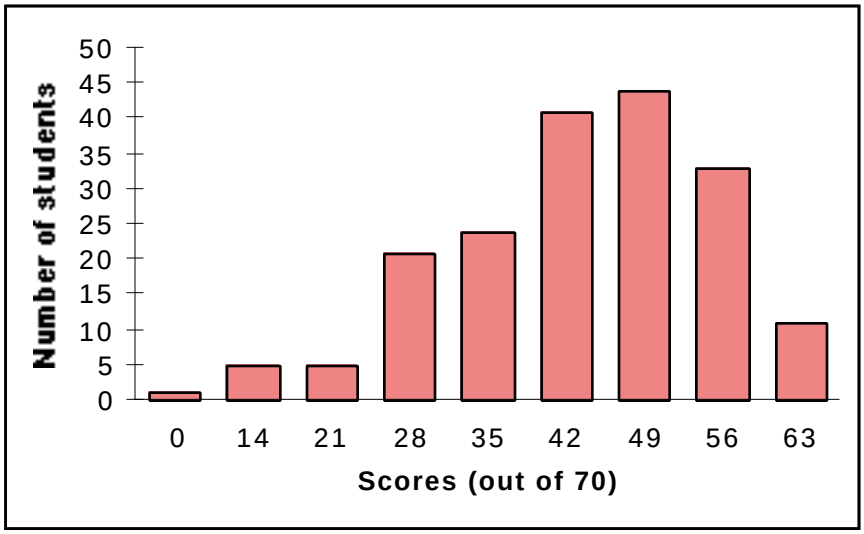
CS107 Midterm Solution

Thanks to the valiant, late-night efforts of my five awesome TAs, the exams are graded and will reside in the Gates Building lobby closest to my office. The average in the exam was just over 46 (or about 67%), the median was a 48, and the standard deviation was between 11 and 12 points. There were no perfect scores, but two students received a 68, and four other scored above 65. A majority of the students fell in the 45-55 range, so those of you in that range are in good company. I thought the exam was difficult—less difficult than this past spring's, but much more difficult than the one I gave this time last year. Grades on the code generation were **way** up over this past spring's, and I commend you on that. I was also pleased with the `lexiconADT` code I saw. Problems 1 and 3 were canonical CS107 midterm problems, and it was nice to see so many of you with a solid understanding of memory manipulation and management.



For those of you who didn't do too well, I'm certainly happy to talk it over with you in office hours. Assuming that it was just a bad day for you, you might come in and identify that it was the lateness of the hour, or the exam statement, or something else that prevented you from performing well. Recall my policy where I drop the midterm if you do substantially better on the final, and just count the final more. Since I don't know that many of you all that well, it's really up to you to determine whether or not the exam is representative of your understanding, and based on that diagnosis decide whether or not you should fight back or just give up. Again, I'm more than happy to talk to anyone about their performance on the exam and perhaps why it doesn't reflect their true understanding of the material. I understand that the exams are important, and I want students to at least feel that the evaluation process is a fair and reasonable one.

Regrade Policy

Many of you will want to contest the score on one or more of the four problems. Here are a few ground rules:

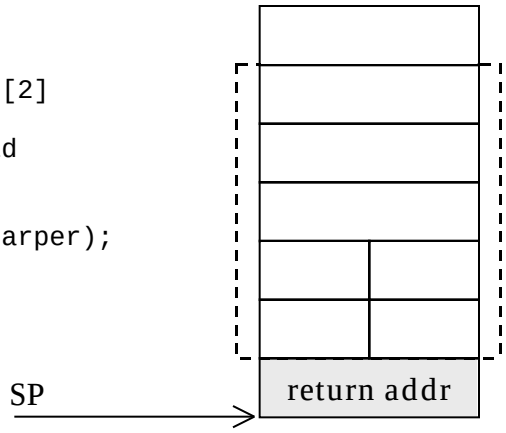
1. All regrade requests must be made within seven calendar days. Since the exams are officially handed back in lecture on Friday, that means that all regrade requests must be made by next Friday. There are zero exceptions to this rule. Zero.
2. All regrade requests must be made in writing and directed to the grader. Each of the TAs initializes his/her work, so it should be clear from your exam who graded what. You should explain why you think your answer is correct or undeserving of as severe a penalty as was applied. The exam solution forms the rest of this handout, and grading criteria will be made available on the Miscellany Page of the course web site. If you contest the grading of more than one problem, then submit all written requests to one of the relevant TAs, and s/he will pass it around.
3. TAs will sometimes need to regrade the entire problem, so they must be given 24 hours to walk away and thoughtfully comb through your full answer again to regain the context they had the first time they graded it.

