8-1-1992

AN X11-BASED JAPANESE LANGUAGE PROCESSING AND LEARNING ENVIRONMENT

Boon-Lock Yeo  
*Purdue University, School of Electrical Engineering*

Minerva Yeung  
*Purdue University, School of Electrical Engineering*

Anthony A. Maciejewski  
*Purdue University, School of Electrical Engineering*

Follow this and additional works at: [http://docs.lib.purdue.edu/ecetr](http://docs.lib.purdue.edu/ecetr)

[http://docs.lib.purdue.edu/ecetr/310](http://docs.lib.purdue.edu/ecetr/310)

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
AN X11-BASED JAPANESE LANGUAGE PROCESSING AND LEARNING ENVIRONMENT

Boon-Lock Yeo
Minerva Yeung
Anthony A. Maciejewski

TR-EE 92-33
August 1992

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285
AN X11-BASED JAPANESE LANGUAGE
PROCESSING AND LEARNING ENVIRONMENT

Boon-Lock Yeo
Minerva Yeung
Anthony A. Maciejewski

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

TR-EE 92-33
August 1992

This material is based upon work supported by the National Science Foundation under Grant No. INT-8818039 and in part by a Purdue Global Initiative Faculty Grant.
Contents

1 Introduction 1

2 Obtaining and Installing New Software 2

3 Tools for Processing Japanese 3
   3.1 Kterm ............................................ 3
      3.1.1 Usage ........................................ 3
      3.1.2 Installation ................................... 3
   3.2 Jserver ........................................... 5
      3.2.1 Usage ........................................ 5
      3.2.2 Installation ................................... 5
   3.3 Kinput ............................................. 7
      3.3.1 Usage ........................................ 7
      3.3.2 Installation ................................... 8
   3.4 Nernacs ............................................... 9
      3.4.1 Installation ................................... 9
      3.4.2 Usage ........................................ 9
   3.5 Stevie ............................................... 11
      3.5.1 Usage ........................................ 11
      3.5.2 Installation ................................... 11
   3.6 Printing plain Japanese text: kanjips & jenscript ....... 12
      3.6.1 Usage ........................................ 12
      3.6.2 Installation ................................... 12
   3.7 \LaTeX\ ................................................ 13
      3.7.1 Usage ........................................ 13
      3.7.2 Installation ................................... 15
      3.7.3 Another plain Japanese text printing program: k2ps . 16
   3.8 \LaTeX\ ............................................. 17
      3.8.1 Usage ........................................ 17
      3.8.2 Installation ................................... 17
   3.9 Tree drawing program: tree ................................ 19
      3.9.1 Usage ........................................ 19
      3.9.2 Installation ................................... 21
   3.10 Code conversion: jis .................................. 22
      3.10.1 Usage ........................................ 22
      3.10.2 Installation ................................... 23
4 EtoNPat: ETL to Nihongo Tutor Parse Tree Conversion

4.1 Function MakeTree ........................................ 26
4.2 Function CheckTree .......................................... 26
4.3 Function AssignNodeNum ...................................... 26
4.4 Function ClimbTree .......................................... 27
4.5 Function PutSentence ........................................ 27
4.6 Termination of Conversion ................................. 27

5 Dictionary formats to support the Nihongo Tutor

5.1 Format Details .................................................. 29
5.1.1 Raw format .................................................. 29
5.1.2 Compiled Dictionary Format ............................. 30
5.1.3 Internal Representation ................................ 31
5.2 Merging Different Dictionaries ......................... 33
5.3 Utility programs and routines ........................... 35
5.3.1 Sorting Japanese text: jsort ......................... 35
5.3.2 Removing single kanji entries: stripsingle ........ 35
5.3.3 Conversion from other dictionary formats ........ 36
5.3.4 Useful functions for Japanese text manipulation 36
1 Introduction

The Japanese text processing environment discussed here is built on top of the X-window system. Various software tools from external sources are used to provide the user with a complete environment consisting of Japanese text display, input of Japanese through the keyboard, output of text to Postscript printers and document-typesetting facilities.

This document seeks to explain briefly the use of such tools. Instead of leaving the user to read through thick and detailed documents that come with the software, the essence of using these tools is demonstrated with step by step examples. The user is left on his own to explore the details of more advanced features.

In section 2, the procedure for installing new software is described. The general techniques described are supplemented with the variations required to install specific software tools in the following sections.

In section 3, the various Japanese text processing tools that are currently available are listed. This section includes information on the version number, where the software was obtained, where it is installed, technical details of the installation and how to use the tools through examples.

In section 4, the program EtoNPat, which serves to automate the process of generating parse trees in the format required by the Nihongo Tutoring System, is described in sufficient detail to allow extensions and modifications.

Finally, section 5 describes the software developed to organize and classify Japanese dictionaries so as to provide better translation and pronunciation information to EtoNPat. For an introduction to Japanese text processing and the various formats for representing Japanese text, please refer to japan1.inf, japan2.inf and japan3.inf located at ~/documentation.

All software referred to in this document is currently installed in the account named japanese and has been tested on a sparc IPC. All system directories are on ECN's NFS-network.
# 2 Obtaining and Installing New Software

Before even discussing installation, one must first obtain the software. Most packages of interest are available via anonymous ftp sites. The procedure for access is no different from normal ftp except that you type in 'anonymous' instead of your login when prompted. If it is in .Z format, enable binary transfer with the bin command. If it is a compressed tar file (one that ends with .tar.Z extensions), do the following to extract everything:

```
% zcat <file> | tar xfv -
```

You may want to consult the man page of zcat and tar for more detailed instructions.

Software packages are usually first created on a particular platform and then moved and ported to suit other machines and architectures. However, since most of the software discussed here is based on X11, the difficulty of installation is reduced. In general, however, the more job the software author puts into creating a universal Makefile, the easier the installation job will be.

Fortunately, most sophisticated software packages based on X11 come with an Imakefile. This file consists of C-type macro definitions that get expanded when run through make. The major problem is that every site has its own path configurations for the various X11 include files, libraries, etc. Imake can thus be made to put together all this path information and the content of Imakefile to produce a Makefile. To make your life even easier, just type 'xmkzf'. It is a shellscript that calls make with the right path to the X11 configuration directory and automatically generates the Makefile you desire. For the curious user, this configuration directory is found in /usr/lib/X11/config.

After the Makefile is generated, it is customary to type 'Make Makefiles' to generate the Makefile in other subdirectories. Otherwise, you can type 'make' to generate the executable and 'make install' to move the software to the directory you desire.

The above steps do not apply to every software package. It is advisable to look at the README file that comes with the package. In most cases, it is necessary to modify some definitions in the Imakefile to suit your own needs. Sometimes, you will find a shell-script that 'interactively asks you for the right information. This shell-script will then generate the Makefile.

