1. (a) 3 marks Rewrite the following two BAD forms of loop exit into GOOD forms.

<table>
<thead>
<tr>
<th>BAD</th>
<th>BAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{for} ( ;; ) {</td>
<td>\texttt{for} ( ;; ) {</td>
</tr>
<tr>
<td>S1</td>
<td>S1</td>
</tr>
<tr>
<td>\texttt{if} ( C1 ) {</td>
<td>\texttt{if} ( C1 ) {</td>
</tr>
<tr>
<td>S2</td>
<td>break;</td>
</tr>
<tr>
<td>} else {</td>
<td>} else {</td>
</tr>
<tr>
<td>break;</td>
<td>S2</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

ii. 1 mark Explain the problem with the BAD form.

(b) 2 marks Why is a labelled break statement better than using a goto statement to perform the same operation?

(c) 1 mark What property makes a variable a flag variable?

(d) i. 2 marks State two problems with return codes to indicate multiple outcomes from a routine.

ii. 1 mark Are return codes faster or slower than exceptions for indicating multiple outcomes?

(e) 2 marks Explain how the \texttt{Finally} clause works with respect to execution of a \texttt{try} statement.

(f) 2 marks Explain the terms source and faulting execution with respect to exception handling.

(g) i. 1 mark Where does control return at the end of a \texttt{catch} clause of a \texttt{try} statement?

ii. 1 mark Where does control return at the end of a \texttt{CatchResume} clause of a \texttt{try} statement?

(h) 1 mark Why is \texttt{Throw} \texttt{E()} \texttt{At} coroutine/task unsupported in \mu C++?

(i) 1 mark State the primary criteria for using heap allocation.

(j) 1 mark Why does concurrent heap-allocation cause significant performance problems?

2. (a) 2 marks Explain why coroutines are sequential versus concurrent.

(b) 1 mark What property of a \mu C++ coroutine allows modularization within the coroutine main?

(c) 1 mark When a coroutine’s interface-member is called, on which stack does the call-frame go?

(d) 2 marks When a non-terminated coroutine is deallocated, what occurs to that coroutine’s stack and why does it occur?

(e) 1 mark What does it mean to linearize (or flatten) a series of executable statements?

(f) 1 mark What is the purpose of the \mu C++ verify routine for a coroutine or task?

(g) 1 mark Given a full-coroutine cycle that is executing, what happens when suspend calls are performed?

(h) 2 marks Explain the difference between Python and \mu C++ coroutines. Give an example of what the difference does or does not allow.

3. (a) 1 mark What is a concurrent bottleneck?

(b) 1 mark What is the most important take-away (lesson) from Amdahl’s law?

(c) 1 mark What is the critical path in concurrent execution?

(d) 2 marks Is \texttt{COBEGIN/COEND} as expressive as \texttt{START/WAIT}? Explain.

(e) 1 mark Explain why static variables are dangerous in a \texttt{Task} type.

(f) 2 marks Explain the difference between liveness (rule 4) and eventual progress (rule 5) in the mutual-exclusion game.
(g) **2 marks** The following is the declare-intent solution for mutual exclusion:

```c
me = WantIn; // entry protocol
while ( you == WantIn ) {}  
CriticalSection(); // critical section
me = Don'tWantIn; // exit protocol
```

Explain what rule of the mutual-exclusion game is violated and how.

(h) **3 marks** Given the Lock variable and an implementation of the test-and-set instruction:

```c
int Lock = OPEN; // shared

int TestSet( int & b ) {
    // begin atomic
    int temp = b;
    b = CLOSED;
    // end atomic
    return temp;
}
```

use them to write the entry and exit protocols for mutual exclusion.

4. (a) **2 marks** What are *independent* and *dependent* critical sections?

(b) **1 mark** How many locks are needed for mutual exclusion?

(c) **2 marks** Explain the difference between barging *avoidance* and *prevention*.

(d) **2 marks** Why does a synchronization lock’s wait member take a mutex-lock parameter?

(f) **2 marks** Explain why a mutex lock cannot perform synchronization, and why a synchronization lock cannot perform mutual exclusion.

