Today

- Quiz 4
- Syntax Analysis: Abstract Syntax Trees

Assignments

- HW3 due
- HW4 out (due next Thurs)
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” ...

\[
\begin{align*}
&- \\
&\quad + \\
&\quad \times \\
&\quad \div 4 \\
&\quad 2 \\
\end{align*}
\]

- perform “in-order traversal” (left, node, right) to “execute” expression tree
- more types of nodes in an AST, e.g., declarations, loops, var assignment, etc.
Running Example

\[
\langle \text{stmt\_list} \rangle ::= \text{VAR } \text{ASSIGN } \langle \text{expr} \rangle \langle \text{stmt\_list\_tail} \rangle \\
\langle \text{stmt\_list\_tail} \rangle ::= \text{SEMICOLON } \langle \text{stmt\_list} \rangle | \epsilon \\
\langle \text{expr} \rangle ::= \text{VAR } \langle \text{expr\_tail} \rangle \\
\langle \text{expr\_tail} \rangle ::= \text{PLUS } \text{VAR} | \text{MINUS } \text{VAR} | \epsilon
\]

Parser class with basic methods and member variables:

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

    def parse(self):
        ...  

    # helper functions
    def __advance(self):
        ...  
    def __eat(self, tokentype, err_msg):
        ...  
    def __error(self, err_msg):
        ...  

    # recursive descent functions
    def __stmt_list(self):
        ...  
    def __stmt_list_tail(self):
        ...  
    def __expr(self):
        ...  
    def __expr_tail(self):
        ...
In our example, AST might contain nodes (objects) representing:

- statement lists (StmtList)
- an assignment with an identifier and an expression (AssignStmt)
- an expression with a single variable (VarExpr)
- an expression with two variables and an operator (OpExpr)

Note we’d also have Expr as a superclass of VarExpr and OpExpr

class StmtList(object):
    def __init__(self):
        self.smts = []  # list of AssignStmt

class AssignStmt(object):
    def __init__(self):
        self.lhs = None  # VAR token
        self.rhs = None  # Expr

class Expr(object): pass

class VarExpr(Expr):
    def __init__(self):
        self.var = None  # VAR token

class OpExpr(Expr):
    def __init__(self):
        self.left_operand  # VAR token
        self.operator       # PLUS or MINUS token
        self.right_operand  # VAR token
Adding AST creation to our Recursive Descent Parser

```python
def parse(self):
    stmt_list_node = ast.StmtList()  # create StmtList node
    self.__advance()  # init lexer
    self.__stmt_list(stmt_list_node)  # descend into stmt_list
    self.__eat(token.EOS, '...')  # ensure EOS
    return stmt_list_node  # return AST root node

    def __stmt_list(self, stmt_list_node):
        assign_node = ast.AssignStmt()  # create Assign node
        assign_node.lhs = self.curr_token  # set lhs
        self.__eat(token.VAR, '...')  # ensure VAR
        self.__eat(token.ASSIGN, '...')  # ensure ASSIGN
        assign_node.rhs = self.__expr()  # descend and set rhs
        stmt_list_node.smts.append(assign_node)  # add Assign node
        self.__stmt_list_tail(stmt_list_node)  # descend to tail
```

**Exercise:** Rewrite the remaining recursive descent functions to build the AST

**Exercise:** Draw the AST (object graph) resulting from the string "A=B+C; B=A"
The AST Class Hierarchy for MyPL (HW 4)