Today

- Wrap up MyPL from lect-1
- Getting started with HW-1
- PL implementation basics

Assignments

- Exercise 1 (due next Tues)
- HW-1 out (due next Thurs)
Compilation (typically) involves the following steps (pipeline)

1. identify language “tokens” in source ... “lexer/scanner”
2. ensure syntax correct ... “parser”
3. ensure program “correct” (use-before-def, type errors) ... “static analyzer”
4. generate intermediate representation (e.g., for optimization)
5. improve performance of code ... “optimizer”
6. generate executable code (machine code) ... “code generator”

Example of “separation of concerns” (engineering design strategy)
- too complex to do “all at once” (single-pass vs multi-pass)
- also easier to manage, maintain, update/improve, reuse
PL Implementation Basics: Interpretation

Abstract Syntax Tree (AST) Interpreters
- execute the program directly from the AST

Bytecode Interpreters (aka VMs)
- intermediate representation is bytecode
- interpreter runs bytecode directly ... “write once run anywhere”

Just-in-time Compiler (JIT)
- instead of interpreting bytecode, generates and runs machine code
- monitor running code (e.g., frequent “hot spots”) and optimize accordingly
**Additional Notes on Approaches**

**Transpilers:**
- Convert from one language into another
- Often include same “front-end” compilation steps (e.g., to an AST)

**Transpiler vs Compiler:**
- Compilers typically go from high-level to low-level languages
- Transpilers typically go from high-level to high-level languages

**Compiler vs JIT:**
- JIT sometimes called a “hybrid” approach (between compiled and interpreted)
- Popular implementation approach today

**Other places where similar approaches used:**
- Read-Eval-Print-Loops (REPLs)
- Integrated Development Environments (IDEs)
- Domain-Specific Languages (DSLs)
- “Data” Languages (e.g., HTML, JSON, XML, SQL, Graph QLs)