Exam 3

15-122 Principles of Imperative Computation, Summer 2011
William Lovas

June 24, 2011

Name: **Sample Solution** Andrew ID: **wlovas**

Instructions

- This exam is closed-book with one double-sided sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 4 problems.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.
- Consider writing out programs or diagrams on scratch paper first.
- And most importantly,

DON'T PANIC!

	Integers	Generic Data	Tries	BDDs	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score	**30**	**40**	**40**	**40**	**150**
Max	30	40	40	40	150
Grader	wjl	wjl	wjl	wjl	

1 Integers (30 points)

Task 1 (15 pts). For each of the following expressions in C, indicate whether they are always true. If not, give a counterexample and state whether the counterexample is guaranteed to be false or undefined in C. You should not assume any particular size for ints and state your counterexamples in terms of `INT_MAX`, `INT_MIN`, and `UINT_MAX`. Assume we are in the scope of declarations

```
int x = ...;
unsigned int u = ...;
```

so x and u are properly initialized to arbitrary values. We have filled in one row for you.

C expression	Always true?	If no, counterexample	False or undefined
`x + 1 == 1 + x`	**no**	`x == INT_MAX`	**undefined**
`x <= 0 || -x < 0`	**yes**		
`x >= 0 || -x > 0`	**no**	`x == INT_MIN`	**undefined**
`u + 1 == 1 + u`	**yes**		
`u + 1 > u`	**no**	`u == UINT_MAX`	**false**
`u - 1 == -(1 - u)`	**yes**		

Task 2 (10 pts). Write a C function `safe_to_add` which returns `true` if adding the two arguments would be well-defined and `false` otherwise.

```
bool safe_to_add(int x, int y) {
  if (x > 0 && y > 0 && x > INT_MAX - y) return false;
  if (x < 0 && y < 0 && x < INT_MIN - y) return false;
  return true;
}
```

(Continued)

Task 3 (5 pts). The following function attempts to build an `int` out of two `short`s by putting the bit pattern of the first `short` next to the bit pattern of the second `short`. It makes the following assumptions about the C implementation:

- `sizeof(int) == 4`,
- `sizeof(short) == 2`,
- bytes are 8 bits, and
- all signed integer types are represented using two's complement.

Even under these assumptions, the function has a bug. Explain briefly both the problem and the fix.

```
int build_int(short s1, short s2) {

    int res = 0;

    res = res | (int) s1;

    res = res << 16;

    res = res | (int) s2;

    return res;

}
```

The problem is that in the line `res = res | (int) s2;`, the cast from `short` to `int` sign-extends, since `short` is signed by default; this sign-extension will in some cases make the high 16 bits equal to 1, overwriting the bits of `res` assigned to `s1`. The fix is to first cast `s2` to an unsigned type before casting to a larger-sized integer: `res = res | (int) (unsigned short) s2`.

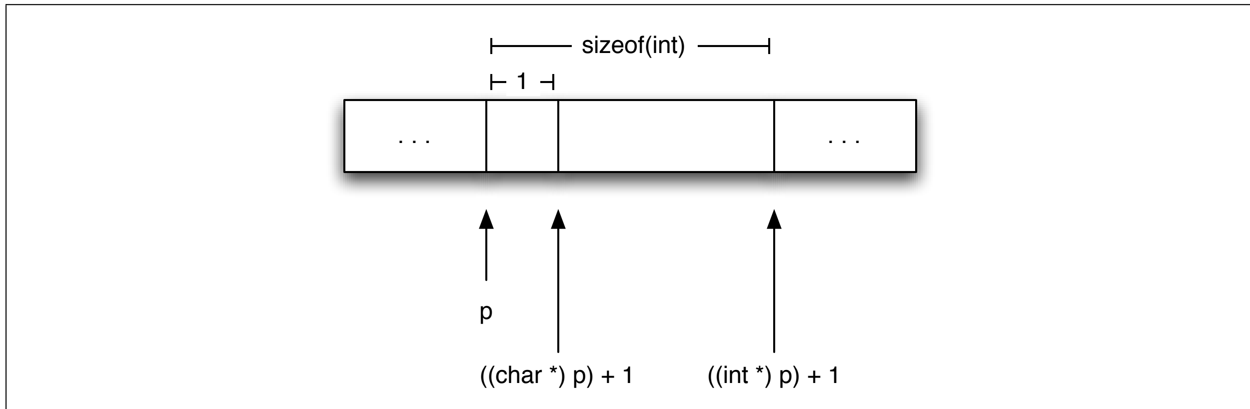
2 Generic Data (40 points)

A pointer to data of unknown type may be represented in C using the type `void *`, but any such pointer must be cast appropriately before it can be used.

