LogicLang: Developing a Logic Programming Language in Lean 4

COMP3740 Project, 2023

Matthew Richards
u7499989

Acknowledgements

I would like to thank Dr Ranald Clouston for his help and guidance over the course of the project.

Abstract

There are a variety of logic courses at ANU, however, there is a limited amount of software to help students with self-guided learning on this topic. While some exist, they are not as student-friendly as they could be, and require domain knowledge on top of the students' existing course loads. This report details the development of a logic problem solving application with a focus on an empathetic approach to its end-users (i.e., providing specific and helpful error messages to help diagnose any problems) to minimise the extra cognitive load of using the software on top of the problem itself with which the student is trying to get help. The project does this successfully, and features syntax very similar to existing courses (regular logic syntax for logical expressions, and Haskell-like syntax for defining functions and types) in order to further minimise problems that the student may have picking up the software.

The system is developed in Lean 4, a programming language and theorem prover, which is used to model many different components of the project with its powerful type system, and also the ability to formally prove properties of functions. While building the software will be the more concrete goal by which this will be shown, the overarching purpose of the project is to demonstrate an understanding of advanced functional programming techniques in Lean 4.

1 Introduction

Many courses at ANU teach logic and writing related deductive proofs. To aid students in these subjects, there exists today software that attempt to deliver an application in which students can enter their problems and be presented with a solution; however, many of these can be difficult to self-study with, as their syntax can be unintuitive, and their issues hard to debug.

The aim of this project will be to develop a system into which students can comfortably describe logical problems and be presented with a solution; that is, error messages should be clear and readable to help students who are less familiar with the system debug their issues, and expressions should closely resemble the mathematical syntax with which students are already familiar. With this, the code should use Lean's language features to stay easily extendible later, as it is likely that due to time constraints not every logical operation will be added. Thus, the project architecture should be flexible enough such that these can be added later without having to rework any existing, working code.

In what follows, the broad architecture of the application will firstly be discussed, followed by an individual analysis of the parser, solver, and editor. Finally, examples of using the software will be shown, to demonstrate how its features come together, and compare the experience against the projects' aims as stated above.

2 Application Architecture

The application can be interacted with by firstly writing a file consisting of the domain-specific logic language. Then, after running the built binary with the path to this file as its argument, the system parses its contents into a syntax tree, using this to set up the solver's environment and compute a solution. Once the solver has finished, the application prints either a formatted table containing a solution to the problem, or a list of errors with hints on how they can be fixed.

To achieve this, the application consists of three key components: the editor, the parser, and the solver. The editor aims to provide a user-friendly and intuitive interface with which the program can be interacted; the parser aims to create a conversational method of converting plaintext code into an expression described by Lean's type system, and to generate human-readable and descriptive errors. The solver aims to solve the provided Boolean formulae under the given constraints in a reasonable amount of time.

These three will interact as depicted in the Component Diagram in Figure 1:

