

Foreign Language Support

Copyright Bruce Wilcox, <mailto:gowilcox@gmail.com> www.brilligunderstanding.com
Revision 7/18/2021 cs11.5

Foreign Language Overview

ChatScript comes natively with full English support. If you want to use a different language you need a variety of things. * Pos-tagging support (optional if you don't use pos-based keywords) * Spell check support * Concepts in the language (that you use) * LIVEDATA substitutions appropriate to the language * Patterns in the language * Output in the language * Lemmas (comes with foreign pos-taggers or available from `^pos(canonical)`)

ChatScript has a command line parameter `language=` that tells CS the language you intend. It defaults to `ENGLISH`. The effects of this parameter are generically these, unless special information is given below about the language: * If not `ENGLISH`, internal pos-tagging (other than marking possible english tags and numbers and dates and such) and parsing are disabled. * If `treetagger` is licensed and has that language, it will pos tag. * The system will use `DICT/language`. * The system will use `LIVEDATA/language` * The script compiler will automatically compile lines marked to be conditionally compiled with the language (see language comments). * The system will store the user's topic file suffixed by language as well.

Embedded Foreign Language Support

Japanese

ChatScript uses the std CS engine, has no special dictionary or LIVEDATA for Japanese. It knows something is japanese because either the a command line parameter has `language=japanese`, or because japanese characters are being used in compiling a pattern (`^compilepattern`), or because a global variable is passed in via `^testpattern` `"$language = japanese"`.

CompilePattern

ChatScript directly supports Japanese coding in `^compilepattern`. A "word" is a sequence of characters with no spaces in it (except for double quoted strings). Note in the texts below, `?` stands for some japanese character.

Scripters are expected to keep Japanese “words” separate. E.g. ?????? would in script be: ?? ?? ?? .

Of course if they actually want to match the sentence exactly anyway, they can keep them all together.

Either way, patterns that are compiled and that have tokens of Japanese UTF8 characters will expand those tokens to put spaces between each character. Each character thus becomes a word internally.

So this, (_??? \$var:=_0), which a scripter writes as a single word to be found and memorized, internally is sent to cs as this:

```
[{"dse": {"compilepattern": "( _??? $var:=_0 )" } } ]
```

and CS returns this for internal use, where each character is now a separate word: [{"dse": {"testpattern": {"input": "???? ?,"patterns": [{"pattern": "^(_(? ? ?) :6\$var:=_0)" } } } }]

This pattern says that the given character sequence MUST be detected as a contiguous sequence. You can use the usual exclusion and memorization operators on such words, and they will be applied to the sequence.

^testpattern

Correspondingly, if you set a global variable “language” to “japanese”, then in calls to ^testpattern (the condition of a node) will adjust user inputs that have Japanese characters in the following ways:

Normal character clusters are separated into discrete characters. These can match the patterns that have been compiled. English AND Japanese terminal punctuation marks corresponding to period, question, and exclamation are internally converted to English (because they will be stripped off the sentence as per usual). If the pattern matcher is required to memorize a “word”, it will find the separated characters and join them back together to return the value. Thus the above testpattern call will return: “output”: {“newglobals”: {“\$var”: “???”} , “match”: 0 } }

Concepts are written per usual, using Japanese words (characters without spaces).

Sentence Limit

CS has a limit of 254 words per sentence, which in this context means 254 characters per sentence of input. Where terminal punctuation is detected, that ends a sentence. Where the limit is reached before finding any terminal punctuation, CS does what it does with English, it breaks the sentence at that point and remaining words will be allocated to the next sentence. Ryan ran some numbers on about 100,000 user inputs from our jp bots. The average character

length was 33, and inputs greater than 254 characters only account for 0.5% of all volleys.

Numbers

For english numbers as digits residing adjacent to japanese characters, the english number is broken off intact. Thus `??72????` becomes `? ? 72 ? ? ? ? ?` on inputs

GERMAN

The spell checker has code to break apart an unknown compound word into its separate recognizable pieces, based on <https://www.dartmouth.edu/~deutsch/Grammatik/Wortbildung/Komposita.> . So “Esszimmer” (dining room) becomes `Ess Zimmer`.

CS comes with a german dictionary, and spell checking will use it. In particular, if the input lacks appropriate accent marks, CS will likely fill them in for you.

SPANISH

The tokenizer will simply delete any upside down question or exclamation marks.

CS comes with a spanish dictionary, and spell checking will use it. In particular, if the input lacks appropriate accent marks, CS will likely fill them in for you.

