# ECE 467 Final
# Exam Type: C

University of Toronto
Faculty of Applied Science and Engineering
Examiner: Adrian Chiu

2022 December 19
2.5 Hours

| Question | Points | Score |
|---|---|---|
| 1 | 7 | |
| 2 | 3 | |
| 3 | 4 | |
| 4 | 6 | |
| 5 | 6 | |
| 6 | 6 | |
| 7 | 5 | |
| Total: | 37 | |

1. Consider the following language $L$ of even-length, 2-bit palindromes (2-bit meaning each character is one of two values, i.e. a bit).

   - The alphabet $\Sigma = \{0, 1\}$.
   - The empty string $\epsilon$ is in $L$.
   - If $s \in L$, then $0s0 \in L$.
   - If $s \in L$, then $1s1 \in L$.

   (a) (1 point) Treating the alphabet as the set of terminals, write a context-free grammar that defines this language.

```
1. S -> 0 S 0
2. S -> 1 S 1
3. S -> epsilon
```

   (b) (1 point) 1001 is a string in $L$. Verify this by writing a derivation for it with respect to your grammar.

```
S
1 S 1 // 2
1 0 S 0 1 // 1
1 0 0 1 // 3
```

   (c) (1 point) Your grammar is not LR(1). How *would* you show that it is not LR(1) (you do not actually have to carry out the process).

   One would try creating the LR(1) parsing table, and check for conflicts.

(d) In general, a grammar might not be LR(1), but the language it accepts may be accepted by another grammar which is LR(1). We wish to determine that *no possible* LR(1) grammar defines the language $L$.

Suppose we had an LR(1) grammar and corresponding LR(1) parsing table that accepted $L$ (upon applying the LR(1) parsing algorithm on input strings from $L$). Refer to this as the parser $P$, e.g. "$P$ accepts (strings from) $L$".

Since the string $1001 \in L$, $P$ accepts 1001. In particular:

- From the start state, $P$ has valid actions for each next character/terminal of 1001.
- Upon reaching the end of the input 1001 (i.e. given \$ as the lookahead), it accepts, i.e. *it reduces by the augmented start rule.*

i. (1 point) Consider the execution of $P$ on 1001, and 00 which is also in $L$. In each case, upon reaching the end of input (i.e. given \$ as the lookahead), what state is $P$ in?

$P$ must be in the state containing the following item involving the augmented start rule: `S' -> S ., $`.
Note that by construction, there can only be one state with this item; for marking, it was sufficient to answer (informally) something like "the accepting state" or "the final state".
Alternatively, "state 0" or "the start state" was accepted, which is where the parser would be *after* performing the reduction. Although subtle, the question was intended to refer to the state *before* performing the action (keyword: *upon*).

ii. (2 points) Consider the execution of $P$ on $10011001 \in L$, and $001001 \notin L$.
Assuming $P$ accepts all strings in $L$, why does $P$ also accept 001001?

- -0.5 for incorrect or unclear reasoning.
- 0.5 if left blank.

Upon consuming 00, $P$ must be in the same state (the accepting state) as consuming 1001. If `$` is the lookahead, then $P$ reduces by the augmented start rule. From this state, given 1001 as the rest of the input (so 1 as the next lookahead), $P$ can transition to the accepting state, since it accepts 10011001. Therefore, it accepts 001001.
Intuitively, $P$ cannot distinguish between 00 and 1001 as the start of the string.

(e) (1 point) Can we define $L$ using a regular expression? If so, give an example. If not, why?

No, because the languages that can be defined by regular expressions are a strict subset of the languages that can be defined by context-free grammars.

For the "why" part of the question, answers received regarding "counting" or "recursion" are not sufficient. For example, the following recursively defined language $L'$ of even-length strings of 1s can be defined by a regular expression.

- $\epsilon \in L'$.
- If $s \in L'$, then $s11 \in L'$.

