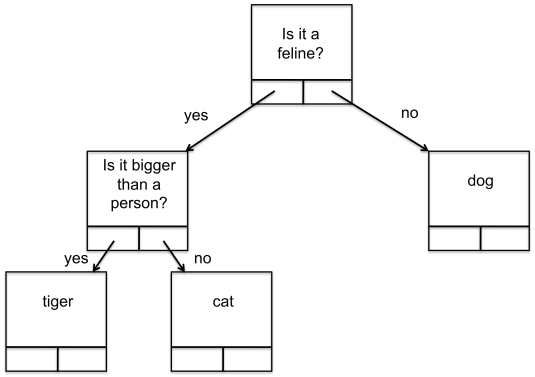
Computer Systems Organization
CSCI-UA.0201
Spring 2024

Programming Assignment 3
Due Sunday, April 15 at 11:55pm

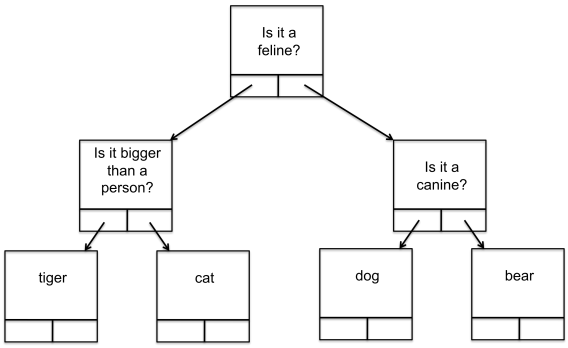
This assignment will exercise your ability to program in x64 assembly language. The task will be to implement the "Animal Guessing Game" in assembly. This game is a version of "20 Questions" where the only category is animals. You think of an animal and your program will try to guess the animal by asking you a series of "yes or no" questions.

The program uses a tree to represent its knowledge about animals. Each leaf node contains the name of an animal, and each interior node contains a question. For example, the tree,



would cause the program to ask if the animal you are thinking of is a feline. If you respond "yes", it will ask you if it is bigger than a person. If you respond "no", it will guess that you are thinking of a cat.

If you had responded "no" when asked if it is a feline, it will guess dog. If that is not correct, the program will augment the tree with the animal you were thinking of by asking you what your animal was and asking for a question to distinguish your animal from a dog. Suppose you indicate that your animal was a bear and the question to distinguish a bear from a dog is "Is it canine?". The program will ask you what the answer for "bear" is and, when you say "no", it will add a "bear" node to the tree as follows:



Note that two new nodes have been created, one for dog and one for bear. The old dog node has been overwritten with the question. Note that the tree is always a strictly binary tree, meaning that every node is either a leaf or has exactly two children.

I have provided a fair amount C code and the skeleton of the assembly code. Your job is to provide the missing assembly. Here are the steps to perform:

1. Download a compressed file appropriate for your computer, by downloading one of the following attached files:

   - assignment3_cygwin.tgz for Windows/Cygwin.
   - assignment3_macos.tgz for macOS.
   - assignment3_linux.tgz for Linux.

   As with previous assignments, save and uncompress the downloaded file in the directory where you want to work on and compile your program. To uncompress the file, in a shell, type

   tar -xzvf *filename*

   where filename is the name of the file that you downloaded.

2. The nine files that are extracted from the compressed file are:

   - animals.s, the file where you will be putting your assembly code. Important: This is the only file you will be turning in.
   - animals.h, the header file for animals.s.
   - main.c, the file containing the main function and other related functions.
   - c_animals.c, the file containing C versions of the assembly functions you will be writing.
   - c_animals.h, the header file for c_animals.c.
   - node_utils.c, a C file I wrote, containing auxiliary procedures to use.
   - node_utils.h, the header file for node_utils.c.
   - makefile, to allow you to type "make" to compile.
   - An executable file, either ben_animals (on Mac OS X or Linux) or ben_animals.exe (Cygwin), which was compiled from my version of the assignment. You can use this to see how your code should behave (although you won't really need it).

