CS107 Midterm Examination Solution

Solution 1: UTF-8 and Four-Byte Encodings [10 points]

In Assignment 1, you implemented a function to transform Unicode code points to UTF-8 byte arrays. The assignment itself limited the scope to encodings that are either one, two, or three bytes in length.

Some code points—those for hieroglyphics, emojis, and more esoteric mathematical symbols, to name a few—are encoded using **four** bytes. In particular, all Unicode code points from **`0x10000`** up through and including **`0x1FFFFF`** can be encoded as:

`11110--- 10------ 10------ 10------`

on the assumption that each of the hyphens in the above pattern can be either 0 or 1, independent of other hyphens.

For this problem, you will implement the **`codepoint`** function, which accepts an array of **`unsigned char`**s known to be of length 4 and returns the corresponding codepoint as an **`unsigned int`**. You should assume the byte at position 0 is the **lead** byte—the one that should start with **`11110`**—and the bytes at positions 1, 2, and 3 are the **continuation** bytes that should start with **`10`**.

a) [6 points] In the space below, complete the implementation of the **`codepoint`** function. For the moment, assume **`is_valid_lead`** and **`is_valid_continuation`** are fully functional and work as you'd expect. You may and should use whatever bitwise operations you'd like.

```
unsigned int codepoint(unsigned char bytes[]) {
   if (!is_valid_lead(bytes[0])) error(1, 0, "Byte 0 is malformed.");
   for (size_t i = 1; i < 4; i++) {
      if (!is_valid_continuation(bytes[i]))
      error(1, 0, "Byte %zu is malformed.", i); // %zu is placeholder for size_t
   }

   unsigned int code = 0;
   code |= (0x07 & bytes[0]) << 18;
   for (size_t i = 1; i < 4; i++) {
      code |= (0x3F & bytes[i]) << (6 * (3 - i));
   }
   return code;
}
```

b) [4 points, 2 and 2] Use this page to present implementations of **`is_valid_lead`** and **`is_valid_continuation`**, subject to the following constraints:

- Your implementation of **`is_valid_lead`** should return **`true`** if and only if the five most significant bits of the incoming byte are **`11110`**. Your implementation is limited to using **`&`**, **`|`**, hexadecimal/decimal/binary constants, **`==`**, and **`!=`**. Your implementation may not use **`<<`**, **`>>`**, or any other logical or arithmetic operators.
- Your implementation of **`is_valid_continuation`** should return **`true`** if and only if the two most significant bits of the incoming byte are **`10`**. Your implementation may use **`<<`**, **`>>`**, **`!=`**, **`==`**, and any constants you deem necessary, though no other bitwise, arithmetic, or relational operators may be used.

```
bool is_valid_lead(unsigned char byte) {
   return (byte & 0xF8) == 0xF0;
}

bool is_valid_continuation(unsigned char byte) {
   return (byte >> 6) == 0x02;
}
```

Here's a test harness, just for fun!

```
int main(int argc, char *argv[]) {
   unsigned char first[] = {0xf0, 0x90, 0x80, 0x80}; // for 🍴
   printf("Codepoint: 0x%x\n", codepoint(first));
   unsigned char nails[] = {0xf0, 0x9f, 0x92, 0x85}; // for 💅
   printf("Codepoint: 0x%x\n", codepoint(nails));
   unsigned char bear[] = {0xf0, 0x9f, 0xa7, 0xb8}; // for 🧸
   printf("Codepoint: 0x%x\n", codepoint(bear));
   return 0;
}
```

Problem 2: Constructing `main`'s `argv` [10 points]

Implement the **`parse`** function code to the following prototype:

```
size_t parse(char *cmd, char *argv[], size_t len);
```

which accepts a string of single space-delimited tokens, updates that string in place so that each token is terminated by a dedicated **`'\0'`**, and then stores the leading addresses of each token in the supplied array of **`char *`**'s. For instance, if the supplied **`cmd`** string at first looks like this (where that **`0`** is really a **`'\0'`**):

```
mywhich ls emacs cat0
```

then the implementation would update **`cmd`** in place to look like this

```
mywhich0ls0emacs0cat0
```

and store the addresses of the leading characters—that is, the addresses of **`mywhich`**'s **`'m'`**, **`ls`**'s **`'l'`**, **`emacs`**'s **`'e'`**, and **`cat`**'s **`'c'`**—in **`argv[0]`**, **`argv[1]`**, **`argv[2]`**, and **`argv[3]`**, respectively, and **`argv[4]`** would be set to **`NULL`**. In this scenario, our **`parse`** function would return a 4 so the client knows how many meaningful tokens were stored in the **`argv`** array.