Problem 1: Boston Public—Where Every Day is a Fight (20 points — average score: 16.4)
You are to generate code for the following nonsense program. Don't be concerned about optimizing your instructions or conserving registers. We don't ask that you draw the activation records, but it can only help you get the correct answer if you do.

```
struct teacher {
   short lauren[4];
   short *harper;
   char *harry[2];
};

static char *TrainedForBattle(struct teacher *lipschultz, short **history);

static void WinslowHigh(struct teacher buttle,
                        struct teacher *marilyn)
{
   marilyn->harper = buttle.lauren + 2;
   buttle.harry[10] = TrainedForBattle(&buttle, &buttle.harper);
}
```

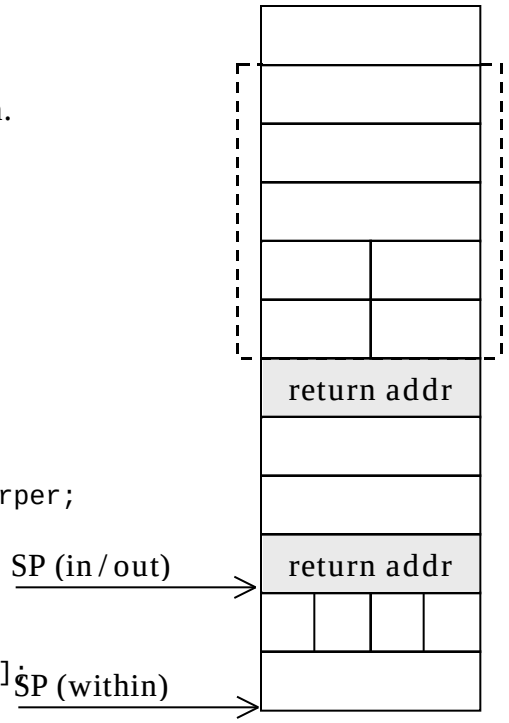
- Generate code for the entire `WinslowHigh` function.

```
// marilyn->harper = buttle.lauren + 2;

R1 = SP + 8;          // calculate &buttle.lauren[2]
R2 = M[SP + 24];      // load marilyn
M[R2 + 8] = R1;       // and write to harper field

// buttle.harry[10] = TrainedForBattle(&buttle,
//                                     &buttle.harper);

R1 = SP + 4           // compute &buttle
R2 = SP + 12          // compute &buttle.harper
SP = SP - 8;
M[SP] = R1;
M[SP + 4] = R2;
CALL <TrainedForBattle>
SP = SP + 8;
M[SP + 56] = RV;
RETURN;
```



```
static char *TrainedForBattle(struct teacher *lipschultz, short **history)
{
   int riley;
   char guber[4];

   guber[2] = *lipschultz->lauren;
   lipschultz = (struct teacher *) lipschultz->harper;
   lipschultz->harry[0] = lipschultz->harry[riley];
   return (char *)(history + 1);
}
```

