My project consists of an investigation into the combination of computer science and linguistics to produce computer-generated haikus on any given topic. To this end, the internet is extensively utilized as an absolutely massive data structure of the English language, in addition to publicly available internet search capabilities. The haiku poetic form offers a medium with strict rules of creation, but sufficiently lax rules of grammar as to allow for a greater range of generated phrases. Four distinct stages are involved in the program’s execution – the collection of relevant words, the part-of-speech labeling of words, the combination of words into short phrases and subsequent syllable determination, and the final selection of these short phrases into a completed haiku. The conclusion of the project, while hard to quantitatively analyze, has produced well-formed haikus narrowly tailored to the seeded topic.

For the purposes of the project, a haiku is defined as any poem of three lines of five, seven, and five syllables respectively. Only English was considered as a language for the poems. This particular form of poetry was chosen for two distinct reasons – strict rules of form, and lax rules of content. The former motivation ensures a framework tight enough to keep the poems accountable. On the other hand, an extensive range in the quality of computer-generated English grammar was found throughout the course of the project, and the forgiving nature of the rules of haiku content allows for a great number of acceptable poems.

The four specific stages of the program’s execution will now be discussed. The first of these stages is the collection of words relevant to the user-supplied topic. The general idea here was to first retrieve the list of webpages returned from an internet engine search on the topic, and then mine every word from those pages. The most frequently occurring of these words would then be considered as a collection of relevant words. Initially, a Google web service was employed that nicely packaged and returned this list of webpages. Unfortunately, the reliability of the service was poor enough that I
was forced to explore other options. Ultimately, the use of Google was scrapped entirely, and the program now relies solely on MSN Search. Without a web service, the list of pages is retrieved by first fetching the HTML of a completed MSN Search by manipulating query-string arguments in the URL. Written perl filters then extract the returned links, carefully avoiding the many ads and promotional links on the page. This list of URLs is visited one at a time. Separate perl filters then strip the page of HTML tags, scripts, meta data, comments, etc. Finally, the remaining words are considered – for each word, a hash value associated with that word is incremented by one. At this point, after some simple sorting, the program has obtained an array of words ordered in a decreasing fashion according to their frequency in the viewed pages.

Once the first stage is completed, the process of determining the part of speech of each word begins. To accomplish this, the webpage www.dictionary.com is extensively used. Again, no developer-friendly web service exists, so the proper dictionary page concerning a specific word is retrieved by the manipulation of URL query strings. From this page, the extraction of the part of speech is relatively trivial. A separate array associated with the array of frequently appearing words then stores these corresponding parts of speech.

The third stage of the process involves the combination of collected words into short phrases of stronger grammar. To accomplish this, the program returns to internet search engines – however, this time only the number of returned hits is considered. Behind this stage is the assumption that the quality of English of any given phrase and the number of hits returned upon a search of that phrase are directly proportional. Here, “quality of English” refers both to correct grammar as well as semantic meaning. In reference to grammar, a phrase like “he go store” will score much lower than “he went to the store.” Likewise, English semantics are involved, as “red ignorance” will score much lower than “red apple,” even though both are grammatically perfect. Therefore, the program is seeking out combinations of words (that it has already deemed relevant to the topic) that score as highly as possible on MSN Search.

To help the program begin selecting words that will create high scoring phrases, the previously determined parts of speech of the words comes into play. While the English language has very few hard and fast rules, some juxtapositions of parts of speech
can be expected to make more or less sense – a noun followed by a verb will quite often make grammatical and semantic sense, while an adjective followed by an adverb will rarely do the same. Therefore, the process of combining words is broken down into 5 categories of one part of speech followed by another – noun/verb, verb/noun, adjective/noun, adverb/verb, random/random. In each category, a group of words is selected from one part of speech, and each is combined with every word from a group selected from the other part of speech. Finally, the idea is to search upon the resultant two word phrase, and record the number of returned hits associated with that phrase. However, the number of required internet searches rises in a squared fashion as compared to the number of words chosen from each part of speech – very quickly a communication bottleneck emerges, and the program takes far too long to run.

To combat this bottleneck and reduce running time, the program forks off a child for each of the five categories of parts of speech combination. As the bottleneck arose from communication and not from computation, sending many requests to MSN Search in parallel with the exact same processing power mitigates the problem. Each child gathers its respective information, and then sends that information down a one-way pipe to the parent. After talking to its parent, each child exits, leaving only the original parent process running with all the information it sought. At this point, after some simple sorting, the program has obtained a list of short phrases, ordered in decreasing fashion according to MSN Search score.

