1-1-2003

The design and implementation of an input/output subsystem for a Farsi language search engine

Ronald L Young
University of Nevada, Las Vegas

THE DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT SUBSYSTEM FOR A FARSI LANGUAGE SEARCH ENGINE

by

Ronald L. Young

Bachelor of Science
University of Nevada, Las Vegas
1981

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2003

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

NOVEMBER 21, 2003

The Thesis prepared by

RONALD L. YOUNG

Entitled

THE DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT SUBSYSTEM FOR A FARSI LANGUAGE SEARCH ENGINE

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

The Design and Implementation of an Input/Output Subsystem for a Farsi Language Search Engine

by

Ronald L. Young

Dr. Kazem Taghva, Examination Committee Chair
Professor of Computer Science
University of Las Vegas, Nevada

The question posed in this thesis is whether it is feasible to use commonly available computer hardware and software equipment found in the United States for querying a web-based native Farsi language search engine. A document conversion utility consisting of a C program called html2unicode was constructed to convert the native language web pages into a common format. A Javascript keyboard application and corresponding embeddable web server was developed to make native Farsi language input and output accessible to a search engine. Although there are some issues inherent in the conversion of Farsi web pages into a format suitable for searching, it is possible to use common hardware and software to retrieve Farsi documents.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Thanks to my wife Cindy, for her love and support.

I would like to thank all of the staff at the Information Science Research Institute for their discussions about this work as it progressed. I especially wish to thank Dr. Tom Nartker for introducing me to the world of Optical Character Recognition accuracy analysis during my time at ISRI, and most of all to Dr. Kazem Taghva for introducing me to the field of information retrieval.

CHAPTER 1

INTRODUCTION

This thesis describes the design and implementation of an input/output subsystem for a Farsi language search engine. The Information Science Research Institute (ISRI) is developing this Farsi language search capability as part of its research into cross-language information retrieval. The Farsi language was also chosen due to its relevance in the present day geopolitical climate. Additionally, Farsi also presents several interesting issues associated with the conversion and processing of documents using standard English language equipment and software tools.

Farsi is the official language of Iran and one of the two official languages in Afghanistan (Pashto is the other). In addition, Farsi is spoken in Tajikistan and parts of Uzbekistan [13]. There are more than two million Iranians living in the United States, and the Farsi language is taught at many universities and institutions in North America and Europe [21].

Farsi is written using an expanded form of the Arabic script and is a member of the Indo-Iranian family of languages [10]. Other languages which use variants of the Arabic script include: Dari, Urdu, Pashto, and Arabic to name a few. Since these languages are all written using variations of the same script, many of the issues (and their solutions) that are discussed in later chapters can easily be expanded to include these languages.

Languages using the Arabic script share the following characteristics: they are entered from a keyboard in left-to-right order (logical order) but are displayed on output devices in right-to-left order (presentation order). The script is written in a cursive manner such that the characters are joined together. Each character may be written as one of four possible presentation forms ("isolated", "initial", "medial" or "final") depending on its position in a

word. Characters may be combined with certain modifiers called "harekats", which behave similarly to the diacritical modifiers found in Latin based languages like French and Spanish.

This thesis describes these issues in the following four chapters:

Chapter 2 provides an introduction to the concept of a standardized document encoding mechanism called Unicode [2] and how it can be applied to a Farsi language document collection. A major issue when dealing with web-based Farsi language documents is the lack of standardized character set encoding. Without a standard encoding method, search and retrieval of Farsi language documents becomes very unreliable.

Chapter 3 explains the issues that are specific to the conversion and processing of Farsi language documents. These issues include conversion between the logical and presentation text layouts into a common form, font encoding standardization (conversion of different character presentation forms into a single unicode encoding). Farsi also imposes some language specific rules regarding how certain character combinations should be displayed, care must be taken during the input conversion process to include an indication to the output processor informing it of the special display requirements.

Chapter 4 describes the methods that were developed to address Farsi language issues and to allow the input and output of native Farsi words using Latin character based computer equipment and software. A web browser keyboard applet was developed to allow the entry of Farsi words using either the mouse or the computer keyboard. A "type foundry" utility was developed to generate bitmap images of Farsi letters, words or lines for display on computers that would not otherwise be capable of displaying the Farsi text. A third mechanism was developed that accepts the keyboard applet text and converts it into a form suitable for processing by a search engine's query processor.

Finally, Chapter 5 presents the overall results regarding the feasibility of this approach in processing Farsi documents and query text. It states the conclusion of this study and offers possible areas for further research.

CHAPTER 2

A STANDARDIZED DOCUMENT ENCODING MECHANISM

Digital computers internally represent information as a collection of binary digits (bits), zero and one values. The number of bits and operations that the computer uses when accessing a data value determines the value's valid contents and type. Probably the most common data type in use is the "byte". Although, the size of a byte may be arbitrary, on modern computers it is normally 8 bits. An 8 bit byte can contain a single value in the range between zero and 255.

The meaning of a byte is determined by the encoding scheme used to associate information with each integer value. In terms of textual information, the information that we are mapping onto the integer values is the alphabet of the language that the text is written in. The complete alphabet is known as a "character set" and the integer values for the letters are known as "code points".

Historically, each computer manufacturer defined their own byte sizes and character sets. For example, Control Data Corporation mainframes used a 6-bit character size with the display code character set, Digital Equipment Corporation mainframes used a byte size of 8 or 9 bits with an encoding called "American Standard Code for Information Interchange" (ASCII) [7], and IBM used an 8-bit byte with the "Extended Binary Coded Decimal Interchange Code" (EBCDIC) [8].

In order to exchange information between computer systems of different manufacturers, it is therefore necessary to know what byte size and character set were used to encode the text.

With the invention and widespread adoption of the personal computer, the encoding

schemes used for English language text documents has standardized into USASCII, an updated version of the original ASCII standard [9]. Throughout the rest of this document, the term ASCII will be used in referring to both ASCII or USASCII.

EBCDIC is still widely used by IBM mainframes to allow for compatibility with legacy software applications.

