# Compilers Lab Exam
## CS3423 / CS6240 / CS6383

*Aug 18 Semester*

October 3rd, 2018, Time: 3+ hours

---

## Instructions

- You are allowed to access flex/bison manuals ONLY through man pages which are available locally (use **info (man) flex/bison** command).
- Internet is allowed ONLY to submit your work at the end of the exam. You should inform TAs while submission. Usage of the internet during the course of the exam is **strictly** prohibited. It is assumed that you have read the “Rules and Logistics” Document and are familiar with Plagiarism policy of CSE@IITH.
- You are not allowed to carry Mobile phones, Tablets, USB sticks, etc.
- Any methods to access solutions through unfair means would result in immediate granting of an FR grade in the entire course (CS3423 / CS6240 / CS6383).
- You can specify any additional assumptions your implementation makes and/or status (if incomplete) in the README file bundled along with the solution.
    - Do note that if you are submitting only a partial solution, your chances of getting partial marks increases if you document your code and give a good README.
- Place all your code files and your readme files into a directory named RollNo. + “_” + “Final” (Eg : CS16BTECH11001_Final). Compress it into a tar.gz file with the same name and upload it in Google Classroom.
- You can assume that the input is error free. In particular, there is no need to do error handling.

## Q1) Translator from Foo to C/C++ (40 pts)

Foo is an esoteric programming language. Foo programs maintain an array of integers and a stack. The values in the array and the stack can be any integer between 0 and 65535 (both inclusive). You can assume the size of the array and the maximum stack size to be 256. There is a pointer to the array that moves across the cells. The following are the commands supported by Foo. (Initially, all the values in the array are 0 and the stack is empty.)

| Command | Description |
|---|---|
| “” | Everything in between is printed to stdout. The strings can contain escaped characters. For example `“\””` prints a double quote to stdout.<br>Partial marks will be awarded if implemented only for strings without escaped characters. |
| `&` | Set the value of the current cell (to which the array pointer is currently pointing) to the number that follows ‘`&`’. If no number is present after ‘`&`’, then a number is popped from the stack and stored in the cell. |
| `@` | Push the number that follows ‘`@`’ to the stack. If no number is given after ‘`@`’, then push the value of the current cell (to which the array pointer is pointing currently) to the stack. |
| `<` | Decrement the array pointer by one. (The pointer wraps around if it reaches the start of the array.) |
| `>` | Increment the array pointer by one. (The pointer wraps around if it reaches the end of the array.) |
| `$` | Print out a single value. The following four modes are supported.<br>● `$i` -> Print the integer in the currently selected cell in decimal form.<br>● `$h` -> Print the integer in the currently selected cell in hexadecimal form (for e.g. if the cell contained `31`, then the output printed to the stdout would be `1f`).<br>● `$b` -> Print the integer in the currently selected cell in binary form.<br>● `$c` followed by an integer -> Print the character in ASCII to the corresponding integer. (For example ‘`$c100`’ prints ‘`d`’; ‘`$c10`’ prints a new line.) |

| | |
|---|---|
| | Partial marks will be awarded if only a subset of the above four modes is implemented. |
| `+, -, *, /, %` | Operates on the value stored in the currently selected cell and the integer that follows the operator.<br><br>For example '`+5`' adds 5 to the current cell. (Suppose the cell had 2, then after the command, it would contain 7.)<br>If no number follows the operator, then the operation is done on the value of the cell and a popped integer from the stack and the result is stored in the current cell.<br><br><u>Note:</u> Overflow can happen i.e. if 1 is added to 65535, then the result is 0. (In other words, all operations are modulo 65536.) |
| `#` | Sleep. For example '`#5`' makes the program sleep for 5 seconds. If no integer is specified, then use the value of the current cell.<br><br><u>Hint:</u> One possible way to implement is to use sleep from POSIX ("`man 3 sleep`"). |
| ( | Beginning of a loop.<br>If an integer follows '`(`', then the loop executes until the value in the current cell equals that number.<br>If no number follows, the loop executes until the value of the current cell becomes zero.<br><br>(Here, "current cell" refers to the cell of the array pointed to at the beginning of each new iteration. It can be different for each iteration, if the pointer moves when inside the body of the loop.) |
| ) | End of the loop.<br>Partial marks will be awarded if nested loops are not handled. |
| Whitespaces (space, line feed, etc.) | Ignore. |

Write a program using Flex/Bison that takes a Foo program as input and emits an equivalent C/C++ program. Extra points will be given if the output is indented properly. You are allowed to use any data structure from STL (An example on how to use stack from STL is provided in the Appendix (last page)). You have to print the C/C++ program to <u>stdout</u>.