It is typical when installing new software that things will not be exactly right the first time and the code will not compile. Here are a few of the things to check:

1. You do not have the definitions correctly specified. Read the README file carefully.
2. You do not have the correct version of X11. Release 5 was out on October 1991. You must make sure that the correct release is used since the libraries are not the same.
3. There are bugs in the program. As anyone who has done significant programming knows, it is hard to write a flawless program, especially if it is to be used on different architectures and platforms. If you are convinced that this is the case, you should email the authors or post news to the relevant news groups to ask for help. There are always new problems with new software or newer releases. You can look out for patches to apply (do a man on patch for more details).
3 Tools for Processing Japanese

3.1 Kterm

<table>
<thead>
<tr>
<th>Tool Name: kterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version: 4.1.2</td>
</tr>
<tr>
<td>Ftp-site: ftp.uu.net: packages/X/contrib</td>
</tr>
<tr>
<td>Executable: ~/bin/sparc</td>
</tr>
<tr>
<td>Source: ~/kterms/src</td>
</tr>
<tr>
<td>X-window: Version 11 Release 5</td>
</tr>
<tr>
<td>Manpage: ~/usr/local/man/man1</td>
</tr>
</tbody>
</table>

Kterm is a kanji terminal emulator that allows the display of Japanese text in several formats. It is modified from xterm. In the default, it starts up in the EUC mode. To switch character mode, press the <control> key and the middle key of the mouse. The options are at the bottom of the menu.

3.1.1 Usage

- You can start up kterm with the following:
  
  ```
  % kterm -lang japanese -km "euc" -ls -fn "a14" -fr "r14" -fk "k14"
  
  It is a good idea to set an alias for the above command.
  ```

- This sets kterm to start in the right fonts to support Japanese language in EUC mode.

- With the alias set, just type `kterm &` to start it.

3.1.2 Installation

- Do
  
  ```
  % grep "#include <X11/copyright.h>" *
  
  and remove all files that do the above include. This is necessary as this file is no longer present in the X11R5 version.
  ```

- Type
  
  ```
  % xmkf
  
  % make
  ```

- If your system does not have the required fonts, ftp them and install in a directory, say `/xfonts`. Then in your X startup file (usually .xinitrc), include `xset fp+ /xfonts` to inform X-window where to find more fonts.

- If you don't want to restart X-windows, in addition to the previous commands, you can type...
\% \texttt{xset fp+ -/xfonts} \\
\% \texttt{xset fp rehash} \\
\% \texttt{xlsfonts}

The last command lists all of the available fonts. \textbf{Make sure the new ones are in the list.}

The kterm that comes with X11R5 does not seem to support \textit{pasting} of Kanji and thus does not allow \texttt{kinput} to work well. The currently installed version, based on X11R4, however, compiles and works well with X11R5.
3.2 Jserver

**Tool Name:** jserver

**Version:** 4.0.3

**Ftp-site:** crl.nmsu.edu : pub/japanese

**Executable:** -/usr/local/bin/X11

**Source:** -/wnn.4.0.3/Wnn

**Libraries:** -/usr/local/lib/wnn & subdirectories

Jserver is a server that does Kana-Kanji conversion. It comes with wnn, which comes with a front-end Kanji input program, wnn and a few dictionaries for Kana-Kanji conversion. As wnn is much more primitive than kinput (see section 3.3), it is not installed. Jserver also supports Kana-Kanji conversion in emacs, the Japanese version of emacs (see section 3.4). Note that jserver alone does not do anything useful.

3.2.1 Usage

- on the machine where you need the jserver, simply type 'jserver'.
- before you log off, type 'wnnkill' to kill the jserver process. If you don't do this, subsequent attempts to start up jserver will result in the error message
  
  `% jserver: can't bind inet-socket`

- typing 'wnnstat' will show you a list of users who are using jserver. If there are any other users, you cannot do a wnnkill.

3.2.2 Installation

- all paths described are relative to the source directory.
- follow the instructions in the `READ.ME`.
- to make installation in a local directory easier, append your path name to all wnn related directories and change `WNOWNER` to your login. Look at `wnnmake.def` for a detailed example.
- apply `wnntouch.patch` if necessary.
  
  `type`
  
  `% xmkf
  
  `% make Makefiles`

- change `USRLIBDIR` in the Makefile in directories `/jlib` and `/jlib.V3` from the system ones to the local ones.
- type 'make'
type `make install'. You might get some complaints about the use of `chovn'. Just ignore them.

finally, edit your `jservverrc` file (wherever it is) to get the right path to the dictionaries.

look at `note.yeo` for additional modifications to the instructions in `READ.ME`.

There is a newer release of `wnn`, version 4.1.0 that comes with the contrib directory of `X11R5`. Installation of this release requires `X11R5` to be built. As there appear to be no additional features in this version, it is not urgent to install the newer version since the older one is working fine.
### 3.3 Kinput

<table>
<thead>
<tr>
<th>Tool Name: kinput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version: 1.0</td>
</tr>
<tr>
<td>Ftp-site: crl.nmsu.edu : pub/japanese</td>
</tr>
<tr>
<td>Executable: -/usr/local/bin/X11</td>
</tr>
<tr>
<td>Source: -/kinput/kinput</td>
</tr>
<tr>
<td>Libraries: -/kinput/kinput/cconv/ccdef</td>
</tr>
<tr>
<td>X-window: Version 11 Release 5</td>
</tr>
<tr>
<td>Manpage: -/usr/local/man/man1</td>
</tr>
</tbody>
</table>

**Kinput** is a front-end for X11 applications that allows easy input of Japanese. A client that wants *kana-kanji* conversion service for Japanese text sends a request to **kinput**. When **kinput** receives the request, it pops up a **conversion window**, does *kana-kanji* conversion, and sends the converted text back to the client. All the communications between **kinput** and clients is done via the X11 inter-client communication mechanisms.

**Kinput** uses **Win** version 4 (see section 3.2) as a conversion engine. Specifically, it requires **jserver** as the conversion server.

It can be used with **vi**. A clone of **vi**, called **stevie** (see section 3.5) that supports 8-bit character code (instead of the normal 7 for ASCII) is also available.

#### 3.3.1 Usage

- for reasons that are unknown, **jserver** must be run from a different machine from where **kinput** is run. All subsequent examples will make reference to the machines named **gnu** and **piano** and the account called **japanese**.

- in addition, if **jserver** is started in an account under a particular machine, then **kinput** will refuse to work even if it is run on another machine but under the same account. Thus the recommended procedure is to start up **kinput** in one account, while having **jserver** run under another. This makes sense, since **jserver** is a server, and a server usually runs under **root**.

- add the following lines to your X resource file (probably **.X11defaults**), which gets xrdbed when you start up X-windows.

```bash
! kinput
! kterm*vt100*translations: Povrrride Shift<Key>space:begin-conversion(JAPANESE_CONVERSION)
```

Make sure to restart X-wintows. This will allow pressing `<shift>-space` to pop up the conversion window.