Example English Language Encoding

To illustrate the effect of different encodings on the same text, consider the text phrase "Information Retrieval". Encoding it in ASCII and EBCDIC results in:

ASCII	73 110 102 111 114 109 97 116 105 111 110 32 82 101 116 114 105 101 118 97 108
EBCDIC	201 213 198 214 216 212 193 227 201 214 213 64 217 133 163 153 137 133 165 129 147

Table 2.1: ASCII and EBCDIC CHARACTER ENCODING

As is obvious from this example, even though these two sequences represent the same phrase, the computer system will not recognize them as such, since the numeric sequences are not equal. The ability to determine if two words are equal, is of fundamental importance when searching and retrieving documents.

The techniques necessary to prepare English language documents so they are suitable for retrieval have been well studied by Salton [14], Frakes[4] and others. While the specific methods used are beyond the scope of this thesis, they include case folding (converting all alphabetic characters to the same case) and stemming (identifying the root of a word).

ASCII fully represents the characters used when writing English language text, but as computer use spread to other non-English language countries, national extensions to the

ASCII character set were developed. Since ASCII only uses the lower 128 code points, these national extensions usually involved assigning the new letters to the code points between 128 and 255. The various 256 character encodings are commonly referred to as "code pages".

With the advent of the World-Wide Web, the process repeated itself. The technology used to access the web was developed around the ASCII character set with the ability to substitute different code pages by adding a special "<FONT FACE>" tag into the web content.

By using this technique, the web user who wished to view the encoded page, had to make sure that the font file covering the code page was loaded onto their computer, and if someone wanted to create a web page using the encoding, they would also need to know the code point assignments.

Example Farsi Language Encoding

Consider the Farsi phrase: "آیت‌الله خامنه‌ای" (Translation: Ayatollah Khameni - the person who currently holds the office of Supreme Ruler of Iran). Table 2.2 shows the results of encoding the phrase using two of the common character sets encountered in the study. These two character sets are known as CP1256 (Microsoft Windows Arabic code page) and Persian Nimrooz (a proprietary encoding used by the Hamashari Newspaper web site). These encodings are discussed in more detail in the next chapter.

Microsoft CP1256	194 237 202 199 225 229 32 206 199 227 228 228 199 236
Persian Nimrooz	141 254 150 144 243 243 249 255 160 145 245 247 249 144 253

Table 2.2: CP1256 and PERSIAN NIMROOZ ENCODING

Documents written using the Arabic script presents two problems with extracting text that are not found in English language documents. The shapes of Arabic letters can be different depending on their location in the word and not all of the shapes can fit in the upper 128 code points of a 256 letter font.

UNICODE Encoding

The genesis of the Unicode Consortium began in 1987 with informal discussions between staff members of Xerox Corporation and Apple Computer. [1] These discussions were the result of difficulties encountered when converting software applications to support non-English languages, particularly Japanese.

The result of these talks was the Unicode Standard. Unicode basically started with the ASCII code points and expanded them to include code points for most of the languages in use world-wide. The most frequently used subset of the Unicode standard provides a 16-bit encoding for these characters. By using a 16-bit encoding allows for the possibilities of 65536 different code points. This encoding is commonly divided into 256 character code pages for various languages. For this research study, we are interested in the U+0600-06FF (Arabic), U+FB50-FDFF (Arabic Presentation Forms-A), and U+FE70-FEFF (Arabic Presentation Forms-B) code pages.

The notation "U+####" signifies the hexadecimal range of Unicode code points. For example: U+067E (1662 in decimal) is the code point assigned to the isolated presentation form of the Farsi letter PE (پ) . The U+0600-06FF range contains the base letters for languages that use the Arabic script. For our purposes, all of the letters that are processed as input data will be converted to a code point inside of this range. The other two ranges, contain the code points for the other presentation formats of the Arabic letters.

Another result of the work by the consortium members was the incorporation of Unicode support into the TrueType/Opentype glyph rendering standard. This was important because

while Unicode provides a standardized method of manipulating code points, it does not address how to display them on output devices. TrueType/Opentype provides the information necessary to render the glyph images correctly on the output device.

Two HTML Web Page Encoding Schemes

Unlike ASCII text, Unicode encoded strings should not be used by software tools that are not Unicode aware. To do so can result in data corruption. To allow ASCII based tools to be used, several schemes were developed to encode Unicode code points so they can pass through ASCII-based tools unchanged. The two representations that are of interest are: HTML escape encoding, and UTF-8.

The HTML escape encoding scheme is very simple: convert the code point value into a character string containing the decimal representation of the code point, prefix the string with "&" (an ampersand) and suffix the string with a semi-colon. So the code point U+067E would be represented as "&1662;". This encoding scheme has the advantage of being very simple to implement, but at the possible cost of expanding the size of a document up to 350% (from 16 bits per Unicode character to 56 bits (7 ASCII characters)).

To minimize the amount of overhead required, another encoding scheme called the "Unicode Transformation Format-8" (UTF-8) [22] was developed. UTF-8 is an "8-bit clean", lossless encoding of Unicode characters. This allows the encoded characters to pass through programs that assume all character data is made up of 8-bit values.

U+0000 - U+007F	0xxxxxxx
U+0080 - U+07FF	110xxxxx 10xxxxxx
U+0800 - U+FFFF	1110xxxx 10xxxxxx 10xxxxxx

Table 2.3: 16-BIT UNICODE UTF-8 ENCODING [5]

The part of the encoding scheme of interest is shown in Table 2.3. Where the xxx bit positions are filled with the bits of the character code point value in binary representation. For example, the Unicode character PE (پ , U+067E) has a binary representation of 00000110 01111110. Converted into UTF-8 results in the two 8-bit patterns: 11011001 10111110. The resulting bit values can pass through ASCII only applications unchanged.

CHAPTER 3

FARSI LANGUAGE ISSUES

In addition to the character set encoding issues described in the previous chapter. There are several other issues that are related to documents written in Farsi (and Arabic script in general). These issues include, logical versus presentation order, character normalization, and character layout & shaping.

Logical versus Presentation Order

Logical order is the order that a person would enter characters on the keyboard. Presentation order, also known as reading order, is the order in which those characters are displayed on an output device. For English language documents, the logical and presentation orders are the same, left-to-right. For Farsi documents, the logical order is left-to-right while the presentation order is right-to-left.

While handling the difference in logical versus presentation order is straightforward for characters entered by the keyboard, it may be difficult to automatically determine the presentation ordering for HTML pages included in the document collection. This is due to the fact that depending on the software used to create the web page, the transformation of the logical ordering may or may not have already been done. In processing the pages for the Farsi-1 collection, two kinds of pages were encountered: the unicode encoded pages used the expected left-to-right ordering, and the proprietary encoded Persian Nimrooz pages contained text with right-to-left ordering.