   You can immediately compile and run the program before you have made any changes. In the shell, type "make". This should generate the executable file animals (on macOS/Linux) or animals.exe (Cygwin). To run the program, type "./animals" and have fun playing. The state of the game will be saved in a file called data.dat (backed up by a file data.dat.bak), which you can ignore. It just means that you'll resume playing the game where you left off. If you want to start over with no animals, just delete data.dat.

3. In the c_animals.c file, the first function you'll see is called yes_response_c(). This function is responsible for reading in a yes/no response from the user (as one of "yes", "y", "no", or "n"), and will keep prompting until it gets such a response. I have translated it myself into assembly, in the animals.s file. Do not modify this assembly code but read it closely to make sure you understand what is going on.

   The next function in c_animals.c is called new_node_c(), which allocates and populates a node of type NODE (see node_utils.h). Your first programming task is to implement the

corresponding function new_node() in assembly language in animals.s. You will see that I have created a skeleton of the new_node() function, so please read the instructions in the file carefully and fill in assembly code where indicated.

When you have finished entering the assembly code for new_node() and are ready to test and debug it, change the three calls to new_node_c() in c_animals.c so that new_node() is called instead (that is, just remove the "_c" from the name of the function being called). Be sure not to change the definition of new_node_c() in c_animals.c, though.

Once you get the code working, proceed to the next step – which is far more extensive.

4. The largest function in c_animals.c is called guess_animal_c() which actually constructs a tree of animals and questions, and carries out the process of playing the game. Your task, of course, is to implement the corresponding function guess_animal() in assembly language in animals.s. As with the previous step, I have created a skeleton of the guess_animal() assembly function, so please read the instructions carefully and fill in assembly code where indicated.

   When you have finished entering the assembly code for guess_animal() and are ready to test and debug it, change the one call to guess_animal_c() in main.c so that guess_animal() is called instead (again, just remove the "_c" from the name of the function being called).

5. When you are finished, do two things:

   - Upload your version of animals.s to the Programming Assigment 3 page on Brightspace.
   - On the Programming Assignment 3 page on Brightspace, type the name of the system you are using (macOS, Cygwin, or Linux) into the box.

As with the previous assignments, you can discuss the assignment with other students, but you need to write your own code. Otherwise, you won't be able to do well on the final exam.

Be sure to read the "Helpful Hints" below. If you get stuck, then:

- Read the discussion board on Brightspace to see if someone posted about a similar problem to yours.
- If you have a general question, and it isn't already answered on the forum, post your question on the forum.
- If you have a question about your code, email your course assistant or co-instructor and/or come to office hours.

Also, please note that I know what compiler-generated assembly code looks like, so do not use any assembly code that is generated by a compiler (or any other tool).

Helpful Hints

Place close attention to the instructions in the animals.s about maintaining a 16-byte alignment for %rsp. It is required in assembly code that calls C library functions. After the initial "push %rbp" operation, just make sure you are decrementing %rsp by a multiple of 16. You can accomplish this by pushing things in pairs (i.e. two 8-btye pushes in a row) or by only subtracting multiples of 16 from %rsp. It's a pain, but your program might crash otherwise when you call malloc, strcpy, etc.

Take a close look at the assembly code that I posted under Resources -> Programs discussed in class for the assembly code lectures. For example, the posted code (once I discuss it) shows how to define literals strings and to make sure that the stack pointer is aligned on 16-byte boundaries.

If you look at the top of main.c, you'll see a function debug_print() that just prints out a message. I used that function to figure out where my assembly code was crashing, by inserting calls to debug_print() before and after a place in the code that I figured might be causing the crash. To call that function, just insert the following code:

macOS:

```
call   _debug_print
```

Linux:

```
call   debug_print
```

Cygwin:

```
subq   $32,%rsp    # mandatory
call   debug_print
addq   $32,%rsp
```

Remember, though, to save any caller-saved registers that you need before you call debug_print(), and restore them after the call (again, maintaining 16-byte alignment for %rsp).