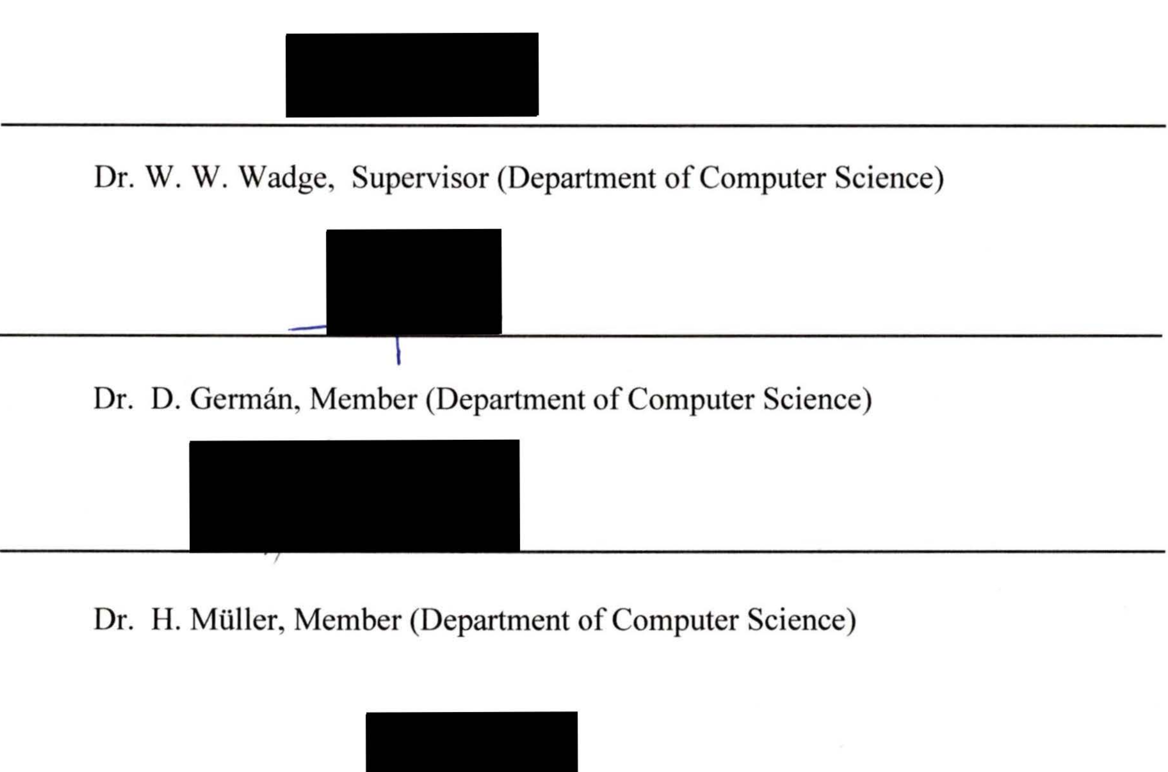
By

Qin Zhu

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
in the Department of Computer Science

We accept this thesis as conforming to the required standard



Dr. W. W. Wadge, Supervisor (Department of Computer Science)

Dr. D. Germán, Member (Department of Computer Science)

Dr. H. Müller, Member (Department of Computer Science)

Dr. P. Driessen, External Examiner (Dept. of Electrical & Computer Engineering)

@ Qin Zhu, 2001
University of Victoria
All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy
or other means, without permission of the author.

Supervisor: Dr. W. W. Wadge

ABSTRACT

This thesis introduces a Microsoft Windows GUI program IWAG (Intensional Web Authoring GUI program) that provides an easy way for even unskilled computer users to produce multi-versioned Web pages, particularly those using *pop/chop* text on a Microsoft Windows platform.

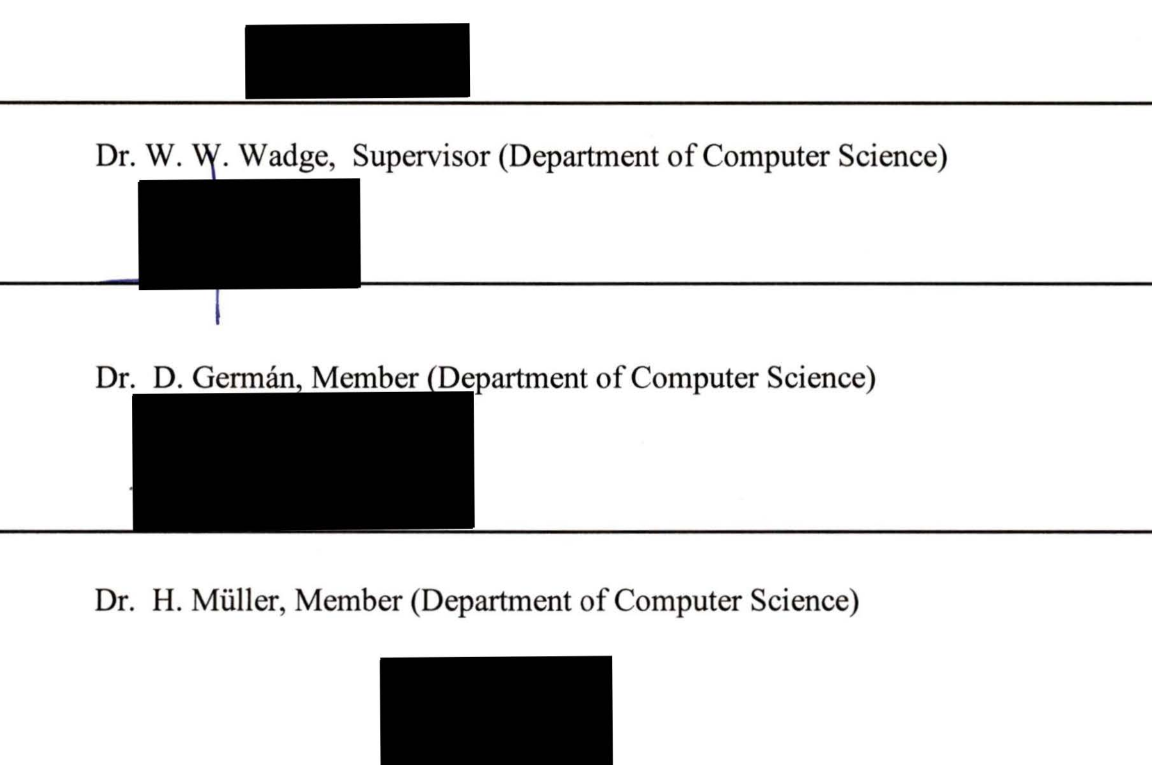
Pop/chop text is a term to describe those multi-versioned Web pages that can display or hide certain paragraph texts according to the user's intensional URL request. ISE, a Perl-like scripting language, uses conditional constructs to output such requested versions of (either pop or chop) HTML files on the fly. On a UNIX platform, IML (a set of *groff* macros) can be used to generate an ISE program for the *pop/chop* text. On a Microsoft Windows platform, IWAG can be used for this purpose instead.

IWAG has two advantages. Firstly, for the average user (Web page author), the Microsoft Windows GUI makes this program much easier to use. Users needn't worry about IHTML, ISE or IML syntax anymore, they can simply type or paste several sections of their text and select pull-down menu items. IWAG will generate an ISE program for the multi-versioned Web pages service. Secondly, for the programmer, although it required a significant initial investment in Microsoft Windows programming, using IWAG avoids defining intricate document macros as in IML. It can also be applied to other (non *pop/chop*) uses by replacing the ISE resources, which are stored in the program, and slightly modifying the program itself.

We have built a bilingual Web site in IHTML using IWAG to demonstrate the advantage of this language. In order to make it easier to create multi-version Web pages,

ISE and IML were also used. In the final analysis, IWAG turns out to be a fairly good solution, although it still needs more improvement.

Examiners:



Dr. W. W. Wadge, Supervisor (Department of Computer Science)

Dr. D. Germán, Member (Department of Computer Science)

Dr. H. Müller, Member (Department of Computer Science)

Dr. P. Driessen, External Examiner (Dept. of Elec. & Comp. Engineering)

CONTENTS

Chapter 1 .... **1**

Introduction **1**

1.1 Motivation, Problem, Approach 1

1.2 Thesis Overview 7

Chapter 2 .... **8**

Background **8**

2.1 Intensional Logic and Version Control 8

2.1.1 Intensional Logic 8
2.1.2 Intensional Programming 9
2.1.3 Version Control 10
2. 2 Intensional HTML 12

2.2.1 Versioning the Web 12
2.2.2 IHTML Syntax 14
2.2.3 Best-Fit Algorithm 19
2.3 A Bilingual Web site Sample 20

2.3.1 Simplest Bilingual Example 20
2.3.2 Switching between two versions 23
2.3.3 Nest pop up 25
2.3.4 Stretchtext 28
2.4 From IHTML To ISE 33

2.5 From ISE To IML 37

Chapter 3 .... **41**

Microsoft Windows GUI Program For ISE **41**

3.1 Design IWAG 41

3.2 Advantages And Disadvantages 42

Chapter 4 .... **43**

Microsoft Windows GUI Program Implementation 43

4.1 Microsoft Windows Programming 43

4.1.1 How does Microsoft Windows work? 43
4.1.2 Why the Win32 API was chosen 44
4.2 The Multiple-Document Interface 46

4.3 The Program Structure 47

4.3.1 Introduction 47
4.3.2 MENU Resources 48
4.3.3 "TEXT" Resources 49
4.3.4 WinMain Procedure 49
4.3.5 `FrameWndProc` Procedure 50
4.3.6 SectionWndProc Procedure 52
4.3.7 `edit` Window 54
4.3.8 ISEWndProc Procedure 54
4.3.9 OrgDlgProc 57
4.3.10 TitleListView 59
4.4 Some Difficulties With The Technique And Solutions 59

4.4.1 Data Storage 59
4.4.2 `Printf` Statement 62
4.4.3 Title and TitleList 65

Chapter 5 67

Conclusions and Future Work 67

5.1 To Generate Other ISE Files 67

5.2 Integrating Different Functions 68

Bibliography 70

Appendix A: 71

Appendix B: 74

Figures 74

FIGURES

Figure 1: Switching between English and Chinese Web pages 24
Figure 2: tangsi.html in the vanilla version 74
Figure 3: tangsi.html in the pop1:on version 75
Figure 4: wangep.html in vanilla 76
Figure 5: wangep.html in the pop131:1version 77
Figure 6: wangep.html in the pop131:1+ pop1311:1 version 78
Figure 7: firstd.html in vanilla version 79
Figure 8: firstd.html in pop2: on version 80
Figure 9: pop/chop text in ISE 81
Figure 10: IWAG for pop/chop text. There is no child window in the client window at the beginning 82
Figure 11: A dialog box pops up to ask the user to input the section title 82
Figure 12: The *section window* includes an *edit window*, in which a user can input his/her section text 83
Figure 13: There are six *section windows* in the client window. 83
Figure 14: The organizer dialog box provides the title list for the user to select options 84
Figure 15: The result – an ISE file is output into the *ISE window*. There are nine *section windows* in the background in the *client window* 84
Figure 16: Window Programming Mode 85
Figure 17: The MDI parent-child hierarchy in IWAG 86
Figure 18: menu resource 87
Figure 19: “TEXT” resource 87

ACKNOWLEDGMENTS

I would like to thank everyone who supported me and offered me guidance throughout this research. In particular, the advice of Dr. W. W. Wadge, who encouraged me to pursue research in a variety of directions and inspired me to find the topic of this thesis, was much appreciated. I would also like to thank Paul Swoboda, who gave me a lot of guidance. Without his excellent work in IHTML3 and ISE, most of my research work could hardly be done.

I am also grateful for the financial support from the University of Victoria and Dr. W. W. Wadge.

Finally, I wish to thank my family and friends for all their encouragement.

Chapter 1
Introduction

1.1 Motivation, Problem, Approach

"Diversity is an integral part of community. In a real community, different members are allowed different points of view and the community is the result of sharing these points of view. Computer systems have a reputation of being hostile to diversity, of imposing a single uniform reality on all their users. Most systems are still at the level of Henry Ford's model T, offered (in Ford's own words) in 'any color you want, as long as that color is black'" [1].

Wadge argues convincingly that computer systems should be more flexible. In other words, computer systems need "diversity engineering." He also quotes John McCarthy's article "Information" in the book *Information* from 1966: "The speed, capacity and universality of computers make them machines that can be used to foster diversity and individuality in our industrial civilization, as opposed to the uniformity and conformity that have hitherto been the order of the day." This suggests that this diversity has been foreseen and anticipated for a long time.

One "modest step" towards making practical computer-supported diversity is in Web site development. "In theory, Web pages, being purely electronic, can be updated frequently and provided in customized versions on demand" [1]. Thus, the "*version*" concept was introduced in Web authoring and multi-versioned Web sites.

However, there are few examples of multi-versioned Web sites existing so far. This is because the maintenance work was more difficult than expected. Brown, the author of IHTML 2, described this situation: "Different versions quickly diverge in more

than the appropriate ways, as fixes are made to one version but not another … , the number of versions which must be maintained may increase dramatically, making the maintenance problem that much worse" [2, page 1]. In fact in the old days, the authors of these Web sites used to clone and edit new versions and store them separately. The result was duplication of information that occupied extra storage space, and the parallel pages were hard to keep consistent.

The straightforward solution is to share the duplicated information. Unfortunately, the HTML language itself offers little support for this purpose. Pages can only be shared clumsily by links and, furthermore, parts of the pages are difficult to share. This means that something more than standard HTML should be used to meet this demand.

Intensional HTML, recently built by Wadge's group, provided a solution to this problem. It allows the Web site author to treat the pages or some parts of the pages, which are the same from version to version as generic. By these means, pages and the components of the pages can be shared. "When a request for a particular version of a page is received, the source for that particular version is assembled from the results of specializing the relevant source files for the components" [1].

Intensional HTML extends the conventional HTML syntax. The component of a page can be chosen by the particular version expression, which is provided by special page requests. These requests not only include a conventional URL, but also include a version expression that provides the version information.

The first version of IHTML was implemented by Yildirim using a CGI program. He also built a bilingual Web site to demonstrate his work. Although it is a good start, the

implementation has performance problems—very slow loading processes. As a result this version was not widely used.

The second version, IHTML 2, was implemented by Brown using the plug-in model of the Apache server. It solved problems such as "how best to present multi-versioned information" and "what features IHTML should have to make the site implementer's job as easy as possible." Later, Liu extended IHTML 2 and implemented Temporal HTML (THTML) for "authoring and maintaining *time-sensitive* Web sites" [13, page 1]. THTML adds a time dimension to the IHTML version space and specifies more time related version control algorithms to eliminate temporal version ambiguity [13, page 5]. However, these implementations were complicated and not easily reusable by other programmers. Based on Brown's work and experience, Swoboda rewrote the whole plug-in and provided a very elegant implementation with Lex and Yacc. An IHTML server for real world users was born.

So far, no real Web site written in IHTML has been built. To explore the usage of this new language, the author built a bilingual (English-Chinese) personal Web site as a research experiment. In the course of this research, it was revealed that *pop/chop text* (text that can be shown or hidden using options) is an extremely useful feature, because it supports diverse needs of readers.

On the other hand, some drawbacks and limitations were revealed as well. It proved that writing a big Web site in IHTML needs a lot of skill and patience. Writing a *pop/chop* section, especially a nested one, is very complex and tedious work. When adding more sections of text into the same page, the author of the page has to repeatedly clone and edit similar layouts within the page. The reason for this is that IHTML is not a

programming language. Besides a simple conditional construct (**`iselect`**), there are no provisions for evaluating expressions, no iterative constructs and no functions [1].

Fortunately, an alternative was found called ISE (Intensional Sequential Evaluator), a Perl-like scripting language devised by Wadge and Swoboda. ISE is not only a real programming language, but it is also versioned. "Briefly, this means that all entities in the language (variables, arrays, functions, etc.) can be versioned; programs execute in an implicit context which determines which versions of the relevant entities are used; and there are also explicit mechanisms for accessing and modifying the context" [1]. When this author was building the bilingual Web site, the ISE plug-in had just been programmed. Based on experiences with the bilingual Web site and the ISE plug-in, Wadge found an alternative way to produce an HTML file - using the "**`print`**" statement in ISE. Using the conditional construct (**`vswitch`**), ISE can output certain version HTML files at runtime. According to different values of the version parameters following the URL, conditional construct determines different versioned HTML file sections being "printed" out.

Although the limitations of the IHTML have been remedied, ironically the difficulty for Web site development is more severe. Due to the complexity of the ISE language itself, few Web site authors will be skilled enough to program their site in ISE directly.

The object of all of the previous effort is to make it easier for a Web site author to program multi-versioned pages. Considering that we only use a small portion of ISE for particular use in Web site development and usually the same section of an ISE program is used for a similar purpose such *pop/chop text*, why not just pack these ISE program

sections as off-the-shelf components and let the Web site author organize them rather than program them?

Wadge provided a solution by programming "a set of macro definitions, which translate text with high-level (intensional) markup into ISE source." He explained this method: "For the time being, we use the macro facility instead of *groff*, the GNU upgrade of the (old fashioned) *troff* macro package included with every UNIX system. We use *groff* strictly as a macro processor; we run it with formatting disabled and with ASCII output selected" [1]. There is no doubt that using a set of macro definitions is much easier than directly programming ISE itself. This brings the goal of easy-to-use Web site development closer. However, remembering the make-up macro definition and which set of them to use for what particular use still involves a steep learning curve. On the other hand, "the definitions of the macros for the intensional features can be rather intricate" [1]. Furthermore, "the realistic droptext macro definitions we actually use are more complex, because we allow any number of retractable sections, arbitrarily nestable" [1]. As a matter of fact, only skilled *groff* programmers can accomplish these definitions. Writing a new set of macro definitions could be an extremely intricate task. Therefore, this *groff* tool method has considerable development and maintenance difficulty itself.

Is it possible to find an easier way for both Web site author and tool developer? The answer is yes. Microsoft Windows is an ideal tool for this propose. "Windows possesses a graphical user interface (GUI), sometimes also called a 'visual interface' or 'graphical windowing environment'" [3, page 6]. Since GUI is visual to the user and easier to use, "it is now quite obvious that the GUI is (in the words of Microsoft's Charles Simonyi) the single most important 'grand consensus' of the personal-computer industry"

[3, page 6]. Obviously most unskilled computer users will feel happier using pull-down menus rather than remembering a set of unfamiliar macro definitions.