INPUT and OUTPUT

ChatScript supports UTF8, so making output or patterns in the language is entirely up to you. Ditto for LIVEDATA.

ChatScript supports two kinds of conditional compile comments. Single line comments look like this:

```
#ENGLISH this line will compile if the language is English.  
#GERMAN this line will not compile if the language is English.
```

As always, such comments run til end of line. The other comment is the block comment like this:

```
##<<ENGLISH these lines will be compiled under English  
until a normal closing block comment ##>>
```

Using conditional compilation, you can make English and other language versions of code sit side by side if you want to.

DICTIONARY

The dictionary file can be just a list of words of the language, one per line. You must list all conjugations of a word because there is no in-built support to figure that out. You may also add english equivalent pos tags (see examples in existing foreign language dictionaries) if you want to use existing keywords tied to pos-tags.

In addition to normal words, there is a file `LIVEDATA/.../numbers.txt` that for a language describes a number word and what it's implied number meaning is.

`:buildforeign language` can be used to rebuild a foreign dictionary given the rawwords data in TreeTagger directory (which you dont have) and

POS-TAGS AND LEMMAS

If you want actual POS values and lemmas (canonical form of a word), you will need a POS-tagger of some sort or use `^pos(canonical)` on a word. While it is possible to hook in an external tagger via a web call, that will be noticeably slower than an in-built system. You would call the service and then appropriately decode its output using `^setcanon`, `settag`, `^setrole` (if you get such from external service), and `^setoriginal` (maybe).

ChatScript supports in-built TreeTagger system, which supports a number of languages. However, you can only use this if you have a commercial license. You can try it out using `^popen`, as is done in the German bot, however it will be slow because it has to reinitialize TreeTagger for every sentence. The in-built system does not. A license (per language) is about \$1000 for universal life-time use. You can contact me if you want to arrange to use it.

Ontology

CS ships with a Spanish and some other dictionaries that provides spelling of words (for spell correction) and parts of speech of words. It also ships with some ontologies like `LIVEDATA/ONTOLOGY/SPANISH` which you can do `:build 0` if you have set `language=SPANISH` in `cs_init.txt` file.

Running CS in a foreign language

To run in a language, use the command line parameter `language=`. For any CS instance, you can only support one language at a time (because CS cannot load

multiple dictionaries or autodetect a language). So if you want to have servers for multiple languages, you need multiple servers, each routed to appropriately.

The user's topic file is named by username-botname-language (which english just omitting the -language component). This means that a user chatting with a german CS server is having an independent conversation to any conversation he had with the english server. If you use LTM files to store long-term data about the user, you can choose whether such data is unique per language or shared, since the name of the file is written in script.

If `language=ideographic` is used, then spell check is disabled and tokenization will make each character be a token. This is useful for languages like Korean and Chinese.

Translating concepts

There is built-in code to translate concepts using Google Translate. It requires you have an api key for Google Translate (but you can sign up for free and get \$300 worth of credit good for 3 months which is enough to do all your translation work probably). You tell CS this as a command line parameter:

```
apikey=AIzaSyAxxxx
```

When I want to translate all level 0 concepts I do the following:

1. erase the contents of TOPIC folder
2. `:build 0`
3. run CS using command line parameter `noboot` and your apikey
4. `:sortconcept x`
5. `:translateconcept german myfilename`

If you run `^csboot` and that generates new concept data then you need `noboot`, otherwise it doesn't matter.

`:sortconcept x` locates all currently defined concepts (hence just `:build 0`) and writes them out to a top level file named `concepts.top` with one concept per line. This file will be read by the next stage.

`:translateconcept` uses the apikey. It reads each line of `concepts.top` (1 line per concept) and calls google translate for the language you named, saving the results in the path/file you gave. Currently this only recognizes the following language names: german, french, italian, spanish, russian, hindi. I could add more if needed.

The resulting file will automatically prepend each line with conditional compile markers for the language you named, so you can directly add it to your bot and it will only compile when you are in that language mode.

If you want to translate concepts from your bot, then do the following:

1. erase the contents of TOPIC folder
2. `:build harry` (or whatever your bot is)
3. run CS using command line parameter `noboot` and your apikey
4. `:sortconcept x`
5. `:translateconcept french myfilename`

If you just want to translate a single concept/topic then you can call

```
:translateconcept ~myconcept french myfilename
```

It will, as a byproduct, provide the sorted english form of the concept on a single line in `cset.txt`. If you dont give a language and filename, then it will just sort your english concept and write it out.