COMP 442/6421 Compiler Design

Instructor: Dr. Joey Paquet paquet@cse.concordia.ca
TA: Zachary Lapointe zachary.lapointe@mail.Concordia.ca

LAB 7 – CODE GENERATION I: ASSEMBLY CODE AND THE MOON PROCESSOR

Augmented AST and Symbol Table

- AST++
  - A rich structure which represents the meaning of the program
    - Contains additional information inferred during semantic analysis
      - Type, symbol table relation
  - It is the primary artifact used for code generation (A4)
- Symbol Table
  - Contains everything the programmer named
  - We will be adding new symbols to it for code generation
  - Secondary artifact used for code generation (A4)

How to do Assignment 4

- Assignment 4 is fairly involved
  - You will likely not have time to implement every feature of the language.

1. Familiarize yourself with the Moon processing environment
   - Implement simple statements for compiler code generation
     - Read/Write → *critical for testing*
     - Simple arithmetic → requires few memory considerations
2. Pick a static memory scheme
   - What is *static* memory?
   - Tags or stack memory
3. Prioritize the implementation of language features
   - By difficulty
   - By utility
   - By grade weight

Language Features

- Read/Write statements
- Literals
- Integer numbers
  - Integer arithmetic
- Variables
- Boolean arithmetic
- Assignment
- If-else-then
- While loops
- Functions
  - Return
  - Parameters
- Recursion*
- Arrays
- Floating point numbers
  - Floating point arithmetic
- Classes
  - Variables
  - Functions
  - Access control
  - Inheritance

Moon Processor

- A *virtual* processor, with a RISC architecture
  - ***R**educed **I**nstruction **S**et **C**omputer*
- It is the *target* of our compiler
- Available from the course site
  - Documentation
  - Source code for processor (in C)
    - You'll need to compile it yourself
      - `gcc -o "moon" moon.c`
  - Sample programs
  - *Libraries*, which can be used to help code generation

- Demo: A simple moon program: `simple.moon`

Moon Processor - features

- Key Points
  - Deals primarily with 4 byte integers (Words)
  - Operations are done through registers
    - Special operations use immediate values as well (constant, literal values)
  - Integers are interpreted using **two's complement**
  - Operations are bitwise, they operate bit by bit
    - Significant for logical operators

Moon Processor – A compiled example

- Moon instructions are very simple
  - Not great for manually writing programs
- It takes many instructions to do simple things
- Generated Moon Code will be an order of magnitude longer than the original source code
- Moon programs (and assembly code) are very difficult to read at a high level
  - When generating code blocks, it would be wise to also generate comments which provide context

- Example

Testing with the Moon Processor

- Moon programs are tedious to read and write.
  - That's why we're making a compiler!
- It's very easy to make mistakes with the generated code
  - It's thus recommended to integrated the moon processor into your test environment
- To get the most out of it, you'll need to have implemented the *write()* language feature
  - Should be prioritized
  - Libraries are helpful here
- This will allow end-to-end testing of your compiler
  - Input source file → analyze compiled program's execution output

Testing with the Moon Processor

- We will have to do the following, within the test code:
  - Run the Moon processor program
  - Pass command line arguments to it
  - Verify it's process status
  - Obtain and verify it's output
- Effectively, we need to run a second program in our testing program
- Fortunately, most programming languages allow doing all of the above

- An example in Java, with Junit5

Code Generation with AST traversal

- We'll walk through an example AST, looking at when and what moon instructions are generated
- Using a tag-based approach
- Generating code with an Euler tour,
  - At pre-visits
  - At post-visits