- suppose, you have logged in to your account at **piano** and to **japanese** at **gnu** and started **jserver** at **japanese**, at your account's **kterm**, type
% kinput -jsserver gnu &

- wait for a short while, then press <shift>-space to get the conversion window. You may see a few

% conversion server not found

errors before the conversion window finally appears for the first time.

- once you see the conversion window at piano, you should be able to get the window at gnu under japanese as well.

- do a ‘man kterm’ for a list of conversion keys.

3.3.2 Installation

- edit Makefile to change WNNSRC:wnn’s source directory pathname, and other relevant paths.

- do the following:

  % xmkmf
  % make Makefiles
  % make

- if you get some error message regarding invalid types, i.e., 'illegal type combination', which are definition problems related to wchar_t, append the flag -DWCHAT_T to CFLAG in the Makefile of the directory where this happens.

There is a newer version of kinput, called kinput2, that supports better input styles, which are root-window, off-the-spot and over-the-spot. It relies on X11R5 and it is worth the effort to install it, if possible. Version 1 compiles and works fine with X11R5 even though it was built originally on R4.
3.4 Nemacs

<table>
<thead>
<tr>
<th>Tool Name:</th>
<th>nemacs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:</td>
<td>3.3.2</td>
</tr>
<tr>
<td>Ftp-site:</td>
<td>crl.nmsu.edu: pub/japanese</td>
</tr>
<tr>
<td>Executable:</td>
<td>/bin/sparc</td>
</tr>
<tr>
<td>Source:</td>
<td>/nemacs/src</td>
</tr>
<tr>
<td>Libraries:</td>
<td>/nemacs/src/lisp</td>
</tr>
</tbody>
</table>

Nemacs is the patched version of emacs that supports 8-bit chars and especially Kanji characters. It requires kterm (see section 3.1) to run. Together with jserver (see 3.2), one can do Kana-Kanji conversion within nemacs.

3.4.1 Installation

- go into src directory and edit ymakefile, config.h and nconfig.h to set the right path. In config.h, you may want to uncomment certain lines if you want to use X windows. Look at the file for more instructions.
- just type
  
  `% make`
- the executable will be emacs-18.55 with some other numbers appended. You may want to rename this to nemacs and move it elsewhere.
- you may want to read the README.Nemacs and INSTALL.Nemacs in the source directory for more general information.
- to use jserver with nemacs, you need two additional things:
  1. .eggrc in your $HOME directory.
  2. a list of .frq files for the wnn dictionaries. These files are created by kinput, in /Wnn/public.

You can copy a sample .eggrc that comes with wnn, in the /jd directory. In .eggrc, the directory that contains the wnn dictionaries (the .dic and .fzk files) is the sys-dic while the one that contains the .frq is the usr-dic.

You do not need the line (setq jserver-host-name "unix").

? 2 Usage

- just type
  
  `% nemacs`

to bring up a separate window. If you want nemacs to appear in the current window, add a -nw command.
• C-\ toggles the mode to enter romaji. The hiragana can be interpreted by typing a space.

  .eggrc is loaded, if it never has been. Initial loading may take a while.

• Once you have .eggrc loaded in, doing an apropos on kanji will list all the available command related to Japanese text input.

• C-"N and C-"T will bring out nemacs news and beginner's tutorial respectively (in Japanese).
3.5 Stevie

Stevie is a vi-clone. It has been modified to support 8-bit codes so that Japanese characters can be displayed. Currently, it has been compiled to support only EUC mode (it can only support one mode). It can interact with kinput (see section 3.3) to allow Kana-Kanji conversion.

3.5.1 Usage

- all commands are the same as vi.
- to do Kana-Kanji conversion, use the same procedure as described in section 3.3.

3.5.2 Installation

- edit onv.h to uncomment the desired kanji mode.
- just type
  
  \% make

  It is never easier!
3.6 Printing plain Japanese text: kanjips & jenscript

| Tool Name: | kanjips |
| Version:   | 1       |
| Ftp-site:  | mindseye.berkeley.edu: /pub/kanji/misc |
| Executables: | -/bin/sparc |
| Source:    | -/kanjips |

Kanjips is a program for printing plain Japanese text. This is useful if you do not need to do any kind of formatting. The font size of the output, however, is small.

3.6.1 Usage

- for a file denoted by `<file>` that you want to print to a postscript printer, do the following:
  ```
  % jenscript <file> | kanjips | lpr -P<printer>
  ```
- if you are using EUC mode, add the -E flag to jenscript.
  since the fonts are small, you can print 2 pages worth of text on one page with the -2 options to jenscript. The second page appears as a second column.
- you might want to read kanjips/README and jenscript/README for more information.

Jenscript requires the perl program as a language processing tool to convert Japanese characters into the right code. However, currently, perl is only available on the sun3, not the sun4.

3.6.2 Installation

- run Configure in the source directory. Just answer the questions correctly. A Makefile will be generated for you.
- after getting the executable, move it to your favorite directory and remember to set the path.
3.7 \texttt{J\TeX}/\texttt{JLa\TeX}

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Tool Name:} & \texttt{J\TeX}/\texttt{JLa\TeX} \\
\textbf{Version:} & NTT version 1.2 \\
\textbf{Ftp-site:} & akiu.gw.tohoku.ac.jp:/pub/tex/NTT-jtex \\
\textbf{Executable:} & -/bin/sparc \\
\textbf{Source:} & -/jtex \\
\hline
\end{tabular}
\end{center}

\texttt{J\TeX}/\texttt{JLa\TeX} is the modified version of \texttt{\TeX}/\texttt{La\TeX} to allow the use of Japanese character in document typesetting. The original \texttt{\TeX} code has been modified to accommodate such Japanese fonts. Use of \texttt{J\TeX}/\texttt{JLa\TeX} is no different from \texttt{\TeX}/\texttt{La\TeX}.

3.7.1 Usage

- either \texttt{jtex} or \texttt{jlatex} the document.
- some \texttt{dvi}-previewers do not support METAFONT font numbers beyond 128. To overcome this limitation, do the following:
  
  \begin{verbatim}
  \% dvips <file>.dvi
  \% gs <file>.ps
  \end{verbatim}

  to preview. \texttt{Gs} is a \texttt{postscript} previewer. Otherwise, if font numbers beyond 128 are not a problem, just type

  \begin{verbatim}
  \% xdvi <file>.dvi
  \end{verbatim}

  to print the \texttt{postscript} file.

  \begin{verbatim}
  \% dvips <file>.dvi
  \% lpr -P<printer> <file>.ps
  \end{verbatim}

  for \texttt{JLa\TeX} user, instead of using \texttt{article.sty}, use \texttt{jarticle.sty}. Similarly, specifying 11pt or 12 pt font size in \texttt{\documentstyle} will cause \texttt{JLa\TeX} to load \texttt{j-art11.sty} and \texttt{j-art12.sty} respectively.