**Examples:**

| **Input** | **Output** |
|---|---|
| `"Hello, World!"` | Output: (Your generated C/C++ program can be different.)<br>`#include <iostream>`<br>`int main() { std::cout << “Hello, World!”;`<br>`return 0; }`<br><br>Output (on running the generated C/C++ program):<br>`Hello, World!` |
| `&256*2$i`<br><br>Explanation:<br>`&256` – Store 256 in the current cell.<br>`*2` – Double the value of the current cell. (Now current cell has 512.)<br>`$i` – Print the current cell value to stdout. (Prints 512 to stdout.) | Output (on running the generated C/C++ program):<br>`512` |
| `&4>&1<(0@-1>*<)>$i` | Output (on running the generated C/C++ program):<br>`24` |
| `&5`<br>`(0`<br>`#1`<br>`-1`<br>`$i`<br>`$c10`<br>`)`<br>`"boom!"`<br>`$c10` | Output (on running the generated C/C++ program):<br>4<br>3<br>2<br>1<br>0<br>boom!<br>(Note that there is a pause (sleep) for 1 second before every print. Also, a new line is printed at the end of the program unlike previous examples.) |
| `(1)` | Output: (Nothing. Program is in an infinite loop.) |

## Q2) Convert a system of Polyhedral inequalities and equations from human-readable Omega *"Calculator"* format to PolyLib/FMLib "*Internal Matrix"* format. (40 pts)

A system of equations and inequalities can represent a set of polyhedral (linear) constraints. Some tools that operate on Polyhedral constraints are `Omega` (a classic library which solves a set of constraints using Fourier-Motzkin elimination), `PolyLib` (which supports several polyhedral operations), `FMLib` (which is a new library primarily based on solving a set of constraints using the Fourier-Motzkin operation).

The goal of this question is to convert a given set of constraints that are in human readable external Omega *"Calculator"* format to internal *"Matrix"* format so that they can be processed by PolyLib/FMLib.

The input is a set of constraints in Omega's format. The set is of the form:

```
{ [SetList] : formula }
```

where *`SetList`* is the list of comma-separated variables (dimensions), and *`formula`* is the list of comma-separated constraints. Variable names start with an alphabet and consist of uppercase and lowercase characters and digits.

- Each constraint in *`formula`* can be of one of the following forms:
    - A linear inequality in the variables, with a comparison operator of "`<=`" or "`>=`" (Example: `2x + 3y <= 4`)
    - a linear equation in the variables, with operator "=" (Example: `2x + 3y = 4`)
- The coefficients of the variables and the constant term can be an integer or a rational number, of the form "p/q", where p and q are integers
- If the coefficient is 1 or -1, it is not explicitly mentioned (Example: (1)(x) + 2y = 3 is written as x + 2y = 3).

<u>Note:</u> Due to various mathematical reasons, polyhedral constraints **cannot** be "`>`" or "`<`", and hence these operators need not be considered.

An example of such a set is provided below:

```
{[x,y,z]: x + 2y - 3z >= 10, -3x + 4z = 1, 2/3y + z - 6x <= 3/4,
y  >= 2}
```

The task is to convert the above into (a restricted version of) FMLib's matrix-like format. The format is as follows:

1. The first line of the output consists of two space-separated integers.
    a. The first integer, `m`, is the total number of constraints, i.e., the sum of the number of equations and inequalities.
    b. The second integer, `n`, is the number of variables in the system, i.e., the number of dimensions.