For each child to retrieve the information it seeks, one additional wrinkle needed ironing out. MSN Search will not continue to serve search pages to a single source asking for many in a short amount of time – presumably to combat denial of service attacks and share resources equally to all interested parties. Unfortunately, the haiku program at this point consists of five children all simultaneously asking for tens if not hundreds of search pages – very quickly they stop receiving responses back from MSN. To ensure the information is correctly gathered, small periods of down-time are built into the program’s structure. If at any time a child does not receive a well-formed response from MSN, it waits a fraction of a second before trying the request again. Subsequent denials are met with progressively longer down times. The length of these pauses has no
real upper limit, but in practice never exceeds a single second. The result of this back-off method is the correct collecting of all MSN search pages.

The third stage of the program is completed upon determining one more piece of information – the correct number of syllables per produced short phrase. Here, a local function was developed that took as input any English phrase, and returned the number of syllables. A general overview of the algorithm is as follows –

```plaintext
Set inVowelGroup to 0
Set numSyllables to 0
for each letter in the phrase
    if the letter is a vowel and inVowelGroup is 0
        increment numSyllables
        set inVowelGroup to 1
    else (the letter is a consonant)
        set inVowelGroup to 0
return numSyllables
```

Unfortunately, the algorithm is not perfect. One source of trouble is the very common silent *e* in English. For instance, the word *baseball* is interpreted by the algorithm as *bas-e-ball*, and the word is assigned three syllables. To combat this, one syllable is subtracted from any word that the algorithm determines contains a silent *e*. At this point, the algorithm is capable of correctly determining the correct number of syllables for the vast majority of English words and phrases. With all the information is has obtained thus far, the program is now ready to create haiku poems.

The last stage of the program consists of selecting a number of the combined short phrases in order to complete a correct haiku. Here, only the number of syllables per phrase and the score of each phrase is considered. Each of the three lines of the haiku is constructed separately, but in similar fashion. First, a single phrase is chosen at random, with higher scoring phrases weighted more heavily, and therefore more likely to be chosen. This weighted choosing continues in order to find another phrase with a sufficiently low number of syllables to not exceed the number allowed for the particular line of the haiku. This process continues until that specific number of syllables (either five or seven) is reached, and the line is finished. When all three lines have been created in this fashion, the haiku is completed.
Possible future work on the project abounds. In particular, a more sophisticated method for combining words and examining the number of returned hits from MSN Search could increase both grammatical correctness and the quality of semantic meaning. The current program combines single words and eventually obtains a list of the highest scoring pairs – another approach would be to consider three or more words at a time. As the number of words increases, the number of possible orderings rises dramatically. Therefore, while ever higher scoring phrases would be discovered, communication and potentially computation bottlenecks would quickly arise. With this in mind, such algorithms would have to be carefully constructed.

Furthermore, I believe work in parallel processing could greatly increase the quality of the produced poems. Several parameters exist that define the operation of the program. For instance, the number of web pages initially mined for words relevant to the seeded topic greatly influences the quality of the end result. Likewise, the number of words chosen from a particular part of speech while creating the short phrases has an enormous impact on the final poem. As each word of a particular part of speech is combined with every word from the other part of speech, increasing the number of words chosen has a dramatic influence on the overall number of phrases considered. When combining nouns and verbs, 125 more phrases are considered when 15 words of each are chosen instead of 10. Of course, while this greatly enhances the performance of the program, running time quickly becomes an issue. With this in mind, running the program in a parallel processing framework becomes an attractive proposition – distributing the MSN Search queries over multiple processors would allow for a large increase in the number of phrases scored.

While it is difficult to quantitatively analyze the haikus produced by the program, they can be subjectively observed. The resulting poems are always related to the seeded topic, however not always with the intended connotation. While the grammar of the poems is rarely perfect, it is often passable, and never terrible enough as to make any

---

1 One might give the topic “tiger,” referring to the animal – however “tiger” is also a version of the Macintosh operating system, the first name of a famous golfer, and involved in countless brandings such as “Tiger Mart” or “Tiger Records.” The particular connotation of the word is chosen entirely by MSN Search.
sense of meaning impossible. The poems do not yet capture the power contained in the haikus written by a true poet – more often they are amusing than moving. I do not find the idea of emotionally charged computer-generated poems impossible, however. With more time to develop the algorithms contained in the program, and more computing power to crunch the words, I certainly believe one day such powerful haikus could be created.
Appendix A: Sample topics and generated haikus

