Contextual Machine Learning Through the Analysis and Chunking of Partially Translated Grade 2 Braille

Catherine Ray

Abstract

I developed a machine learning program that uses contextual analysis algorithms to deduce the complex grammar rules of Grade 2 Braille given partially translated text. Using a known set of symbols (Grade 1 Braille), the parser translates the known symbols of Braille to English, and marks leftover unknown patterns and discrepancies. The parser matches unknown patterns to word groups and abbreviations. These learned patterns are stored in a persistent Map[String, TranslationOptions] format.

Acknowledgements

I'd like to thank: Dr. Marr for introducing me to the beautiful complexity of Computational Semantics; Dr. Jahangeer for her amazing encouragement and her willingness to answer questions; Dr. Borne, Dr. Marr, and Dr. Papaconstantopoulos for supporting me and convincing me to major in Computational and Data Sciences; www.braillebookstore.com for printing and distributing books in Grade 2 Braille, excerpts of which helped train CAMEL.

Contents

1 Introduction to Braille

Standard Braille is an approach to creating documents which can be read through touch. As English words are composed of letters, Braille words are composed of Braille cells. A cell consists of six dots arranged in the form of a rectangular grid of two dots horizontally and three dots vertically. With six dots arranged this way, one can obtain sixty three different patterns of dots. The sixty-fourth pattern, a blank cell, represents a space.

In addition to letters, the Braille alphabet includes combination of dots for punctuation, capitalization and numbers. In the Braille alphabet is depicted by a cell that contains six raised dots. The cell is divided into three rows of two columns. A letter is indicated by which dots are raised and which are smooth. Any letter can be capitalized by placing an indicator in front of the letter.

Capitalization is indicated by a cell with only the sixth dot, or the last dot of the cell in the lower right hand corner of the cell, raised while the rest are smooth. This cell appears in front of a letter cell to show capitalization. To capitalize an entire word, two cells with only the sixth dot raised in each cell is placed in front of the first letter of the word.

For example: 'ccc' = ⠉⠉⠉ ; 'Ccc' = ⠠⠉⠉⠉ ; 'CCC' = ⠠⠠⠉⠉⠉

Numbers are represented using the first ten letters of the alphabet, "a" through "j", and a special number sign, ⠼. This number sign is placed in front of a word to convert the entire word into numbers. If one wishes to switch from numbers to letters within a word (i.e. 212a) the letter sign, ⠰, is used.

For example: 'cc' = ⠉⠉ ; '33' = ⠼⠉⠉ ; '3c' = ⠼⠉⠰⠉

Contractions are special characters used to reduce the length of words. English includes contractions (for example, "don't" is a contraction of the two words "do" and "not"). In Braille there are 189 additional contractions. Some contractions stand for a whole word.

For example: 'for' = ⠿ ; 'and' = ⠯ ; 'the' = ⠮ . Other contractions stand for a group of letters within a word.

In the example below, the contraction "ing" is used in the word "sing" and as an ending in the word "playing." Likewise, the contraction "ed" is used in the word "edge" and as an ending in the word "played." '

{ing} = ⠬ ; 's' + {ing} = ⠎⠬ ; 'play' + {ing} = ⠏⠇⠁⠽⠬

{ed} = ⠫ ; {ed} + 'ge' = ⠫⠛⠑ ; 'play' + {ed} = ⠏⠇⠁⠽⠫

Short-form contractions are abbreviated spellings of common longer words. For example: "tomorrow" is spelled "tm", "friend" is spelled "fr", and "little" is spelled "ll" in Braille.

Translating the phrase "you like him" into uncontracted (a.k.a Grade 1 Braille) and contracted (a.k.a Grade 2 Braille), the effect contractions have on sequence length is evident.

⠽⠕⠥ ⠇⠊⠅⠑ ⠓⠊⠍ (Uncontracted)

⠽ ⠇ ⠓⠍ (Contracted)

The reader is encouraged to see the appendices for further information on Grade 1 and 2 Braille translations.