On the other hand, as a high level language, C++ is more powerful than the *groff* macro processor. There is no doubt that C++ can be used to replace *groff* as a parser in the new method. Therefore, some intricate macro definitions could be avoided when a new set needs to be defined. Although extra effort will be needed in the new Microsoft Windows GUI item design, adding new features will be easier to most programmers with C language background.

To demonstrate this method, a Microsoft Windows GUI program IWAG (Intensional Web Authoring GUI program) for *pop/chop text* has been designed. Not only is it a demonstration for the purposes of research, but it also provides a platform for future work. It can be used to replace other sets of macro definitions by modifying the menu items and changing the ISE resource files. Furthermore, it can be used to produce IHTML programs by changing the resource files from ISE to IHTML. In the future, different functions (*slide show*, *pop/chop text*) can be glued together and more complicated GUI integrated programs could be built.

In developing this program, an advanced technique in Microsoft Windows programming called Multiple-Document Interface (MDI) has been used and some technological difficulties have been solved. To make the program smaller and platform-independent, the API was used directly. Compared to using Microsoft Foundation Class library (MFC), it is a lower-level method and gives the programmer more control although it is tedious. After all is said and done, this implementation is still primitive and a lot of improvement is still needed.

Before this thesis was finished, Phong made some progress with a different approach using other common and existing technologies, such as XML, XSL, Java, Java Servlets, and Cocoon, which is called IXML Although this approach is still primitive and doesn't even include the best-fit mechanism [14, page 68], the research work is promising. In the future, multi-version Web page examples provided in this thesis, such as multi-language pages and *pop/chop* text, could not only be implemented in IXML, but also be more organized [14, page 2].

1.2 Thesis Overview

Chapter 2 provides some background information including: intensional logic, the versioning approach adopted in IHTML, intensional HTML, ISE and IML. The bilingual Web site sample that the author built is described as well. Chapter 3 gives the design of the Window GUI program for *pop/chop text* and the interface and dataflow diagram are described there. In Chapter 3 some advantages and disadvantages of this design are also discussed. Chapter 4 provides some basic knowledge about Microsoft Windows and MDI programming and the structure of IWAG is described. Some difficulties and deficiencies are described there and solutions are presented as well. Chapter 5 discusses some future work including how to generate other ISE files and how to integrate different functions.

Chapter 2
Background

2.1 Intensional Logic and Version Control

2.1.1 Intensional Logic

Rudolph Carnap addressed the problem of formalizing natural languages. He observed that many (perhaps most) natural language expressions are not statements in the logical sense: they do not have truth values in and of themselves [4, page 2]. For example, if someone claimed that "Last year's Ford is the most popular car in this country," we can hardly judge the truth or falsity of the sentence even though we know that between 1991 and 1995, Ford was the most popular car in the U.S. In order to have a truth value assigned, these sentences require a context. Only when the time and place in which this particular sentence was uttered is known can the conclusion be reached. Although the sentence itself does not have a single truth value, there is a unifying concept underlying all statements of the form "At Year T and Country C, last year's Ford is the most popular car in this country." Carnap called this unifying concept an intension [4, page 23].

"Intensional logic is a mathematical formal system for describing entities whose value depends on implicit contexts" [5]. In intensional logic, the basic concept is "possible world semantics" purposed by Montague [6, page 41]. Possible world semantics refers to an entity that varies over a space of indices—its implicit context. In the above example, "*last year*" could be any year between last year and the first year this model car was made. It depends on when "*this year*" is and usually we can get it from the context.

To evaluate context sensitive expressions like the above example, the distinction between what is called the intension and the extension of the expressions should be made. "The extension of an expression indicates the value in a given context" [7, page 17]. For example, when someone surfs a Web site such as **`http://carpoint.msn.com`** to check the estimated value of a used car, he should provide the *make*, *model*, and the *year* of manufacturer. The contents of the estimated value depend on three parameters: *make, model* and *year*. In fact, the contents can be expressed as a mathematical expression of the form *estimate_value(make, model, year)*. Thus, *estimate_value(Ford, Mustang, 1998)* is the extensional value of the expression which denotes the estimated value of a 1998 Ford Mustang.

In natural language, there are many situations in which the semantics of an expression rely on implicit context, such as "last year's Ford is the most popular car in this country" where the semantic relies on the value of "*last year*". By an *intension*, we mean an indexed family of entities that relies on the implicit context [5] such as what time (*year*) we use the above expression.

"The intension of an expression represents all the possible extensions and also defines the relationship between these extensions and their contexts. In other words, the intension of an expression can be regarded as a mapping function which assigns a value to the expression in a given context" [7, page 17].

2.1.2 Intensional Programming

Programming based on the paradigm of intensional logic is called *intensional programming*. "The main characteristic of intensional languages is that they have

context-switching operators, called intensional operators. These operators can combine the values from different contexts without explicit context manipulation" [7, page 18]. The World Wide Web can be considered the first large-scale experiment in intensional programming. The Web consists of a family of pages indexed by their URLs. The intensional operators of the Web are the hyperlinks; since they switch the context to the page they link to. If some images must be filled to complete a page, separate connections might be made to retrieve the images, possibly from a different host.

The Web can be perceived as a family of pages, which requires a family of implementations—a family of HTML files. There is no need to produce separate implementations since these files are already members of a family, and have strong family resemblances. In fact, the entire Web can be considered versions of a single Web page and related members can share code. Therefore a version control system, which allows code sharing among versions, is required.

2.1.3 Version Control

Version control systems are intended to solve the problems of software systems evolution. They keep track of file changes made by different programmers by keeping a single copy for each file and recording a history of changes. Usually a string of numbers such as *1.2.1.1* is used to describe the possible version of a file. In fact, the version control systems provide a tree-like structure to describe the revisions among the different versions. However, in a situation where several versions are being worked on simultaneously, the version control system cannot describe the merged versions properly.

In Plaice and Wadge's approach [8], version labels (which are not necessarily numbered as *1.2.1.1*) are intended to have a global, uniform significance. Any version of the complete system can be formed by choosing the corresponding versions of the individual components. For example, on a Web site where every page of the site exists in a *Chinese* version as well as an *English* version, the *Chinese* version of the site is formed by linking all the *Chinese* version pages to each other. The *Chinese* version pages are formed by linking *Chinese* version text (Chinese text) and *Chinese* version images together. In general, all the pages and their components do not have to exist in the *Chinese* version (especially graphics or images). It may be possible to create the *Chinese* version of the whole site by altering only the textual parts, which will be in *Chinese*. Therefore, to create a *Chinese* version of a Web site, the *Chinese* version of each page is used; otherwise, if there is no special *Chinese* version, the ordinary *vanilla* version page is used instead. Similarly, to create a *Chinese* version of a *page*, *Chinese* version components are used when they exist; otherwise the ordinary *vanilla* version components are used.

Plaice and Wadge generalize this approach by defining a partially ordered algebra of version labels: "The partial order is the refinement relation: $V \sqsubseteq W$, read as "V is refined by W," or "V is relevant to W," which means (informally) that W is the result of further developing version V. The basic principle is that in configuring version W of a system, we can use version V of a component as long as that component does not exist in version V' with $V \sqsubseteq V' \sqsubseteq W$, where $V' \neq W$. The simplest possible algebra would allow only one version, which is called the *vanilla* version and represented by ε" [8].

2. 2 Intensional HTML

2.2.1 Versioning the Web

Yildirim, in his thesis [7, page 2], listed half a dozen reasons explaining why the Web designer should provide multi-versioned Web sites; for example: "the sites must be available in many different languages," "User have a wide range of bandwidths," "different browsers have different capabilities," "local information would differ in different parts of the world," etc. .... In fact, today the rapidly growing worldwide Web community puts much more pressure on the Web site constructor to provide this feature.

As Wadge points out, "In theory, Web pages, being purely electronic, can be updated frequently and provided in customized versions on demand. In practice, however, this is rarely the case." [9] In the real Internet world, there are a few multi-lingual or frequently updated Web sites in existence, but most of them are relatively simple. The reason for this is, "HTML itself provides very little support for production and maintenance of multi-versioned sites" [9].

The straightforward solution to providing a new version of a Web site is making a copy of the existing HTML source files and modifying them. Since the updating frequency of the different pages or different parts of a page is uneven, some pages or some parts of the pages are not changed in the new version as the others are. These should be treated as a shared resource. The problem is that HTML does not provide the mechanism to share these sources. At the Web site level, "different versions of the same site cannot share pages except through links; but then they have to share all the pages

accessible from the shared page." At the page level, "pages with analogous layout cannot share the markup which describes the layout" [9].

Therefore, this kind of cloning-and-editing method of updating Web site versions will spawn a large quantity of duplicated information, which will cause two problems. Firstly, this information will occupy a large quantity of extra storage space. Secondly, in the long run, the amount of information will make maintaining the consistency of many separate but parallel sites a severe problem. As the number of families of versions increase, maintenance costs increase as well, and any change made to the original source has to be propagated in the source for all the versions which use the original source [7, page 3].

Since HTML itself provides little support for the production and maintenance of multi-versioned Web sites, the version control mechanism was introduced into HTML by Wadge and his group of collaborators, and an extension of HTML called Intensional HTML (IHTML) was born. This made sharing the duplicated information among the multi-versioned Web pages practical.

Wadge, in the paper "The Malleable Document", briefly described this ideal: "It allows the author to create source files for pages and parts of pages which, in general, are generic—valid for a whole family of versions of the component in question. When a request for a particular version of a page is received, the source for that particular version is assembled from the result of specializing the relevant source files for the components. For example, the image on a page giving operation instructions for a copier might depend upon the model parameter, but not on the language parameter, whereas the instructions for loading paper will depend on the language parameter but perhaps not on the model

parameter (because they are all loaded the same way). In this situation an IHTML author must provide an image for each model, and a paragraph of loading instructions for each language, but not a separate entire page for each combination" [9].

Intensional HTML (IHTML) introduces the version control mechanism into HTML. The purpose of doing this is to share the duplicated information among the multi-versioned Web pages. In its implementation, IHTML keeps only one copy of the generic version and the current version is assembled at request time. After sending it to the client, the server discards the current version. In this manner, the storage space of the current page will be free later and the maintenance for that page is unnecessary. IHTML saves storage space and simplifies the maintenance work in multi-versioned Web pages.

2.2.2 IHTML Syntax

IHTML uses an intensional approach to versioning. The different versions of a document can be specified by expressions in a simple algebraic version language [9]. A particular version of a page does not necessarily exist in the server, but when the request comes in, it will be assembled from the relevant source files and be dynamically created by the server.

The request for a certain IHTML page consists of two parts: a conventional URL and a version expression. The former indicates the location of the requested page; the latter will decide which version of the page is requested. This request itself might be considered an intensional URL, which will be displayed in the address bar of the Web browser. The request, which includes a conventional URL and a version expression, can either be provided by a previous IHTML anchor or directly typed in by the user.

The simplest, one-dimensional version expression consists of a dimension identifier and a version value; both defined by the user and separated by a colon. An example of this is **Language:English**, where **English** is the version value of the dimension **Language**. Another example is **pop:on**, where **on** is the version value of the dimension **pop**. A more complicated and generally used version expression can express several dimensions. It can be expressed as the sum of two one-dimensional expressions, where each of the given dimensions has its respective value [9]. For example, **Language:English + pop:on**.

Besides appearing in URLs, version expressions are also introduced in HTML tags. HTML tags mark text as headings, paragraphs, list and so on. They are also used to organize images, applets within documents and to set up hypertext links, which connect the current page to other pages or Internet resources. "IHTML extends four elements of conventional HTML, namely anchor, image, applet and body (note: technically speaking, the elements are A, IMG, APPLET and BODY) and adds a new element (note: should be called as directive) that serves as an intensional equivalent of conventional server-side includes (note: might be named as Intensional Server-side Include). In IHTML, links to other pages, graphics, Java applets and server-side includes have intensional interpretations" [7, page 10]. Three of these elements, anchor and image and server-side includes, will be discussed below.

Anchor

Anchor provides the most important feature in IHTML-- intensional link, as in the following tag:

```
<a href= "index.html" VERSION= "Language:English">
Index.html</a>
```

In the above statement the absolute version value given by the expression **Language:English** leads to the *Language: English* version of the Index.html page, no matter what version the current page is. This means if the current page is in *Language:Chinese+pop:off* (or any other version), then the version value of the above hyperlink is defined as *Language:English* which is provided by the **VERSION** parameter in the tag.

The attribute **VERSION** specifies that the requested version of the linked page is exactly *Language:English*, ignoring the current version of the present page. Any dimensions not explicitly mentioned in the **VERSION** are considered to be *vanilla* (i.e. the most generic, or least refined, value), such as dimension *pop*.

In the browser, the request for a next page will be displayed as the URL: **http://.../Index.html<Language:English>**, a conventional URL followed by a version expression.

In Swoboda's IHTML 3 (third version), the above hyperlink can also be coded as:

```
<a href= "index.html<Lang:English>"> Index.html</a>
```

Thus, the version value can be directly coded into a URL in a tag and a unified format is set up between the hyperlink and the URL. Swoboda has already noted that an ambiguous situation can arise. If the following statement is given:

```
<a href= "index.html<Lang:English>"  VERSION=
"Lang:Chinese"> Index.html</a>
```

"The processor will ignore the attribute and use the versioning information given in the URL... since it is completely nonsensical for the user to explicitly give both" [12].

An IHTML hyperlink also has an optional attribute **VMOD** that will link to the target page that has a relative different version from the current page. **VMOD** is defined as version modifiers or dimension modifiers of the link so that the user can switch to the relative version based on the current version. In the above example, if the **VERSION** is given as **VMOD**,

** Index.html**

This means if the current page is in *Language:Chinese+pop:off* , then the version value of the above hyperlink will switch to *Language:English+pop:off.* This is the result of the merge of the current *Language:Chinese+pop:off* and the version modifier on dimension *Language --Language:English*. The dimension *pop* and the value *off* will still exist.

There are two types of version switching provided in the IHTML hyperlink, as seen above. The attribute **VERSION** leads to an "absolute" switching, which specifies the requested version of the hyperlink, and ignores the current version of the present page. Therefore all default values of the dimensions, which are not explicitly given in the **VERSION** expression, will be treated as *vanilla.* The attribute **VMOD** leads to a "relative" switching, which specifies the modifier of the hyperlink, and the version of the target page is modified from the version of the current page. Therefore all default values of the dimensions, which are not explicitly given in the **VMOD** expression, will be unchanged.

If neither **VERSION** nor **VMOD** is given in the hyperlink tag, the intensional link is syntactically identical to a conventional (extensional) link. For example:

** Index.html**

The version value of the target page will be given as the current version by default. If the current page version value is *Language:English*, then this hyperlink will lead to the *Language:English* version of the *Index* page. If the current page version value is *Language:Chinese* + *pop:off*, then this hyperlink will lead to the *Language:Chinese* + *pop:off* version of the *Index* page.

Image

The Intensional Image tag has similar parameters to the intensional hyperlink, for example:

```
<img src="image.gif" VERSION="Language:English"> Index.html
</a>
```

This statement defines using the absolute version *Language:English* of the **image.gif**. Thus, even in the *Language:Chinese* version of the Index page, the *Language:English* of the **image.gif** will be retrieved and displayed.

The same similarities are seen in the following statement:

```
< img src="image.gif" VMOD="Language:English"> Index.html
</a>
```

This statement defines using the version of the **image.gif** modified by the **VMOD** expression based on the current version of the page. Thus, in the *Language:Chinese* + *pop:off* version of the Index page, the *Language:English* + *pop:off* version of the **image.gif** will be retrieved and displayed.

For the default situation in which the version information is not given by the **VERSION** or **VMOD**, the current version will be the default version. For instance:

```
<img src="image.gif"> Index.html </a>
```

If the current version is *Language:English*, then the corresponding *Language:English* version **`image.gif`** will be retrieved and displayed.

Intensional Server-Side Include

IHTML also has a *server-side include* feature, which inserts the contents of the included file into the current page at the server site before it is sent to the client. This feature keeps the current version context in the included file as well, i.e. the version request of the included file will be the same as the current page.

As in the following example of a *server-side include*,

```
<!--#include file=extra.html" -->
```

if the version request of the current page is *English*, the file included should be the *English* version of **`extra.html`** if it is in storage. Otherwise, a more generic version will be included instead according to the *Best Fit Algorithm*. If even the most generic/*vanilla* version cannot be found, an error message will be sent to the user.

2.2.3 Best-Fit Algorithm

Brown, in his thesis "Intensional HTML 2: A Practical Approach", describes the *Best Fit Algorithm*: "If the requested version of a component does not exist, the server software considers the other versions, looking for one which could be refined to the requested version, and which is closer to the requested one than any other existing version (i.e., any other existing version which refined to the requested version also refined to this one). If such a version exists, it is used in place of the requested version. If not, it is an error" [2, page 13].

This algorithm applies to all page assembly processes corresponding to every particular version request. It will provide the solution for some "default" situations. For example, when a *Language:Chinese* + *pop:off* version **`image.gif`** is requested and only the *Language:Chinese* version **`image.gif`** exists, then the *Language:Chinese* version **`image.gif`** will be chosen as the "Best Fit" one for the current request.

2.3 A Bilingual Web site Sample

When Yildirim implemented the first version of Intensional HTML using CGI, he also built a bilingual (Turkish/English) site to demonstrate his work. This Web site can be switched between these two different languages with or without some images. It also provides options for some font and background color. However the server itself is immature and its response is quite slow, especially when the page is very large. Therefore it can hardly support a more ambitious trial.

The third version IHTML subsequently implemented by Swoboda provides a sophisticated foundation for building a more practical multi-versioned Web site. For this purpose, this author conducted a series of experiments and explored the different uses of the programming language.

2.3.1 Simplest Bilingual Example

In IHTML3, a new index file system was invented by Swoboda to manage the versions of the files. Compared to the encoding system used in IHTML2, it is simpler and more readable. Here is a simple example: a bilingual Web site consists of two different

language pages linked to each other. It has one dimension "*Language*" and two versions "*English*" and "*Chinese.*" The index file records the version information for the system.