Character Normalization

During processing of input document text, "combined" characters may be encountered. A combined character is a base character from the Unicode Arabic code page that has been modified by the addition of one or more harekats. Harekats are used to indicate the presence of vowels and/or repetition of consonants. Text written by native Farsi speakers normally only includes vowels when necessary to prevent ambiguity. Because of the possibility that harekats may be present on a query term entered from the keyboard, but not in the collection's document text, it is important that the deconstruction of a combined character into its individual components be performed in a consistent manner. For this study, the unicode conventions of processing combined characters will be used.

This convention is that the base character will appear first in the text's logical order, followed any harekat modifiers. The keyboard applet also enforces this behavior for text entered manually. This convention simplifies the term matching routines used by the search engine. When comparing two terms, the routines will match the harekats only if they appear in both, and will ignore them otherwise.

An additional issue related to character normalization, is in recognizing that the different presentation forms of a letter are equivalent for comparison purposes. To address this issue, all of the presentation forms for a letter will be mapped to the letter's isolated code point in the U+0600 unicode code page. It is during this mapping, that some Farsi specific hints on which presentation form should be used for display may also be inserted. These hints, known as zero-width joining (ZWJ, U+200D) and zero-width non-joining (ZWNJ, U+200C) characters are discussed further in the next section.

The final issue uncovered in this study, is that there are two code points that may be represented by the same glyph. The alef maksura (U+0649) and the Farsi Ye (U+06CC) share the same glyph (ی). This is addressed by mapping the Farsi Ye to the U+0649 code point.

Character Layout and Shaping

Character layout and shaping prepare the Unicode text strings so they can be displayed on an output device. They can be thought of as the inverse of the logical & presentation ordering and character normalization operations for input text. There are three major steps involved in preparing Farsi text for display on an output device:

- Conversion from logical to presentation order.
- Construction of combined characters from their normalized components. Also, substitute ligatures for common letter sequences where possible (i.e. lam (ل) followed by alef (ا) becomes the ligature (ﻻ)).
- Selection of the correct presentation form for each character.

The Unicode Standard[2] provides generalized mechanisms to accomplish each of these steps. Unicode provides a bidirectional algorithm to correctly layout text. The bidirectional algorithm provides the ability to layout intermixed left-to-right and right-to-left ordered text. Since this study's purpose is to provide input and output for a Farsi language search engine, we can simply reverse the ordering of the characters instead of using the full bidirectional algorithm.

Similarly, we can simplify the mechanism used to select the presentation form for each letter. Since we are only working with the Arabic script, we can use a variation of a simple table lookup described by Shaikli [16] instead of using the more involved method provided by the Unicode standard. The modifications that were made to Shaikli's algorithm was the addition of letters specific to Farsi and the ability to override the default shaping behavior with the use of zero-width joiner (ZWJ, U+200D) and zero-width non-joiner (ZWNJ, U+200C) characters.

The ZWJ and ZWNJ unicode code points do not have widths, and therefore do not have a glyph, assigned to them. Their purpose is to override the default shaping behavior between two characters.

If two characters normally would be joined, the insertion of the ZWNJ character between them will disable this joining. The first character will be converted into a final presentation form and the second character will be converted to an isolated or initial form. The insertion of a ZWJ character between two normally joining characters will have no effect.

If two characters normally would not be joined, the insertion of the ZWJ character between them will force a joining. The first character will be converted from an isolated/initial form to an initial/medial form, and the second character will be converted from an initial/medial to a medial/final form. The insertion of a ZWNJ character between two normally non-joining characters will have no effect.

The algorithm uses the Farsi character tables shown in tables 3.1 and 3.2 to select the presentation form for a letter, or the base character for a combined letter, based on the value of the letter and the presence or absence of letters before (*prev*) and after (*next*) the character being shaped. If *prev* and *next* would fall outside of the bounds of the text string, they are set to reflect no character present.

- if both *prev* and *next* reflect no character present, select the isolated form.
- if *prev* is present and *next* is not present, tentatively select the final form.
- if *prev* is not present and *next* is present, tentatively select the initial form.
- if both *prev* and *next* are present, tentatively select the medial form.
- if there is not an entry in the table for the tentatively selected form, select the isolated form instead.

Additional checking is performed to substitute ligature characters for common character sequences.

Letter	Presentation Form			
	Isolated	Initial	Medial	Final
sadda	ّ	harekat / repetition		
ze.bar	ـَ	harekat		
zer	ـِ	harekat		
pesh	ـُ	harekat		
alef w/ madd	آ			ـآ
alef	ا			ـا
vav	و			ـو
ye	ی			ـی

Table 3.1: FARSI LETTERS (VOWELS and HAREKATS)

Letter	Presentation Form			
	Isolated	Initial	Medial	Final
be	ب	بـ	ـبـ	ـب
pe	پ	پـ	ـپـ	ـپ
te	ت	تـ	ـتـ	ـت
se	ث	ثـ	ـثـ	ـث
jem	ج	جـ	ـجـ	ـج
che	چ	چـ	ـچـ	ـچ
he	ح	حـ	ـحـ	ـح
khe	خ	خـ	ـخـ	ـخ
dal	د			ـد
zal	ذ			ـذ
re	ر			ـر
ze	ز			ـز
zhe	ژ			ـژ
sen	س	سـ	ـسـ	ـس
shen	ش	شـ	ـشـ	ـش
sad	ص	صـ	ـصـ	ـص

Letter	Presentation Form			
	Isolated	Initial	Medial	Final
zad	ض	ضـ	ـضـ	ـض
teyn	ط	طـ	ـطـ	ـط
zeyn	ظ	ظـ	ـظـ	ـظ
eyn	ع	عـ	ـعـ	ـع
gheyn	غ	غـ	ـغـ	ـغ
fe	ف	فـ	ـفـ	ـف
qaf	ق	قـ	ـقـ	ـق
kaf	ک	کـ	ـکـ	ـک
gaf	گ	گـ	ـگـ	ـگ
lam	ل	لـ	ـلـ	ـل
lam lig.	لا			ـلا
mem	م	مـ	ـمـ	ـم
noon	ن	نـ	ـنـ	ـن
vav	و			ـو
he	ه	هـ	ـهـ	ـه
ye	ی			ـی

Table 3.2: FARSI LETTERS (CONSONANTS)

CHAPTER 4

SOLUTIONS

This chapter describes the program solutions that were implemented to provide Farsi language capabilities for document and keyboard input/output. The following portions of the input/output subsystem are described: utilities for both document input and output processing, character/word image generation (the "type foundry"), a keyboard input/output applet, and an embedded HTTP server interface module for inclusion into a search engine.