1.1 Binary Braille

The Braille alphabet is depicted by a cell that contains six raised/flat dots, numbered one through six beginning with the dot in the upper left-hand corner with the number descending the columns (see figure below). In order to create a bitstring easily parsable by the computer, "0" = flat, "1" = raised. The 3x2 matrix (Braille cell) is represented as a 1x6 bitstring (Binary Braille).

1	4
2	5
3	6

=

1	2	3	4	5	6

; Thus, "c" = ⠉ ≡

·	·

≡

1	1
0	0
0	0

≡

1	0	0	1	0	0

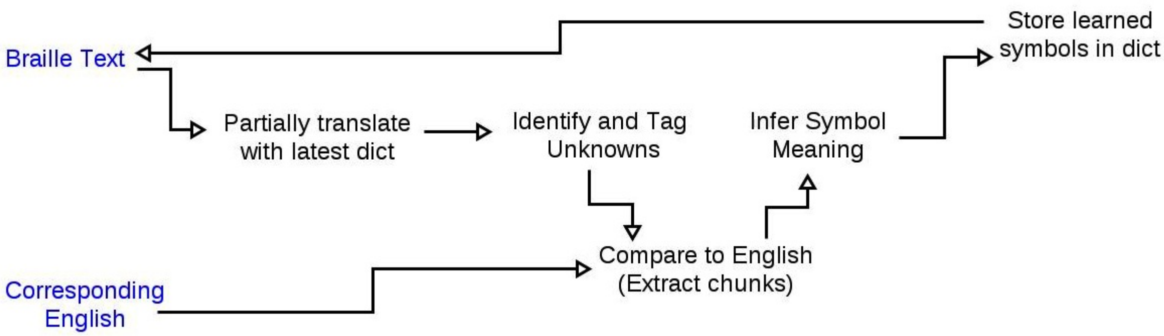
≡ 100100

2 Introduction to CAMEL

CAMEL is an acronym of **C**ontextu**A**l **M**achin**E** **L**earning - with Braille as a language platform, this machine learning program uses the context of unknown symbols to deduce meaning and compress information. Provided the meaning of an initial set of symbols (a dictionary, or `dict`), CAMEL infers the meanings of unknowns and adds these meanings to the `dict`. Some symbols differ in meaning depending on their context. These translation options are stored in the `dict` in the form of `Map[String, TranslationOptions]`.

