

ENGLISH-PIG LATIN TRANSLATOR

Jennifer Baldwin
LING 360 – Perl Programming
Spring 2001

1 Introduction and Organization

As children (and sometimes as adults), native speakers of English often play a word game called Pig Latin. The rules of the game tend to vary slightly by region, but the general concept remains the same.

If an English word begins with a consonant, the initial consonant or consonant cluster moves to the end of the word, and the letters *-ay* are appended after the moved letter(s). If a word begins with a vowel, no letters move, and the suffix *-hay* is added to the word-final position. See examples (1) and (2) below. Note that existing dialect differences have to do with whether the suffix for vowel-initial words is *-hay*, *-way*, *-yay*, or simply *-ay*. I have chosen to use the *-hay* suffix, as it seems to be the most common.

- (1) English word: *spring*
The word begins with a consonant cluster, so we move it to the end of the word. *ingspr*
Next, we add the *-ay* suffix to derive the final Pig Latin form of the word. *ingspray*
Pig Latin word: *ingspray*
- (2) English word: *apple*
The word begins with a vowel, so do nothing yet with the word. *apple*
Next, we add the *-hay* suffix to derive the final Pig Latin form of the word. *applehay*
Pig Latin word: *applehay*

Given the nature of language, it is not at all surprising, that converting a word from English to Pig Latin may sometimes be tricky. For example, the vowel *u* typically follows the consonant *q*, and when it does, both the vowel and the consonant must move to the end of the word, as shown in (3). When the *q* appears without a *u*, only the *q* moves, as shown in (4).

- (3) English word: *quick* Pig Latin word: *ickquay*
(4) English word: *Qatar* Pig Latin word: *Atarqay* (See Section 2.4 for capitalization rules.)

An English-Pig Latin translator must account for situations like the one described above. Using Perl, I have written a program called **piglatin.pl** that translates English text from a web page, a file or direct input to Pig Latin based upon logical rules and restrictions. The program is of use for anyone, especially children, interested in communicating through a fun and elementary level of encryption. A natural counterpart to the program would be a Pig-Latin English translator, which is beyond the scope of this class project.

Section 2 presents the layout of the program with descriptions of the key components: the main code and the four subroutines. Section 3 focuses on certain exclusions made for untranslatable expressions, and Section 4 outlines various enhancements that cover special cases. Finally, in Section 5, I summarize and conclude my discussion of **piglatin.pl**, offering suggestions for future improvements to the program.

2 Description of Key Components

The two functions of the main code within the **piglatin.pl** program are accepting input and printing output. To process the input, the main code calls the **translate** subroutine, which in turn calls the remaining three subroutines **pigLatin**, **caps**, and **punctuate**.

2.1 Main Code

The program `piglatin.pl` accepts input in three formats: a URL, a file name, and direct input.

The user may input a URL – with or without the leading `http://` – as a command-line argument, as shown in (5), or one may input a file name, as shown in (6).

```
(5) piglatin.pl http://www.yahoo.com or piglatin.pl www.yahoo.com
(6) piglatin.pl NYT.txt
```

If a command-line argument exists, `piglatin.pl` checks its format against a URL regular expression. Of course, a match indicates a URL, so `piglatin.pl` uses `lynx` to write the contents of the website to a variable called `$input`. It then performs a `split` on the variable, assigning each line of data to an array named `@input`. A `foreach` loop sends the data from `@input` to the `translate` subroutine line by line and outputs each line (as standard output) as it returns from the subroutine.

On the other hand, if a command-line argument exists but the URL regular expression does not find a match, the argument must be a file name. The contents of the file, if it truly exists, write to standard input. Similarly, when the program is run without an argument, direct input writes to standard input. The program therefore treats both uses of standard input equally, calling the `translate` subroutine and printing the output within a simple `while(<>)` loop.

2.2 The translate Subroutine

Nearly half of the lines of code within `piglatin.pl` belong to the `translate` subroutine. Its function is to process a line of English text and return a line of Pig Latin text. To do this, it splits the input line of English text by white space, writing the resulting words to an array called `@Words`. Data in `@Words` then enters a `foreach` loop, where each word (as the variable `$W`) is immediately validated against several untranslatable patterns. See Section 3 for details on untranslatables. If the word is determined to be untranslatable, it passes straight through to the output variable `$pigLine` in its original English form. Translatable words continue through the `foreach` loop.