- Generate code for the entire `TrainedForBattle` function.




```
// int riley;
// char guber[4];

SP = SP - 8;

// guber[2] = *lipschultz->lauren;

R1 = M[SP + 12];        // load lipschultz
R2 =.2 M[R1];           // load lauren field
M[SP + 6] =.1 R2;       // store as a char

// lipschultz = (struct teacher *) lipschultz->harper;

R1 = M[SP + 12];
R2 = M[R1 + 8];         // load harper field
M[SP + 12] = R2;        // force pointer in

// lipschultz->harry[0] = lipschultz->harry[riley];

R1 = M[SP + 12];        // load lipschultz
R2 = R1 + 12;           // cache offset to harry field
R3 = M[SP];             // load riley
R3 = R3 * 4;            // multiply by sizeof(int) for raw offset
R4 = R2 + R3;           // computer &harry[riley]
R5 = M[R4];             // load value
M[R2] = R5;             // store value

// return (char *)(history + 1);

R1 = M[SP + 16];
RV = R1 + 4;
SP = SP + 8;
RETURN;
```

Problem 2: Extending the DArray (10 points —average score: 7.3)

Behind the scenes in `darray.c`, your `struct DArrayImplementation` looked something like this:

```
struct DArrayImplementation {
   void *elems;
   int logicalLength;      // number of actual elements stored
   int allocatedLength;    // number of elements allocated
   int elemSizeInBytes;
   void (*freefn)(void *elem);
};
```

Your job here is to provide the implementation of one additional function, `ArrayFilter`. Assume your implementation appears in the `darray.c` file, so that it has full visibility of the `struct DArrayImplementation`. However, you should not use any of the other `DArray` functions such as `ArrayNth` or `ArrayDeleteAt`. **Hint**: it's best to traverse from back to front.

```
/*
 * Function: ArrayFilter
 * ---------------------
 * Disposes of and removes all client elements which fail
 * the specified predicate test, compressing the elements which
 * remain while maintaining their relative order.
 */

void ArrayFilter(DArray array,
                 bool (*predFn)(const void *elem, void *clientData),
                 void *clientData)
{
   int i;
   void *elemToTest;

   for (i = array->logicalLength - 1; i >= 0; i--) {
      elemToTest = (char *) array->elems + i * array->sizeInBytes;
      if (!predFn(elemToTest, clientData)) {
         if (array->freefn != NULL)  // if obligated, dispose
            array->freefn(elemToTest);
         memmove(elemToTest, (char *) elemToTest + array->elemSizeInBytes,
                 array->elemSizeInBytes * (array->logicalLength - i - 1));
         array->logicalLength--;
      }
   }
}
```

Problem 3: The LexiconADT (30 points — average score: 18.1)

One of the most common data structures around is the lexicon—specifically, a dictionary which stores the words, but doesn't bother to store any of the definitions. While many applications might need more than that, some applications only need to know whether or not a given string is a meaningful, properly spelled word in the English language. For this exam problem, you'll be implementing a `lexiconADT` to store an arbitrarily large collection of words, but you'll be pinned to storing them in the way that I lay out for you very clearly.

Here is the condensed interface file:

```
// File: lexicon.h
// ---------------

typedef struct lexiconCDT *lexiconADT;

lexiconADT LexiconNew();
void LexiconEnterWord(lexiconADT lex, const char *word);
bool LexiconContainsWord(lexiconADT lex, const char *word);
bool LexiconContainsWordBeginningWith(lexiconADT lex, const char *prefix);
void LexiconMap(lexiconADT lex,
                void (*lexmapfn)(const char *word, void *auxData),
                void *auxData);
void LexiconFree(lexiconADT lex);
```

You will be given the `struct lexiconCDT`, but it'll be up to you to implement the three functions typed in boldface. In particular, you'll need to implement `LexiconNew`, `LexiconEnterWord`, and `LexiconMap`.

