Final Project: CMPT-379

Due by Dec 15

Anoop Sarkar – anoop@cs.sfu.ca

(1) † **Decaf Compiler**

Submit your compiler as a self-contained package that can be used to compile **Decaf** programs into MIPS assembly and subsequently execute them using the spim simulator for the MIPS processor. Make sure that the binary of your compiler can be created by running make.

Create a script called decafcc (or decafcc.sh) that is used to run the entire compiler chain from lexical analysis to code generation to running the MIPS simulator (assume spim is in the PATH).

In your submission, provide in a subdirectory called positives any number of **Decaf** programs that work with your compiler (the programs should be valid **Decaf** based on the language definition and execute using spim) along with the legitimate output for that **Decaf** program, e.g. for a program called exprTest.decaf also include the legitimate output in a file called exprTest.decaf.output. Also provide a subdirectory called negatives with **Decaf** programs that should exit with an error. Your makefile should include an entry such that when make testall is run, it should run your **Decaf** compiler on *all* the **Decaf** programs in the positives and negatives directory.

Please check that you do not have any spelling errors in the names of the directories and that the directories are relative and not absolute, e.g. use ../testdir/ rather than /home/username/cmpt379/testdir (to enable automatic testing). Also, for non-trivial **Decaf** programs, provide a readme file explaining the code and the desired output or why it should not produce any output (the syntax or semantic error involved). For exprTest.decaf provide a readme file called exprTest.decaf-readme.txt.

Note that in your makefile, to ignore errors in a command line that executes a program, write a '-' at the beginning of the line's text (after the initial tab). The '-' is discarded before the command is passed to the shell for execution. For example,

```
test:
	-sh decafcc.sh negatives/exprTest.decaf > negatives/exprTest.decaf.output
```

You could try to break the compilers written by your peers, but only if your compiler can survive those **Decaf** programs itself.

(2) † Extending your **Decaf Compiler**

You should implement some non-trivial extension of your **Decaf** compiler for your final project.

Here are some ideas for projects. However, you don't have to use these in particular for your project. Perhaps these possible project ideas might trigger ideas of your own.

a. Do proper (optimal) register allocation and spilling in your **Decaf** compiler. Provide benchmark tests to confirm running time improvements with and without the more advanced algorithm for register allocation and spilling. This will also involve writing some **Decaf** programs programs that are challenging for the non-optimized register allocation strategy.

b. Target a different CPU (from the one you used in your assignments) and perform code generation for it. e.g. use LLVM as your backend intermediate representation instead of MIPS in order to target x86 CPUs. You should target the more advanced CPU instructions in the chipset to speed up your compiled **Decaf** programs.

c. Use data-flow analysis or peephole techniques to perform instruction level code optimization. Provide benchmark tests to confirm running time improvements with and without the optimizer.

d. $*$ Write a debugger (like gdb) for **Decaf**.

e. Extend the **Decaf** language to support one or more of the following programming language concepts. In each case, provide example **Decaf** programs that exploit the new language features:

1. Introduce memory allocation/de-allocation commands into the **Decaf** standard library to allow **Decaf** programs to allocate and manage memory on the heap. e.g. you might implement a function `new` that creates space in the heap for objects based on class definitions in **Decaf**. The reference to the object should be passed to function calls, and the **Decaf** program is responsible for eventually freeing up the heap memory.
2. Implement closures and other functional programming concepts in **Decaf**.
3. Add new types to **Decaf** like double or float. Create a dynamically typed version of **Decaf** that still produces proper types for registers during code generation. This will involved changing the language definition.
4. Implement object-oriented concepts like inheritance and/or interfaces in **Decaf**.
5. $*$ Implement garbage collection (or use an existing library) to allow automatic allocation and de-allocation of memory on the heap.
6. $*$ Write variants of **Decaf** that exploit hardware capabilities like multi-core, distributed computing, or GPUs.