Lecture 5:

- Quiz 1
- Lexer (wrap up)
- Formal Grammars (intro)

Announcements:

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

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)

Parsing: Languages and Grammars

Parsers perform two jobs:

(1) ensure programs are syntactically correct

(2) generate abstract syntax trees (ASTs)

We'll start with (1), but first we have to define the MyPL syntax!

A PL's syntax is defined using a formal grammar:

- a set of *grammar rules* that defines a "language" as a *set of (finite) strings*

Note: In a formal grammar, a language is just a set of (allowable) strings

- e.g., the language of all 2-letter strings ({"aa", "ab", "ac", ...})
- all binary numbers w/ same # of 0's and 1's ({"10", "1010", ...})
- in our case, the strings are all the syntactically valid MyPL programs

More on Formal Grammars

There are many types (classes) of formal grammars ... expressive power

- The **regular** grammars specify regular languages (think regular expressions)
- The **context free** grammars specify context-free languages (most PLs)
- ... and so on

Can use grammar rules in "two directions": ... parsing is a mix of both

- Given a string, check if it follows the rules
- Apply the rules to generate (*derive*) strings

Aside: Grammars closely tied to computation ... models of computation

- Applying rules to derive strings is "computing" the strings of language
- Viewing strings as algorithm outputs, the grammar "computes" the outputs