Here's the design you **must** adhere to:

- Each word is stored in one of 676 `DArrays`. In particular, words beginning with `"aa"` are stored in the first `DArray`, words beginning with `"ab"` are stored in the second `DArray`, and so forth. The first 26 `DArrays` all store words beginning with the character `'a'`, so that the 27th `DArray` stores words beginning with `"ba"`, the 28th `DArray` stores words beginning with `"bb"`, and so forth. For simplicity, we'll assume that all words are of length two or more—specifically, you needn't worry about the empty string or single-character words. You should also assume that all words are lower case and free of punctuation marks.
- Because the first two characters are effectively captured by the `DArray` number, these two characters aren't explicitly stored anywhere. Only letters 3 and up are ever copied into the `DArray`. `"abacus"`, for example, is effectively stored in a `lexiconADT` provided the 2nd `DArray` (storing words that begin with `"ab"`) contains the suffix `"acus"`. `"ab"` plus `"acus"` makes `"abacus"`.
- Because the size of the elements stored in these `DArrays` must be the fixed at the time of construction, you're going to devote eight bytes to the storage of each suffix, even if the suffix is considerably longer than that. A large fraction of words in the lexicon will probably be less than 10 letters long, and since the first two characters can be inferred

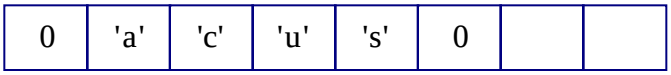
from the DArray storing the suffix, the remaining eight characters can potentially reside directly in the DArray.

- Larger words can't be compactly stored this way. Therefore, the eight bytes need to be used differently when storing larger suffixes. Here's the final heuristic:

4. The first of the eight bytes will be to tell us whether the remaining seven characters are enough to store the entire suffix. This first bytes will store a 0 (equivalently, a '\0') when the suffix of the word being stored is 7 characters or less. The suffix should itself be terminated with a '\0' if its length is less that 7, but suffixes of length 7 should not store the '\0', since there won't be any room for it.
5. Should the first byte store anything nonzero, the remaining seven bytes are divided up. Bytes 2, 3, and 4, store the first three characters of the suffix, but bytes 5 through 8 store the address of a dynamically allocated character array large enough to store the rest of them.
6. When analyzing the suffix, it's the implementation's responsibility to check this first byte to see if the suffix is fully stored in the remaining seven bytes, or if the suffix is broken up into two separate arrays.

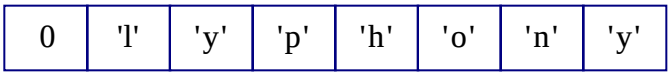
Here are a few examples:

- The word "abacus" would take up eight bytes in the 2nd (or the "ab") DArray. The suffix would be stored as follows:



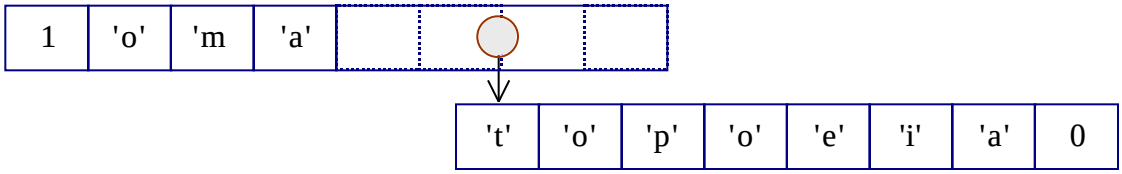
The first bytes stores a zero, because the "acus" suffix can be fully stored in the remaining seven bytes. The leading 0 reminds the implementation that everything resides in the eight-byte chunk.

- The word "polyphony" would take up eight bytes in the "po" DArray. The suffix would be stored as follows:



Again, the "lyphony" suffix can be wedged into the seven-byte chunk. The only difference here is that the '\0' can't be stored. Your implementation shouldn't be confined by this, as it should just realize that at most seven characters can be accommodated.

- The word "onomatopoeia" is a mighty big one. The "on" DArray would contain eight bytes on behalf of "omatopoeia", but those eight bytes would look this:



Note the 1 in the very first byte. That the signal that the last four bytes point to dynamically allocated space (allocated by the implementation, of course) to store the suffix that just couldn't fit in the eight primary bytes. The dynamically allocated portion is always null-terminated, and is always exactly the size it needs to be to store the rest of the characters.