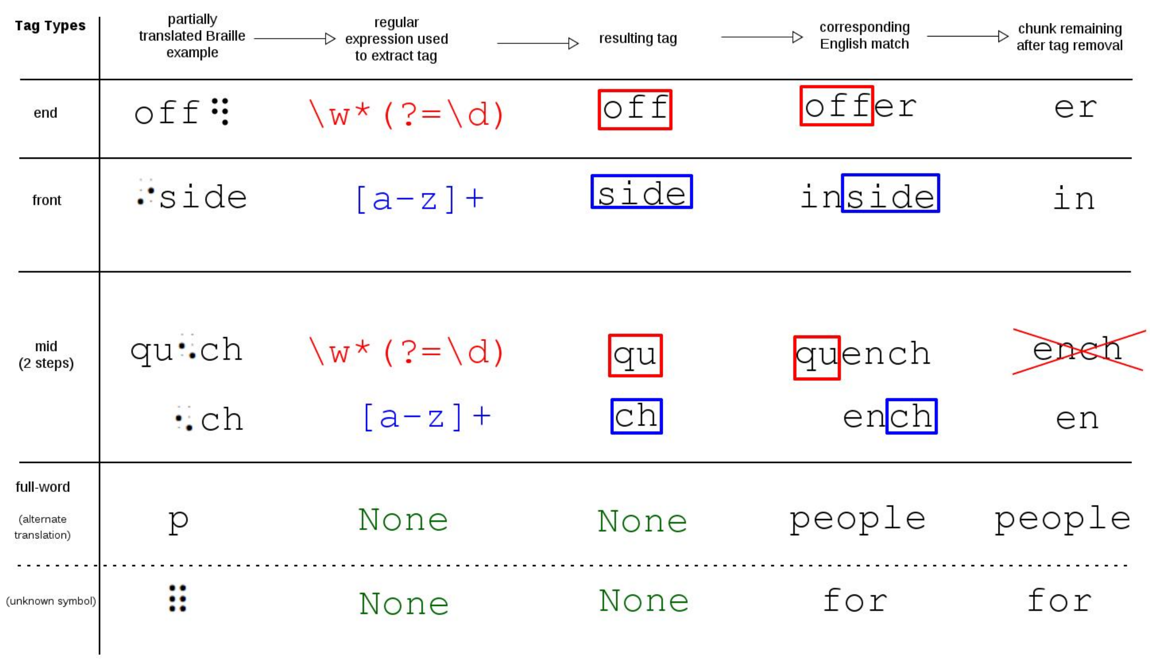
2.1 String Processing Method

CAMEL deduces the complex grammar rules of Grade 2 Braille given partially translated text. It learns new symbols by taking 2 input text files (Braille text and corresponding English text), and analyzing them until all unknowns are identified, their meanings are found, and said symbols and their meanings are added to the dictionary.



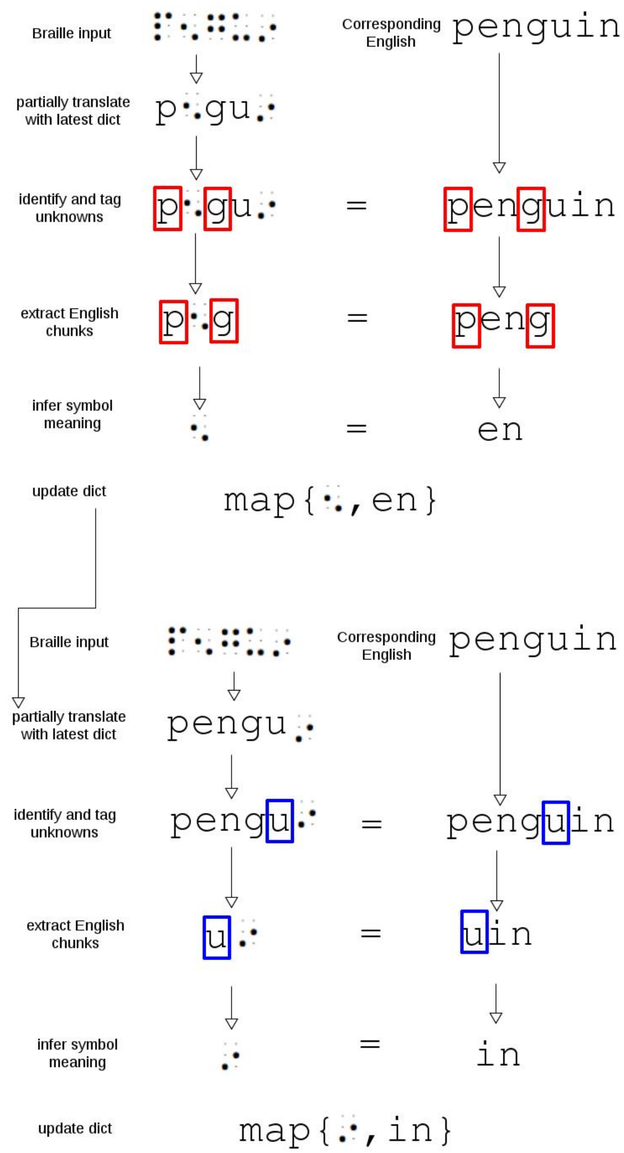
2.2 Methods of Tagging and Text Extraction

CAMEL must *Tag Unknowns & Compare to English(Extract Chunks)* to infer symbol meaning. Four different tag types were used: end, front, mid, and full-word. Below are examples of how these different types of tags were each used to extract meaning.



2.3 Using Contracted Braille As a Platform

Below is an example of the process of *Tagging and Text Extraction*, in which CAMEL infers the symbols that represent *en* and *in* using the word *penguin* (contracted to *p{en}gu{in}* in Grade 2 Braille):



3 Evolution of the Program

CAMEL was programmed in Python 2.7.3. This language was chosen because of the mutable nature of the `dict` constructor[2]. Henceforth, this paper will refer to the set of symbols in Uncontracted Braille as ”G1” and the set of symbols in Contracted Braille as ”G2.” This table demonstrates (using arbitrary example inputs) the evolution of the program, version by version.