  As an example, consider the short \texttt{JLa\TeX} document:

  \begin{verbatim}
  \documentstyle[11pt]{jarticle}
  \begin{document}
  赤い J\TeX はNTTの斎藤康己さんによって DEC-2065 TOPS-20 \\
  上で作られました。
  \end{document}
  \end{verbatim}

  It becomes:

  最初の J\TeX はNTTの斎藤康己さんによって DEC-2065 TOPS-20 上で作られました。

  13
You may notice that the fonts do not look too great. These are standard JIS fonts generated from the bitmap file \texttt{jis24}. If you want better fonts, they must be purchased. Look at the file \texttt{README} in the \texttt{jtex} directory for such information.
3.7.2 Installation

- you don't have to modify any names in the makefile. You can set environment variables to point to the right directories later, if necessary.
- just type 'make all' in the source directory.
- if you get the following errors

  
  /: write failed, file system is full
  compiler(iropt) error: write-irfile: No space left on device

  in the texware directory, try turning off the -O flag to cc.

- the main executable is virjtex. It has to be loaded with the right format file, for \TeX//\LaTeX. To make your job easier, use the following shell script and named it jtex and jlatex:

  ```
  #!/bin/sh
  case "$0" in
  */jtex|jlatex) me=jtex; fmt=jtex;;
  */jlatex) me=jlatex; fmt=jlatex;;
  */jslitex) me=jslitex; fmt=jslitex;;
  *) echo "don't know how to be $0" 1>&2; exit 1;;
  esac

  # verify 1 or 2 arguments
  case $# in
  1|2) ;;
  *) echo "usage: $me foo[.tex [my[.fmt]]]" 1>&2; exit 1;;
  esac

  exec virjtex "&"${2-$fmt} ${1+"$1"}
```

Then, running jtex or jlatex will call virjtex with the right format file. You have to set the environment variables TEXFDRMATS to point to the directory where the format files are found. They are named jtex.fmt and jlatex.fmt.

- there are a bunch of \TeX related environmental variables which you might want to set for future use. These are especially handy if you teclite to add others files (e.g. fonts, formats, input files, etc.) for \TeX's use.

  **TEXINPUTS:** where you put files to be included into \TeX.
  **TEXFONTS:** where you put additional font files.
  **TEXFORMATS:** where the pre-defined \TeX format files, which are loaded upon running \TeX, are stored.
  **TEXPOOL:** where the file tex.pool resides.
3.7.3 Another plain Japanese text printing program: k2ps

K2ps comes with the \TeX\ package and it uses the \TeX\ fonts to print plain Japanese text. The fonts are much bigger than the ones offered by jenscript (see section 3.6). It is useful for printing out plain Japanese documentation.

It is located at -/\tex/Printer/Postscript/k2ps. To compile, modify the Makefile to set the right directories to the \TeX\ font directory and k2ps directory. The README in that directory gives you the details of using it. Alternatively, you can type:

```
%! k2ps -help
```
to get a list of possible options.

Briefly, to print a file called example, type

```
%! k2ps -Ke -kj example | lpr -P<printer>
```

The -Ke options means EUC codes. To get JIS and shift-JIS, use - kj and -ks respectively. The -kj requests the use of standard JIS \TeX\ fonts. You have to use this if you do not have better \TeX\ fonts.
3.8 \texttt{JemTeX}

\begin{tabular}{|p{\textwidth}|}
\hline
\textbf{Tool Name: JemTeX} \\
Version: 2 \\
Ftp-site: utsun.s.u-tokyo.ac.jp:/TeX \\
Executable: -/bin/sparc \\
Source: -/jemtex \\
\hline
\end{tabular}

\texttt{JemTeX} is a simple system to typeset Japanese using any standard version of \TeX/\LaTeX. It was created by François Jalbert to allow \TeX/users in DOS to typeset Japanese documents. He created better fonts than the bitmap ones from \TeX through a program which automatically generates METRTEXT files from the standard \texttt{jis24} bitmap file.

It works differently from \TeX/\LaTeX in the sense that it first converts any Japanese fonts into predefined font numbers using the program \texttt{jem2tex}. The processed file can then be run through \TeX/\LaTeX.

For a complete documentation of \texttt{JemTeX}, refer to \texttt{jguide.tex} in -/jemtex. It is a \LaTeX document. Or, you can also print out \texttt{jguide.ps} in the same directory.

3.8.1 Usage

- create a source document with extension \texttt{.jem} instead of the normal \texttt{.tex}. At the start of the document, put in the following line:

\begin{verbatim}
\% JEM2TEX /Space /NoPercent /LaTeX /EUC /Extended /2.0
\end{verbatim}

This specifies the right parameters to \texttt{jem2tex}. Alternatively, you can specify the parameters on the command lines. Refer to page 36 of \texttt{jguide} for more details. If you are going to use \TeX instead replace \LaTeX by \TeX in the line to be added.

- suppose the file created is called \texttt{example.jem}, type

\begin{verbatim}
\% jem2tex example
\end{verbatim}

to process the document. Note that the extension \texttt{.jem} is not needed.

- a file \texttt{example.tex} is then produced. Run it through \TeX or \LaTeX.

3.8.2 Installation

to unpack the compressed file, which is in \texttt{.zip}, use \texttt{unzip}. It is available at utsun.s.u-tokyo.ac.jp:/misc/unzip. Use it with the -a option which converts all CR LF to LF for unix.

- edit \texttt{jem2tex.c} and change all \texttt{fprintf(OutFile,"\%")} to \texttt{fprintf(OutFile,"%%%")}.

- compile \texttt{jem2tex.c} with \texttt{gcc}. \texttt{cc} will not work because the author used ansi-C standards.
• you will still need to install the fonts. Refer to pages 9 - 17 in jguide for detailed instructions. You will need to use METRFONT. If it is not available on your system, I suggest that you obtain a copy of emTeX that runs on DOS, and follow the instructions in jguide. It will work fine because the instructions to generate fonts are meant for DOS users.

• there are six different font sizes you could generate. They are of sizes inagstep(0.0) to magstep(6.0). You will need the .pk and .tfm files. If you are generating these files in DOS, make sure to rename the .pk to .<num>.pk to conform to the standards in unix, where <num> = 300 x 1.2magstep, and 300 is the assumed dpi of the output printer and magstep is 0–6, as described above. For example, if you are creating fonts of magstep 2.0, then a .pk file would have an extension of .432pk since 300 x 1.2^2.0 = 432.

• put the fonts in a directory, and make sure you have set the environment variable TEXFONTS to include this new directory.

