Final Exam

15-122 Principles of Imperative Computation
André Platzer  Rob Simmons
May 13, 2013

Name:  ____________  
Andrew ID:  ____________  
Recitation:  ____________  

Instructions

• This exam is closed-book with one sheet of notes permitted.
• You have 3 hours to complete the exam.
• There are 7 problems on 16 pages.
• 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.
• You can assume the presence of #use <util> for C0 programs throughout the exam.
• All C programs can be assumed to #include the usual header files stdbool.h, stdint.h, limits.h, xalloc.h, contracts.h, and stdio.h

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity Square</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Heaps</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Tries</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Graphs</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Linked Lists</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Unicorn Office</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Safety in C</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>250</td>
<td></td>
</tr>
</tbody>
</table>
1 Complexity Square (45 points)

**Task 1** Fill in how the complexity class in the row relates to the complexity class in the column of the table below. Use the following relations:

<table>
<thead>
<tr>
<th>write</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊂</td>
<td>row is strict subset of column</td>
</tr>
<tr>
<td>⊆</td>
<td>row is subset of or equal to column</td>
</tr>
<tr>
<td>=</td>
<td>row is equal to column</td>
</tr>
<tr>
<td>⊃</td>
<td>row is strict superset of column</td>
</tr>
<tr>
<td>⊇</td>
<td>row is superset of or equal to column</td>
</tr>
</tbody>
</table>

For example, if \(O(f(n))\) is the complexity class in row \(i\) and \(O(g(n))\) is the complexity class in column \(j\). Then the cell at row \(i\) and column \(j\) should indicate “\(\subset\)’ if, indeed \(O(f(n)) \subset O(g(n))\). Use the most informative relation (e.g., \(\subset\) is more informative than \(\subseteq\) and \(=\) is more informative than \(\subseteq\)).

You will receive positive points for correct answers but **negative points** if you give an answer that is wrong. If you are unsure about one entry you may prefer to leave it empty. If you score a negative total for this task, however, you will receive zero points instead.

<table>
<thead>
<tr>
<th>row relation to column</th>
<th>(O(n))</th>
<th>(O(n^2 - 2n))</th>
<th>(O((cn)^2))</th>
<th>(O(n(\log n)^2))</th>
<th>(O(c \cdot 2^n))</th>
<th>(O(2^{cn}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(n))</td>
<td>=</td>
<td>⊂</td>
<td>⊂</td>
<td>⊂</td>
<td>⊂</td>
<td>⊂</td>
</tr>
<tr>
<td>(O(n^2 - 2n))</td>
<td>⊃</td>
<td>=</td>
<td>=</td>
<td>⊃</td>
<td>⊂</td>
<td>⊂</td>
</tr>
<tr>
<td>(O((cn)^2))</td>
<td>⊃</td>
<td>=</td>
<td>=</td>
<td>⊃</td>
<td>⊂</td>
<td>⊂</td>
</tr>
<tr>
<td>(O(n(\log n)^2))</td>
<td>⊃</td>
<td>⊂</td>
<td>⊂</td>
<td>=</td>
<td>⊂</td>
<td>⊂</td>
</tr>
<tr>
<td>(O(c \cdot 2^n))</td>
<td>⊃</td>
<td>⊃</td>
<td>⊃</td>
<td>⊃</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>(O(2^{cn}))</td>
<td>⊃</td>
<td>⊃</td>
<td>⊃</td>
<td>⊃</td>
<td>⊃</td>
<td>=</td>
</tr>
</tbody>
</table>

where \(c > 1\) can be any integer constant

**Task 2** You have a program \(\text{alg1}(\text{int}[] \ A)\) that runs in \(O(n)\) time and an program \(\text{alg2}(\text{int}[] \ A)\) that runs in \(O(n^2)\) time, where \(n\) is the length of the array.