“football”
Football said the draft
One draft their fantasy
Now there out team

“haiku”
How still our haiku
Form has the Japanese form
Now such poet use

“software”
Try well buy software
Computer offer quickly
Will get network buy

“wine”
The wine find best free
Wine tasting pinot tasting
Wine get the new best

“new york”
City in the State
Programs travel all their state
Health find city travel

“dinosaur”
Fossil found park find
National park dinosaur
New long such the find

“yellow”
Yellow search web design
Yellow help their business plan
Pages search the web find
“train”
Train travel rail visit
Train announced train passes
One simulator

“kitchen”
Chicken add food cook
Faucet and chicken cooking
Kitchen and their stove

“eagle”
Eagle help our sport
Sport scout national eagle
Eagle see how good

“happy”
Comment know book now
April birthday like happy
Best way birthday help

“apple”
Mac inc using mac
Mac inc computer using
Apple says mac show

“human”
Prompt report been said
Friendly not genome review
Event say been said

“mouse”
System and search click
Click explain computer use
System and our click

“book”
One book their review
Codes and building book review
Book find codes new use
“toyota”
Their part make part find
Quality find quality
The part supplier

“flag”
Flag call custom see
Banner see he have item
American banner see

“heart and soul”
Heart see the heart like
People know people design
Work like soul know

“gilbert and sullivan”
Music opera
Gilbert show opera show
Music like know there
#!/local/usr/bin/perl

use SOAP::Lite;
use LWP::Simple;

$usePoS = 1;
print "Begin Haiku writer.\n";

if($#ARGV < 0){
    die "Usage: perl wordassoc.pl [topic]\n";
}

my $topic = "\\" . $ARGV[0];
for($i=1; $i<=$#ARGV; $i++){
    $topic .= " ";
    $topic .= $ARGV[$i];
}
$topic .= "\\";

WordAssociation($topic, 1, 0);

#FUNCTION: WordAssociation
#----------------------------------------
#INPUTS
#  topic          the topic to build word associations on.
#  numSearches    the number of 10 result searches to make.
#  followLinks    boolean to follow returned URLS
sub WordAssociation {
    my $topic = $_[0];
    my $numSearches = $_[1];
    my $followLinks = $_[2];

    %googleAllHash = ();
    %googleProximityHash = ();
    %msnHash = ();
    my $googleResults;
    my $result;

    #HACKHACKHACK
    $numSearches = 4;

    print "Topic is: $topic\n";

    my @MSNResults = ();
    for($currentSearch = 0; $currentSearch < $numSearches; $currentSearch++){
        @MSNResults = ();
        @MSNResults = MSNSearch($topic, $currentSearch*10);
        #Extract
        for($i=0; $i<10; $i++){
            Extract($topic, get($MSNResults[$i]));
        }
    }

    Order();
}
#printing results?
if(0){
    print "Increasing frequency:\n";
    foreach $key (sort HashSortByValueFuncGoogleAll(keys(%googleAllHash))){
        if($googleAllHash{$key} > 9){
            print "\t\t$googleAllHash{$key} \t\t$key\n";
        }
    }
}
print "PROX HITS!\n";
foreach $key (sort HashSortByValueFuncGoogleProximity
    (keys(%googleProximityHash))){
    if($googleProximityHash{$key} > 1000){
        print "\t\t$googleProximityHash{$key} \t\t$key\n";
    }
}
}

#FUNCTION: Order
#----------------------------------------
#Assigns parts of speech to all revelant words, calls BuildPhrases()
sub Order {
    @sortedHitsAll = sort HashSortByValueFuncGoogleAll(keys(%googleAllHash));
    my @sortedHitsProx = sort HashSortByValueFuncGoogleProximity
        (keys(%googleProximityHash));
    my $i; my $j; my $k;
    my $numAll = $#sortedHitsAll;
    my $numProx = $#sortedHitsProx;
    my $numHitsPerPartOfSpeech = 5;
    #my @allNouns; @allVerbs; my @allAdjs; my @allAdvs;
    my $numAllNouns; my $numAllVerbs; my $numAllAdjs; my $numAllAdvs;
    #my @proxNouns; my @proxVerbs; my @proxAdjs; my @proxAdvs;
    my $currentPoS;
    print "About to Order.\n";
    if($usePoS == 0){
        BuildPhrasesNoPoS(@sortedHitsAll, $#sortedHitsAll);
    } else{
        #Read in the already seen word list to try and save some time.
        my %alreadySeenHash = ();
        my $currentSeenLine;
        if(open(FILE, "poslist.dat")){
            while(<FILE>){
                /\S+\s+(.+)/;
                $alreadySeenHash{$1} = $2;
            }
            close(FILE);
        }
        $numAllNouns = $numAllVerbs = $numAllAdjs = $numAllAdvs = 0;
        for($i=0; $i<=$numAll && $i<=100; $i++){
            if((exists($alreadySeenHash{($sortedHitsAll[$i])})){
                $currentPoS = $alreadySeenHash{($sortedHitsAll[$i])};
            } else{
                $currentPoS = PartOfSpeech($sortedHitsAll[$i]);
            }
        }
    }
}
alreadySeenHash{sortedHitsAll[$i]} = $currentPoS;
}