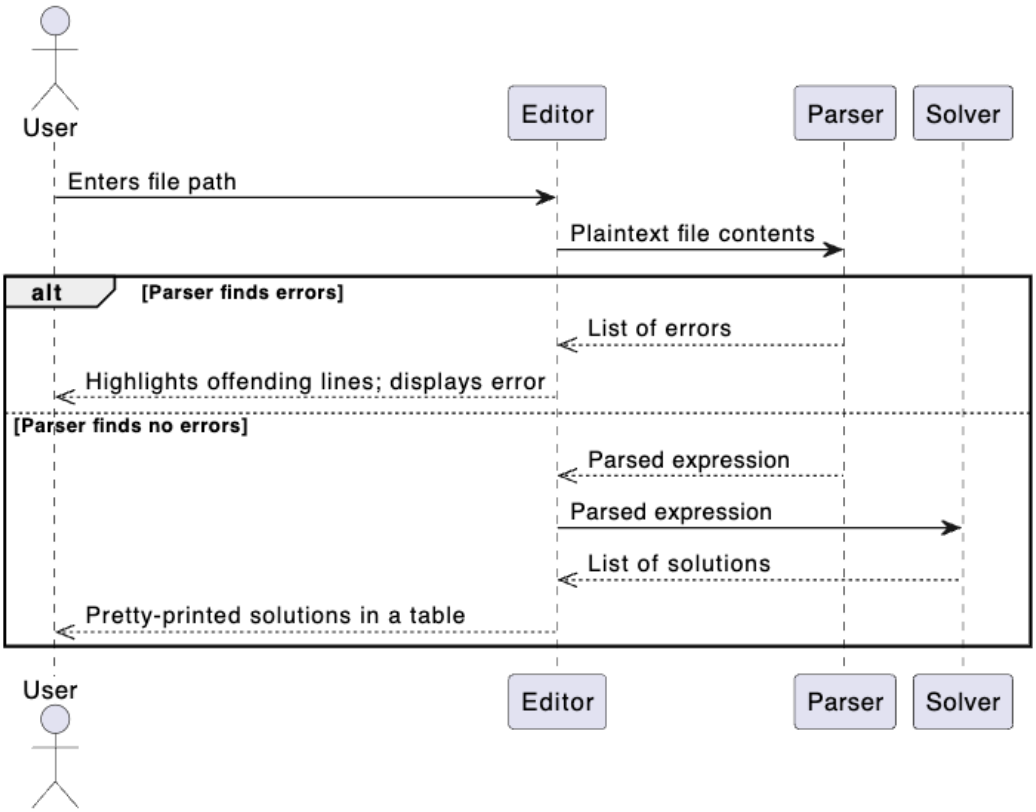

Figure 1: A Description of the Interactions between Components of the System

As is shown, the editor is the only component that links the parser and the solver; other than that, they do not know the other exists. This fits into the functional style

of Lean, as the entire parser can be thought of as one pure function that maps plain text to either a response, or errors; and the solver can also be thought of as one pure function that maps an expression to a list of solutions. Abstracting away implementation details like this means that a developer maintaining the system does not have a high cognitive load to worry about, as any specific details past the type signature of the component is relatively unimportant.

3 The Parser

3.1 Expression Parsing

The parser should follow the process:

1. Read some plain-text code
2. Scan the text to convert it into a list of tokens
3. Consume tokens while trying to match them against a "parsing rule", *i.e., check if they match the structure of an enum definition, or a function definition, etc...*
4. Finally, if all tokens were consumed and matched against parsing rules, return the list of parsed expressions.

Lean's type system facilitates this kind of system with its inductive types. Firstly, the types of tokens in the language can be represented as a simple inductive type, as show in Figure 2:

```
4   inductive TokenType where
5       -- single-character symbols
6       | LeftParen
7       | RightParen
8       | Pipe
9       | Semicolon
10      | Equals
11      | Not
12      | And
13      | Or
14      | ForAll
15      | Exists
16      | FullStop
17
18      -- two-character symbols
19      | RightArrow
20      | NotEquals
21      | DoubleColon
22
23      -- literals
24      | Identifier
```

Figure 2: An Excerpt of the Artefact Code Showing Tokens as a Simple Inductive Type

The plain-text code is then transformed into a list of these expressions through the *scanTokens* function, a recursive function which is formally proven to terminate, a process which already highlighted and eliminated a bug that was originally written into the code that would cause an infinite loop in an edge case.

However, the type system is much more powerful than this. Inductive types can be used to represent the entire formal grammar of the language; Figure 3 shows this idea using an excerpt from the artefact code.

```
20  inductive Value where
21      | literal : String → Value
22      | functionCall : String → Value → Value
23  deriving Repr, BEq
24
25  inductive LogicalPredicate where
26      | connect : LogicalPredicate → LogicalConnective → LogicalPredicate → LogicalPredicate
27      | predicate : Value → ComparisonOperation → Value → LogicalPredicate
28      | invertedPredicate : LogicalPredicate → LogicalPredicate
29  deriving Repr, BEq
30
31  inductive Expression where
32      | expressions : List Expression → Expression
33      | functionDefinition : (Option FnModifier) → String → String → String → Expression
34      | enumDefinition : String → List String → Expression
35      | assertion : LogicalPredicate → Expression
36  deriving Repr
```

Figure 3: Example of Grammar Represented as an Inductive Type in Lean 4

Compared to an object-oriented language, like Java, where these types would have had to be represented each as a class, it is apparent that the more appropriate abstraction comes in Lean 4, showing the value of the language's type system for this use case.

As parsers can be much more complicated than required in this project, to simplify the development and maintainability of the system, the plan for the architecture of the parser was to be easily open for extension—as more parsing rules will inevitably be added in the future, so they should be easy to add in—but closed for modification—that is, when one rule is complete it should not have to be touched ever again when adding new functionality. To accomplish this, the chain of responsibility pattern was employed.