For \(n = 10,000\), we experimentally determined that \(\text{alg1}\) takes 1 second and \(\text{alg2}\) takes 0.1 seconds. What are the approximate running times you would expect for arrays of size:

a. \(\text{alg1}\) for \(n = 10,000\) \hspace{1cm} a. 1s
b. \(\text{alg1}\) for \(n = 40,000\) \hspace{1cm} b. 4s
c. \(\text{alg1}\) for \(n = 80,000\) \hspace{1cm} c. 8s
d. \(\text{alg2}\) for \(n = 10,000\) \hspace{1cm} d. 0.1s
e. \(\text{alg2}\) for \(n = 40,000\) \hspace{1cm} e. 1.6s
f. \(\text{alg2}\) for \(n = 80,000\) \hspace{1cm} f. 6.4s
2 Heaps (35 points)
Recall that we used heaps to implement priority queues. A heap is a binary tree satisfying
the ordering invariant that the key of a node is less or equal to the keys of its children. Heaps
also satisfy the shape invariant, i.e., that we fill the tree level by level, from left to right. We
implemented the heap as an array, where index 0 is ignored, index 1 holds the root element
and for a node at index $i$ we obtain its left child as $2i$, its right child as $2i + 1$, and its parent
as $i/2$. We use the following C0 header struct to represent heaps.

```c
struct heap_header {
    int limit; /* limit = capacity+1 */
    int next; /* 1 <= next && next <= limit */
    int[] data; /* \length(data) == limit */
};
typedef struct heap_header* heap;

bool is_heap(struct heap_header* H);
// this function checks the ordering and structure invariant of the heap.
```

Task 1 The following shows an array representing the priority queue with index 0 unused
and with element 10 at the root:

```
| 10 | 12 | 15 | 20 | 30 | 18 | 22 | 25 | 40 | 50 |
```

Draw the binary heap represented by this array.

```
  10
  / \ \\
 /   \\
/     \\
/       \\
/         \\
/           \\
/             \\
12 15
/   \\
/     \\
/       \\
/         \\
/             \\
/               \\
20 30 18 22
/   \\
/     \\
25 40 50
```

Task 2 After deleting one key from the priority queue, what are the contents of the array?

```
| 12 | 20 | 15 | 25 | 30 | 18 | 22 | 50 | 40 | X |
```
Task 3  Write a function that increases the priority of the element that is stored at index i in the heap to the new priority new_pri.

```c
void heap_raise_priority(heap H, int i, int new_pri)
//@requires is_heap(H);
//@requires new_pri < H->data[i];
//@ensures is_heap(H);
{
    H->data[i] = new_pri;
    while(i > 1 && H->data[i/2] > H->data[i])
    {//@loop_invariant is_heap_except_up(H, i);
        int temp = H->data[i/2];
        H->data[i/2] = H->data[i];
        H->data[i] = temp;
        i = i/2;
    }
}
```
3 Tries (25 points)

In this problem we look at tries used to store words and prefixes as implemented in class. We use the following representation of tries in C0:

```c
struct trie {
    bool isword;
    char data;
    trie center;
    trie left;
    trie right;
}
typedef struct trie* trie;
```

In this structure, all words that are in the center trie have data as a prefix. The left trie holds all words starting with a letter less than data and right tries hold all those words starting with a letter alphabetically higher than data.

The boolean parameters isword determines whether this node represents a complete word or not. The following figure shows an example of this tree. Nodes whose top left corner is marked with x represent a valid word.

![Trie Diagram]

**Task 1** Write down all the valid words that are in the above trie.

dc, dpr, eg, egy, ev, hi, hia, py, r, ta, v, vin
Task 2 Write a function in C0 that counts all words in the given trie. You can assume that your function is called with a valid trie. You are free to write any helper functions that you need. You can implement the helper function before or after countAllWords.

```c
int countAllWords(trie T)
{
    if ( T == NULL ){
        return 0;
    }
    int count = 0;
    if (T->isword){
        count = 1;
    }
    return count +
           countAllWords(T->left) +
           countAllWords(T->center) +
           countAllWords(T->right)
}
```
4 Graphs (35 points)

The following two tasks concern this weighted, undirected graph. The vertices are represented by letters, and the edge weights are represented by numbers. (For example, the edge $A \rightarrow C$ has weight 59.)