# if($currentPoS =~ m/\^\s*verb/ || $currentPoS =~ m/\^\s*\S+\s*verb/){
  if($currentPoS =~ m/\^\s*verb/){
    allVerbs[numAllVerbs++] = $sortedHitsAll[$i];
  }
  elsif($currentPoS =~ m/\^\s*adv/){
    allAdvs[numAllAdvs++] = $sortedHitsAll[$i];
  }
  elsif($currentPoS =~ m/\^\s*adj/){
    allAdjs[numAllAdjs++] = $sortedHitsAll[$i];
  }
  elsif($currentPoS =~ m/\^\s*noun/){
    allNouns[numAllNouns++] = $sortedHitsAll[$i];
  }
}

open(FILE, "\+>poslist.dat") or die "Can't open: $!\n";
for($i=0; $i<=$numAll; $i++){  
  if(exists($alreadySeenHash{sortedHitsAll[$i]})){
    print FILE
      "$sortedHitsAll[$i]\t$alreadySeenHash{sortedHitsAll[$i]}\n";
  }
}
close(FILE);

BuildPhrases();
}

#FUNCTION: BuildPhrases
#----------------------------------------
#Combines words into short phrases before scoring them with MSN Search.
sub BuildPhrases {
  my $i; my $j; my $k;
  my $magicNumber = 4;
  my $numEachPhrase = 3;
  my $currentPhrase = ();
  my $currentBestScore = 0;
  my $currentScore;

  @phrases;
  @scores;
  @syllables;
  @firstPoS;
  @lastPoS;

  my $numAdjNoun = 20;
  my $numNounVerb = 20;
  my $numVerbNoun = 20;
  my $numAdvVerb = 4;
  my $numRandom = 25;

  @tempPhrases = ();
  my @tempPhrasesSorted = ();
  @tempScores = ();
  my $numTemp = 0;

