15-122 : Principles of Imperative Computation, Spring 2014

Written Homework 11

Due before class: Thursday, April 24, 2014

Name: ____

Andrew ID: ____

Recitation: ____

The written portion of the last homework this semester will give you some practice working with more C programming issues, the C0 virtual machine and tries. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

Question	Points	Score
1	7	
2	4	
3	8	
Total:	19	

You must do this assignment in one of two ways and bring the stapled printout to the handin box on Thursday:

1) Write your answers *neatly* on a printout of this PDF.

2) Use the TeX template at `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/theory11.tgz`

1. **Typecasting and Function Pointers in C**

Suppose that we are working with the expected implementation-defined implementation of unsigned and signed (2's complement) `short` (16 bits, two bytes) and `int` (32 bits, four bytes).

(3) (a) We begin with the following declarations:

```
short w = -15;
unsigned short x = 65521;
int y = -65521;
```

Fill in the table below. In the third column, always use four hex digits to represent a `short`, and eight hex digits to represent an `int`. You might find these numbers useful: $2^{16} = 65536$ and $2^{32} = 4294967296$.

Solution:

C expression	Decimal value	Hexadecimal
`w`	-15	`0xFFF1`
`(unsigned short)w`	65521	`0xFFF1`
`(int)w`	-15	`0xFFFFFFF1`
`x`	65521	
`(int)x`		
`(int)(short)x`		
`y`	-65521	
`(unsigned int)y`		

(2) (b) Consider the following C definition for the factorial function:

```
int factorial(int n)
{
    REQUIRES (n >= 0);
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Use `typedef` to define a C type named `int2int` that represents a function pointer that requires an int as its parameter and returns an int as its return type.

Solution:

```
typedef ____________________________________________________;
```

Let the variable `f` be of type `int2int`. (That is, `f` is a function pointer to a function that has one parameter of type `int` and returns a result of type `int`.) Show how to initialize `f` with the address of the `factorial` function given above using the address-of operator.

Solution:

```
int2int f = ___________________________________________________;
```

Write a C instruction that prints out 10! using the variable `f` defined above. Use an explicit derefencing operation on `f` to get to the factorial function.

Solution:

```
printf( "10! = %d\n", _________________________________________);
```

(2) (c) Suppose we have a (signed) `char` array of length 4 and we want to store that array in a single (4-byte) `int` by storing the `char` array `{1, 2, 3, 4}`, for example, as `0x01020304`. Remember that `char` is an integer type in C.

Write a C function that takes a length-4 `char` array named `F` and condenses it into a single `int` as outlined above. Do not cast directly between signed and unsigned types of different sizes, and make sure your solution works for `char` arrays containing negative values.

Your solution should be clear and straightforward; convoluted code will not receive full credit.

Solution:

```
int condense(char *F) {

}
```

2. **C0VM**

Consider the following code that populates a structure with values

```
typedef struct gap_buffer* gapbuf;
struct gap_buffer {
    int limit;     /* limit > 0 */
    char[] buffer; /* \length(buffer) == limit */
    int gap_start; /* 0 <= gap_start */
    int gap_end;   /* gap_start <= gap_end <= limit */
};

int main() {
  gapbuf gb = alloc(struct gap_buffer);
  gb->limit = 65536;
  gb->buffer = alloc_array(char, 65536);
  gb->gap_start = 48112;
  gb->gap_end = gb->gap_start;
  return 1;
}
```

(2) (a) Fill in the missing instructions in the following bytecode that corresponds to the above C0 code. You don't need to fill in the hex opcodes. Just the instruction name and its argument(s) in decimal is sufficient. Be careful, the answers may or may not match the bytecode output generated compiling the C0 code directly.

Solution:

```
C0 C0 FF EE       # magic number
00 09             # version 4, arch = 1 (64 bits)

00 03             # int pool count
# int pool
00 00 BB F0
00 01 00 00
00 01 00 00