Here is the CDT completion and the requisite documentation:

```
/*
 * Completion of struct lexiconCDT
 * -------------------------------
 * The implementation uses a variation
 * on hashing, where the first two characters
 * of a word determine which of the 676
 * DArrays the word will be stored in.
 *
 * Words beginning with "aa" will be
 * stored in buckets[0], those begining
 * with "ab" will be stored in buckets[1], and
 * those beginning with "zz" will be stored
 * in buckets[675].
 *
 */

struct lexiconCDT {
  DArray buckets[676];
};
```

Using the implementation comments to guide you, provide implementations for LexiconNew, LexiconEnterWord, and LexiconMap over the next several pages.

Provide implementations for the routine `LexiconNew`.

```
static bool WordFitsInSpace(void *elem)
{
   return ((*(char *)elem) == '\0');
}

static void StringTailFree(void *elem)
{
   if (!WordFitsInSpace(elem))
      free(*(void **)((char *) elem + 4));
}

/*
 * Function: LexiconNew
 * --------------------
 * Allocates space for a brand new
 * lexiconADT, free of any words, and
 * returns it.  You may assume that malloc
 * always succeeds, so no assert statements
 * need be included.  You will need to write
 * the free function that knows how
 * to deal with eight-byte chunks that in some
 * situations store a pointer.
 */

lexiconADT LexiconNew() // 8 points
{
   int bucket;
   lexiconADT lex;

   lex = malloc(sizeof(struct lexiconCDT));
   if (lex == NULL) return NULL; // just my choice... could have asserted

   for (bucket = 0; bucket < 676; bucket++) {
      lex->buckets[bucket] = ArrayNew(8, 32, StringTailFree);
   }

   return lex;
}
```

Now take this and the next page to implement the `LexiconEnterWord` function. Take the time to decompose out code to helper functions if you foresee them being useful in the conext of searching for and mapping over words. Think about this before you write. You have as much time as you want, so don't rush yourself. Be clean and tidy about everything.

```
static int HashToBucket(const char *word)
{
   int msb = word[0] - 'a';            // more significant byte
   int lsb = word[1] - 'a';            // less significant byte
   return msb * 26 + lsb;              // base 26 number in range [0, 676)
}

static void CreateSuffix(char suffixdest[], char *suffixsource)
{
   suffixdest[0] = (strlen(suffixsource) <= 7) ? '\0' : '\001';
   if (suffixdest[0] == '\0') {
      strncpy(suffixdest + 1, suffixsource, 7);
   } else {
      strncpy(suffixdest + 1, suffixsource, 3);
      suffixsource = strdup(suffixsource + 3);
      memcpy(suffixdest + 4, &suffixsource, sizeof(char *));
   }
}

/*
 * Function: LexiconEnterWord
 * --------------------------
 * Updates the specified lexicon to store the
 * specified word.  The word is assumed to contain
 * only lower case letters and nothing else.  Remember
 * that words or length 10 or more are broken
 * up into several portions.
 *
 * Ideally, the DArray storing the word would
 * be sorted, and you would maintain that sorted
 * order.  But here just append the new entry to
 * the back of the DArray and don't worry about
 * sorting it.  Also, assume that the word isn't
 * already in the lexicon, and just blindly add it
 * even if it was previously inserted.
 */

void LexiconEnterWord(lexiconADT lex, const char *word) // 10 points
{
   int bucket;
   char suffix[8];

   bucket = HashToBucket(word);
   CreateSuffix(suffix, word + 2);
   ArrayAppend(lex->buckets[bucket], suffix);
}
```