2. This is followed by an `m x (n+2)` matrix that represents the actual constraints. Each row represents a constraint and each column represents a variable. The variables can be in any order (that you can choose), but they must use the same position in all the constraints (preserving semantic equivalence between the input and output formats). The values are as follows:
    a. All constraints must be represented in the form $\sum_i a_i x_i - k = 0$ or $\sum_i a_i x_i - k >= 0$. For example, `3x + 2y <= 5` should be converted to `-3x - 2y + 5 >= 0`, and `2x + 3y = 4` should be converted to `2x + 3y - 4 = 0`, before adding the coefficients to the matrix.
    b. The first column in each row is either a `1` or a `0`; It is `1` if the constraint is an inequality, and `0` if it is an equation.
    c. The last column for the `i`th row is the constant term in the constraint.
    d. In the `i`th row and `j`th column, where `2 <= j <= (n+1)`, the value is coefficient of the `j`th variable in the `i`th constraint. For example, let there be a system with 3 variables - `x`, `y`, and `z`, and they are stored in the same order. A few examples:
        i. `3x + 2y = 1` will have columns `0 3 2 0 -1`
        ii. `3/4x - y - 2/3z >= 1` will have columns `1 3/4 -1 -2/3 -1`
        iii. `x <= 2` will have columns `1 -1 0 0 2`. This constraint is first converted to `-x >= -2`, which gives a coefficient of `-1` for `x` and `2` as the constant term.

**Example** (Colours are an aid to understand the example and **not** a part of the problem):
<u>Input</u>

```
{[x,y,z]: x + 2y - 3z >= 10, -3x + 4z = 1, 2/3y + z - 6x <= 3/4,
y  >= 2}
```

Output
```
4 3

1 1 2 -3 -10
0 -3 0 4 -1
1 6 -2/3 -1 3/4
1 0 1 0 -2
```

Write a program using Lex/Flex and/or Yacc/Bison to perform this conversion. This problem is structured as follows: there is a base assumption, which highly restricts the possible inputs and makes the task easier. This is followed by a list of relaxations on the assumptions on the input. Write a single implementation that covers inputs following the assumption as many relaxations as possible. Covering all relaxations correctly will fetch full marks, and covering a subset will specify partial marks as indicated. Partially covering a relaxation may fetch partial marks for that subpart if explained appropriately in the README.

Base Assumptions: [10 pts]
Assume that there are *exactly* 10 variables and 10 constraints. The variables are named `x1, x2, …, x10`. Each variable appears in *every* constraint and has *only* integer coefficients `> 1`. Inequalities contain only the `>=` operator.

Write an implementation that can work for the base assumption as well as for as many relaxations as possible from below:

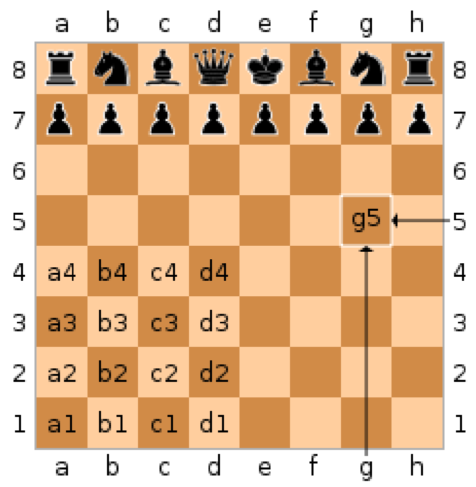
1. Number of variables is not restricted. No restriction on variable names. [5 pts]
2. Number of constraints is not restricted. [3 pts]
3. Every constraint need not contain every variable. This means that the coefficients can be either zero or non-zero ( `> 1` ). Variables may appear in any order. [8 pts]
4. Inequalities with `<=` may also appear. [5 pts]
5. Coefficients and constants can be any integer or rational number. [9 pts]

For example, if your implementation covers the base assumption and relaxations #1,#2,#3, and #4, then it should work for inputs that can have any number of variables and constraints, can have either <= and >= operators in inequalities, can have constraints where some variables do not appear, but assumes that all coefficients and constants are integers > 1.