In the *English* version file **1.html**:

```
<html>
<title> ENGLISH TITLE</title>
<body>
HERE IS THE ENGLISH TEXT
<a href="art.html" version=Lang:Chinese>TO CHINESE
VERSION</a>
</body>
</html>
```

In the *Chinese* version file **2.html**:

```
<html>
<title> CHINESE TITLE</title>
<body>
HERE IS THE CHINESE TEXT
<a href="art.html" version=Lang:English>TO ENGLISH
VERSION</a>
</body>
</html>
```

The content of the index file:

```
Lang:English    1.html
Lang:Chinese    2.html
```

This index file is not edited by the user. It is a datum file controlled by the system and can only be changed through the command **ici** (intensional check in) and **ico** (intensional check out) by the user. When the user types the command,

```
ici  -v Lang:English text.html
```

it will create a new directory "**text.html,i**" (if such directory does not exist yet). It will then copy the text.html to **1.html** under this directory as the *Lang:English* version of the **text.html**. A new index file also will be created and the first line will show:

```
Lang:English    1.html
```

There is a similar process for checking in the *Lang:Chinese* version of **text.html**. The second line in the index file will then show:

```
Lang:Chinese    2.html
```

The user can also use the **ico** command to check out one particular version from the "**text.html,i**" and the corresponding record will be deleted from the index file. When the user types the command,

```
ico  -v Lang:English text.html
```

the *Lang:English* version content will be copied into **text.html** file from the **1.html** and the **1.html** file will be deleted from the "**text.html,i**" directory.

When the server gets the intensional URL request for a certain version such as **http://.../text.html<Lang:English>,** it will search the "**text.html,i**" directory to find the right file and send it to the user.

In the above example, if the user asks for the *vanilla* version of the **text.html** such as: **http://.../text.html**, he or she will get an error message because the *vanilla* version does not exist. The better way to set up this bilingual Web page is by choosing one language version as the *vanilla* version. Thus, the user may use the following command to check it in:

```
ici  text.html
```

In the index file there will be a third line: **(vanilla) 3.html**. Of course, there will be a duplicated file, which has the same content as the *vanilla* version. In this case, the user can just check it out and may simply discard it.

The above example shows how the IHTML version system works but it does not really convince us that it can save some storage space or simplify the maintenance work. This is because no information can be shared between these two pages. A more complicated example will show some powerful features of this new IHTML language.

2.3.2 Switching between two versions

Resource sharing typically occurs when sharing some images and layout in the same Web page. An example could be a Web page where there are two language versions, *Lang:English* and *Lang:Chinese*, which have different text but need to share the same images in one page. For this case, IHTML has a kind of intensional case statement which, when evaluated, selects the text to include based on the particular version being produced. Thus, the bilingual page will be:

```
<img src="image.gif">      //SHARED IMAGES
<iselect>
<case version= "Lang:English">
ENGLISH TEXT
<a herf=text.html vmod= "Lang:Chinese"> TO CHINESE </a>
</icase>
<case version= "Lang:Chinese">
CHINESE TEXT
<a herf=text.html vmod= "Lang:English"> TO ENGLISH </a>
</icase>
</iselect>
```

In the browser, these two pages are shown as in Figure 1:

```
http://... /text.html<Lang:English>
http://.../text.html<Lang:Chinese>
```

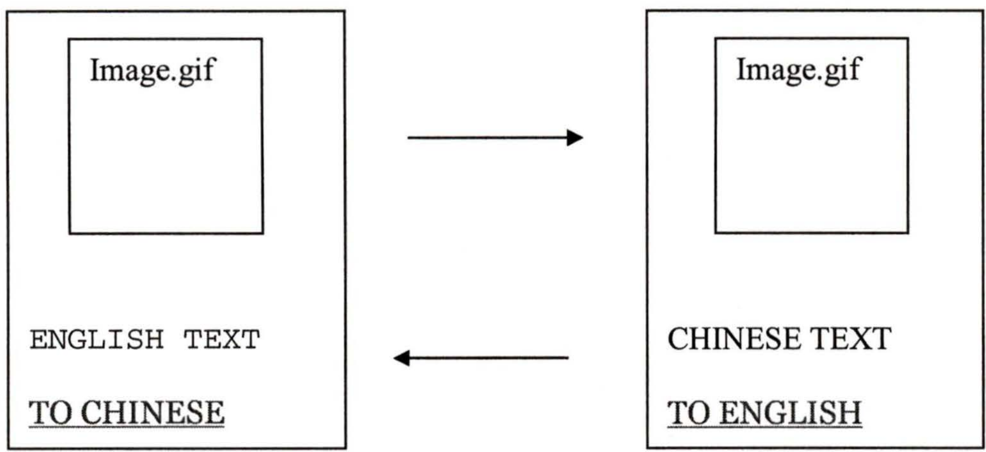


Figure 1: Switching between English and Chinese Web pages

In the above example, this same piece of code works for both *English* and *Chinese* versions. They share the same images and layout. For the *English* request, it will show English text; for the *Chinese* request, it will show Chinese text. The code also provides links to the user, which can freely switch between the *English* and *Chinese* versions.

A more interesting application of the above **iselect** statement is using it asymmetrically, which can pop up a section of text dramatically. In the author's bilingual Web page, **http://i.csc.uvic.ca/~i/Qin/Tangframe/tangsi.html**, it is used to pop up a paragraph of Chinese explanation after a paragraph of English text. The pop up part statements are as following:

```
<iselect>
 <icase version=popc1:on>
     <a NAME="LABEL1" href=#LABEL1 vmod=popc1:-
><CENTER><H2><IMG SRC="2birds2.gif" BORDER=0 HEIGHT=48
WIDTH=480></H2></CENTER></a>
</icase>
 <icase>
```

```
      <a NAME="LABEL1" href=#LABEL1
vmod=popc1:on><CENTER><H2><IMG SRC="2birds2.gif" BORDER=0
HEIGHT=48 WIDTH=480></H2></CENTER></a>
</icase>
</iselect>

<iselect>
      <icase version=popc1:on>
CHINESE TEXT
</icase>
<icase>
</icase>
</iselect>
```

The **2birds2.gif** is an animation image which is an intensional link to different versions. When the image is clicked in the *pop:-* (*vanilla*) version, which is shown in Figure 2 in **Appendix B,** the page will switch to the *pop:on* version. The *pop:on* version page will then pop up a paragraph of Chinese text, which is shown in Figure 3 in **Appendix B**. When the image is clicked in the *pop:on* version, the page will switch to the *pop:-* (*vanilla*) version, and the pop up Chinese text will disappear. Thus, besides the same English text being displayed in both versions, there is an extra paragraph of Chinese text displayed in the *pop:on* version. As Wadge comments, "This is easy enough to do in ordinary HTML but is almost never attempted. The main reason is that it requires creating and maintaining a separate copy of the original file. Even with only one caption this is a bad idea, but if there are dozens of images with captions it becomes completely impractical" [9].

2.3.3 Nest pop up

A more ambitious design is nesting the pop up texts. Such examples are provided in the four *Chinese Tang Poems* page written by the author. In these poems, not only is

every Chinese character given a dimension, but Chinese words (consisting of two characters) are also given a dimension. Each Chinese character or word dimension has a *pop:on* value, which will lead to the page displaying the English explanation for it. When all dimensions go to *vanilla* value all the explanations disappear as shown in Figure 4 in **Appendix B**. Since some Chinese characters have different meanings on their own and as a part of a word, the word explanation link will be provided in the character explanation, which leads to the explanation of the whole word. For example (in this paper, the Chinese characters are all romanised):

HUANG-yellow
HE-river
HUANG HE-the name of a particular river

In the Web page

http://i.csc.uvic.ca/~i/Qin/Tangframe/wangep.html, these explanations are nested in the following structure:

```
<iselect>
      <icase version=pop131:1> <a vmod=pop131:-> HUAN -
yellow;</a>See word:
     <iselect>
           <icase version=pop1311:1> <a vmod=pop1311:->
     HUAN HE "-river name;</a>
          </icase>
          <icase> <a vmod=pop1311:1> HUAN HE
"  </a>
      </iselect>
      </icase>
      <icase> <a version=pop131:1> HUAN </a>
</iselect>
<iselect>
      <icase version=pop132:1> <a vmod=pop132:-> HE -
river;</a>See word:
     <iselect>
          <icase version=pop1311:1> <a vmod=pop1311:-> "
HUAN HE "-river name;</a>
          </icase>
```

```
            <icase> <a vmod=pop1311:1> " HUAN HE
"  </a>
      </iselect>
      </icase>
      <icase> <a version=pop132:1> HE </a>
</iselect>
```

When the reader clicks the character **HUANG**, the page will not only pop up its explanation "**yellow**", but also provide the further link to pop up the word explanation of **HUANG HE.** At this time, the version value is *pop131:1.* The following text will show on the screen:

HUANG-yellow; See word: "HUANG HE"

(shown in Figure 5 in **Appendix B**)

Since the link, which pops out the **HUANG HE** word explanation, is modified by **vmod**, when the explanation "**a river name**" pops out, the original *pop131* version value *1* is still unchanged. Thus the explanation "**yellow"** is still on display. At that time, the request version is: *pop131:1+pop1311:1.* The following text will be shown on the screen:

HUANG-yellow; See word: "HUANG HE" -river name

(shown in Figure 6 in **Appendix B**)

In contrast, when the character **HE** is clicked and its explanation "**river"** pops out, since its link is attributed by a version number, the requested version will switch to *pop132:1*, and *pop131* and *pop1311* will revert back to *vanilla*. The explanation about **HUANG**, both for character and for word, will disappear.

In this way, a two-layered nested pop text is achieved in the above example. Further along, such a pop up text will turn out to be an extremely useful feature since it leads to an interesting research topic *Stretchtext* [9].

2.3.4 Stretchtext

Traditionally, documents usually written on paper or other mediums are considered fixed and unchangeable in content and form. An example is a book: when a book is published, each copy has the same content and the same form from cover to illustrations with no built-in flexibility. As Wadge points out, "the lack of flexibility is often, for authors, a distinct advantage. Authors (then and now) want law codes and religious text to be immutable, and contracts, account and property records to be incorruptible" [9].

In other situations, this immutability will cause some problems. For example, when the author tries to buy a computer book about Microsoft Windows GUI programming as a reference, usually there are only a couple of chapters of the book which are exactly what the author is interested in. However, the author has to pay the whole book price to get these chapters rather than cut out these pages. Wadge, in his paper titled "The Malleable Document", provides another example: "Problems also arise with documents which are directed towards a large heterogeneous readership. How, for example, can a firm write an instruction manual for a product (such as a car) which is produced in many different versions and sold in many different countries? Ideally each car sold would be delivered with an instruction manual tailored for that particular customer - say, a francophone who purchased the economy model with stereo and ABS but without air conditioning.... In practice, this is impossible, because there may be thousands of different possible versions of the manual. This is far too many to print and stockpile beforehand, especially considering that only a fraction of them will ever actually be needed" [9].

Another more sophisticated example was programmed by this author in the process of publishing the article "Putting the hyper back in hypertext" written by Wadge and m.c. schraefel [`http://i.csc.uvic.ca/~i/Qin/Wadge/firstd.html`].

In this Web page, the article initially consists of section titles (heading) alone, and the reader may choose which section to unfold and read. Furthermore, the reader can also choose the degree of detail. The initial page appears as follows:

1> Summary X
2> Two Cheers for the World Wide Web X
3> Intensional HTML X
4> Degree of Detail in IHTML X
5> Stretchtext for the Millions X
6> Poptext X
7> Choptext and Beyond X
8> Limitations of HTML X
9> Conclusion X
10> Acknowledgement X
11> References X
12> Appendix: Examples X

(shown in Figure 7 in Appendix B)

To the left of each section title, there is an arrowhead image, pointing sideways when the section is folded, pointing down when the section is unfolded. When the section is unfolded, the content of it is initially displayed in a *shorter* version. There is a second level option button provided to the reader, which reads "**full version**." When the reader clicks on it, the full text of the section will be displayed. At this time, the second level button switches to "**shorter version**", which provides the link back to the *shorter* version. The page will appear as follows when the second paragraph is displayed in the *shorter* version:

1> Summary X
2V Two Cheers for the World Wide Web X
full Version

```
The Web is based on the notion of hypertext, a concept
invented and named by Ted Nelson in [Nelson 1987].
Nelson, however, proposed a much more general and open-
ended definition of hypertext.
In traditional, paper-based sequential writing the reader
proceeds from beginning to end, and from the top of each
page to the bottom.
The individual articles, however, are entirely sequential,
and the magazine or book as a whole is intended to be read
more or less from front to back.
The Web, in spite of its enormous success, is based on a
very conservative notion of hypertext.
In Nelson's words, the Web is "the minimal concession to
hypertext that a sequence-and-hierarchy chauvinist could
possibly make".
```

3> Intensional HTML X
4> Degree of Detail in IHTML X
5> Stretchtext for the Millions X
6> Poptext X
7> Choptext and Beyond X
8> Limitations of HTML X
9> Conclusion X
10> Acknowledgement X
11> References X
12> Appendix: Examples X

(shown in Figure 8 in Appendix B)

Unlike the design in the Tang Poems examples, different sections not only can be unfolded simultaneously, but also be displayed in different degrees of detail. Since this kind of detail option is provided under each section, it may be called a *local option*.

On the other hand, of course, there is a *global option*. At the top of the page, there is a toggle button switch between the statuses "**exclude examples**" and "**include examples.**" It is a *global option* which controls whether the examples are displayed or underplayed in the Section 4,5,6 under the *full* version degree. By this button, the reader may choose if they want to read the example programs or not.

A more interesting example is making the text disappear. To the right of each section title, there is an **X** image. By clicking on it, the entire image is removed. This is

called *chop text* by the author. This is also a useful feature, one which Wadge mentions in [9]: "This would be useful, for example, in preparing a lecture. If the lecture notes were already on line as a malleable document, we could click away those sections (say, on hyperbolas) which we want to skip in the lecture. In fact, if we use video projection, we can erase them in real time - just as we can remove notes for topics we have finished covering. With malleable documents, we can begin to erase the distinction between browsing and editing."

Chopping the text deletes the whole section including the title; its left arrowhead and its right **X** image all disappear. Of course, we need not mention that the second level option and section text, whether in the *shorter* or *full* version, are all removed. How can the reader call them back? In the end, there is always a "**vanilla version**" button on the top of the page which can bring everything back to the initial status just as at the very beginning.

The following IHTML file is the part of the code of the second paragraph of the above example. Although it does not have the globe example *include/exclude* option, its logic is still complicated for most reader.

```
<iselect>
     <icase version=chop2:off>
     </icase>
     <icase>
     <iselect>
           <icase version=pop2:on>
           <a name="lab2" href=#lab2 vmod=pop2:-><img
     version="" border=0 src=2d.gif></a>
           </icase>
           <icase>
           <a name="lab2" href=#lab2 vmod=pop2:on><img
     version="" border=0 src=2u.gif></a>
     </iselect>
```

```
<B><FONT COLOR="#008000"><FONT FACE="Geneva"><FONT
SIZE=+1>Two Cheers for the World Wide
Web</FONT></FONT></FONT></B>
<a vmod=chop2:off><img version="" border=0
src=x.gif></a>
<iselect>
    <icase version=pop2:on><br>
    <iselect>
        <icase version=dep2:on>
        <a href=#lab2 vmod=dep2:-><img version=""
    border=0 src=brief.gif></a>
        </icase>
        <icase>
        <a href=#lab2 vmod=dep2:on><img version=""
    border=0 src=detail.gif></a>
    </iselect>
    <table border=0><tr><td width=0
bgcolor=yellow> </td><td>
    <iselect>
        <icase version=dep2:on>
        <P><FONT SIZE=+1>The World Wide Web may be
        one of the technological wonders of the
        world, but there's still plenty of room for
        improvement.</FONT></P>
        </icase>
    </iselect>
    <P><FONT SIZE=+1>The Web is based on the notion
    of hypertext, a concept invented and named by Ted
    Nelson in the [Nelson 1987]).
    <iselect>
        <icase version=dep2:on>In many people's
        minds, hypertext is the Web - a collection
        of pages with embedded links to other pages.
        </icase>
    </iselect></FONT></P>
    <P><FONT SIZE=+1>Nelson, however, proposed a much
more general and open-ended
    definition of hypertext.
    <iselect>
        <icase version=dep2:on> He described it
    simple as nonsequential writing,
        in which the reader &quot;can go instantly
    in a choice of directions from
        any given point&quot;.
        </icase>
    </iselect></FONT></P>
    .
```

```
          .
          .
          </icase>
           <icase></icase>
     </iselect>
     <P><FONT FACE="Geneva"><FONT
SIZE=+1> </FONT></FONT></P>
     </icase>
</iselect>
```

2.4 From IHTML To ISE

During IHTML programming, some drawbacks and limitations were revealed. It was shown that writing a *pop/chop text*, especially a nested one, is complex and tedious work, which needs a lot of skill and patience. The structure and the layout of each section are almost identical in the same page. When adding more sections of text into the same page, the author of the page has to clone and edit (within the page) similar layouts repeatedly, which makes the work tedious. Wadge points out these limitations in his paper [1]: "The first is that the extra markup required for even simple effects (like droptext) can be impracticably complex when faced with a document with a large number of droptext sections. The second problem is that IHTML is not a programming language. There are no provisions for evaluating expressions - this rules out constructs (like a slide show) because the author cannot generically specify a link in which a parameter is *incremented.* There is no iterative construct, so that sequences of a similar marked-up copy have to be (re)produced by hand. There are no functions, and this means (among other things) that copy must appear on the rendered page in the same order it appears in the marked up source."

Fortunately, these limitations have recently been remedied, when the author was able to build the bilingual Web site with ISE (Intensional Sequential Evaluator), a Perl-like scripting language devised by Wadge and Swoboda. Swoboda, the implementer, describes the language as follows: "ISE is an attempt to create the first imperative scripting language with versioned identifiers. The basic notion in ISE is that everything that can be identified, including user-defined functions, can also be defined in multiple versions. When constructs such as variables and functions are accessed in expressions, intensional best fits are used to find the appropriate version of the entity referred to" [11]. "Briefly, this means all entities in the language (variables, arrays, functions etc) can be versioned; that programs execute in an implicit context which determines which versions of the relevant entities are used; and that there are also explicit mechanisms for accessing and modifying the context" [9].

Based on experience with the bilingual Web site and the ISE plug-in, Wadge found an alternative way to produce an HTML file - using the "**`print`**" statement in ISE. Using the conditional construct (**`vswitch`**), ISE can output certain versions of HTML files at runtime. A different value of the version parameter following the URL determines a different versioned HTML file section being "printed" out. As suggested by Wadge, this author rewrote the *pop/chop text* program in ISE. To simplify the program, the arrowhead on the left and the **X** image on the right in the IHTML example are replaced by corresponding character ">", "V" and "X" in the ISE example. The arrowhead and the **X** are all moved to the right of the section titles.