**Task 1** If we perform a breadth-first search (BFS) starting from the vertex $G$ to determine if $G$ and $E$ are connected, what is one order in which we may mark nodes? (BFS ignores weights.)

Marking order:
- $G$ first, then $F$ and $H$ can appear in any order, $D$ must appear next
- Then $B$ and $C$ may appear in any order, but not after $E$
- (E does not have to appear but if does it has to be last.)

**Task 2** We can use Kruskal’s algorithm to determine a minimum-weight spanning tree from the graph above. In the table below on the left, fill in the edges in the order considered by Kruskal’s algorithm, and indicate for each edge whether it would be added to the spanning tree (Yes) or not (No). Do not list edges that would not even be considered. When you are done, draw the spanning tree below on the right (you don’t have to include edge weights).
<table>
<thead>
<tr>
<th>edge considered</th>
<th>added?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A - B$</td>
<td>yes</td>
</tr>
<tr>
<td>$F - H$</td>
<td>yes</td>
</tr>
<tr>
<td>$C - E$ or $D - F$</td>
<td>yes</td>
</tr>
<tr>
<td>$D - F$ or $C - E$</td>
<td>yes</td>
</tr>
<tr>
<td>$G - H$</td>
<td>yes</td>
</tr>
<tr>
<td>$F - G$</td>
<td>no</td>
</tr>
<tr>
<td>$C - D$</td>
<td>yes</td>
</tr>
<tr>
<td>$D - E$</td>
<td>no</td>
</tr>
<tr>
<td>$B - D$</td>
<td>yes</td>
</tr>
</tbody>
</table>

10 Task 3 A connected component of a graph is a set of vertices where each node can reach every other node in the component along the given edges, and which is connected to no additional vertices by edges. For example, the graph below has three connected components.

Explain briefly (two or three sentences at most) how you could modify a breadth-first search algorithm to compute the number of connected components in a graph.

**Solution:** Rather than looking for a specific vertex, pick a vertex and use breadth-first search to mark all other vertices connected by a path. Then pick an unmarked vertex and repeat the process. The number of times you repeat this process gives you the number of connected components in an undirected graph.
5 Linked Lists (30 points)

This problem deals with operations on a linked list in C. You are given the following code:

```c
struct list_node
{
    int data;
    struct list_node *next;
};

typedef struct list_node list;

struct list_header
{
    list *start;
    list *end;
};

typedef struct list_header *linkedlist;

You are also given the following specification functions:

bool leq(list *start, list *end, int e);
This function returns true if every node in the list from start (inclusive) to end (inclusive) is less than or equal to e.

bool is_in(list *start, list *end, int e);
This function returns true if value e exists in the list from start (inclusive) to end (inclusive).

Task 1 In this task we ask you to analyze why some pointer accesses are safe (not null). Instead of giving an explicit proof, however, we just ask you to indicate the lines in the code you need to conclude that an access is safe Your analysis must be precise: only list the lines upon which the safety of a pointer dereference depends. If a line has not pointer dereference, indicate that by writing NA after the line. We will deduct points if you have too many or two few lines. We show this analysis for the is_segment function as an example to get you started.

bool is_segment(list *s, list *e)
{
    if (s== NULL || e == NULL) /*_________NA________________ */
        return false; /*_________NA________________ */
    list *c = s; /*_________NA________________ */
    while (c != e && c != NULL){ /*_________NA________________ */
        c = c->next; /*________ Line 04 __________ */
    }
    if (c == NULL) /*_________NA________________ */
        return false; /*_________NA________________ */
    return true; /*_________NA________________ */
}
```
void exam(linkedlist a, linkedlist b){
01  REQUIRES(a!=NULL); /* NA */
02  REQUIRES(b!=NULL); /* NA */
03  REQUIRES(is_segment(a->start,a->end)); /* 1 */
04  REQUIRES(is_segment(b->start,b->end)); /* 2 */
05  list **nptr = &b->start; /* 2 */
06  list *t1 = a->start; /* 1 */
07  list *t2 = b->start; /* 2 */
08  while(t1 != a->end && t2 != b->end){ /* 1, 2 */
09    ASSERT(is_segment(t1, a->end)); /* 1 */
10    ASSERT(is_segment(t2, b->end)); /* 2 */
11    list *t = t2; /* NA */
12    t2 = t2->next; /* 10 */
13    t->next = t1->next; /* 9, 10, 11 */
14    t1->next = t; /* 9 */
15    t1 = t1->next->next; /* 9, 10, 11, 14 */
16  }
17  *nptr = t2; /* 5 */
}

