CSE 304 – Compiler Design
Spring, 2023
Assignment 06 – Final Code Generation

Points: 40
Assigned: Never
Due: **Never**

1. Description:

In this assignment, you will complete your `Decaf` compiler by generating MIPS assembly code. For this assignment, you may assume that your input `Decaf` program is syntactically and semantically valid.

As in the past, you will produce a compiler that will take a single command-line argument, which is a file name; if that file contains a syntactically valid `Decaf` program, then it will construct an AST, perform type checking with name resolution, and intermediate code generation. Finally, (and the main step of this assignment) the compiler will generate final machine code for MIPS. It will output the machine code as a .asm file with the same name as the input `Decaf` file. For instance, if your compiler were invoked with `foo.decaf`, it should produce `foo.asm`.

Main Method:

This assignment will introduce an important assumption: that a program `foo.decaf` has a class called `foo`, which has a public, static, void method called `main` (that has no parameters).

The `main()` method will form the entry point to your entire program.

You may assume that every program used with your compiler will satisfy the above requirement, and you do not have to explicitly check for this.

On output, the code for the main method must have as a label "`main_entry_point`" at its entry point. This is to make sure that we know where to start your program execution from.

In and Out:

For the purposes of this assignment, assume that the input decaf programs do not use the input and output methods in `In` and `Out`. Again, you do not have to check to see if the program satisfies this assumption.

Simplifying Assumptions:

Students are expected to generate code for a subset of `Decaf`, the same subset used in prior homeworks. It is the whole language except the following:

- Arrays: there are no arrays in our subset of `Decaf`.
- Implicit field accesses: all field accesses in our subset of `Decaf` will specify the object explicitly.
- Implicit method calls: all method calls in our subset of `Decaf` will specify the object explicitly.
- No overloaded methods/constructors: Assume that in each class, there is at most one constructor, and at most one method with a given name. Also assume that the names of the methods defined in a class are distinct from the method names in its super classes. You do not need to check for these constraints. Your type-checker can safely assume these are true.

2. Available Material:

MIPS and SPIM:

MIPS has been used for over 25 years in undergraduate Computer Organization and Assembly Language Programming courses. So it also has tons of very good reference material.

- SPIM's pages on SourceForge: http://spimsimulator.sourceforge.net/
  You can download SPIM from there and it has links to very useful documents on SPIM and MIPS.
- A very nicely written guide to Assembly Language Programming using MIPS, by Daniel Ellard. This (old is gold) tutorial is here: https://ellard.org/dan/www/Courses/cs50-asm.pdf
- **Offline Resource:** Britton, MIPS Assembly Language Programming, Prentice Hall, 2004. (ISBN 0-13-14204405, 978-0131420441) is a textbook used in CSE 220.

3. Project Path:

1. Start with building control flow graphs for each function in the intermediate code.
2. Convert the graphs into SSA form.
3. For each function, generate final code by:
   a. Mapping temporary registers of the abstract register machine to MIPS registers.
   b. Translating abstract machine instructions to MIPS instructions.
4. Go through MIPS emulator SPIM. (You may use MARS if you are more familiar with it, but my personal recommendation is to use SPIM.)
   a. First generate code for programs that do not access objects and have no dynamic memory allocation (halloc).
   b. Implement halloc using features of the emulator.
   c. Specify .data layout to implement the static area and heap in MIPS.
   d. Output final code to a .asm file that can be immediately executed in the emulator.

4. Deliverables:

Your AST checker for `Decaf` should be organized into the following files:

1. **Lexer:** `decaf_lexer.py` – PLY/lex scanner specification file.
2. **Parser:** `decaf_parser.py` – PLY/yacc parser specification file.
3. **AST:** `decaf_ast.py` – table and class definitions for `Decaf`'s AST.
4. **Type Checker:** `decaf_typecheck.py` – definitions for evaluating the type constraints and for name resolution.
   - **Note:** This may be folded into decaf_ast.py as long as readability is maintained.
5. **Code Generator:** `decaf_codegen.py` – definitions for generating code
   - **Note:** This may be folded into decaf_ast.py as long as readability is maintained.
6. **Abstract Machine:** `decaf_absmc.py` – definitions for the abstract machine and for manipulating abstract programs (e.g., printing)
7. **Main:** `decaf_compiler.py` – containing the main python function that ties the modules together, takes the command-line argument of the input file name, and processes the `Decaf` program in that file.

   The translated MIPS program will be written to a file with the same base name as the input file, but with a ".asm" extension.

8. Generate any other files necessary to complete the translation process from abstract register machine code to MIPS code—along with completing and using any required static analysis along the way. Make sure that the purpose of any additional files are clearly explained in the README.
9. **Documentation:** `README.txt` which documents the contents of all the other files.