Total for Question 1: 7

2. Consider the following code.

```
x0 = 5;
while (x1 = phi(x0, x4); x1 != 1) {
    if (x1 % 2) == 0 {
        x2 = x1 / 2;
    } else {
        t = 3 * x1;
        x3 = t + 1;
    }
    x4 = phi(x2, x3);
}
return x1;
```

(a) (1 point) Show the values of all variables/names (when applicable) in the first iteration of the loop.

```
x0 = 5
x1 = 5
t = 15
x3 = 16
x4 = 16
```

(b) (1 point) Show the values of all variables/names (when applicable) in the second iteration of the loop.

```
(x0 = 5)
x1 = 16
x2 = 8
x4 = 8
```

(c) (1 point) Does the loop terminate?

Yes; future iterations of the loop divide what is effectively `x` by 2 until it reaches 1.

Total for Question 2: 3

3. Consider the data-flow analysis of constant propagation for three 32-bit (signed) integer variables $x, y, z$.

   The lattice $V$ for a single variable consists of $2^{32}+2$ values: $\top$, $\bot$, and each of the possible 32-bit (signed) integer values. A value in the domain of this constant propagation is a 3-tuple in the product lattice $V \times V \times V$.

   The meet operator $\wedge$ for a single lattice $V$ is defined as follows (where $c, d$ are any two distinct 32-bit (signed) integer values).

| $x_1$ | $x_2$ | $x_1 \wedge x_2$ |
|---|---|---|
| $\bot$ | $\bot$ | $\bot$ |
| $\bot$ | $d$ | $\bot$ |
| $\bot$ | $\top$ | $\bot$ |
| $c$ | $\bot$ | $\bot$ |
| $c$ | $d$ | $\bot$ |
| $c$ | $c$ | $c$ |
| $c$ | $\top$ | $c$ |
| $\top$ | $\bot$ | $\bot$ |
| $\top$ | $d$ | $d$ |
| $\top$ | $\top$ | $\top$ |

   The meet operator for the product lattice is defined in terms of the individual lattices as follows.

$$(x_1, y_1, z_1) \wedge (x_2, y_2, z_2) = (x_1 \wedge x_2, y_1 \wedge y_2, z_1 \wedge z_2).$$

   The transfer function $f_s$ for a statement $s$ of the form $x = y + z$ is defined as follows, where $\_$ denotes any value ($c, d$ are any 32-bit (signed) integer values).

| $(x, y, z)$ | $f_s(x, y, z)$ |
|---|---|
| $(\_, \bot, \bot)$ | $(\bot, \bot, \bot)$ |
| $(\_, \bot, d)$ | $(\bot, \bot, d)$ |
| $(\_, \bot, \top)$ | $(\top, \bot, \top)$ |
| $(\_, c, \bot)$ | $(\bot, c, \bot)$ |
| $(\_, c, d)$ | $(c + d, c, d)$ |
| $(\_, c, \top)$ | $(\top, c, \top)$ |
| $(\_, \top, \bot)$ | $(\top, \top, \bot)$ |
| $(\_, \top, d)$ | $(\top, \top, d)$ |
| $(\_, \top, \top)$ | $(\top, \top, \top)$ |

(a) (1 point) Write an equivalent definition for following inequality using the meet operator.

$$(x_1, y_1, z_1) \leq (x_2, y_2, z_2).$$

Any of the following two suffice.

$$(x_1, y_1, z_1) \wedge (x_2, y_2, z_2) = (x_1, y_1, z_1)$$
$$(x_1 \wedge x_2, y_1 \wedge y_2, z_1 \wedge z_2) = (x_1, y_1, z_1).$$

(b) (2 points) Select values $(a, b, c)$ and $(d, e, f)$ such that the following is true.

$$f((a,b,c) \wedge (d,e,f)) \lneq f(a,b,c) \wedge f(d,e,f).$$

- -1 point for incorrect computation or result.
- 0.5 for leaving this question blank.
- Note: this question is *not* marked with respect to your previous definition. If you are not confident in your previous definition, you *may* wish to leave this question blank.