  print "About to build phrases.\n";
}
if(open(FROMAN, "-|")){
    print "IN AN Parent.\n";
} else{
    for($i=0; $i<$magicNumber; $i++){
        for($j=0; $j<$magicNumber; $j++){
            $currentPhrase = "\"$allAdjs[$i] $allNouns[$j]\""
            $MSNScore = MSNNum($currentPhrase);
            if($MSNScore > 1){
                $tempPhrases[$#tempPhrases + 1] = $currentPhrase;
                $tempScores[$#tempScores + 1] = $MSNScore;
            }
        }
    }
    @tempPhrasesSorted = sort TempSortFunc @tempPhrases;
    @tempScoresSorted = sort NumericSortFunc @tempScores;
    print STDOUT $#tempPhrasesSorted, "\n";
    for($i=0; $i<$#tempPhrasesSorted; $i++){
        print STDOUT @tempPhrasesSorted[$i], "\n";
        print STDOUT @tempScoresSorted[$i], "\n";
        print STDOUT "Adj\n";
        print STDOUT NumSyllables(@tempPhrasesSorted[$i]), "\n";
    }
    #don't let the child continue.
    exit;
}
@tempPhrases = ();
@tempPhrasesSorted = ();
@tempScores = ();
@tempScoresSorted = ();
if(open(FROMNV, "-|")){
    print "IN NV Parent.\n";
} else{
    for($i=0; $i<$magicNumber; $i++){
        for($j=0; $j<$magicNumber; $j++){
            $currentPhrase = "\"$allNouns[$i] $allVerbs[$j]\""
            $MSNScore = MSNNum($currentPhrase);
            while($MSNScore == -1){
                $MSNScore = MSNNum($currentPhrase);
                sleep(0.2);
            }
        }
    }
    @tempPhrasesSorted = sort TempSortFunc @tempPhrases;
    @tempScoresSorted = sort NumericSortFunc @tempScores;
    print STDOUT $#tempPhrasesSorted, "\n";
    for($i=0; $i<$#tempPhrasesSorted; $i++){
        print STDOUT @tempPhrasesSorted[$i], "\n";
        print STDOUT @tempScoresSorted[$i], "\n";
        print STDOUT "Noun\n";
print STDOUT "Verb\n";
print STDOUT NumSyllables($tempPhrasesSorted[$i]), "\n";
}
#kill child
exit;
}
@tempPhrases = ();
@tempPhrasesSorted = ();
@tempScores = ();
@tempScoresSorted = ();
if(open(FROMVN, "-|")){
    print "In VN Parent.\n";
}
else{
    for($i=0; $i<$magicNumber; $i++){  
        for($j=0; $j<$magicNumber; $j++){  
            $currentPhrase = "\"$allNouns[$i] $allVerbs[$j]\"";
            $MSNScore = MSNNum($currentPhrase);
            while($MSNScore == -1){  
                $MSNScore = MSNNum($currentPhrase);
                sleep(0.2);
            }
            if($MSNScore > 1){  
                $tempPhrases[$#tempPhrases + 1] = $currentPhrase;
                $tempScores[$#tempScores + 1] = $MSNScore;
            }
        }
    }
    @tempPhrasesSorted = sort TempSortFunc @tempPhrases;
    @tempScoresSorted = sort NumericSortFunc @tempScores;
    print STDOUT $#tempPhrasesSorted, "\n";
    for($i=0; $i<$#tempPhrasesSorted; $i++){  
        print STDOUT $tempPhrasesSorted[$i], "\n";
        print STDOUT $tempScoresSorted[$i], "\n";
        print STDOUT "Verb\n";
        print STDOUT "$tempPhrases[$#tempPhrases + 1] = $currentPhrase; 
        $tempScores[$#tempScores + 1] = $MSNScore;
    }
}
exit;
}
@tempPhrases = ();
@tempPhrasesSorted = ();
@tempScores = ();
@tempScoresSorted = ();
if(open(FROMAV, "-|")){
    print "IN AV Parent.\n";
}
else{
    for($i=0; $i<3; $i++){  
        for($j=0; $j<3; $j++){  
            #make sure I have enough adverbs and verbs to do this
            if($#allAdvs >= $i && $#allVerbs >= $j){  
                $currentPhrase = "\"$allNouns[$i] $allVerbs[$j]\"";
                $MSNScore = MSNNum($currentPhrase);
                while($MSNScore == -1){  
                    $MSNScore = MSNNum($currentPhrase);
                    sleep(0.2);
                }
            }
        }
    }
}
$tempPhrases[$#tempPhrases + 1] = $currentPhrase;
$tempScores[$#tempScores + 1] = $MSNscore;
}
}
@tempPhrasesSorted = sort TempSortFunc @tempPhrases;
@tempScoresSorted = sort NumericSortFunc @tempScores;
print STDOUT $#tempPhrasesSorted, "\n";
for($i=0; $i<$#tempPhrasesSorted; $i++){  
    print STDOUT $tempPhrasesSorted[$i], "\n";
    print STDOUT $tempScoresSorted[$i], "\n";
    print STDOUT "Adv\n";
    print STDOUT "Verb\n";
    print STDOUT NumSyllables($tempPhrasesSorted[$i]), "\n";
exit;
}
@tempPhrases = ();
@tempPhrasesSorted = ();
@tempScores = ();
@tempScoresSorted = ();
if(open(FROMRAND, "-|")){
    print "IN RAND Parent.\n";
} else{
    for($i=0; $i<$magicNumber; $i++){  
        for($j=0; $j<$magicNumber; $j++){  
                #build a totally random phrase, PoS AND index.
                $whichPoSUno = int(rand(4));
                $whichPoSDos = int(rand(4));
                $wordUno = 0; $wordDos = 0;
                if($whichPoSUno == 0){
                    $wordUno = $allVerbs[int(rand($#allVerbs))];
                }
            elsif($whichPoS == 1){
                $wordUno = $allNouns[int(rand($#allNouns))];
            }
            elsif($whichPoS == 2){
                $wordUno = $allAdjs[int(rand($#allAdjs))];
            }
            else{
                $wordUno = $allAdvs[int(rand($#allAdvs))];
            }
            if($whichPoSDos == 0){
                $wordDos = $allVerbs[int(rand($#allVerbs))];
            }
            elsif($whichDos == 1){
                $wordDos = $allNouns[int(rand($#allNouns))];
            }
            elsif($whichDos == 2){
                $wordDos = $allAdjs[int(rand($#allAdjs))];
            }
            else{
                $wordDos = $allAdvs[int(rand($#allAdvs))];
            }
            $currentPhrase = "\\"$wordUno $wordDos\\"";
            $MSNscore = MSNNum($currentPhrase);
}
if($MSNScore > 1){
    $tempPhrases[$#tempPhrases + 1] = $currentPhrase;
    $tempScores[$#tempScores + 1] = $MSNScore;
}

