Project 2: Program analyses, code transformations, and improved code generation — assigned incrementally and finally due on Sunday 7 December — final presentations on Thursday 11 December (the official final exam day)

This third and final incremental release consists of Tasks 7–11. Release date 24 November. Submissions due 7 December.

Project 2 is a continuation of Project 1. You will now implement an optimizing compiler for the language WHILE.

Task 1: Consolidate all work from Project 1. (Tidy up and fix any problems you may have noticed.)

2.1 Labelled syntax

The labelled syntax decorates computational blocks ("program points") with labels ℓ:

Commands c ::= $[x := a]^\ell \mid [\mathsf{skip}]^\ell \mid c_1; c_2 \mid \mathsf{if}\ [b]^\ell\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \mid \mathsf{while}\ [b]^\ell\ \mathsf{do}\ c_1\ \mathsf{od}$

Task 2: Given an AST, produce a decorated AST with unique labels (conventionally, we use distinct numbers as labels). Display the decorated AST. Display (pretty-print) the annotated program in source form (with labels inside comments inserted at the appropriate place).

2.2 Control flow graphs

Task 3: Come up with a data structure (whichever is convenient in your implementation language) for representing control flow graphs. Convert the decorated AST into a CFG. Display the CFG.

2.3 Code generation

Task 4: From the CFG, generate virtual machine code. The code will superficially look like RISC-V assembly. Each variable of the WHILE program will be mapped to a RISC-V register, `s1`, `s2`, ..., and we will pretend that there are as many `s` registers as we need for all the variables in the program.

You should use comments in the generated virtual RISC-V machine code to document which parts of the CFG / WHILE source the code corresponds to.

The function prologue will include instructions like this:

```
#s1<-input
ld s1,8(a0)
```

to load all the `s` registers from the array that was passed by reference from the main program (in register `a0`), and the function epilogue will include instructions like this:

```
#output<-s10
sd s10,80(a0)
```

to dump all the `s` registers back into this array so the main program can then print the final values of the variables.

For smaller WHILE programs, with no more than 11 variables, the generated virtual RISC-V machine code will be valid RISC-V assembly, which you can assemble and link with the generated C main program (which remains the same as in Project 1) and then run on our RISC-V machine.

2.4 Testing and performance evaluation

Task 5: Extensively test your compiler. If you haven't already, write unit test programs to demonstrate complete coverage of WHILE language features.

Task 6: Evaluate the performance of the generated code. Measure execution times using some example programs with few variables. Compare execution times achieved by your current Project 2 compiler with those achieved by your Project 1 compiler. (This is only a preliminary evaluation, so do not spend too much time on it, but you should already observe considerable improvement.)

2.5 Liveness analysis

Liveness analysis is used for register allocation. It can also detect dead code.

Task 7: Formally define `gen` and `kill` sets for the liveness analysis problem for each relevant construct in the language WHILE. Then implement liveness analysis (LA). The analysis will take as input (1) a CFG and (2) a list of variables assumed to be live at the exit from the program. Define a data structure for liveness analysis equations. Produce the liveness analysis equations, and print them.

Task 8: Solve the liveness analysis equations. The output will be an annotation of all labelled points with `in` and `out` sets (in the notation we use in class this would be written $\mathrm{LA}_\circ$ and $\mathrm{LA}_\bullet$). Print that solution.

2.6 Dead code elimination

From this point on, i.e., in Tasks 9–11, assume that the program variable `output` is special: it is assumed to be live at the exit from the program. All other variables are assumed to be dead at the exit from the program.

In Tasks 9–11, use the results of liveness analysis to identify which variables are live at the entry to the program. Adjust how the main program (in C) is generated, such that when it accepts initial values of variables from the command line it only takes values for the variables that are live at entry (but still in alphabetical order). The main program still needs to pass the array with slots for all source program variables; the non-live variables' slots can be left uninitialized.

Task 9: Apply the results of liveness analysis to eliminate dead code.

2.7 Register allocation and code generation

Task 10: Apply the results of liveness analysis to perform register allocation. Use RISC-V registers `s1`–`s11` to hold variables. Any variables that do not fit in these eleven registers must be spilled. (Normally they would be spilled to the current stack frame. We already have a dedicated area for all variables that appear in the source WHILE program, namely the array that was passed to the compiled code by reference. We can simply use this area for spills. However, if you have introduced additional variables those will need to be spilled to the stack frame.) Having allocated variables to registers, generate RISC-V assembly code.

2.8 Testing, performance evaluation, and analysis

Task 11: Once again evaluate the performance of your compiler.

Evaluate the performance of the generated code. Measure execution times for a set of benchmarks.

Compare execution times achieved by your final Project 2 compiler with those achieved by the Project 1 compiler.

What to turn in

Submit all the code and all the tests in source form and a suitably compiled form (depending on your implementation language), and a project report. In the project report, **please remember to describe your team composition and how the team collaborated on the task, and comment on the level of effort needed for the project**.