Translator Design - Lab 1

1. Write a program in IMP to read an integer n and print $n + 100$.
2. The lexer cannot parse integers yet, failing with an error saying the character '1' was not expected. Implement integer tokenisation.
   (a) Add a static factory method to the token class, constructing a token from a location an a `uint64_t` payload.
   (b) Adjust the copy constructor (`Token::Token(const Token &t)`) and the assignment operator (`Token &Token::operator=(const Token &t)`) to also copy the integer payload of the token.
   (c) In `lexer.cpp`, find a suitable point to detect the presence of digits, convert them to an integer and return the appropriate token.
3. At this point, the compiler should be failing with a parser error at the `INT(100)` token. Extend it to accept integers.
   (a) In `ast.h`, add a new expression kind to represent integers and create another subclass of `Expr` to represent integers.
   (b) In `parser.cpp`, extend the `ParseTermExpr` method to also handle integers, constructing an instance of the node you have defined earlier.
4. Add an opcode to encode constant integers.
   (a) In `program.h`, define and document a new opcode `PUSH_INT` which will place a constant at the top of the stack.
   (b) In `interp.cpp`, implement the evaluation rules for the new opcode. Decode a 64-bit signed constant from the program (similarly to the way offsets are defined for `PEEK`) and push it onto the stack.
5. Compile the integer node to the newly defined opcode.
   (a) In `codegen.cpp`, find the point where the node is traversed and emit the opcode followed by the integer payload of the node as its sole argument.
   (b) The AST node encodes an unsigned constant, while the instruction and the interpreter expect a signed one. Find an appropriate bound and enforce it at some stage of compilation.
6. In some instances, the `ADD` opcode can produce invalid results. Instead of silently continuing, throw a runtime error if this happens.