## Q3) Let’s play Chess! (20 pts)

Given a regular 8 X 8 chessboard in initial state and a set of moves, show the resulting board after making the listed moves, using Lex/Yacc.

Each tile of the chess board is represented by a coordinate pair consisting of a letter and a number. Columns are identified with letters from *a* through *h* starting from White’s left to right. Rows are identified with numbers from *1* to *8* from White’s side of the board.



(Source: Wikipedia)

A variation of *Algebraic notation* called, *Long Algebraic Notation* would be used to describe the set of moves. Following are the symbols used in this notation to denote the appropriate pieces in a chess board.

| Symbol | Name |
|---|---|
| K | King |
| Q | Queen |
| R | Rook |
| B | Bishop |
| N | Knight |

Pawns are identified by the absence of upper-case letter.

Moves are represented by the piece's uppercase letter followed by the current position and the destination. For example: *e2-e3* denotes that a pawn moves from *e2* to *e3*; *Ra8-a6* denotes the move of a Rook from *a8* to *a6.*

When a move results in *capture,* instead of -*,* an *x* symbol is used. For example: *Ra6xa5* denotes capture of a piece in *a5* when Rook moves from *a6* to *a5.*

Pawn promotes to Queen / Bishop / Knight / Rook when it moves to the last rank in the enemy line. It is represented by an = symbol followed by the symbol of promotion. For example: *a2-a1=Q* indicates promotion of a Pawn to Queen after making a move from *a2* to *a1*.

## Input and Output notation:

Input is the sequence of numbered *turns*, where each turn contains a space separated information on move made by White followed by Black in that turn.
Output is the resulting chess board in the following format:

```
        a       b       c       d       e       f       g       h

1       <p>     <p>     <p>     <p>     <p>     <p>     <p>     <p>

2       <p>     <p>     <p>     <p>     <p>     <p>     <p>     <p>

3       <p>     <p>     <p>     <p>     <p>     <p>     <p>     <p>

4       <p>     <p>     <p>     <p>     <p>     <p>     <p>     <p>

5       <p>     <p>     <p>     <p>     <p>     <p>     <p>     <p>

6       <p>     <p>     <p>     <p>     <p>     <p>     <p>     <p>

7       <p>     <p>     <p>     <p>     <p>     <p>     <p>     <p>

8       <p>     <p>     <p>     <p>     <p>     <p>     <p>     <p>
```

Where <p> denotes the symbol for each piece. Use R, N, B, K, Q, P for Rook, Knight, Bishop, King, Queen and Pawn respectively. <p> slot may be empty if there are no pieces in that slot. There should be a tab space between each literal in the output and one new line space (i.e. two ‘\n’) between adjacent lines. Note that the white pieces start from *a1* as described earlier. It would be good to assume that the inputs would have moves only in the said format - No checks/checkmates shall be encountered, no castling, etc.

<u>Sample:</u>

Input:

1. e2-e4 e7-e5
2. Ng1-f3 Nb8-c6
3. Bf1-b5 Ng8-f6
4. Bb5xc6 b7xc6

Output:

```
        a       b       c       d       e       f       g       h

1       R       N       B       Q       K                       R

2       P       P       P       P               P       P       P

3                                               N

4                                       P

5                                       P

6                       P                       N

7       P               P       P               P       P       P

8       R               B       Q       K       B               R
```

## Appendix:

### A. How to use stack from STL?

Example:

```
#include <stack>
#include <iostream>

int main() {
  std::stack<int> s;
  s.push(5);
  s.push(7);
  std::cout << s.top() << std::endl;
  s.pop();
  s.push(11);
  std::cout << s.top() << std::endl;
  s.pop();
  std::cout << s.top() << std::endl;
  s.pop();
  return 0;
}
```

```
Output:
7
11
5
```

Member functions:

1. `push(value)` - Push value to the stack.
2. `top()` - Returns the top most value from the stack.
3. `pop()` - Delete the top most value from the stack. (Note that this does not return any value.)
4. `clear()` - Clear the stack.
5. `empty()` - Returns true if the stack is empty.
6. `size()` - Returns the size of the stack.

---

ALL THE BEST
IITH-Compilers-Admin team

---