$$\begin{aligned}
a &= 0 \\
b &= 2 \\
c &= 5 \\
d &= 0 \\
e &= 3 \\
f &= 4 \\
(a,b,c) \wedge (d,e,f) &= (0,\perp,\perp) \\
f((a,b,c) \wedge (d,e,f)) &= (\perp,\perp,\perp) \\
f(a,b,c) &= (7,2,5) \\
f(d,e,f) &= (7,3,4) \\
f(a,b,c) \wedge f(d,e,f) &= (7,\perp,\perp) \\
(\perp,\perp,\perp) \lneq (7,\perp,\perp).
\end{aligned}$$

Couple notes: any values could have been chosen for $a$ and $d$. Also, regardless of what $a \wedge d$ is, after applying the transfer function, the value for $x$ is overwritten (in this case, by $\perp$).

This was highlighted in lecture by something like the following.

```
if (...) {
    y = 2;
    z = 5;
} else {
    y = 3;
    z = 4;
}
x = y + z;
```

There are two paths for the corresponding control-flow graph. If we were to apply the transfer functions for each statement along each path separately, we get that $x = 7$ at the end of each path, and taking the meet at the end, we retain the value of 7 for $x$.

On the other hand, if we take the meet function between each statement/node (as done in the iterative algorithm), we "lose track" of $y$ and $z$ when the if-statement merges, and the final value of $x$ is unknown (not a constant/bottom).

(c) (1 point) How do the "meet over paths" (MOP) solutions compare to the "maximal fixed point" (MFP) solutions for this data-flow analysis?

The MFP solution is less (precise) than the MOP solution. (In the presence of code like that demonstrated above, the MFP solution is *strictly* less (precise) than the MOP solution.)

Total for Question 3: 4

4. Relaxed atomic memory ordering is often only sufficient for counters. However, in general, it is *not* sufficient for thread-safe reference counting. Consider the following code.

```
struct RefCountedData {
    std::atomic<int> count;
    Data data;
};

void increment(RefCountedData* rc) {
    int old_count = rc->count.load(relaxed);
    rc->count.store(old_count + 1, relaxed);
}

void decrement(RefCountedData* rc) {
    int old_count = rc->count.load(relaxed);
    rc->count.store(old_count - 1, relaxed);
    if (old_count == 1) {
        // Destroy data
        rc->data.~Data();
    }
}
```

(a) (3 points) Consider the `decrement` operation. Describe how it may be possible for the thread destroying `data` to read stale memory for `data`.

CLARIFICATION: there are other potential problems with the code. Specifically, this question is concerned with the access of the `data` field in the `decrement` operation.

*Hint: consider multiple threads using `data`, and then calling `decrement`.*

- -1 point for incorrect or unclear reasoning.
- 1 point if left blank.

(In hindsight, this question is just a repetition of the next two parts; informally, any of the answers from the next two parts would suffice here, but here is the technical explanation.)

Suppose thread 1 modifies `data`, then decrements the count from 2 to 1. Then thread 2 decrements the count from 1 to 0, and calls the destructor of `data`.

There is no "happens before" relationship between the usages of `data` in different threads. A "happens before" relationship only exists between operations that the language spec says there is. One way to impose a "happens before" relationship between different threads is with a release-acquire pair; a release store to a memory location "happens before" an acquire load from the same memory location that reads the value stored.

Note: this level of technical detail should have been stated in the lecture for atomics (but admittedly not elaborated much upon). Although it has been stated at the beginning of the course, and again multiple times, that the videos are not a replacement for the lectures, I realize that many students inevitably only watched the videos. Regardless, this question is still answerable (informally) by only reasoning about "memory reorderings"; see below.

The probably-more-obvious problem is that the read-modify-write of count is not atomic; we may "lose counts". However, *specifically concerning `decrement`* (if we assume `increment` worked correctly), the "worst" thing that could happen by losing a count in `decrement` is that memory is not freed whatsoever (leaked); the `data` field would not be accessed at all. Note that it is only possible for the count to be 1 while two threads have the same pointer if there was a previous problem in `increment`.