Version	Input	(Partial)Translation	Output	Note
0	⠎⠞⠕⠏ ⠊⠞	`stop it`		only accepts G1 input
1	⠌⠕⠏ ⠊⠞	`*op it`		accepts G2 input
2	⠌⠕⠏ ⠊⠞ ⠏⠢⠛⠥⠔ ⠓⠁⠎ ⠋⠇⠂⠎	`*op it` `p*gu* has fl*s`	`st = *` `eninea = ***`	doesn’t differentiate between unknowns
3	⠏⠢⠛⠥⠔ ⠓⠁⠎ ⠋⠇⠂⠎	p**0**gu**1** has fl**2**s	en = **0** = ⠢ in = **1** = ⠔ ea = **2** = ⠂	differentiates between unknowns
4	⠕⠋⠋⠻ ⠊⠎ ⠏⠪⠻ ⠅ ⠊⠎ ⠏⠪⠻	off**0** is p**12** offer is p**3**er k is power	er = **0** = ⠻ ow = **3** = ⠪	improves accuracy of inferred meaning retrieves unknowns to improve partial translation
5	⠅ ⠊⠎ ⠏⠪⠻ ⠭ ⠊⠎ ⠅	k is p**01** k is pow**2** k is power x is knowledge	ow = **0** = ⠪ er = **3** = ⠻ ⠅ = {k or knowledge} x = {x [in word] or it}	detectsa and infers one-letter contractions
6	⠠⠠⠯ ⠼⠚⠃⠉⠙	0 **0234**	and = **0** = ⠯	capital letter and number functionality

3.1 Uncontracted Braille-to-English Translator

The first step was coding a functional program that translated raw G1 input into English text. Using a hard-coded Python `dict`, this was rather simple. CAMEL was given a `dict` containing only G1 symbols.

When the Braille input contained the string `011100 011110 101010 111100 000000 010100 011110`, ”stop it” was returned.

This function can be represented in Python-esque psuedocode as follows.

```
function (array of G2-words):
        translated_word = empty string
        for each word:
                for symbols in word:
                        symbol = dict[symbol]
                        translated_word = translated_word + symbol
                translated_sentence[index of word] = translated_word
        return translated_sentence
```

3.2 Partial G2-English Translator

Recall that G1 is uncontracted Braille. The set of G2 symbols consists of G1 symbols and additional contracted cells. Since the given `dict` consists only of G1 symbols, contracted cells are not recognized.

Recall that $G2 = G1 \cup \text{Contracted Cells}$, thus $G1 \subset G2$. In Version0, if CAMEL processed a symbol that it did not recognize (i.e. symbol $\notin$ `dict`), a `KeyError` was thrown. This is due to the nature of Python `dict` constructors[2].

In order to take G2 input and translate the known patterns, a try-except error catcher was added. The error catcher translated unknown cells as *.

```
function (array of G2-words):
        translated_word = empty string
```

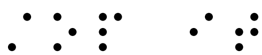
```
for each word:
        for symbols in word:
                if symbol in dict:
                        symbol = dict[symbol]
                else:
                        symbol = *
                translated_word = translated_word + symbol
        translated_sentence[index of word] = translated_word
return translated_sentence
```

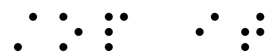
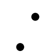
When 'braille.txt' contained the string `001100 101010 111100 000000 010100 011110`, `*op it` was returned. The string `001100` ≡ "st" in G2, but "st" $\notin$ G1, and was consquently represented as an asterisk.