Finally, implement the `LexiconMap` function. You should assume for the sake of convenience that no word in the lexicon will ever be larger than 64 characters long. You will need to define some helper records and functions in order implement this properly. In particular, you might find the following structure helpful:

```
struct mapdata {
   char word[64];    // static spot where string can be synthesized
   void (*clientmapfn)(const char *, void *);
   void *clientdata;
}

void LexiconArrayMap(const void *elem, void *clientdata)
{
   struct mapdata *data = clientdata;
   const char *suffix = (char *) elem + 1;

   if (WordFitsInSpace(elem)) {
      strncpy(data->word + 2, suffix, 7);
      data->word[9] = '\0'; // just in case word is of length 9
   } else {
      strncpy(data->word + 2, suffix, 3);
      strcpy(data->word + 5, *(char **)(suffix + 3));
   }

   data->clientmapfn(data->word, data->clientdata);
}

/*
 * Function: LexiconMap
 * --------------------
 * Maps the specified function (which takes a word
 * in the dictionary, expressed as an ordinary, null-
 * terminated string, and client data) over all the
 * words in the lexicon.  Since the client doesn't
 * know how the strings are being stored, they need
 * to assume the strings are being stored as traditional
 * C-strings, and it's the job of the implementation
 * to pass the words through to the callback function
 * that way.
 */

void LexiconMap(lexiconADT lex,
                void (*mapfn)(const char *, void *),
                void *auxdata)  // 12 points
{
   int bucket;
   struct mapdata data;

   data.clientmapfn = mapfn;
   data.clientdata = auxdata;
   for (bucket = 0; bucket < 676; bucket++) {
      data.word[0] = 'a' + bucket / 26;
      data.word[1] = 'a' + bucket % 26;
      ArrayMap(lex->buckets[bucket], LexiconArrayMap, &data);
   }
}
```

Problem 4: Short Answers (10 points — average score: 4.5)

For each of the following questions, we expect short, insightful answers, writing code or functions only when necessary. Clarity and accuracy of explanations is the key.

- Assuming all instructions and pointers are 4 bytes, explain why instructions of the form `M[x] = M[y] + M[z]`, where `x`, `y`, and `z` are legitimate but otherwise arbitrary memory addresses, aren't included in the instruction set. (3 points)

  Since all instructions are 4 bytes, information about three addresses would somehow need to be compressed to 32 bits. Each pointer needs its own 4 bytes = 32 bits in order to faithfully represent an address, so there's no way to compress 96 bits into 32 without losing information. Even if three addresses could be compressed somehow, no bits would be left for the opcode (which in this example is an addition.)

- Our activation record model wedges the return address information below the function arguments and above the local parameters. Why not pack all variables together, and place this return address at the top or the bottom of the activation record? (4 points)

  The return address can't be placed on the bottom, because the calling function doesn't know where the bottom of the full layout even is. The caller knows nothing of what locals are allocated by the callee's instruction list.

  The return address could, in most cases, be placed on the top, since the callee function generally knows what all of the variables are and how large everything is. The key exception—the reason that this as a convention cannot be adopted—is that some functions take a variable number of arguments, such as `printf` and `scanf`. In those situations, no information about the number and size of the parameters is around, so it can't reliably understand where the stored `PC` would be.

· Consider the following program, one very similar to an example I gave in lecture:

```
main()
{
   int i;
   int scores[100];

   for (i = 0; i <= 100; i++) {
      scores[i] -= 4;
   }
}
```

Why does this program run forever? (3 points)

No, this wasn't a typo, and no I didn't change the activation layout conventions on you. The stack frame for `main` looks like that below. The integrity of i is never compromised. It travels through all values from `0` up and stops at `101`. That's when the `for` loop ends. The first 100 revolutions of the loop took whatever integers happened to reside in the `scores` array and made all of them `4` less than they used to be. The 101st revolution took the stored PC value, stored in the four bytes overlayed by `scores[100]`, and made it `4` less, too. Once the loop exits, so does the function, and that stored `PC` is once again interpreted a pointer into the code segment, but unbeknownst to it, that stored address references the `CALL<main>` instruction (all instructions are 4 bytes, recall). The `PC` is restored when main exits, only to fetch the `CALL<main>` instruction again, execute it and start the whole cycle over again. Infinite loop again, but for a different reason.