To typeset the sample Japanese document in section 3.7 in magstep of 2.0, the source looks like:

```
\documentstyle[11pt](article)
\begin{document}

最初の JTeX はNTTの斎藤康己さんによって DEC-2065 TOPS-20
上で作られました。
\end{document}
```

Observe the effect of the better font quality: 最初の JTeX はNTTの斎藤康己さんによって DEC-2065 TOPS-20 上で作られました。
3.9 Tree drawing program: tree

<table>
<thead>
<tr>
<th>Tool Name:</th>
<th>tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:</td>
<td>1.1</td>
</tr>
<tr>
<td>Ftp-site:</td>
<td>ohccux.uhcc.hawaii.edu: linguist (no longer available)</td>
</tr>
<tr>
<td>Executable:</td>
<td>-/jemtex/tree</td>
</tr>
<tr>
<td>Source:</td>
<td>-/jemtex/tree</td>
</tr>
</tbody>
</table>

This program was developed at the University of Hawaii for formatting trees for screen display or printing with \( \TeX \). Together with Jem\( \TeX \) (see section 3.8), one can create parse trees that contain Japanese characters. Its use with \( \TeX \), however, is not possible as \texttt{tree} does not recognize non-ascii characters, which are used to represent Japanese characters. Jem\( \TeX \), however, converts Japanese characters into predefined font numbers; thus the input to \texttt{tree} is merely ascii after being preprocessed by Jem\( \TeX \).

3.9.1 Usage

- this sequence only shows how to produce parse trees from the ETL parser's output. The desired tree is in the file \texttt{example.jap}

  a do the following:
  - \texttt{convtree example.jap > example.jem}
  - \texttt{jem2tex example}
  - \texttt{mv example.tex example.tt}
  - \texttt{tree -t example.tt > example.tex}
  - \texttt{latex example.tex}
  - \texttt{rm example.tt}

- \texttt{convtree} is a program in the same directory to convert ETL's output to the format required by \texttt{tree}.
Suppose a parse tree is produced from the ETL-parser as follows:

\tree
(TOP
  \- 用言 句
     \- 用言 句
        \- 用言 句
               \- 語幹  --- 購
               \- 語尾  --- わ
               \- 助動詞  --- せ
               \- 助動詞  --- られ
               \- 助動詞  --- ない
     \- END ---)

To represent it in \texttt{tree} format it becomes:

\texttt{\tree}

\texttt{(TOP}

\texttt{  \(\textbackslash\texttt{\textbackslash} \text{ 用言 句}

\texttt{    \(\textbackslash\texttt{\textbackslash} \text{ 用言 句}

\texttt{       \(\textbackslash\texttt{\textbackslash} \text{ 用言 句}

\texttt{          \(\textbackslash\texttt{\textbackslash} \text{ 用言}

\texttt{             \(\textbackslash\texttt{\textbackslash} \text{ 語幹  --- 購}

\texttt{                 \(\textbackslash\texttt{\textbackslash} \text{ 語尾  --- わ}

\texttt{                    \(\textbackslash\texttt{\textbackslash} \text{ 助動詞  --- せ}}

\texttt{                     \(\textbackslash\texttt{\textbackslash} \text{ 助動詞  --- られ}}

\texttt{                        \(\textbackslash\texttt{\textbackslash} \text{ 助動詞  --- ない}}

\texttt{                        \(\textbackslash\texttt{\textbackslash} \text{ END ---}}

\texttt{\textbackslash\texttt{\textbackslash})

The \texttt{\textbackslash} above is used to force a blank space for a better looking display. For detailed instructions, refer to use \texttt{.tex} in the same directory. Also, you can find \texttt{texexample} that contains a lot of useful examples.
After the proper sequence of processing presented above, the result is:

```
TOP
  ↓
文
  ↓
用言句。
  ↓
用言句 助動詞
  ↓
用言句 助動詞
  ↓
用言句 助動詞
  ↓
用言句 助動詞
  ↓
用言 せ
  ↓
言幹 言尾
  ↓
貰 わ
```

3.9.2 Installation

- just type 'make' to get the program tree.
3.10 Code conversion: jis

<table>
<thead>
<tr>
<th>Tool Name:</th>
<th>jis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ftp-site:</td>
<td>ucdavis.edu: /pub/JIS/C</td>
</tr>
<tr>
<td>Executable:</td>
<td>/bin/sparc</td>
</tr>
<tr>
<td>Source:</td>
<td>/bin/src</td>
</tr>
</tbody>
</table>

This program is written by Ken R. Lunde, the person who wrote japan.inf, to allow the change of Japanese codes within a textfile. It can handle Shift-JIS and EUC, as well as all of the three 7-bit Japanese codes (New-JIS, Old-JIS, and NEC-JIS).

3.10.1 Usage

- the following is lifted from the comments in jis.c

- the command-line is
  
  jis [-kanji-code] [-half-to-full] [infile] [outfile]

  The [-kanji-code] and [-half-to-full] flags can be in any order, but must come before the file names (if used). Note that [infile] and [outfile] can be replaced by redirecting stdin/stdout on UNIX systems.

- the [-kanji-code] flag, which determines the outfile's kanji code, can be 1 of 5 possible values:
  -j New-JIS (.new)
  -o Old-JIS (.old)
  -n NEC-JIS (.nec)
  -e EUC (.euc)
  -s Shift-JIS (.sjs)

- the [-half-to-full] flag, which determines whether to convert half-size katakana to their full-size counterparts, can be indicated by -f. The default is to NOT convert half-size katakana, but if the output file is in New-JIS, Old-JIS, or NEC-JIS, ALL half-size katakana will be converted to their full-size counterparts (necessary, since those three 7-bit codes do not support half-size katakana). Again, upper- and lower-case are acceptable.

- the [infile] field is optional as one can redirect stdin and stdout.

- the [outfile] field is also optional. If no [outfile] field is specified, the program will semi-intelligently change the file's name. The program simply scans the [outfile] field, finds the last period in it, terminates the string at that point, and tacks on one of 5 possible extensions (those listed above). Here are some example command lines, and the resulting outfile names:

  jis -e sig.jpn
  jis sig.jpn
  jis -j sig.jpn.txt
  jis -o sig

  sig.euc
  sig.sjs (defaulted to Shift-JIS)
  sig.jpn.new
  sig.old
This is very useful for MS-DOS users since a filename such as sig.jpn.new will not result in converting a file called sig.jpn.
Also note that if the outfile and infile have the same name, the program will not work, and data will be lost. For example, note how the program will change the outfile name so that it does not overwrite the infile:

```
jis -f sig.sjs          sig-.sjs
jis -f sig.sjs sig.sjs  sig-.sjs
jis sig-.sjs            sig-.sjs
```

If only the [infile] is given, a hyphen is inserted after the last period, and the extension is then reattached. If the outfile is specified by the user, then it will search for the last period (if any), attach a hyphen, and finally attach the proper extension. This sort of protection is not available from this program if stdin/stdout are used.

