Computer Science 237
Assignment 4
Due before Spring Break

Lewis Carroll loved word puzzles. Let's explore one of his favorites: how to tranform one word to another through a series or *ladder* of words that differ, in steps, by single letters. In the process, maybe we'll learn a bit more about array and struct access. This week I would like you *finish* a partial translation of binary search into assembly.

You can find the starter kit on the web site as `ladder.tar.gz`. Unpack it and finish translating the `isWordC` dictionary search routine into assembly:

1. Untar the starter kit:

   ```
   tar xvfz /usr/cs-local/share/cs237/kits/ladder.tar.gz
   ```

   This creates a subdirectory, `ladder`, that contains all of the usual files, including `ladder.c` and `search.s`.

2. If you type `make` it will make a version of `ladder` based on the `C` code we hope to ultimately translate. A copy of that code is found on the reverse side of this page.

3. The program works, as shipped. If you type

   ```
   % ladder ape man
   0. ape
   1. apt
   2. opt
   3. oat
   4. mat
   5. man
   ```

   you how to solve Lewis Carroll's puzzle in five steps. Similarly,

   ```
   % ladder egg fig
   ```

   prints a one-time Google interview challenge. If you wish, you can change the default dictionary with `-d`. Don Knuth, for example, hand built a list of 5,757 five-letter words that is part of the (wonderful) Stanford GraphBase; you'll find that in `knuth`. Knuth's dictionary might construct the ladder as follows:

   ```
   % ladder -d knuth black raven
   ```

   Play with it, look at the code, watch the pieces move.

4. I have *begun* a line-by-line translation translation of `isWordC` into an assembly language routine, `isWordAsm`, in `search.s`. I make use of what we've recently learned about call frames. I have, however, left some of the more difficult parts for you, all marked `??`. Please finish the translation. Where there is multiplication or division avoid their direct implementation and come up with fast alternatives. *Otherwise, do not optimize the code.* (The **Notes**, below, may be useful.)

5. When you believe you have a working version of your assembly language code, modify the line

   ```
   lookerUpper *dictSearch = isWordC;
   ```

   of `ladder.c` to read

   ```
   lookerUpper *dictSearch = isWordAsm;
   ```

   This is a *pointer to a function* that governs how searching occurs in this utility. If you look for calls to `dictSearch`, they look pretty much like any other call. That's because *a function is simply represented by a pointer to its first instruction*. Changing `dictSearch` to point to `isWordAsm` causes the assembly code to be called.

6. Run `ladder`. It should work as expected. You can test your code against expected results with:

   ```
   make tests
   ```

   The script that performs these tests can be found in `test-suite`.

7. When you are satisfied turn in `search.s`:

   ```
   turnin -c 237 search.s
   ```

Notes.

1. Typically, you would carefully construct (and use) a stack frame that includes all the local variables and parameter values passed to `isWordAsm`. I have done much of that work for you. Please review my work. Notes from recent lectures will help. I've sized everything appropriately (pointers are quads, integers are longs, etc.). Make sure that every n-byte value is at an offset that is a multiple of n from the base pointer (`%rbp`). Notice that, following good practice, I describe the frame in comments and declare the equates so you can avoid the use of numeric frame offsets in your code.

2. This program makes heavy use of C `structs`, as well. The `dict` struct describes a dictionary, and the `entry` struct are its entries. These large objects are always referenced through pointers and, as a result, the fields in these structures are accessed through small offsets from the reference. We've not performed this type of translation in class, but it parallels, in an obvious manner, the layout of a call frame. *You will find it useful* to perform a few experiments to determine (1) the size of each structure and (2) the offsets to each field within the structure. The C compiler prefers not to pack these structures tightly, so you'll have to use a little ingenuity to figure out what those offsets should be. We'll discuss this in lab.

3. With care, I think we can avoid using any scratch registers other than `%rax` and the parameter registers (`%rdi` and friends).

4. This level of code very much models work you'll need to perform on your own on the midterm. See how much of this you can follow and finish without the help of others. It'll make you stronger when interviewers ask you questions.

The code we're converting.

```
/* Search for and return pointer to word (entry) in dictionary.
 * Assumes dictionary entries are in sorted order.
 * Return 0 if w is not a word.
 */
entry *isWordC(char *w, dict *dtn)
{
  // Binary search!
  int lo = 0, hi = (dtn->count)-1;    // remaining candidates
  int mid,dif;                        // middle and difference with w
  entry *ent, *ans=0;                 // ans: entry if found; else 0
  while (lo <= hi) { // ??
    mid = (lo+hi)/2;                  // entry in middle of candidates
    ent = (dtn->dict)+mid;            // warning: sharp curve! ??
    dif = strcmp(w,ent->word);        // effectively 'w minus word'. ??
    if (dif < 0) { hi = mid-1; }      // this entry too big,
    else if (dif > 0) { lo = mid+1; } // this entry too small, or
    else { ans = ent; break; }        // this entry just right
  }
  return ans; // inform the masses
}
```