@tempPhrasesSorted = sort TempSortFunc @tempPhrases;
@tempScoresSorted = sort NumericSortFunc @tempScores;

print STDOUT $#tempPhrasesSorted, "\n";
for($i=0; $i<$#tempPhrasesSorted; $i++){
    print STDOUT $tempPhrasesSorted[$i], "\n";
    print STDOUT $tempScoresSorted[$i], "\n";
    print STDOUT "Rand\n";
    print STDOUT NumSyllables($tempPhrasesSorted[$i]), "\n";
}
}

$amount = <FROMAN>;
for($i=0; $i<$amount; $i++){
    $c = <FROMAN>; chomp($c);
    push(@phrases, $c);
    $c = <FROMAN>; chomp($c);
    push(@scores, $c);
    $c = <FROMAN>; chomp($c);
    push(@firstPoS, $c);
    $c = <FROMAN>; chomp($c);
    push(@lastPoS, $c);
    $c = <FROMAN>; chomp($c);
    push(@syllables, $c);
}
close(FROMAN);

$amount = <FROMNV>;
for($i=0; $i<$amount; $i++){
    $c = <FROMNV>; chomp($c);
    push(@phrases, $c);
    $c = <FROMNV>; chomp($c);
    push(@scores, $c);
    $c = <FROMNV>; chomp($c);
    push(@firstPoS, $c);
    $c = <FROMNV>; chomp($c);
    push(@lastPoS, $c);
    $c = <FROMNV>; chomp($c);
    push(@syllables, $c);
}
close(FROMNV);

$amount = <FROMVN>;
for($i=0; $i<$amount; $i++){
    $c = <FROMVN>; chomp($c);
    push(@phrases, $c);
    $c = <FROMVN>; chomp($c);
    push(@scores, $c);
    $c = <FROMVN>; chomp($c);
    push(@firstPoS, $c);
    $c = <FROMVN>; chomp($c);
    push(@lastPoS, $c);
    $c = <FROMVN>; chomp($c);
    push(@syllables, $c);
}
close(FROMVN);
}$amount = <FROMAV>
for($i=0; $i<$amount; $i++){  
  $c = <FROMAV>; chomp($c);
  push(@phrases, $c);
  $c = <FROMAV>; chomp($c);
  push(@scores, $c);
  $c = <FROMAV>; chomp($c);
  push(@firstPoS, $c);
  $c = <FROMAV>; chomp($c);
  push(@lastPoS, $c);
  $c = <FROMAV>; chomp($c);
  push(@syllables, $c);
}
$amount = <FROMRAND>
for($i=0; $i<$amount; $i++){  
  $c = <FROMRAND>; chomp($c);
  push(@phrases, $c);
  $c = <FROMRAND>; chomp($c);
  push(@scores, $c);
  $c = <FROMRAND>; chomp($c);
  push(@firstPoS, $c);
  $c = <FROMRAND>; chomp($c);
  push(@lastPoS, $c);
  $c = <FROMRAND>; chomp($c);
  push(@syllables, $c);
}
print "Amount was: $amount\n";
prompt "phrases length was: $#phrases\n";
for($i=0; $i<=$#phrases; $i++){
  print "$phrases[$i]\t$scores[$i]\t$syllables[$i]\t$firstPoS[$i]\t$lastPoS[$i]\n";
}
Poemize();
}
sub MSNNum {
  my $tizopic = $_[0];
  if($html =~ m/Page(.*?)of (.*?) results cont/) {
    $result = $2;
    $result =~ s/,//g;
    return $result;
  } else {
    return -1;
  }  
    # had
    #if(!(defined($result))){
    #return -1
    #}
    #after the =~ line.
  }
sub MSNSearch {
    my $topic = $_[0];
    my $start = $_[1];
    @results = ();