The chain of responsibility pattern, while primarily a technique of object-oriented programming, is appropriate here as it describes how pure functions are sequenced together such that exactly one's return value is propagated, rather than composing functions like a monad. Hence each parsing rule was written as a function that returns an optional result, and its input would be passed to the next function if nothing was returned, otherwise the return value would be propagated out and returned early if it had one. This was implemented as the *ChainableHandler* type class, one which contains a delegate function, which passes the function input onto the next function and returns its result instead; and a yield function, which signifies an early return. Originally, the Monad type class was used instead, with a specific implementation of the bind function; however, the result of this was a "leaky" abstraction, as *pure*, and hence the *return* syntactic sugar in do-notation, would cause the input to be delegated to the next function in the sequence, and thus the abstraction was unintuitive and difficult to use. Furthermore, once the *ChainableHandler* type class was implemented, a new infix operator '~ >' was created to show chaining two functions together in a more concise

way. An example of this is shown in the excerpt from the main parser function that chains each parsing rule together in Figure 4 below:

```
def parseTokens (tokens : List Token) : Except String Expression :=

    let handledResult : TokenParserContext Expression :=
        checkFunctionDefinition tokens -- (⤳ is the chain of responsibility operator)
        ⤳ λnext ⇒ checkInjectiveFunctionDefinition next
        ⤳ λnext ⇒ checkEnumDefinition next
        ⤳ λnext ⇒ checkAssertionExpression next
```

Figure 4: An Example of the Chain of Responsibility Delegation Operator Used to Chain Different Functions Together

This implements the third step at the top of this section, as it effectively allows the program to try to map a list of tokens against every possible parsing rule until one generates an expression. An alternative may have been to have one very large pattern matching expression; however, this was deemed to have been harder to maintain and read, as too much logic for different rules would have been sitting adjacent to each other, and it would thus violate the goals for the parser architecture outlined at the beginning of the chapter, as existing code would need to be modified to add new rules (e.g., changing existing pattern matching conditions to split up the patterns and change to which rules they delegate).

3.2 Error Handling

What has not yet been addressed, however, is the way in which errors will be propagated to the end-user, as one of the main aims of the project was to report human-readable and useful errors to users who might not understand the syntax yet.

This is implemented within the *ParserContext* type, which is the return type of all parsing functions, being a structure that contains an optional result, the original input, and a list of errors.

```
structure ParserContext (ε i α : Type) where
  input : i
  result : Option α := none
  errors : List ε := []
deriving Repr
```

Figure 5: The ParserContext Type

Hence, the error handling effect on this type is encoded through the previously mentioned *delegate* function of the *ChainableHandler* type class. Specifically, when two functions are sequenced by *delegate*, their errors will be combined regardless of whether one produced a result value, effectively acting as a monadic bind.