(b) (1 ½ points) What is the problem if *only* the `load` is changed to an acquire ordering?

- -0.5 for incorrect or unclear reasoning.
- 0.5 if left blank.

If the `store` is still relaxed, then in the scenario described above, the modifications to `data` in thread 1 may appear/be reordered to happen after the decrement the count from 2 to 1. Thread 2 may decrement the count from 1 to 0, and call the destructor, before the modifications to `data` in thread 1 happen/become visible to thread 2.

(c) (1 ½ points) What is the problem if *only* the `store` is changed to a release ordering?

- -0.5 for incorrect or unclear reasoning.
- 0.5 if left blank.

Verbosely, the line `rc->data. Data();` can be broken down into two operations: reading the value of data, then calling the destructor on that value.

```
void decrement(RefCountedData* rc) {
    int old_count = rc->count.load(relaxed);
    rc->count.store(old_count - 1, release);
    if (old_count == 1) {
        // Read data
        Data tmp = rc->data;
        // Destroy data
        tmp.~Data();
    }
}
```

Rewritten verbosely, it can be seen that the compiler may move the read-only line `Data tmp = rc->data;` before the release store, *and* before the relaxed load. Intuitively, the following sequence of things may happen:

1. Thread 2 pre-emptively reads `data`;
2. Thread 1 modifies `data`;
3. Thread 1 decrements the count from 2 to 1;
4. Thread 2 decrements the count from 1 to 0;
5. Thread 2 destructs `data` using the prefetched value, which doesn't reflect thread 1's modifications.

Note that all these operation reorderings (including those in part (b)) are valid in a single-threaded scenario; the compiler doesn't reason about interactions between different threads unless there are the proper synchronization operations indicating to it that there are other threads' operations that it needs to be aware of.

Total for Question 4: 6

5. Suppose we have the following pseudocode implementation of a stack-based interpreter that operates only on integers.

```
while (bytecode != nullptr) {
    switch (*bytecode) {
        case "load x":
        push(local_variables[x]);
        bytecode++;

        case "store x":
        local_variables[x] = pop();
        bytecode++;

        case "constant c":
        push(c);
        bytecode++;

        // CORRECTION: added "less" operation
        case "add/sub/mul/div/less":
        right = pop();
        left = pop();
        push(left add/sub/mul/div/less right);
        bytecode++;

        case "jump n":
        bytecode += n;

        case "test n":
        condition = pop();
        if (condition != 0) {
            bytecode++;
        } else {
            bytecode += n;
        }
    }
}
```

Consider the following C code.

```
t = 1;
for (int i = 0; i < 10; i++) {
    t = t * 2 + 1;
}
```

(a) (2 points) Draw a control-flow graph for the above C code (break up the clauses of the for-loop). You may group multiple operations into basic blocks.

- -0.5 for each mistake.

Too lazy to draw the graph in latex; this should have been straightforward; infer from part (c).

(b) (2 points) Translate the code in *each individual node* into bytecodes for the interpreter above. For `jump` and `test` bytecodes, you do not need to include the offset.

- -0.5 for each mistake.

Infer from part (c).

(c) (2 points) Combine the fragments of bytecodes from each node into *a single array of bytecodes*. Then, fill in the offsets for `jump` and `test`.

- -0.5 for each mistake.

```
1. constant 1
2. store t
3. constant 0
4. store i
5. load i
6. constant 10
7. less
8. test 12 // to instruction 20
9. load t
10. constant 2
11. mul
12. constant 1
13. add
14. store t
15. load i
16. constant 1
17. add
18. store i
19. jump -14 // to instruction 5
20. (exit/after the loop)
```

Note: as pointed out by a student during the exam, functionally equivalent bytecode could have been produced without the `less` bytecode (and this may be an optimization a compiler makes, since the observable behaviour is indistinguishable). The loop would always produce the same result if the condition was `i != 10`, which is equivalent to `i - 10 != 0`, which could be implemented with the original bytecodes.