Since the initial test collection consists only of HTML documents, all of the processing utilities developed in this study only process documents in this format. However, the processing methods can easily be extended to support additional formats like PDF, optical character recognition software output files, or Microsoft Word document formats using techniques similar to those described below. The only requirement is that the output format conform to the standard format described in the next section.

Input Document Processing Utilities

The input document processing utilities are a set of stand-alone programs that accept documents in a variety of formats and converts them to a standard format that can be processed by the search engine. The prototype I/O system utilities produces files containing a unicode byte-order-marker (BOM, u+FEFF) followed by zero or more 16-bit unicode code points. The unicode BOM and its counterpart u+FFFE identifies the byte ordering of the code points contained in the file. The u+FEFF will be read as the first unicode character if the host computer system stores binary data in "big-endian" (most significant byte first) order. If the host reads the BOM as u+FFFE, it stores binary data in "little-endian"

(least significant byte first) and indicates that the two bytes of each character needs to be swapped before processing the data. Using the BOM, allows data files to be portable across different computer systems. For example, during the development of the document utilities, two different computer systems were used for testing, an IBM-PC clone (Intel Pentium 4 – little endian) and a Sun Microsystems Blade 2000 (Ultra SPARC – big endian). Because of the presence of the byte order marker, processed document files could be transferred between the systems without having to worry about the byte order.

An input utility was developed, *html2unicode* that converts input HTML documents into the standard unicode file format described above. The *html2unicode* utility is executed in the following manner:

```
html2unicode <URI> [ reverse ]
```

Where <URI> is a uniform resource identifier (URI) identifying the location of the input document to be converted. By using a URI to specify the input document, both web-based text (i.e. http://www.hamashari.org) and disk based text files (file:///home/isri/ron/test.html) can be processed without having to modify the utility. The optional parameter `reverse` controls whether *html2unicode* will reverse the order of the input text to correct the documents presentation ordering. The converted document is written to the standard output.

The document processing utilities use the LIBwww[23] library routines to parse the HTML documents and to process URI arguments. LIBwww allows a program to register a set of callback routines for processing of each syntactic element found in the HTML document. Callbacks can be invoked for the start and end of each HTML tag, for each text element, and for the start and end of a document.

The callback functions for the *html2unicode* utility are defined as follows:

- For each HTML tag, except for <FONT FACE=*name*>, no action is taken or output produced.
- For each HTML <FONT FACE=*name*> tag, the font name is placed on a stack, with the topmost element on the stack becoming the active font face.
- For each HTML </FONT> tag, the stack is popped, and the new topmost element on the stack becomes the active font face.
- For each text string, each individual code point is mapped from the input font into unicode by the use of a translation table identified by the active font face. After this translation, the unicode code points are written (in binary) to the standard output.
- For the start and end of document callbacks, no actions are taken and no output produced.

Output Document Processing Utilities

In addition to the input document processing utilities, two other utilities were developed, *html2unihtml* and *html2imghtml*. They are executed in the same manner as *html2unicode* above.

The callbacks for the *html2unihtml* utility are defined similar to those for *html2unicode*, except that the callback for HTML tags is modified to write the tag to the standard output. This allows for the conversion of document text while retaining the HTML mark-up directives. The converted document can then be displayed with the same presentation format as the original document but using unicode fonts instead of the originally encoded proprietary font.

The *html2imghtml* utility is one of the key parts of this study. It provides the same translation capabilities as the other utilities, but instead of outputting unicode code points, it renders the text into graphic images with a corresponding HTML file to display them. This allows a computer with a web browser but no unicode support to display unicode documents. Because of the image rendering, the input text is converted into lines of unicode and then a separate image file is created for each text line. The type foundry (described in the next section) is used to create the images.

The callback functions for the *html2imghtml* utility are defined as follows:

- For each HTML <FONT FACE=*name*> tag, the font name is placed on a stack, with the topmost element on the stack becoming the active font face.
- For each HTML </FONT> tag, the stack is popped, and the new topmost element on the stack becoming the active font face.
- For each HTML
 (break) tag, the current line buffer is inserted into a linked list for later processing. The line buffer is then emptied.
- For each HTML <P> (paragraph) tag, the current line buffer followed by a blank line is inserted into the linked list. The line buffer is then emptied.
- For all of the other HTML tags, no action is taken or output produced.
- For each text string, each individual code point is mapped from the input font into unicode by the use of a translation table identified by the active font face. After this translation, the unicode code points are stored in a buffer containing the current line.
- For the start document callback, no action is taken or output produced.

- For the end document callback, each entry in the linked list of text lines is sent to the type foundry for rendering and the resulting graphic image is written to a unique filename. A HTML file is written containing an <IMG SRC=*filename*> tag such that when the HTML document is displayed in a web browser, the images of the lines will appear.

Type Foundry

The type foundry library consists of a set of application callable functions that create graphic bitmaps of unicode character strings. There is also a stand-alone application program called *rendermain* that can be called directly from a command shell to generate bitmap images.

These routines are used by the keyboard applet to generate individual Farsi letters for display and by the utility *html2imghtml* to render HTML documents for display on non-unicode capable browsers.

Two external libraries are used: Freetype 1[12] which renders strings using TrueType[3] font files into memory-based bitmap images, and libpng[15] which writes memory-based bitmap images into a disk file in the "Portable Network Graphics" (PNG) format. The process used to generate the images from unicode strings is briefly described as follows:

- A unicode encoded string is stored in memory. When using the stand-alone application this string is retrieved from the command line arguments when the program is executed. When using the library, the unicode string is typically retrieved from the HTML web form (keyboard applet) or from a parsed HTML file (*html2imghtml*).

- A glyph is assigned to each character code point in the string. This is accomplished by calling the Freetype routine TT_Char_Index (returns an index entry for the character code point inside of the font) and the type foundry routine assign_glyphs() (using the font index, selects the glyph for the code point).

- The type foundry function ShapeLetters() is called to select the correct presentation form for the characters in the string. This is implemented using the shaping algorithm described in the previous chapter.