Discussion of line 15:
- t1 is not null (line 9)
- t1->next == t (line 14), t == t2 (line 11), t2 is not null (line 10)
  ... therefore t1->next is not null

6 Task 2 The following picture shows the contents of two linked lists A and B. Draw the contents of A and B after the following function call.
exam(A, B);
In this task you will implement a function called `remove_max` that will remove the biggest number from a linkedlist and return it. You can assume that there are no duplicates in the list. Make sure to write insightful post conditions (`ENSURES`) for this function - the preconditions are already given:

```c
int remove_max(linkedlist a){
    REQUIRES(a != NULL);
    REQUIRES(is_segment(a->start, a->end));
    REQUIRES(a->start != a->end); // List cannot be empty

    list* start = a->start;
    list* curr = start->next;
    list* prev = start;

    list* max = start;
    list* max_prev = start;

    while (curr != a->end) {
        if(curr->data > max->data) {
            max_prev = prev;
            max = curr;
        }
        prev = curr;
        curr = curr->next;
    }

    if(max == max_prev) a->start = a->start->next;
    else max_prev->next = max->next;

    result = max->data;
    free(max); // Optional, wasn’t specified who owned the list

    ENSURES(is_segment(a->start, a->end));
    ENSURES(leq(a->start, a->end, result)); // Optional
    ENSURES(!is_in(a->start, a->end, result)); // Optional
    return result;
}
```
6 Unicorn Office (40 points)

In this problem, you are given C programs that may have undefined behavior. If the program can have undefined behavior, find a test case that triggers it, so that, according to the C semantics, your C compiler would be allowed to display pictures of unicorns instead. For each of the programs in the following tasks:

- If the program has defined behavior for all cases, cross out the test case and write down "NO UNICORNS" instead.
- If the program can have undefined behavior, complete the code for the `test()` function so that, when run, it makes the given program code run an operation of undefined behavior. **Your tests are not allowed to violate contracts or have undefined behavior themselves.**
- Circle the position in the code where the test data will trigger the undefined behavior and explain what is undefined about it in a few words. In case there are multiple errors, only circle the first one that will occur when running the program with the test data you provide.

Here’s one example to get you started. Consider this code.

```c
int addy(int x, int y) {
    return x + y; // addition with overflow undefined
}
```

Running the following test code will trigger the undefinedness indicated above:

```c
void test() {
    int x = INT_MAX;
    int y = 1;
    addy(x, y);
}
```
Task 1

```c
struct tree_node {
    elem data;
    struct tree_node *left;
    struct tree_node *right;
};
typedef struct tree_node tree;

tree *top(tree *T) {
    REQUIRES(T != NULL);
    tree *S = T->left;
    T->left = S->right;
    S->right = T;
    ENSURES(S != NULL);
    return S;
}

Fill in test code so that running it will trigger the undefinedness you circled above:

```c
void test() {
    tree *T = xmalloc(sizeof(struct tree_node));
    T->left = NULL;
    T = top(T);
} // UNICORN at ... = S->right;
```

Task 2

```c
int sub(int x, int y) {
    if (x > 0 && y < 0 && x-y < 0)
        return 0x7FFFFFFF;
    else
        return x-y;
}