The output is shown in Figure 9 in Appendix B and the ISE program is as follows:

```
#!/usr/bin/ISE
```

```
print("Content-type: text/html\n\n");
print(
@[
<html>
<head></head>
This is a test for pop chop text with macro
]@);
vswitch
{
<chop1:off> {print(@[]@);}
<> {print(@[<p>
TITLE1
]@);
print("<a href=\"");
print($ENV{"SCRIPT_NAME"});
print("<",[chop1:off],">");
print("\">");
print(
@[X</a>
]@);
   vswitch {
   <pop1:on>{
print("<a href=\"");
print($ENV{"SCRIPT_NAME"});
print("<",[pop1:-],">");
print("\">");
print(
@[V</a>
]@);
print(@[<p>SECTION ONE
]@);
}
    <>   {
print("<a href=\"");
print($ENV{"SCRIPT_NAME"});
print("<",[pop1:on],">");
print("\">");
print(
@[></a>
]@);
             }
       };
   }
};

vswitch
{
```

```
<chop2:off>
.
.
.
};

vswitch
{
<chop3:off>
.
.
.
};

print(
@[<p>
]@);
print("<a href=\"");
print($ENV{"SCRIPT_NAME"});
print("<",">");
print("\">");
print(
@[Back to Vanilla</a>
]@);
```

Although the above example looks quite complicated, only a few features of ISE have been used in it. The "**print**" statements are used to output the HTML file sections and the conditional construct (**vswitch** statement) is used to decide which version of the HTML file section will be output at runtime. A different value of the version parameter following the URL determines a different versioned HTML file section being "printed" out. The HTML produced as the output of this program depends on the value of the parameters: **chop1**, **chop2, chop3**, which are in the values of *vanilla* or *off*, and parameters **pop1, pop2, pop3**, which are in the values of *vanilla* or *on*. When the **chop1/chop2/chop3** is *off*, the corresponding section will disappear. When the **pop1/pop2/pop3** is *on*, the corresponding section text will be displayed. Otherwise,

as the *vanilla* version, only the section titles will be shown in the page. For the purposes of readability, the *pop/chop* features in this sample are simpler than the example in IHTML; it has no "*globe/local options*" and no nesting structure.

2.5 From ISE To IML

ISE itself is a powerful language and can be programmed in many intricate ways. In fact, they are so complicated that they are hard to use by most Web site authors. Considering that only a small fragment of ISE will be used in Web site development, and usually the same section of an ISE program is used for a similar purpose such as *pop/chop text*, it is a good idea to pack these ISE program sections as off-the-shelf components and to let the Web site author organize them rather than program them.

Wadge found: "There is a remarkably simple way to make the advantages of ISE available to authors who want only to mark up a copy, and not a program. We provide a set of macro definitions, which translate text with high-level (intensional) markup into ISE source.... For the time being, we use instead the macro facility of *groff*, the gnu upgrade of the (old fashioned) *troff* macro package included with every UNIX system. We use *groff* strictly as a macro processor; we run it with formatting disabled and with ASCII output selected" [1].

Following this idea, the author defined a set of macros for the *pop/chop text*. Of course, the definition of the macros may vary by different authors, even for the same piece of ISE code, but the basic ideal is the same—using one macro to represent a paragraph of ISE code.

For example, the beginning of the document can be defined as:

```
.de bdoc
#!/usr/bin/ISE
print("Content-type: text/html\en\en");
.bhtml
<html>
<head></head>
..
.
```

Here, the macro **bdoc** uses another macro **bhtml** in its definition block. This means we can nest our uses of macros to get blocks within blocks. The macro **bhtml** is defined as:

```
.de bhtml
print(
@[
..
.
```

With another macro **ehtml**:

```
.de ehtml
]@);
..
.
```

ISE's **@[...]@** construct allows embedded new lines between the macros. These new lines include *titles* and *text* sections. The complete macro definitions for *pop/chop text* are listed in Appendix A. By these macro definitions, the *pop/chop text* program in IML is:

```
.bdoc
This is a test for pop chop text with macros
.btext
.btitle
TITLE1
.bsection
SECTION ONE
.esection
.btitle
TITLE2
.bsection
SECTION TWO
.esection
.btitle
```

```
TITLE3
.bsection
SECTION THREE
.esection
.binit
```

Obviously, this fragment is clearer and more elegant than the one in ISE and more convenient for the common user building a *pop/chop text* page. There is no doubt that using a set of macro definitions is much easier than directly programming ISE itself.

However, remembering the macro definition and which set of them to use for what particular use still involves a steep learning curve, since different authors may define different macro sets for different use or even for the same use. The macro names, the content of the definitions and the significance of the arguments are totally decided by the individual authors. When the number of the macros increases beyond a certain point, it will be hard for the common user to remember or even find them. Another difficulty for the user is that the sequence of the macros usually cannot be changed in one program, and the user is restricted to following this order. For example, if the IML program misses the **btext** or puts it after the **btitle** in the above example, the program will not work properly.

On the other hand, IML itself has considerable development and maintenance difficulty, since "the definitions of the macros for the intensional features can be rather intricate ... The realistic droptext macro definitions we actually use are more complex, because we allow any number of retractable sections, arbitrarily nestable" [1]. As a matter of fact, only very skillful *groff* programmers can come up with these definitions. Writing a new set of macro definitions could be an extremely intricate task. Furthermore,

the low readability of the macro definitions makes the effort of reusing other people's definitions or adding a new macro definition a challenge.

Chapter 3
Microsoft Windows GUI Program For ISE

3.1 Design IWAG

Although for the average user, using IML makes Web page authoring much easier than using ISE directly, the macro definition makes IML itself hard to implement for the developer. Is it possible to find a way which is both easier for the Web site author and the macro developer? The answer is positive. This author found that Microsoft Windows provides an ideal solution. A Microsoft Windows GUI program called IWAG (Intensional Web Authoring GUI program) can be used to generate the ISE code for a particular purpose such as *pop/chop text*. The ISE code corresponding to the macro definitions could be stored in IWAG as a resource. The user may type in the text part, such as titles through the GUI, and IWAG will combine them into the ISE program as the output. Most unskilled computer users will feel happier using pull-down menus rather than remembering a set of unfamiliar macro definitions.

The basic scenario for *pop/chop text* is fairly simple. All steps are shown in Figure 10 to Figure 15 in **Appendix B**. The Window GUI program provides a set of menus. The user may select the menu item to input the *section titles* (shown in Figure 11 in **Appendix B**) and *section texts* (shown in Figure 12 in **Appendix B**). These section contents are recorded by IWAG. When the user selects the menu for producing the ISE program, IWAG offers a list of the section titles to use as options (shown in Figure 14 in **Appendix B**). The order of the *chop/pop text* sections is decided by the sequence of selecting these list items. Finally, a new ISE program is generated and output into a window called *ISE window* (shown in Figure 15 in **Appendix B**). This Microsoft Windows GUI program IWAG is implemented in the programming language C.

3.2 Advantages And Disadvantages

As a high level language, C is more powerful than the *groff* macro processor. There is no doubt that C can be used to replace *groff* as a parser in the new method. In fact, a parser for the macro program will be unnecessary when IWAG becomes the input facility. The user may type in the *text sections* into the *edit window* instead of weaving them into the macro program. When a new set of functions need to be added, only the Microsoft Windows menu item and the source files need to be changed rather than new Marcos needing to be defined. Therefore, some intricate macro definitions could be avoided. Although extra effort is needed in the new Microsoft Windows GUI item design, adding new features will be easier for most C programmers.

In future work, IWAG can be used to replace other sets of macro definitions by modifying the menu items and changing the ISE resource files. It can also be used to produce an IHTML program by changing the resource files from ISE to IHTML. In these works, a large part of the code will be reusable. Different functions, such as *slide show*, *pop/chop text*, can be glued together and more complicated GUI integrated programs could be built. In contrast, it will be difficult for IML to reuse the old definitions when the number of macros increases significantly, although macros added later on do reuse other macros in their definitions.

Chapter 4
Microsoft Windows GUI Program Implementation

4.1 Microsoft Windows Programming

4.1.1 How does Microsoft Windows work?

In traditional operating systems, programs are written using a *procedural programming model* in which programs are executed from top to bottom. For example, in a C program, execution begins at the first line in the function named **main()** and ends when **main()** returns. Basically, this is a sequential process. In between, **main()** might call other functions and these functions might call even more functions, but ultimately it is the program-not the operating system-that determines what should be called and when the calls are made.

For the Microsoft Windows environment, programs are written using an *event-driven programming model* in which the application program responds to events by processing messages sent by the operating system. All Microsoft Windows application programs include a function called **WinMain()** and a function known as the *window procedure.* **WinMain()** is the entry point.

As Jeff Prosise explains, "The *window procedure* processes messages sent to the window. **WinMain()** creates that window and then enters a *message loop,* alternately retrieving messages and dispatching them to the *window procedure.* Messages wait in a message queue until they are retrieved" [10, page 4].

The message might come from Microsoft Windows GUI in the form of a button click, or from the application program itself such as **WM_QUIT**.

"A typical Windows application performs the bulk of its processing in response to the messages it receives, and in between messages, it does little except wait for the next message to arrive. . . . The message loop ends when a WM_QUIT message is retrieved from the message queue, signaling that it's time for the application to end. . . . When the message loop ends, **`WinMain()`** returns and the application terminates" [10, page 4].

See Figure 16.

4.1.2 Why the Win32 API was chosen

The *window procedure* usually calls other functions to process the messages it receives; it can call functions local to the application, or it can call API (Application Programming Interface) functions provided by Microsoft Windows. The Win32 API includes hundreds of functions that an application can call to perform various tasks such as creating a window, drawing a line, and performing file input and output. API functions are grouped together in special modules called *Dynamic-link Libraries,* or DLLs. Calling API functions is a basic method to write Microsoft Windows programs and a lot of programmers still use it today.

Besides this there is an alternative way of programming Microsoft Windows called *Microsoft Foundation Class library*, better known by the acronym MFC. MFC is the C++ class library that Microsoft provides to place an object-oriented wrapper around the Microsoft Windows API. "In an MFC program, you don't often call the Windows API directly. Instead, you create objects from MFC classes and call member functions belonging to those objects." In the common view, "combined with a class library that abstracts the API and encapsulates the basic behavior of windows and other objects in reusable classes, [it] makes Windows programming simpler" [10, page 3].

However, "MFC is not the panacea some of its proponents would have you believe." For a small program such as the one we are discussing in this paper, MFC is not perfectly suited for the case. As a matter of fact, MFC hides the complexity of Microsoft Windows programming rather than making it simpler. On the one hand, it allows the programmer to concentrate on the structural design rather than the details in it. As Jeff Prosise explains: "the higher order it (MFC) lends to Windows programs frees programmers to spend more time developing the structural components of a program and less time worrying about the style bits passed to **`CreateWindow()`** and other nuances of the API" [10, page 11]. However, in our Windows GUI program IWAG design, we do care about the details of the style, especially in some menu sheets. Obviously, MFC is too big of a wrench for these screws.

Further more, using the API can help us understand more about how the Microsoft Windows GUI program works, as well as making the executable file shorter. Since MFC hides the details of some mechanisms and replaces it with computer-generated code, it will be less readable for the programmer. In our research approach, we prefer to understand as much detail as possible. Using MFC also has other drawbacks. The menus generated by the wizard might not exactly fit what we need and the automatically generated code would be much longer. In addition, compiling the MFC program varies depending on the version of the platform.

In future work, when IWAG is more integrated and turns into a bigger project, MFC might be taken into consideration. Since IWAG will be more complicated, MFC will make the programming simpler. Thus, the programmer will be able to concentrate on the program structure rather than already known details.

4.2 The Multiple-Document Interface

There are two different types of document interfaces in Microsoft Windows programming: *Single-Document Interface* and *Multi-Document Interface.* "A *Single Document Interface* (SDI) application can present two or more views of the same document in resizable 'panes' that subdivide the frame window's client area" [10, page 609]. An example of this would be Notepad that has only one document open at a time. The currently open file will be closed when a new one is opened.

"The *Multiple-Document Interface* (MDI) is a specification for applications that deal with documents in Microsoft Windows" [3, page 1173]. The specification describes a window structure and user interface that allows the user to process multiple documents within a single *application window.* "Just as (Microsoft) Windows maintains multiple application windows within a single screen, an MDI application maintains multiple document windows within a single client area." Programs, such as Word and Excel, are all of this type.

In MDI, the main application window is called the *frame window.* The client area in this *frame window* has a *client window.* This window is created by a call to **`CreateWindow()`** using a predefined window class **`MDICLIENT()`**. This *client window* covers the client area of the *frame window* and is responsible for much of the MDI support. The client area, or "*workspace*", is not directly used to display documents; rather it contains zero or more *child windows/document windows*, each of which displays a document. There is one and only one *client window* in the *frame window.* When this author tried to create two *client windows* in one *frame* at the beginning of the program

design, it turned out to be a failure. It was discovered that if the *frame window* has more than one *client*, the menu couldn't respond to the corresponding *child windows* correctly.

The *child windows* are attached to the *client* area and never appear outside the *application window.* The main *application window* and the *child windows* all have a *title bar,* a *sizing border*, a *system menu icon*, and *minimize/maximize/close buttons*. None of these *child windows* has a menu. However, the menu on the main application window applies to the document windows. Only one *child window* is active at any one time, (indicated by a highlighted title bar), and it will appear in front of all the other windows [3, page 1174].

Child windows or *document windows* are created by "initializing a structure of type **MDICREATESTRUCT** and sending the *client window* a **WM_MDICREATE** message with a pointer to this structure" [3, page 1175].

See Figure 17.

As Charles Petzold explains, "A *frame window* sends messages to the *client window* to perform an operation on a *child window* or to obtain information about a *child window*. For example, a *frame window* sends a **WM_MDICREATE** message to a *client window* to create a *child window*" [3, page 1176].

The Microsoft Windows GUI program IWAG uses all of these message mechanisms in its *main window* implementation.

4.3 The Program Structure

4.3.1 Introduction

In IWAG, there are one **`WinMain()`** procedure and three window callback procedures – **`FrameWndProc()`**, **`SectionWndProc()`**, **`ISEWndPro()`**, which are associated with three specific kinds of windows- *frame window*, *section window* and *ISE window*. Figure 15 in **Appendix B** shows one *ISE window* and nine *section windows* in one *frame window*. There are also two Dialog Box Procedures- **`TitleDlgProc()`** and **`OrgDlgProc()`**, which are associated with two different kinds of Dialog Box - *Title Dialog Box* (shown in Figure 11 in **Appendix B**) and *Organizer Dialog Box* (shown in Figure 14 in **Appendix B**) in this program. All these procedures are discussed in detail in the following sections. Before that, it is necessary to describe resources files which will be used in some of these procedures. In IWAG, there are two kinds of resource are special: *Menu* and "*Text*".

4.3.2 MENU Resources

There are three menus used by this program: *Initmenu*, *Sectionmenu* and *ISEmenu*. They are associated with conditions of zero window (no child window) in active, *section window* in active and *ISE window in active*. The templates of these menus are stored in the resource files **`MDIMENUINIT`**, **`MDIMENUSECTION`** and **`MDIMENUISE`** under **`Menu`** directory (shown in Figure 18 in **Appendix B**).

When IWAG begins to execute, there is no *document window* (*child window*) in the *main application window*. At this time, the *Initmenu* will be displayed (shown in Figure 10 in **Appendix B**). This menu simply allows a user to create a *section window*, *ISE window* or exit the program. When a new *section window* is created or an existing *section window* is active, the *Sectionmenu* will be displayed (shown in Figure 12 and Figure 13 in **Appendix B**). When a new *ISE window* is created or an existing *ISE window*

is active, the *ISEmenu* will be displayed (shown in Figure 15 in **Appendix B**). When all document windows including both *section window* and *ISE window* are closed, there is no child in the main application window and the *Initmenu* will be selected to display again.

4.3.3 "TEXT" Resources

ISE resource codes are stored under the resources **`TEXT`** directory. For ease of remembering, they have familiar names: **`BDOC`**, **`EDOC`** and **`BTITLE`** (shown in Figure 19 in **Appendix B**). These names are similar to the names in IML, although they do not have exactly the same content as the macro definitions in IML. In fact, the contents of **`BDOC`** and **`EDOC`** are almost same. **`BTITLE`** contains the sum of the **`btitle`**, **`bsection`** and **`esection`** macro definitions in IML.

These resources are set up by the programmer from the plain text file and are easily replaced. To achieve this change, the programmer just selects the Visual C++ menu item "Insert Resource" and follows the step-by-step instructions. This process is inaccessible to the common user, who will be more interested in programming it rather than modifying it.

4.3.4 WinMain Procedure

At the beginning of the program, three window classes are registered in **`WinMain()`** including the *frame window class* and two *child window classes*. Three menu handles are also obtained there. Associated with the three windows are three *window procedures* (*WinProc*): **`FrameWndProc()`**, **`SectionWndProc()`** and **`ISEWndProc()`**.

After the *frame window* is created by a call to **CreateWindow()** in **WinMain()**, **FrameWndProc()** will process the message **WM_CREATE**. In this case, **FrameWndProc** creates the *client window* by another call to **CreateWindow()**. It then goes back to **WinMain()**, where the newly created *frame window* will be displayed and the program will run into the message loop to retrieve and dispatch the messages.

```
int WINAPI WinMain ( )
{
// Register the frame window class
// Register the section child window class
// Register the ISE child window class
// Obtain handles to three possible menus & submenus
// Create the frame Window
// Enter the modified message loop
// Clean up by deleting unattached menus when quitting the
program
}
```

4.3.5 FrameWndProc Procedure

Besides **WM_CREATE**, **FrameWndProc()** also handles other messages such as **WM_COMMAND**. All messages generated by selecting the item in *Initmenu* will be processed here. The menu ID value that indicates which menu item has been selected will be passed by the **FrameWndProc()**'s parameter **wParam**.

The menu ID value **IDM_FILE_NEWSECTION** and **IDM_FILE_NEWISE** is associated with the menu item *create newsection* and *create newISE*. In the **IDM_FILE_NEWSECTION** case, a new child window *section window* will be constructed by sending a **WM_MDICREATE** message to the *client window*. Similar to the message **IDM_FILE_NEWISE** handler, a new child *ISE window* will be constructed by

sending a **WM_MDICREATE** message to the *client window*. Of course, the parameters are different.

When the menu item **IDM_APP_EXIT** is selected, a **WM_CLOSE** message will be sent from the *frame window* procedure **FrameWndProc()** to itself and the message **WM_CLOSE** will be handled later.