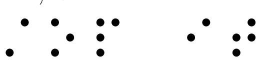
3.3 Matching Partial G2-English Translation to Corresponding English Chunks

We must be able to remove duplicates from sets whilst keeping order. The Python `set` is an unordered collection with no duplicate elements. We need an ordered collection with no duplicate elements, so another method is introduced.
I will feed Version2 "stop it = *op it", and Version must see that *=st, for all other characters are accounted for. I did this by taking the `translated_sentence`.

```
function (translated sentence)
```

When 'braille2.txt' contained the string "001100 101010 111100 000000 010100 011110", evaluated by the 'translate' method as ≡ "*op it", and 'eng2.txt' contained the string "stop it". "st = *" was returned!
Given two inputs
input1 = "stop it"
input2 = 

The program translates input2 using the given G1 dictionary. However, input2 has symbols not in the G1 dictionary, thus these symbols are left untranslated. input2 =  → output1 =  op it
Given that input1 = input2, and input2 = output1 the program compares the overlap between input1 and output1 to find the most likely meaning of the unknown symbol. "stop it" = op it ⇒ ≡ "st"
Summarized, Version2 does the following:

"stop it" =  →  op it ⇒  ≡ "st"

3.4 Storing and Retrieving Unknowns to Improve Partial Translation

This seemed to work wonderfully, but when applying Version1 to the English phrase "penguin has fleas." The partially translated string ≡ "p*gu* has fl*s" was created; this gave "eninea = ***" as the output. This finds the English chunks, but doesn't differentiate between the unknowns.

In order to differentiate the unknowns, the program was altered such that the asterisks were replaced with numbers. Thus, instead of "p*gu* has fl*s", we differentiate the unknowns as "p0gu1 has fl2s." Furthermore, regex was used to extract the flags before and after each digit. So, "0" was preceeded by "p" and followed by "g", when this search was applied to "penguin", the "en" was extracted. Success, and the birth of Version3 (mostly edits to the translate method). Once the unknowns were successfully stored, they were utilized by the program.

This version takes into account the following tags (see Introduction to CAMEL[2.2])
(1) Letters before and after the unknown
(2) Letters after the unknown
(3) Letters before the unknown
(4) Unknown alone

3.5 Matching Separate Occurances of Unknowns to Infer Meaning

This version (Version4) looks at consequent unknowns. For example, the G2 braille form of ”offer is power” is ”off{er} is p{ow}{er}.” Note that {er} is present in two seperate occurances, additionally, {ow} and {er} are consequent.

⠕⠋⠋⠻ ⠊⠎ ⠏⠪⠻ → off⠻ is p⠪⠻

This is taking ”power” = ’p10’, and returning 1 = ”ower”, 0 = ”er”; this error is eliminated by retranslating the input each time a new unknown was found.

3.6 Storing Translation Options in Map[String, TranslationOptions] Format

This required use of the known text to match TranslationOptions to the given English text. For example, the pattern bb is used for both ”bb” and ”be”. Thus, once the computer infers a meaning, the computer will translate other instances of this pattern incorrectly. Using the structure of a Python `dict`, it is simple to add options for keys.

This version takes into account the following tags (see Introduction to CAMEL 2.2)
(1) Letters before and after the unknown
(2) Letters after the unknown
(3) Letters before the unknown
(4) Unknown alone
(5) Known alone (a.k.a. translation options for one-letter contractions)

The first instance of this I dealt with is one word contractions. The output of Version5 was
110111 = er
010101 = ow

Input: offer knowledge is power

Output: offer k is power

One letter contractions exist: k means knowledge in this context.

In Version5, I mainly optimized the system of detecting translated letters vs. untranslated ones.
The context in which ”k” = knowledge is found is The source code used to implement this algorithm is in Source Code[7.6].

3.7 Adding Functionality

In order to include number and capitalized letter functionality, new checks were added to the translate method.

3.8 Optimizing the Program

In order to save time and processor power, a generator was used (instead of an iteratable array) when printing partially translated text.
Using regular expressions (instead of repeatedly searching combinations of the array indices) improved the time and accuracy of English chunk extraction.

4 Results and Conclusions

4.1 Safety of Community

- commercial application in development that will prevent future mislabeling, such as

- allows sighted people to protect the blind community

4.2 Proof of Concept

- 1^{st} succesful automated program that learns compressed Braille
- translation system is effective for arbitrary symbol systems
- language platform easily changed

CAMEL is the 1^{st} succesful automated program that learns compressed Braille. This translation system is effective for arbitrary symbol systems, and the language platform easily changed.