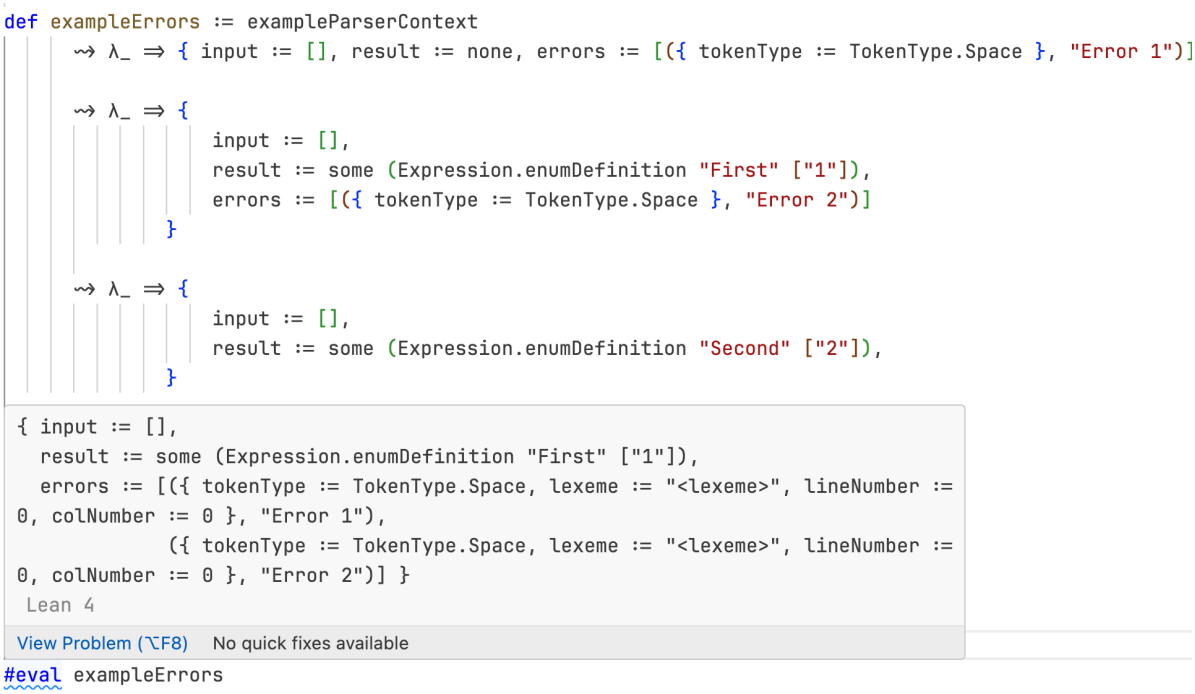

Figure 6: An Example of the Delegate Function Combining the Errors from Consecutive Computations

Ultimately, this allows different parsing rules to raise multiple errors over different lines. This allows an IDE-like experience of showing every error in the file at once, rather than printing the first error and exiting.

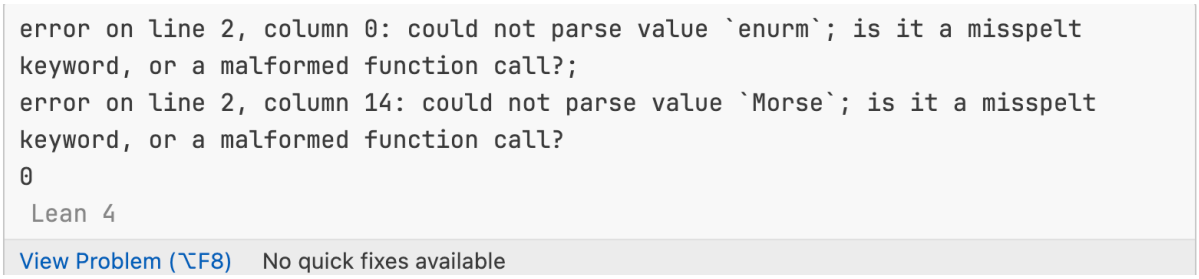

Figure 7: An Example of How Errors are Propagated by the Parser to Help the End-User with Syntax Issues

4 The Solver

To find a solution to the parsed expression the application uses backtracking, by considering the expression as a description of a *constraint satisfaction problem (CSP)*. Backtracking solves these problems by considering a set of constraints, and a set of possible variable assignments. One variable assignment is applied, then the algorithm checks to see if any of the constraints were violated; if not, another variable assignment is applied; if any are violated, then the operation is undone, and a different assignment is tried (Ghaderi, 2010). This is implemented recursively and forms a tree.

For an example, consider a *getPet* function that maps a Person to a Pet, and the constraints that Matt owns a Cat, and Dean owns a different pet to Matt:

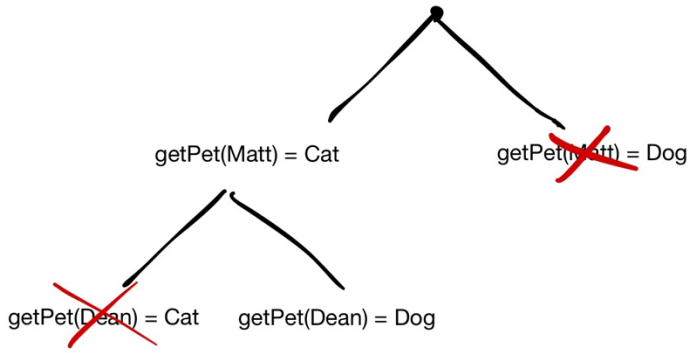

Figure 8: Example of a Backtracking Tree for a Simple Constraint Satisfaction Problem

As is shown in Figure 8, for every level down the tree that the algorithm develops, the new variable assignment is tested against every existing constraint (where the red crosses in Figure 8 show where a node violated some constraint), and thus the final leaf nodes of the tree are all valid solutions.

