Today

- Recursive Descent Parsing (cont)

Assignments

- HW3 out (due Thurs)
- Quiz on Thurs
- Exam 1 next Tues
Parsing: An example grammar

Simple list of assignment statements

\[
\begin{align*}
\langle \textit{stmt\_list}\rangle & \ ::= \ \text{VAR ASSIGN } \langle \textit{expr}\rangle \langle \textit{stmt\_list\_tail}\rangle \\
\langle \textit{stmt\_list\_tail}\rangle & \ ::= \ \text{SEMICOLON } \langle \textit{stmt\_list}\rangle \mid \epsilon \\
\langle \textit{expr}\rangle & \ ::= \ \text{VAR } \langle \textit{expr\_tail}\rangle \\
\langle \textit{expr\_tail}\rangle & \ ::= \ \text{PLUS } \text{VAR } \mid \text{MINUS } \text{VAR } \mid \epsilon
\end{align*}
\]

Where \(::=\) is used in place of \(\rightarrow\) (read as “becomes”)
Recursive Descent Parsing

A simple approach for ad hoc parsing

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

We’ll use a Parser class with basic methods and member variables:

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

    public Parser(Lexer lexer) {...}
    // start recursive descent
    public void parse() throws MyPLEException {...}

    // move forward in token stream
    private void advance() throws MyPLEException {...}
    // check currToken's type and advance
    private void eat(TokenType t, String errmsg) ...
    // generate a standard error exception
    private void error(err_msg) ...

    /* recursive descent functions */
    private void stmtList() ...
    private void stmtListTail() ...
    private void expr() ...
    private void exprTail() ...
}
```
The Helper Functions

// get and store the next token
private void advance() throws MyPLEException {
    currToken = lexer.nextToken();
}

// ensure current token in stream is of given type, and advance
private void eat(TokenType t, String errmsg) throws MyPLEException {
    if (currToken.type() == t)
        advance();
    else
        error(errmsg);
}

// handle errors
private void error(String err_msg) throws MyPLEException {
    String s =errmsg + " found '" + currToken.lexeme() + '"';
    int row = currToken.row();
    int col = currToken.column();
    throw new MyPLEException("Parser", errmsg, row, col);
}

Note that throwing an exception stops parser at first error

• as opposed to printing error, and then continuing
The Recursive Descent Functions

The example:

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

How it works:

```java
public parse() throws MyPLEException {
    advance(); // set current token
    stmtList(); // check stmt list rule
    eat(TokenType.EOS, "expecting end of file");
}
```

```java
private void stmtList() throws MyPLEException {
    eat(TokenType.VAR, "expecting variable"); // check if var, advance
    eat(TokenType.ASSIGN, "expecting '='"); // check if =, advance
    expr(); // check expr rule
    stmtListTail(); // check tail rule
}
```

```java
private void expr() throws MyPLEException {
    eat(TokenType.VAR, "expecting variable");
    exprTail();
}
```

Watch out for aligning calls to advance in each recursive descent function

- either assume the current token is already advanced
- or else the function starts by advancing the current token

**Exercise:** Finish the recursive descent functions for this language
How to handle rule decisions (which version of a rule to take)

- assuming grammar is in $LL(1)$, only need to look at the current token
- as an example, consider this (modified) MyPL (r-value) rule:

  \[
  \texttt{<rvalue> ::= NIL} \mid \texttt{NEW ID} \mid \texttt{NEG <expr>} \mid \texttt{<idrval>}
  \]

- an **r-value** is a value that can occur on the right-hand-side of an assignment

Current token is either NIL, NEW, NEG, or else the last case ...