Fill in test code so that running it will trigger the undefinedness you circled above:

```c
void test() {
    int x = INT_MAX;
    int y = -1;
    x = sub(x, y);
} // UNICORN at ... && x-y < 0

void test() {
    int x = INT_MIN;
    int y = 1;
    x = sub(x, y);
} // UNICORN at return x-y
```
10 Task 3

```c
int f(int a, int b) {
    int r = a;
    a = a + b;
    b = a - b;
    return r;
}

int g(int x, int y) {
    int a = x;
    int b = 0;
    int c = 1;
    c = f(a, b);
    c = c + b*b*b + 1/2*c;
    return c;
}
```

Fill in test code so that running it will trigger the undefinedness you circled above:

```c
void test() {
    int x = _____________________________;
    int y = _____________________________;

    // HERE BE NO UNICORNS
    x = g(x, y);
}
```

15 Task 4

```c
int scatterbox(int n) {
    REQUIRES(3 < n && n*n < 100*n);
    int *A = calloc(n, sizeof(int));
    A[0] = 0;
    A[1] = 1;
    for (int i = 1; i < n-1; i++) {
    }
    return n;
}
```

Fill in test code so that running it will trigger the undefinedness you circled above:

```c
void test() {
    int n = 7; // or 6

    scatterbox(n);
} // UNICORN at ... A[A[i]]
```
7 Safety in C (40 points)

In C, we consider a statement safe if it has defined behavior. A precondition for safety is an assertion which, if true, guarantees that the following statement will execute safely. As an example, assume we have a variable \( x \) declared and initialized with \( \text{int } x = \ldots \) for some unknown value. The assertion \( x < 0 \) is a safety precondition for an increment of \( x \).

```
ASSERT(x < 0);
x = x+1;
```

However, the precondition \( x < 0 \), while guaranteeing safety, poses unnecessary restrictions, because the meaning of \( x+1 \) is also defined for many other values of \( x \). The weakest precondition for safety imposes the fewest restrictions under which the statement after it is safe. In the above example, the weakest precondition would be \( x < \text{INT\_MAX} \).

```
ASSERT(x < \text{INT\_MAX});
x = x+1;
```

If a statement is always safe, the weakest precondition is just “true”; if a statement is never safe, the weakest precondition is just “false”.

Fill in the weakest preconditions for safety for each of the following statements. You may use macros in \texttt{stdlib.h}, \texttt{bool.h}, and \texttt{limits.h}. You should assume that all initialization code returns safely and without aborting and that there are no intervening statements. When necessary, you can assume that \texttt{int} is 32bit. But use quantities like \texttt{INT\_MAX} whenever possible.

**Task 1**

```
int a = \ldots; int b = \ldots;
int A[42];

ASSERT(0 <= b && b < 42 && a <= \text{INT\_MAX} - b*2);
A[b] = a+b*2;
```

**Task 2**

```
int n = \ldots; int y = \ldots;
int *C = malloc(42 * sizeof(int));

ASSERT(C != \text{NULL} && 0 < n && n < 42 && y < \text{INT\_MAX});
A[n] = 1/2*(1+y)/(2*n); // n.b. evaluates to 0
```

**Task 3**

In this task, the compound statement (for loop) is considered as a single statement which requires a precondition for safety.

```
int x = \ldots;
int A[14];
```
A[0] = 2;

ASSERT(x <= 5);

for (int n = 1; n < x; n++)

### Task 4

In this task, the *compound statement* (while loop) is considered as a single statement which requires a precondition for safety.

```c
int x = ...; int y = ...;
if (x <= 0) x = -x + 1;

ASSERT(false);

while (x != -x) {
    x++;
    y--;
}
```

### Task 5

```c
unsigned int n = ...;
unsigned int sq = 0;

ASSERT(true);

for (unsigned int u = 0; u < n; u++)
    sq = sq + u * u;
```

### Task 6

In this task, do **not** assume that unsigned ints are 32 bits wide. You may use functions and macros in `<stdlib.h>` and `<limits.h>`.

```c
unsigned int k = ...;

ASSERT(k < 8*sizeof(unsigned int));

k = 1 << k;
```