The implementation uses the State monad in Lean to hold an optional solution, so that a recursive call can check the current state to find out whether any solution has been found yet in a different branch (in which case nothing more is searched), or to otherwise set the solution if one is found. This allows the search to stop early, and thus the function can run much faster than if it were forced to search the entire solution space to find every possible solution and return a list instead.

An interesting implementation detail, however, is that Lean does not contain a State monad directly. Instead, the *StateT* monad transformer was used with its effects composed onto the identity monad *Id.* This is also how the standard library defines it, as shown in Figure 9, as an excerpt from *State.lean*.

```
25  @[reducible]
26  def StateM (σ α : Type υ) : Type υ := StateT σ Id α
27
```

Figure 9: The Definition of the StateM monad in the Lean 4 Standard Library

5 The Editor

The editor, as discussed in the introduction, is the link between the parser and the solver. The editor is the only part of the application that makes use of the IO monad, allowing the rest of the codebase to remain pure and thus easily testable (especially by #eval).

In this section, the imperative style was fully embraced, as seems appropriate when working with IO, and thus mutable variables and imperative-style while loops were used in many functions, including one shown in the below example that converts an IO file stream to a string:

```
def convertStreamToString (stream : IO.FS.Stream) : IO String := do
    let mut accumulator := ""
    let mut buffer ← stream.read bufsize

    while !buffer.isEmpty do
        let bufferAsString := String.fromUTF8Unchecked buffer
        accumulator := s!"{accumulator}{bufferAsString}"
        buffer ← stream.read bufsize

    return accumulator
```

Figure 10: An Example of a Function that Uses Lean's Imperative Monad Features

This consistent style keeps the code more readable across this section.

This style remains over the entire editor segment, which can also be seen in a helper file, *SolutionTableConstructor.lean*, which builds the ASCII table to pretty-print the solutions for the terminal.

6 Results and Conclusion

With the IO pieces in place, we can build the project and run it from the terminal. The artefact files include a set of example files with which the program can be interacted and tested to facilitate this.

Returning to the start of the report, the goals of the project can be summarised as follows:

- Clean code allows for new parsing rules to easily be added without refactoring existing code
- Solves logical expressions written in regular mathematical syntax
- Error messages help users understand syntax errors in simple and clear language

The first of which, while subjective, seems to have been met, especially in how the chain of responsibility pattern was implemented. This can be quickly shown by adding a new simple grammar rule that matches expressions of the type:

$$\forall < identifier >. < expression >$$

We firstly introduce the new grammar rule in our inductive type for expressions:

```
31  inductive Expression where
32      | expressions : List Expression → Expression
33      | functionDefinition : (Option FnModifier) → String → String → String → Expression
34      | enumDefinition : String → List String → Expression
35      | assertion : LogicalPredicate → Expression
36      | forAll : String → Expression → Expression
37  deriving Repr
```

Figure 11: The Grammar Rule for Expressions with a new For-All Rule

Then we can write a new parsing function that attempts to match tokens into our new structure, or otherwise delegates the input:

```
def checkUniversalQuantifierDefinition (tokens : List Token) : TokenParserContext Expression := do
    match tokens with

    | { tokenType := TokenType.ForAll, .. }
        :: { tokenType := TokenType.Identifier, lexeme := id, .. }
        :: { tokenType := TokenType.FullStop, .. }
        :: xs ⇒ return Expression.forAll id (← checkAssertionExpression xs)

    | _ ⇒ delegate tokens
```

Figure 12: The New Parsing Rule for For-All Statements

Then, we can simply include it in the parser pipeline:

```
def parseTokens (tokens : List Token) : Except String Expression :=

    let handledResult : TokenParserContext Expression :=
        checkFunctionDefinition tokens -- (⤳ is the chain of responsibility operator)
        ⤳ λnext ⇒ checkInjectiveFunctionDefinition next
        ⤳ λnext ⇒ checkEnumDefinition next
        ⤳ λnext ⇒ checkAssertionExpression next
        ⤳ λnext ⇒ checkUniversalQuantifierDefinition next
```

Figure 13: The parseTokens *Function with the New Parsing Rule in the Pipeline*

This will be sufficient. We can test our new rule, and it will simply work:

```
def forAllQuestion := parseMultiLineString
    "
    ∀x. getHorse(x) = Jorse;
    "

Except.ok (Expression.expressions
  [Expression.forAll
     "x"
     (Expression.assertion
       (LogicalPredicate.predicate
         (Value.functionCall "getHorse" (Value.literal "x"))
         (ComparisonOperation.equals)
         (Value.literal "Jorse"))))])
 Lean 4
View Problem (⌥F8)   No quick fixes available
#eval forAllQuestion
```

Figure 14: The Depiction of the Successful Parsing of the Introduced Rule

Due to time constraints, it is clear that not everything will be added as of this first version, and thus this design allows for further extension in the future without it taking too much work or refactoring.

