

## Computer Science 237

### Assignment 5

Due before lab next week

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

You can find the starter kit on the web site as `ladder.tar.gz`. Unpack it and translate 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 working version of `ladder`. This is because I have given you the C code I would like you 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. Carefully translate the routine `isWordC` into an assembly language routine called `isWord`. Your source should be stored in the file `search.s`. Perform a line-by-line translation using what we've recently learned about call frames. Where there is multiplication or division, however, I'd like you to avoid their direct implementation and come up with fast alternatives. *Otherwise, do not optimize the code.* You may find the notes that follow are useful.
5. When you believe you have a working version of your assembly language code, you should modify the line

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

to read

```
lookerUpper *dictSearch = isWord;
```

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 `isWord` causes your 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 with your solution, turn it in:

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

## Notes.

1. For the most credit, you must carefully construct (and use) a stack frame that includes all the local variables and parameter values passed to `isWord`. Notes from recent lectures will help. Remember to size everything appropriately (pointers are quads, integers are longs, etc.). Also make sure that every  $n$ -byte value is at an offset that is a multiple of  $n$  from the base pointer (`%rbp`). I would like you to 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 you 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 do without the help of others. It'll make you stronger when interviewers ask you questions.

## The code you need to convert.

```
/* 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
}
```