For simplicity, we'll assume the tokens are comprised of lowercase letters delimited by single spaces and that there aren't any gratuitous spaces before the first token or after the last one. The number of tokens in **`cmd`** needn't be four but could be any nonnegative number. You should ensure at most **`len`** - **`1`** meaningful addresses get written in the **`argv`** array. If your implementation fills up **`argv`**, you can stop parsing and simply return **`len`** - **`1`**.

Your implementation **must use** of **`strspn`**, **`strcspn`**, **`strchr`**, and/or **`strstr`** to determine how long each token is. In particular, you may not effectively reimplement any of these four functions instead of using them directly. Your implementation must take care to write a **`NULL`** in the **`argv`** slot just beyond that slot storing the last token, and you must return the number of meaningful addresses stored in **`argv`**. Your function shouldn't dynamically allocate any memory, since all of the memory it needs already exists before **`parse`** is even called. The **`parse`** function you're implementing here is a reduced version of what the terminal does to prepare the argument vectors passed to **`main`** functions.

Present your implementation of **`parse`** on the next page.

```
/*
 * Arguments: cmd is a valid C string of single-space delimited tokens
 *            argv is the array of char *s that should be populated with
 *                 the leading addresses of each token
 *            len is the length of the supplied argv array
 * Return: the number of tokens in the supplied cmd string
 */
size_t parse(char *cmd, char *argv[], size_t len) {
    assert(len >= 1);
    size_t i = 0;
    while (i < len - 1) {
        argv[i] = cmd;
        cmd += strcspn(cmd, " ");
        i++;
        if (*cmd == '\0') break; // we're already at the end of the full cmd
        *cmd = '\0';
        cmd++;
    }
    argv[i] = NULL;
    return i;
}
```

Check out this full test harness I've set up for this problem right here:

https://cplayground.com/?p=cheetah-pony-sardine

Problem 3: Hashing Strings by Length [10 points]

Given an array of C strings (all guaranteed to be words in the English language), implement a function called **distribute** that dynamically allocates, populates, and ultimately returns an array of 45 records of the following type:

```
typedef struct {
   char *words;   // serialization of all words in this group
   size_t count;  // number of English words stored in this group
} group;
```

All words stored in any single group are of the same length. In particular, the 0th **group** will store English words of length 1, the 1th **group** will store English words of length 2, and in general, the kth **group** will store English words of length k + 1. Initially, all **words** and **count** fields are zeroed out as **NULL**'s and 0's, respectively. The empty string isn't a word in the English language, so we don't need a **group** for length 0 strings. And the longest word in the English language—pneumonoultramicroscopicsilicovolcanoconiosis, by the way—is 45 letters long. That's why we need 45 records.

Here's the prototype of the **distribute** function you'll implement:

```
group *distribute(const char *words[], size_t n);
```

This **distribute** function should traverse the supplied **words** array **exactly once** and populate the 0th slot of an array with copies of all strings of length 1, the 1th slot with all strings of length 2, and so forth, until all strings have been processed. You should ensure the memory reachable from each of the **words** fields is only as large as needed. Restated, **words** is initially **NULL** to reflect the absence of string but needs to be repeatedly reallocated to be just large enough to store the strings that have been copied there so far. If, say, there are a total of 112 strings of length 23 in the **words** array passed to **distribute**, then the **group** at index 22 will be reallocated a total of 112 times.

