CS52 - Assignment 4
Due Monday 2/29 at 11:59pm

Getting Started

• For this assignment we will be coding up functions in CS41B. Make sure you’ve read the CS41B manual on the course web page and refer to it as you work through this assignment.

• Each problem will be in it’s own file, so you will be creating four files: positive.a41, power2.a41, ackermann.a41 and smash.c.

• Refer to section A of the CS41B handout for information about running the CS41B machine. If you’re working on your own laptop, you can obtain a copy from:
  http://www.cs.pomona.edu/~dkauchak/classes/s16/cs52-s16/handouts/cs41b.jar
Save this in the same folder as your .a41 files to make your life simpler.

• Aquamacs note: Often Aquamacs uses a variable-width font which makes it difficult to line up columns. To force it to use a fixed-width font, go to
  Options → Appearance → Auto Faces
and uncheck Auto Faces.
**Optional: Touching base**

We're about a third of the way through the course and I wanted to touch base with you all, see how the course is going and see if there is any way we can make it better. I have created a very short survey. Please take 5 minutes and fill out the anonymous survey by the assignment due date (optional). I will briefly go over the results in class.

https://docs.google.com/forms/d/1_CXui0Zx_nxu5OD9r8v8v42NQlCeGPutCEGNG1iIa0Ak/

**A note on debugging in CS41B**

Eventually (hopefully already!) you’ll get the hang of the CS41b language, the stack and how to make function calls. Once you reach this stage, if all goes well, you’ll code up your functions and with minimal tweaking, they’ll work!

While I try and remain optimistic, most of you will still probably run into one last bug or two (I almost always do!) that you can’t figure out by just looking at the code. If you reach this point and you really understand what’s going on in your code, but can’t find the bug, then the next step is to step through the execution of your code and follow the execution. If you do this a step at a time, you’ll eventually find the problem.

A few thoughts on this:

- Open your code in the simulator and run it a line at a time using the ”step” button.

- Get out a piece of paper and write down a place for r2, r3 and the stack. As you execute each line of code, update what you think should be happening on the piece of paper and then double check that’s what happened in the simulator. If they’re different, figure out why!

- Some of the instructions that you write show up, and are executed as, two instructions. In particular,
  - **psh**: is a store based on r1 (sto) followed by a decrement of r1 (sbc)
  - **pop**: is a load based on r1 (loa) followed by an increment of r1 (adc)
  - **lcw**: loads the lower bits (lcl) and loads the higher bits (lch)
  - **jmp**: is a load (loa) followed by a function call (cal)

Besides these four, there will be a direct one-to-one mapping between the instructions you write and the instruction view on the right. Don’t get thrown off by this. You can still easily step through an instruction at a time and see your code execute.

- Remember, r1 holds the location of the next location where a value would be put on stack. To view the stack:
  1) Look at the memory location stored in r1, i.e. the value in r1.
2) Look at the data view on the left of the simulator and find the address (left part before
the colon) corresponding to the address in r1.

3) The address in r1 is not part of the stack. The first value in the stack is the address
immediately below the address in r1 (e.g. if r1 is 00da then the first value of the stack is
00dc). Remember, the stack will grow towards smaller memory addresses, so the stack
itself will be in the larger memory addresses.

Using these things, you should be able to step through your code a line at a time. If you understand
what’s supposed to be happening, then this can be very, very helpful at identifying small bugs.

When I first wrote Ackerman, I had a small bug in my function (I was doing a “loa r2 r1 6” when
it should have been “loa r2 r1 4”) and I only found it by doing what I describe above. I know it
may seem a bit tedious, but I promise you will find the bug much, much faster using this approach
than most others.

1 Warming up

Write a CS41B program positive.a41 that:

- Prompts the user for a number
- then prints 1 if the number is positive, 0 otherwise.
- You must include in your program a separate function that takes a single parameters as “input”
and “returns” 1 if the number is positive, 0 otherwise.

For example, if we ran the program in the simulator and input 10 we would see the following:

CS41B wants a value > 10
CS41B says > 1

2 If you push, you must pop

Write a CS41B program power2.a41 that takes as input a number n and prints out 2^n. To
accomplish this write a function that computes powers of 2 by following the recursive pattern
suggested by the SML function below.
fun power2 k = 
    if k < 0 then
        0
    else if k = 0 then
        1
    else
        double(power2(k-1));

“Doubling” may be implemented by adding a number to itself; there need not be a separate doubling function. On the other hand, the power2 function itself must be implemented recursively. That is, the function must save the return address on the stack and eventually jump back to it, and the body of the function contains a call to itself.

For example,

CS41B wants a value > 5
CS41B says > 32

3 To understand recursion, you must understand recursion

The Ackermann function is defined as:

\[
A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m-1, 1) & \text{if } m \neq 0 \text{ and } n = 0 \\
  A(m-1, A(m, n-1)) & \text{otherwise}
\end{cases}
\]  

Write a CS41B program that takes two values, first \( m \) and then \( n \), and prints out the corresponding value of the Ackermann function. Place your work in a file named ackermann.a41.