- The type foundry function exe_render_common() is called with the shaped string and a name of the image file where the PNG image will be stored. The purpose of this function is to render the glyph images of each character in the proper presentation order while taking care to insure that each character is aligned and joined correctly.

The type foundry routines are most commonly called from the keyboard applet to generate the images for combined characters as they are entered by the user.

Keyboard Applet

The capability specifications for the user input module of this study required that Farsi language input be able to be entered by the user using either a normal ASCII keyboard and/or the mouse into a web browser. This a secondary goal was to accomplish this without requiring the user to manually load additional software or font files onto the system. Two different approaches were tried to address the issue of accepting Farsi language input from the user.

The first approach was to develop a Java Language applet. While this approach was successful, it suffered from two major shortcomings: not all of the web browsers used in this study support Java (and those that did supported different versions, mostly Java 1.1). While Java uses Unicode encoded strings internally, only recent versions fully support Arabic script strings. The major problem was that Java did not completely handle Arabic script rendering until version 1.4.2. To provide a consistent Java environment, would require the user to download and install the newer Java runtime environment. Another shortcoming was the amount of time required to download the code files for the actual applet and fonts whenever the input web page was initially loaded. These two issues were judged to be sufficiently severe such that the Java language approach was abandoned.

The second approach was to develop a set of browser independent javascript functions coupled with some server based support capabilities to provide Farsi language input. This approach solved the shortcomings presented by the Java applet, but presented different issues that needed to be resolved. The Javascript issues were mainly caused by differences in the event handling models implemented by the two main browsers used: Microsoft's Internet Explorer and the Netscape/Mozilla family.

Web browsers allow Javascript functions to interact with the browser's input/output capabilities by implementing the concept of "events". Events are similar to interrupts in other computer hardware and software. The events that we are interested in are: MOUSEDOWN, KEYPRESS, and KEYDOWN. There are other events available, but these are the ones of interest. Each of the events can have a Javascript function attached to them instead of the default behavior provided by the browser.

In order to capture characters typed on the keyboard, it is necessary to attach a function to the KEYPRESS event for the current document being displayed on the browser. The mechanism to accomplish this is similar to the following javascript code fragment:

```
function ReadKey(e)
{
  whKey = event.which;
  Type(far[whKey]);
  return false;
}
```

Where *event.which* returns the code that was pressed on the keyboard. *Type(far[whKey])* maps the keycode into the Farsi character set and calls the Type() function to actually process the Farsi character. The *ReadKey* function is associated with the KEYPRESS event using something like:

```
document.captureEvents(Event.KEYPRESS);
document.onkeypress = ReadKey;
```

This works as expected in the Netscape and Mozilla family of browsers. They correctly accept all of the keyboard characters including control characters. It does not work with the Internet Explorer browsers. While the above code fragment will capture normal (printing) keyboard characters with slight modifications, not all of the control characters are returned to the *ReadKey* function. This is problematic since the backspace key is normally used in text input functions as a character delete key, but is interpreted by Internet Explorer as the "back page" function to have the browser display the previously viewed page. Internet Explorer processes the browser control characters before calling the KEYPRESS event handler, so the *ReadKey* function is never called for the backspace character. This problem can be overcome by assigning another event handler for the KEYDOWN event when Internet Explorer is used. The KEYDOWN event processes the backspace character (by calling the Type function directly) and ignores any other characters.

With the exception of the above event handler differences, the rest of the Javascript code is the same regardless of the type of browser being used. This is different than most other web applications where common practice is to code for a single browser (Internet Explorer).

The remainder of the keyboard code implements a simplified version of the type foundry functionality. The major difference, in addition to including normal text editing functions, is that the images used to display the Farsi letters are previously generated and stored on the

server. The character images are loaded into the browser when the Javascript code is initially loaded and then used for displaying the shaped Farsi characters. Only base characters are cached in this manner, combined characters are still generated by the server and downloaded as needed.

The keyboard applet tries to conform to the defined standards for Farsi keyboards. These standards are maintained by the Institute of Standard and Industrial Research of Iran (ISIRI). The relevant standard for keyboards is ISIRI 2901 first developed in 1988 and revised in 1994. The original document is written in Farsi, so information that was extracted and translated in an electronic mail message by Pournader[11] was used instead. This information consisted of the keyboard layout and placement of the ZWJ and NZWJ characters.

Figure 4.1 shows the keyboard applet as it is used by the Farsi-1 collection relevancy question data entry application. Note: the Farsi text being displayed in the lower part of the figure are actually graphic images that were rendered by the type foundry utilities described in the previous section.

Embedded HTTP Server Interface Module

This section describes the mechanism that is used by the retrieval engine server component for communication with the client browser. From a design standpoint, there are two basic ways to accomplish this communication: using a stand-alone web server with a set of Common Gateway Interface (CGI) programs, or by using a web service directly embedded into the retrieval engine. Either method could be used to perform the server side processing required. For the purposes of this study, the embedded approach was chosen for the following reasons:

- Ease of integration. By using an embedded server it is very easy to allow access to server functionality from the client by simply attaching a processing routine to a specific uniform resource identifier (URI). The only requirement for the event handling routine is to produce a valid HTML document as output.

- Ease of support. With an embedded server, the issues of installation, operation, and security of stand-along web server and CGI programs are avoided. Development resources can remain focused on the development of the retrieval application instead.

- Allows the retrieval engine to be completely self-contained and easily ported to different host platforms. Since the application doesn't rely on an external web server, it is unaffected by external configuration changes. As long as the host system is operational and network connections are accepted, the application should function correctly. Also, by choosing an embedded server, the application can be easily ported to any platform that has a standard C language compiler and networking based on the Berkeley Unix "sockets" model (practically all modern operating systems provide networking capabilities sufficient to allow the operation of the software produced in this study).

The I/O subsystem described in this study was designed to be activated on demand by entering a "httpserver" statement to the retrieval engine's interactive command line interface. Because of its modular nature, the I/O subsystem may also be activated directly from within a customized retrieval engine.

The embedded web server that was selected for use is called libHTTP and was developed by Hughes Technologies[6]. By using a small set of API routines, libHTTP allows an application to accept and terminate HTTP connections, define valid URI's and their actions, and retrieve values from the client browser. Assuming that the retrieval engine accepts http connections on the local computer system using port 1234, the following URI's are defined:

- *http://localhost:1234.*

  This URI causes the embedded server to display the retrieval engine's "home page". This page is intended to have introductory text describing the function of this particular search engine as well as hyperlinks to other URI's that control the retrieval engine.