Task 1 (5 pts). Suppose we declare a generic pointer

```
void *p;
```

Draw a picture demonstrating the difference between `((char *) p) + 1` and `((int *) p) + 1`.



(Continued)

Consider binary search trees implemented with generic elements.

```
struct tree {
    struct tree *left;
    void *data;
    struct tree *right;
    /* ... other fields ... */
}

typedef struct tree * tree;
```

An *in-order traversal* of a binary tree is one that first visits all elements in the left subtree of a node, then visits the node itself, and finally visits all elements in the right subtree of a node.

Task 2 (10 pts). Complete the `traverse` function which performs an in-order traversal on a binary search tree containing generic data. The second argument, `visit`, should be a pointer to a function taking generic data and returning nothing.

```
void traverse(tree T, _____void ( * _____ visit __)( void * x )_____ ) {

    if (T == NULL) return;

    traverse(T->left, visit);
    (*visit)(T->data);
    traverse(T->right, visit);

}
```

(Continued)

Suppose we wish to store a roster of students in a binary search tree.

```
struct student {
    char *name;
    int id;
    char *major;
};

typedef struct student * student;
```

Task 3 (5 pts). Write a `print_student` function suitable for passing to `traverse` (i.e., of the appropriate type for a `visit` function) that prints a `student` in the following format:

```
Name: William Lovas, ID: 12345, Major: Computer Science
```

```
void print_student(void *x) {
    REQUIRES(x != NULL);
    student S = (student) x;
    printf("Name: %s, ID: %d, Major: %s\n", S->name, S->id, S->major);
}
```

Task 4 (5 pts). Suppose our roster is stored in a tree:

```
tree roster = ...;
```

Show how to call `traverse` to print all of the students in the roster using your `print_student` function.

```
traverse(roster, &print_student);
```

(Continued)

Now suppose we wish to store pointers to `ints` in a binary search tree.

Task 5 (5 pts). Write a `print_intptr` function suitable for passing to `traverse` (i.e., of the appropriate type for a `visit` function) that prints the pointed-to integer.

```
void print_intptr(void *x) {
    REQUIRES(x != NULL);
    printf("%d\n", *((int *) x));
}
```

Task 6 (10 pts). Suppose we have an array `A` of `n` distinct integers. Using `traverse` and `print_intptr`, write a function that prints the integers in sorted order *without sorting the array `A` itself*. You may also assume the following `tree` functions:

```
tree itree_new();                        /* construct a new BST storing int pointers */
tree itree_insert(tree T, void *data);   /* insert data into T, returning a new tree */
void itree_free(tree T);                 /* free the storage associated with tree T */
```

```
void print_in_order(int *A, int n) {
    tree T = itree_new();

    for (int i = 0; i < n; i++)
        T = itree_insert(T, &A[i]);

    traverse(T, &print_intptr);

    itree_free(T);
}
```

