Lab 2: Parsers, ASTs, and Visitors

Objectives

- Implement a recursive-descent parser for a small grammar.
- Wire up visitor hooks by writing accept(...) in each AST node.
- Use the provided driver with --debug to generate parser traces (stderr) and the parse tree (stdout).
- Practice OO concepts: inheritance, smart pointers, double dispatch.

Grammar

```
<sentence>          → <noun phrase> <verb phrase> <noun phrase>
<noun phrase>       → <adjective phrase> NOUN
<adjective phrase>  → (ARTICLE | POSSESSIVE) ADJECTIVE
<verb phrase>       → VERB | ADVERB <verb phrase>
```

All input words must come from rules.l.

Unknown words become TOK_UNKNOWN and will (by design) cause some tests to fail.

Provided Files

Build

- makefile — use as-is.

Lexical Analysis

- rules.l — maps words to token codes.
- lexer.h — token constants (e.g., ARTICLE, NOUN, TOK_EOF, TOK_UNKNOWN).

AST

- ast.h — AST node class definitions (do not edit).
- ast.cpp — you implement the accept(...) one-liners.

Parser

- parser.h — function prototypes (do not edit).
- parser.cpp — you implement all recursive-descent functions and insert debug calls.

Tree Printer

- printer.h / printer.cpp — complete ASCII tree printer (read to understand; no edits).

Driver

- driver.cpp — already supports --debug and wires lexer → parser → printer. Use it as-is.

Debug Support

- debug.h — helpers you'll call from the parser; they print to stderr only.

Student Tasks

Step 1 — ast.cpp: Visitor Hooks and Double Dispatch

Why this step matters

- In C++, a Node* can point to many possible derived types (AdjectivePhrase, NounPhrase, etc.).
- If we just had one virtual print() function, each node would have to "know" how to format itself. That makes the AST classes depend on the printing logic.
- Instead, we separate concerns:
    - AST nodes know their structure and data.
    - The Printer class knows how to draw them.
- To connect the two, we use **double dispatch**:
    1. Call node->accept(printer) (dispatch on the node's dynamic type).
    2. That calls printer.visit(*this) (dispatch on the overload chosen by the exact node type).

This way, the Printer can implement different logic for each kind of node, while the AST nodes themselves stay lightweight.

What you need to do

Open ast.cpp and implement the accept(...) functions for all four node types.
Each is a one-liner that hands control back to the Printer:

For example, here's the NounPhrase version:

```
void NounPhrase::accept(Printer& p) { p.visit(*this); }
```

This says: "when someone visits me, I'll pass myself back to the visitor, so it can decide what to do with a NounPhrase."

Now do the same pattern for:

- AdjectivePhrase
- VerbPhrase
- Sentence

How to check your understanding

1. Open printer.h: you'll see four overloads of visit(...). Each is specific to one node type.
2. When you run the parser, it builds a tree of unique_ptr<Node>.
3. At the end, the driver does:
4. Printer pp(cout);
5. root->accept(pp);
    - root is a unique_ptr<Sentence>.
    - That calls Sentence::accept(pp).
    - Which calls pp.visit(*this).
    - Which resolves to Printer::visit(Sentence&).
    - Inside that, the printer then recurses on the children.

So every node gets dispatched twice: once on its own type (Sentence, NounPhrase, etc.), and again on the printer's overload. **That's double dispatch in action**.

Step 2 — parser.cpp (Recursive-Descent Parser)

Goal

Implement a parser for this grammar:

<sentence> → <noun phrase> <verb phrase> <noun phrase>
<noun phrase> → <adjective phrase> NOUN
<adjective phrase> → (ARTICLE | POSSESSIVE) ADJECTIVE
<verb phrase> → VERB | ADVERB <verb phrase>

- Inputs must use lexemes from rules.l.
- Unknown words tokenize as TOK_UNKNOWN and will cause parse errors (expected in some tests).
- Tree output must go to **stdout**.
- Debug trace must go to **stderr** (use --debug).

What's already provided

- parser.h declares:
  parseStart, parseSentence, parseNounPhrase, parseAdjectivePhrase, parseVerbPhrase
- lexer.h declares token codes (e.g., ARTICLE, NOUN, VERB, ADVERB, POSSESSIVE, TOK_EOF, TOK_UNKNOWN).
- debug.h provides: gDebug, dbgLine(...), DebugIndent, and tokenName(int).

Helpers included in parser.cpp

- **Lookahead** (an int) to store the current token.
- **next()** to advance lookahead via yylex() and (if debug on) log it.
- **expect(tok, msg)** to consume the expected token and return the matched lexeme (yytext), or throw runtime_error(msg).

Only next() and expect() should advance tokens.

Exact error messages (must match)

- <sentence> did not start with an article or possessive.
- <noun phrase> did not start with an article or possessive.
- <noun phrase> did not have a noun.
- <adjective phrase> did not start with an article or possessive.
- <adjective phrase> did not have an adjective.
- <verb phrase> did not start with a verb or an adverb.
- Extra input after complete sentence.

Debug tracing (required)

At the **top** of each parse function:

```
dbgLine("enter <noun phrase>");
DebugIndent _scope;   // auto-indents this function's trace block
```

On each successful token match:

```
dbgLine(string("matched NOUN: ") + yytext);
```

(These print to **stderr**; they will not affect the graded stdout.)