One caveat: Because all words in any single **group** are of the same length, the **words** fields within any given **group** should point to **single character array** capable of storing all words back-to-back, without any intervening **'\0'**s. If, for instance, a **group** stores seven words of length 6, **words** would address a character buffer of length 7 * 6 = 42 bytes. If, say, these seven **words** were **"eschew"**, **"dragon"**, **"pillow"**, **"ground"**, **"ragged"**, **"zephyr"**, and **"humble"**, then those 42 bytes of heap memory would be populated with:

```
eschewdragonpillowgroundraggedzephyrhumble
```

Your task is to implement the **distribute** function according to the above specification. You shouldn't worry about sorting the words, and you shouldn't worry about repeated words either. You can assume all words are of length 45 or less, and you needn't do any error checking or use **assert** anywhere. Present your implementation on the next page.

```
typedef struct {
   char *words;   // serialization of all words in this group
   size_t count;  // number of English words stored in this group
} group;

/*
 * Arguments: words is an array of C strings, all assumed to be
 *                  English words of length 45 or less
 *            n is the number of strings in the words array
 * Return: the base address of the 45 group records, fully
 *         initialized with all of the words as described above.
 */
group *distribute(const char *words[], size_t n) {
   group *groups = malloc(45 * sizeof(group));
   for (size_t i = 0; i < 45; i++) {
      groups[i].words = NULL;
      groups[i].count = 0;
   }

   for (size_t i = 0; i < n; i++) {
      size_t len = strlen(words[i]);
      size_t slot = len - 1;
      groups[slot].words = realloc(groups[slot].words,
                                   (groups[slot].count + 1) * len);
      strncpy(groups[slot].words + groups[slot].count * len,
              words[i], len);
      groups[slot].count++;
   }

   return groups;
}
```

Here's the test harness I created to make sure everything worked as intended:

https://cplayground.com/?p=elephant-elephant-siamang

Problem 4: Generics and Sorting [10 points]

Implement the generic function **`is_sorted`** to return **`true`** if and only if the supplied array is sorted from low to high. If, for instance, our **`is_sorted`** function was called on the following integer array of length 8, we would like **`is_sorted`** to return **`true`**, since no two elements are out of order.

14	26	31	31	46	60	71	81

If that 46 at index 4 were a 26 instead, then we'd want **`is_sorted`** to return **`false`**.

Of course, we want our function to work on arrays of **any** type. With that in mind, present your implementation of a generic **`is_sorted`** function that accepts the base address of an array, the number of elements in that array, the size of each array element, and a generic comparison function that returns a negative, zero, or positive value to signal that the first argument is less than, equal to, or greater than the second argument, respectively.

a) [7 points] Implement the generic **`is_sorted`** function that returns that returns **`true`** if and only if the array is sorted according to the supplied comparison function.

```
/*
 * Arguments: base is the base address of the array
 *            num_elems is the logical number of elements in the array
 *            elem_size is the size of each array element, in bytes
 *            cmpfn is a generic comparison function that returns
 *                  0 if the two addressed elements are identical,
 *                  a negative number if the first element is less than
 *                  the second, and a positive number otherwise.
 */
bool is_sorted(void *base, size_t num_elems, size_t elem_size,
               int (*cmpfn)(void *, void *)) {
    for (size_t i = 0; i < num_elems - 1; i++) {
        void *first = (char *) base + i * elem_size;
        void *second = (char *) first + elem_size;
        if (cmpfn(first, second) > 0) return false;
    }
    return true;
}
```

Given a fully functional **is_sorted**, we should be able to verify that an array of C strings is sorted alphabetically (or formally, lexicographically). For this question, you're to implement a **compare_strings** comparison function so that the following program passes through the **assert** statements without crashing, prints **Everything works!** on a line by itself, and returns from **main** without incident.

```
int main(int argc, char *argv[]) {
   char *fruits[] = {"apple", "banana", "cranberry", "date"};
   assert(is_sorted(fruits, 4, sizeof(char *), compare_strings));
   char *veggies[] = {"carrot", "potato", "zucchini", "artichoke"};
   assert(!is_sorted(veggies, 4, sizeof(char *), compare_strings));
   printf("Everything works!\n");
   return 0;
}
```

b) [3 points] Present your implementation of **compare_strings** to return a negative number, a zero, or a positive number to signal that the C string reachable from the first argument is less than, the same as, or greater than the C string reachable from the second argument, respectively. Your implementation should ultimately rely on a call to **strcmp** to actually do the comparison.

```
int compare_strings(void *first, void *second) {
   return strcmp(*(char **)first, *(char **)second);
}
```

Feel free to inspect this fully functional test framework set up for this problem:

https://cplayground.com/?p=chameleon-coati-kudu