    $html =~ m/(<div id=.results..*?</div>)/;
    $resultsSection = $1;
    while($resultsSection =~ m/<a href="(.*)"/g){
        if(!($1 =~ m/lang=en-US/)){
            $results[#results + 1] = $1;
        }
    }
    return @results;
}

sub Poemize {
    print "About to Poemize.\n";

    #HERE phrases is broken down into separate ordered decreasing sets.
    $numPhrases = $#phrases;
    $haikuLineOne = ();
    $haikuLineTwo = ();
    $haikuLineThree = ();
    my $i; my $sel1; my $sel2; my $j;
    my $gotLine = 0;
    if(usePoS == 1){
        #added
        for($i=0; $i<=$numPhrases; $i++){
            if($firstPoS[$i] =~ m/Verb/){
                $phrases[$i] = "to " . $phrases[$i];
                $syllables[$i]++;
            }
            elsif($firstPoS[$i] =~ m/Noun/){
                $phrases[$i] = "the " . $phrases[$i];
                $syllables[$i]++;
            }
        }
        #end added
    }

    print "NUMP: $numPhrases\n";
    for($r=0; $r<3; $r++){  
        $i = 0;
        while($i<3 && !$gotLine){
            $sell = int(rand($numPhrases));
            #for($j=0; $j<=$numPhrases; $j++){  
            #don't want to start at zero every time.
            for($j=int(rand($numPhrases)); $j<=$numPhrases; $j++){  
                if($syllables[$sell] + $syllables[$j] == 5){
                    $haikuLineOne = $phrases[$sell] . " " . $phrases[$j];
                    $j = $numPhrases + 10;
                    $gotLine = 1;
                }
            }
        }
    }
    if($i >= 2){
        print "Didn't come up w/ a line.\n";
    }
$i = 0; $gotLine = 0;
while($i<3 && !($gotLine)){
    $sel1 = int(rand($numPhrases));
    #for($j=0; $j<=$numPhrases; $j++){
    for($j=int(rand($numPhrases)); $j<=$numPhrases; $j++){
        if($syllables[$sel1] + $syllables[$j] == 7){
            $haikuLineTwo = $phrases[$sel1] . " " . $phrases[$j];
            $j = $numPhrases + 10;
            $gotLine = 1;
        }
    }
    if($i >= 2){
        print "Didn't come up w/ a line.\n";
    }
    $i = 0; $gotLine = 0;
}
while($i<3 && !($gotLine)){
    $sel1 = int(rand($numPhrases));
    #for($j=0; $j<=$numPhrases; $j++){
    for($j=int(rand($numPhrases)); $j<=$numPhrases; $j++){
        if($syllables[$sel1] + $syllables[$j] == 5){
            $haikuLineThree = $phrases[$sel1] . " " . $phrases[$j];
            $j = $numPhrases + 10;
            $gotLine = 1;
        }
    }
    if($i >= 2){
        print "Didn't come up w/ a line.\n";
    }
    $i = 0; $gotLine = 0;
}
$haikuLineOne =~ s/"//g;
$haikuLineTwo =~ s/"//g;
$haikuLineThree =~ s/"//g;
print "$haikuLineOne\n";
print "$haikuLineTwo\n";
print "$haikuLineThree\n";
print "\n\n\n";

sub NumSyllables {
    my $phrase = $_[0];
    my $numSyllables = 0;
    my $inVowelGroup = 0;
    my @letters = ();
    my @words = ();
    my $i; my $j;
    @words = split(/\s+/, $phrase);
    for($j=0; $j<=$#words; $j++){
        $inVowelGroup = 0;
        @letters = split(/, $words[$j]);
    }
}
#print "\n";

for($i=0; $i<=$#letters; $i++){
    if($letters[$i] =~ m/[a|e|i|o|u]/){
        if($inVowelGroup == 0){
            $inVowelGroup = 1;
            $numSyllables++;
        }
    } else{
        $inVowelGroup = 0;
    }
}

if($words[$j] =~ m/[aeiou][^aeiou]e\S*$/){
    $#numSyllables--;
}

return $numSyllables;
}

sub TempSortFunc {
    my $i = 0;
    my $j = 0;
    while($tempPhrases[$i] ne $a){
        $i++;
        if($i > $#tempPhrases){
            die "TempSortFunc went outta bounds. beat.\n";
        }
    }
    while($tempPhrases[$j] ne $b){
        $j++;
        if($j > $#tempPhrases){
            die "TempSortFunc went outta bounds. beat.\n";
        }
    }
    $tempScores[$j] <=> $tempScores[$i];
}

sub NumericSortFunc {
    $b <=> $a
}

sub PartOfSpeech {
    my $word = $_[0];
    my $result = ();