Total for Question 5: 6

6. Suppose we have *some* GC algorithm that satisfies the tri-color invariant:

   - All objects are labeled exactly one of: black, grey, or white.
   - No black-labeled object contains a pointer to a white-labeled object.

   You are given no other information about this GC algorithm; reason about it based on transitions between colours.

   (a) (1 point) What is a terminating condition for this GC algorithm?

   There are no grey objects.

   (b) (3 points) What are all the transitions we should disallow to ensure that the GC algorithm will terminate?

   - -1 point for incorrect or missing answer.
   - 1 point if left blank.

   There are 6 possible transitions. The following should be forbidden.

   - Black to grey.
   - Black to white.
   - Grey to white.

   Explanation is not necessary, but here are some comments.

   Note that black to white and grey to white transitions may violate the invariant (break correctness), and in general would be forbidden on that premise. However, even disregarding the invariant/correctness, and *only* considering the terminating condition, they still need to be forbidden.

   Suppose black to white is allowed (and grey to white may be forbidden). Note that grey to black must be allowed; if we don't assume anything about how the objects are initially labeled, some may be grey (and in practice that is what the root set is initialized to), in which case they need to transition to something else (i.e. to black). Then we could have the following cycle of transitions such that the grey set might never empty: white -¿ grey -¿ black -¿ white.

   Suppose grey to white is allowed (and black to white may be forbidden). Then we could have the following cycle of transitions such that the grey set might never empty: white -¿ grey -¿ white. In practice, white to grey is necessary as it represents "unscanned to 'to be scanned'", but hypothetically, if both white to/from grey transitions were forbidden (as well as black to grey), the algorithm would also be guaranteed to terminate. For the purposes of this question, that is also a valid answer.

   (c) (2 points) Suppose this GC algorithm may run concurrently with the mutator threads (by the general definition of concurrent). The mutators may allocate objects while GC is running. What colour should newly allocated objects be labeled to ensure that the algorithm terminates, and why?

   - -1 point for incorrect or unclear reasoning.
   - 0.5 points if left blank.

   Newly allocated objects should be labeled black. Otherwise, allocating white objects (which may transition to grey) or grey objects may outpace emptying the grey set.

   Note that in general, barriers are required for correctness (e.g. upholding the invariant/correctness) for concurrent GC. Some students answered grey or white based on correctness of the invariant, but the question specifically asks about *termination.* Note that in practice, a new *allocation* cannot contain any pointers to existing code. This may be subtle coming from C++ or Java where an expression like 'new Foo' allocates memory *and* calls the constructor. However, it is in the latter step that barriers would uphold correctness.

Total for Question 6: 6

7. Based on the lab.

   (a) (1 point) Draw the general control-flow graph/basic block structure for a while loop *without* `break` or `continue` statements in the body.

   Too lazy to draw the graph in latex, but it should be straightforward. Consists of the loop header/entry, the loop body, and loop exit/"after". The entry is succeeded by either the body or the exit. The body is succeeded by only the entry.

   (b) (2 points) Ignoring nested loops, how would you support `break` and `continue`?

   - No penalty for error.

   Keep track of both the loop entry and exit blocks when inside a loop. A `break` statement is an unconditional jump to the exit. A `continue` statement is an unconditional jump to the entry.

   (c) (2 points) Suppose `break` and `continue` only ever apply to the immediate enclosing loop. How would you extend your solution in the previous part to handle nested loops? For example, the following is valid code.

```
while (...) {
    while (...) {
        if (...) {
            break_again = true;
            break;
        }
    }
    if (break_again) {
        break;
    }
}
```

   - No penalty for error.

   Keep a stack of "loop infos". Upon encountering a loop, create the blocks upfront, so pointers to the entry and exit can be pushed on to the "loop info" stack. Inside a loop, `break` and `continue` apply to the blocks on the top of the stack. Pop a single "loop info" after generating code for the body of the loop.

   (d) (0 points) Have a good holiday break.

   Left as an exercise to the reader.

Total for Question 7: 5