For the second point, the requirement was that the parser is required to understand mathematical syntax, to keep the application close to students' existing knowledge. This is supposed to be a tool that facilitates teaching, and thus requiring students to learn domain knowledge to use the application goes against its philosophy. While this is a more holistically judged aspect, it seems that by looking at the examples provided in the artefact's *Examples* folder this is true; however, the parser is still flexible enough to change the current syntax if a more natural syntax is suggested later.

Finally, the third goal was to provide error messages that help students to understand how to fix their code.

Below we see *valid-example1.logic*:

```
1   enum Person = Matt | Liz | Joe | Kate;
2   enum Horse = Morse | Lorse | Jorse | Korse;
3
4   fn getHorse :: Person → Horse;
5
6   getHorse(Matt) = Morse ∨ getHorse(Matt) = Lorse;
7   getHorse(Liz) = Morse ∨ getHorse(Liz) = Lorse;
8   Korse = getHorse(Kate);
9   getHorse(Matt) ≠ Lorse;
10  getHorse(Joe) = getHorse(Liz);
```

Figure 15: The Contents of valid_example1.logic

Then once the project is built using *lake build*, we can run *./build/bin/logiclang ./Examples/valid_example1.logic* in the terminal to be returned the following:

```
(base) matthewrichards@Matts-MacBook-Air logic_lang % ./build/bin/LogicLang ./Examples/valid_example1.logic
 | | | |
 +----------------------+
 | getHorse
 +----------------------+

  getHorse(Kate) = Korse
  getHorse(Liz) = Morse
  getHorse(Matt) = Morse
  getHorse(Joe) = Morse

 +----------------------+
```

Figure 16: The Terminal Output After Running the Program on the Example Code

It should be clear through creating the ASCII table what the solution is.

Alternatively, we could make a small change to the example file, and swap the double colon for an equals. This is an understandable mistake, as this syntax (while inspired by Haskell to try and keep the syntax familiar to students) is fairly arbitrary.

```
1  enum Person = Matt | Liz | Joe | Kate;
2  enum Horse = Morse | Lorse | Jorse | Korse;
3
4- fn getHorse :: Person → Horse;
5
6  getHorse(Matt) = Morse ∨ getHorse(Matt) = Lorse;
7  getHorse(Liz) = Morse ∨ getHorse(Liz) = Lorse;
8  Korse = getHorse(Kate);
9  getHorse(Matt) ≠ Lorse;
10 getHorse(Joe) = getHorse(Liz);

1  enum Person = Matt | Liz | Joe | Kate;
2  enum Horse = Morse | Lorse | Jorse | Korse;
3
4+ fn getHorse = Person → Horse;
5
6  getHorse(Matt) = Morse ∨ getHorse(Matt) = Lorse;
7  getHorse(Liz) = Morse ∨ getHorse(Liz) = Lorse;
8  Korse = getHorse(Kate);
9  getHorse(Matt) ≠ Lorse;
10 getHorse(Joe) = getHorse(Liz);
```

Figure 17: The Diff Tool from Visual Studio Code Showing the Introduced Error

Now, re-running the program surfaces the following message:

```
(base) matthewrichards@Matts-MacBook-Air logic_lang % ./build/bin/logiclang ./Examples/valid_example1.logic
error on line 4, column 12: unexpected token `=`; expected to see `::` here
```

Figure 18: The Error Message from Swapping Double Colon for Equals

As shown, the program can tell exactly which token is incorrect, and tell the student the line and column number at which they need to fix their mistake. Furthermore, the correct symbol is suggested.