For completeness, let us specify that \( A(m, n) = 0 \) when \( m \) or \( n \) is negative

For example, here are two runs of the program:

CS41B wants a value > -3
CS41B wants a value > 1
CS41B says > 0

CS41B wants a value > 3
CS41B wants a value > 1
CS41B says > 13

The function grows very quickly, and your program will make lots of recursive calls. Below is a table with some values of the function. The value \( A(4, 2) \) requires nearly 20,000 decimal digits (see the appendix)!
Your program will actually only be able to produce results for only the smallest values of \( m \) and \( n \). If you squeeze in as much stack space as you can, you ought to be able to compute \( A(3, 2) \) but perhaps not \( A(3, 3) \). Nevertheless, your code must be structured as an accurate representation of the recursive definition above.

## 4 C anyone?

This short exercise gives us a glimpse into unsafe languages, in this case, the language is the C programming language.

The program in Figure 1 at the end of this document declares a three-element array of integers, initializes all three values to 47, calls a procedure that does nothing, and then prints out the values.

Begin by creating a file `smash.c` with exactly the code in Figure 1. Then compile it with the terminal command

```
gcc -o smash smash.c
```

and run it with the command

```
./smash
```

Next, modify the program by changing the procedure `do_nothing` so that when you run the resulting program the output values are, in order, 47, 29, and 47. Make changes only between the braces of `do_nothing` (i.e. you’re only writing the function `do_nothing`). For this exercise, we are not concerned with documentation, so you need not add comments (except for your name at the top of the file!).

The idea is to recognize that the elements of the array `a` are stored on the stack, and `do_nothing` can reach into the stack and change one of those values. You will only have to add a few lines of code, but it may take a little experimentation to find exactly the right lines. When you are finished, submit your modified version of `smash.c` in the usual way.

Here is what you need to know about the C programming language.

1. The syntax of C is very much like Java. Variables must be declared before they are used, and the equals sign is the assignment operator.

2. The variable declarations

```
int k;
int* p;
```
declare an integer \( k \) and a “pointer” \( p \) to an integer. That is, \( p \) is a variable that holds the address in memory where an integer is stored. The expression \( *p \) refers to the integer at address \( p \); it can be used on either sign of an assignment statement. Be sure that you understand the difference between the two assignment statements:

\[
p = 8;
*p = 8;
\]

The first sets the value of the variable \( p \) to be 8. The second sets the value at the memory address stored in \( p \) to be 8.

iii. It is possible to do arithmetic on addresses. For example, if \( p \) is the address of an integer in memory, then \( p + 1 \) is the address of the very next integer in memory. The compiler computes the right number of bytes to add to the address based on the word size. Be sure that you understand the difference between the two statements:

\[
p = p + 8;
*p = *p + 8;
\]

iv. Finally, the \& operator returns the address of a variable, so that the assignment

\[
p = \&k;
\]

stores the memory location of variable \( k \) in the pointer variable \( p \). (More precisely, \( p \) receives the address in memory which has been reserved for \( k \).) After that assignment, \( k \) and \( *p \) can be used interchangeably.

v. \texttt{printf} is a built-in function that prints values. It takes two parameters a string pattern and the value to be printed. There are many different patterns, but the one used in the code prints out the value as a number and followed by an end of line character. Adding some temporary print statements may help you in figuring out the solution, but make sure to remove these before you submit. Your submitted program should only print three numbers and this printing should happen with the \texttt{printf} statement already in the code.

\textbf{When you’re done}

Make sure your code compiles. If it does not, you will not get any points for that problem.

Make sure you’ve properly commented your code. You should include:

- A comment header at the top of the file with your name, the date, the assignment number, etc.
- Almost every line of assembly should have a comment to the right of it.
When you’re ready to submit, upload your assignment via the online submission mechanism. You should upload each of the four problems separately, that is, you should submit four files positive.a41, power2.a41, ackermann.a41 and smash.c. You may submit as many times as you’d like up until the deadline. We will only grade the most recent submissions.

### Grading

<table>
<thead>
<tr>
<th>File</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive.a41</td>
<td>4</td>
</tr>
<tr>
<td>power2.a41</td>
<td>6</td>
</tr>
<tr>
<td>ackermann.a41</td>
<td>8</td>
</tr>
<tr>
<td>smash.c</td>
<td>3</td>
</tr>
<tr>
<td>comments/style</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>24</strong></td>
</tr>
</tbody>
</table>

https://xkcd.com/378/
```c
#include <stdio.h>

void do_nothing()
{
}

int main()
{
    int j;
    int a[3];

    for (j = 0; j < 3; j++)
    {
        a[j] = 47;
    }

    do_nothing();

    for (j = 0; j < 3; j++)
    {
        printf("%d\n", a[j]);
    }

    return 0;
}
```

Figure 1: A “do nothing” program in the C language.
For those that are curious, Ackermann(4,2) in base 10 (all 19,729 decimal digits of it):

80638900336690743659892263495641146655030629659019724877746354937531889958786628212457076890459607829163423482876231846029079291445975710248727345958343611518492888555058919729914950057477040609974164897439

Appendix