Lecture 11:

- Quiz 3
- Recursive Descent

Announcements:

- HW-2 out

Recursive Descent Parsing

Basic Idea:

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

In HW-2 we use a `SimpleParser` class:

```
class SimpleParser:

    def __init__(self, lexer):
        self.lexer = lexer
        self.curr_token = None

    def parse(self):

    #--------------------------------------------------
    # Helper functions
    #--------------------------------------------------

    def error(self, message):
    def advance(self):
    def match(self, token_type):
    def match_any(self, token_types):
    def eat(self, token_type, message):
    def is_bin_op(self):

    #--------------------------------------------------
    # Recursive descent functions
    #--------------------------------------------------

    def struct_def(self):
    def fields(self):
    def fun_def(self):
    def params(self):

    ... etc ...
```

The Helper and Parse Functions

```python
def advance(self):
    self.curr_token = self.lexer.next_token()
    # skip comments
    while self.match(TokenType.COMMENT):
        self.curr_token = self.lexer.next_token()


def match(self, token_type):
    return self.curr_token.token_type == token_type


def match_any(self, token_types):
    for token_type in token_types:
        if self.match(token_type):
            return True
    return False


def eat(self, token_type, message):
    if not self.match(token_type):
        self.error(message)
    self.advance()


... etc ...

# note: combines start with <program> rule
def parse(self):
    self.advance()
    while not self.match(TokenType.EOS):
        if self.match(TokenType.STRUCT):
            self.struct_def()
        else:
            self.fun_def()
    self.eat(TokenType.EOS, 'expecting EOF')
```

Basic Idea of Recursive Descent Functions

The running example:

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

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

<expr> ::= VAR <expr_tail>

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

The parser:

```
def parse(self):
    self.advance()                              # sets current token
    self.stmt_list()                            # check stmt list rule
    self.eat(TokenType.EOS, 'expecting EOF')    # ensure out of tokens


def stmt_list(self):
    self.eat(TokenType.VAR, 'expecting var')    # check var, advance
    self.eat(TokenType.ASSIGN, 'expecting =')   # check =, advance
    self.expr()                                 # check expr rule
    self.stmt_list_tail()                       # check tail rule


def expr(self):
    self.eat(TokenType.VAR, 'expecting variable')
    self.expr_tail()


def expr_tail(self):
    # note can be simplified, but here is general idea
    if self.match(TokenType.PLUS):
        self.advance()
        self.eat(TokenType.VAR, 'expecting var')
    elif self.match(TokenType.MINUS):
        self.advance()
        self.eat(TokenType.VAR, 'expecting var')
```

Watch out: align calls to **advance()** (**eat()**) in each recursive descent function