Concordia University
Department of Computer Science and Software Engineering

SOEN341 Fall 2020 **Project B** [40%]

—

The Design and Implementation of a Cross-Assembler for a Virtual Machine

Terminology and Background

An **assembly language** program is a text source file containing the symbolic representation of a computer's binary instructions–machine language. This language is represented by symbolic names: labels, meemonics (opcodes), and operands. Labels identify and name addresses that hold instructions or data. An **assembler** is a program that translates assembly language into binary instructions as machine language object or executable files (.obj or .exe). This program refers to a resident or self-assembler, which assembles programs for the same processor it runs on. A **cross-assembler** is an assembler running on a computer or operating system (the host system) of a different type from the system on which the resulting code is to run (the target system). A **virtual machine** (VM) emulates a hardware processor that provides a physical computer's functionality. In this project:

1. **the host system** is your computer, whatever its OS (Windows, Mac, or Linux). Since we will use Java as the programming language for the implementation, it is portable on all the mentioned OS.

2. **the target system** is a VM that can be implemented on any hardware processor. Similar to the Java VM (JVM). The main difference is that the Cm VM[1] targeted in this project is greatly simplified (only 54 instructions) to support a subset of the C programming language.

An assembler makes two passes. During pass 1, the assembler traverses the assembly language code to generate the instructions (label, mnemonic, operand, etc.) that comprise the machine language code. It also generates a symbol table; each binding of the symbol table relates a label to an offset within a particular instruction. Finally, it generates offset(s) that it can be resolved and indicates offset(s) that must be resolved in the second pass. Then during pass 2, the assembler traverses the sequence of instructions to set the non-resolved offset(s).

Purpose

This project will use the Java programming language to create a cross-assembler that can process the Cm assembly language specified in the document [1]. More precisely, your code must-read an assembly language source code file and create an in-memory intermediate representation (IR) of the assembly language program. Your task is to write code that accepts the IR, performs two passes (as described above), and thereby produces an IR of the assembly program. Finally, you should traverse this IR and writes the executable file. I will provide a suite of assembly language source files [2] as a benchmark to validate your cross-assembler implementation's proper functioning.

[1]The JVM is way more complicated containing more than 202 instructions!

Before tackling the sprints, you need to respect some basic rules to fulfill the cross-assembler development. There is a preliminary task that each team must do. You should use the AUnit testing tool to your platform to regress all your tests from day one up to the due dates.

A typical assembler is composed of two main subsystems (front-end and back-end):

1. The **front-end** consists of a lexical analyzer and parser, which translate the source code into an in-memory intermediate representation (IR) built from a sequence of instructions. Two additional helpers are a symbol table (for all opcodes and labels) and an error reporter.
2. The **back-end** consists of a code generator that traverses the IR and generates assembly source code (.asm) listing and/or binary (virtual) machine code.

The cross-assembler:

- Must support three options: help, verbose, and listing.
- Must generates errors if the source format is not respecting the EBNF grammar.
- May produce a source listing (if the listing option is enabled).
- May produce a source listing and the after pass 1 and a label table (if the verbose option is enabled).

Note: The lexical analyzer must read the source code file character per character. You cannot use any Java Tokenizer or Scanner classes.

Deadline and Evaluation Criteria

For all sprints and demos, **you must work as a team. Only electronic submissions will be accepted.**

Project B [weeks 10-13] **40%**	**Due date**	**Demo during tutorial**
Team Project Sprint 1 - week 10 (starts Nov 10)	Wed, Nov 18, 11:59 PM (5%)	Thurday, Nov 19 (2.5%)
Team Project Sprint 2 - week 11 (starts Nov 19)	Wed, Nov 25, 11:59 PM (5%)	Thurday, Nov 26 (2.5%)
Team Project Sprint 3 - week 12 (starts Nov 26)	Wed, Dec 2, 11:59 PM (5%)	Thurday, Dec 3, Final Demo (5%)
Team Project Sprint 4 - week 13 (starts Dec 4)		
Team Project Deliverable	Mon, Dec 7, 11:59 PM (5%)	
Project contribution and Peer-evaluation	Mon, Dec 7, 11:59 PM (10%)	

