Lecture 4:

- Lexical Analysis

Announcements:

- HW-1 out (due Mon, 2/3)

Lexical Analysis: Tokens

Tokens are the smallest meaningful units of a program

- Reserved ("special") words ... `int`, `if`, `while`, `new`, `class`, `public`, etc.
- Operators and Punctuation ... `+`, `=`, `==`, `<=`, `(`, `;`, `.`, etc.
- Identifiers (names) ... variable, function, class names, etc.
- Literal ("constant") values ... `42`, `3.14`, `true`, `"abc"`, etc.
- Others ... e.g., whitespace, annotations

Tokens include a type, a lexeme, and a location

- the lexeme is just the token's value in the source file
- the location is typically the line and column where the token occurs
- the type and lexeme are useful for literals and identifiers

Lexical Analysis: Tokens

Consider the assignment statement: `x = 42;`

- token types might be: `ID`, `ASSIGN`, `INT_VAL`, `SEMICOLON`
- the corresponding lexemes "`x`", "`=`", "`42`", and "`;`"
- the locations (line, column): `(1,1)`, `(1, 3)`, `(1,5)`, `(1,7)`

Check In: What about for:

... assuming types: `INT_TYPE, ID, LPAREN, RPAREN, LBRACE, RBRACE, VAR, ASSIGN, STR_VAL`

```
int main() {
  var x = "hi!"
}
```

```
INT_TYPE("int", 1, 1), ID("main", 1, 5),  LPAREN("(", 1, 6),  LPAREN(")", 1, 7),
LBRACE("{", 1, 9),  VAR("var", 2, 3),  ID("x", 2, 7),  ASSIGN("=", 2, 9),
STR_VAL("hi!", 2, 11),  RBRACE("}", 3, 1)
```


Lexer Basics

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

- Lexer only deals with building tokens, not checking how they "go together"
- Allows parser to focus on checking syntax (separation of concerns) (*)

What the Lexer does:

- Lexer converts source code into a stream of tokens
- A stream is a sequence but produced one-item-at-a-time (like an iterator)
- Skips over non-tokens (white space)
- Tracks line and column numbers, finds "illegal" tokens

(*) parser also converts token sequence into a tree-like structure (more later)

Lexer Basics: Errors

Our Lexer will only detect two types of errors:

- unexpected characters/symbols (like an exclamation mark)
- poorly formed string and number literals

Dealing with errors

- lexer returns a special error token ... enters "panic mode"
- lexer raises an exception ... what we'll do
- compilers stop (e.g., Python) or keep going (e.g., C++, Java)

MyPL Lexer: Token Types

```
public enum TokenType {
  // punctuation symbols
  DOT, COLON, COMMA, LPAREN, RPAREN, LBRACKET, RBRACKET, LBRACE, RBRACE,
  // arithmetic operators
  PLUS, MINUS, TIMES, DIVIDE,
  // assignment and comparator operators
  ASSIGN, EQUAL, NOT_EQUAL, LESS, LESS_EQ, GREATER, GREATER_EQ,
  // primitive values and identifiers
  STRING_VAL, INT_VAL, DOUBLE_VAL, BOOL_VAL, NULL_VAL, ID,
  // boolean operators
  AND, OR, NOT,
  // data types
  INT_TYPE, DOUBLE_TYPE, CHAR_TYPE, STRING_TYPE, BOOL_TYPE, VOID_TYPE,
  // reserved words
  STRUCT, VAR, WHILE, FOR, FROM, TO, IF, ELSE, NEW, RETURN,
  // comment token and end-of-stream
  COMMENT, EOS
}
```

Note: `EOS` = end-of-stream, types vs values (e.g., `INT_TYPE` vs `INT_VAL`)

MyPL Lexer: Token Class

```
public class Token {

  public TokenType tokenType;  // the token's type
  public String lexeme;        // the token's string representation
  public int line;             // the line the token appears on
  public int column;           // the column the token appears on

  // constructor
  public Token(TokenType tokenType, String lexeme, int line, int column) {
    ...
  }

  // pretty print a token
  public String toString() {
    ...
  }

}
```

Note: Easy access via (public) member variables (e.g., `t.lexeme`)

MyPL Lexer: Lexer Class

`Lexer` implements `nextToken()` by reading lexeme one-character-at-a-time

```
public class Lexer {
  private BufferedReader buffer; // handle to the input stream
  private int line = 1;          // current line number
  private int column = 0;        // current column number

  // constructor
  public Lexer(InputStream input) {...}
  // helper to read a single input-stream character
  private char read() {...}
  // hepler to look ahead one character in the input stream
  private char peek() {...}
  // helper to check if given character end-of-line symbol
  private boolean isEOL(char ch) {...}
  // helper to check if given character is an end-of-file
  private boolean isEOF(char ch) {...}
  // helper to print an error message and exit
  private void error(String msg, int line, int column) {...}

  // TODO: implement next token function (see starter code & hints)
  public Token nextToken() {...}

}
```

MyPL Lexer: Lexer Class

```
public Token nextToken() {
  // read initial character
  char ch = read();
  // read past whitespace
  while (Character.isWhitespace(ch)) {
    ...
  }
  // check for one-character symbols
  if (isEOF(ch))
    return new TokenType(TokenType.EOS, "end-of-stream", line, column);
  else if (ch == '.')
    return new TokenType(TokenType.DOT, ".", line, column);
  ...
}
```

Some hints: ... note: can implement as large method or break up

- start with whitespace (including newlines and EOF)
- then single-character tokens, then two-character tokens
- then comments, strings, numbers (integers and doubles),
- finally reserved words & ids (letter followed by letters, numbers, _'s)

MyPL Lexer: Lexer Class

Additional hints:

- use given helper functions: `read()`, `peek()`, `isEOL()`, `isEOF()`, `error()`
- look through unit tests, get them to pass
- check that works with example files (can use `diff` to check)
- `Character` class: `isDigit(ch)`, `isLetter(ch)`, `isLetterOrDigit(ch)`
- if you don't find a token, must be an error
- errors also in strings (end of line before closing "), numbers (leading zeros)