One worked example (study this pattern, then do the rest)

parseAdjectivePhrase()

Spec: <adjective phrase> → (ARTICLE | POSSESSIVE) ADJECTIVE

- FIRST check: lookahead ∈ {ARTICLE, POSSESSIVE} else
  throw "<adjective phrase> did not start with an article or possessive."
- Set detType (Article/Possessive) and store the determiner lexeme.
- Require ADJECTIVE else
  throw "<adjective phrase> did not have an adjective."
- Return the filled node.

Your implementation must also emit the debug “enter …” line and “matched …” lines as you consume tokens.

Like this:

```
// <adjective phrase> → (ARTICLE | POSSESSIVE) ADJECTIVE
// Errors:
//   "<adjective phrase> did not start with an article or possessive."
//   "<adjective phrase> did not have an adjective."
unique_ptr<AdjectivePhrase> parseAdjectivePhrase() {
  dbgLine("enter <adjective phrase>");
  DebugIndent _scope;

  // FIRST check
  if (lookahead != ARTICLE && lookahead != POSSESSIVE) {
    throw runtime_error(
      "<adjective phrase> did not start with an article or possessive."
    );
  }

  auto node = make_unique<AdjectivePhrase>();

  // Determiner (ARTICLE | POSSESSIVE)
  if (lookahead == ARTICLE) {
    node->detType   = AdjectivePhrase::DetType::Article;
    node->detLexeme = expect(ARTICLE,
      "<adjective phrase> did not start with an article or possessive.");
  } else { // POSSESSIVE
    node->detType   = AdjectivePhrase::DetType::Possessive;
    node->detLexeme = expect(POSSESSIVE,
      "<adjective phrase> did not start with an article or possessive.");
  }

  // ADJECTIVE
  node->adjLexeme = expect(ADJECTIVE,
    "<adjective phrase> did not have an adjective.");

  return node;
}
```

Now implement these:

parseNounPhrase()
Spec: <noun phrase> → <adjective phrase> NOUN

1. dbgLine("enter <noun phrase>"); DebugIndent _scope;
2. FIRST check: ARTICLE or POSSESSIVE, else
   throw "<noun phrase> did not start with an article or possessive."
   (Tip: do this FIRST, before constructing children.)
3. auto node = make_unique<NounPhrase>();
4. node->adj = parseAdjectivePhrase();
5. node->nounLexeme = expect(NOUN, "<noun phrase> did not have a noun.");
6. return node.

parseVerbPhrase()
Spec: <verb phrase> → VERB | ADVERB <verb phrase> (normalize to ADVERB* then VERB)

1. dbgLine("enter <verb phrase>"); DebugIndent _scope;
2. FIRST check: VERB or ADVERB, else
   throw "<verb phrase> did not start with a verb or an adverb."
3. auto node = make_unique<VerbPhrase>();
4. While lookahead == ADVERB:
   - node->adverbs.push_back(expect(ADVERB, "..."))
   - (Use a precise message only where required; FIRST errors are already covered above.)
5. node->verbLexeme = expect(VERB, "<verb phrase> did not start with a verb or an adverb.");
6. return node.

parseSentence()
Spec: <sentence> → <noun phrase> <verb phrase> <noun phrase>

1. dbgLine("enter <sentence>"); DebugIndent _scope;
2. FIRST check: ARTICLE or POSSESSIVE, else
   throw "<sentence> did not start with an article or possessive."
3. auto node = make_unique<Sentence>();
4. node->subjectNP = parseNounPhrase();
5. node->verbP = parseVerbPhrase();
6. node->objectNP = parseNounPhrase();
7. return node.

Quick self-check before you run

- Do all FIRST checks throw the **exact** strings above?
- Do you only advance input via next()/expect()?
- Does parseStart() enforce TOK_EOF and throw the exact “Extra input …” message if not?
- Do your debug lines use **stderr** (they will, via debug.h)?

Running & Expected I/O

You can run the parser either with a filename or from stdin.

Run with a file argument

```
./parse input1.in
```

Run with stdin (no filename)

```
./parse
```

Here you can type a sentence interactively, then press **Ctrl+D** to end input.

Debug tracing

Add `--debug` before or after the filename (or with stdin) to enable parser/lexer traces to **stderr**:

```
./parse --debug input1.in
```

OR

```
./parse --debug
```

Example workflow with provided tests

There are 6 sample inputs (`input1.in … input6.in`):

- **3 valid** (parse successfully, print a full tree).
- **3 invalid** (fail with one of the required error messages).

For example:

```
./parse input1.in
```

- prints the ASCII parse tree for that sentence on **stdout**.

```
./parse --debug input1.in
```

- prints the same tree (or error) on **stdout**,
- and a full trace of recursive descent steps on **stderr**.

Report

Answer briefly (1–3 sentences each):

1. Double Dispatch — What problem does double dispatch solve in our lab, and how does it work with `accept(...)` and `visit(...)`?
2. Abstract Class — Why is `Node` defined as an abstract base class with a pure virtual `accept(...)`?
3. ASTs — What is the advantage of building an Abstract Syntax Tree (AST) instead of just checking tokens directly?

4. Debugging Tests — Pick one of the three failing test inputs. Explain why it failed and what change(s) (to the grammar, lexer, or parser) could make it succeed. Be SPECIFIC!

Deliverables

After your live demo is successful, prepare the following for submission on Canvas:

1. **Source Code**
   - Zip your entire Lab 2 source directory (including all `.cpp`, `.h`, `rules.l`, and `makefile`).

2. **Report**
   - Include your completed short-answer report (see prompts above).
   - Save it as a PDF or Word file.

A successful demo should look like this: