Lecture 4:

- Compilation and Interpretation (cont)
- Lexical Analysis

Announcements:

- HW-1 out
- Quiz 1 on Friday: MyPL (e.g., write code), Compilation/Interpretation steps
**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)
Lexical Analysis – Tokens

Tokens are the smallest meaningful units of a program

Some examples:

- Special words ("reserved" words)
  
  `int, if, while, new, class, public, and so on`

- Operators and Punctuation
  
  `+, =, ==, <=, (, ;, ., and so on`

- Identifiers
  
  variable names, function names, class names, etc.

- Constant (i.e., "literal") values
  
  `42, 3.14, true, "abc", and so on`

- Others (e.g., comments, annotations)

White space (usually) not tokens

- some exceptions such as Python

Tokens include a type and a lexeme (a value)

- the lexeme is just the token's value in the source file

- e.g., in the statement: `x = 42;
  
  - the token types might be ID, ASSIGN, INT_VAL, SEMICOLON
  
  - and the corresponding lexemes "x", "="", "42", and ";"

- for some tokens, the lexemes are needed for program execution

  - e.g., the variable name of the identifier ("x") and the int value "42"
**Lexical Analysis – Basics**

**Goal:** simplify syntax analysis (parsing) and detect (token) errors early

- a “lexer” only deals with building tokens, not checking how they “go together”
- allows parser to focus on checking syntax rules (separation of concerns)

**The basic idea:**

**Source Code:**

```
int f() INT_TYPE("int"), ID("f"), LPAREN("(") , RPAREN(")"),
{ LBRACE("{"),
  int x = 0; INT_TYPE("var"), ID("x"), ASSIGN("="), INT_VAL("0"), SEMICOLON(";"),
  return x; RETURN("return"), ID("x"), SEMICOLON(";"),
} RBRACE("}")
```

**How it works:**

- Source code converted to a sequence (or a stream) of tokens
- Skip over non-tokens (white space, comments)
- Keep line and column numbers as part of tokens

**Note:**

- a sequence is similar to a list
- a stream is similar to an iterator
Check in: Give the token sequence (token type, lexeme, line, column) for the following MyPL code snippets. Assume the token types:

- ASSIGN, ID, INT_VAL, LPAREN, RPAREN, LBRACE, RBRACE, LESS_EQ, PLUS, STRING_VAL, WHILE, EOS

Snippet 1:

```plaintext
print("Hello World!")
```

Snippet 2:

```plaintext
int x = 0;
while (x <= 10) {
    x = x + 2;
}
```
Summary – Things to Know

1. Difference between compilation and interpretation (steps)

4. The basic idea of a bytecode interpreter

5. What is meant by a transpiler

6. Whether a given language’s primary implementation is an interpreter or a compiler (e.g., C/C++, Python, Java, ASM)

7. What a token represents and its basic components

8. Given a code snippet, the corresponding token stream