```java
private void rvalue() throws Exception {
    if (currToken.type() == TokenType.NIL) {
        advance();
    }
    else if (currToken.type() == TokenType.NEW) {
        advance();
        eat(token.ID, "expecting identifier in rvalue");
    }
    else if (currToken.type() == TokenType.NEG) {
        advance();
        expr();
    }
    else {
        idrval();
    }
}
```

**Note:** don’t always need one recursive descent function per non-terminal

- this is more of a rule of thumb ...
- sometimes you can condense ... do what makes sense
- when we add the AST, we may also need to reorganize a bit
Some “trickier” cases ...

\[ <expr> ::= ( <rvalue> \mid \text{NOT} <expr> \mid \text{LPAREN} <expr> \text{RPAREN} ) \]
\[ \quad ( <\text{operator}> <expr> \mid \text{epsilon} ) \]

Note that this rule is really six separate rules:

- The general structure of the rule is:

  \[ e \rightarrow ( v_1 \mid v_2 \mid v_3 )( r_1 \mid r_2 ) \]

- Which corresponds to:

  \[ e \rightarrow v_1 r_1 \]
  \[ e \rightarrow v_1 r_2 \]
  \[ e \rightarrow v_2 r_1 \]
  \[ e \rightarrow v_2 r_2 \]
  \[ e \rightarrow v_3 r_1 \]
  \[ e \rightarrow v_3 r_2 \]

We can handle the entire thing in one recursive descent function:

```java
private void expr() throws MyPLEException {
    if (currToken.type() == TokenType.LPAREN) {
        advance();
        expr();
        eat(TokenType.RPAREN, "expecting ‘)’ in expression");
    }
    else if (currToken.type() == TokenType.NOT) {
        advance();
        expr();
    }
    else
        rvalue();
    if (isOperator(currToken.type())) { // helper function ...
        advance();
        expr();
    }
}
```
Some helper functions to check multiple cases can be useful ...

```java
private boolean isOperator(TokenType t) {
    Set<TokenType> s = Set.of(TokenType.PLUS, TokenType.MINUS, ...);
    return s.contains(t);
}
```

As another “tricky” case ... use (while) loops for Kleene star

- For example, consider this rule:

\[
\langle exprlist \rangle ::= \langle expr \rangle (\ COMMA \langle expr \rangle)^* \mid \epsilon
\]

Again, we can do this in a single function call ...

```java
private void exprlist() throws MyPLEException {
    if (isExpr(currToken.type())) { // another helper function ...
        expr();
        while (currToken.type() == TokenType.COMMA) {
            advance();
            expr();
        }
    }
}
```
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” …

\[\begin{array}{c}
\text{−} \\
\text{+} \\
\div \\
4 \\
2 \\
\end{array}\]

– 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

\[
\langle \text{stmt\_list} \rangle ::= \text{VAR ASSIGN} \langle \text{expr} \rangle \langle \text{stmt\_list\_tail} \rangle
\]

\[
\langle \text{stmt\_list\_tail} \rangle ::= \text{SEMICOLON} \langle \text{stmt\_list} \rangle | \epsilon
\]

\[
\langle \text{expr} \rangle ::= \text{VAR} \langle \text{expr\_tail} \rangle
\]

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

Parser class with basic methods and member variables:

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

    public Parser(Lexer lexer) {...}
    // start recursive descent
    public void parse() throws MyPLEException {...}

    private void advance() throws MyPLEException {...}
    private void eat(TokenType t, String errmsg) ...
    private void error(err_msg) ...

    private void stmtList() ...
    private void stmtListTail() ...
    private void expr() ...
    private void exprTail() ...
}
```

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

- include the signatures as needed
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

```java
public class StmtList {
    public ArrayList<AssignStmt> stmts = new ArrayList<>();
}

public class AssignStmt {
    public Token lhs = null; // lvalue (VAR token)
    public Expr rhs = null; // right-hand side expression
}

public class Expr { // or as an interface
}

public class VarExpr extends Expr {
    public Token var = null; // VAR token
}

public class OpExpr extends Expr {
    public Token leftOperand = null; // VAR token
    public Token operator = null; // PLUS or MINUS token
    public Token rightOperand = null; // VAR token
}
```