After the exam, indicate on the line above where you fall in the emotion spectrum between “sad” & “smiley”...

<table>
<thead>
<tr>
<th>Last Name</th>
<th>Answer Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td></td>
</tr>
<tr>
<td>Student ID Number</td>
<td></td>
</tr>
<tr>
<td>Login</td>
<td>cs61c-</td>
</tr>
<tr>
<td>Login First Letter (please circle)</td>
<td>a b c d e f g h i j k l m</td>
</tr>
<tr>
<td>Login Second Letter (please circle)</td>
<td>a b c d e f g h i j k l m n o p q r s t u v w x y z</td>
</tr>
<tr>
<td>The name of your LAB TA (please circle)</td>
<td>Bing Eric Long Michael Scott</td>
</tr>
<tr>
<td>Name of the person to your Left</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your Right</td>
<td></td>
</tr>
</tbody>
</table>

All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet. (please sign)

a) Instructions (Read Me!)

- Don’t Panic!
- This booklet contains 7 numbered pages including the cover page. Put all answers on these pages; don’t hand in any stray pieces of paper.
- Please turn off all pagers, cell phones & beepers. Remove all hats & headphones. Place your backpacks, laptops and jackets at the front. Sit in every other seat. Nothing may be placed in the “no fly zone” spare seat/desk between students.
- Question 0 (1 point) involves filling in the front of this page and putting your login on every sheet of paper.
- You have 180 minutes to complete this exam. The exam is open book, no computers, PDAs, calculators.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided. You have 3 hours...relax.

<table>
<thead>
<tr>
<th>Question</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minutes</td>
<td>1</td>
<td>45</td>
<td>15</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>(+30 to review) = 180</td>
</tr>
<tr>
<td>Points</td>
<td>1</td>
<td>20</td>
<td>10</td>
<td>18</td>
<td>11</td>
<td>15</td>
<td>75</td>
</tr>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 1: Brian Harvey would be proud... (20 pts, 45 min)

Working with a partner, you want to implement a scheme sentence ADT (a linked list of words), and you're going to start with the constructor se(word, sentence). Your partner writes the interface, and shows you how they want to call se, and your job will be to write the subroutine. You agree on the standard struct definition on the right. Here's how your friend wants to call se:

```c
#include <string.h>
#include <stdio.h>
int i;
int main()
{
  char word[8];
  sentence_t *head = (sentence_t *) malloc (sizeof(sentence_t)); // assume succeeds...
  head->word = "go!";
  head->next = NULL;
  for (i=1;i<=3;i++) {
    printf("Word? "); scanf("%s",word);
    se(word,head);
  }
  // Right here, if you ran this program, and typed the following:
  // Word? one
  // Word? two
  // Word? three
}
```

Here's how your f riend wants to call se:

```c
// Assume compiler packs tight
typedef struct sentence_node {
  char *word;
  struct sentence_node *next;
} sentence_t;
```

```c
void se(char *word, sentence_t *head) {
  sentence_t *new = (sentence_t *) malloc (sizeof(sentence_t));
  if(!new) exit_with_msg("se: Couldn't allocate space for a new list node");
  new->word = head->word;
  new->next = head->next;
  head->next = new;
  head->word = (char *) malloc (sizeof(char)*strlen(word)+1);
  if(!head->word) exit_with_msg("se: Couldn't allocate space for the string");
  strcpy(head->word,word);
}

void exit_with_msg(char *msg) { printf("Error! %s\n",msg); exit(1); }
```

a) Complete se so it will work with your friend’s main code, even if he later supports longer words. Think about error checking you might need, and call exit_with_msg below, if needed. Your solution should not need any looping/recursion, just statements that connect things up correctly.

```c
void se(char *word, sentence_t *head) {
  sentence_t *new = (sentence_t *) malloc (sizeof(sentence_t));
  if(!new) exit_with_msg("se: Couldn’t allocate space for a new list node");
  new->word = head->word;
  new->next = head->next;
  head->next = new;
  head->word = (char *) malloc (sizeof(char)*strlen(word)+1);
  if(!head->word) exit_with_msg("se: Couldn’t allocate space for the string");
  strcpy(head->word,word);
}
```

b) At “Right here”, how much space (Bytes) is used by i, word, & head (and their data) together?