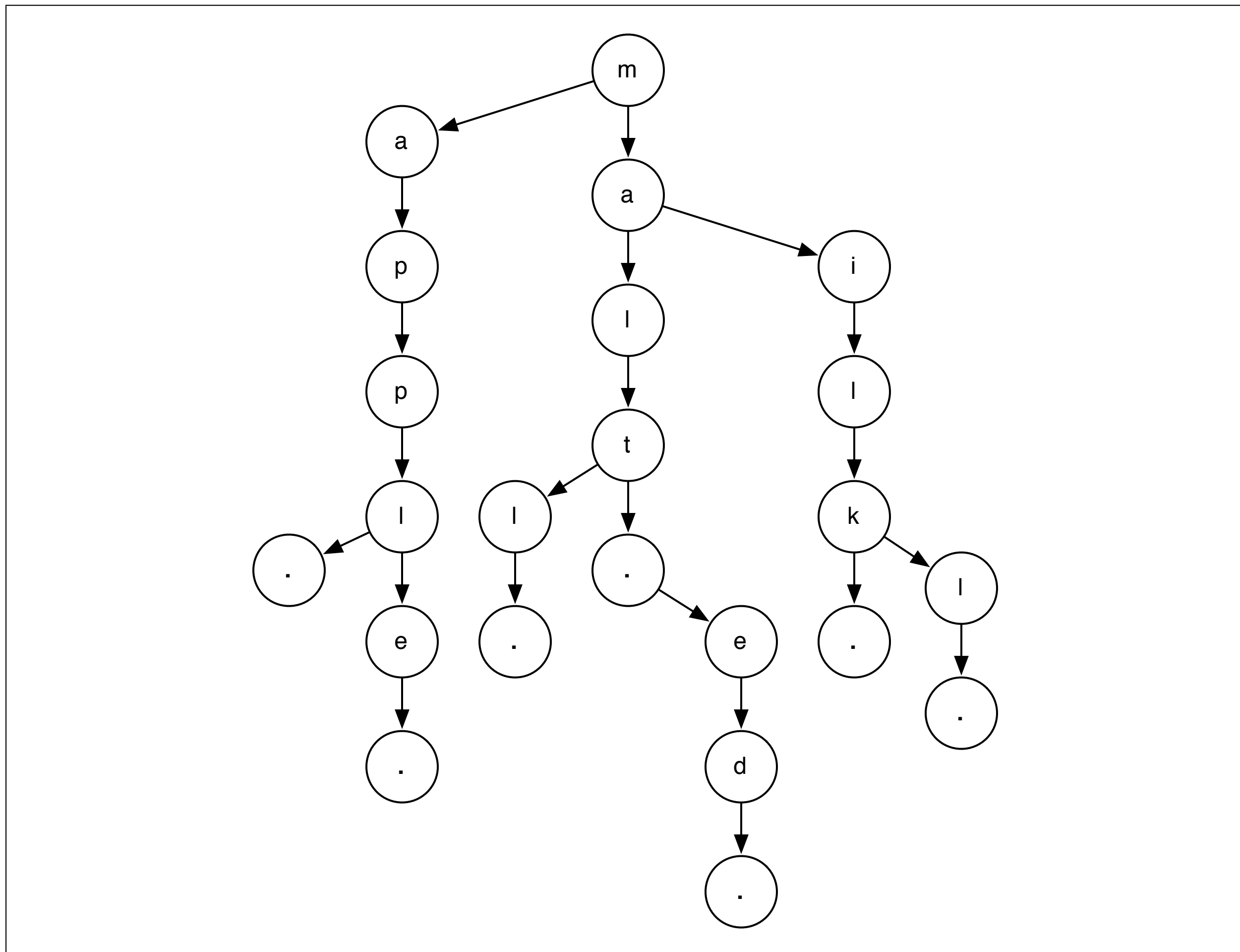
3 Tries (40 points)

Tries are an efficient data structure for storing sets of strings. We saw in class the *ternary search trie*, a data structure combining ideas from ordinary multi-way tries and binary search trees.

Task 1 (15 pts). Construct the ternary search trie (TST) that results from inserting the following strings into an initially empty TST in order:

malt, milk, mall, mill, apple, app, malted

You only need to show the final trie, but if you show intermediate steps we may more easily assign partial credit.



(Continued)

Task 2 (10 pts). Describe an algorithm for checking whether any string with a given prefix is stored in a trie. For example, given the trie from Task 1, it should return true for the prefix *"ma"*, but false for the prefix *"mu"*. You do not need to write code—though you may if you wish—but your description should be precise enough that it would be clear how to write the code.

> Keep track of the remainder of the prefix that you haven't found yet, initially the entire prefix. Continually try to match the first letter of the remainder of the prefix with the key letter at the current trie: if the key letter is too large, move to the left subtrie; if the key letter is too small, move to the right subtrie; and if the key letter is equal to the first letter of the remainder of the prefix, move to the middle subtrie and consume the first letter of the remainder of the prefix.
>
> If you ever completely consume the prefix, return true, since there must be a word in the trie beginning with that prefix. If you ever encounter a `NULL` trie, return false, since there must be no word beginning with the given prefix in the trie: if there were, it would be contained in this `NULL` trie.

