Decaf PP4: Gode Generation

Date Due: **05/12/2019 Sunday 11:59pm**

1 Goal

In this project, you will implement a back end for your compiler that will generate code to be executed on the SPIM simulator. Finally, you get to the true product of all your labor-running Decaf programs!

This pass of your compiler will traverse the abstract syntax tree, stringing together the instructions for each subtree - to assign a variable, call a function, or whatever else is needed. The instructions need to be in the form of MIPS assembly. Your finished compiler will do code generation for all of the Decaf language as well as reporting link errors.

The debugging can be intense at times since you may need to drop down and examine the MIPS assembly to sort out the errors. By the time you're done, you'll have a pretty thorough understanding of the runtime environment for Decaf programs and will even gain a little reading familiarity with MIPS assembly.

Your program needs to take decaf source code as input and output .s file which are executable on SPIM simulator. The spim executable, which actually executes the code can be found in the posted package for phase 4 (directory "/spim"). The -file argument allows you to specify the file of MIPS assembly to execute.

NOTE: You need to concatenate defs.asm to the end of your .s code because `defs.asm` contains the utility functions (e.g., Print and ReadInteger). `defs.asm` is in the posted package of the phase.

```
% cat defs.asm >>program.s
% ./spim -file program.asm
```

2 Building Your Own SPIM Simulator If Necessary

The SPIM simulator executable is provided under the "spim" directory. This section provides information for compiling SPIM on your own machine if the provided executable does not work for you. The SPIM simulator source code is also available in the package of phase 4, under the "spimsource" directory.

By default, the SPIM simulator will be installed on your */usr/bin* directory. If you want to install it on a different location, please change the "BIN_DIR", "EXCEPTION_DIR" and "MAN_DIR" in the *Makefile* to desired locations.

Then in the *spimsimualtor/spim* directory, type in the following commands to install the SPIM simulator:

```
% make
% make test
% make install
```

For the folks who have problem with *make test*, you can download the latest version of SPIM simulator from Source Forge. The SPIM source code in decaf is incompatible with (at least) newer Linux distributions, and on these distributions, the *make test* will run forever. Therefore, if you have issues with compiling and using SPIM from the provided source code, you can directly download the executable or source code for SPIM from http://spimsimulator.sourceforge.net/. Executable is provided for all major OSes on Source Forge.

The SPIM executable in PP5.zip was compiled and tested on fox servers, so it is safe to compile SPIM on fox servers. This executable is statically linked, which should work on all Linux boxes.

3 Extra Points

Students who finished the main project can further implement the register allocation for 5 extra points. To get extra credit, you need to (1) pass all given tests for phase IV; (2) implement liveness analysis, coloring algorithm and register allocation in MIPS code generation. Note that students whose phase IV submission does not pass all 5 given tests for phase IV cannot get any extra points.

For the extra analysis, your program needs to output the 3-address code (in any readable form) and live analysis result (the set of live variables) for each program point. Then, your program needs to perform color algorithm to assign each variable to registers. Your program needs to further output the mapping of variables to registers, and the MIPs code based on the mapping. The TA and instructor will manually grade this part.

4 Code Generator Implementation

Here is a quick sketch of a reasonable approach:

- Plot out your strategy for assigning locations to variables. A Location object is used to identify a variable's runtime memory location, i.e. which segment (stack vs. global) and the offset relative to the segment base. Every variable, be it global or local, a parameter or a temporary, will need to have an assigned location. First of all, you try to deal with local variables, global variables, and temporaries (eventually you will also support parameters and instance variables). Figure out how/when you will make the assignment. As a first step, you may want to print out each location before doing any code generation and verify all is well. If you aren't sure you have the correct locations before you move on to generating code, you're setting yourself up for trouble. Once you have assigned locations for all variables located within the stack frame, you can calculate the frame size for the entire function, and backpatch that size into BeginFunc instruction.
- The label for the main function has to be exactly the string "main" in order for Spim to start execution properly.
- Start by generating code to load constants and add support for the built-in Print so you can verify you've got this part working. Simple variables and assignment make a good next step.
- Generate instructions for arithmetic, relational, and logical operators.
- Generating code for the control structures (if/while/for) will teach you about labels and branches. Correct use of the break statement should work for exiting while and for loops.

After this, you should be able to handle any sequence of code in a single main function. Proceeding with the rest of code generation includes:

- Generating code for other function definitions isn't much different than it was for main, other than that you need to assign locations to the function parameters and figure out your strategy for assigning function labels. Our solution uses the function name prefixed with an underbar as the function label and for classes we further prefix with the class name. You're welcome to use any scheme you like as long as it works (i.e. assigns unique labels with no confusion).
- You are to add one piece of "linker"-like functionality to verify that there is a definition for the global function *main* The error reported when the program contains no main is:

  ```
  ***Linker: function 'main' not defined
  ```

  If there is a link error, no code should be emitted.

5 Testing your compiler

We provide 5 test files for you to test your code generator. If the source code has linking error, no code should be emitted at all. For t4.dcaf, the program is interactive, so we do not provide the output. But you can run the MIPS code to see what is the expected behavior of the program. For each source file, we provided the reference MIPS code. Your MIPS code does not necessarily to be exactly the same with them. You can get full points as long as your final output is as expected.

6 Grading

The final submission is worth 5 points. We will thoroughly test your submission against code samples. We will run the object code produced your compiler on the spim simulator and diff against the correct runtime output.

7 Deliverables

Electronically submit your entire project to Blackboard. You should submit a tar.gz of the project directory. Be sure to include a brief README file.

Because we grade the submissions using scripts, it is important that everyone uses the same directory structure. The uploaded folder should contain your source code, all dependencies, and a sub-folder called 'workdir' where you should put two files 'build.sh', and 'exec.sh'. Running 'build.sh' should build your project, and running 'exec.sh <filepath>' should execute your compiler on a source code file at <filepath>', and all output should be written to the standard output stream, so that the grader can use 'exec.sh <filepath> > <outputpath>' to redirect your output into files and compare with the ground truth. For t4.dcaf, we will check manually.