Another example is when the user forgets a semicolon to end an expression:

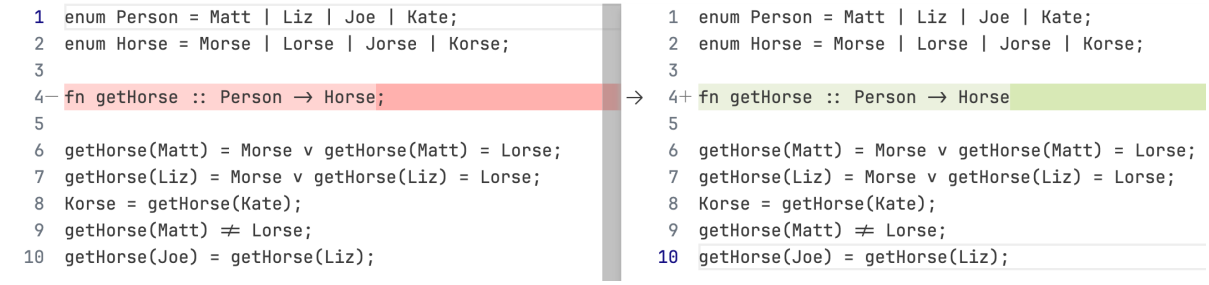

Figure 19: The Visual Studio Code Diff Tool Showing the Introduced Error

The error in the console aims to be similarly as helpful:

```
(base) matthewrichards@Matts-MacBook-Air logic_lang % ./build/bin/logiclang ./Examples/valid_example1.logic
error on line 4; expected semicolon at the end of the line but did not find one.
```

Figure 20: The Error Message from Missing a Semicolon

One more example comes in the form of a non-existent keyword:

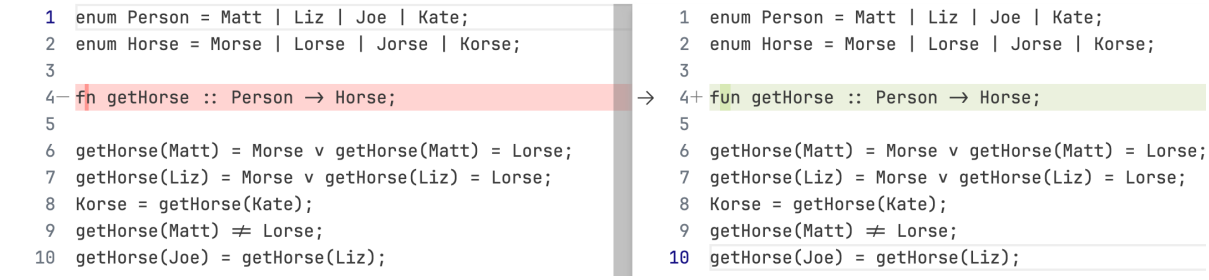

Figure 21: Visual Studio Code Diff Tool Showing the Keyword Difference

```
(base) matthewrichards@Matts-MacBook-Air logic_lang % ./build/bin/logiclang ./Examples/valid_example1.logic
error on line 4, column 0: could not parse value `fun`; is it a misspelt keyword, or a malformed function call?
```

Figure 22: The Error Message when a Non-Existent Keyword is Used

These errors aim to be both descriptive, and give useful hints, which seem to meet the goal of being able to help students unfamiliar with the software to use it anyhow without requiring as much guidance from tutors/teaching staff based on domain knowledge, nor unique syntax.

Ultimately, this project aimed to create a system which could help students with their studies in logic-based subjects. By creating a solver that does not require much domain knowledge to use, and when it does, provides error messages that are both helpful (by providing hints), and specific (i.e., providing line and column numbers), it is demonstrated that this has been achieved.

While time constraints dictated that not every feature would be fully implemented (e.g., quantifiers), the open-for-extension architecture allows that these can be added at a later date without the risk of increasing complexity.

Bibliography

Christiansen, D 2023, *Functional Programming in Lean*, viewed 27 October 2023, <https://lean-lang.org/functional_programming_in_lean/>

Ghaderi, H 2010, *Backtracking Search (CSPs)*, viewed 27 October 2023, <https://www.cs.toronto.edu/~hojjat/384w09/Lectures/Lecture-04-Backtracking-Search.pdf>

Nystrom, R 2021, *Crafting Interpreter*s, viewed 27 October 2023, <https://craftinginterpreters.com/scanning.html>

Refactoring Guru 2023, *Chain of responsibility*, viewed 27 October 2023, <https://refactoring.guru/design-patterns/chain-of-responsibility>