MS2 Report - Group 57

Vision

We're building an Interactive Propositional Logic Prover that allows users to construct and explore proofs through a REPL interface centered on automated reasoning and proof state management. Users can add premises one at a time or load them in batches from a script, after which the system automatically applies inference rules—currently Modus Ponens rule—to expand the proof state by deriving new consequences. The prover maintains both the original and derived premises in its proof state, and users can query whether a given goal follows from the accumulated proof state. In MS1, we focus more on the general functionality of the prover and now we have shifted to a more concrete system design. We have specified our data structures that represent propositions, sequents, and proof states and designed and polished our interface and parser so that users could clearly understand the available commands and how to interact with the system.

Summary of progress

Between MS1 and MS2, our team moved from a purely conceptual project outline to a working foundation for our Propositional Logic Prover. During MS1, we focused on brainstorming the system's purpose, sketching out our intended workflow, and identifying the main modules we would need, but we had not yet written any code. In MS2, we began implementing the essential building blocks of the prover. We created an abstract syntax tree (AST) capable of representing basic propositional logic constructs, variables, conjunction, implication, and negation, which serves as the core internal representation of formulas throughout the system. Building on this foundation, we implemented a recursive-descent parser that interprets user input and converts it into this AST format. The parser currently supports formulas such as A, !A, A & B, and A -> B, and includes basic validation and error handling to reject malformed expressions.

Alongside the parser, we began laying the groundwork for the reasoning component of the prover. We implemented a simple proof-state structure that stores premises, derived formulas, and goals, as well as a preliminary version of the Modus Ponens inference rule, which can derive q from p and p -> q when both appear in the sequent. To make the parsing functionality testable and accessible, we also developed a minimal REPL interface that allows users to enter propositional formulas interactively and displays their parsed structure. Additionally, we wrote a small suite of unit tests to verify the behavior of the parser across valid and invalid inputs. Although the prover is not yet fully interactive or capable of complete derivations, the progress made in MS2 establishes the structural and functional groundwork needed to expand toward a complete interactive proof system in MS3.

Activity Breakdown

Person A (Parser) - Anna

Responsibilities:

- Design and implement the formula parser for propositional logic
- Create a command-line demonstration tool for parser testing

Activities & Features Delivered:

- Implemented parser for basic propositional formulas supporting:
    - Atomic propositions (A)
    - Conjunction (A & B)
    - Implication (A -> B)
    - Negation (!A)
- Built a mini REPL (Read-Eval-Print Loop) that accepts user input, parses formulas into internal representation, and displays formatted output
- Created an interactive loop that continues until user types "quit"
- Provided foundation for parsing premises and goals used by other components

Hours Spent: [2-3]

Person B (Data Structures + Rules) - Shirley

Responsibilities:

- Design core data structures for proof state representation
- Implement inference rules and derivation logic
- Create helper functions for proof state management

Activities & Features Delivered:

- Created three core modules: `lib/ast.ml`, `lib/rule.ml`, `lib/sequent.ml`
- Implemented proof state representation (`type t`) tracking premises, derived propositions, and goals
- Built Modus Ponens inference rule: if p and p → q are known, then q can be inferred
- Developed recursive derivation logic to repeatedly apply Modus Ponens until no new propositions can be derived
- Implemented helper functions:
    - `empty` - initializes empty proof state
    - `add_premise` - stores user assumptions
    - `add_derived` - stores inferred propositions (with duplicate checking)
    - `add_goal` - sets target proposition
    - `apply_modus_ponens` - applies inference rule to all available propositions
    - `judge_goal` - verifies if goal has been proven
    - `print_result` - displays proof state to user

Hours Spent: 2-3

Person C (CLI + State Management) - Tyson

Responsibilities:

- Design and implement the interactive user interface
- Manage user experience and command handling
- Create proof state visualization

Activities & Features Delivered:

- Built interactive REPL loop with enhanced user prompts and output
- Redesigned interface elements including welcome banner, command prompt, multi-line input support, and graceful quit handling
- Replaced basic `print_result` with better state visualization featuring numbered premises and highlighted derived facts/goal status
- Expanded command UX with:
  - Shortcuts for common operations
  - Undo/reset functionality with confirmation
  - Command history and help system
  - User-friendly error messages for typos
- Added support for batch input and script files for loading multiple premises/goals simultaneously
- Implemented derivation traces showing inference steps (e.g., "from A and A → B via Modus Ponens")

Hours Spent: 2.5

Person D (Integration + Polishing) - Nicole

Responsibilities:

- Refine and relocate parser for system-wide integration
- Integrate all components into cohesive system
- Implement testing and improve error handling

Activities & Features Delivered:

- Parser Refinement:
  - Moved parser from `src/parser.ml` to `lib/parser.ml` for shared access across all components

    - Added `strip_outer_parens` function and general "find top-level operator" helper
    - Enhanced parsing to correctly handle complex formulas like `(A & B) -> C` and `!(A -> B)`
    - Improved error messages for unmatched parentheses and invalid variables
- System Integration:
    - Organized modules into shared library (`lib/dune`) for code reuse across REPL and tests
    - Integrated commands (`premise`, `goal`, `derive`, `show`, `reset`, `help`, `quit`) with formula parsing, `Sequent.t` updates, and `apply_modus_ponens` calls
    - Connected all components to work seamlessly together
- Testing:
    - Added minimal regression tests (`test/parser_tests.ml`) covering parser basics and Modus Ponens functionality

Hours Spent: 2.5

Productivity analysis (Tyson)

Overall, our sprint went very well, and we ran into no issues regarding scoping or productivity. We think we did a very good job delegating tasks so that we could all work asynchronously without dealing with any merge conflicts. We also kept constant communication on every change so that the integration went smoothly. All members used Slack and GitHub consistently to make sure both code and communication were up to date. For example, at one point we had to reconsider the project structure and organization of our files to ensure that our program was modular and designed in an intuitive way. We were able to accomplish what we had discussed in our sprints and the first meeting with our PM, as we have been able to show the basics of our interface with one working rule. If we had to do something differently, however, we agreed that we would spend more time determining core functionality and expectations of outputs so that we don't have to make certain assumptions that could prevent consistency across parts. We also thought it would be a good idea to enforce functions specifications so it's easier for others to understand.