Task 3 (5 pts). What is the asymptotic complexity of your algorithm? Explain briefly.

> The algorithm runs in $O(k)$ time, where k is the length of the prefix. For each character in the prefix, it can traverse at most 27 levels of the trie, one for each possible alphabet character, including the terminating period ".".

(Continued)

Task 4 (10 pts). Describe how you would extend your algorithm above to collect *all* strings that begin with a certain prefix (e.g., to perform UNIX-style tab completion or predictive text input).

Continue from the place where the previous algorithm left off: that subtrie contains all strings beginning with the given prefix.

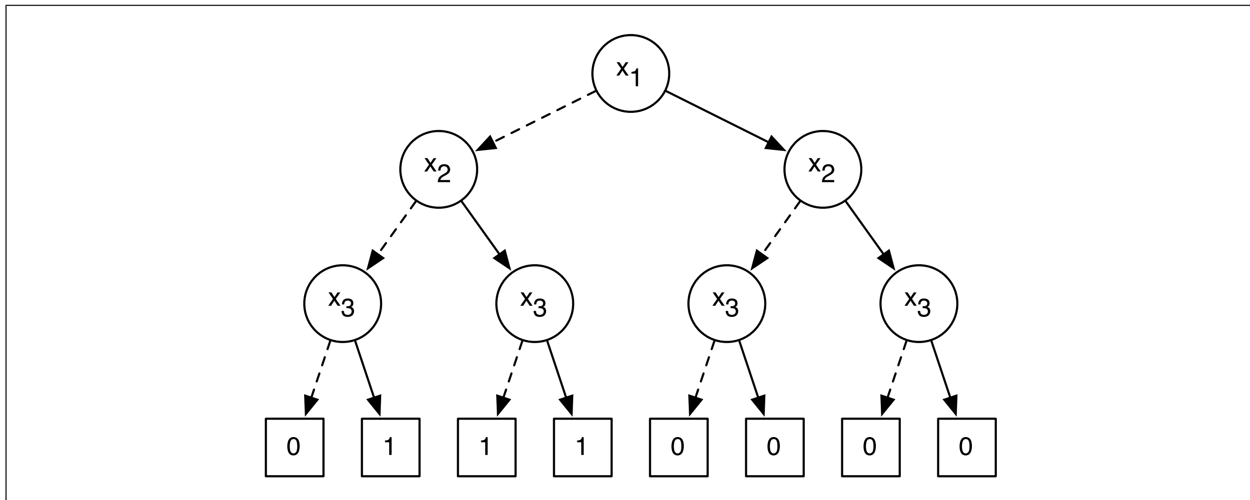
From there, perform an in-order traversal of the trie, recursively collecting all the words in the left subtrie, then the middle subtrie, and finally the right subtrie. When collecting words from the middle subtrie, prepend the current key letter to each one.

4 Binary Decision Diagrams (40 points)

An important operation in the construction of digital circuits is the *nor* operation,[1] which inputs booleans x_1 and x_2 and outputs $\neg(x_1 \vee x_2)$. We write $\mathbf{nor}(x_1, x_2)$. In C, this might be defined as

```
bool nor(bool x1, bool x2) {
  return !(x1 || x2);
}
```

Task 1 (15 pts). Construct an ordered binary decision tree (BDT, i.e., without sharing) for the formula $\mathbf{nor}(x_1, \mathbf{nor}(x_2, x_3))$, where the variables are tested in the order x_1, x_2, x_3.



(Continued)

[1]The first embedded system, the Apollo Guidance Computer, was built exclusively out of NOR gates.

Task 2 (10 pts). Ordered binary decision diagrams (OBDDs) are like binary decision trees that may exploit sharing. State the two conditions required for a BDD to be *fully reduced* (i.e., to be an ROBDD), explaining each briefly.

1. **Irredundancy:** The low and high successor of every node must be distinct.

2. **Uniqueness:** No two nodes test the same variable and have the same successors.

Task 3 (15 pts). Convert your binary decision tree from Task 1 into a fully-reduced ROBDD, where the variables are tested in the same order. You only need to show the final ROBDD, but if you show intermediate steps we may more easily assign partial credit.