4.3 GUI

For users not familiar not comfortable with the binary representation of Grade 2 Braille, a GUI option was created. This immediately converts from a graphical braille representation into a binary string for CAMEL to parse.

This program could be expanded to an application for sighted-people not familiar with braille. Instead of manually entering Braille with the GUI system, a user could point their phone's camera at the Braille string they wish to translate, and image processing techniques could identify the Braille listed. This Braille would be translated using the dictionary generated by CAMEL.

4.4 Further Research

The method of detection for one letter contractions is comparing the original text, not checking likely matches for the one-letter contractions in a weighted sentence dictionary. CAMEL implements word-level analyzation, and could be improved by using sentence-level analyzation.

The results of this project are the foundation for a usable app (Section[4.3]), for the sighted to decode Braille writing.

5 Documentation for CAMEL

Function: `translate`
Translates known characters, assigns integers to unknown characters.
Parameters:
`brl_array` - array of binary strings, one string per index
Returns:
(Partially-)translated string.

Function: `file2array`
Extracts text from file into parsable format.
Parameters:
`filename` - name of the file from which to extract the strings
Returns:
array of strings, one word per index (whitespace is used to determine word seperation)

Function: `matching`
Recursively infers and stores (updates `dict`) the meanings of unknown symbols.
Parameters:
`partially_translated_text` - array of partially translated words, one word per index
Returns:
Recursive output of `translate` using the updated `dict`

Function: `testing`
Searches for, identifies and translates capitals, numbers, and one-letter contractions.
Parameters:
`new_translation` - newest translation of input
Returns:
None

Function: `weight` (*in progress*)
Weights the translation options from the given symbol's `dict` entry using the context of the sentence.
Parameters:
`dict_symbol` - `dict` entry for given symbol
`brl_array` - array of binary strings, one string per index
Returns:
Most likely translation of a given symbol.