- *http://localhost:1234/command.*

  Browser supplied values: command=<*engine command*>

  query=*text string*

  Accept a command line request to be processed by the retrieval engine from the client's web browser. This is done by injecting the entered command line into the command line processor's standard input and capturing the standard output and standard error files produced when the request has been processed by the retrieval engine. Note: while one of the parameters passed by the browser is named *query*, it is not intended for the *command* URI to used to perform normal retrieval queries. The *query* and *process_query* URIs (described below) should be used instead. The intended use for this URI with the query value is to allow for special processing like reporting term cooccurrence and frequency information.

- *http://localhost:1234/exit.*

  Terminate the HTTP connection, cleanup any rendered text and letter images, and exit the embedded web server. When the web server exits, control is returned to either the interactive command line processor (for the research prototype engine) or to the calling function (for a customized retrieval engine).

- *http://localhost:1234/display.*

  Browser supplied value: docid=<*document id*>

  Requests that the retrieval engine display the requested document on the browser. At the current time this URI's action routine is a simple debugging stub. However, since this is a single routine, its functionality can be overridden by the retrieval engine to provide capabilities such as term highlighting, attaching hyperlinks to words, word substitution, etc.

- *http://localhost:1234/renderletter.*

  Browser supplied value: farsi=<*code points*>

  OutStr=<*code points*>

  s=<*point size*>

  The *renderletter* URI requests that the unicode string encoded as hexadecimal digits contained in the *farsi* and/or *OutStr* variables be rendered as an image for display. The *farsi* variable contains the Farsi text encoded without any code point substitutions that reflect shaping. The *OutStr* variable contains these substitutions. The *s* variable contains the requested point size value to be used when rendering the letters. The output for this URI is the actual PNG image, making it suitable for use as source targets in html <IMG> statements.

- *http://localhost:1234/tmp/<filename>.*

  The *tmp* URI provides access to a temporary store of images for use by HTML pages generated by the retrieval engine. For example, if the keyboard applet requests the rendering of a compound character, by using the *renderletter* URI, the generated image will assigned a unique name and placed in the temporary store. This name is also added to an internal list structure for use by the *exit* URI to cleanup these temporary data files.

- *http://localhost:1234/kbd.*

  This URI is intended to allow the user to manually enter Farsi text for submission to the retrieval engine. It is tied to a filesystem directory located in the web server's document tree. This directory contains a html page that loads the keyboard applet and a results page into separate frames. It is configured to submit textual information to the *process_query* URI.

- *http://localhost:1234/kbd/js/<filename>.*

  Allows the keyboard applet loaded in the client's web browser to load the keyboard layout and pregenerated basic letter images from the server. For example the html statement: <img src="*http://localhost:1234/kbd/js/keyboard.png*"> will cause the graphic image for the Farsi keyboard to be display on the browser.

- *http://localhost:1234/query.*

  This URI causes the embedded server to display the retrieval engine's "query page". This page is intended to allow the user to enter Farsi search terms by using the keyboard applet. This page uses the *kbd* URI to activate the keyboard applet such that the entered text will be directed to the *process_query* URI when the submit button is pressed.

- *http://localhost:1234/process_query.*

  Browser supplied value: farsi=<*code points*>
  OutStr=<*code points*>
  s=<*point size*>

  The *process_query* URI requests that the retrieval engine process the contents of the *farsi* variable as search terms against the current document collection. This processing consists of the normal information retrieval type tasks, the resulting html page should be a list of ranked documents that are relevant to the query terms. The *OutStr* and *s* variables are made available for use to the retrieval engine to possibly aid in rendering Farsi text displayed in the result page.

