Code generation

Comp 520: Group 9 - Wei, Omar, Deepanjan

We decided to generate jasmin assembly code so that the jasmin assembler can produce Java byte code. We have spent lot of time exploring Java byte code, including which primitive types are supported and which binary/unary operators need to be implemented. For this milestone, we printed the jasmin instruction directly from prettyPrint (exp_gen.ml), so we can have an idea which parts are difficult. For the final code generation, we will translate the go ast to a jasmin ast, and write a code emitter that takes a jasmin ast and produce jasmin code. The basic structure of the jasmin ast and code emitter is written and for details about the code generation design, see code_gen_design.pdf in the same folder.

List of things implemented for milestone 3:

- Types: int, float, string, (bool)
- Function declarations: both void, and with return types, and with arbitrary number of arguments. Note: since java requires String[] args for its main() method, we need to always generate "[Ljava/lang/String;" as its arguments.
- if statements: Expects a bool in the stack (0 or 1). Uses a label to skip the the else portion.
- print/println: the only difference between the two is invokevirtual java/io/PrintStream/**println** vs invokevirtual java/io/PrintStream/**print**, we are using 2 very similar helper functions for now, they need to be combined.
- return: similar to print, return instructions are type-dependent, so we need to find out the expression type. Note: we always generate a return instruction for void functions because all methods in java bytecode need a return.
- empty statement
- expression statement
- expressions :
    - unary: for boolean type operations, it is assumed that false = 0, true = 1 (we may need to change to true !=0 because apparently that's how boolean is usually implemented)
    - binary: mostly implemented except for label jumping instructions. Since we won't be directly printing the jasmin instructions for the final code generation. It should be handled more elegantly.
      -operators are implemented assuming both expressions are on top of the stack.
    - function calls: since we didn't implement variable-related instructions, it only works if we don't use any function arguments inside a function.
    - literals: int, float64, string