    $dictionaryPage = get("http://www.dictionary.com/search?q=$word");

    #First, whack all html after the horizontal rule for other words.
    $dictionaryPage =~ s/<HR ALIGN=.*//s;
    while($dictionaryPage =~ m/<I>(.*?)</I>/g){
        if($1 eq "v." && (!$result =~ m/verb/)){
            $result .= "verb ";
        } elsif($1 eq "tr.v." && (!$result =~ m/verb/)){
            $result .= "tverb ";
        } elsif($1 eq "n." && (!$result =~ m/noun/)){
            $result .= "noun ";
        }
    }
    $result
}
elsif($1 eq "adj." && !$result =~ m/adj/){
    $result .= "adj ";
}else if($1 eq "adv." && !$result =~ m/adv/){
    $result .= "adv ";
}

return $result;
}

sub Extract {
    my $topic = $_[0];
    my $text = $_[1];
    my @words = ();
    my @topicWords = ();
    my $topicWordMatch = 0;
    @topicWords = split(/ /, $topic);
    #filter html tags from the title
    @words = HTMLFilter($text);
    for($q=0; $q<=$#words; $q++){
        $words[$q] =~ tr/A-Z/a-z/;
        $words[$q] =~ s/\s*//;
        $words[$q] =~ s/\s*$//;
        #THIS IS NEW, TRYING TO REMOVE trailing -S and -ING
        #$words[$q] =~ s/s\s*$//;
        if($words[$q] =~ m/[^se]s\s*$/){
            $words[$q] =~ s/s\s*$//;
        }
    }
    for($q=0; $q<=$#words; $q++){
        if(!CommonWord($words[$q])){
            #All words.
            $googleAllHash{$words[$q]}++;
        }
    }
    #Proximity words.
    for($p=0; $p<=$#topicWords; $p++){
        if($words[$q] eq $topicWords[$p]){
            #this word is a match, proximity hit the surrounding words.
            for($m=$q-1; $m>=0 && $m>$q-10; $m--){
                if(!CommonWord($words[$m])){
                    $googleProximityHash{$words[$m]}++;
                }
            }
            for($n=$q+1; $n<=$#words && $n<$q+10; $n++){
                if(!CommonWord($words[$n])){
                    $googleProximityHash{$words[$n]}++;
                }
            }
        }
    }
}

sub CommonWord {
    my $w = $_[0];
    my $exceptions = " in on a and the then them we I of around all with - from because since thus an if or but it at has for to is...";
    $w = lc($w);
will other was were your what who where when why this get

it's its any mr mrs that they have could would should
whether only about off also than are did more over end amp
had ";

#check for numerics.
if($w =~ m/[123456789]/){
  return 1;
}
#check for too short.
if($w =~ m/^..$/ || $w =~ m/^\$/){
  return 1;
}
#check for # sign word
if($w =~ m/\#/){
  return 1;
}
#check for : html tags
if($w =~ m/::/){
  return 1;
}
#check for html specifics
if($w =~ m/font/ || $w =~ m/text/ || $w =~ m/px/){
  return 1;
}
#check for at least one alphabetical value
if(!(^$w =~ m/[a-zA-Z]/)){
  return 1;
}

###sketch added, want only ALL alphabetical values.
if($w =~ m/^[a-zA-Z]/){
  return 1;
}
###end sketch

#check for exception words.
if($exceptions =~ m/$w$\s+/i){
  return 1;
}
#seems fine
return 0;

sub HashSortByValueFuncGoogleAll {
  $googleAllHash{$b} <=> $googleAllHash{$a}
}
sub HashSortByValueFuncGoogleProximity {
  $googleProximityHash{$b} <=> $googleProximityHash{$a}
}

sub HTMLFilter {
  $text = $_[0];
  $text =~ s/<script(.*?)\/>\//gsi;
  1 while $text =~ s/<([^>]*)>//gs;
  $text =~ s/\&nbsp;//g;
  $text =~ s/\quot;/g;
  $text =~ s/sans-serif//gi;
  $text =~ s/geneva//gi;
  $text =~ s/helvetica//gi;
}
$text =~ s/arial//gi;
$text =~ s/verdana//gi;
my @w = split(/\s+/, $text);
my $p;
for($p = 0; $p<=$#w; $p++){
    $w[$p] =~ s/\[{}.,()\/\*<>?+=\[\]&;]/\]/g;
}
return @w;