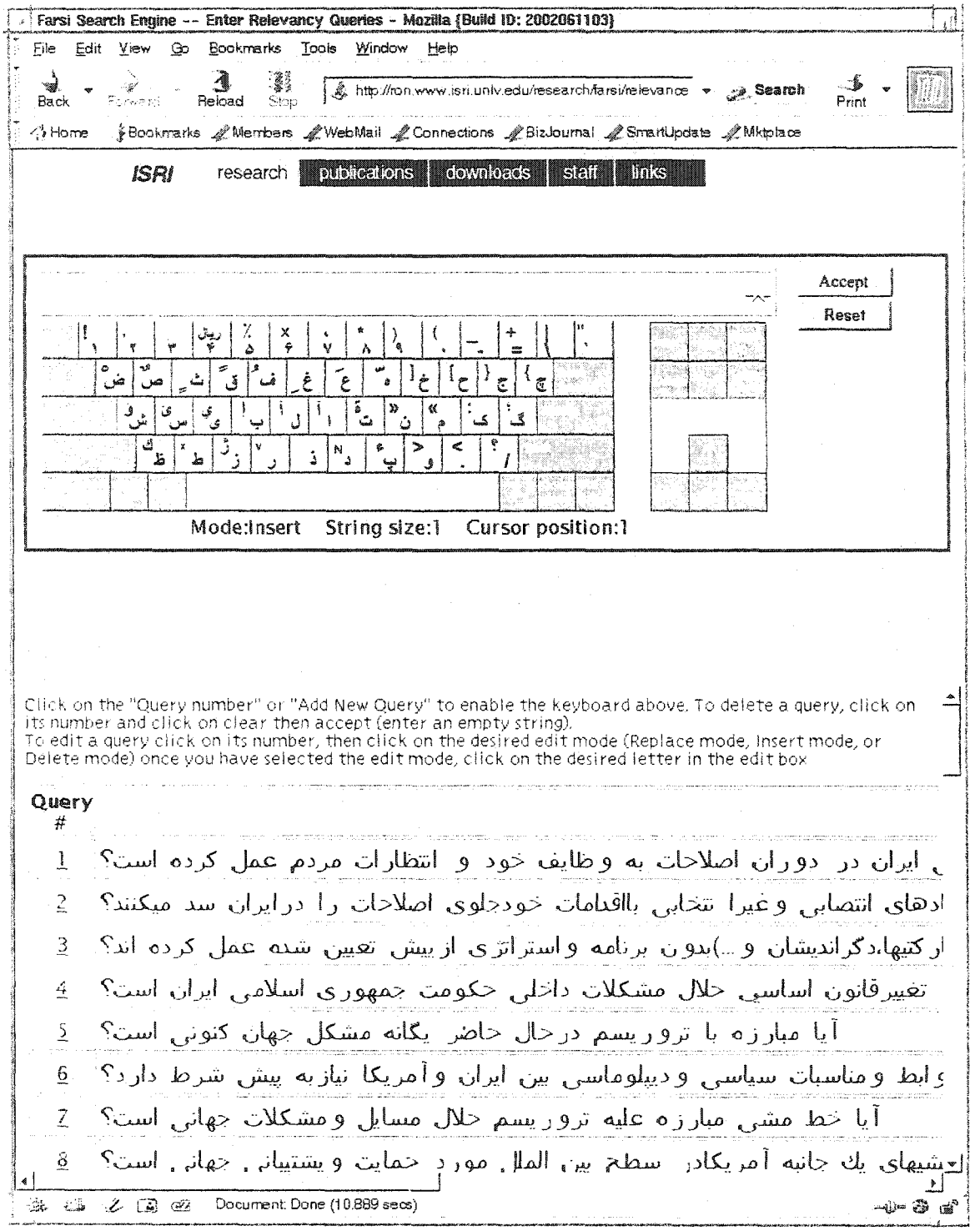

Figure 4.1: Example use of the keyboard applet[17]

CHAPTER 5

CONCLUSION AND FURTHER RESEARCH

The software described in this study has been included in a research prototype Farsi language search engine developed and implemented by staff members of the UNLV Information Science Research Institute (see figure 5.1 for a functional block diagram of the prototype system)[19][18][20]. Based on the preliminary evaluation of the system, we can conclude that it is indeed feasible to use this software to provide Farsi text input and output capabilities without the need for language specific hardware or operating system support.

The evaluation of the prototype system also provides several areas for improvement and further research:

- Multiple script keyboard support.

  While testing the Farsi language keyboard applet, it became clear that using this mechanism of text input to a search engine would be even more useful if keyboard support for additional languages and scripts could be incorporated.

  The current design of the I/O subsystem could be extended for additional keyboards by a slight modification applied to the *kbd* URI described in the previous chapter. Probably the easiest way to add this capability is to replace the single *kbd* URI with one for each of the desired languages, i.e. *farsikbd* for Farsi, *arabickbd* for Arabic, and *pashtokbd* for Pashto. Like the current *kbd*, they would map to a specific directory located in the document root of the embedded web server. The main advantage of this method is that it is simple to add the additional keyboard support. The main disadvantage is that each language would then have its own separate set of applet code,

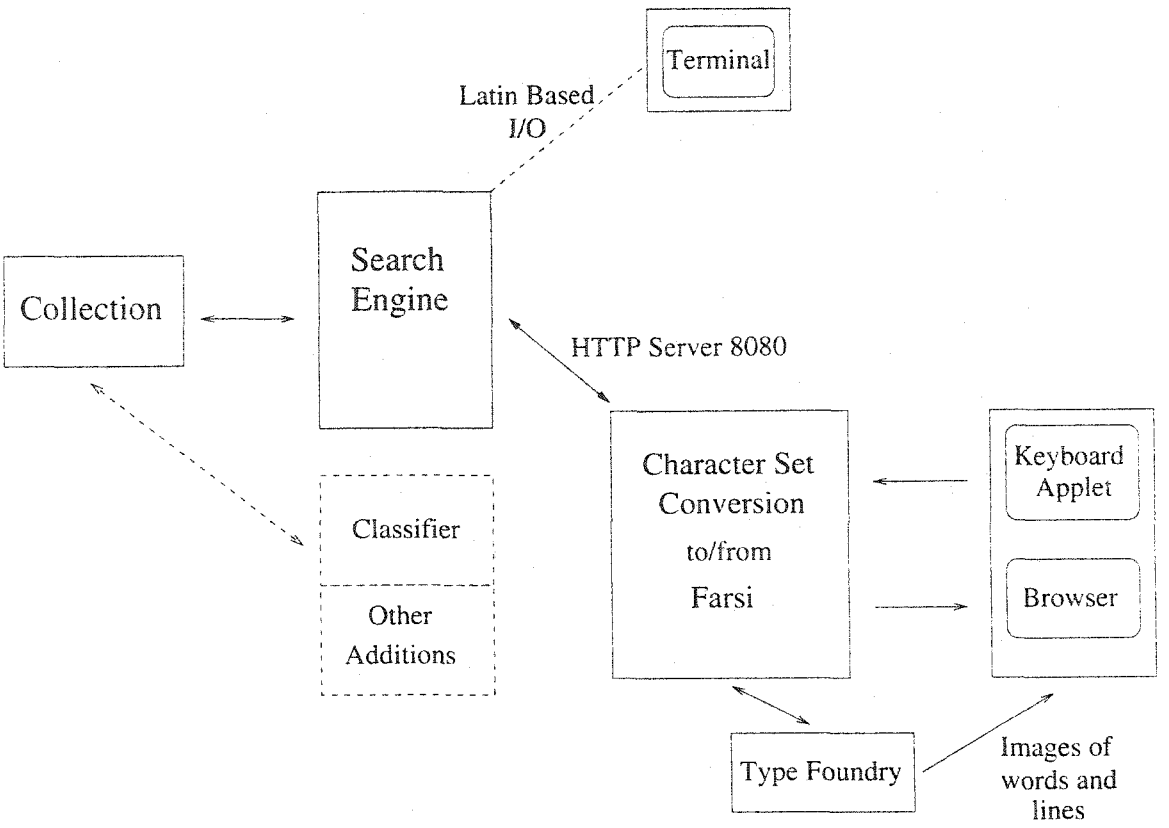

Figure 5.1: Prototypical system architecture[17]

layout images, and letter images.

Another method would be to add an argument specifying which language the applet should process. The advantage to this method is that there is a single set of code and images. The disadvantage is that depending on how different the processing requirements of the supported languages, the complexity and size of the applet code may grow larger. This would not pose a problem if the system running the client browser is fast enough, but on older systems the additional processing and memory requirements may cause a delay in response time from the keyboard applet.

- Improved response time to display documents.