6 Source Code

Version6

```
from collections import Counter as mset

import re

all_powerful_counter = 1 #counts the times the partially translated text is run through the program and
    re-translated

#dictionary of known Grade 1 patterns
dict = {'100000' : 'a', '110000': 'b', '100100' : 'c', '100110' : 'd', '100010' : 'e', '110100':'f', '
    110110' : 'g', '110010' : 'h', '010100' : 'i', '010110' : 'j', '101000': 'k', '111000' : 'l', '
    100101': 'm', '101110': 'n', '101010': 'o', '111100': 'p', '111110' : 'q', '111010': 'r', '011100' :
     's', '011110': 't', '101001' : 'u', '111001': 'v', '010111' : 'w', '101101': 'x', '101111' : 'y', '
    101011' : 'z', '000000' : ' ', '000001' : '*', '001111' = '#'}
```

```
dictnum = ['010110','100000','110000', '100100', '100110', '100010', '110100', '110110', '110010', '
    010100'] #index of string corresponds to number

filename = 'braille5.txt' #filename of G2 Braille
filename0 = 'eng5.txt' #filename of equivalent English

def execute():
        testing(matching(translate(file2array(filename)))) #Finds meaning of braille2

def file2array(filename):
        with open(filename, 'r') as f:
                array = [word.strip() for word in f] #extracts strings from file
                array = [word.strip() for word in str(array[0]).split(' ')]
        return array

dictU={}
def translate(brl_array, count=0):
        t=[]; duplicates = []
        count = count
        for key in brl_array:
                if dict.get(key) == None and duplicates.count(key) == 0:
                        duplicates.append(key)
                        t.append(dict.get(key, str(count))) #Substitutes unknown pattern for integer to
                            hold its place
                        dictU[str(count)] = key #Appends substituted char and corresponding keyvalue, ex
                            : '0':'010010' for future lookup

                        count = count+1
                elif dict.get(key) == None and duplicates.count(key) != 0:
                        t.append(dict.get(key, str(duplicates.index(key))))
                else:
                        t.append(dict[key])
        x = ''.join(t)
        x = [word.strip() for word in x.split(' ')] #a list of the partially translated words
        #Note that the indexes of x and e show the relationship between the PT and eng words.
        return x #partially translated text

e = file2array(filename0) #a list of the english words

def testing(new_translation):
        words = matching(new_translation) #Print translation
        for word in words:
                if re.search('^\#', word) != None: #checks for number indicator
                        for letter in range(len(word)-1):
                                nums.append(dict_num.index(word))
                        nums = ''.join(nums)
                        words[words.index(word)] = nums

                elif re.search('^\*', word) != None: #Checks for Caps Symbols
                        if word[0] == "*" and word[1] != '*':
                                        new = word[1:]
                                        new = new.capitalize() #Capitalize First Letter
                        else:
                                        new = word[2:]
                                        new = new.upper() #Capitalize entire word
                        words[words.index(word)] = new

        print "\nGoal: %s \nTranslation: %s\n"%(' '.join(e),' '.join(words))
        result = words
        for x in xrange(len(e)):
                if result[x] != e[x]:  #Check that translated and original words are the same
                        print "\nOne letter contractions exist:\n%s means %s in this context." %(result[
                            x],e[x])
                else: pass

def matching(partially_translated_text):
        duplicates1 = []; index_of_word = 0
        z = partially_translated_text
        counter = 0 #Used to determine which cell is being evaluated within word, if word contains
            multiple unknown cells
```

```
        for word in z:
                for m in re.finditer("[a-zA-Z\s]*?\d\w*", word): #Finds words with unknown symbols
                        if re.search('[a-z]*', m.group()) != None: #If word has no characters besides
                            the unknown symbol, ignore.
                                unknown = (re.search('\d+', m.group())).group() #find unknowns in word
                                for n in re.finditer('\d{1}', m.group()): #iterate through unknowns
                                        if len(unknown) != 1: break
                                        else:
                                                #for c in re.finditer('[a-z]+', m.group()): #Note to
                                                    self: Use of [a-z]+ ==> 2er3hg ==> er, hg
                                                        #h = e[index_of_word].replace(c.group(), '')
#\w*?(?=\d) ==> abc1nn ===> abc
                                                tag = (re.search('\w*(?=\d)', m.group())).group() #find
                                                    unknowns in word
                                                h = e[index_of_word].replace(tag, '') #look at
                                                    corresponding english word, remove tagged char from
                                                    word, in this case, characters preceeding the
                                                    unknown
                                                l = m.group().replace(tag, '')
                                                if re.search('[a-z]+', l) != None: #If there are
                                                    translated letters after the unknown, clear the
                                                    translated letters.
                                                        next=(re.search('[a-z]+', l)).group()
                                                        h = h.replace(next, '')
                                                else: pass #set unknown equal to english match
                                                dict[dictU[n.group()]] = h
                                                print dictU[n.group()] + ' = ' + h #Prints learned
                                                    symbols

                        counter = counter+1
                index_of_word = index_of_word +1
        version = all_powerful_counter + 1
        return translate(file2array(filename),version)


execute()
```

A Braille Alphabets

⠁	a 1
⠃	b 2
⠉	c 3
⠙	d 4
⠑	e 5
⠋	f 6
⠛	g 7
⠓	h 8
⠊	i 9
⠚	j 0
⠅	k
⠇	l
⠍	m
⠝	n
⠕	o

⠏	p
⠟	q
⠗	r
⠎	s
⠞	t
⠥	u
⠧	v
⠺	w
⠭	x
⠽	y
⠵	z
⠯	{and}
⠜	{ar}
⠼	{ble}
⠡	{ch}

⠫	{ed}
⠢	{en}
⠻	{er}
⠿	{for}
⠣	{gh}
⠔	{in}
⠬	{ing}
⠔⠖	{into}
⠷	{of}
⠳	{ou}
⠪	{ow}
⠩	{sh}
⠹	{th}
⠮	{the}
⠱	{wh}
⠾	{with}

⠂	, {ea}
⠆	; {bb} {be}
⠒	: {cc} {con}
⠲	. $ {dd} {dis}
⠖	! {ff} {to}
⠶	( ) {gg} {were}
⠦	{‘‘} ? {his}
⠔⠔	*
⠴	{’’} {by} {was}
⠄	’
⠤	- {com}
⠌	/ {st}
⠠⠶	[
⠶⠄	]
⠠⠦	{.‘}
⠴⠄	{’.}
⠒⠏	{percent}

