Lecture 13:

- Abstract Syntax Trees

Announcements:

- HW-2 out
- *Note:* Exam next week (Wed), no class Mon
Generating Abstract Syntax Trees (ASTs)

1. The parsing step both checks syntax and builds the AST

2. An AST is typically used for:
   - semantic analysis, e.g., type checking, ensuring items defined before used
   - interpretation, e.g., in an AST interpreter
   - conversion to intermediate representation (like bytecode)

3. An AST is like an “expression tree” ...

- do “in-order traversal” (left, node, right) to “execute” expression tree
- more node types in an AST, e.g., declarations, loops, var assignment, etc.
Running Example: with \(<expr>\) resurrected

\[
\begin{align*}
<\text{stmt\_list}> & \ ::= \ <\text{stmt}> \ <\text{stmt\_list\_tail}> \\
<\text{stmt\_list\_tail}> & \ ::= \ \text{SEMICOLON} \ <\text{stmt\_list}> | \epsilon \\
<\text{stmt}> & \ ::= \ \text{VAR ASSIGN} \ <\text{expr}> \\
<\text{expr}> & \ ::= \ \text{VAR} \ <\text{expr\_tail}> \\
<\text{expr\_tail}> & \ ::= \ \text{PLUS} \ \text{VAR} | \ \text{MINUS} \ \text{VAR} | \epsilon
\end{align*}
\]

The AST might contain nodes (objects) representing:

- statement lists \((\text{StmtList})\)
- an assignment with a var and an expression \((\text{Stmt})\)
- expressions with single var and (optional) op and expression \((\text{Expr})\)

We’ll be using “Plain-Old Data” (POD) classes ... “Data Classes” in Python

```python
@dataclass
class Expr:
    lhs: Token
    op: Token
    rhs: Token

@dataclass
class Stmt:
    var: Token
    expr: Expr

@dataclass
class StmtList:
    stmts: List[Stmt]
```

CPSC 326 (Spring 2024) S. Bowers 3 of 4
```python
def parse(self):
    self.advance() # init lexer
    stmt_list_node = StmtList([]) # empty statement list
    self.stmt_list(stmt_list_node) # descend into stmt_list
    eat(TokenType.EOS, "...") # ensure EOS
    return stmt_list_node # return AST root node

def stmt_list(self, stmt_list_node):
    stmt_node = Stmt(None, None) # empty Stmt
    self.stmt(stmt_node) # descend into stmt
    stmt_list_node.smts.append(stmt_node) # add the stmt
    self.stmt_list_tail(stmt_list_node) # continue to tail

def stmt_list_tail(self, stmt_list_node):
    if self.match(TokenType.SEMICOLON):
        self.advance() # advance past it
        self.stmt_list(stmt_list_node) # descend into stmt list

def stmt(self, stmt_node):
    stmt_node.var = self.curr_token # store var token
    self.eat(TokenType.VAR, "...") # ensure VAR
    self.eat(TokenType.ASSIGN, "...") # ensure ASSIGN
    expr_node = Expr(None, None, None) # empty expr node
    self.expr(expr_node) # descend into expr
    stmt_node.expr = expr_node # connect expr node

Check In: Rewrite the remaining recursive descent functions to build the AST
```