The other two menus *Sectionmenu* and *ISEmenu* have the same menu items as well. They are processed by the same procedure discussed above. Beside those items, these two menus also have menu item **IDM_FILE_CLOSE**, which close the active *child window*. As well, four other menu items correspond to the *child windows* arrangement: *tile*, *cascade*, *arrange* and *close all*.

All other **WM_ COMMAND** messages, which do not fit the above cases, will not be processed in **FrameWndProc()**. They will be sent to the active *child window* so that the *child window* can process those messages that pertain to its window.

```
LRESULT CALLBACK FrameWndProc (HWND hwnd, UINT message,
                               WPARAM wParam, LPARAM
lParam)
{
switch (message)
      {
      case WM_CREATE:
                  // Create the client window
      case WM_COMMAND
                switch (LOWORD (wParam))
                      {
                      case IDM_FILE_NEWSECTION:
                      // Create a Section child window
            case IDM_FILE_NEWISE:
                      // Create a ISE child window
            case IDM_FILE_CLOSE:
                      // Close the active window
            case IDM_APP_EXIT:
```

```
                // Exit the program
        case IDM_WINDOW_TILE:
        case IDM_WINDOW_CASCADE:
        case IDM_WINDOW_ARRANGE:
        case IDM_WINDOW_CLOSEALL:
                // Attempt to close all children
        default:
                // Pass to active child...
        }
    case WM_QUERYENDSESSION:
    case WM_CLOSE:
                // Attempt to close all children
    case WM_DESTROY:
    }
    return DefMDIChildProc (hwnd, message, wParam, lParam);
}
```

Besides the message **WM_ CREATE** and **WM_COMMAND**, there are three other messages which are processed in the **FrameWndProc(): WM_QUERYENDSESSION, WM_CLOSE** and **WM_DESTROY**. These messages will eventually result in closing and exiting all windows of the program.

4.3.6 SectionWndProc Procedure

When a new *section window* is created by the Microsoft Windows system (such as the one in Figure 12 in **Appendix B**), the window procedure **SectionWndProc()**, which is associated with this new window, will receive a **WM_CREATE** message. In the **SectionWndProc()**, there is a message handler under switch case **WM_CREATE** dealing with this message. In this handler, a dialog box will pop up to ask the user to input the *section title* (shown in Figure 11 in **Appendix B)**. The template of this dialog box is stored in the resource directory. The dialog box template has one input line and two buttons: *OK* and *Cancel*. The dialog box is associated with a dialog box procedure **TitleDlgProc()** to process these input messages: when the *OK* button is pressed, the

tile name typed in by the user on this line will be stored into a variable and will be displayed as the caption of the *section window*. When the *Cancel* button is pressed, the title name will not be stored.

After the above operation, **SectionWndProc()** will create a *child window* of the same size as the *section window*. This new window, called the *edit window*, will process every edit command from the users (such as the one shown in Figure 12 in **Appendix B**). Thus the **SectionWndProc()** only deals with windows' regular messages such as changing window size. The *edit window* has its own window procedure, which will focus on edit message processing.

When **SectionWndProc()** receives a **WM_SIZE** message (caused by changing the window border by the user), it uses the **MoveWindow()** function to resize the window.

When **SectionWindowProc()** receives a **WM_MDIACTIVATE**, which means the *section window* comes into focus, it will change the *frame window* menu to *Sectionmenu*.

Other messages are processed using similar handlers as in **FrameWndProc()**.

```
 LRESULT CALLBACK SectionWndProc (HWND hwnd, UINT message,
                                   WPARAM wParam, LPARAM
lParam)
 {
 switch (message)
       {
        case WM_CREATE:
                 // Pop up an dialog box asking for title of
                 the new section and create an edit window
       case WM_SIZE :            // Resize the section window
       case WM_MDIACTIVATE:      // section window got focus
       case WM_QUERYENDSESSION:
```

```
            case WM_CLOSE:          // Attempt to close all
children
        case WM_DESTROY:
        }
        return DefMDIChildProc (hwnd, message, wParam, lParam);
}
```

4.3.7 `edit` Window

In **`SectionWndProc()`**, an *edit window* is created in the *section window* by sending a **`CreateWindow()`** call, which includes the class name **`edit`**, to the Microsoft Windows system. The **`edit window`** is a predefined class, which defines a rectangle based on the *x* position, *y* position, width, and height parameters of the **`CreateWindow()`** call. "This rectangle contains editable text. When the *child window* control has the input focus, you can type text, move the cursor, select portions of text using either the mouse or the Shift key and a cursor key, delete selected text to the clipboard by pressing Ctrl-X, copy text by pressing Ctrl-C, and insert text from the clipboard by pressing Ctrl-V" [3, page 395]. (such as the one shown in Figure 12 in **Appendix B**). All of these messages can be handled by Microsoft Windows if the programmer does not want to handle them.

4.3.8 ISEWndProc Procedure

When a new *ISE window* is created by Microsoft Windows, **`ISEWndProc()`**, which is associated with this new window, will receive a **`WM_CREATE`** message as well. In **`ISEWndProc()`**, there is a message handler under the switch case **`WM_CREATE`** dealing with this message. In this handler, the following jobs are done:

First, searching of all *section windows* (*ISE window* excluded) within the *frame w*

Window, then recording the *section tile* to an array. Since the program is executing in the **`ISEWndProc()`**, the tiles cannot be retrieved from these windows directly; they have to be fetched from a storage space. We discuss this storage space in detail as **Technique Difficulty 1** later.

Second, a dialog box will pop up to ask the user to select the *text sections* that need to be organized. This dialog box, called the *organizer* (shown in Figure 14 in **Appendix B**), is much more complicated than the one in **`SectionWndProc()`**. A dialog box procedure **`OrgDlgProc()`** is associated with it which is discussed in detail later in Section **4.3.9 OrgDlgProc**.

Third, the ISE file is output into the *ISE window* (such as the one shown in front of Figure 15 in **Appendix B**). The actual text organization work is achieved here. In brief, **`ISEWndProc()`** loads the ISE resource code from the resource file, then inserts the text into the resource code in the order provided by the **`OrgDlgProc()`**. The total output text length also needs to be calculated as it will help the *ISE window* control the scroll button in the right border of the *ISE window*.

The actual process is a little more complicated than the above brief description. Besides **`BDOC`** and **`EDOC`**, which only output once, **`BTITLE`** often outputs repeatedly since usually more than one *section text* needs to be output.

The information for the *section order* is provided by the **`OrgDlgProc()`**, and consists of a list of the *section titles*. According to the *section titles*, **`ISEWndProc()`** searches every *section window* and finds the matching one. It then reads out the *section text* that the user edits in the *section window*. The *section title* and the *section text* will be inserted into the right place within the ISE code. Thus, the ISE program is produced and

will be displayed in the *ISE window*. The insert process is a "**`printf()`**" statement which is an elegant method inspired by *groff*; this is discussed in the section titled **Technique Difficulty 2**. To avoid data loss and for future use, the *section title* and the ISE program is stored in a reserved space which is discussed in **Technique Difficulty 3**.

Other messages dealt with in **`ISEWndProc()`** are **`WM_PAINT`** and **`WM_VSCROLL`**. When **`ISEWndProc()`** receives the **`WM_PAINT`** message, it will output the content of the ISE program produced in the **`WM_CREATE`** handler mentioned at the beginning of this section. The **`WM_VSCROLL`** message is received when the user clicks the mouse on the scroll bar. Then, according to the parameter that indicates the direction of scrolling, the content in the *ISE window* will be adjusted - scroll up or down.

Other message processes, such as **`WM_SIZE`**, **`WM_SETFOCUS`**, **`WM_MDIACTIVATE`**, are dealt with in similar ways as in the **`FrameWndproc()`** and do not need to be repeated.

```
LRESULT CALLBACK IseWndProc (HWND hwnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
{
 switch (message)
{
case WM_CREATE:
// Search all section windows and record the section tile
to an array
// Pop up a dialog box to ask user select the text sections
//the ISE file is output in the ISE window
case WM_SIZE:
case WM_SETFOCUS:
case WM_VSCROLL:
case WM_PAINT:
case WM_MDIACTIVATE:
case WM_QUERYENDSESSION:
case WM_CLOSE:
case WM_DESTROY:
```

```
}
      return DefMDIChildProc (hwnd, message, wParam,
lParam);
}
```

4.3.9 OrgDlgProc

OrgDlgProc() is the procedure associated with the *Organizer* dialog box which provides the *title list* to the user for making options. The template of the dialog box is stored in the resource. It includes two text template areas - **IDC_ISE_LIST_TEMPLATE** and **IDC_SECTION_LIST_TEMPLATE** - and four buttons - *Select*, *Unselect*, *OK* and *Cancel* (shown in Figure 14 in **Appendix B**). When the *Organizer* dialog box pops up to offer the user choices, the *section title list* is in the **IDC_SECTION_LIST_ TEMPLATE** area, on the right side of the dialog box. By focusing on one *section title* and clicking the *Select* button, this *section title* will be moved into the **IDC_ISE_ LIST_TEMPLATE** area, at the left side of the dialog box. When more than one *section titles* have been moved in, they will form a new list, which is called the *ISE list*. This *ISE list* will provide the sequence information to the **ISEWndProc()** and the **ISEWndProc()** will organize the *text sections* typed in by the user according to this information. Of course, the *section titles* in the *ISE tile* list can also be moved back to *section title list* by clicking the *Unselect* button.

The **OrgDlgProc()** itself is quite a complicated procedure, and deals with the messages from the dialog box and the *title lists*. When the dialog box is created at the beginning of the procedure by a **WM_INITDIALOG** message, two title lists, *Sectionlist* and *ISElist,* are created from the **TitleListView** class. Unlike the **edit** class, which is predefined, **TitleListView** is a Microsoft Windows class built by the author. Not

only does it have normal class functions, such as *Create*, *Destroy*, *Moveto*, etc., but also the callback function **Util_WndProc()**, which will send a message **WM_USER_PROVACYLISTVIEW_LEFTMOUSE_CLICK** to the **OrgDlgProc()** when the *title list* is clicked.

After these two *title lists* have been created, they will be moved to the position defined by the template in the resource, **IDC_ISE_LIST_TEMPLATE** and **IDC_SECTION_LIST_TEMPLATE**. The *section titles* will then be loaded into the *section list.*

When **OrgDlgProc()** receives a message **WM_USER_PROVACYLISTVIEW_LEFTMOUSE_CLICK** from *title list* (*SectionList* or *ISEList*), the mouse position will be detected to judge if any *tile* has been clicked. If so, this *title* will be highlighted and the *Select / Unselect* button will be enabled.

When **OrgDlgProc()** receives a **WM_COMMAND** message, it will judge if it is a button click. If it is, then **OrgDlgProc()** will do a different process according to which button has been clicked. If the button is *Select / Unselect*, the *title* will be moved out from *SectionList / ISEList* and put into the *ISEList / SectionList*. If this *title* is not the last item in the *SectionList / ISEList*, the next item should be highlighted. Otherwise the last one should be highlighted. If the *SectionList / ISEList* is already empty, the *Select / Unselect* button will be disabled. Thus, the next (default) *title* item can be automatically highlighted and continuously moved by using the "*Select / Unselect*" button. This is more comfortable for the user than highlighting them manually one by one. If the button is *OK*, the order of the *ISEList* will be stored and this information will be provided to the **ISEWndProc()**.

```
BOOL CALLBACK OrgDlgProc (HWND hDlg, UINT iMsg,
                            WPARAM wParam, LPARAM lParam)
{
switch (iMsg) {
     case WM_INITDIALOG: {
     //Create ListView for the ISEList and SectionList
     //Load the name sections titles to the SectionList
case WM_COMMAND:
          {
          switch (HIWORD(wParam)) {
          case BN_CLICKED:
               switch (LOWORD(wParam)) {
                    case IDC_BUTTON_SELECT:
               case IDC_BUTTON_UNSELECT:
               case IDOK :    //When press OK button, store
               the List items
               case IDCANCEL
     }
}
case WM_USER_PROVACYLISTVIEW_LEFTMOUSE_CLICK:
}
}
```

4.3.10 TitleListView

`TitleListView` is a window class built by the author to implement the **`OrgDlgProc()`**. It includes eleven functions and one callback function **`Util_WndProc()`**. Most of them are concerned with *list* operations. Since it is a "supporting role" in IWAG, the details of the structure are not further discussed in this thesis.

4.4 Some Difficulties With The Technique And Solutions

4.4.1 Data Storage

One common issue is that almost every window procedure needs to store data. For example: in **`SectionWndProc()`**, when a new *section window* is created, the *edit*

window handle **hwndEdit**, which is created in the **WM_CREATE** case handler, should be stored for future use. When a **WM_SIZE** message arrives and the **SectionWndProc()** is called again, the **hwndEdit** will be fetched in the **WM_SIZE** case process. Now, the question is how to store the data? The obvious solution is using a static variable as in the following example:

```
LRESULT CALLBACK SectionWndProc (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
staticHWND hwndEdit;

switch (message)
          {
          case WM_CREATE:   //Edit window's handle created
and store in hwndEdit
     hwndEdit = CreateWindow (TEXT ("edit"), NULL,.....);
case WM_SIZE :        //Fetch the hwndEdit
     MoveWindow (hwndEdit, 0, 0, LOWORD (lParam), HIWORD
     (lParam), TRUE);
     }
}
```

This method seems correct because the static variable keeps the value unchanged rather than initialized when the same procedure is called again. However, the actual situation here is more complex than that.

As Charles Petzold explains, "As with any window class used for more than one window, static variables defined in the window procedure (or any function called from the window procedure) are shared by all windows created based on that window class" [3, page 1194]. This means using static variables in the window procedure only works in one window condition. When the window procedure is called by the Microsoft Windows system repeatedly, the static variable can be used to store data for this associated window

only. When more than one window uses the same window procedure, this data will be corrupted.

The correct method "uses memory space reserved by defining a nonzero value in the **cbWndExtra** field of the **WNDCLASS** structure which is used to register the window class" [3, page 1194]. As a matter of fact, in the Window GUI program this space stores a pointer that references a block of memory the size of **SECTIONDATA**. The structure **SECTIONDATA** is defined at the beginning of the program:

```
typedef struct tagSECTIONDATA
{
HWND hEdit;
char cTitle[MAX_STR_LEN];
}
SECTIONDATA, * PSECTIONDATA;
```

Thus, the correct solution is as follows:

```
LRESULT CALLBACK SectionWndProc (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
static PSECTIONDATA pSectionData;
switch (message)
          {
          case WM_CREATE:
          //create edit window handle
     hwndEdit = CreateWindow (TEXT ("edit"), NULL,.....);
     .
     .
     //put hwndEdit in a data structure pSectionData
     pSectionData->hEdit = hwndEdit;
     strncpy(pSectionData->cTitle, szTitleName, sizeof
     (szTitleName));
     //Store data to memory space
     SetWindowLong (hwnd, 0, (long) pSectionData);
case WM_SIZE :
     //Get pSectionData from memory space
     pSectionData = (PSECTIONDATA) GetWindowLong (hwnd, 0);
     //Get hwndEdit
     hwndEdit = pSectionData->hEdit;
```

```
        MoveWindow (hwndEdit, 0, 0, LOWORD (lParam), HIWORD
        (lParam), TRUE);
        }
}
```

Although there are other solutions, this is the one adopted by Charles Petzold [3, page 1194] and it works perfectly in the **SectionWndProc()** for storing the **hwndEdit**. This author also found another advantage: since this memory space is associated with every window, not only can it restore the data within the **SectonWndProc()**, but it can also provide data for other procedures. The second item **cTitle** of structure **SECTIONDATA** is for that purpose. The variable **szTitleName**, which records the *title* of *section window*, can be copied into the **cTitle** and stored in memory as well. The content of this memory can be retrieved from **ISEWndProc** by knowing the window handle **hCurrent**. This part of program is as follows:

```
pSectionData = (PSECTIONDATA) GetWindowLong (hCurrent, 0);
strncpy(sSectionWindow[iSectionWindow].cTitle,
pSectionData->cTitle, sizeof(sSectionWindow[iSectionWind])
```

4.4.2 **Printf** Statement

When **ISEWndProc()** organizes the *section texts* and produces the ISE program, the most important job is finding the right place and filling in the right value (including number and text). In IML, page numbers are represented by the escape variable in the macro definition, which can be increased when the parser goes though the source code. This work is quite difficult for the average C programmer since the escape variable is tricky in the macro definition.

In IML, the *section texts* are arranged to insert among the macros in the source code by users. When a new *section* is added, a title will be put after a title macro **btitle** and then a section of text will be put between a pair of macro **bsection** and **esection**. These macros are the same as in other *sections* and all titles and texts are inserted manually by the user.

Is there a method to make this process simpler? The answer is yes. Fortunately, the high level language C provides us with a more effective way to deal with the above problem. One straightforward solution is using special characters to mark out the places in the ISE source to put the number, title and the text so that the whole insert process can be done automatically by IWAG. Except for the beginning and the end of the ISE program, the body part of the ISE program can be produced by a loop statement, in which the numbers, titles and texts can be inserted repeatedly. Thus, tiles and the texts can be inserted in the same loop and only three ISE resources are needed (**BDOC, BTITLE** and **EDOC**) rather than five in macros.

At first, a simple but clumsy program did this process in the loop. The program copied the ISE resource **BTITLE** character by character to a buffer, while searching for the special *mark character* in it. When the special *mark character* is encountered, it is replaced by the corresponding number, title or text in the buffer. Eventually the ISE program is formed in the buffer.

Later, inspired by the *groff* escape variable, the author found that the **printf()** is an excellent solution to achieve this. In the ISE resource **BTITLE**, **%d** marks out the position where numbers will be inserted and **%s** marks out the places where titles or texts will be inserted. Thus the **BTITLE** is as following:

```
vswitch
{
<chop%d:off>
{
.
.
print(
@[
      <p>
%s
]@);
print("<a href=\"");
print($ENV{"SCRIPT_NAME"});
print("<",[chop%d:off],">");
.
.
   vswitch {
   <pop%d:on>{
print("<a href=\"");
print($ENV{"SCRIPT_NAME"});
print("<",[pop%d:-],">");
.
.
print(
@[
       <p>
%s
]@);
            }
    <>   {
print("<a href=\"");
print($ENV{"SCRIPT_NAME"});
print("<",[pop%d:on],">");
.
.
```

In the above ISE resource code **BTITLE**, there are 5 places which need numbers and 2 places which need text. These are demarcated by symbols **%d** and **%s**.