B Prefix Indicator

⠠	{Capital}
⠠⠠	{Upper}
⠨	{Italic}

⠼	{Number}
⠰	{Letter}

C Contraction for Part of Word

Anywhere	
{and}	and
{ar}	ar
{ch}	ch
{ed}	ed
{en}	en
{er}	er
{for}	for
{gh}	gh
{in}	in

Anywhere	
{of}	of
{ou}	ou
{ow}	ow
{sh}	sh
{st}	st
{th}	th
{the}	the
{wh}	wh
{with}	with

Beginning	
{be}	be
{com}	com
{con}	con
{dis}	dis

Middle	
{bb}	bb
{ble}	ble
{cc}	cc
{dd}	dd
{ea}	ea
{ff}	ff
{gg}	gg
{ing}	ing

End	
{ble}	ble
{ing}	ing

D Final Letter Contraction for Middle or End of Word

Prefix ⠨	
d	ound
e	ance
n	sion
s	less
t	ount

Prefix ⠰	
e	ence
g	ong
l	ful
n	tion
s	ness
t	ment
y	ity

Prefix ⠠	
n	ation
y	ally

E Initial Letter Contraction for Whole or Part of Word

Prefix ⠘	
{the}	these
{th}	those
u	upon
{wh}	whose
w	word

Prefix ⠸	
c	cannot
h	had
m	many
s	spirit
{the}	their
w	world

Prefix ⠐	
{ch}	character
d	day
e	ever
f	father
h	here
k	know
l	lord
m	mother
n	name
o	one
{ou}	ought

Prefix ⠐	
p	part
q	question
r	right
s	some
{the}	there
{th}	through
t	time
u	under
{wh}	where
w	work
y	young

F Abbreviation for Whole Word

ab	about	c	can	xs	its	rjcg	rejoicing
abv	above	{ch}	child	xf	itself	sd	said
ac	according	{ch}n	children	j	just	{sh}	shall
acr	across	{con}cv	conceive	k	knowledge	{sh}d	should
af	after	{con}cvg	conceiving	lr	letter	s	so
afn	afternoon	cd	could	l	like	{st}	still
afw	afterward	dcv	deceive	ll	little	s{ch}	such
ag	again	dcvg	deceiving	m	more	t	that
ag{st}	against	dcl	declare	m{ch}	much	{the}	the
alm	almost	dclg	declaring	m{st}	must	{the}mvs	themselves
alr	already	d	do	myf	myself	{th}	this
al	also	ei	either	nec	necessary	{th}yf	thyself
al{th}	although	{en}	enough	nei	neither	{to}	to
alt	altogether	e	every	n	not	td	today
alw	always	f{st}	first	o'c	o'clock	tgr	together
{and}	and	{for}	for	{of}	of	tm	tomorrow
z	as	fr	friend	{one}f	oneself	tn	tonight
{be}	be	f	from	{ou}rvs	ourselves	u	us
{be}c	because	g	go	{ou}	out	v	very
{be}f	before	gd	good	pd	paid	{was}	was
{be}h	behind	grt	great	p	people	{were}	were
{be}l	below	h	have	p{er}cv	perceive	{wh}	which
{be}n	beneath	h{er}f	herself	p{er}cvg	perceiving	w	will
{be}s	beside	hm	him	p{er}h	perhaps	{with}	with
{be}t	between	hmf	himself	qk	quick	wd	would
{be}y	beyond	{his}	his	q	quite	y	you
bl	blind	imm	immediate	r	rather	yr	your
brl	braille	{in}	in	rcv	receive	yrf	yourself
b	but	{into}	into	rcvg	receiving	yrvs	yourselves
{by}	by	x	it	rjc	rejoice		

G Bibliography

[1] http://texdoc.net/texmf-dist/doc/latex/braille/summary.pdf
[2] http://docs.python.org/2/library/stdtypes.html#dict