Lecture 9:

- Wrap up $LL(k)$
- Recursive Descent Parsing

Announcements:

- HW-2 out

$LL(k)$ Grammars

Check In: Can you spot any of the "$LL(k)$" problems in our example?

```
<stmt_list> ::= <stmt> | <stmt> ';' <stmt_list>
<stmt> ::= <var> '=' <expr>
<var> ::= 'A' | 'B' | 'C' | ... | 'Z'
<expr> ::= <var> | <expr> '+' <expr> | <expr> '-' <expr>
```

Q: Is it left-recursive? Yes ... (`<expr>`)

Q: Can it be left factored? Yes

Q: Is it ambiguous? No

$LL(k)$ Grammars

Check In: What about now? ... after removing left recursion

```
<stmt_list> ::= <stmt> | <stmt> ';' <stmt_list>
<stmt> ::= <var> '=' <expr>
<var> ::= 'A' | 'B' | 'C' | ... | 'Z'
<expr> ::= <var> | <var> '+' <expr> | <var> '-' <expr>
```

Q: Is it left-recursive? No ... (<expr>)

Q: Can it be left factored? Yes

Q: Is it $LL(k)$? ... No (<stmt_list>)

$LL(k)$ Grammars

Check In: What if we change it to this? ... fixed-sized <expr>

```
<stmt_list> ::= <stmt> | <stmt> ';' <stmt_list>
<stmt> ::= <var> '=' <expr>
<var> ::= 'A' | 'B' | 'C' | ... | 'Z'
<expr> ::= <var> | <var> '+' <var> | <var> '-' <var>
```

Q: Is it left-recursive? No ... (<expr>)

Q: Can it be left factored? Yes

Q: How many look ahead tokens needed? 6 for <stmt_list> (A=B+C;...)

$LL(k)$ Grammars

After dealing with left-recursion and left-factoring:

<stmt_list> ::= <stmt> <stmt_list_tail>

<stmt_list_tail> ::= ';' <stmt_list> | ε

<stmt> ::= <var> '=' <expr>

<var> ::= 'A' | 'B' | 'C' | ... | 'Z'

<expr> ::= <var> <expr_tail>

<expr_tail> ::= '+' <expr> | '-' <expr> | ε

Q: Is it left-recursive? No ... (<expr>)

Q: Can it be left factored? No

Q: How many look ahead tokens needed? 1

Recursive Descent Parsing

Basic Idea:

- divide parse into separate methods ... roughly one per non terminal
- corresponding grammar rule(s) encoded in each non-terminal's function
- "descend" the parse tree using method calls (possibly recursion)

```
public class SimpleParser {
  private Lexer lexer;       // the lexer
  private Token currToken;   // the current token

  public SimpleParse(Lexer lexer) {...}

  // helper functions:
  private void error(String msg)           // generate error
  private void advance()                   // get next token, skip comments
  private boolean match(TokenType target)  // compare target to currToken
  private boolean matchAny(List<TokenType> targets) // compare to many
  private void eat(TokenType target, String msg) // if match advance, else err

  public void parse()       // run the parser

  ... recursive descent functions ...
}
```

Recursive Descent Parsing

`SimpleParser` Helper and Parse Functions (Basics)

```java
  private void advance() {
    currToken = lexer.nextToken();
    while (match(TokenType.COMMENT))
      currToken = lexer.nextToken();
  }

  private void eat(TokenType target, String msg) {
    if (!match(target))
      error(msg);
    advance();
  }

  public void parse() {
    advance();                       // get first token
    program();                       // call program rule
    eat(TokenType.EOS, "...");       // ensure no tokens left
  }

  private void program() {...}
  private void funDef() {...}
  private void structDef() {...}
  ... etc. ...
}
```


Recursive Descent Parsing

How it works ...

The running example: ... *Simplified*: fold in <stmt>, use ID, ASSIGN

<stmt_list> ::= ID ASSIGN <expr> <stmt_list_tail>

<stmt_list_tail> ::= SEMICOLON <stmt_list> | ε

<expr> ::= ID <expr_tail>

<expr_tail> ::= PLUS <expr> | MINUS <expr> | ε

```java
  public void parse() {
    advance();
    stmtList();
    eat(TokenType.EOS, "...");
  }

  private void stmtList() {
    eat(TokenType.ID, "...");
    eat(TokenType.ASSIGN, "...");
    expr();
    stmtListTail();
  }
```

```java
private void stmtListTail() {
  if (match(TokenType.SEMICOLON)) {
    advance();
    stmtList();
  }
}

private void expr() {
  eat(TokenType.ID, "...");
  exprTail();
}
```