00 00             # string pool total size
# string pool

00 01             # function count
# function_pool

#<main>
00 00             # number of arguments = 0
00 01             # number of local variables = 1
```

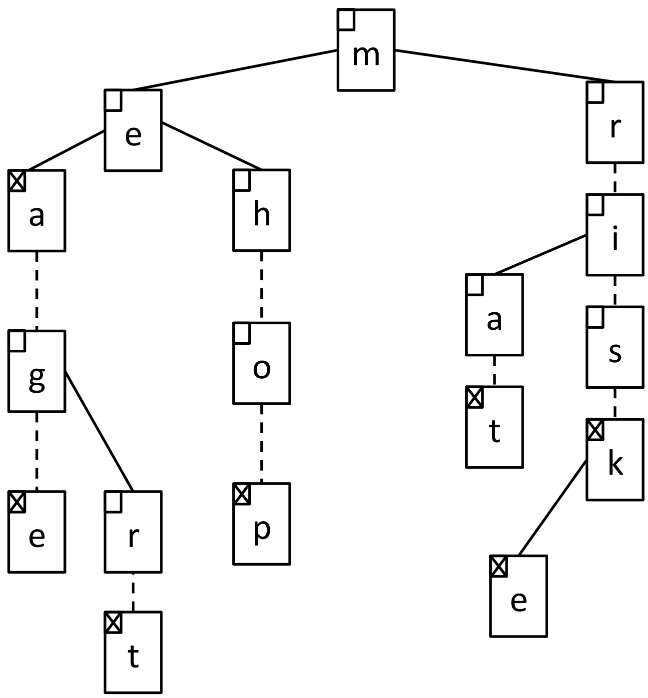
```
19 00 2B              # code length = 43 bytes
20 BB 18    # new 24           # alloc(struct gap_buffer)
21 _______  # ________         # gb = alloc(struct gap_buffer);
22 15 00    # vload 0          # gb
23 62 00    # aaddf 0          # &gb->limit
24 ________ # ________         # 65536
25 4E       # imstore          # gb->limit = 65536;
26 15 00    # vload 0          # gb
27 62 08    # aaddf 8          # &gb->buffer
28 ________ # _________        # 65536
29 BC 01    # newarray 1       # alloc_array(char, 65536)
30 4F       # amstore          # gb->buffer = alloc_array(char, 65536);
31 15 00    # vload 0          # gb
32 62 10    # aaddf 16         # &gb->gap_start
33 13 00 00 # ildc 0           # 48112
34 4E       # imstore          # gb->gap_start = 48112;
35 15 00    # vload 0          # gb
36 62 14    # aaddf 20         # &gb->gap_end
37 15 00    # vload 0          # gb
38 _______  # _________        # &gb->gap_start
39 2E       # imload           # gb->gap_start
40 4E       # imstore          # gb->gap_end = gb->gap_start;
41 10 01    # bipush 1         # 1
42 B0       # return           #
43
44 00 00              # native count
45 # native pool
```

(2) (b) After executing line 26 of the byte code, assume that the only value in the operand stack is `0x3fff0000`. Draw the four operand stack states after each of lines 26-29 is executed. The elements in your stack should be 32-bit hexadecimal numbers. Assume that `alloc_array` returns `0x80000000`.

Solution:

3. **Ternary Search Tries**

Consider the TST shown below.



As in the lecture notes, the dotted lines connect a node to its `middle` child, and solid lines connect a node to its `left` and `right` children. An X in the top left indicates that this node ends a valid word. There could be a link to a corresponding value, like a word definition, for example.

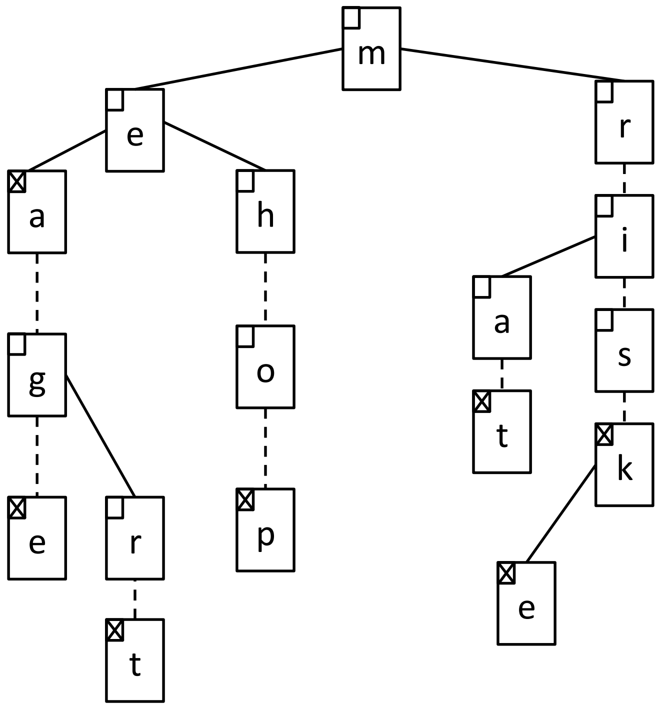
The lecture notes and accompanying code describe a desirable invariant of TSTs: that if the middle child is `NULL`, the node has to end a valid word. This TST does not have this invariant due to the topmost `m` and `e` nodes, but it is not a necessary invariant for safety or correctness. (The insertion and lookup algorithms work even without that invariant.)

(2) (a) List all of the valid words stored in the TST above, in alphabetical order.

Solution:

(3) (b) Add the words `me`, `rake`, `hope`, `hot`, `top`, and `act` to the TST given on the previous page, one at a time, in the order given.

Solution:



(3) (c) For this question, review the published code for tries from lecture.

It is possible to implement `trie_lookup` as an iterative function rather than a recursive one. Fill in the blanks so that the function shown below correctly implements lookup in a TST.

The lines involving the variables `lower` and `upper` are used only to prove that the loop invariant (written in the incorrect location as an assertion in the code below) is preserved. You should not use `lower` or `upper` when filling in the blanks.

Solution:

```
elem trie_lookup(trie TR, char *s) {
    REQUIRES(is_trie(TR));
    REQUIRES(s != NULL);
    tnode *T = TR->root;
    int charmin = 0;
    int charmax = (int)CHAR_MAX + 1;
    int lower = charmin;
    int upper = charmax;

    while (T != ___________________________________) {
       ASSERT(is_tnode(T, lower, upper)); // Loop invariant
       if (*s == T->c) {
           if (*(s+1) == '\0') {

               return ___________________________________________;
           } else {
               lower = charmin;
               upper = charmax;
               s++;

               T = ______________________________________________;

       } else if (_____________________________________________) {
           lower = T->c;

           T = __________________________________________________;

       } else {
           upper = T->c;

           T = __________________________________________________;
       }
    }
    return NULL;
}
```