3.10.2 Installation

• just type:
  
  `% gcc -o jis jis.c`

23
4 EtoNPat: ETL to Nihongo Tutor Parse Tree Conversion

<table>
<thead>
<tr>
<th>Tool Name: EtoNPat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version: 1</td>
</tr>
<tr>
<td>Executable: /home/zoo2/japanese/yeo/ETLtoNTS/dict</td>
</tr>
<tr>
<td>Source: /home/zoo2/japanese/yeo/ETLtoNTS/dict</td>
</tr>
</tbody>
</table>

The Nihongo Tutor is a Japanese Language Intelligent Tutoring System that is capable of selecting articles that are suitable for different users based on their individual needs and capabilities. An impressive feature of this system is the ability to show the user the grammatical structure of a sentence, once this structure has been stored. Within this grammatical structure, the user can also request translation and pronunciation information about any of the constituents. This feature, which is especially useful for beginners, has traditionally required the laborious task of manually inputting the grammatical structure of the sentences as well as their translations and pronunciations.

The program EtoNPat, written by Minerva Yeung and later modified by Boon-Lock Yeo, seeks to overcome the shortcomings of the Nihongo Tutor and to remove as much of the human interaction as possible from the preparation of the articles. It takes as inputs parse trees from the ETL_Parser, and then converts them into the form required by the Nihongo Tutor. At this point, this program is not yet capable of inserting the correct translation and pronunciation information at the node level. The code, however has been written with this addition in mind. To execute, type:

```
% EtoNPat <infile> <outfile> <tmpfile>
```

The `<outfile>` is the required .tr file for the Nihongo Tutor. `<tmpfile>` contains the lexemes of the parse trees. They will appear in the last lines of `<outfile>`, and are used for merging purpose.

This section will describe the details of the algorithms involved and provide suggestions for how the translation and pronunciation information can be incorporated. An understanding of how the algorithm works requires a knowledge of the data representation format used by the Nihongo Tutor. Before describing the algorithm, it is a good idea to look at the output format of the ETL-parsr. A sentence '買われない' may have a parse tree and an output (hereafter, referred to as Nihongo Tutor format) as follows:
As the intermediate nodes are not important in constructing the final parse tree, the program first strips off all intermediate nodes before the conversion to the Nihongo Tutor format is done.

After stripping off these nodes, the tree would look like:
The key way to identify a parse tree is to look for the word TOP, which always precedes a parse tree. This is done in the routine \texttt{MakeTree()}, which then goes on to input all of the subsequent lines of the parse tree, assuming it is not corrupted, and creates a general tree. All relevant information is stored in the structure \texttt{TreeNode}, which allows a maximum of MAXCHILD children per node. \texttt{MakeTree()} then returns the parsed tree to the loop in \texttt{main()}. The intermediate nodes are then eliminated with \texttt{CheckTree()}. Another important data structure is the \texttt{Lezeme}. It will contain the string of the leaf nodes in the parse tree. This information must be located at the end of the Nihongo Tutor translation file, so that the word boundaries can be identified.

The next task is to assign node numbers to each node in the created parsed tree. This is done via \texttt{AssignNodeNum()}. It is also in this function that provisions for the addition of translation and pronunciation information has been made. Finally, \texttt{ClimbTree()} is called to fill in the appropriate information in the \texttt{Sentence} structure, which is used to supply the indexing information for each node required by the Nihongo Tutor. The routine \texttt{PutSentence()} is then used to output the sentence information.

The following sections describe these functions in greater detail.

### 4.1 Function \texttt{MakeTree}

The main task of this function is to construct the data structure used to represent a general tree from the format provided by the ETL parser. Each line of the parse trees is input and the tree is created top-down, in an iterative fashion. As such, a parent pointer is needed to traverse up the tree after a leaf node is encountered.

A \texttt{NewLevel} variable is used to keep track of which level of the tree is currently being built. The level information is determined simply by counting the first position of the first \texttt{-}, a task which is performed in \texttt{FindLevel()}. This depends solely on the way the ETL parser outputs its parse tree. Observe that the first \texttt{-} occurs at the \textit{3n+2} column. Therefore, by finding \textit{n} one can then determine the level of the tree. Thus 文 is on level 1 and 語幹 is on level 8. Observe also that immediately after a sequence of leaf nodes, there will be a line without any \texttt{-} character. This will indicate to the algorithm that it should progress to a lower level for any additional input. Thus another variable \texttt{Level} is used to keep track of the old level, where tree building should continue.

Finally, an array called \texttt{ListTable} is used to keep track of all of the leaf nodes. This information is used in \texttt{ClimbTree()}. 

### 4.2 Function \texttt{CheckTree}

This routine simply traverses the tree recursively and eliminates all nodes with only one child. In addition, this routine returns the number of nodes in the new tree.

### 4.3 Function \texttt{AssignNodeNum}

The only task performed by this function is to assign node numbers to individual nodes. Node numbering is carried out in preorder, i.e., top-down, left-to-right. In the process of numbering, the relevant subfields of the \texttt{Nodeinfo} field of tree are initialized as follows:
These variables represents all of the information requiretl at each node by the Nihongo Tutor translation file format. Provisions for the inclusion of translation and pronunciation information have been provided. The kanainfo and translation field of tree->NodeInfo are currently set to NULL. If needed, additional routines may be written to perform an intelligent dictionary lookup for such information and have it included. This, however, allows only lookup based on the information in the current node, and not on any ancestors or descendants, which may be required in practice. In this case, a more sophisticated tree traversal technique may need to be developed.

4.4 Function ClimbTree

The tree is traversed bottom-up in this function (using the ListTable array) to fill in all of the relevant information for each node. This means filling in all the fields of tree->NodeInfo, by calculating the string position, length, etc. In addition, the parent information is filled in. At the same time, the NewSentence structure is constructed and filled in as the tree is traversed. In essence, it contains the Japanese sentence, the pronunciations for all kanji concatenated together, the English translation of the entire sentence and the individual translations all concatenated together. The information in sentence will be the first two lines of each record for the parse tree that is output. In the case of the example shown, it corresponds to the following.

```
貫わせられない。"-1
(null)"-1
```

Since there is currently no pronunciation and translation information, the (null) appears in place of the English translation. The code for calculating the position and length of translations and pronunciations is included in this routine. If this information is inserted in the kanainfo and translation field of tree->NodeInfo, then these calculations are done, otherwise, the respective entries are zeroed.

4.5 Function PutSentence

This function simply outputs all of the information in the Sentence and NodeInfo structures in the format required by the Nihongo Tutor translation files.

Termination of Conversion

The reading/conversion process is terminated when MakeTree() hits the end-of-file when reading in the input file. At this point, MergeFiles() is called to merge the lexeme information stored in <tmpfile>, describe above, with <outfile>, which up to this point, contains only the records of the parsed trees in Nihongo Tutor format.
5 Dictionary formats to support the Nihongo Tutor

In section 4, it was mentioned that the automation of parse-tree generation for the Nihongo Tutor still requires manual intervention for supplying translation and pronunciation information to each node. In this section, this issue is addressed through the investigation of a dictionary organization that assists in this task.

One major problem with a good on-line dictionary for lookup is that none is yet available. There are many small pieces, each with its bit of information, be it pronunciation or translation. Some good public-domain dictionaries are listed below:

**Wnn dictionary**: these contain only the pronunciations. But they contain words from various specialized areas, such as technology, area and people’s names, biological terms, etc. Such dictionaries end with a .u extensions and reside in the wnn source directory in `-/wnn.4.0.3/Wnn/public`.