By these format symbols, in the **ISEWndProc()**, a **printf()** statement matched to the **BTITLE** resource will insert the number and text and output them into a buffer **pTemp**. The variable **dwIndex** is a loop counter which matches the **%d** in the

BTITLE, and which will be "printed" in the **pTemp** as the counter value. The **ptitleBuffer** contains the title which matches the first **%s**, and which will be "printed" in the **pTempe** as the *title*. The variable **pstrBuffer** contains the *section text*, which matches the second **%s**, and which will be "printed" in the **Ttemp** as the *section text*. The **printf** statement is shows as follows:

```
sprintf(pTemp,pBtitleText,dwIndex,ptitleBuffer,dwIndex,dwIn
dex,dwIndex,pstrBuffer,dwIndex);
```

So, a paragraph of a clumsy program in the loop was replaced by a single **sprintf()** statement. What an elegant solution!

4.4.3 Title and TitleList

The length of *title* of the *section windows* and the length of the *title list* item are different. The latter is equal or shorter than the former.

To make it more efficient, an array of union is used to record the *title* and the handle of the *section windows* in the **ISEWndProc()**. The *titles* recorded here will be the *title list* item; they are limited to 31 characters in length.

```
struct sWnd{
char cTitle[32];
HWND hSection;
 }  sSectionWindow[1000];
```

In the **WM_CREATE** handler in **ISEWndProc()**, all the *child windows* within the *frame window* are searched out and all *section windows' title* and handle are recorded in this array.

Later, **OrgDlgProc()** will fetch the *title* from this array and form a *title list*. The *title list* items will not be longer than 31 characters so they will fit the size of the dialog box.

After **ISEWndProc()** has gotten the *section order* information, which still consists of the limited length *title*, **ISEWndProc()** will use the above array to find out the corresponding *section window* handle. Thus, the actual *title* of *section windows* can be retrieved from the memory space using **GetWindowLong (hCurrent, 0)** just as we discussed in the Section **Data Storage.**

Chapter 5
Conclusions and Future Work

5.1 To Generate Other ISE Files

The Microsoft Windows GUI Program IWAG discussed in this thesis is designed to generate a piece of an ISE program, which can be put into the ISE server. Supported by the server, the ISE program can produce pages of *pop/chop text* for a client - Web browser. Obviously, the *pop/chop text* is a simple application for both ISE and IWAG.

To generate other kinds of ISE programs, such as *slideshow*, which links the Web pages in linear sequence, the following steps may be followed:

First, replace the ISE resource code. The programmer may design a new ISE program and try it on the server to make it work. Then this new ISE program should be changed into an ISE resource. All the places where text needs to be inserted may be divided by **%s**. If the output text has repeat paragraphs like *pop/chop text*, the ISE resource may be divided into three parts: *begin, body* and *end*, just like **`BDOC, BTITLE`** and **`EDOC`**. The body part can be repeated by a loop statement just like **`BTITLE`**. The version numbers, which are needed in the ISE program, may be replaced by **%d**. If the output text has no repeat paragraphs, one single resource ISE should be enough. Otherwise, three resources are needed to produce the ISE program. Of course, it is possible that more than three ISE resources are needed, something which can be decided by the programmer. The ISE resources are feasible for every instance.

Second, change the **`printf()`** statement. The parameters in the **`printf()`** statement should match the format symbols **%d** and **%s** in the resource file, both in the sequence and quantity.

Third, modify IWAG. To those ISE programs which are quite different from or more complicated than *pop/chop text* in structure, the program itself might need to be changed as well. For example, if the *pop/chop text* needs an option for background color, another dialog box providing the color choice needs to be added in. Usually, the frame of the main window will not change a lot.

Unlike IHTML, ISE is a programming language. It provides us with adequate space for many tricky designs and far more freedom beyond *pop/chop text*. However, someone wanting a simple design would tend to use IHTML, and using IWAG to generate IHTML code could be an option. To generate an IHTML file using IWAG, the critical thing to do is simply replace the ISE source by IHTML source. The other changes in the program are the same as the changes for the ISE program.

Of course, the IHTML program should be put into an IHTML server rather then ISE server, and hopefully it will produce multi-versioned Web pages according to the user's wish.

5.2 Integrating Different Functions

If the resource is changed, IWAG can be modified into a different form for producing different ISE programs. Each form could be thought of as a special function. In future work, it is possible that all these functions could be grouped together as an integrated environment. There could be a *main menu*, listing all these functions as options for the user. The user could choose to produce different ISE programs, even IHTML programs. For each function, a separate window could pop up as the function's main window, just like the *frame window* in the *pop/chop text*. Thus, the MDI structure would still keep working and its code could be reused in the new environment. Of course, this

is just one of the many possible designs; all possibilities and details should be researched and considered in the future.

When the Microsoft Windows GUI programs integrate into an environment, dozens of different functions might combine into one big Microsoft Windows program. This program (might be called the same name IWAG) would be much more complicated. Thus, considering the complication involved, MFC and C++ might be taken into consideration as development platforms. There are several reasons for this:

First, since several things such as style details have investigated with IWAG, the programmer would be free to consider more structural problems rather than focusing on details.

Second, by using C++, some code such as dialog boxes would be reusable. When many function parts need similar dialog boxes in their program, reusability of object-oriented design can be really helpful to reduce the amount of code.

Third, because the C++ program tightly binds the code and data for encapsulation, different functions may have more independence. The program will not be disrupted by later modifications, no matter how big.

Above all, for a big Microsoft Windows project, more effort spent in finding the right MFC library and rewriting IWAG will pay off.

Bibliography

[1] W.W. Wadge, "Intensional Markup Language," unpublished, 2000.

[2] G.D. Brown, *Intensional HTML 2: A Practical Approach,* M.Sc. Thesis, University of Victoria, 1998.

[3] C. Petzold, *Programming Windows (Fifth Edition),* Microsoft Press, 1999.

[4] R. Carnap (1891-1970), *Meaning and Necessity: A study in semantics and modal logic*, University of Chicago Press., 1958.

[5] A.A. Faustini and W.W. Wadge, "Intensional Programming," Technical Report-DCS-55-IR, Department of Computer Science, University of Victoria, 1986.

[6] R. Thomason, editor, *Formal Philosophy,* Selected Papers of R. Montague, Yale University Press, 1974.

[7] T. Yildirim, *Intensional HTML*, M.Sc. Thesis, University of Victoria, 1997.

[8] J.A. Plaice and W.W. Wadge, "A new approach to version control," *IEEE Transaction on Software Engineering,* vol.19, no. 3, pp. 268-276, March 1993.

[9] W.W. Wadge, "The Malleable Document," unpublished, 1999.

[10] J. Prosise, *Programming Windows With MFC (Second Edition),* Microsoft Press, 1999.

[11] P. Swoboda, "The Language ISE," unpublished, 1999.

[12] P. Swoboda, "IHTML 3 Implementation Guide," unpublished, 1999.

[13] L. Liu, *Temporal HTML,* M.Sc. Thesis, University of Victoria, 1999.

[14] V.T. Phong, *Intensional XML*, M.Sc. Thesis, University of Victoria, 2000.

Appendix A:

macro Definition for the pop/chop text

```
.nf
.na
.de bdoc
#!/usr/bin/ISE
print("Content-type: text/html\en\en");
.bhtml
<html>
<head></head>
..
.
.de bhtml
print(
@[
..
.
.de ehtml
]@);
..
.
.de btext
.ehtml
.nr s 0 1
..
.
.de btitle
vswitch
{
<chop\\n+s:off>
{
.bhtml
.ehtml
}
<> {
.bhtml
      <p>
..
.
.de balink
print("<a href=\e"");
print($ENV{"SCRIPT_NAME"});
print("<",[\\$1],">");
print("\e">");
..
```

```
.
.de ealink
</a>
..
.
.de binit
.bhtml
<p>
.ehtml
print("<a href=\e"");
print($ENV{"SCRIPT_NAME"});
print("<",">");
print("\e">");
.bhtml
Back to Vanilla
.ealink
.ehtml
..
.
.de bsection
.ehtml
.balink chop\\ns:off
.bhtml
X
.ealink
.ehtml
   vswitch {
   <pop\\ns:on>{
.balink pop\\ns:-
.bhtml
V
.ealink
.ehtml
.bhtml
      <p>
..
.
.de esection
.ehtml
             }
    <>   {
.balink pop\\ns:on
.bhtml
>
.ealink
.ehtml
           }
```

Appendix B:

