CS107 Midterm Examination

This is a closed-book, closed-note exam. Your exam should have 15 pages.

All stack and heap layout should be as done in class and in handouts: stack on the left, growing down, heap on the right. For code generation, use only legal RISC instructions in the "R1 = R2 + R3" syntax. You do not need to optimize your instructions or use of registers. You should assume that integers, floats, and pointers are each 4 bytes and that there are no alignment restrictions. Comments help you receive partial credit if your intentions are correct but your code is wrong. You are strongly encouraged to comment your machine code, since it is basically unreadable in its native form.

Good luck! And take your time!

`leland` username: _____

Last Name: _____

First Name: _____

SUID: _____

I accept the letter and spirit of the honor code. I have not given nor received aid on this exam. I pledge to write more neatly than I ever have before in my entire life.

(signed) _____

		Score	Grader
1. Boston Public	(20)	_____	_____
2. DArray Extended	(10)	_____	_____
3. The LexiconADT	(30)	_____	_____
4. Short Answers	(10)	_____	_____
Total	**(70)**	_____	

Problem 1: Boston Public—Where Every Day is a Fight (20 points)

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.

```
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.

Problem 2: Extending the DArray (10 points)

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)
{
```

Problem 3: The LexiconADT

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

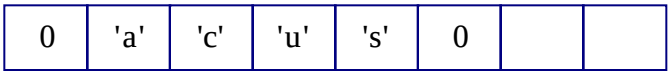
from the DArray storing the suffix, the remaining eight characters can potentially reside directly in the DArray.

- 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:

  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.
  Should be first byte store anything nonzero, then 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.
  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.

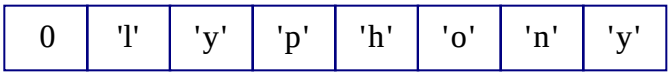
Here are a few examples:

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



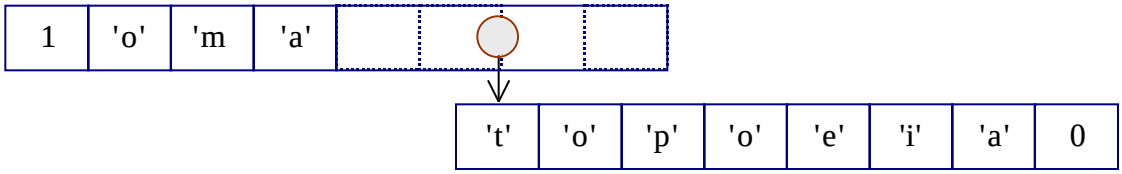
  The first bytes stores a zero, because the "acus" suffix can be fully stored in the remaining seven bytes. The leading 0 reminds the implementation that everything resides in the eight-byte chunk.

- 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 smaller routines `LexiconNew`.

```
/*
 * 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
{
```

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)
}

/*
 * 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
{
```

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;
}

/*
 * 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
{
```

Problem 4: Short Answers (10 points)

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)

- 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)

· 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)

Yours to tear out

```
// darray.h

DArray    ArrayNew(int elemSize, int numElemsToAllocate, ElemFreeFn fn);
    void  ArrayFree(DArray array);
     int  ArrayLength(DArray array);
    void *ArrayNth(DArray array, int n);
    void  ArrayAppend(DArray array, const void *newElem);
    void  ArrayInsertAt(DArray array, const void *newElem, int n);
    void  ArrayReplaceAt(DArray array, const void *newElem, int n);
    void  ArrayDeleteAt(DArray array, int n);
    void  ArraySort(DArray array, ArrayCompareFn comparator);
     int  ArraySearch(DArray array, const void *key, ArrayCompareFn comp,
                      int fromIndex, bool isSorted);
    void  ArrayMap(DArray array, ArrayMapFn fn, void *clientData);


// hashtable.h

HashTable TableNew(int elemSize, int nBuckets, TableHashFn hashFn,
                   TableCompareFn compFn, TableElemFreeFn freeFn);
   void   TableFree(HashTable table);
   void   TableEnter(HashTable table, const void *newElem);
   void  *TableLookup(HashTable table, const void *elemKey);
   void   TableMap(HashTable table, TableMapFn fn, void *clientData);
    int   TableCount(HashTable table);
```