Another good feature is that the dictionary entries also contain grammatical information. Typical entries appear as follows:

| かいり | 解離 | 名詞 | 1 |
| かおん | 加湿 | 名詞 | 1 |
| かおんぱん | 加温盤 | 名詞 | 1 |

**Edict**: Edict is an attempt to produce a public domain Japanese/English dictionary in machine-readable form. It was intended initially for use with MOKE (Mark's Own Kanji Editor) and related software such as JDIC and JREADER (all available on PCs only). All entries in edict have pronunciation and translation information. The format of the entries is:

```
KANJI [KANA] /english_1/english_2/.../
```

or

```
KANA /english_1/...
```

Typical entries are given below:

京頼 [きょうとう] /Kyoutou (pn)/
供应 [きょうきゅう] /supply (vs)/provision/

*Edict* is available at monu6.cc.monash.edu.au. It has been ftped and placed in `-/edict.j`.

**Other local dictionaries** These dictionaries are part of the Nihongo Tutor and can be used for providing useful translation and pronunciation information. All such dictionaries are found in `-/dictionaries`. Of special interests are the dictionaries in sub-directories character, newspaper, etc. The first contains all of the jouyou kanji, with each associated Katakana and Hiragana pronunciation, and translation. The second contains a list of 2-character-Kanji word, with the Hiragana pronunciation and translation. The last contains a list of technical words. The first two are named CDKanji and GDKanji respectively, and they will be referred to by these names in the following text.
The task then is to try to combine the informations scattered over these many different dictionaries. This entails designing a structure that permits both easy lookup of entries and almost-automatic combining of information. This structure must also be able to handle multiple pronunciations associated with a word including both On and Kun readings. In addition, attentions must be paid to eliminating redundant information during the process of merging dictionaries. All of these issues are addressed and answered in the following sections. Detailed descriptions of the data structures used and the algorithm needed to do the searching and merging are described. Suggestions for future improvement and enhancement are also discussed.

It is worth noting that even if a good online dictionary is available for fast lookup of information, the structure to be described could still be used by converting the dictionary format into the one needed, instead of rewriting applications to do the lookup.

All programs related to dictionary processing are kept in -/yeo/ETLtoNTS/dict with the dictionaries in the sub-directory dict-out.

5.1 Format Details

Throughout the discussion, the following example from CDKanji will be used:

加 716 19 5 か．くわ(える) add, append くわ(わる) join, take part (in)

Observe that there are two possible interpretations to the word 加, one related to the idea of adding and appending, and the second relating to involvement. In what follows, a set will be used to describe a particular interpretation of a given word. A set will contain a list of related English translations, and a list of possible pronunciation for this interpretation. In the above example, 'add' and 'append' belong to the same set. But they will be treated as different translations, for the purpose of more accurate Japanese-to-English translation.

To facilitate easy manual addition of entries to the dictionaries, a raw format is available. In actual processing, another format, hereafter referred to as the Compiled Dictionary format, will be used. Such files will have an .out extension. A program procdict is available to convert a dictionary from the raw format to the Compiled Dictionary format.

5.1.1 Raw format

The raw format must conform to the following property:

1. all relevant information must be on the same line.

2. the order of the information is important: the Japanese word must first appear, followed by its first set of pronunciations and English translations, then optionally followed by more sets of pronunciations and English translations.

3. a space is used as a delimiter between each field of Japanese words, each set of pronunciations and each set of English translations.

4. in cases where the pronunciation depends on what follows the kanji of interest, a ( ) should be used to enclose the words. Possibilities of such words are separated by a /.

5. any other information can be specified between the Japanese text and the first set of translations or even before the Japanese text so long as it is stored in ascii.
In dealing with translation, a ‘,’ or ‘;’ will be treated equally to separate different translation in a set. Thus in the above example, there are 2 possible translations given the first interpretation.

Observe that CDKanji and GDKanji conform to this format. In particular, the above example is in raw format.

5.1.2 Compiled Dictionary Format

The compiled dictionary format for the above example becomes:

```
 加<DELI>
 add<DELI>
 append<DELI>
 カ<DELI>
 <お える<DELI>
 join<DELI>
 take part (in)<DELI>
 <お わる<DELI>
```

where `<DELI>` denotes the field delimiter. In practice, no carriage returns of line feeds separate the fields. Only `<DELI>` is used. For the purpose of better differentiating between the Japanese characters and the English characters, the `<DELI>` is usually a control character, i.e. one that yields 1 when tested with `iscntrl()` of the `<ctype.h>` library. Observe also that a space separates a pronunciation with the text that follows. In general, the structure is:

```
 Jap_Word<DELI>
 set_1_translation_1<DELI>set_1_translation_2<DELI>...
 set_1_hiragana<DELI>
 set_1_katakana set_1_katakana_follow_1 set_1_katakana_follow_2 ...
 set_2_translation_1<DELI>set_2_translation_2<DELI>...
 set_2_hiragana<DELI>
 set_2_katakana set_2_katakana_follow_1 set_2_katakana_follow_2 ...
 ...
```

Again, the fields are shown on different lines for the sake of clarity. It is possible to have only one Hiragana or Katakana pronunciation. A `$` is used in those cases where there is no translation.
5.1.3 Internal Representation

The internal representation is defined with the following data structure. This definition is found in `dictstruct.h`.

```c
typedef struct _transl { /* list of English translation */
    char *eng;
    struct _transl *nextptr;
} transl;

typedef struct _hpronun {
    char *hpronun;
    int numfollow;
    char *follow[MAXFOLLOW];  /* maximum number of possible follow */
    struct _hpronun *nextptr;
} hpronun;

typedef struct _kpronun {
    char *kpronun;
    struct _kpronun *nextptr;
} kpronun;

typedef struct _info {
    transl *translptr;
    hpronun *hpronunptr;
    kpronun *kpronunptr;
    struct _info *nextptr;
} info;

typedef struct _dictentry { /* each Japanese word entry */
    char *japword;
    info *infoptr;
} dictentry;
```

The use of the various structures is summarized as follows:

- each dictionary entry is contained in a `dictentry`. The `inforptr` points to the information pertaining to this word. It is maintained as a linked list.

- each `info` structure is the same as the `set` structure defined above. It contains informations for the various translations as well as the On and Kun pronunciations.

- the `kpronun` and `hpronun` correspond to records to store the Katakana and Hiragana pronunciations respectively. In `hpronun`, a `follow` array is maintained to keep track of the words that could follow.

- the `transl` structure is a linked list of translations for the particular `set`.

