Phase 2 Report — LL(1)-Compatible State-Machine Parser for the C-minus Compiler

1 Introduction

This report describes the implementation of the syntactic analysis (Phase 2) component of the C-minus compiler. Unlike a purely table-driven LL(1) parser, our implementation uses a **deterministic LL(1)-compatible state machine** generated from the grammar. The parser operates in a single pass, receiving tokens directly from the Phase 1 scanner via repeated calls to `get_next_token()`.

The parser outputs:

- a full parse tree in an ASCII-art box-drawing format, and
- a list of syntax errors encountered during parsing.

Additionally, because our compiler integrates Phase 2 and Phase 3, the parser also executes semantic actions embedded inside productions.

2 Design Philosophy

2.1 LL(1)-Compatible Parsing via States

Although the grammar is LL(1)-compatible, the parser is not implemented using:

- FIRST/FOLLOW prediction tables, nor
- a recursive-descent function per nonterminal.

Instead, the system uses:

- a **hand-crafted deterministic state machine**,
- defined in `parser_states.py`,
- where each state encodes transitions based on expected terminals or nonterminals.

Each C-minus grammar rule is represented as a sequence of states, through which the parser advances as it matches tokens or expands nonterminals.

2.2 Integration With the Scanner

The parser maintains a `current_token` retrieved from the scanner. The token stream is handled in a true single-pass manner:

- the parser consumes input only through `get_next_token()`,
- no backtracking or token caching is used,
- the scanner and parser form one continuous pipeline.

2.3 Parse Tree Output

Unlike the assignment specification, which required tab-indented nodes, our implementation prints the parse tree using Unicode box-drawing characters, e.g.:

```
Program
 Declaration-list
     Declaration
     Declaration
     Declaration
```

This tree is generated by `build_parse_tree_string()` in `parser.py`.

2.4 Semantic Integration

The parser is shared with Phase 3. Thus grammar productions in `productions.py` contain embedded semantic actions (marked with #. . .), which are executed via:

```
codegen.act(action_name)
```

This makes the parser both a syntactic and semantic driver.

3 Grammar and Production Handling

3.1 Grammar Representation

Grammar rules are defined in `productions.py`. Each production consists of:

- a left-hand-side nonterminal,
- a right-hand-side list of terminals, nonterminals, and semantic hooks,
- the name of the production.

The grammar matches the official C-minus specification, with minor formatting adjustments for state-machine encoding.

3.2 Use of FIRST/FOLLOW Sets

FIRST and FOLLOW sets were computed offline and used to verify LL(1) compatibility. However, the parser does **not** use these sets at runtime. Instead, the state machine incorporates the necessary predictability.

4 Parser Architecture

4.1 Core Components

The parser consists of:

- **ProductionParser**: the main parsing controller.
- State objects (from `parser_states.py`) encoding LL(1) transitions.
- **ParseNode**: tree nodes used to construct the parse tree hierarchy.
- semantic hook mechanism integrated with Phase 3's code generator.

4.2 Parsing Process

The parser maintains:

- `self.current_state`: the current state in the state machine.
- `current_token`: the next token from the scanner.

At each step, the parser:

1. checks available transitions from the current state,
2. distinguishes terminal vs. nonterminal transitions,
3. consumes a token if a terminal matches,
4. expands a nonterminal by jumping to the starting state of the corresponding production,
5. triggers embedded semantic actions when encountered.

4.3 Parse Tree Construction

A **ParseNode** object is created for each terminal or nonterminal. The parse tree is rendered using:

- “” for last children,
- “” for intermediate children,
- “” for vertical continuation lines.

This format is more expressive than simple indentation and provides a clear structural view of the parsed program.

5 Syntax Error Handling

5.1 Actual Recovery Strategy

The parser does **not** use FOLLOW-set panic-mode skipping. Instead, error handling is state-machine based:

- **Missing constructs** produce exceptions of type "missing", and the parser moves to the state expected after that element.
- **Illegal tokens** produce exceptions of type "illegal", and **only one token is skipped**.
- The parser then resumes in the next predicted state.

Thus, recovery is localized and guided by state transitions rather than scanning ahead to synchronization tokens.

5.2 Error Logging

Errors are accumulated in an internal list and reported together with semantic errors at the end of compilation. The parser does not write a standalone `syntax_errors.txt`; this is handled by the unified compiler in `compiler.py`.

6 Testing and Integration

Testing is performed via `test_all.py`, which runs:

- scanning,
- parsing,
- tree generation,
- semantic analysis,
- and code generation

as a single pipeline. The parser is not intended to function as a pure, isolated Phase 2 module.

7 Challenges and Solutions

7.1 Managing a Large Grammar

The grammar contains many productions with subtle differences. By encoding productions into states, the parser can deterministically follow LL(1)-compatible paths.

7.2 Coordinating Syntax and Semantics

Semantic hooks must be executed at exactly the correct time, requiring careful alignment between grammar symbols and parser state transitions.

7.3 Producing Readable ASCII Parse Trees

Designing a recursive box-drawing tree printer required meticulous control of prefixes, child ordering, and indentation.

7.4 State-Based Recovery

Error recovery must maintain parser progress without discarding excessive input, requiring tailored logic in the state machine.

8 Conclusion

This phase produced a fully functional LL(1)-compatible parser implemented as a deterministic state machine. The parser:

- integrates tightly with the Phase 1 scanner,
- executes semantic actions embedded in grammar productions,
- outputs expressive ASCII-art parse trees,
- performs localized state-based error recovery,
- and functions as the syntactic backbone of the single-pass C-minus compiler.

The result is a robust, extensible parser suitable for both syntactic and semantic analysis in subsequent phases.

Contributors

Mohammad-Saeed Arvanaghi, and Ilya Farhangfar, Herbod Pourali

References

[1] Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* 2nd Edition, Addison-Wesley.

[2] Terence Parr. *The ANTLR Parser Generator Documentation.*