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
<th>Static</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 (8 word,4 head)</td>
<td>4*8(cons)+14(str)=46</td>
<td>4 (i) [or 8..go!\0]</td>
<td>4 (go!\0) [or 0]</td>
</tr>
</tbody>
</table>
Question 1: Brian Harvey would be proud... (continued) (20 pts, 45 min)

That interface is counterintuitive, you say. You decide the interface should be changed so that the boxed call to se looks like this: \( \text{head = se(word, head)} \) and you now ask them to write se, but to save space, only keep ONE copy of any word. That is, they should check if the word is already in your sentence, and if it is, to share it. They write the following but you suspect share_or_new has bugs.

```c
sentence_t *se(char *word, sentence_t *head) {
    sentence_t *new = (sentence_t *) malloc (sizeof(sentence_t)); // assume succeeds...
    new->word = share_or_new(word, head);
    new->next = head;
    return new;
}

char *share_or_new(char *word, sentence_t *head) {
    if (word == head->word) // Found it, share!
        return head->word;
    if(head == NULL) {
        char new[10];
        strcpy(new,word);
        return new;
    }
    return share_or_new(word, head->next); // Keep looking...
}
```

(c) Simulate running the program with the same user input as before.
If the code returns without error, show in scheme format, like
(three two one go!), what head is at “\( \text{← Right here} \)”.
If the code crashes, indicate the value of i when the error occurs. _____________________________

d) Just for fun, change the “=” in line 1 to “==” (make that change permanent in the code above).
Instead of our previous user input, we type the word go! three times. If there is no crash by the time we get to “\( \text{← Right here} \)”, draw the box-and-pointer diagram for head (just like on the previous page).
If there is a crash, list the functions in the stack frame before the error.
E.g., if main called sub, which called subsub, which crashed, you’d write: main→sub→subsub→ERROR.

```c
main → se → share_or_new → share_or_new → ERROR
```

e) Finally, fix all the bugs so se works in general (not just for this particular main). Your fixes can be of the form: “change line i to (fill in the blank)”, or “insert (fill in the blank) before/after line i”, or “move lines i through j above line k. You may not need all the blanks.

Change line 1 to if(!strcmp(word,head->word))

Change line 4 to be char *new = (char *) malloc (strlen(word)*sizeof(char)+1);

Add if(!new) exit_with_msg("share_or_new: Couldn’t malloc space for string"); below 4

Move lines 3 through 7 (8 with the addition of the line above) above line 1
Question 2: L1 and L2 below are Booth needed for the algorithm… (10 pts, 15 min)
Below is a MIPS function. Read it carefully and use it to answer the following questions.

```mips
myst:    move $v0, $0
L1:      andi $t0, $a0, 1
         beq $t0, $0, L2
         addu $v0, $v0, $a1
L2:      sll  $a1, $a1, 1
         srl  $a0, $a0, 1
         bne $a0, $0, L1
         jr $ra
```

a) In one sentence, briefly summarize what `myst` does, precisely (assume the inputs are unsigned). **Don’t** describe the algorithm, abstract at a high level what the algorithm is doing. (e.g., “it returns the # of bits in common of its three arguments” or “sets $a1 to -$a0”)

**It returns the lowest 32 bits of $a0*$a1 in $v0 = ($a0*$a1) mod 2^{32} = ($a0*$a1)&0xFFFFFFFF.**

For question (b) only, assume $a1$ is a two’s complement negative number.

b) Would `myst` still work? Briefly explain why or why not.

**Yes. Adding two’s complement numbers is no different than adding unsigned numbers, The answer’d be a 2s complement #. We’re using addu, so there are no overflow worries.**

For question (c) and (d), let’s change the `srl` to `sra`.

c) For what values of its inputs would `myst` still do what you said it does in (a)?

**When the MSB of $a0$ is 0 (i.e., $a0$ & 0x80000000 == 0)**

d) When it *doesn’t* return the same answer, what does `myst` return / do?

**loops indefinitely ($a0 never gets to 0)**
**Question 3: Tasting Menu… (18 pts, 30 min)**

Thanks to a breakthrough, we can store 3 values per digit instead of the usual 2. Rather than encoding 0, 1, and 2, we choose to encode -1, 0, and 1 (We call these binary-with-negative base 2 digits *binets*). To make notation cleaner, we’ll use \( \dagger \) to represent \(-1\), and we’ll precede binets with \( \text{obn} \). Binet numbers allow us to express negative numbers with an “unsigned” encoding, which is great. However, now some numbers have multiple representations, like \( 1_{10} = \text{obn}0001 = \text{obn}001\dagger \) (i.e., \( 1 \times 2^1 + -1 \times 2^0 = 1 \)).

a) How many unique numbers can be represented using \( N \) unsigned binets? \( 2^{N+1} - 1 \)

b) Recall that we negated 2’s complement binary numbers by inverting each bit and then adding a constant offset (1). It turns out we can negate unsigned binets with a similar technique! Fill in the truth table for what the binet “inversion” function should do *per bit*, and the constant offset that should be added at the end to make it work perfectly.

