How Languages Work In MOZ

Robin Lee Powell

This manual is for MOZ (MOO in Oz) version 1.0.

Table of Contents

Languages (as in the language used to input commands and read descriptions in the MOZ, not programming languages) are handled very differently in MOZ than in any other MUD I'm aware of (not that I've looked very hard). This document describes the differences.

1 Localization

This was taken from the Design doc, and needs editing and stuff.

Localization is the proccess of converting a particular program to work with a given language. Localization is rather important to the MOZ project as it was originally concieved as a way for me to write a mud that responded to commands in lojban (see http://www.lojban.org/), but I don't want to leave out the ability for MOZ to work in English or any other language.

Note that all localizations in MOZ use standard 2 letter codes where available, and made-up 2 letter codes where not.

The way that localization is normally handled, as far as I'm aware, is to have a big file somewhere that contains the text of all the messages that the program ever outputs to users. This is very impratical in something that grows with user programming the way MOZ does. Instead, the way I have decided to handle it is to have all the information for all supported languages stored on every object.

There are basically three issues that need to be dealt with for localization: parsing, verbs, and output strings.

Parsing is, unavoidably, going to be different across different languages. The way that this gets handled is to have different parser classes for each language, and to have the command line argument that sets the language have the effect of telling the Server which of those classes to use for the Parser object it associates with each player connection.

Verbs are the various commands associated with any given object. Essentially, verbs are a way of translating between what a player types (i.e. "get chair") and an Oz method (probably, eventually, the 'get' method on the 'chair' object). The names of the verbs are going to be different across different languages ('get' is 'cpacu' in lojban) as are the parts of speech ('chair' is the 'direct object' in English, but the 'second sumti' in lojban). However, all the different languages' ways of saying "get chair" are going to map to the same method on the same object (one hopes). So, each object has seperate lists of verbs for each language, but those lists should all essentially be doing the same thing.

Sooner or later, strings need to be output to the user. The way localization of arbitrary strings is handled in MOZ is to have the strings stored as Oz records named string with one different field for each language (using the two letter codes). Which string out of the list to pick is handled by a substitution method on the Connection object which also handles substitution of codes that represent the player's name and such.

Note that this system allows for different people on the MOZ to be using different languages to interface to the MOZ.

Also, storing verbs and localized strings as attributes allows the possibility of writing a piece of code that writes a file with Oz code in it that shows every localized piece of information currently in the MOZ. This can then be added to with a new language's information and then executed.

2 A Note On Character Sets

The inital release of MOZ is only able to parse ASCII strings. However, there is no limitation, at all, in the program that causes this to be the case; it is merely due to the definitions of 'letter' and 'number' in the ParserMonads class.

The MOZ has been shown to pass UTF-8 untouched when players say something, for example. I, personally, know nothing about any character sets other than ASCII, which is why I haven't changed the parsing. If someone else wants to do localization for a language that uses a more complicated character set, I would love to see the results, and incorporate them into the main distribution.

3 Conceptual Overview

First off, I want to be absolutely clear that this document has nothing to do with programming languages. It is about how MOZ deals with human languages. This is rather important because MOZ was originally concieved as a way to make a mud that could be interacted with in lojban (See http://www.lojban.org/). The main MOZ code has English and lojban modes.

MOZ is unique, as far as I'm aware, in supporting more than one language at once. Different players can see things in their own language, and their actions (although not their words) will be seen by each player in their own language. So, if a player named 'Foo' opens a door, an English speaking player might see

```
Foo opens the door.
```

whereas a lojban speaking player might see

```
la fus. kargau le vorme
```

in response to the same action.

Note that the player can have different names in each language (Foo is not proper lojban).

The command one types to open the door will also be different depending on one's language:

```
open door
```

versus

```
se kalri le vorme
```

Each player can set their preferred language on the fly.

4 Parsing

Obviously, this sort of functionality is a little tricky to achieve, so here's how we do it.

When a player enters a command, it is parsed by a parser appropriate to their language. The command is parsed into a command word(s) and numbered arguments. Other grammar words are thrown away. The parser then asks around for an object willing to proccess the command given. On each object, the verb records are then used to map the command word(s) to a method on the object and to map the numbered arguments to arguments to the method.

The method is then called with these arguments, and an argument specifying the language the request came in. Normally, the language argument should only be used to pass on to other methods. In particular, for example, the word for 'door' is very different in lojban, so the method for the 'open' command (which will be the same method for both languages, as the same functional thing is being done) will need to pass the 'door' or 'vorme' argument on to a method that looks for an object of that name; that method will need to know what language it should be using to do name comparison.

If the method is happy with the arguments (i.e. the open command on a door will accept the example given, but the open command on a jar will not), it returns an indication of same and the parser stops proccessing. Otherwise, the parser moves on to the next object.

All strings in the MOZ should use the string record format that allows for multilingualism, which is another use for passing language information to methods: they can then output strings in the appropriate language.

In case of failing to find language information, the MOZ will fall back to a default which can be set by a wizard, and which starts at English. If the default can't be found, the first language for which the appropriate information can be found is used. If that fails, an error results.

5 Output

Each verb should have localized strings which are used to output results in whatever languages it accepts input in. In some languages, the strings will be blank (because they represent things not necessary in than language in that case). For example, open might output:

PRE˙VERB + VERB + POST˙VERB + ARG1 + POST˙ARG1

In both english and lojban, the PRE˙VERB will be the user's (localized) name. VERB will be the localized verb name. However, for both of them, POST˙VERB will be empty.

The localized strings are stored by keys (which are Oz atoms) in the LanguageStrings object. Each database has only one such object. It is of the LanguageStrings class. Strings are stored using the `setLanguageString` method; see the Programmer's Guide for more details.

5.1 Localized String Contents

Each localized string key looks up a record. The record contains a language code label for each known language. The label points to the appropriate content in that language. Example:

```
string(
en: "[unknown; timeout occured]"
jbo: "to na djuno ni'i le nu dukse le ni denpa toi"
      )
```

The only special case is that some languages do unusual things when a word is the first word in a sentence. These sentence-initial productions are handled by appending "SI" after the language tag:

```
key: indefiniteVowelArticle string: string(
    en: "an "
    enSI: "An "
    lb: "le "
)
```

5.2 Special Strings

Certain strings are rather more important than others, because they have low-level functions, such as handling article issues and capitalization.

There needs to be a string named `<language>Name` for each known language, with the full name for that language in all known languages as the values.

The key `listSeperator1` stores the equivalent of the English comma, for lists of things. The key `newlineString` stores the localized newline character.

The following are used for articles:

- properNameArticle
- properNameVowelArticle
- indefiniteArticle
- indefiniteVowelArticle