15-122: Principles of Imperative Computation

Lab B: Once you C1 you C them all

Shyam Raghavan

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with your neighbors as you work through this lab!

Setup: Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab-cbst .
% cd lab-cbst
```

Grading: Finish (2.a)-(2.d) for 2 points. Finish through (3.a) for 3 points.

Modified binary search trees

In this lab, we'll translate a simplified binary search tree implementation into a C interface and C implementation. This interface for binary search trees is more set-like than the one we discussed in class: we're not allowed to add an element if an equivalent element is already in the set, and instead of lookup returning the stored element, it just returns true or false.

You can compile the existing code with `cc0 -d -x bst.c1 bst-test.c1`

Translating to C

The first tasks will just get you running C code.

(2.a) Take the interface portion of bst.c1 and put it in a file `bst.h`. The first lines in this file should be

```
1 #include <stdbool.h>
2 #ifndef __BST_H__
3 #define __BST_H__
```

and the last line should be

```
1 #endif
```

(2.b) Take the implementation of `bst.c1` and put it in a file `bst.c`. In order to include the allocation library, C0-style contracts, and the BST interface, this file should begin with these lines:

```
1 #include "lib/xalloc.h"
2 #include "lib/contracts.h"
3 #include <stdlib.h>
4 #include "bst.h"
```

(2.c) Translate the test code in `bst-test.c1` as a file `bst-test.c`. Because this code uses assert statements, standard input/output, allocation, and the BST interface, you'll need to begin the file with these lines:

```
1 #include <assert.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include "lib/xalloc.h"
5 #include "lib/contracts.h"
6 #include "bst.h"
```

(2.d) Correctly handle allocation: in both `bst.c` and `bst-test.c`, change code like `alloc(list)` or `alloc(int)` to the comparable C code `xmalloc(sizeof(list))` or `xmalloc(sizeof(int))`. In `bst-test.c`, you'll also need to change the `print` to `printf`.

You can now compile your code by using the provided Makefile:

```
% make
% ./bst-test-d
Test passed!
% ./bst-test
Test passed!
```

The file `bst-test-d` was compiled with contracts, and `bst-test` was compiled without them. This is easier than what you'd need to write without the Makefile:

```
% gcc -Wall -Wextra -Werror -Wshadow -std=c99 -pedantic -g -DDEBUG
    -o bst-test-d lib/*.c bst.c
% gcc -Wall -Wextra -Werror -Wshadow -std=c99 -pedantic -g
    -o bst-test lib/*.c bst.c
```

(2.e) Translate the preconditions and postconditions from the C1 code into C macro-language `REQUIRES` and `ENSURES` statements. In C, `REQUIRES` statements come at the beginning of a function but after the opening curly-brace, and `ENSURES` statements come at the end, right before the `return`. Instead of using `\result`, we just use the expression that is about to be returned.

(2.f) Modify your test cases to add an element to the BST when that element already exists in the set. Make sure that you get a precondition violation using `bst-test-d`, but that the code just runs with `bst-test`. (Then remove that test.)

Memory leaks

As we learned in lecture, C doesn't do memory management like was done in C0. As a result, we need to use the `free` function to free memory for use by the operating system.

We can't free everything, though! As the implementation (because our data structure is generic), we have no way of knowing how to free the data provided by the client. As such, our freeing function takes a function pointer from the client that frees elements (or is `NULL` in the case of the client wishing to not free the memory).

(3.a) Extend the BST interface with a new type and a new function:

```
1 typedef void elem_free_fn(void* x);
2 void bst_free(bst_t B, elem_free_fn* F)
3   /*@requires B != NULL; @*/ ;
```

Implement this function that frees all the BST's internal memory, and that, if `F` is not `NULL`, also runs the provided function to free all the `void*` pointers stored in the tree's `data` fields.