The structure for the example above is shown on the following page.
Figure 1: Illustration of dictionary structure
The program `readict` can be used to see the content and structure of a *Compiled Dictionary* file. For the example, the program output would appear as follows:

Japanese word:  加

Set # 1

<table>
<thead>
<tr>
<th>English Translation</th>
<th>Katakana pronunciation</th>
<th>Hiragana pronunciation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>ガ</td>
<td>&lt;が --&gt; える</td>
</tr>
<tr>
<td>append</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set # 2

<table>
<thead>
<tr>
<th>English Translation</th>
<th>Hiragana pronunciation</th>
</tr>
</thead>
<tbody>
<tr>
<td>join</td>
<td>&lt;び --&gt; わ</td>
</tr>
<tr>
<td>take part (in)</td>
<td></td>
</tr>
</tbody>
</table>

5.2 Merging Different Dictionaries

In order to create and maintain a good Japanese dictionary for the purposes discussed previously, it is quite useful to be able to automatically merge and combine different dictionaries.

The program `mergedict` is developed with the above goal in mind. It is designed to achieve the following:

1. take as input a sorted dictionary that may contain redundant information, i.e. extraneous entries, and produce an output version with all of the extraneous details removed according the rules defined below.

2. the dictionary used as input to `mergedict` must confirm to the *Compiled Dictionary* format described in the previous section.

3. eliminate lines that are exact duplicates of one another first. This is useful since there are many occasions when a word has only pronunciation information and such information can occur in different dictionaries.

4. in cases when only the first field, i.e. the word of interest, is the same, merging is performed according to the following rules:
   - merge all English translations associated with the same pronunciation. In cases where the translations are the same, only one copy is kept. Similarity between translations can be measured using more sophisticated algorithms that use more sophisticated approximate string comparisons.
   - combine sets with different pronunciations. No effort is made to decide as to which one should appear first in the output. In the future, sets without pronunciations can be made to appear later than complete sets.

5. the output will retain the sorted order of the input. Note that if the input is not sorted, you will not get the desired output. Use `jsort` (see section 5.3) to sort the input.
As an illustration, consider the following before being processed by mergedict:

Num. of entries : 3
Japanese word : 一時
Set # 1
   English Translation : temporarily
   Hiragana pronunciation : いちじ

Japanese word : 一時
Set # 1
   English Translation :<none>
   Hiragana pronunciation : ひととき

Observe that いちじ occurs 2 times, once with "temporarily" as the translation and once with "a time" and "at one time" as the translations. ひととき occurs once with no translation at all. After being processed by mergedict the resulting output dictionary is:

Num. of entries : 1
Japanese word : 一時
Set # 1
   English Translation : <none>
   Hiragana pronunciation : ひととき
Set # 2
   English Translation : at one time
   Hiragana pronunciation : いちじ
5.3 Utility programs and routines

This section describes a few utilities to support the building and management of dictionaries. In some cases, the algorithm will be described so that the techniques can be applied to create similar programs. Most importantly, there are a few fundamental functions, which are useful for handling Japanese text, that will be of special interest as building blocks for more applications.

All processing is performed in Shift-JIS character mode. Thus it is important that any file be converted to this format first.

5.3.1 Sorting Japanese text: jsort

To make use of the available resources, the sort program available on every Unix system is modified to handle Japanese text. The new modified version is called jsort. The executable and source are kept in -/bin/sparc and -/bin/src respectively.

The modification is trivial. Since the problem with sort is that it works with characters, which in C is defined to have only 7 bit, it suffices to fix the way the characters are compared with each other. In sort, this comparison is done via the cmpa() function.

```c
cmpa(pa, pb)
    register char *pa, *pb;
{
    while(*pa == *pb++)
        if(*pa++ == '\n')
            return(0);
    return(
        *pa == '\n' ? fields[0].rf lg:
        *--pb == '\n' ?-fields[0].rf lg:
        *pb > *pa ? fields[0].rf lg:
        ~fields[0].rf lg
   );
}
```

To force the comparison to be done on all bits, replace cmpa() by a new function, say jcmpa() and just replace the line

```c
*pa == *pb++
```

with

```c
(unsigned) (*pa) == (unsigned) (*pb++)
```

Do the same to the > comparison. Finally, set the global pointer-to-function variable compare to jcmpa.

2 Removing single kanji entries: stripsingle

As there are only a finite number of single-kanji word entries, it is a good idea to separate out all such entries into a separate dictionary. This program simply strips all single-kanji word entries from a Compiled Dictionary format (see section 5.1.2). It takes input from either the command line or standard input and outputs to standard output.
As an illustration of its use, suppose you want to merge two dictionaries, dict1 and dict2 that are in raw format (see section 5.1.1) and have all single-kanji entries removed. In addition, both the dictionaries are not sorted. Calling the desired file (of Compiled Dictionary format) mdict.out, you can do the following:

```%
% prodict; dict1 > dict1.out
% prodict: dict2 > dict2.out
% cat dict1.out dict2.out | jsort | stripsingle | mergedict > mdict.out
```

To view the result, just type

```%
% readict mdict.out
```

### 5.3.3 Conversion from other dictionary formats

As mentioned before, public domain online dictionaries that are of special interest are the wnn and edict dictionaries. Two programs, getwnndict and getedict, are written to convert these two formats into the Compiled Dictionary format.

It should be noted that the edict does not conform exactly to its stated format. It is not uncommon to find missing English translations or other strange undocumented features in the entries. Thus, the task of converting the entire dictionary still remains a challenge.

The programs are trivial. They make use of the functions in japutil.c (see section 5.3.4) to perform the conversion. This is a good place to look for examples of Japanese text manipulation.

### 5.3.4 Useful functions for Japanese text manipulation

The functions described in this section are utilities available in japutil.c and readictutil.c. The functions in these two files can be used for general Japanese text processing and are not just limited to dealing with the handling of Japanese tlictionaries. The first four are found in japutil.c with the rest in readictutil.c.

- **JapPrefix** checks to see if the second argument is a prefix of the first argument.
- **IsJap, IsHiragana, IsKatakana, IsKanji** checks to see if the first 2 characters of the character pointer argument are a Shift-JIS Japanese, Hiragana, Katakana and Kanji character code respectively. This is useful for testing a particular field for the kind of code it contains.
- **getjapstr** returns the next longest string of Japanese characters from the given string argument (as a double pointer to character). This is useful for grabbing off the longest string of Japanese text from the current buffer pointer. Note that the string pointer will be updated.
- **getengstr** returns the next longest string of ASCII characters (excluding control characters) from the given string argument (also as a double pointer to character).
- **readdict** reads in a dictionary of format Compiled Dictionary from the file pointer argument and returns the structure of type dictentry in the second argument.
- **readentry** actually does the reading of each entry from a line of characters. Builds the data structure with simple scanning done via yetjapstr and getengstr.
printdict, printdictentry prints out the entire dictentry structure and the individual entry of the structure respectively.

printdictformat prints a dictentry structure to a Compiled Dictionary format. This is used for writing an in-memory dictionary to an external file.

copydictentry, copyinfo, copykpronun, copyhpronun, copytransl copies the entire dictentry structure and its various subfields respectively.