БГДЕЖЗИКЛНОП

Cyrillic Font Project Page

Description:

As with most non-roman fonts, Cyrillic fonts exist in a variety of incompatible character mappings. This can cause serious problems, especially in shared documents. In order to eliminate such problems, and particularly the difficulties non-standard fonts pose for a structured curriculum, the Department of Slavic Languages at Brown University began a standardization program. The goal of the program was to have shared course materials in one font. Once the font was chosen, the task was to convert old documents into the new font, and this required a program which could search through each document, translating, character by character, any part that was not in the specified font.

The project of writing this conversion utility was taken on by STG, and we first identified the main requirements for the final product:

- The conversion method had to be robust to any future conversion problems (one to many and many to one conversions)
- The converter had to be able to read RTF header and control sequences
- Only applicable fonts were to be converted; the conversion had to be font-specific.

In addition, it was hoped that the process could be encapsulated to, in some way automate the save-as-RTF step. Also, as other applications for such a program were easily imagined, the script was to be kept as general-purpose as possible.

Further technical information and version history is also available.

Latest Version:

Here are copies of the latest versions of the utility:

- MacPerl version (v3)
- Unix version

Character Maps:

Here is a complete list of current Character Maps

STG HOME

БГДЕЖЗИКЛНОП

Cyrillic Font Project

Technical Information

Algorithm:

The script was written in PERL or more specifically, MacPerl, as the translations would take place on the macs at the department. All of the folowing code fragments are in PERL, and are taken more or less directly from the script itself.

The first task was to determine a working idea of the translation algorithm. The translation would need to be able to convert not only single ASCII points, but multiple character sequences to deal with some accenting problems. To start with, the problem was simplified. First the algorithm would search each character, and look up its translation. This takes care of one to one, as well as one to many translations. After the simple translation algorithm was developed, a second-pass search might be done to find any multiple-character sequences.

The translation maps would be read in from a file, in order make the translation process generalized. To make sure that the map-file format would not have to be revised, it was designed around the most general parameters which could be conceived. This meant that the format would have to be able to handle character sequences of differing lengths. In addition it was decided that the map-files should be fairly legible in their raw format. Finally, the font names should appear in the file itself, rather than the file name. (Font names can be quite long) The format decided upon was as follows

```
Lektorek Russian --> CyrillicII

3b --> fc

3c --> c7

8e --> 65 b1

3e --> c8
```

The simple translation algorithm follows fairly quickly, once this map is loaded into a hash:

```
s/(.)/$map{$1}/g
```

The second-pass algorithm would need to be a little smarter that that. It needs to search for different strings, so alternation would appear to be needed. Also, since they're literally in the regex, they need to be quoted. So when, in reading the map-file, we encounter a many to one translation, we add it to a different hash, called map_tto. When the map-file is read completely, the keys to map_tto are quoted, individually, and joined on '|', the alternation character. This is the search string in the following code fragment:

```
s/($search)/$map_tto{$1}/g
```

RTF Reader:

The documents were to be read in RTF format, which would allow the preservation of formatting, but still allow us to work on ASCII text. The RTF reader had to be able to understand the basics of RTF, ignoring most control words, recognizing font changes, special characters, hex-codes, etc. The RTF specification was obtained from Microsoft's WWW site.

The reader was designed recursively. Each level would handle a block, the function was called `parse_block()`. The text would be searched for an RTF meta (a forgivingly small set of three characters: \,{, and}) open curlies would call another level of interpretation, closed curlies would finish off a level, and backslashes would be parsed for control word content. Each instance of the reader would store its own font number, which it inherited from the previous level. Any intervening text would be translated if necessary, or copied out directly, by another subroutine `write()`

The RTF header also had to be read,the header function was called `read_header()`. The imporetant tasks of this function were to find the appropriate (source) font tags, and substitute in the new font names. The font tags are later used by `write()` to detemine if a passage needs translating.

Version History:

- **29/8/96 Version 3** added support for multiple concurrent translations.

  Keeps hash of references to hash-maps (keyed on font tag).

- **26/8/96 Version 2.5** Fixed memory usage

  Now makes only one copy of document in memory. RAM usage is now about 3000K.

  Keeps file in memory as an array, searches only through the necessary lines of text, using new search subroutines. This cuts down on long string copies. ($`, $&, and $' are much much shorter)

- **Version 2** Word Save as RTF encapsulation.

  Perl Script reads in file names, and sends literal-text applescripts to Microsoft Word for each file. This approach is used to avoid a nasty bug in Word 6.0.1 AppleScript support.

  The whole script is encapsulated in one short MacPerl runtime, which 'requires' the other scripts, saved in plain text. This makes minor bug fixes possible without the actual MacPerl Editor.

- **Version 1**Basic RTF conversion utility. includes one to one, one to two and two to one conversion capability.