5. **19 marks** Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

```c
_Coroutine Grammar {
    char ch;   // character passed by cocaller
    void main();  // YOU WRITE THIS ROUTINE

public:
    _Event Match {};
    _Event Error {};
    void next( char c ) {
        ch = c;
        resume();
    }

};
```

which verifies a string of characters matches the language `'(+'n (XY)+')n)'+n`, where X and Y are not equal to `(')', n ≥ 1, and the number of open/closing parentheses and repeated pairs of XY characters are equal; i.e., there are 1 or more opening parenthesis, followed by the same number of repeated pairs of any characters, followed by the same number of closing parenthesis, e.g.:

<table>
<thead>
<tr>
<th>valid strings</th>
<th>invalid strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ab)</td>
<td>(ab)</td>
</tr>
<tr>
<td>((xyxy))</td>
<td>((xy))</td>
</tr>
<tr>
<td>(((#@@#@@)))</td>
<td>ab</td>
</tr>
<tr>
<td>((www))</td>
<td>(a)</td>
</tr>
</tbody>
</table>
After creation, the coroutine is resumed with a series of characters (one character at a time). The coroutine accepts characters until:

- the characters form a valid string in the language, and it then raises the exception Grammar::Match at the last resumer;
- the last character results in a string not in the language, it then raises the exception Grammar::Error at the last resumer.

After the coroutine raises a Match or Error exception, it must terminate; sending more characters to the coroutine after this point is undefined. (You may use multiple return statements in Grammar::main.)

Write ONLY Grammar::main, do NOT write a main program that uses it! No documentation or error checking of any form is required.

Note: Few marks will be given for a solution that does not take advantage of the capabilities of the coroutine, i.e., you must use the coroutine’s ability to retain data and execution state.

6. **31 marks** Divide and conquer is a technique that can be applied to certain kinds of problems. These problems are characterized by the ability to subdivide the work across the data, such that the work can be performed independently on the data. In general, the work performed on each group of data is identical to the work that is performed on the data as a whole. What is important is that only termination synchronization is required to know the work is done; the partial results can then be processed further.

Write a COMPLETE µC++ program to efficiently check if any row of a matrix of size $N \times M$ contains at least 2 Schmilblicks. For example, in:

$$
\begin{pmatrix}
1 & -1 & 3 & 4 & -1 \\
2 & 1 & 4 & -1 & 6 \\
3 & -1 & -1 & 6 & -1 \\
-1 & 6 & 7 & -1 & 1 \\
4 & -1 & 6 & 1 & 8 \\
\end{pmatrix}
$$

the Schmilblick value is $-1$, and rows 1, 3, 4 contain at least 2 Schmilblicks. The matrix is checked concurrently along its rows. Each checking task has the following interface (you may only add a public destructor and private members):

```cpp
Evento Schmilblick {}; // concurrent exception
_Task Schmilblicks { // check row of matrix
    // YOU ADD HERE
};
void main(); // YOU WRITE THIS ROUTINE

public:
    _Event Stop {}; // concurrent exception
    Schmilblicks(
        const int row[], // one row of the matrix
        const int cols, // number of columns in row
        uBaseTask & pgmMain, // contact when Schmilblicks found
        int schmilblick // schmilblick value
    );
};
```

The program main reads from standard input the Schmilblick value and matrix dimensions ($N \times M$), declares any necessary matrix, arrays and variables, reads (from standard input) and prints (to standard output) the matrix, concurrently checks the matrix values in each row, and prints a message to standard output.
output if two Schmilblicks are found in any matrix row. **No documentation or error checking of any form is required.**

As an optimization, each Schmilblicks task that finds a second Schmilblick raises the concurrent exception Schmilblick at the pgmMain and then returns, and when the program main receives this concurrent exception, it raises exception Schmilblicks::Stop at any non-deleted Schmilblicks tasks. When the concurrent Stop exception is propagated in a Schmilblicks task, it stops performing the Schmilblicks check and returns.

An example of the input for the program is:

```
-1 5 5       Schmilblick value and matrix dimensions
1 -1 3 4 -1  matrix values
2 1 4 -1 6
3 -1 1 6 -1
4 -1 6 1 8
-1 6 7 -1 1
```

(The phrases “Schmilblick value and matrix dimensions” and “matrix values” do not appear in the input.) In general, the input format is free form, meaning any amount of white space may separate the values.

Example outputs are:

```
1, -1, 3, 4, -1, original matrix | 1, 2, 3, 4, 5, original matrix
2, 1, 4, -1, 6,                   | 2, 1, 4, 5, 6,                   
3, -1, 1, 6, -1,                  | 3, 4, 1, 6, 7,                  
4, -1, 6, 1, 8,                   | 4, 5, 6, 1, 8,                  
-1, 6, 7, -1, 1,                  | 5, 6, 7, 8, 1,                  

Schmilblicks found                | Schmilblicks not found
```

(The phrase “original matrix” does not appear in the output.) Note, the comma is a terminator not a separator.