

Exam 3

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

June 24, 2011

Name:

Andrew ID:

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					
Max	30	40	40	40	150
Grader					

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
<code>x + 1 == 1 + x</code>	no	<code>x == INT_MAX</code>	undefined
<code>x <= 0 -x < 0</code>			
<code>x >= 0 -x > 0</code>			
<code>u + 1 == 1 + u</code>			
<code>u + 1 > u</code>			
<code>u - 1 == -(1 - u)</code>			

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) {
```

```
}
```

(Continued)

Task 3 (5 pts). The following function attempts to build an `int` out of two `shorts` 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;  
  
}
```

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, _____ 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
```

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.

(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.

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) {
```

```
}
```

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.

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

(Continued)

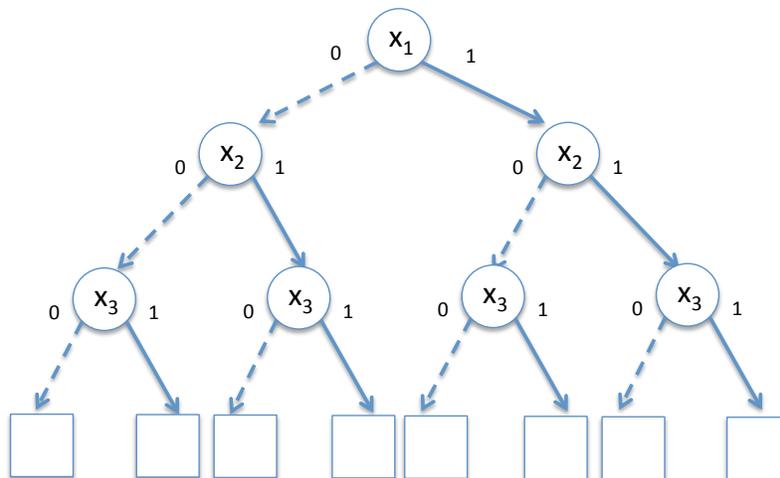
Task 4 (10 pts). Describe how you would extend your algorithm above to collect *all* strings that begin with a certain prefix (e.g., to perform UNIX-style tab completion or predictive text input).

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 $\text{nor}(x_1, x_2)$. 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 $\text{nor}(x_1, \text{nor}(x_2, x_3))$, where the variables are tested in the order x_1, x_2, x_3 .



(Continued)

¹The 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.

2.

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.