Phase 2 Project Report

1. Introduction

In the second phase of our C-minus compiler project, we transitioned from a proof-of-concept lexer/parser to two fully-featured parsing pipelines:

1. A **manual, regex-driven LL(1)** parser built from scratch in Python.
2. An **ANTLR-generated lexer and parser** integrated into our existing infrastructure.

Additionally, we computed the LL(1) parsing table for our grammar using a professional parsing tool, ensuring robust error recovery and synchronization sets. This report outlines the architecture, core logic, and workflow that brought these components together into a cohesive compiler front end.

---

2. High-Level Architecture & Workflow

1. **Input Processing**
   - Source code is read via `BufferReader`, which streams characters in fixed-size chunks, tracks line numbers, and supports pushback for lookahead in the manual scanner.
2. **Lexical Analysis**
   - **Regex-Based Scanner**: A hand-built DFA recognizes identifiers, numbers, symbols, comments, and whitespace.
   - **ANTLR Lexer**: Generates tokens automatically from our `Cminus.g4` grammar, skipping whitespace/comments.
3. **Symbol Table & Token Table**

- Both parsers share a **symbol table** with scoped `Scope` objects tracking declarations, and a **token table** logging `(line_no, token)` pairs for later export.

4. **Parsing**

   - **Manual LL(1)**: Uses a predictive parsing table to drive a stack-based parser that builds an AnyTree parse tree and collects syntax errors with panic-mode recovery.

   - **ANTLR Parser**: Relies on ANTLR's runtime to produce a parse tree, then walks it with a custom listener/adaptor to populate our symbol and token tables, and to log syntax errors.

5. **Parse Table Computation**

   - We imported our grammar into Parcon (a professional LL(1) analysis tool), automatically computed FIRST, FOLLOW, and PREDICT sets, and exported the resulting parse table in CSV form.

---

3. Regex-Based LL(1) Parser

3.1 Scanner Design

- **Edge & DFANode** classes implement character classes and DFA transitions.

- **FinalStateNode** indicates acceptance, potentially pushing back the final character.

- We incrementally build the start state with five sub-DFAs for:

   1. **Numbers** (`[0-9]+`)

   2. **Identifiers/Keywords** (`[A-Za-z][A-Za-z0-9]*`)

   3. **Symbols** (`+`, `-`, `*`, `==`, `<`, `=`, punctuation)

4. **Comments** (`//…\n` and `/*…*/`)

5. **Whitespace & EOF**

Each accepting state invokes a generator (e.g. `num_token_gen`, `id_token_gen`) that:

1. Constructs a `Token(TokenType, lexeme)`.

2. Adds it to the token table.

3. For identifiers, invokes the symbol table to register declarations or lookups.

3.2 Grammar Representation & LL(1) Engine

- **Terminal** and **NonTerminal** objects represent grammar symbols.

- **Rule** objects store left/right sides plus a PREDICT set placeholder.

- **Grammar** imports:

    1. **First sets** (from `Firsts.txt`)

    2. **Follow sets** (from `Follows.txt`)

    3. **Productions** (`grammar.txt`)

    4. **Predict sets** (`Predicts.csv`)

- An **LL1** class:

    1. **Builds the parse table**: entries `(NonTerminal, Terminal) → production` or `synch`.

    2. **Parses** by maintaining a stack of expected symbols.

3. On mismatches, uses **panic-mode** recovery: skipping illegal tokens or popping on synch, logging errors with context.
4. Constructs a **parse tree** via AnyTree nodes labeled with either rule names or token tuples `(TYPE, lexeme)`.

---

4. ANTLR-Based Parser

4.1 Grammar & Lexer

- We authored an ANTLR grammar file `Cminus.g4` mirroring our language spec:
  - Rules for declarations, statements, expressions, arrays, functions, etc.
  - Lexer modes for keywords, identifiers, numbers, symbols, comments, whitespace.
- ANTLR generates `CminusLexer.py` and `CminusParser.py`, ensuring full compliance with standard lexing and LL(*) parsing.

4.2 Integration & Listeners

- **ANTLRTokenAdapter**: Converts ANTLR's tokens to our `Token(TokenType, lexeme)` format, mapping keywords to `TokenType.INT`, `TokenType.VOID`, etc.
- **ParseTreeBuilder**: Recursively walks ANTLR's parse tree, creating AnyTree nodes, adding identifier tokens to the symbol table, and populating the token table.
- **CminusErrorListener**: Captures syntax errors during ANTLR parsing, registering them in our `ErrorTable`.

---

5. Parsing Table Computation

5.1 Tool Selection & Workflow

To ensure our LL(1) parser's correctness, we used **Parcon 3.2** (a professional LL(1) analyzer):

1. Loaded our grammar in BNF format.
2. Ran automated FIRST/FOLLOW computation.
3. Generated PREDICT sets and the complete parsing table.
4. Exported the table to CSV, which our `Grammar.import_predict_sets()` ingests.

5.2 Ensuring Synchronization

We augmented the PREDICT table with **synchronizing tokens** drawn from each non-terminal's FOLLOW set, marking them as `"synch"` to enable error recovery when the parser falls out of sync.

---

6. Integration & Build Process

1. **Grammar Changes** → Re-run ANTLR tool & Parcon to regenerate parser and parse table.
2. **Code Compilation** → Python modules require only standard libraries plus `antlr4-python3-runtime` and `anytree`.

Execution →

 $ python phase2_manual.py <source>.cminus   # Regex + LL(1)
$ python phase2_antlr.py <source>.cminus   # ANTLR-based

3.

4. **Outputs** →
   - `parse_tree.txt`: Human-readable parse tree.
   - `tokens.txt`: Token stream.
   - `syntax_errors.txt`: Collected errors.
   - `symbol_table.txt`: Final symbol table with scoped identifiers.

---

7. Conclusion

By developing **dual parsing pipelines**, we now possess:

- A **lightweight, transparent LL(1)** implementation that demonstrates the inner workings of DFAs, parsing tables, and panic-mode recovery.
- A **robust ANTLR-generated** alternative that leverages industry-standard tooling.