The second validation point checks the word for a number enclosed in brackets in the initial position of the word. A matching pattern is a linked word from a web page, and such words must be translated separate from their link number. The word itself is stored back into the variable `$W`, and the link number is stored in the variable `$linkNum` so that it can output with the Pig Latin form of the word.

```
(7) Input: [12]Photos
      Separate the link number from the word.      [12] → $linkNum Photos → $W
      Translate the value of $W to Pig Latin.      Otosphay
      Return the link number to the front of the word. [12]Otosphay
      Output: [12]Otosphay
```

The third and fourth validation points check for punctuation at the beginning and end of the word, respectively. If punctuation exists, it is saved to either the `$initialPunc` or `$finalPunc` variable, and the rest of the word is saved back into `$W`. Like link numbers, punctuation is restored to the translated word before it is returned as output, as illustrated in (8).

```
(8) Input: "Great!"
      Separate the initial punctuation from the word. " → $initialPunc Great!" → $W
      Separate the final punctuation from the word.  !" → $finalPunc Great → $W
      Translate the value of $W to Pig Latin.        Eatgray
      Return the punctuation to the word.           "Eatgray!"
      Output: "Eatgray!"
```

The fifth validation point looks for words with the format numbers-letters, as in *6-person*. In the style of the previous few lines of code, the number and hyphen are removed and stored as `$initialNum`, and the letters of the word are saved in `$W`. `$initialNum` and the translated word are combined into the output.

(9) Input: `6-person`
Separate the number and hyphen from the word. `6- → $initialNum person → $W`
Translate the value of `$W` to Pig Latin. `ersonpay`
Return the punctuation to the word. `6-ersonpay`
Output: `6-ersonpay`

The sixth `if` statement in the `foreach` loop checks if the value in `$W` is a number (containing no letters at all). If the match returns true, the original word must have been a punctuated number. Recall that unpunctuated numbers pass through as untranslatables at the start of the `foreach` loop. A punctuated number is also untranslatable, but it must first be sent through to `punctuate` subroutine (explained in Section 2.5) to regain its punctuation before output. The returned value appends to the variable `$pigLine` to be returned from this `translate` subroutine to the main code.

If the `if` statement described in the previous paragraph returns false, the next step is an `elsif` statement that checks if `$W` contains a letter. If it does, `translate` passes `$W` to the `pigLatin` subroutine (explained in Section 2.3) which returns to a variable called `$pigWord` the Pig Latin counterpart of the clean English word in `$W`. Next, it passes both the Pig Latin word `$pigWord` and the English word `$W` to the `caps` subroutine (explained in Section 2.4) which restores original capitalization to the Pig Latin word. Then, it passes `$pigWord`, along with the variables `$linkNum`, `$initialNum`, `$initialPunc`, and `$finalPunc` to the `punctuate` subroutine (explained in Section 2.5) which restores the word's original punctuation and number prefixes. Finally, it appends `$pigWord` to `$pigLine` to be returned from the `translate` subroutine.

In the case that the `if` and `elsif` statements both return false, the value in `$W` must be an untranslatable string of non-alphanumeric characters. Like the other untranslatables encountered, `$W` is sent directly to `$pigLine` for output.

At this point, both `$linkNum` and `$initialNum` are undefined so as not to cause problems in future iterations of the `foreach` loop. The variables `$initialPunc` and `$finalPunc` do not need to be undefined because they receive a fresh value with each pass through the `foreach` loop.

Finally, the `translate` subroutine returns the value in `$pigLine` to the main code.

2.3 The pigLatin Subroutine

The `pigLatin` subroutine is responsible for performing the function at the heart of the `piglatin.pl` program. Called from within the `translate` subroutine, it accepts a clean English word as input and returns a clean Pig Latin word as output.

A series of `if` and `elsif` statements use regular expressions to check for patterns within the English word input. The first `if` statement matches words that begin with the letters *qu*. When it finds a match, it reorders the letters of the word, moving the letters *qu* to the final position before appending the letters *-ay*. The resulting Pig Latin word is then assigned to the variable `$pigWord`.

When there is no match for *qu*-initial words, the subroutine attempts to match words beginning with the letter *y* followed by a vowel. In such cases, *y* behaves as a consonant, and the word is converted to Pig Latin by moving the consonant *y* to the end of the word and adding *-ay*. Again, the new word writes to the variable `$pigWord` for output.

If there still has not been a matching pattern found in the input `$englishWord`, the next `elsif` statement is performed. It matches words that begin with one of the vowels *a*, *e*, *i*, *o*, or *u* or the letter *y*. Because we've already

handled the consonant *y*, the *y* in untreated *y*-initial words must behave as a vowel. Having determined the word as vowel-initial, the subroutine follows the steps as stated above in (2) to translate the word to Pig Latin by simply adding *-hay* to the end of the word. While testing the program, however, I found it bizarre to see a double-*h* pattern in Pig Latin words such as *Englishhay*. Pig Latin has no formal rule for this situation, so I took the liberty to write a line embedded within this **elsif** statement to add only *-ay* instead of *-hay* for vowel-initial words ending with the letter *h*. The result is stored in the variable **\$pigWord**.

Of course, the final case within the **pigLatin** subroutine deals with English words beginning with a consonant (excluding *qu* or consonant *y* which have already been treated). Here, the subroutine follows the steps stated above in (1), shifting the initial consonant or consonant cluster to the end of the word and adding the letters *-ay* and writing the result to the variable **\$pigWord** for output

Lastly, the value of **\$pigWord** returns from the **pigLatin** subroutine to the **translate** subroutine from which it was called.

2.4 The caps Subroutine

Before any Pig Latin word can be displayed for the user as output of **piglatin.pl**, it must contain its original capitalization. The **caps** subroutine restores proper capitalization to a translated Pig Latin word immediately after the **pigLatin** subroutine has completed. Also called from within the **translate** subroutine, it accepts both a raw English word and its translated Pig Latin word as input. If at least one letter in the raw word is capitalized, the subroutine returns a capitalized Pig Latin word as output. Otherwise, it returns the same Pig Latin word that it received as input, unchanged.

The first **if** statement within the **caps** subroutine checks for raw words that consist entirely of capital letters. Such words must return in all caps, so I have used the **uc** command to set the entire matching word to caps and write the result to the variable **\$cappedWord**. The regular expression in this **if** statement contains an optional special case for last names that begin with **Mc**, **Mac**, or **O'**. In these last names, raw data for all-caps words may contain some lower-case letters, as found in the **NYT.txt** test file. The smoothest method for dealing with such data is to capitalize all letters of the name in the Pig Latin output.

Most capitalization in the raw data appears at the beginning of sentences, where only the first letter of a word is capitalized. An **elsif** statement checks a raw word for a capital letter, and when it finds a match, it uses the **ucfirst** command to capitalize the first letter of the Pig Latin word. The resulting word is assigned to the variable **\$cappedWord**.

Remaining unmatched words must not contain any capitalization in the raw data, so the existing Pig Latin version of the word is simply written to **\$cappedWord** for output.

The value of **\$cappedWord** returns from the **caps** subroutine to the **translate** subroutine from which it was called.

2.5 The punctuate Subroutine

The last subroutine in **piglatin.pl** is the **punctuate** subroutine. Its purpose is to restore any characters that were removed within the **translate** subroutine in order for the English word to be converted to Pig Latin. The **translate** subroutine sends the translated word along with its original extra characters, if any. These characters are word-initial link numbers (**\$linkNum**), word-initial numbers (**\$initialNum**), word-initial punctuation (**\$initialPunc**), and word-final punctuation (**\$finalPunc**). Output of the **punctuate** subroutine is a Pig Latin word with all of the extra characters from the corresponding original word.

There are four **if** statements within the **punctuate** subroutine. Each one checks for the existence of one of the four extra character variables listed in the previous paragraph. If the variable contains a value, its value is concatenated onto the Pig Latin word in **\$puncWord** for output.

Finally, the **punctuate** subroutine returns the contents of the variable **\$puncWord**.

3 Exclusions for Untranslatable Expressions

In the absence of a formal guide for translating expressions from English to Pig Latin, I depended on my own judgment to determine which expressions must be excluded. The untranslatables I encountered are:

- | | | |
|--------|--|---------------------------------|
| (10)a. | Words consisting only of numerals (with and without punctuation) | ex. <i>56</i> or <i>"100"</i> |
| b. | Ordinal numbers | ex. <i>10th</i> |
| c. | Word-initial numbers with a hyphen | ex. <i>14-day</i> |
| d. | Link numbers for web pages | ex. <i>[5]Click</i> |
| e. | URLs | ex. <i>http://www.yahoo.com</i> |
| f. | Titles | ex. <i>Mr.</i> or <i>Jr.</i> |
| g. | Initials (only with periods) | ex. <i>U.S.A.</i> |
| h. | Word strings consisting only of underscores | ex. <u> </u> |
| i. | Nonsense strings of special characters | ex. <i>?\$&!@!</i> |

(10a), (10c), (10d), (10h), and (10i) are untranslatable simply because they do not contain any letters, a natural requirement of Pig Latin. The string of underscores (10h) had to be singled out because Perl treats an underscore as a word character. (10b) contains letters, but they are so much a part of the number to which they are attached that they become completely unacceptable if converted to Pig Latin. URLs (10e) only have meaning in their original form, so there is no point in translating them, and (10f) and (10g) become nearly incomprehensible in Pig Latin.

Understandably, untranslatables merely pass from input to output without being transformed in any way.

4 Enhancements for Special Cases

Not surprisingly, the rules presented above as (1) and (2) do not account for every translatable English word. For this reason, I included a number of enhancements within **piglatin.pl** in order to correctly handle as many words as possible. The following word types are accounted for:

- | | | |
|--------|--|--------------------|
| (11)a. | Standard consonant-initial words | ex. <i>dog</i> |
| b. | Standard vowel-initial words (vowel = <i>a, e, i, o, u</i>) | ex. <i>outside</i> |
| c. | Vowel-initial words ending with <i>h</i> | ex. <i>English</i> |
| d. | Consonant <i>y</i> -initial words | ex. <i>yellow</i> |
| e. | Vowel <i>y</i> -initial words | ex. <i>yclept</i> |
| f. | Vowel <i>y</i> -internal words | ex. <i>style</i> |
| g. | <i>qu</i> -initial words | ex. <i>queen</i> |

Unlike the untranslatables in (10), the special cases in (11) consist strictly of alphabetic characters. Cases (11d), (11e), (11f), and (11g) are special because they include, typically at the beginning of the word, a letter with unusual behavior. Depending on adjacent letters, the letter *y* may behave as a vowel or a consonant. At the beginning of a word, *y* is a vowel when the second letter is a consonant, and it is a consonant when the second letter is a vowel. Inside a word, it functions as a vowel when embedded between consonants. Case (11g) is special because, in Pig Latin, the letter *q* moves with the following vowel if and only if that vowel is the letter *u*. Otherwise, *q* moves alone. Case (11c) is special merely out of a personal preference to avoid a double-*h* pattern when *-hay* is appended at the word-final position.

All special cases are processed within the **pigLatin** subroutine. The most restrictive cases, that is all cases but (11a) and (11b), are handled first. Remaining words continue through the subroutine to the default consonant-initial and vowel-initial lines of code at the end of the subroutine.

5 Conclusion

As far as translation software is concerned, **piglatin.pl** is incredibly simple. There is no need to understand any meaning within the program's English language input to determine the correct Pig Latin output, as Pig Latin is derived entirely from the letters of an English word.

However, in writing an English-Pig Latin translator, I encountered several interesting challenges. At first, I thought that the program would be easy to write in little time. After all, Pig Latin is so simple that we experiment with it as children. When we use it, however, we are typically speaking, not writing, so the tricky situations that I discussed in Section 4 do not surface. Also, spoken Pig Latin treats the expressions I labeled as untranslatable in Section 3 differently than does written Pig Latin. Numbers and personal titles are fully pronounced and therefore translated as regular words, and nonsense strings such as (10h) and (10i) are never spoken. In addition, I dedicated entire subroutines to restoring capitalization and punctuation; both are invalid in spoken Pig Latin.

Through much testing and fine-tuning, I believe that **piglatin.pl** is a very accurate English-Pig Latin translator built from neatly organized Perl code, but there is always room for improvement. An admitted shortcoming within the existing version is an overly restrictive regular expression for matching URL patterns. Ideally, a future version of the program will improve upon the existing URL regular expression and possibly even function over the Internet, accepting English input from a form on a web page and printing Pig Latin output to another web page, even retaining HTML code when the input is a URL. It could also allow users to select the suffix (i.e. *-hay*, *-way*, *-yay*, or *-ay*) they prefer for vowel-initial words.

Naturally, a similar program can be written to complement this one, translating Pig Latin text to English. The **caps** and **punctuate** subroutines can be used exactly as they appear in **piglatin.pl**. In lieu of the **pigLatin** subroutine, it would require a subroutine that converts Pig Latin strings to English, and the **translate** subroutine would need to be modified to call that subroutine where it now calls the **pigLatin** subroutine. The main code, which simply accepts input and prints output, would need no functional changes, but it would be helpful to use a variable called **\$english** instead of **\$pigLatin** to better reflect its value.