The **team project demos** are scheduled with a Zoom meeting during tutorials. **All members MUST be present. If one of the team members is absent, or unable to answer questions, a Zero mark will be given to that member.**

The **Project contribution and Peer-evaluation** is a small individual report that all team members must submit and explain their contribution to the project. The **Team Project Deliverable** is a complete report integrating all sprints of the project, demonstration of software, and quality of the application.

Very important: Three first sprints are evaluated on Thursdays during tutorials. Most sprints (and the work) start on the Wednesdays of the week. Remember, Scrum provides two inspect-and-adapt opportunities at the end of each sprint: the sprint review and the sprint retrospective.

At the **sprint review (demo)**, the team is allowed to present only completed work. Scrum teams need to have a robust definition of done, one that provides a high level of confidence that what they build is of high quality and can be shipped. At the **sprint retrospection (just after the sprint review)**, all team members must discuss questions such as:

- What worked well this sprint that we want to continue doing?
- What didn't work well this sprint that we should stop doing?
- What should we start doing or improve?

Based on your discussions, team members must determine a few actionable changes to make and then get on with the next sprint with an incrementally improved process.

The last sprint (4) is the final sprint retrospection to prepare all your (team and individual) reports.

Definition-of-Done Checklist

The following checklist will be done by the PO (TA) during your team demo. The **definition of done** is a checklist of the types of work that the team is expected to successfully complete:

- `[ ]` Design reviewed and class diagram done
- `[ ]` Code completed
  - Code is refactored
  - Code is commented
  - Code is inspected
- `[ ]` Tested
  - Unit tested
  - Integration tested
  - Regression tested
- `[ ]` Acceptance tested

Final Team Report

Your final team report must contain:

1. Writing a domain dictionary of the 10 most important concepts and operations.
2. Using a modeling tool (www.draw.io) to make the following diagrams:
   (a) a key concept model (block diagram between 7-14 blocks maximum);
   (b) a use case UML diagram that will extract the essential functionalities, including their corresponding scenarios; and
   (c) a class UML diagram with a simplified representation of classes (no methods and no attributes), and interfaces (with their methods) with all UML relationships (dependency, association, and generalization).
   (d) a UML sequence diagram (landscape orientation) illustrating a complete assembly process by specifying the cooperation between all implied objects: fileReader, lexer, parser, symbolTable, errorReporter, irSequence, and codeGenerator.
3. Writing a report containing:
   (a) a domain dictionary (one page);
   (b) a key concept model (one page);
   (c) a use case diagram (one page);
   (d) a class diagram (one page);
   (e) a sequence diagram (one page - landscape orientation);
   (f) a full listing of all individual Java source files (one class or interface per file).

Two separate submissions must be made:

1. A final team report.
2. An individual report (1 page) that should review your own contribution (from 0 up to 5 pts) and peer-evaluation of each team member (from 0 up to 5 pts). For each contribution and evaluation, you should write a small summary paragraph.

The Team zip file should contain the PDF file of the team report, the UML diagrams files from Draw, and all Java source files. Naming convention for zip file is:

```
ProjectB_TeamXX.zip
```

The individual PDF report file must have the following naming convention:

```
ProjectB_TeamXX_StudentName_Report.pdf
```

Late submission (-1% per hour).

Documents provided for this project:
[1] The Design and Implementation of a Cross-Assembler for Virtual Machine
[2] Code Patterns: A Suite of Assembly Language Programs (.asm) for the Cross-Assembler
[3] AUnit - A Simple Language-agnostic Tool for Automated Unit Testing (including source file in Java)
[4] Dump Utility Program (including source file in Java)