Today

- Abstract Syntax Trees

Announcements

- HW-3 due fri
- Exercises ...
- Quiz-3 thurs (grammars and parsing)
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

```plaintext
<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
```

Parser class with basic methods and member variables:

```java
public class Parser {
    private Lexer lexer;
    private Token currToken;

    public Parser(Lexer lexer) {
        // later returns AST
    }

    // helper functions
    private void advance() {
        // call next_token
    }
    private void eat(TokenType t, String msg) {
        // check and advance
    }
    private boolean match(TokenType t) {
        // check currToken against t
    }
    private void error(String msg) {
        // output error
    }
    // ...

    // recursive decent functions
    private void stmt_list();
    private void stmt_list_tail();
    // ...
}
```

We’ll be modifying the above functions to build up the AST ...

- including the signatures as needed
In our example, 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 \((\text{Expr})\)

Note that below we are using “Plain-Old Data” (POD) classes

```java
public class Expr {
    public Token lhs;
    public Token op;
    public Token rhs;
}

public class Stmt {
    public Token var;
    public Expr expr;
}

public class StmtList {
    public List<Stmt> stmts = new ArrayList<>();
}
```
Adding AST creation to our Recursive Descent Parser

```java
public StmtList parse() {
    advance(); // init lexer
    StmtList node = new StmtList();
    stmt_list(node); // descend into stmt_list
    eat(EOS, "..."); // ensure EOS
    return node;
}

private void stmt_list(StmtList node)
{
    Stmt s = new Stmt();
    s.var = currToken;
    eat(VAR, "..."); // ensure VAR
    eat(ASSIGN, "..."); // ensure ASSIGN
    Expr e = new Expr();
    expr(e); // descend into expr
    s.rhs = e;
    node.stmts.add(s);
    stmt_list_tail(node); // descend into stmt_list_tail
}

private void stmt_list_tail(StmtList node)
{
    if (match(SEMICOLON)) {
        advance();
        stmt_list(node);
    }
}
```

**Exercise:** Rewrite the remaining recursive descent functions to build the AST
private void expr(Expr node)
{
    node.lhs = currToken;
    eat(VAR, "...");
    expr_tail(node);
}

private void expr_tail(Expr node)
{
    if (match(PLUS) || match(MINUS)) {
        node.op = currToken;
        advance();
        node.rhs = currToken;
        eat(VAR, "...");
    }
}

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