Figures

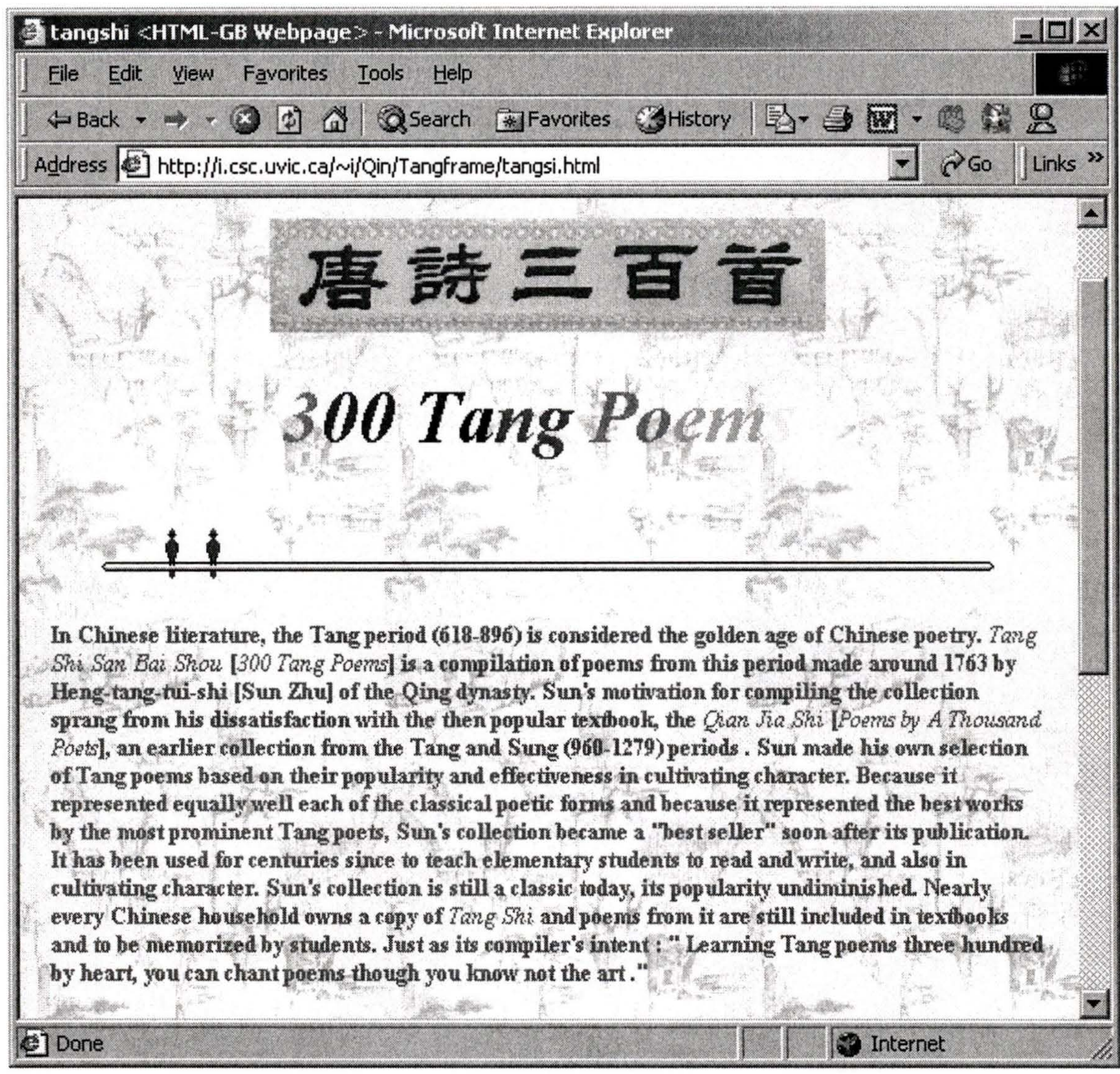


Figure 2: tangsi.html in the vanilla version

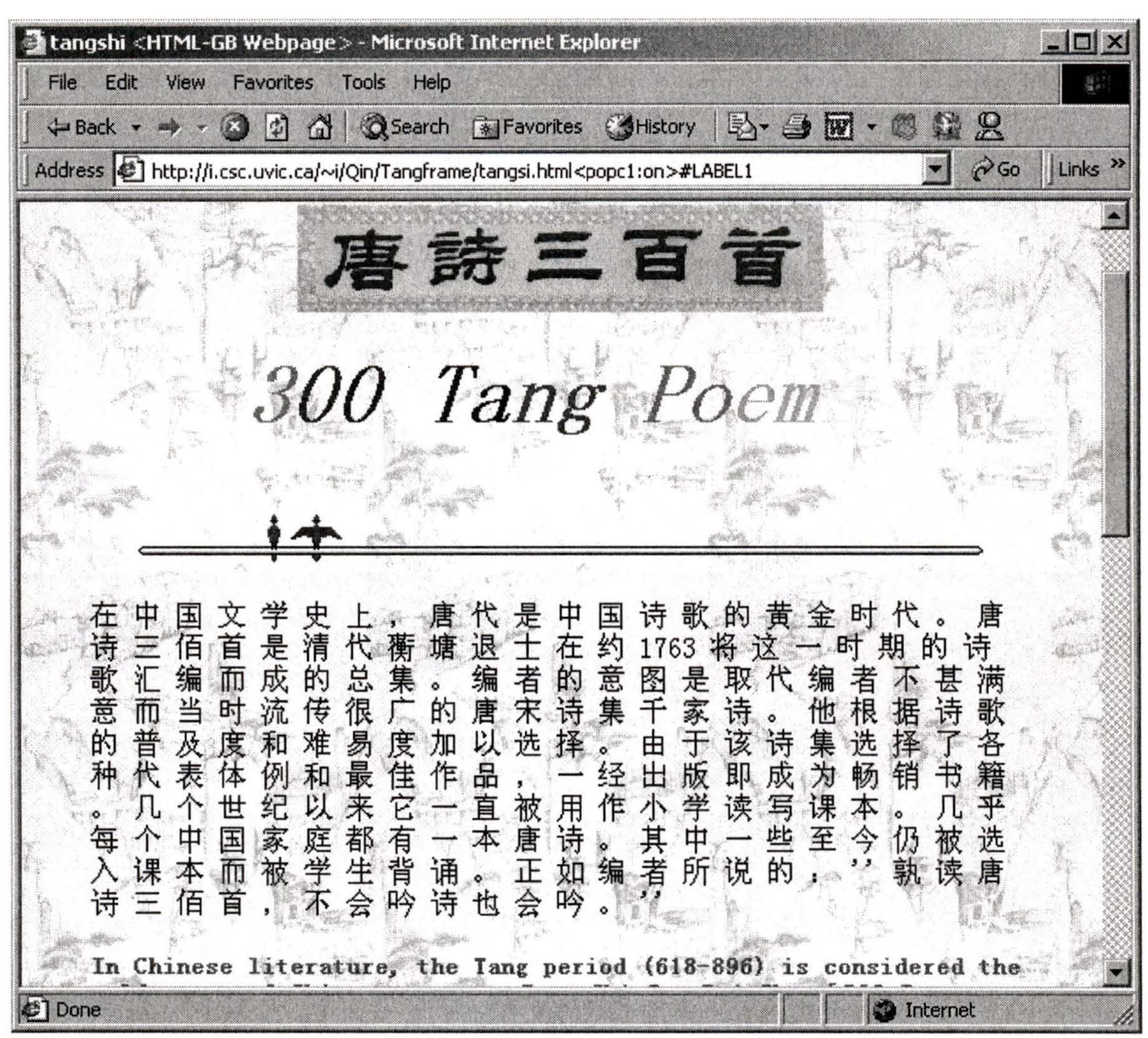


Figure 3: tangsi.html in the pop1:on version

Figure 4: wangep.html in vanilla

Figure 5: wangep.html in the pop131:1version

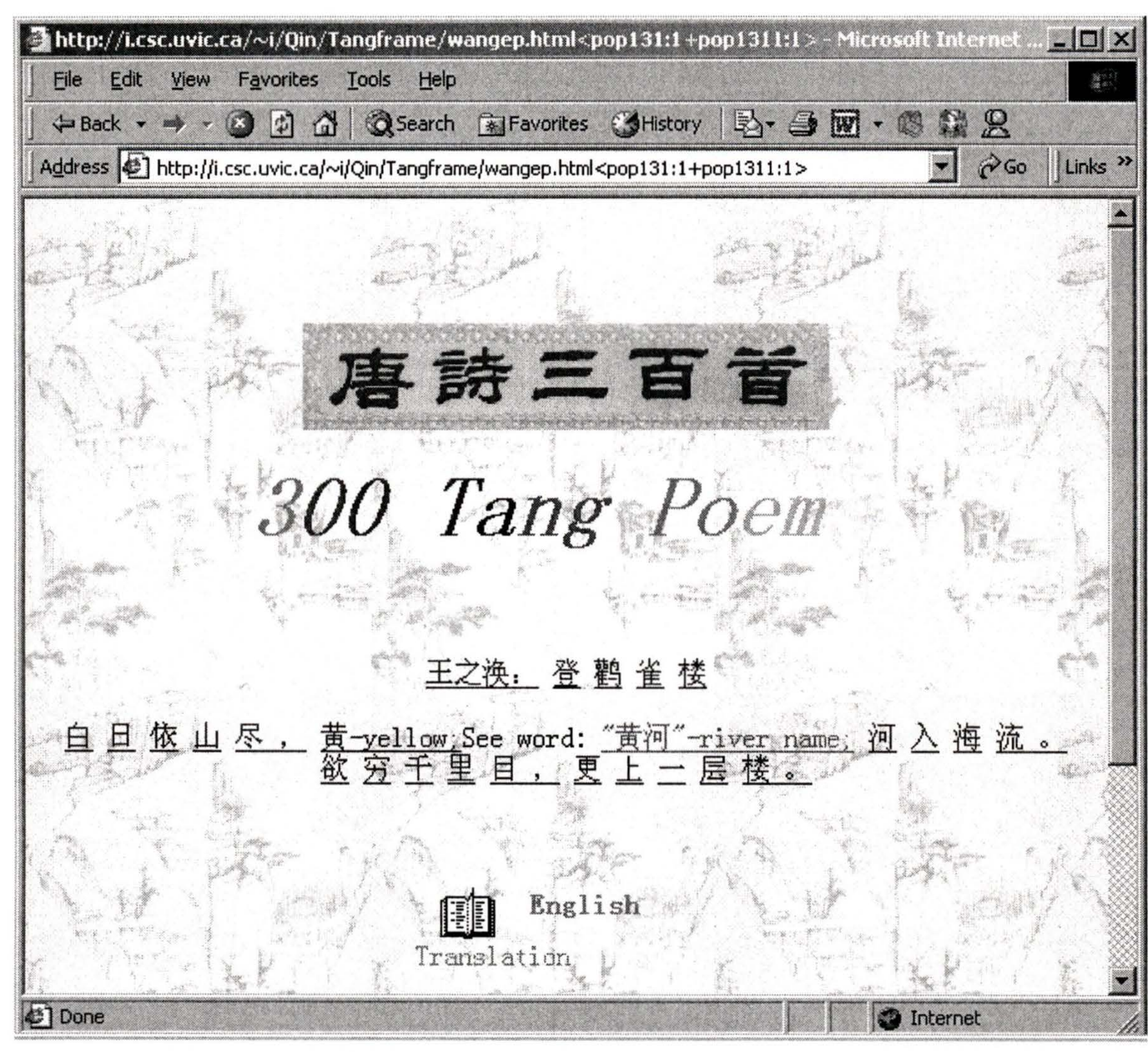
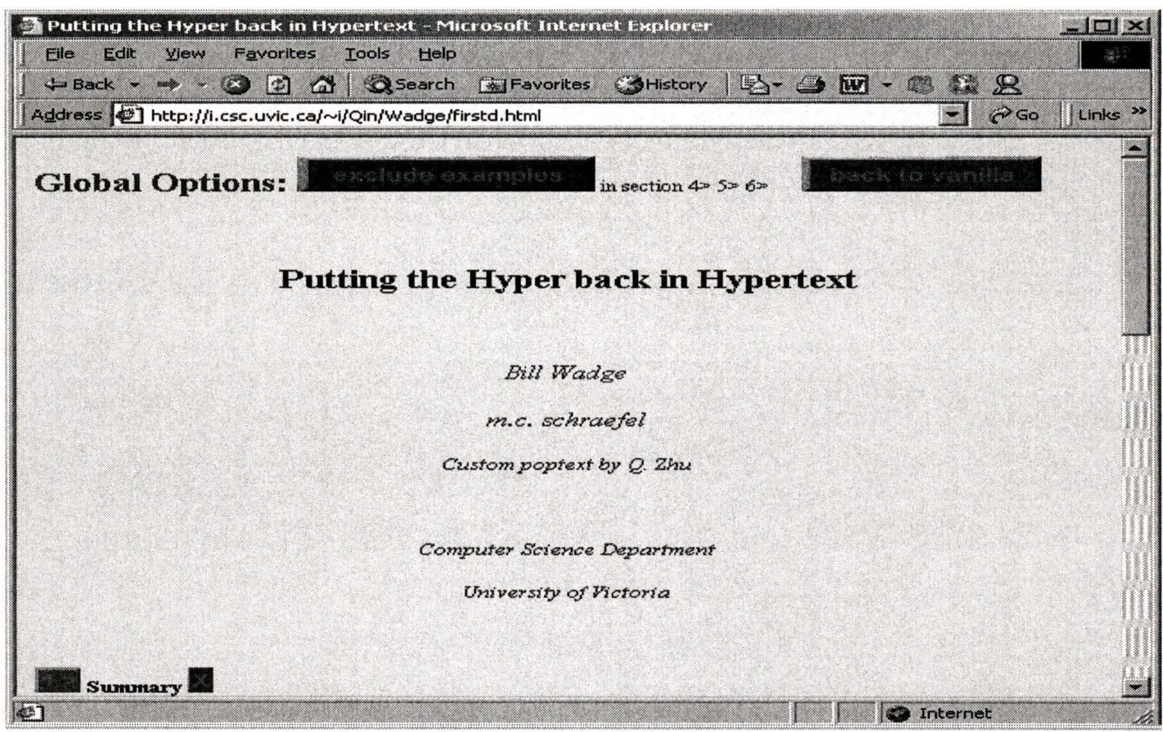


Figure 6: wangep.html in the pop131:1+ pop1311:1 version



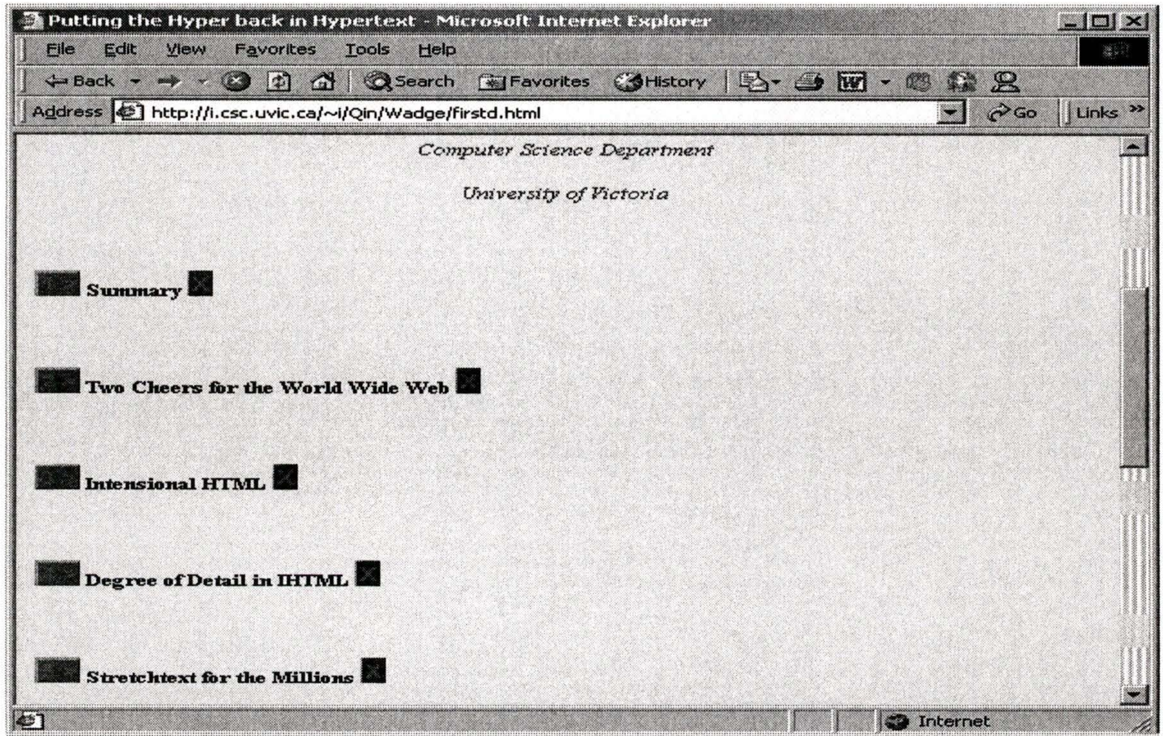


Figure 7: firstd.html in vanilla version

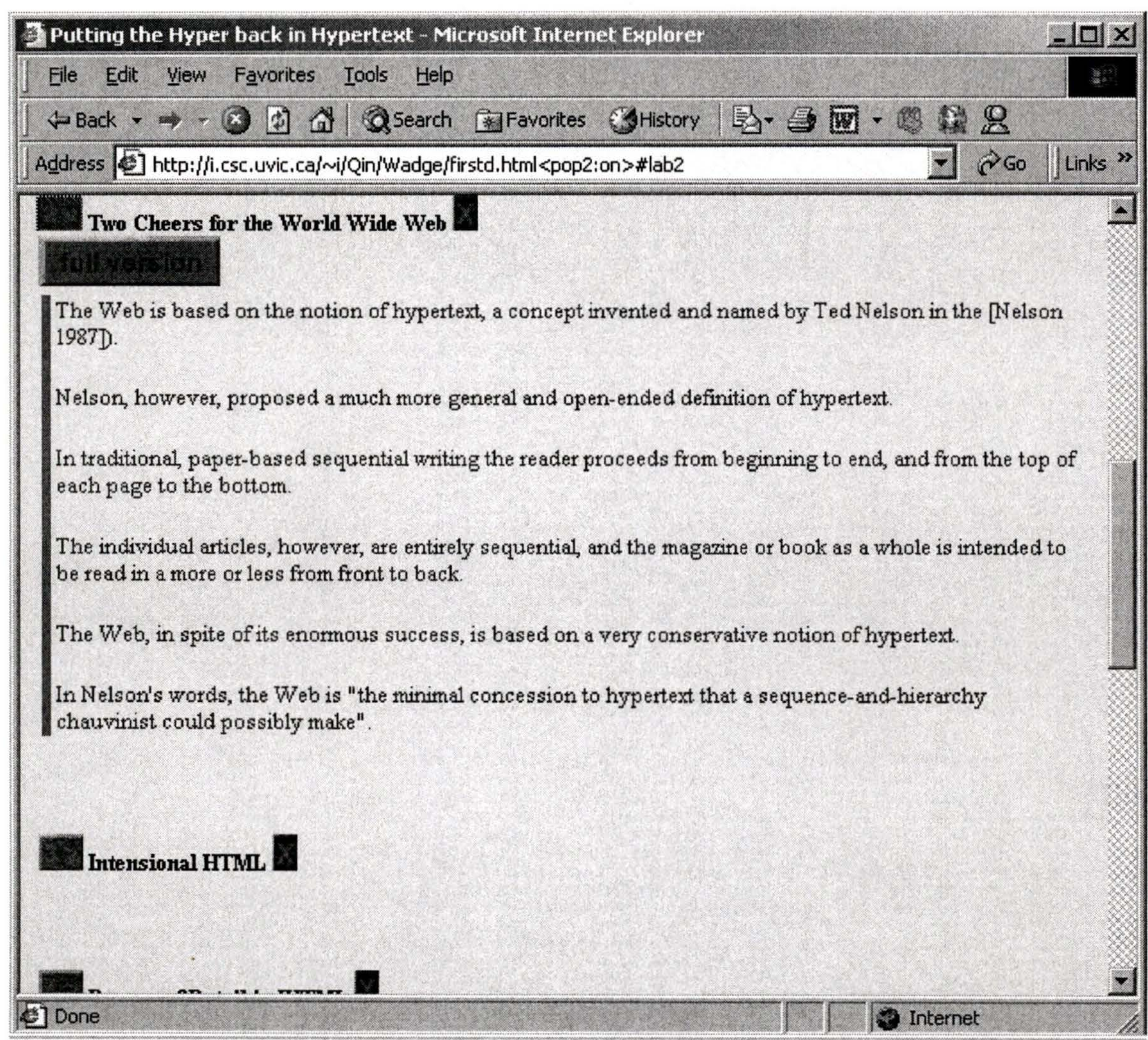


Figure 8: firstd.html in pop2: on version

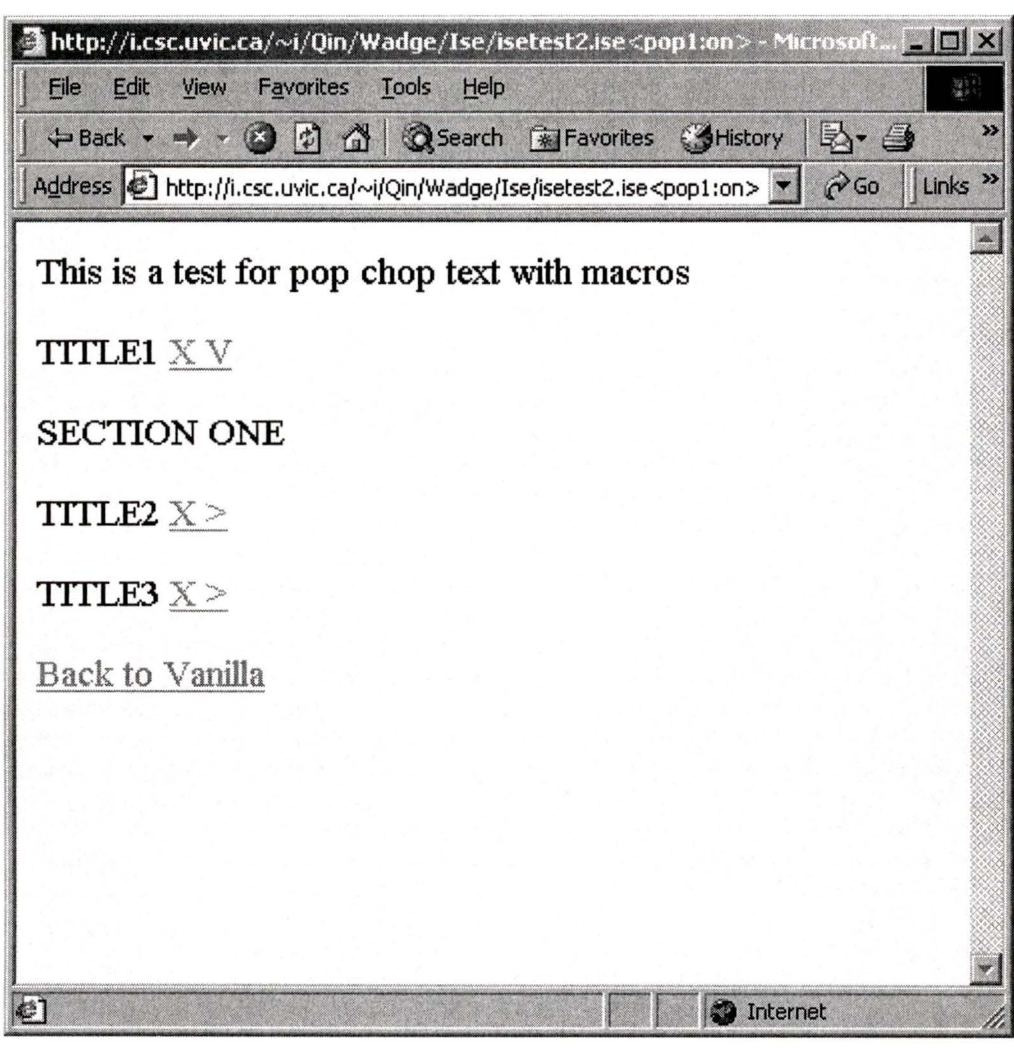


Figure 9: pop/chop text in ISE

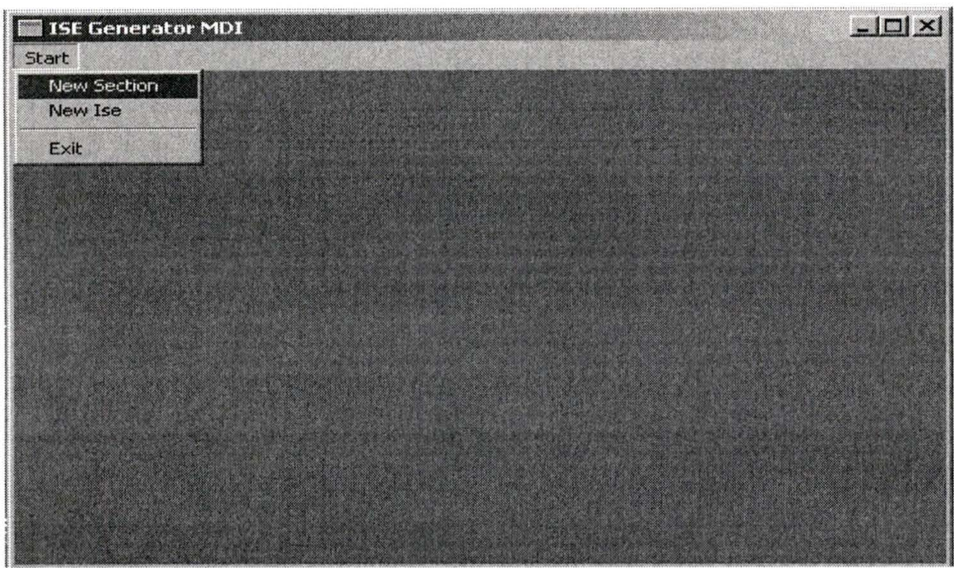


Figure 10: IWAG for pop/chop text. There is no child window in the client window at the beginning

Initmenu is displayed in the *frame window.*

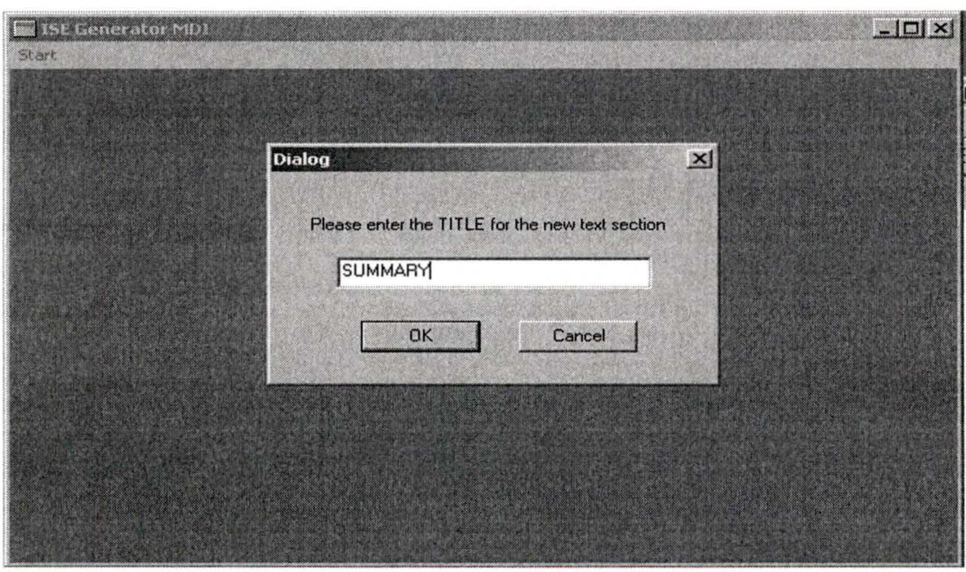


Figure 11: A dialog box pops up to ask the user to input the section title

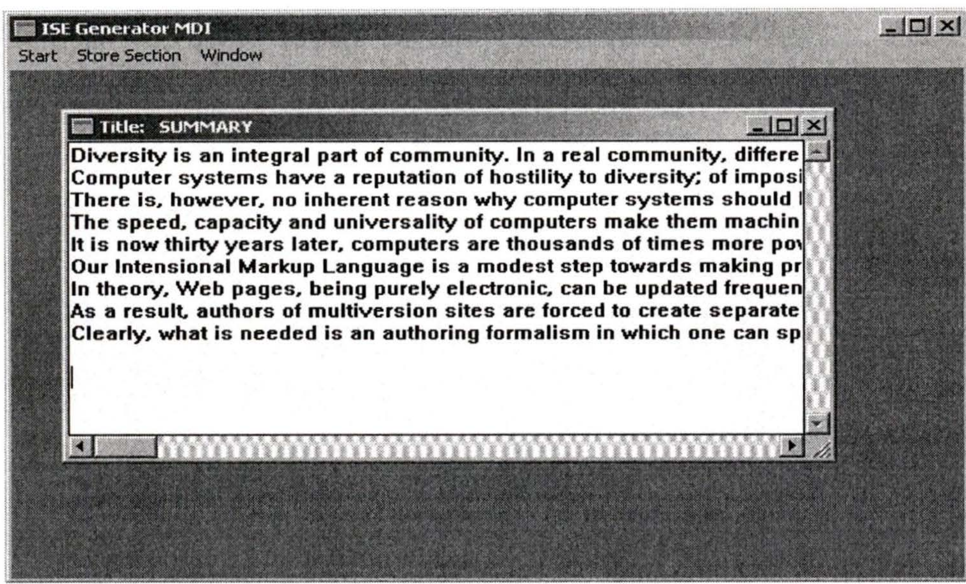


Figure 12: The *section window* includes an *edit window*, in which a user can input his/her section text