  The major shortcoming uncovered during the prototype development was that there was a significant amount of time required to display a rendered document in the browser. This is caused by two main factors: each document is completely rendered before being displayed and entire lines are rendered into each image, so the amount of data required to be sent to the browser can take several seconds particularly on low-speed dial-up lines.

  A test was performed during the development of the software, instead of rendering entire lines, individual words were used instead. By using words, the time required to download the rendered document was reduced, but the text could be displayed incorrectly. The incorrect display appears to be caused by the differences in word wrapping of text between the English language default for the browser, left-to-right, and the Farsi language requirement of right-to-left.

  A compromise approach was adopted for this study, each document would be rendered into line images as part of creating the collection. This would eliminate the time delay required by rendering the document, but did not address the download time issue.

- Allow the creation of stand-alone CD-ROM based collections.

  The design of the I/O subsystem doesn't place any requirements on accessing collections through the network. By simply using the network loopback interface supplied by the host operating system, the I/O subsystem should be able to run using a local CD-ROM based collection. This assumes, however, that the host operating system is capable of running the search engine application and supporting libraries.

- Decrease the time required to process search queries.

  The current I/O subsystem and search engine prototype is currently single threaded. It is possible to improve response time for searches as well as I/O processing by adopting some combination of concurrent processing, i.e. begin rendering of the top ranked documents while continuing to process the search query. Another possible approach is to have multiple threads or processes rendering the document images in parallel.

BIBLIOGRAPHY

[1] The Unicode Consortium. Ten years of unicode 1988-1998. URL: <http://www.unicode.org/history/tenyears.html> (viewed Aug. 24, 2003).

[2] The Unicode Consortium. The Unicode Standard, Version 3.0. Addison-Wesley, 2000.

[3] Apple Computer Corporation. Truetype reference manual. URL: <http://developer.apple.com/fonts/TTRefMan/index.html> (viewed Sept. 18, 2003).

[4] W. B. Frakes. Stemming algorithms. In William Frakes and Ricardo Baeza-Yates, editors, Information Retrieval: Data Structures & Algorithms, pages 131–160. Prentice Hall, 1992.

[5] Markus Kuhn. Utf-8 and unicode faq. URL: <http://www.cl.cam.ac.uk/~mgk25/unicode.html#utf-8> (viewed Aug. 24, 2003).

[6] Hughes Technologies Pty Ltd. Libhttp. URL: <http://www.hughes.com.au/products/libhttpd> (viewed Oct. 10, 2003).

[7] Free On-Line Dictionary of Computing (FOLDOC). American standard code for information interchange. URL: <http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?ascii> (viewed Aug. 20, 2003).

[8] Free On-Line Dictionary of Computing (FOLDOC). Extended binary coded decimal interchange code. URL: <http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?ebcdic> (viewed Aug. 20, 2003).

[9] Free On-Line Dictionary of Computing (FOLDOC). Us-ascii. URL: <http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?usascii> (viewed Aug. 20, 2003).

[10] Encyclopaedia of the Orient. Persian language. URL: <http://lexicorient.com/e.o/persian_1.htm> (viewed Aug. 18, 2003).

[11] Roozbeh Pournader. Farsi keyboard (fwd). URL: <http://lists.sharif.edu/pipermail/persiancomputing/2000-August/000129.%html> (viewed Sept. 22, 2003).

[12] The FreeType Project. Freetype 1. URL: <http://freetype.sourceforge.net/freetype1/index.html> (viewed Sept. 18, 2003).

[13] UCLA Language Materials Project. Persian language profile. URL: <http://www.imp.ucla.edu/profiles/profp01.htm> (viewed Aug. 18, 2003).

[14] Gerard Salton. Automatic text processing: the transformation, analysis, and retrieval of information by computer. Addison-Wesley, 1989.

[15] Guy Eric Schalnat, Andreas Dilger, and Glenn Randers-Pehrson. Png (portable network graphics. URL: <http://www.libpng.org/pub/png> (viewed Sept. 18, 2003).

[16] Nadim Shaikli. Perl script: shape_arabic.pl. URL: <http://www.arabeyes.org/viewcvs/tools/shape_bidi/shape_arabic.pl> (viewed Sept. 7, 2003).

[17] K. Taghva, R. Young, J. Coombs, R. Beckley, M. Sadeh, and R. Pereda. Farsi searching and display technologies. In Proceedings of the 2003 Symp. on Document Image Understanding Technology, pages 41–46, 2003.

[18] Kazem Taghva, Russell Beckley, and Mohammad Sadeh. A stemming algorithm for the farsi language. Technical report, Information Science Research Institute, University of Nevada, Las Vegas, August 2003.

[19] Kazem Taghva, Russell Beckley, and Mohammad Sadeh. A list of farsi stopwords. Technical report, Information Science Research Institute, University of Nevada, Las Vegas, July 2003.

[20] Kazem Taghva, Ray Pareda, and Jeffrey Coombs. Language model-based retrieval for farsi documents. Technical report, Information Science Research Institute, University of Nevada, Las Vegas, To Appear.

[21] Center for Advanced Research on Language Acquisition University of Minnesota. Find where a lctl is taught in north american colleges and universities. URL: <http://carla.acad.umn.edu/lctl/db/search-wlw.html> (viewed Aug. 18, 2003).

[22] Unknown. Utf-8 and unicode standards. URL: <http://www.utf-8.com> (viewed Aug. 24, 2003).

[23] World Wide web Consortium. Libwww - the w3c protocol library. URL: <http://www.w3c.org/Library> (viewed Sept. 15, 2003).

VITA

Graduate College
University of Nevada, Las Vegas

Ronald L. Young

Local Address:
8507 Shelly Rd.
Las Vegas, NV 89123

Degrees:
Bachelor of Science, Computer Science, 1981
University of Nevada, Las Vegas

Publications:
Kazem Taghva, Ron Young, Jeff Coombs, Russell Beckley, Mohammad Sadeh, and Ray Pereda, Farsi Searching and Display Technologies, Proceedings of the 2003 Symp. on Document Image Understanding Technology, Washington D.C., 2003.

Thesis Title:
The Design and Implementation of an Input/Output Subsystem for a Farsi Language Search Engine

Thesis Examination Committee:
Chairperson, Kazem Taghva, Ph. D.
Committee Member, Dr. Thomas Nartker, Ph. D.
Committee Member, Dr. Wolfgang Bein, Ph. D.
Graduate Faculty Representative, Dr. Emma Regentova, Ph. D.