Pre-Proj4 Alternate Assignment

Member: Evelyn Vo

1) What bugs in your compiler prevented you from achieving correctness? Provide minimal examples of code your compiler is unable to handle for each of the bugs.

For personal reasons (I can expand if asked), I ended up not being able to spend as much time on the project as I would have liked. The biggest issues aside from that that kept me from achieving correctness was that I was trying to add RISC-V code files. I kept getting a bug when running tests (after calling "mvn clean package") that said that "Venus expected X number of arguments but got Y many on lines Z". I couldn't figure out if I added the code incorrectly and where the problem was as searching in my code base wouldn't show the lines flagged by Venus. It turned out to be some syntax errors in the RISC-V files ("addi t0, 16" was used instead of "li t0, 16").

The compiler wasn't able to run the binary expression code generation as a result.

2) Pick three optimizations from the list earlier, and for each one:
– Explain in English with pseudocode how you would implement this on top of a working PA3.
– Take a specific benchmark program from the class submissions, and provide examples of how the optimization would affect the RISC-V (please show both before and after).

(Regarding the benchmark changes, I wasn't able to complete the checkpoint and wouldn't be able to show the cycle count differences.)

a) Boxing elimination (don't box integers and booleans unless necessary)

To eliminate boxing for integers and booleans, we would need to have 2 passes of the RISC-V code. In the first pass, we would generate the code as in PA3, but replace any use of the integers / booleans and their boxed values with labels. This first pass would have no boxing as a result. Then, in the second pass, if we see that the integers / booleans are called by a function (i.e. print) that requires an object to be passed or are used in some other computation requiring them to be boxed, we insert the code to box those values and replace the labels as necessary.

i.e. for "print(1 + 2)"

pass 1:
print(1 + 2) >> print(LABEL_1 + LABEL_2) >> print (LABEL_3)

Within the binary expression (LABEL_1 + LABEL_2), we know that boxing would not be necessary in calculating its result, so there is no boxing code inserted when resolving the PRINT arguments. We represent the output with LABEL_3.

pass 2:
print(1 + 2) >> print(LABEL_1 + LABEL_2) >> print (LABEL_3) >> print (BOXED_3)

Since we know that the value represented by LABEL_3 needs to be boxed before it is passed to PRINT, we insert code boxing the value represented by LABEL_3.

Benchmark: exp.py
When finding the exponent of a(x), the benchmark also calculates a(x - 1), a(x - 2), ..., a(0). Print is not called between calls to exp(x: int, y: int) as f(int i) only calculates the next value to return. Thus, the optimized compiler would not box the intermediate values a(x - 1), a(x - 2), ..., a(0) when calculating a(x) and would only box the final value as it will be passed to print(...), which requires an object.

**b) Common subexpression elimination (use existing computation results rather than re-computing
the same expression later)**

We can create a dictionary where the key is the operation / subexpression using known values and the key is the label representing the resulting object / value. Much like the symbol table, we can have one that represents expressions belonging to the outer frame surrounding the current frame (ex: involving global values) and one representing expressions belonging to the current frame. When the expression is first encountered, add it to the current frame's dictionary. Then, if it is encountered again, replace the expression with the label corresponding to that subexpression result.

Benchmark: exp.py
When finding the exponent of a(x), the benchmark also calculates a(x - 1), a(x - 2), ..., a(0). Since the labels of a(x - 1), a(x - 2), ..., a(0) would already be in the dictionary, the runtime of each consecutive call to a larger x would go from running as many cycles to call on all the previous x values to a constant value (finding the correct label, loading the label value, calculating the result).

c) Register allocation (instead of pushing temporaries to the stack, use registers)

Within the Java code, we can create an ArrayList (or other List) within the Stmt Analyzer initializer that contains all the registers not in use. When a temporary would be pushed to the stack in PA3, pop a register from the list (getUnusedRegister). If none can be popped, save the value on the stack. When the register is no longer needed / no longer in use, add it back to the List.

```
Register getUnusedRegister() {
   if (registers_not_in_use.size() > 0) {
      // pop register
   }
   return null // save on stack if null
}
```

We can use Labels to represent the registers and the addresses of values that had to be saved on the stack. This way, functions will know what to refer to when trying to access the values.

Benchmark: exp.py

When finding the exponent of a(x), the benchmark also calculates a(x - 1), a(x - 2), ..., a(0). This leads to at least x-1 many nested calls. We can pop registers before starting the nested calls, store the function args in the registers, call the function, and return the registers to the list of unused registers after checking the return statement does not use values within the unused registers. Since f(int i) has only one arg and returns only 1 arg, the compiler ends up never having to store on the stack.