CSC 453 Milestones 11-14, and Final Project

Goal:

These milestones build the code generator for Tau in a series of steps until the whole project is finished.

Common Interface to module `codegen`

```
def generate(p: ast.Program) -> [Insn]:
    ...
```

The code generator will be implemented in the file `codegen.py`.

Milestone 11: Just calls and prints

Milestone 11 may be the hardest milestone. Start early.

Milestone 11 will generate code for the following AST declaration, statement, and expression nodes:

- `Program(AST)`
- `FuncDecl(AST)` (for m11, exclude parameter passing)
- `PrintStmt(Stmt)`
- `CompoundStmt(Stmt)`
- `IntLiteral(Expr)`
- `IdExpr(Expr)` (for m11, only function names as identifiers)
- `CallStmt(Stmt)`
- `CallExpr(Expr)` (for m11, no parameters/arguments)

A complex program the compiler should handle for M11:

```
func g(): void {
    print 1
}
func f(): void {
    print 3
    call g()
    print 5
}
func main(): void {
    print 4
    call f()
    print 6
    call g()
    print 7
}
```

Milestone 12: Adding locals, parameters, returns, and assignments

Milestone 12 will add the following:

- `ReturnStmt(Stmt)` (including return values)
- `IdExpr(Expr)` (include local variables and parameters)
- `CallStmt(Stmt)`
- `CallExpr(Expr)` (for m11, **include** parameter passing)
- `AssignStmt(Stmt)`
- `BoolLiteral(Expr)`

I would recommend adding the unary and binary operators if you have time, to get a head start on Milestone 13.

Milestone 13: Expressions and control flow

Milestone 13 will add the following:

- `IfStmt(Stmt)`
- `WhileStmt(Stmt)`
- `BinaryOp(Expr)`
- `UnaryOp(Expr)`

Milestone 14

Milestone 14 will add the following:

- Recursion

Final Project

Same as Milestone 14, but with the following additions:

- Error messages

Specifications

The Tau language specification is a separate document.

`main.py` controls compilation

The provided file `tau/main.py` will control the compilation process. It will call the parser, the semantic analyzer, and the code generator. It will also call the VM to execute the generated code.

Invocation is straightforward:

```
python3 -m tau.main.py --file source.tau
```

There is also a `--verbose` option that will cause the VM to print out the instructions as they are executed.

Calling Convention

The calling convention for Tau has been described in class. This is a summary:

Caller Invocation:

- Caller put's outgoing arguments at negative offsets from its frame pointer. (The first is at offset −2, the second at offset −3, etc.)
- Caller uses `Call` instruction to call a function.

Caller Upon Return:

- Caller makes sure the return value is fetched from offset `-1` from the `FP/SP` stack if appropriate. (See below.)

Callee Prologue:

- Callee saves its return address at offset 0 from its FP
- Callee saves its caller's FP at offset 1 from its FP
- Callee *may* save caller's SP at offset 2 from its FP, but is not obliged to do so.
- Callee allocates its own frame, adjusting the FP and SP as appropriate.
- Callee's FP **must** be equal to the caller's SP.

Callee Epilogue:

- Callee must restore the caller's FP and SP before returning.
- Callee must put the return value at offset −1 from its FP if appropriate.

Errors

The milestones only requires that you correctly annotate a correct Tau program.

Grading

Standard milestone grading applies.

The final project will be graded based on passing test cases as well as an examination of the code. (I.e., there is no 80% rule for the final project.)