Sectionmenu is displayed in the *frame window.*

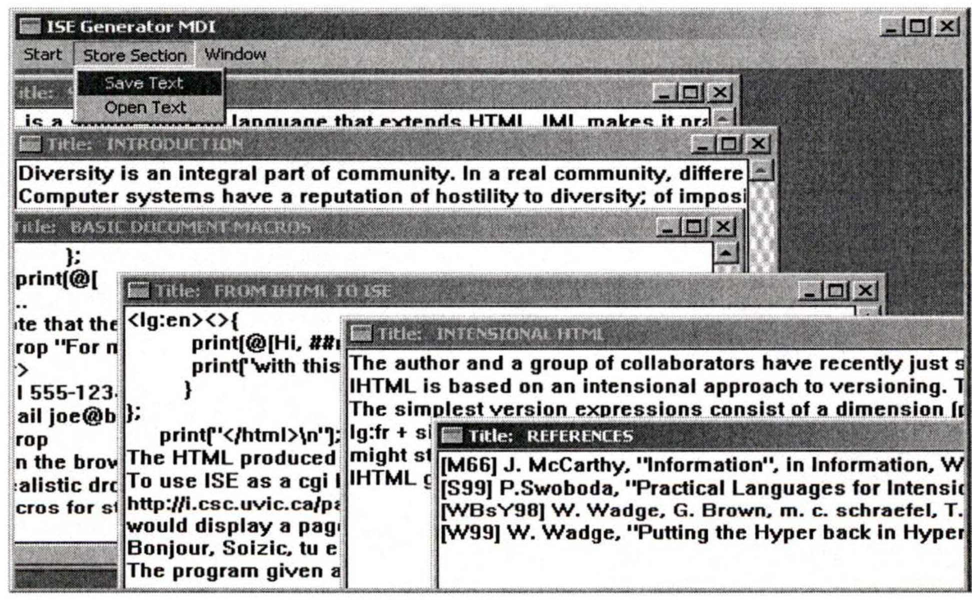


Figure 13: There are six *section windows* in the client window.

Sectionmenu is displayed in the *frame window.*

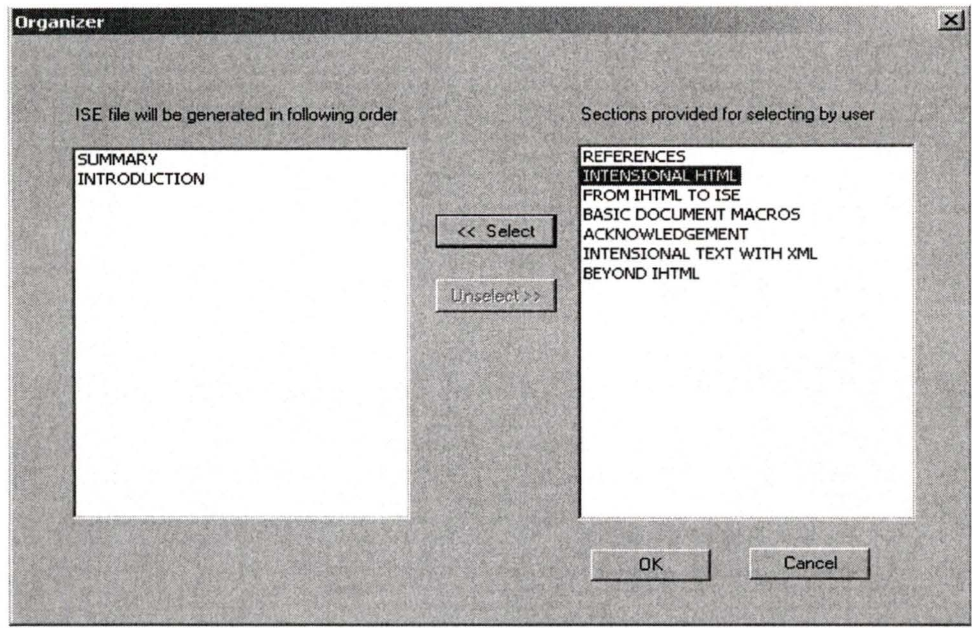


Figure 14: The organizer dialog box provides the title list for the user to select options.

The *section title list* is in the `IDC_SECTION_LIST_TEMPLATE` area, at the right side of the dialog box. By focusing on one *section title* and clicking the *Select* button, this *section title* will be moved to *ISE list* in the `IDC_ISE_LIST_TEMPLATE` area, on the left side of the dialog box.

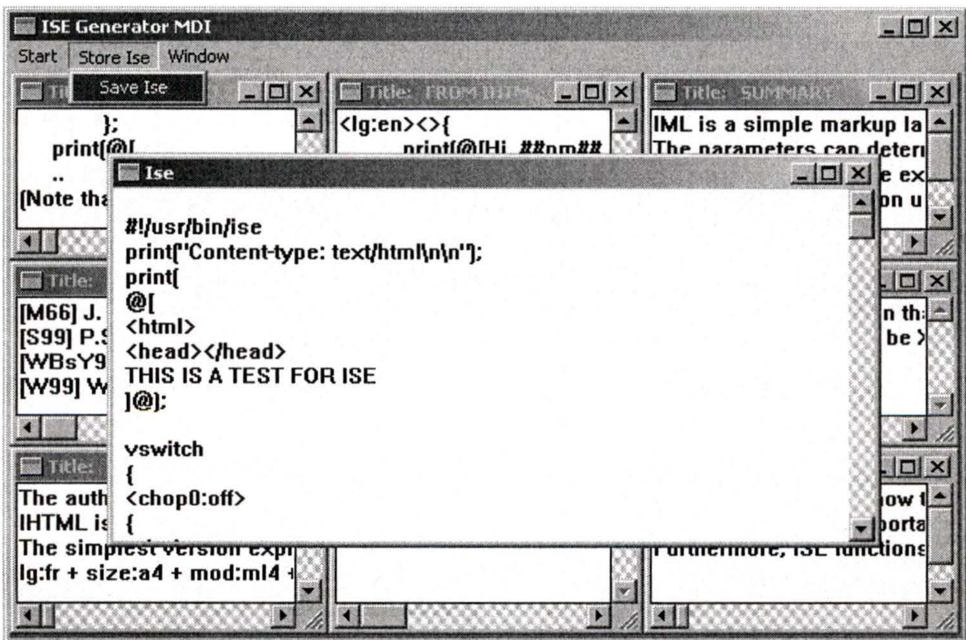


Figure 15: The result – an ISE file is output into the *ISE window.* There are nine *section windows* in the background in the *client window.*

ISEmenu is pop up at the top of the *frame window.*

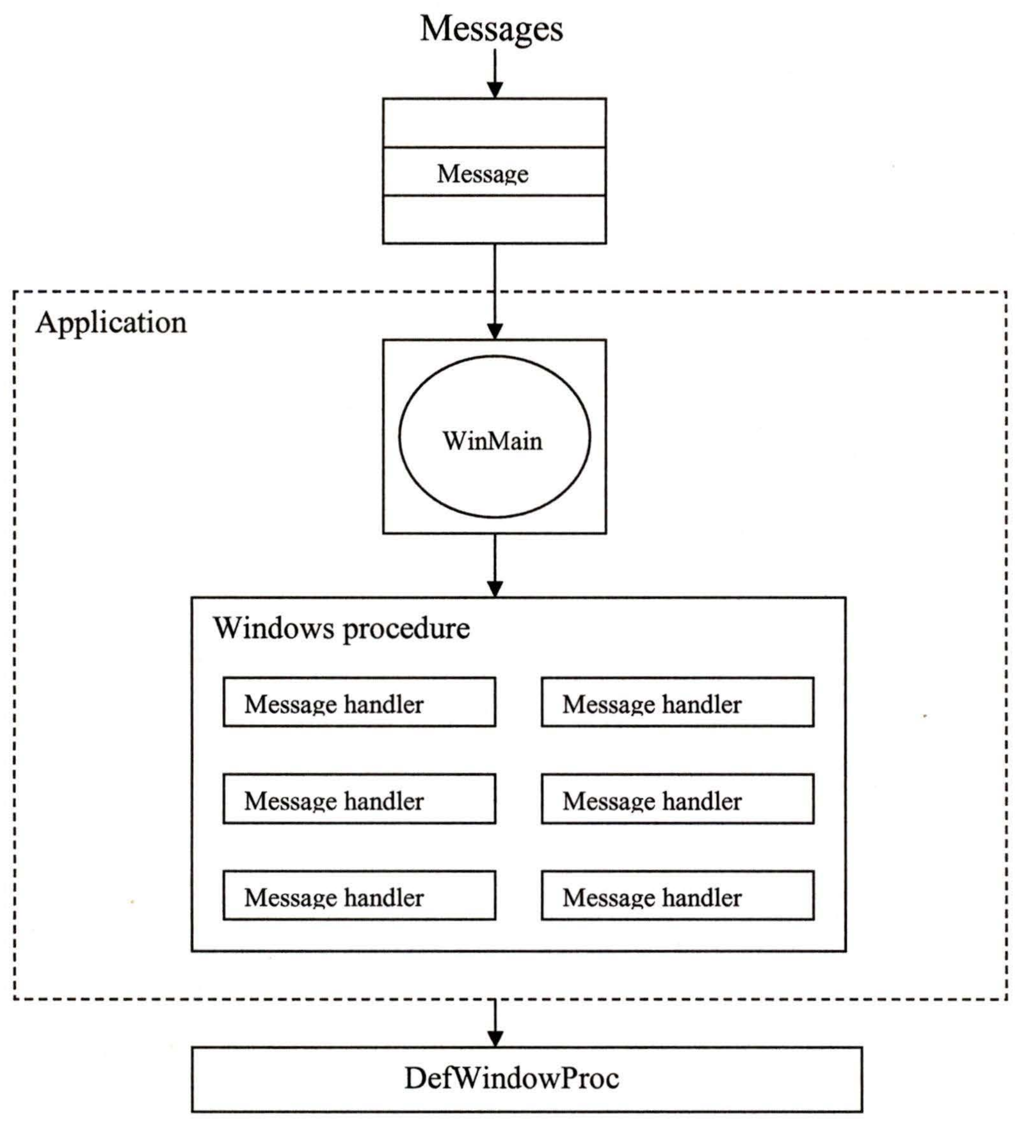


Figure 16: Window Programming Mode

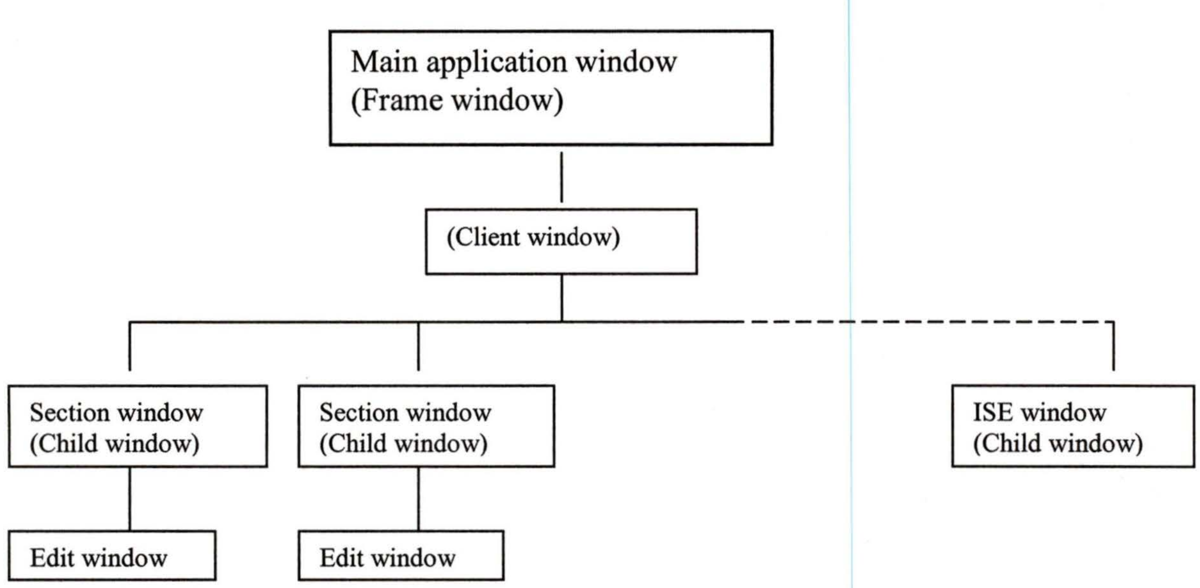


Figure 17: The MDI parent-child hierarchy in IWAG

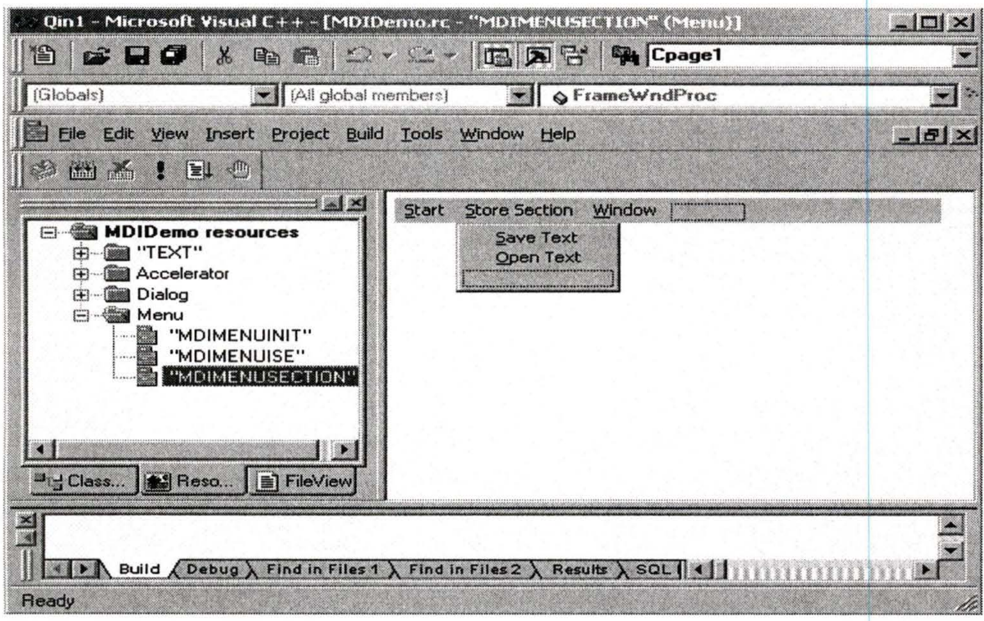


Figure 18: menu resource

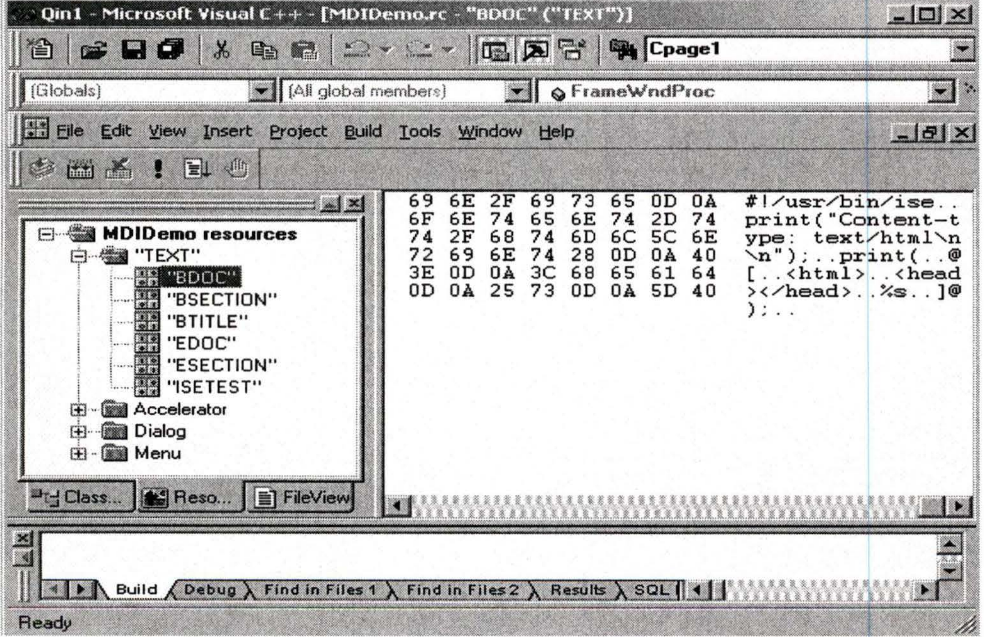


Figure 19: "TEXT" resource

Vita

Surname: Zhu Given Name: Qin

Place of Birth: Shanghai, P.R. China

Education Institute Attended:

Nanjing University of Aeronautics and Astronautics 1985-1989

Degree Awarded:

B.Sc. Nanjing University of Aeronautics and Astronautics 1989

Publications:

Partial Copyright License

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library from any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purpose may be granted by me or a member of the University designated by me. It is understood that coping or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

A Windows GUI for Intensional Web Authoring

Author ______________________

Date Jan 31. 2002