<table>
<thead>
<tr>
<th>Original bit</th>
<th>New bit</th>
<th>Constant Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Assume the heap has 7 contiguous free bytes left (below), and *best fit*, given two equally good candidates, will pick the leftmost one.

c) List all the representation(s) for zero for a binet nibble (4 binets).

d) Fill in the gaps in the C snippet on the right such that the location of \( e \) is *different* depending on whether `malloc` uses first-fit, next-fit, or best-fit. Indicate in the diagram below where the remaining data is (by a single letter) and where the fits would place \( e \) (by writing *FIRST*, *NEXT* and *BEST* in the appropriate slot).

e) We’re using C99 and need to store \( n \) integers, but can’t decide how to reserve space for them. What is the *best argument* for using each one? Also, if \( \text{$s0$} \) contains the value of \( n \), what *minimal* MAL MIPS would the C translate to, that reserves space and puts \( \&A[0] \) (i.e., where \( A \) points) in \( \text{$s1$} \)? All volatile registers have been saved & we’ll soon need \( \text{$s0$} \) and \( \text{$s1$} \). (Note: `malloc` is just a function)

```c
char *a, *b, *c, *d, *e;
int A[n];
int *A = (int *)malloc(sizeof(int)*n);
a = (char*) malloc(_2__);
b = (char*) malloc(_1__);
c = (char*) malloc(_1__);
d = (char*) malloc(_1__);
e = (char*) malloc(1);
free(_a__);
free(_c__);
```

<table>
<thead>
<tr>
<th>Why?</th>
<th>int A[n];</th>
<th>int *A = (int *)malloc(sizeof(int)*n);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Really fast (3 fast MIPS instructions)</td>
<td>Reliability (we can catch if malloc fails) OR can give storage to caller</td>
<td></td>
</tr>
<tr>
<td>MAL MIPS</td>
<td>sll $t0 $s0 2</td>
<td>sll $a0 $s0 2</td>
</tr>
<tr>
<td></td>
<td>subu $sp $sp $t0</td>
<td>jal malloc</td>
</tr>
<tr>
<td></td>
<td>move $s1 $sp</td>
<td>move $s1 $v0</td>
</tr>
</tbody>
</table>

f) How many total different instructions could we list on the green sheet if, instead of an explicit *shamt* field, we stored the shift amt in the unused \( rs \) register and expanded the *funct* field to use the *shamt* bits also? Be exact. \( 63+2^{11}(=2,048)=2,111 \)
**Question 4:** Did somebody say “Free Lunch”?! (11 pts, 30 min)
Consider two competing 8-bit floating point formats. Each contains the same fields (sign, exponent, significand) and follows the same general rules as the 32-bit IEEE standard (denorms, biased exponent, non-numeric values, etc.), but allocates its bits differently. To save you time, you only need to complete and circle the (LEFT or RIGHT) blank whose value is closest to zero, that’s the only one we’ll grade! (If they’re the same value, write the answer in both, & circle both). E.g., The number represented by 0x00 was 0 for both, so we circled both. But for “exponent bias”, just from the # of EE...E bits in each, we know |LEFT’s bias| < |RIGHT’s bias|, so there’s no need to calculate or write the answer on the RIGHT.

"LEFT" format: 

<table>
<thead>
<tr>
<th>S</th>
<th>EE</th>
<th>MMMM</th>
</tr>
</thead>
</table>

```
scratch space (show all work here)
```

**Number represented by 0x00:**

- Exponent Bias: **1**  
  Numbers (0 ≤ n < 1): ____________________________  
  Numbers (1 ≤ n < 2): ____________________________  

- Difference between two smallest positive values: 2^-5  
- Difference between two biggest non-∞ values: 2^4  
- Positive Integer closest to 0 it cannot represent: (4)  

f) Which implementation is better for approximating π, LEFT or RIGHT? (circle one)
**Question 5: Euclid’s Revenge… (15 pts, 30 min)**

The *Euclidean Algorithm* is used to find the greatest common divisor (GCD) of two numbers $a$ and $b$. E.g., GCD(8,6)=2. The algorithm works because $(a \geq b)$: GCD$(a,0)=a$, GCD$(a,b)=\text{GCD}(b,a \mod b)$.

a) Implement the Euclidean Algorithm below recursively (but as efficiently as you can) in MAL MIPS. Follow the hints given by the comments; you may not need all the lines. To assist you, assume that mod is correctly implemented as a MIPS subroutine that returns $a0 \mod a1$ in $v0$. The first letter of the first MIPS instruction is $b$.

```mips
ne $a1, $0, recGCD: b__________________________ # Handle base case first; we assume $a \geq b
addu $v0, $a0, $0 ## return a
j done
addiu $sp, $sp, -8 ## prologue rec:
gCD:

Recursive case

sw $ra, 4($sp) # < this should be circled (part d)
sw $a1, 0($sp) ## save $b
use

$s0
jal mod
move $a1, $v0 ## b = a mod b
lw $a0, 0($sp) ## restore a
jal GCD

## epilogue

0($sp)

lw $ra, 4($sp)
addiu $sp, $sp, 8

done: jr $ra
```

For questions (b) & (c), assume the address of GCD is 0x1000008

0x14A00002, 0x14050002 Translate the first MIPS instruction to hex:

```
0x0C400002
```

0x0C400002 Translate the jal GCD to hex:

```
0x0C400002
```

d) You notice your code sometimes crashes when you call GCD with large numbers. Circle the instruction that causes the crash, and in two words, explain why does it crash.

```
stack overflow
```

In two words, what can you do to combat this instability?

```
rewrite iteratively
```